ICOT Technical Report: TR-296

TR-296

QPC : QJ-based Proof Compiler Simple Examples and Analysis

by
Y. Takayama

September, 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

QPC: QJ-based Proof Compiler
--- Simple Examples and Analysis ---

(Draft)

Yukihide Takayama

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108 Japan
takayama@icot.jp

ABSTRACT

The QPC system was implemented in the experimental study of the application of the notion of realizability to program synthesis. This paper presents the program extraction algorithm of QPC. Two examples, a proof of the course of value induction with mathematical induction and a program that calculates the greatest common divisors of natural numbers, are investigated. The computational meaning of the course of value induction schema is explained with some performance analysis. Optimization techniques using proof normalization and the modified $\vee$-code are also presented.

1. Introduction

As has been discussed by many researchers in terms of the relationship between intuitionism and computation, constructive proofs can be seen as a very high level description of algorithm. Following this notion, intelligent programming systems such as the Nuprl system [Constable 86] based on Martin-Loef's theory of types [Martin-Loef 82] and the PX system [Hayashi 87] based on Feferman's theory of functions and classes [Feferman 82] have been implemented. The notion of proof compilation is important in these kinds of systems to make constructive proofs executable on computers. In [Takayama 87], the basic design of our programming system with proof checker and proof compiler system based on QJ, Sato's theory of typed logical calculus QJ [Sato 85], is presented. The proof compiler in our system is called QPC, QJ based proof compiler.

This paper presents the proof compilation technique used in QPC. QJ is a constructive logic based on logic of partial terms. Types and logic are identified in the formulation of constructive type theory, but they are strictly separated in QJ, and a realizability interpretation of QJ, a kind of Curry-Howard's notion of formulae-as-types [Howard 80], is given to link types and logic. A slightly modified subset of QJ and its realizability interpretation is used here to write formal proofs and to extract programs from the proofs.

To generate efficient code from proofs, there are several problems to be considered. As is well known, the course of value induction can be proved by mathematical induction and vice versa, so that as long as one is interested in the expressive power of formal logical systems to write theorems and proofs, there is no need, at least from the theoretical point of view, to add the course of value induction schema to the formal system with mathematical induction schema. The course of value induction schema corresponds, in terms of the formulae-as-type isomorphism, to the divide and conquer strategy such as that used in the quick sort program and the program that calculates greatest common divisors (gcd) of natural numbers efficiently.

However, if the course of value induction schema is simulated by mathematical induction and introduced in the programming system as an induced rule, the extracted codes are not efficient.

The second problem is how optimization is performed on the codes extracted through realizability interpretation. The notion of proof normalization such as Gentzen's cut elimination is likely to be effective for this purpose.

Section 2 gives the algorithm of proof compilation. Sections 3 and 4 show the extraction of a gcd program from a proof tree. In Section 5, the problem of the course of value induction schema and efficiency is discussed through the examples given in Sections 3 and 4. Section 6 works on the optimization technique based on normalization of proof trees, and an easy and powerful optimization technique, the modified \/-code, is introduced. Section 7 overviews the experimental implementation of the proof compiler system, and Section 8 give the conclusion.

2. Proof Compiler

This section gives the formalization and program extract algorithm of the QPC system. The original version of the QPC has only mathematical induction as the induction schema. It does not have any optimization facilities.

2.1 Notational Preliminaries

1) nat
   Type expression of natural number: 0, 1, 2, ... Two canonical number theoretic functionS, 'succ' and 'pred', are associated with this type structure.
   QJ has other type structures such as disjoint sum, Cartesian product, function and recursive type structure with which nat and list type are defined. However, these type structures are not used explicitly in this paper.

The term expressions used in this paper are listed here. '==' is used to denote definitional equalities throughout the following description.

2) 0, 1, 2, ...
   Elements of type 'nat'.

3) 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'.

4) lambda [X]. A(X)
   Lambda abstraction. A(X) means that formula A has X as a free variable.

5) if A then B else C

6) left/right
   Flags to be used as the realizer code of \/-rule. Note that intuitionistic interpretation of A\/B is the proof of A or the proof of B. Following this interpretation, realizability of this formula is the conjunction of the information that shows which of the formulae in A and B actually holds and is proof.

7) void

Contradiction. Note that negation of a formula, A, is defined as ~A == A -> void.

8) mu [ZZ]. A(ZZ)
'mu' is the fixed point operator.

9) a(b1)(b2)...(bn)
Function application. Associate to left.

10) X mod Y
Residue of fraction of X by Y. This can be defined in QJ as follows, but is treated as a built-in function in this paper.

```
mod == mu [Z].
         lambda [X]. lambda [Y].
           if Y = 0 then void
           else if X < Y then X
                else Z(X-Y)(Y)
```

11) TERM1, TERM2
This has the same meaning as the conjunction of literals in PROLOG. If TERM1 and TERM2 are of types Type1 and Type2, then the above terms are seen to be of Cartesian product type: Type1 X Type2.

12) TERM1 = TERM2
Equality of terms.

13) Logical constants:
$\wedge$, $\vee$, ->, ~, all, exist

14) Proof tree:
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, as 'PI(X)'.

2.2 Inference Rules

The inference rules used are listed here. They are a subset of the inference rules of QJ. The natural number induction rule, 'nat-ind', is introduced as an instance of the recursive type induction schema of QJ.

Rules on formulae:

$$\frac{A \quad B}{A \wedge B}(\wedge I) \qquad \frac{A \wedge B}{A}(\wedge E) \qquad \frac{A \wedge B}{B}(\wedge E)$$

$$\frac{A}{A \vee B}(\vee I) \qquad \frac{B}{A \vee B}(\vee I) \qquad \frac{A \vee B \qquad \begin{matrix}[A]\\ C\end{matrix} \qquad \begin{matrix}[B]\\ C\end{matrix}}{C}(\vee E)$$

$$\frac{\begin{matrix}[A]\\ B\end{matrix}}{A \to B}(\to I) \qquad \frac{A \qquad A \to B}{B}(\to E)$$

$$\frac{\begin{matrix}[X:Type]\\ A(X)\end{matrix}}{\text{all } X:Type.\ A(X)}(\text{all-I}) \qquad \frac{t:Type \qquad \text{all } X:Type.\ A(X)}{A(t)}(\text{all-E})$$

```
t:Type        A(t)                 exist X:Type.A(X)      [x, A(x)]
------------------(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)
```

```
A   ~A                          void
-------(void-I)                --------(void-E)
 void                             A
```

```
t = s (in Type)  A(t)              t:Type                t = s (in Type)
----------------------(=)        -----------------(=)    -----------------(=)
    A(s)                          t = t  (in Type)        s = t (in Type)
```

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

Rules of term reduction:

```
A    if A then B else C              ~A    if A then B else C
-----------------------(if-=1)      -----------------------(if-=2)
if A then B else C = B               if A then B else C = B
```

```
        mu [Z]. A(Z)
-------------------------------(mu-=)
mu [Z]. A(Z) = A(mu [Z]. A(Z))
```

QJ also has rules of the number theory, formation of terms, and type inferences. However, these rules are not necessary as far as program extraction is concerned. For this reason, they are not listed here, and in the following description, the names of these rules are abbreviated to '*'.

2.3 Program Extraction Algorithm

The program extraction algorithm of the QPC system is given here. This algorithm performs q-realizability interpretation of QJ [Sato 85].

1) Notations for algorithm description

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

The top level procedure (function) of program extraction. It is often abbreviated as Ext(B). When a conclusion, B, depends on a list of formulae, Gamma, this procedure is denoted by Ext(A|-B).

Rv(A)

The realizing variable sequence. Realizing variables are the special variables to which realizer codes for the formula are assigned. Realizer codes are the programs that realize the computational meaning of given formulae. They can be seen as the programs that satisfy given specifications.

@

Substitution. If A is an expression, the application of @ to A is denoted by A@.

pI

p0 and p1 are projection functions on Cartesian product type TYPE-1 X TYPE-2. This type is extended as TYPE-1 X TYPE-2 X ... X TYPE-N, and pI is the extended projection function.

2) Definition of Ext procedure

```
    A    B
Ext(------(/\I)) == Ext(A), Ext(B)
     A/\B

     A/\B
Ext(------(/\E)) == p0(A/\B)
      A

     A/\B
Ext(------(/\E)) == p1(A/\B)
      B

       A
Ext(-------(\/I)) == left, Ext(A)
     A\/B

       B
Ext(-------(\/I)) == right, Ext(B)
     A\/B

               [A]      [B]
     A\/B       C        C
Ext(----------------------(\/E)) == if left = p0(Ext(A\/B))
               C                    then Ext(A|-C)@
                                    else Ext(B|-C)@

          where @ is the substitution:
               @ == [ Rv(A) <- p1(Ext(A\/B))),
                      Rv(B) <- p2((Ext(A\/B))) ]

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

    where @ = [Rv(X:nat, A(X)) <- ZZ(pred(X))


     A   ¯A
Ext(-------(void-I)) == (nil)
     void

     void
Ext(--------(void-E)) == (nil)
       A

     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) Note on (nil)

The atom 'nil' in Franz Lisp is used in the PX system [Hayashi 87] instead of '(nil)' in our formulation. The chief difference between 'nil' and '(nil)' is that 'nil' is an object while '(nil)' denotes empty codes. '(nil)' in treated as follows in the actual implementation of the QPC system.

If codes containing '(nil)' are extracted during proof compilation, the '(nil)' codes are eliminated through translation as follows:

```
'(nil)', '(nil)' -->> '(nil)'
 ..., Code_k, '(nil)', Code_k+1, .... -->> ..., Code_k, Code_(k+1)
(lambda [X]. '(nil)') -->> '(nil)'
(lambda [X]. Code)('(nil)') -->> (lambda [X]. Code)
'(nil)'(Code) -->> '(nil)'
if A then B then '(nil)' -->> B
if A then '(nil)' then C -->> C
```

3. Proof of Course of Value Induction in QJ

As is well known, the course of value induction schema

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

can be proved by mathematical induction. The following is the its proof in QJ.

```
          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:

---(*)
 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:

                          [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                              [Y:nat]
     ------------------------------------------------------(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)
```

Here is the course of value induction schema simulated by mathematical induction.

```
                          Proof of course of value induction
Course of value proof     by mathematical induction given above
-------------------------------------------------------------------(->E)
                  all Z:nat. P(Z)
```

Course of value proof:

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

4. Simple Example: GCD Program

The GCD program is taken as a simple example which uses the course of value induction schema.

4.1 GCD Proof

The specification of GCD is defined as follows:

```
all N:nat. all M:nat. exist D:nat. (D|N/\D|M)
```

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 later. The proof is called as 'Proof of GCD program' or just 'GCD proof' in the following description.

Let P(X) == all M:nat. exist D:nat. D|X/\D|M and
N|M == exist L:nat. M = L*N (where N:nat, M:nat), and perform

the course of value proof, then the GCD proof can be contained.

The course of value proof of the GCD Proof is as follows:

```
TREE_1              TREE_2                    TREE_3
-------------------------------------------------------(\/-E)
                                                P(N)
                    ------------------------------------(->I)
                    all L:nat. (L < N -> P(L)) -> P(N)
          ---------------------------------------------(all-I)
          all N:nat. (all L:nat. (L < N -> P(L)) -> P(N))
```

TREE_1: Proof of N=0 \/ N>0 (the same proof as TREE-2-2 in 3.)

TREE-2: Proof of N=0 |- P(N) (== all M:nat. exist D:nat. D|N/\D|M)

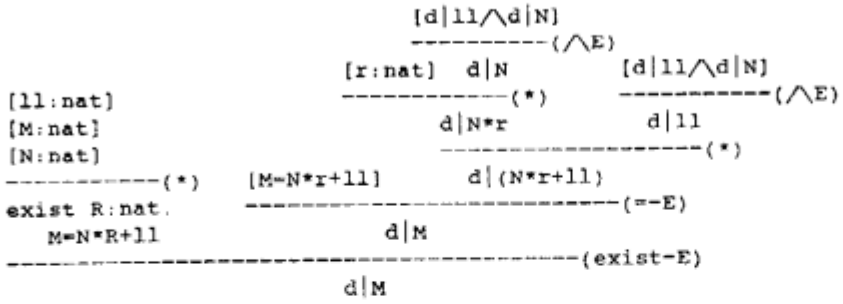
```
                      ------(*)                          -----(*)
                       0:nat  [P:nat]                     1:nat  [Q:nat]
             ----(*)   --------------(*)        ----(*)   --------------(*)
             0:nat       0=0*P                  1:nat     Q=1*Q
             --------------------(exist-I)      --------------------(exist-I)
             exist D':nat. 0=D'*P               exist D":nat.Q=D"*Q
             -------------------(all-I)         -------------------(all-I)
       [M:nat]  all P:nat. P|0           [M:nat]  all Q:nat.Q|Q
       ------------------------(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]     all M:nat. exist D:nat. D|0/\D|M
-----------------------------------------(=E)
all M:nat. exist D:nat. D|N/\D|M
```

TREE-3: Proof of N>0 |- P(N)

```
           [N>0] [M:nat]
           ------------(*)
           (M mod N):nat       HYP
           ------------------------(all-E)
                 (M mod N)<N ->
[N>0] [M:nat]  all M:nat. exist D:nat.
 ----------(*)       D|(M mod N)      [d|ll/\d|N]
  (M mod N)<N         /\D|M            ----------(/\E)
  ------------------------(->E)            d|N          TREE-3-sub
     all M:nat. exist D:nat.              ------------------------(/\-I)
[N:nat]       D|ll/\D|M            [d:nat]   d|N/\d|M
---------------------(all-E)     -------------------------(exist-I)
exist D:nat. D|ll/\D|M                exist D:nat. D|N/\D|M
-------------------------------------------------------------(exist-E)
           exist D:nat. D|N/\D|M
---------------------------------(all-I)
all M:nat. exist D:nat. D|N/\D|M
```

where ll == (M mod N)
HYP == all L:nat. (L<N -> all M:nat. exist D:nat. D|L/\D|M)

TREE-3-sub: Proof of d, d|ll∧d|N, M:nat, N:nat, ll:nat |- d|M



The GCD Proof written in PDL-QJ is given in Appendix 1. PDL-QJ is the proof description language of our theorem prover based programming system, CAP-QJ [Takayama 87].

4.2 Proof Compilation of GCD Proof

(1) The extracted code from the proof of the course of induction by mathematical induction is as follows:

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

        where
        AA ==  mu [Z0]. lambda [X]. lambda [Y].
                   if X=0 then right
                   else if left = (lambda [P].
                                     if P=0 then left else right)(Y) then left
                        else if left = Z0(pred(X))(pred(Y)) then left
                             else right
```

The code 'left = AA(X)(Y)' is the conditional equation which is logically equivalent to 'Y<X'. Note that the right side of this equation is extracted from TREE-2-1 in Section 3 that proves X:nat, Y:nat, Y<X+1 |- Y<X ⋁ Y=X.

The variable sequence, W0, W1, W2, 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). The computational meaning, i.e., realizer code, is to be assigned to W0, W1, W2.

The reason why the length of the variable sequence is 3 is that P(X) is now defined as all M:nat. exist D:nat. D|X∧D|M. Hence, by intuitionistic interpretation of logical constants and quantifiers, the computational meaning of this formula is a pair of the following:

1. A natural number, d, that satisfies d|X and d|M
2. Computational meaning of d|X and d|M

For 2., as X|Y can be defined as exist P:nat. Y = P*X, the computational meaning is the pair of the natural numbers, p and q, that satisfies X = p*d and M = q*d. Consequently, a triplet d, p and q, is the realizer code of all Y:nat. Y<X -> P(Y) |- P(X). (More precisely,

it is

```
        lambda [M] d(M), lambda [M]. p(M), lambda [M] q(M)
```

because d, p and q depends on M.)

The variable sequence, ZZ0, ZZ1, ZZ2, denotes the realizing variables of all Y:nat. (Y<X -> P(Y)), the hypothesis of course of value induction. The reason why the length of this sequence is 3 is similar to that given for W0, W1, W2.

(2) As the proof strategy used in Section 3 does not depend on the definition of P(X), a skeleton of the above realizer code can be used as the program schema. The skeleton is as follows. Only the variable sequence W1, W2, W3 is changed to a variable for sequence WW:

```
Skeleton == lambda [WW]. lambda [Z]. WW(Z)(A(Z))
A == mu [ZZ]. lambda [X]. lambda[Y].
                  if left = AA(X)(Y) then ZZ(pred(X))(Y) else WW(X)(ZZ(pred(X)))
AA ==  mu [Z0]. lambda [X]. lambda [Y].
             if X=0 then right
             else if left = (lambda [P].
                                 if P=0 then left else right)(Y) then left
                  else if left = Z0(pred(X))(pred(Y)) then left else right
```

where WW is the realizing variable sequence of the course of value proofs, and ZZ is the realizing variable sequence of the hypothesis of the course of value induction. Note that the length of realizing variable sequences can be mechanically calculated by analyzing the structure of formulae.

This schema can be used as follows:
1. The course of value induction schema is introduced in the programming system as an induced rule. 2. The course of value proof is written: all Y. (Y<X -> P(Y)) |- P(X) 3. The OPC system extracts the realizer code from the proof in 2. 4. The skeleton is instantiated with WW instantiated to the code obtained in 3. 5. The code obtained in 4 is the realizer code of all X. (all Y. (Y<X -> P(Y)) -> P(X)) -> all Z. P(Z).

(3) The extracted code from the course of value proof part is as follows.

```
g == lambda [N]. lambda [ZZ0, ZZ1, ZZ2].
          if left = (lambda [P]. if P=0 then left else right)(N)
          then lambda [M]. M, (lambda [Q].0)(M), (lambda [R].1)(M)
          else lambda [M]. (ZZ0, ZZ1, ZZ2)(M mod N)(N)
```

As the course of value proof and the proof of course of value induction by mathematical induction are used in (->E) inference to infer the conclusion, the extracted GCD program is an application of f to g:

```
        gcd == f(g)
```

5. Performance Evaluation of the GCD Program

5.1 Execution of the GCD Program

The code extracted in Section 4 is the 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 the verification information that can be seen as the decoded proof to show that the first element of the pair is actually the gcd.

In practical situations, verification information is not necessary, so that the code is used in the following form by projection.

```
f == lambda [W]. lambda [Z]. W(Z)(A(Z))
        where
        A = mu [ZZ].
            lambda [X]. lambda[Y].
               if left = AA(X)(Y) then ZZ(pred(X))(Y)
               else W(X)(ZZ(pred(X)))
        AA ==  mu [Z0]. lambda [X]. lambda [Y].
                    if X=0 then right
                    else if left = (lambda [P].
                                    if P=0 then left else right)(Y) then left
                         else if left = Z0(pred(X))(pred(Y)) then left
                              else right


g == lambda [N]. lambda [ZZ].
        if left = (lambda [P]. if P=0 then left else right)(N)
        then lambda [M]. M,
        else lambda [M]. ZZ(M mod N)(N)

gcd == f(g)
```

The operational semantics of the codes generated from the QPC is defined in the reduction model. The following reductions are used here:

```
A --> B:  beta-reduction of a lambda expression
A ==> B:  the if-then-else reduction as formulated in (if-=1)
          and (if-=2) rules.
A ~~> B:  beta-reduction of lambda expression with the fixed point operator.
           ~~> reduction is counted as a combination of two reductions:
                 (mu [Z]. lambda [X]. A(X, Z))(t)
           is reduced to
                 (lambda [X]. A(X, mu [Z]. lambda [X]. A(X, Z)))(t)
           by (mu-=) rule. Then, ordinal beta reduction is performed.
                 --> A(t, mu [Z].  lambda [X]. A(X, Z)).
```

The performance of programs are evaluated by regarding that the reductions --> and ==> require one unit of time, and the reduction ~~> requires two unit of times. The time required to execute number theoretic functions is regarded as much more smaller than that of reduction, so that it is not counted.

The reduction strategies used here are listed below. They are basically call-by-value evaluation and the sequential version of operational semantics given in QJ. The subset of expressions of QJ which is executable is called Quty, and the operational semantics of parallel execution is given in [Sato 87].

```
(1) if A then B else C
        1) Evaluate A.
        2) If A succeed, then evaluate B.
        3) If A fails, then evaluate C.
        4) Otherwise, suspend.
(2) (mu [Z]. A)(B)
        1) If B is a variable, then suspend.
        2) Otherwise, perform ~~> reduction.
(3) (lambda [X]. A)(B)
        1) Evaluate A to get A'.
        2) Evaluate B to get B'.
```

```
        3) If (lambda [X]. A) is obtained by "~> reduction and
           B' is a variable, then suspend.
                Example:
                   Let F == mu [Z]. lambda [X]. lambda [Y]. f(X, Y, Z).
                   F(2) is reduced to lambda [Y]. f(2, Y, Z), but
                   the evaluation of (lambda [Y]. f(2, Y, Z))(W) suspends
                   if W is a variable.
        4) Otherwise, evaluate A'{B'/X}.
(3) A(B)
        1) Evaluate A to get lambda [X]. A'.
        2) Evaluate (lambda [X]. A')(B).
(4) A = B
        1) Evaluate A to get A'.
        2) Evaluate B to get B'.
        3) If A' is a variable, then suspend.
        4) If B' is a variable, then suspend.
        5) Otherwise, if A' is identical to B' then succeed
           else fail.
(5) Evaluation of terms proceeds from left to right.

For the execution of gcd(2)(3), the following is observed:

1) Code A' simulates the divide and conquer strategy:
      The key idea of the proof of the gcd program given in 4.1 is to
      reduce the problem in the following way:
                problem to solve gcd(2)(3)
                ->>
                problem to solve gcd(3 mod 2)(2) = gcd(1)(2)
                ->>
                problem to solve gcd(2 mod 1)(1) = gcd(0)(1)
      Code A' defines the operation to reduce a problem. It takes
      several reduction steps to perform one problem reduction ->>.


2) The execution of code AA' takes a large part of the whole execution:
      As explained in 4. 2, AA' is the code to check Y<X when Y<X+1,
      X:nat, and Y:nat that can be performed much more efficiently in most
      programming languages by calling a few machine instructions of the
      computer. However, the algorithm that is realized in AA' is as follows:
        a) If X is 0 then fail, otherwise go to b).
        b) If Y is 0 then succeed, otherwise go to c).
        c) Subtract X and Y by 1, and go to a).
      This is essentially an inefficient algorithm.

3) Consequently, the number of reductions needed to perform the calculation
   of gcd is quite large (for gcd(2)(3), more than 60 reductions).
```

5.2 Comparison with Divide and Conquer Type Program

```
A gcd program may be written using the divide and conquer strategy as follows:

f == mu [Z]. lambda [N].
                if N = 0 then lambda M. M
                else lambda [M]. Z(M mod N)(N)

A similar program can be obtained from the code extracted in 4. 2:

g == lambda [N]. lambda [ZZ].
        if left = (lambda [P]. if P=0 then left else right)(N)
        then lambda [M]. M,
        else lambda [M]. ZZ(M mod N)(N)

by taking a fixed point, h, with regard to ZZ.
```

```
h ==  mu [ZZ]. lambda [N].
        if left = (lambda [P]. if P=0 then left else right)(N)
        then lambda [M]. M,
        else lambda [M]. ZZ(M mod N)(N)
```

This program reduces a problem directly without performing any operation. In other words, problem reduction ->> is performed without any help from A' and AA', so that the execution of h(2)(3) needs only 13 reductions.

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 in slightly more systematically.

6.1 Proof Normalization and Partial Evaluation of Programs

S. Goto uses proof normalization to analyze the behavior of logic programs [Goto 85]. Another application of proof normalization is optimization by partial evaluation because normalization of proofs corresponds to partial evaluation of extracted codes from the proofs through realizability interpretation. The following is the normalization rule listed in [Prawitz 65].

(1) all-normalization

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

(2) exist-normalization

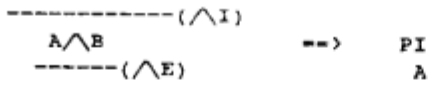
```
        PI
       P(t)                [P(a)]                      PI
    -------(exist-I)  PI'(a)                  ==>   [P(t)]
    exist X. P(X)          C                         PI'(t)
    ------------------------(exist-E)                 C
              C
```

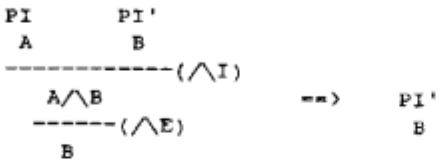
(3) -> normalization (Cut elimination)

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

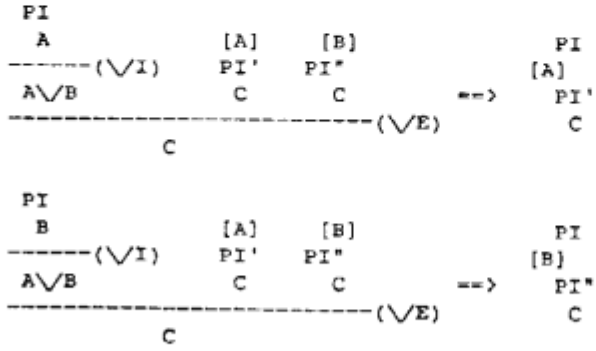
(4) ∧ normalization

```
    PI      PI'
     A       B
```



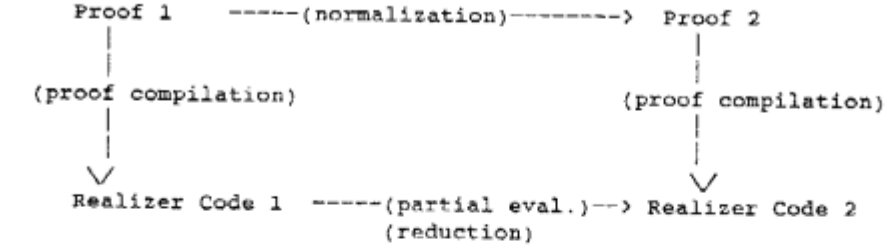


(5) ∨ normalization



Rules (2) and (4) are not effective for the optimization in proof compilation because, as can be seen by the definition of the Ext procedure for the (exist-E) and (/\E) rules given in 2. 3, the realizer codes of proofs before and after the normalization are equal. Rules (1) and (3) correspond to beta-reduction of lambda expressions. Rule (5) corresponds to if-then-else reduction of realizer codes.

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



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 extracted codes are partially evaluated.

From the aesthetic point of view, both proofs and codes should be transformed simultaneously to maintain the 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., first, the all-normalization rule can be applied to the proof of M|0 and M|M in TREE_2, as follows. This kind of proof normalization can be used very often, especially when the well known theorem such as 'all P:nat. P|0' and 'all Q:nat. Q|Q' is refereed from the data bases that store theorems that have already been proved.

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

                                      Note: M|0 == exist D:nat.
                                                     0 = D*M
```

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

By this normalization, the GCD program is translated as follows. The underlined part (----) of g' is changed.

```
f == lambda [W0, W1, W2]. lambda [Z]. (W0, W1, W2)(Z)(A(Z))
        where
        A = mu [ZZ0,ZZ1, ZZ2].
            lambda [X]. lambda[Y].
               if left = AA(X)(Y) then (ZZ0, ZZ1, ZZ2)(pred(X))(Y)
               else (W0, W1, W2)(X)((ZZ0, ZZ1, ZZ2)(pred(X)))
        AA == mu [Z0]. lambda [X]. lambda [Y].
                    if X=0 then right
                    else if left = (lambda [P].
                                    if P=0 then left else right)(Y) then left
                         else if left = Z0(pred(X))(pred(Y)) then left
                              else right


g' == lambda [N]. lambda [ZZ0, ZZ1, ZZ2].
          if left = (lambda [P]. if P=0 then left else right)(N)
          then (lambda [M]. M, 0, 1)
                             ----
          else lambda [M]. (ZZ0, ZZ1, ZZ2)(M mod N)(N)

gcd == f(g')
```

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:

Course of value proof:

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

Proof of course of value Schema by mathematical induction:

```
[all X. (all Y.(Y<X)->P(Y)) -> P(X)]
                PI'
          all Z. P(Z)
------------------------------(->I)
all X. (all Y.(Y<X)->P(Y)) -> P(X)
--> 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 gcd code is as follows:

```
gcd ==  lambda [Z].
        ( lambda [N]. lambda [ZZ0, ZZ1, ZZ2].
             if left = (lambda [P]. if P=0 then left else right)(N)
             then (lambda [M]. M, 0, 1)
             else (lambda [M]. (ZZ0, ZZ1, ZZ2))(M mod N)(N)
        )(Z)(A'(Z))

        where
        A' = mu [ZZ0,ZZ1, ZZ2].
            lambda [X]. lambda[Y].
               if left = AA(X)(Y) then (ZZ0, ZZ1, ZZ2)(pred(X))(Y)
               else (lambda [N]. lambda [ZZ0, ZZ1, ZZ2].
                        if left = (lambda [P]. if P=0 then left else right)(N)
                        then (lambda [M]. M, 0, 1)
                        else (lambda [M]. (ZZ0, ZZ1, ZZ2)(M mode N)(N))
                    )(X)((ZZ0, ZZ1, ZZ2)(pred(X)))
        AA == mu [Z0]. lambda [X]. lambda [Y].
                    if X=0 then right
                    else if left = (lambda [P].
                                        if P=0 then left else right)(Y) then left
                         else if left = Z0(pred(X))(pred(Y)) then left
                              else right
```

The all-normalization rule can also be applied in Gamma1 and Gamma2 in Section 3 combined with the course of value proof given in Section 4; the code is as follows:

```
gcd ==  lambda [Z].
        (lambda [ZZ0, ZZ1, ZZ2].
             if left = (lambda [P]. if P=0 then left else right)(Z)
             then (lambda [M]. M, 0, 1)
             else (lambda [M]. (ZZ0, ZZ1, ZZ2)(M mod Z)(Z))
        )(A'(Z))

        where
```

```
A' = mu [ZZ0,ZZ1, ZZ2].
     lambda [X]. lambda[Y].
        if left = AA(X)(Y) then (ZZ0, ZZ1, ZZ2)(pred(X))(Y)
        else (lambda [ZZ0, ZZ1, ZZ2].
                if left = (lambda [P]. if P=0 then left else right)(X)
                then (lambda [M]. M, 0, 1)
                else (lambda [M]. (ZZ0, ZZ1, ZZ2)(M mode X)(X))
             )((ZZ0, ZZ1, ZZ2)(pred(X)))
AA == mu [Z0]. lambda [X]. lambda [Y].
             if X=0 then right
             else if left = (lambda [P].
                             if P=0 then left else right)(Y) then left
                  else if left = Z0(pred(X))(pred(Y)) then left
                       else right
```

No other normalization rules can be applied any more. The number of reduction needed to calculate gcd of 2 and 3 with this code is 59. The code is slightly more efficient than that given in Section 4. 2.

6.3 Optimized \/ Code

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 5.1, 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 section 4. 2.

On the other hand, as given in 4., 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:

```
gcd ==  lambda [Z].
        (lambda [ZZ0, ZZ1, ZZ2].
             if Z=0
             then (lambda [M]. M, 0, 1)
             else (lambda [M]. (ZZ0, ZZ1, ZZ2)(M mod Z)(Z))
        )(A'(Z))

        where
        A' = mu [ZZ0,ZZ1, ZZ2].
             lambda [X]. lambda[Y].
                if Y < X then (ZZ0, ZZ1, ZZ2)(pred(X))(Y)
                else (lambda [ZZ0, ZZ1, ZZ2].
                        if X=0
                        then (lambda [M]. M, 0, 1)
                        else (lambda [M]. (ZZ0, ZZ1, ZZ2)(M mode X)(X))
                     )((ZZ0, ZZ1, ZZ2)(pred(X)))
```

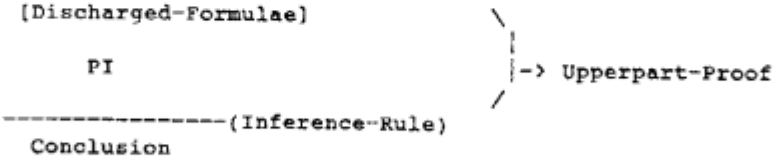
The number of reductions needed to calculate gcd of 2 and 3 with this code is 24. The code is much more efficient.

7. Implementation of the QPC System

An experimental implementation of the QPC is running on the DEC-2060 system. It is the core part of the program extractor with a simple proof checker system. This program is about 800 lines in DEC-10 Prolog.

Proof trees are written in the following data structure:

```
pt( Upperpart-Proof, Conclusion, Inference-Rule, Discharged-Formulae)
```



This data structure is the intermediate code of the CAP-QJ system: proofs written in a proof description language for end users, PDL-QJ, are translated to this data structure. A proof of the GCD specification and extraction of the GCD program are given in Appendix 2 and 3.

The object code of QPC is as follows. A tiny ML [Gordon 79] interpreter is implemented on the DEC-2060 system. All the extracted programs presented in this paper are translated manually to ML programs in a straightforward way and checked. The QPC code to ML code translator program will also be implemented in the near future.

```
IF-THEN-ELSE(Code1, Code2, Code2)  --- if-then-else term
APP(Code1, Code2) --- application of Code1 to Code2
FUN [Variable], Code  --- lambda expression
MU [Variable]. Code --- fixed point operator
[Code-1, ... Code-n] --- conjunction of Code-1, ... and Code-n
```

8. Conclusion

This paper presents proof compiling technique based on typed logical calculus, QJ. Its purpose is to compile formal proofs of specifications of programs into functional style programs.

The current version of the QPC system has mathematical induction schema, but several other induction schemas are necessary to write proofs of a wider range of programs. An additional induction schema is that of course of value induction. This induction schema allows us to write the divide and conquer strategy as typically used in the quick sort algorithm, and it is a the powerful way of writing efficient programs. One way to introduce the schema in QPC is to give a realizability interpretation directly. However, theoretical justification is not as simple as that. Another way is to simulate course of value induction by mathematical induction, and to define the realizer code through that of the mathematical induction rule. However, as stated explicitly in this paper, the extracted codes from the course of value induction schema introduced in the latter way are not efficient. This can be easily seen intuitively: the divide and conquer strategy makes it possible to jump down to any smaller

problems while the strategy corresponding to mathematical induction schema goes down to smaller problems step by step. This fact is illustrated by the GCD program example.

The proof normalization technique corresponds to partial evaluation of the realizer code. This can be used as an optimization technique for the proof compiler system. Several normalization rules are worked on with simple examples. A slightly modified $\vee$ code, i.e., the program extracted from the $\vee$ introduction rule, is introduced as another powerful technique of optimization.

Acknowledgment

Thanks must go to Dr. Aiba, Dr. Murakami, and Mr. Sakai of ICOT, and Mr. Sakurai at Tokyo Metropolitan University, Mr. Kameyama and Professor Sato at Tohoku University, and Dr. Hayashi at Kyoto 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

[Feferman 79] Feferman, S., "Constructive Theories of Functions and Classes", in Logic Colloquium '78, North-Holland, Amsterdam, pp. 159-224, 1979

[Gordon 79] Gordon, M. J. et al, "Edinburgh LCF", LNCS-78, Springer, 1979

[Goto 85] Goto, S., "Concurrency in Proof Normalization", IJCAI-85, pp. 726-729, Los Angeles, 1985

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

[Howard 80] Howard, W. A., "The Formulae-as-types Notion of Construction", in 'Essays on Combinatory Logic, Lambda Calculus and Formalism', eds. J. P. Seldin and J. R. Hindley, Academic Press, 1980

[Martin-Loef 82] Martin-Loef, P., "Constructive Mathematics and Computer Programming", in Logic, Methodology, and Philosophy of Science VI, North-Holland, pp.153-179, 1982

[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, 1986

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

[Takayama 87] Takayama, Y., "Writing Programs as QJ-Proofs and Compiling into PROLOG Programs", 4th. Symposium of Logic Programming, San Francisco, 1987

Appendix 1: Description of GCD Proof in PDL-QJ

```
all N, M:nat. exist D:nat. (D|N & D|M)
since course-of-value induction on N
   let N:nat be arbitrary
   induction-hypothesis is
        all L:nat. (L< N -> all M:N. exist D:nat.
                          (D|L & D|M))
   all M:nat. exist D:nat. (D|N & D|M)
   since divide-and-conquer
     case N=0
        all M:nat. exist D:nat. (D|N & D|M)
        since
            let M:nat be arbitrary
            M|N clearly
            M|M clearly
            hence exist D:nat. (D|N & D|M)
        end_since
     else
        N > 0
        all M:nat. exist D:nat. (D|N & D|M)
        since
           let M:nat be arbitrary
            R := M mod N
            R < N clearly
            hence all M:nat. exist D:nat
                (D|R & D|M)    by induction-hypothesis
            hence exist D:nat. (D|R & D|M)
            let D:nat be such that D|R & D|N
            exist Q:nat. M = N*Q + R  by definition of r
            hence D|M clearly
            hence exist D:nat. (D|N & D|M)
        end_since
end_since
```

Appendix 2: Proof of GCD Specification

```
proof_tree(PROOF_TREE) :-

F_1 = all(N,
        (all(LL, ((LL<N) -> all(M, exist(D,
                                    exist(D0, LL = D0*D)/\
                                    exist(D0, M = D0*D)))))
          -> all(M, exist(D, exist(D0, N = D0*D)/\exist(D0, M=D0*D)
                 ))))),
F_2 =   (all(LL, ((LL<N) -> all(M, exist(D,
                                    exist(D0, LL = D0*D)/\
                                    exist(D0, M = D0*D)
                                ))))
         -> all(M, exist(D,
                    exist(D0, N=D0*D)/\exist(D0, M = D0*D)
                 )))),
F_3 = all(M, exist(D, exist(D0, N=D0*D)/\exist(D0, M = D0*D)
        )),
F_4 = (N = 0) \/ (N> 0),
F_5 = F_3,
F_6 = F_3,
F_7 = all(P, (P=0)\/(P>0)),    % Axiom 1
F_8 = (0 = N),
F_9 = all(M, exist(D, exist(D0, 0=D0*D)/\exist(D0, M=D0*D)
        )),
F_10 = exist(D, exist(D0, N=D0*D)/\exist(D0, M=D0*D)
             ),
F_11 = exist(D, exist(D0, 0=D0*D)/\exist(D0, M=D0*D)
             ),
F_12 = exist(D, exist(D0, 'M mod N' = D0*D)/\exist(D0, N = D0*D)
             ),
F_13 = exist(D, exist(D0, N = D0*D)/\exist(D0, M=D0*D)
             ),
F_14 = exist(D0, 0=D0*M)/\exist(D0, M = D0*M),
F_15 = all(M, exist(D, exist(D0, 'M mod N'=D0*D)/\exist(D0,M=D0*D))),
F_16 = exist(D0, N=D0*T)/\exist(D0, M=D0*T),
F_17 = exist(D0, 0= D0*M),
F_18 = exist(D0, M = D0*M),
F_19 = ('M mod N' < N),
F_20 = (('M mod N' < N) -> all(M, exist(D,
                                  exist(D0, 'M mod N' =D0*D)/\
                                  exist(D0, M=D0*D)
                              ))),
F_21 = exist(D0, N = D0*T),
F_22 = exist(D0, M = D0*T),
F_23 = all(Q, exist(D0, 0 = D0*Q)
          ), % Axiom 2
F_24 = all(R, exist(D0, R= D0*R)
          ), % Axiom 3
F_25 = (N > 0),
F_26 = (L == M mod N),
F_27 = all(LL, ((LL< N) ->
                 all(M, exist(D, exist(D0, LL=D0*D)/\exist(D0, M=D0*D)
                    )))), % HYP
F_28 = exist(D0, 'M mod N' = D0*T)/\exist(D0, N  = D0*T),
F_29 = exist(QQ, M = (N*QQ) + 'M mod N'),
F_30 = F_22,
F_31 = F_25,
F_32 = F_26,
F_33 = (((N*QQ) + 'M mod N') = M),
F_34 = exist(D0, (N*QQ)+'M mod N' = D0*T),
F_35 = exist(D0, 'M mod N' = D0*T),
F_36 = exist(D0, N*Q= D0*T),
```

```
F_37 = exist(D0, 'M mod N' = D0*T)/\exist(D0, N = D0*T),
F_38 = exist(D0, N = D0*T),
F_39 = exist(D0, 'M mod N'= D0*T)/\exist(D0, N= D0*T),

PROOF_TREE_1 =
pt([
    pt([
        pt([
            pt([pt([], N, '$assum$', []), PROOF_TREE_OF_LEMMA_1],
               F_4, '$all-E$',[]),
            pt([
                pt([], F_8, '$assum$',[]),
                pt([
                    pt([
                        pt([], M, '$assum$',[]),
                        pt([
                            pt([
                                pt([], M, '$assum$', []),
                                PROOF_TREE_OF_LEMMA_2
                               ],
                                F_17, '$all-E$', []),
                            pt([
                                pt([], M, '$assum$', []),
                                PROOF_TREE_OF_LEMMA_3
                               ],
                                F_18, '$all-E$', [])
                           ],
                           F_14, '$and-I$',[])
                       ],
                      F_11, '$ex-I$', [])
                  ],
                  F_9, '$all-I$', [M])
              ],
              F_5, '$=-E$', []),
            pt([
                pt([
                    pt([
                        pt([], N, '$assum$',[]),
                        pt([
                            pt([
                                pt([], F_25, '$assum$', []),
                                pt([], M, '$assum$',[]),
                                pt([], F_26, *, [])
                               ],
                               F_19, *, []),
                            pt([
                                pt([
                                    pt([], F_31, '$assum$',[]),
                                    pt([], F_32, *,[]),
                                    pt([], X, '$assum$',[])
                                   ],
                                   'M mod N', *, []),
                                pt([], F_27, '$assum$',[])
                               ],
                               F_20, '$all-E$',[])
                           ],
                           F_15, '$->E$', [])
                       ],
                       F_12, '$all-E$', []),
                    pt([
                        pt([], T, '$assum$',[]),
                        pt([
                            pt([
                                pt([], F_28, '$assum$',[])
                               ],
                               F_21, '$and-E$',[]),
```

```
                                pt([
                                    pt([], F_29, *, []),
                                    pt([
                                        pt([], F_33, '$assum$',[]),
                                        pt([
                                            pt([
                                                pt([], F_37,'$assum$',[])
                                                ],
                                                F_35, '$and-E$',[]),
                                            pt([
                                                pt([], QQ, '$assum$',[]),
                                                pt([
                                                    pt([], F_39, '$assum$',[])
                                                    ],
                                                    F_38, '$and-$',[])
                                                ],
                                                F_36, *, [])
                                            ],
                                            F_34, *,[])
                                        ],
                                        F_30, '$=-E$',[])
                                    ],
                                    F_22, '$ex-E$', [QQ, (M*QQ)+L])
                                ],
                                F_16, '$and-I$',[])
                            ],
                            F_13, '$ex-I$', [])
                        ],
                        F_10, '$ex-E$', [T, F_28])
                    ],
                    F_6, '$all-I$', [M])
                ],
                F_3, '$\/-E$', [N = 0, F_25])
            ],
            F_2, '$->I$', [F_27])
        ],
        F_1, '$all-I$', [N]),
% ------------------------------------------------------------
FF_1 = (all(N, (all(LL, ((LL < N)
                        -> all(M, exist(D, exist(D0,LL=D0*D)
                                          /\exist(D0, M=D0*D)))))
                -> all(M, exist(D, exist(D0, N=D0*D)
                                  /\exist(D0, M=D0*D)))))
        -> all(X, all(M, exist(D, exist(D0, X=D0*D)
                                  /\exist(D0, M=D0*D))))),
FF_2_3 = all(X, all(M, exist(D, exist(D0, X=D0*D)
                                /\exist(D0, M=D0*D)))),
FF_3 =  all(M, exist(D, exist(D0, X=D0*D)/\exist(D0, M=D0*D))),
FF_4 =  all(LL, ( (LL<X) -> all(M, exist(D, exist(D0,LL=D0*D)
                                          /\exist(D0, M=D0*D))))),
FF_5 = (all(LL, ( (LL<N)
                  -> all(M, exist(D, exist(D0,LL=D0*D)
                                    /\exist(D0, M=D0*D)))))
        ->  all(M, exist(D, exist(D0, N=D0*D)
                           /\exist(D0, M=D0*D)))),
FF_6 =  N,
FF_7 =  all(N, (all(LL, ( (LL<N)
                         -> all(M, exist(D, exist(D0,LL=D0*D)
                                           /\exist(D0, M=D0*D)))))
                -> all(M, exist(D, exist(D0, N=D0*D)
                                  /\exist(D0, M=D0*D))))),
FF_8 =  X,
FF_9 =  all(N, all(LL, ( (LL<N) -> all(M, exist(D, exist(D0,LL=D0*D)
                                                  /\exist(D0, M=D0*D)))))),
FF_10 = all(LL, ( (LL<0) -> all(M, exist(D, exist(D0,LL=D0*D)
                                          /\exist(D0, M=D0*D))))),
```

```
FF_11 = all(LL, ( (LL<(N+1)) -> all(M, exist(D, exist(D0,LL=D0*D)
                                                  /\exist(D0, M=D0*D))))),
FF_12 = ((LL<0) -> all(M, exist(D, exist(D0,LL=D0*D)
                                  /\exist(D0, M=D0*D)))),
FF_13 = ((LL<(N+1)) -> all(M, exist(D, exist(D0,LL=D0*D)
                                    /\exist(D0, M=D0*D)))),
FF_14 = all(M, exist(D, exist(D0,LL=D0*D)
                        /\exist(D0, M=D0*D))),
FF_15 = all(M, exist(D, exist(D0,LL=D0*D)
                        /\exist(D0, M=D0*D))),
FF_16 = '$bottom$',
FF_17 = (LL<N)\/(N=LL),
FF_18 = all(M, exist(D, exist(D0,LL=D0*D)
                        /\exist(D0, M=D0*D))),
FF_19 = all(M, exist(D, exist(D0,LL=D0*D)
                        /\exist(D0, M=D0*D))),
FF_20 = ((LL<0) -> '$bottom$'),
FF_21 = (LL<0),
FF_22 = (LL<N),
FF_23 = ((LL<N) -> all(M, exist(D, exist(D0,LL=D0*D)
                                  /\exist(D0, M=D0*D)))),
FF_24= (N = LL),
FF_25 = all(M, exist(D, exist(D0, N=D0*D)
                        /\exist(D0, M=D0*D))),
FF_26 = LL, % FF_26 = N,
FF_27 = all(LL, ((LL<N) -> all(M, exist(D, exist(D0,LL=D0*D)
                                    /\exist(D0, M=D0*D))))),
FF_28 = all(LL, ((LL<N) -> all(M, exist(D, exist(D0,LL=D0*D)
                                    /\exist(D0, M=D0*D))))),
Gamma1 = pt([ pt([], FF_6, '$assum$',[]),
              pt([], FF_7, '$assum$',[])],
            FF_5, '$all-E$',[]),
Gamma2 = pt([pt([],X,  '$assum$',[]),
             pt([], FF_7, '$assum$',[])],
             FF_29, '$all-E$', []),
FF_29 = (all(LL, ( (LL<X)
                  -> all(M, exist(D, exist(D0,LL=D0*D)
                                    /\exist(D0, M=D0*D)))))
       ->  all(M, exist(D, exist(D0, X=D0*D)
                          /\exist(D0, M=D0*D)))),

PROOF_TREE_2 =
pt([
    pt([
        pt([
            pt([
                pt([], FF_8,'$assum$',[]),
                pt([
                        pt([
                            pt([
                                pt([
                                    pt([
                                        pt([], FF_21, '$assum$',[]),
                                        pt([pt([],M, '$assum$',[])],
                                            FF_20, *, [])
                                       ],
                                       FF_16, '$->E$',[])
                                   ],
                                   FF_14, '$bottom-E$',[])
                               ],
                               FF_12, '$->I$',[FF_21])
                           ],
                           FF_10, '$all-I$',[LL]),
                        pt([
                            pt([
                                pt([
```

```
                                    pt([
                                        pt([], LL<(N+1), '$assum$',[]),
                                        pt([
                                            pt([], LL, '$assum$',[]),
                                            pt([
                                                pt([], N, '$assum$',[]),
                                                PROOF_TREE_OF_LEMMA_4
                                               ],
                                               all(Y,
                                                (Y<(N+1) -> (Y<N)\/(N=Y))),
                                               '$all-E$',[])
                                           ],
                                           (LL<(N+1) -> (LL<N)\/(N=LL)),
                                           '$all-E$', [])
                                       ],
                                       FF_17,'$->E$',[]),
                                    pt([
                                        pt([], FF_22, '$assum$', []),
                                        pt([
                                            pt([], FF_26, '$assum$',[]),
                                            pt([], FF_27, '$assum$',[])
                                           ],
                                           FF_23, '$all-E$', [])
                                       ],
                                       FF_18, '$->E$',[]),
                                    pt([
                                        pt([], FF_24, '$assum$',[]),
                                        pt([
                                            pt([],FF_27, '$assum$',[]),
                                            Gamma1
                                           ],
                                           FF_25, '$->E$',[])
                                       ],
                                       FF_19, '$=-E$', [])
                                    ],
                                    FF_15, '$\/-E$', [FF_22, FF_24])
                                    ],
                                    FF_13, '$->I$',[(LL<(N+1))])
                                ],
                                FF_11, '$all-I$', [LL])
                            ],
                            FF_9, '$nat-ind$',[FF_27])
                        ],
                        FF_4, '$all-E$',[]),
                    Gamma2
                    ],
                    FF_3, '$->E$',[])
                ],
                FF_2_3, '$all-I$',[X])
            ],
            FF_1, '$->I$', [FF_7]),

PROOF_TREE
= pt([PROOF_TREE_1, PROOF_TREE_2],
    all(X, all(M, exist(D, exist(D0, X=D0*D)/\exist(D0, M=D0*D)))),
    '$->E$', []),

%%%% LEMMA_1: all(P, (P=0)\/(P>0)) %%%%
PROOF_TREE_OF_LEMMA_1 =
pt([
    pt([
        pt([
            pt([],0,'*',[])
           ],
           0=0, '*',[])
       ],
```

```
        (0=0)\/(0>0), '$\/-I$',[]),
     pt([
         pt([
             pt([], (P=0)\/(P>0), '$assum$',[]),
             pt([
                 pt([pt([], P=0, '$assum$',[])],
                    (1= (P+1)), *,[]),
                 pt([],
                    (1>0), *, [])
                ],
                ((P+1)>0), '$=-E$', []),
             pt([
                 pt([
                     pt([], P>1, '$assum$',[])
                    ],
                    (P+1)>1, *,[]),
                 pt([], 1>0, *,[])
                ],
                ((P+1)>0), *,[])
            ],
            ((P+1)>0), '$\/-E$',[P=0, (P>0)])
        ],
        ((P+1)=0)\/((P+1)>0), '$\/-I$',[])
    ],
    F_7, '$nat-ind$', [(P=0)\/(P>0)]),

%%%% Lemma 2: all(Q, exist(D0, 0 = D0*Q) %%%%
PROOF_TREE_OF_LEMMA_2 =
pt([
     pt([
         pt([], 0, *,[]),
         pt([], 0=0*Q, *,[])
        ],
        exist(D0, 0 = D0*Q),'$ex-I$', [])
    ],
    F_23, '$all-I$', [Q]),

%%%% Lemma 3: all(R, exist(D0, R= D0*R)  %%%%
PROOF_TREE_OF_LEMMA_3 =
pt([
     pt([
         pt([], 1, *,[]),
         pt([], R = 1*R, *, [])
        ], exist(D0, R= D0*R), '$ex-I$',[])
    ],
    F_24, '$all-I$', [R]),

%%%% Lemma 4: all(X, all(Y, ( (Y<(X+1) -> (Y<X)\/(X=Y))))) %%%%
PROOF_TREE_OF_LEMMA_4 =
         pt([TREE_1, TREE_2],
            all(X, all(Y, ( (Y<(X+1) -> (Y<X)\/(X=Y))))),
           '$nat-ind$', [all(Y, ( (Y<(X+1) -> (Y<X)\/(X=Y))))]),
TREE_1 =
         pt([
             pt([
                 pt([
                     pt([
                         pt([
                             pt([
                                 pt([],0,'*',[])
                                ],
                                0=0, '*',[]
                               ),
                             pt([],0<1, '$assum$', [])
                            ],
                            (0=0)/\(0<1),
```

```
                        '$and-I$',[]
                         )
                      ],
                      0=0,
                     '$and-E$',[]
                     )
                   ],
                   (0<0)\/(0=0),
                   '$\/-I$',[])
                ],
                ((0<1) -> (0<0)\/(0=0)),
               '$->I$',[(0<1)]),
             pt([
                 pt([
                     pt([
                         pt([
                             pt([], (Y+1)<1, '*',[])
                            ],
                            Y<0,'*',[]),
                          pt([], Y, '$assum$',[])
                         ],
                         '$bottom$', '*', []
                        )
                     ],
                     ((Y+1)<0)\/(0=(Y+1)),
                     '$bottom-E$',[]
                    )
                 ],
                 (((Y+1)<1) -> ((Y+1)<0)\/(0=(Y+1))),
                 '$->I$', [ (Y+1)<1 ]
                )
             ],
             all(Y, ((Y<1) -> (Y<0)\/(0=Y))),
            '$nat-ind$', [((Y<1) -> (Y<0)\/(0=Y))]),
TREE_2 =
        pt([
            pt([
                pt([TREE_21, TREE_22, TREE_23],
                   (Y<(X+1))\/((X+1)=Y),
                   '$\/-E$', [(0=Y), (Y>0)])
                ],
                ((Y<(X+2)) -> (Y<(X+1))\/((X+1)=Y)),
               '$->I$', [ Y<(X+2) ]
               )
            ],
            all(Y, ((Y<(X+2)) -> (Y<(X+1))\/((X+1)=Y))),
           '$all-I$', [Y]
           ),
TREE_21 =
        pt([
            pt([], Y, '$assum$',[]),
            TREE_211
           ],
           ((0=Y)\/(Y>0)),
          '$all-E$', []),

        TREE_211 =  % Proof of all N:nat. 0=N\/N>0
                pt([
                    pt([
                        pt([
                            pt([],0,'*',[])
                           ],
                           0=0, '*',[])
                       ],
                       (0=0)\/(0>0), '$\/-I$',[]),
                    pt([
```

```
                    pt([
                        pt([], (0=N)\/(N>0), '$assum$',[]),
                        pt([
                            pt([pt([], 0=N, '$assum$',[])],
                               (1= (N+1)), *,[]),
                            pt([],
                               (1>0), *, [])
                           ],
                           ((N+1)>0), '$=-E$', []),
                        pt([
                            pt([
                                pt([], N>1, '$assum$',[])
                               ],
                               (N+1)>1, *,[]),
                            pt([], 1>0, *,[])
                           ],
                           ((N+1)>0), *,[])
                       ],
                       ((N+1)>0), '$\/-E$',[0=N, (N>0)])
                   ],
                   (0=(N+1))\/((N+1)>0), '$\/-I$',[])
               ],
               all(N, (0=N)\/(N>0)), '$nat-ind$', [(0=N)\/(N>0)]),

TREE_22 =
        pt([
            pt([
                pt([], 0=Y, '$assum$',[]),
                pt([
                    pt([], X, '$assum$',[])
                   ],
                   (0<(X+1)), '*', [])
               ],
               (Y<(X+1)),
              '$=-E$',[])
           ],
          (Y<(X+1))\/((X+1)=Y),
          '$\/-I$', []
          ),

TREE_23 =
        pt([
            pt([
                pt([pt([], Y<(X+2), '$assum$',[])],
                   (Y-1)<(X+1), '*', []),
                pt([
                    pt([
                        pt([], Y>0, '$assum$',[])
                       ],
                       Y-1, '*', []),
                    pt([],
                       all(Y, ( (Y<(X+1) -> (Y<X)\/(X=Y)))),
                       '$assum$', [])
                   ],
                   (((Y-1)<(X+1)) -> ((Y-1)<X)\/(X=(Y-1))),
                   '$all-E$', [] )
               ],
               ((Y-1)<X)\/(X=(Y-1)),
               '$->E$', []),
            pt([
                pt([
                    pt([], (Y-1)<X, '$assum$', [])
                   ],
                   Y<(X+1), '*', [])
               ],
               (Y<(X+1))\/((X+1)=Y),
```

```
      '$\/-I$',[]),
   pt([
       pt([ pt([], X=(Y-1), '$assum$',[]) ],
          (X+1)=Y, '*',[])
      ],
      (Y<(X+1))\/((X+1)=Y),
      '$\/-I$',[])
 ],
(Y<(X+1))\/((X+1)=Y),
 '$\/-E$', [ (Y-1)<X, X=(Y-1) ]
).
```

Appendix 3: Extraction of GCD Program

```
          ### Proof Compiler System ###
            August 19 in 1987   Y.Takayama

Proof File Name >> 'gcd.pr'.


gcd.pr consulted   5692 words      3.61 sec.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Proof of Common Divisor  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

yes
| ?- pcomp.
Display Conclusions? [y/n]n.

Display Rules? [y/n]n.

[Conclusion]
all(_153, all(_102, exist(_103,
    exist(_104,_153=_104*_103)/\exist(_104,_102=_104*_103))))



Code is ==>
APP(
  FUN([_647,_657,_662], FUN([_153],
    APP(
      APP([_647,_657,_662],_153),
      APP(MU([_771,_781,_786], FUN([_100], FUN([_101],
           IF-THEN-ELSE(
              left=APP(
                      APP(MU([_976],FUN([_153],
                             IF-THEN-ELSE(_153=0,
                                FUN([_187],right),
                                FUN([_187],
                                  IF-THEN-ELSE(
                                     left=APP(
                                            FUN([_100],
                                              IF-THEN-ELSE(_100=0,left,right)),
                                            _187),
                                     left,
                                     IF-THEN-ELSE(
                                          left=APP(
                                                APP([_976],pred(_153)),
                                                _187-1),
                                          left,right)))))
                                ),
                              _100),
                            _101
                            ),
                 APP(
                    APP([_771,_781,_786],pred(_100)),
                    _101
                    ),
                 APP(
                   APP([_647,_657,_662],_100),
                   APP([_771,_781,_786],pred(_100))))))
              ),
           _153))
      )),

     FUN([_100],FUN([_1407,_1417,_1422],
```

```
      IF-THEN-ELSE(left=APP(FUN([_111],IF-THEN-ELSE(_111=0,left,right)),_100),
         FUN([_102],[_102,[APP(FUN([_129],0),_102),APP(FUN([_131],1),_102)]]),
         FUN([_102],APP(APP([_1407,_1417,_1422],M mod N),_100)))))
  )

yes
| ?-
```