ICOT Technical Report: TR-244

TR-244

# Writing Program as QJ Proof and Compiling into PROLOG Program

by

Y. Takayama

March, 1987



ICOT

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

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

# Writing Program as QJ Proof and Compiling into PROLOG Program

Yukihide Takayama

Institute for New Generation Computer Technology

Mita Kokusai Building, 21F,
1-4-28 Mita, Minato-ku, Tokyo 108 Japan
takayama%icot.uucp@eddie.mit.edu

**Abstract**

This paper presents a method for writing a *constructive proof* of a formal specification of a program in the typed logical system QJ. Compilation into executable code, such as PROLOG on the PSI machine is also described. We use the provisional proof description language PDL/QJ, a modified version of PDL for the CAP system which we have developed as a proof checker system for linear algebra. We assume here a proof checker system like CAP with QJ as its underlying logic.

## 1 Introduction

The idea of *realizability interpretation* which extracts executable code from a constructive proof is rather old ([Kleene 45],[McCarty 84]). From the point of view of building an intelligent programming system based on the programming paradigm *writing programs as constructive proofs* ([Beeson 83], [Martin-Löf 82], [de Brujin 80]), this idea can be applied to *a proof compiler* that generates the executable code on a computer. Several systems which implement the proof compiling facility have been developed ([Hayashi 86],[Constable 86]) and they generate code of functional style programs, such as LISP, because the code extracted using the realizability interpretation is a combination of terms expressing functions. We use a provisional proof description language similar to PDL([Sakai 86]) allowing us to write in a style almost as natural as that found in ordinary mathematics textbooks.

We have studied the proof compilation technique based on QJ – typed logical calculus – ([Sato 85],[Sato 86]) that generates PROLOG programs. The theoretical aspects of the realizability interpretation are covered in that theory and more generally in [Beeson 85] and [McCarty 84]. A proof compilation algorithm is implicitly given in the proof of "soundness of the realizability interpretation", in that the object code is contained in a subset of the typed functional and logical language *Quty* , the typed version of Qute([Sato 84]).

Several examples are presented in this paper to show how a program can be written as a constructive proof of a formal specification like that found in a mathematics textbook and how it can be compiled into a PROLOG program . We chose PROLOG as the

language in which target programs are written because we are going to implement our experimental environment on the PSI machine ([Uchida 83]).

## 2 Program Development in QJ

The three examples below how are used to demonstrate how we can develop programs within the concept of QJ.

1. A program which take a natural number list as input and returns the list whose elements are the elements of the original list multiplied by two
2. Bubble sort algorithm
3. Quick sort algorithm

Proofs are written, for human readability, mainly in natural language, but they can be easily rewritten into a machine readable proof description language, called PDL/QJ, a slightly modified version of PDL of the CAP system([Hirose 86],[Sakai 86]). Notice that in our programming paradigm, a proof checker like the CAP system is essential to fill the gaps in proofs written in a rather natural style for the human mathematician. Tedious inference steps like arithmetic operations and term manipulations are often skipped in ordinary proofs written by mathematicians. Examples of such proofs are given below.

### 2.1 A Simple Example

Here is a simple example illustrating how we can write formal specifications and proofs. We don't use the 'refinement logic' style, which allows a highly interactive programming support system like the Nuprl system ([Constable 86]), because we want to write proofs in a style that is natural for ordinary mathematicians, and we want the proof to be the documentation of the program. For the same reason we do not adopt the 'formulae as type' notion of mathematical descriptions. We want to distinguish *type* and *logic*.

(1) Informal Specification
A program which doubles each element of a natural number list X
(2) Formal Specification
We write a formal specification of a program as a logical sentence, i.e., a theorem, in first order logic with (in)equality which has the rich term structure defined in QJ. The logical sentence must have the following structure:

$$\forall X{:}\mathrm{Type}.\exists Y{:}\mathrm{Type}.F(X,Y)$$

where 'X : Type' and 'Y : Type' specify input and output parameters and their types respectively. F(X,Y) defines the logical relation between input and output. In this paper `nat` and `L(nat)` mean the type of non-negative integers and the type of lists with `nat`-type elements respectively. We use # and -> as Cartesian product and function type constructors.

**Theorem 1** *For any natural number list $X$ there exists a natural number list $Y$ such that*
*a) length of $X$ is equal to length of $Y$*
*b) for any natural number $I$, if $1 \leq I \leq length(X)$ then the $I$-th element of $Y$ is equal to the $I$-th element of $X$ multiplied by 2*

We write this formal specification as follows in PDL/QJ. The proof part is given later.

```
theorem  SIMPLE_EXAMPLE :
  all X : L(nat), exist Y : L(nat).
      (
       length(X) = length(Y)
       &
       all I : nat.
        ( 1 =< I =< length(X)
           -> 2*elem(I,X) = elem(I,Y)
        )
      )
  proof
       ..................
  end_proof
end_theorem
```

where length and elem are QJ-definable functions which we can define in PDL/QJ as follow

```
function  length : L(nat) -> nat
  attain
      length = fun [X]. if X = nil then 0
                        else length(tl(X)) + 1
  existence
      ..............
  uniqueness
      ..............
end_function

function elem : nat#L(nat) -> nat
  attain
      elem = fun [I,X]. if I = 0 then $abort$
                        else if I = 1 then hd(X)
                             else elem(I-1,tl(X))
  existence
      ..............
  uniqueness
      ..............
end_function
```

fun means lambda abstraction. hd,tl refer to the head and tail of a list respectively, and those are also QJ-definable function, but we skip the definition. Proofs of existence and uniqueness of the function can also be written in QJ, but we skip them too.

(3) Proof of Specification
The proof proceeds by induction on the structure of list $X$
1) Base Step
Assume $X = nil$. We can take $Y = nil$ as the list satisfying conditions a) and b)

2) Induction Step

Suppose for list $X$ we have obtained the list $Y$ satisfying condition a) and b). Then we have to construct a list $Ls$ from $Y$ satisfying the following conditions. For an arbitrary natural number $a$

a') length of list $a.X$ is equal to that of $Ls$

b') the $N$-th element of $Ls$ is equal to the $N$-th element of $a.X$ multiplied by 2 where $1 \leq N \leq length(a.X)$

We can take $Ls = 2a.Y$. We now check the conditions a') and b').

a') $length(a.X) = length(X) + 1$, $length(2a.Y) = length(Y) + 1$ and because of condition a) for $X$ and $Y$, we get our conclusion.

b') We denote here $elem(I, X)$ as $I$-th element of list $X$. Now $elem(N+1, Ls) = elem(N+1, 2a.Y) = elem(N, Y)$ and because of condition b) for $X$ and $Y$, $elem(N, Y) = 2 \cdot elem(N, X)$ so that we get $elem(N+1, Ls) = 2 \cdot elem(N, X)$. Now let $M = N + 1$ then $2 \leq M \leq length(X) + 1 = length(a.X)$. And $elem(M, Ls) = 2 \cdot elem(M, a.X)$ because $elem(M, a.X) = elem(N, X)$. In addition, $elem(1, Ls) = 2 \cdot a$ and $elem(1, Ls) = 2 \cdot elem(1, a.X)$ because $elem(1, a.X)$. So that we get $elem(N, Ls) = 2 \cdot elem(N, a.X)$ where $1 \leq N \leq length(a.X)$

q.e.d.

We give this proof below to demonstrate the flavor of *programming* in PDL/QJ.

```
since induction of X : L(nat)
base /* X = nil */
   length(X) = length(nil)
   all I : nat.
     (1=<I=<length(X) -> 2*elem(I,X) = elem(I,nil)
   since
      let I : N be such that
      1=<I=<length(X)
      contradiction
      hence 2*elem(I,X) = elem(I,nil)
   end_since
   hence concluded
step /* not (X = nil) */
   let a : nat, X : L(nat) be arbitrary
   ind_hyp_is
   some t : L(nat).
      (HYP_1: length(X) = length(t)
       &
       HYP_2: all I : nat.
                 (1=<I=<length(X)
                      -> 2*elem(I,X) = elem(I,t))
   length(a.X) = length(X) + 1
   length(2*a.X) = length(X) + 1
   hence length(a.X) = length(2*a.t) by HYP_1
   all I : N
     (1=<I=<length(a.X)
      -> 2*elem(I,a.X) = elem(I,2*a.t))
```

```
since
   let I : nat be such that
   1=<I=<length(a.X)
   then 2*elem(I,a.X) = elem(I,2*a.t)
   since divide and conquer
                      I=1 | 2=<I=<length(a.X)
      case I=1
         2*elem(I,a.X)
         == 2*[if I=0 then $abort$
               else if I=1 then hd(a.X)
                    else elem(I-1,tl(a.X))]
         = 2*hd(a.X) = 2*a
         elem(I,2*a.t)
         == if I=0 then $abort$
            else if I=1 then hd(2*a.t)
                 else elem(I-1,tl(2*a.t))
         = hd(2*a.t) = 2*a
         hence 2*elem(I,a.X) = elem(I,2*a.t)
      case 2=<I=<length(a.X)
         Label_1: 2*elem(I,a.X)
         == 2*[if I=0 then $abort$
               else if I=1 then hd(a.X)
                    else elem(I-1,tl(a.X))]
         = 2*elem(I-1,tl(a.X))
         = 2*elem(I-1,X)
         on_the_other_hand
         2=<I=<length(a.X) by assumption
         length(a.X) = length(X)+1
         hence 1=<I-1=<length(X)
         hence LABEL_2: 2*elem(I-1,X)
               = elem(I-1,t) by HYP_2
         hence 2*elem(I,a.X) = elem(I-1,t)
                          by LABEL_1,LABEL_2
         elem(I,2*a.t)
         == if I=0 then $abort$
            else if I-1 then hd(2*a.X)
                 else elem(I-1,tl(2*a.t))
         = elem(I-1,tl(2*a.t)) by assumption
         = elem(I-1,t)
         hence 2*elem(I,a.X) = elem(I,2*a.t)
    end_since
  end_since
  hence concluded
end_since
```

If you do not want to write a long proof of such an easy program as this, you can directly define it as a function and refer to it from within other proofs.

## 2.2 Sorting Algorithm

Two extreme examples are used to demonstrate how algorithmic features are reflected in the proof strategy of formal specification. We take the bubble sort and quick sort algorithms([Aho 74]), which exibit a remarkable contrast in execution efficiency.

### 2.2.1 Bubble Sort Algorithm

(1) Formal Specification

**Theorem 2** *For any natural number list $X$, there exists a natural number list $Y$ such that*
*a) Length of $Y$ is equal to length of $X$*
*b) Every element occuring in $X$ is also occurs in $Y$*
*c) $Y$ is sorted*

Before proceeding to proof, we must define the notion 'sorted'. This can be done generally in QJ. But to avoid the tedious definition of ordering in the underlying type structure of lists we just refer to 'list' as a list whose element is a natural number in the following description.

**Definition1::** Let $X$ be a list of length $N$.
$X$ is *SORTED*
$\Longleftrightarrow$
for any natural number $I$,$J$
If $1 \leq I \leq J \leq N$ then ($I$-th element of $X$) $\leq$ ($J$-th element of $J$)

We can write the Theorem and Definition above formally in PDL/QJ as follows.

```
predicate sorted(X:L(nat)) is
   all I, J : nat.
     ( 1 =< I=<J=<length(X)
       -> elem(I,X) =< elem(J,X))
end_predicate.

function occur_no : nat#L(nat) -> nat
   attain
     occur_no = fun [N,X].
                  if X = nil then 0
                  else if N = hd(X)
                       then 1+occur_no(N,tl(X))
                       else occur_no(N,tl(X))
   existence
      ..............
   uniqueness
      ..............
end_function

theorem LIST_SORT :
```

```
   all X : L(nat). exist Y : L(nat).
     (all N : nat.(occur_no(N,X) = occur_no(N,Y))
      &
     sorted(Y)  )
   proof
      ..............
   end_proof
end_theorem
```

(2) Proof Strategy

The outline of the proof of Theorem 2 considered here corresponds to the bubble sort program. The statement of the theorems and definition below are given both informally and formally, i.e., in PDL/QJ. But only an informal mathematical style description of the proof will be given here for brevity.

**Lemma 1** *Let $X$ be an arbitrary list of length $N$, and $J$ be an arbitrary natural number such that $J \leq N$. Then exists the list $Y$ satisfying the following conditions;*
*a) Length of $Y$ is equal to length of $X$, and every element occuring in $X$ is also occurs in $Y$ [ PERMUTATION CONDITION ]*
*b) if $J \neq 0$, for every natural number $I$ such that $J \leq I \leq N$*
*(J-th element of Y) $\leq$ (I-th element of Y)*

This is written in PDL/QJ thus;

```
theorem LEMMA_1
   all X : L(nat), all J : nat.
      (
       J=<length(X)
       -> exist Y : L(nat).
            (all K : nat.
                occur_no(K,X)=occur_no(K,Y)
             &
             all I : nat.
               (1 =<J=<I=<length(X)
                -> elem(J,Y) =< elem(I,Y))
            )
       )
   proof
      ..............
   end_proof
end_theorem
```

≪Informal Proof of lamma1≫

The proof proceeds by the divide and conquer strategy; $X = nil$ or $X \neq nil$. If $X = nil$, it is sufficient that $Y$ is nil. If $X \neq nil$, let $M \equiv N - J$, and we use mathematical induction on $M$. (note : $0 \leq M < N$ ) If $M = 0$(i.e.,$J = N$), it is sufficient that $Y$ is just $X$ itself. Now assume that if $1 \leq M < N$(i.e.,$0 < J \leq N - 1$) there is a list $Y_0$ such that a) $Y_0$ is a permutation of $X$, and b) for any natural number I such that $J \leq I \leq length(X)$, $elem(J, Y_0) \leq elem(I, Y_0)$. Assume also that $M + 1 < N$, otherwise

we have nothing to prove. Now consider the case when $M+1(=N-(J-1))$ and define a list $Y$ as follows: When $elem(J-1,Y_0) \leq elem(J,Y_0)$ , we define $Y \equiv Y_0$. When $elem(J,Y_0) < elem(J-1,Y_0)$, we define $Y$ as

$$elem(J-1,Y) = elem(J,Y_0)$$

$$elem(J,Y) = elem(J-1,Y_0)$$

$$elem(K,Y) = elem(K,Y_0) \quad (if\ K \neq J-1,J).$$

From the above definition and the induction hypothesis we can check easily that this Y satisfies the following;

a') $Y$ is a permutation of $X$

b') for all natural number $I$ such that $J-1 \leq I \leq N$

$$elem(J-1,Y) \leq elem(I,Y)$$

q.e.d.

**Definition2::** Let $X \equiv X_1.....X_N.nil$ be a list of length $N$ and $J$ be a natural number.
Then $X$ is *PARTIALLY SORTED WITH REGARD TO J*
$\Longleftrightarrow$
If $J \leq N$ then list $X_1.....X_J.nil$ is sorted

We write this definition in PDL/QJ as follows;

```
predicate partially_sorted(J:nat, X:L(nat)) is
   J =< length(X)
   -> all K,L : nat.
        ( 1 =< K =< L =< J
          -> elem(K,X) =< elem(L,X) )
end_predicate
```

**Lemma 2** *Let $X$ be an arbitrary list. If $X$ is partially sorted with regard to length$(X)$, then $X$ is sorted.*

```
theorem LEMMA2 :
   all X : L(nat) .
     (partially_sorted(length(X),X)
      -> sorted(X) )
   proof
     let X : L(nat) be arbitrary
     assume
      partially_sorted(length(X),X)
      == length(X) =< length(X)
         -> all K,L:nat.
              (1=<K=<L=<length(X)
               -> elem(K,X)
                  =< elem(K,X) )
```

```
      length(X) =< length(X)
      hence
      all K,L:nat.
        (1=<K=<L=<length(X)
         -> elem(K,X) =< elem(K,X))
      == sorted(X)
    end_proof
end_theorem
```

Theorem 2 follows from Lemma 2 and the following proposition:

**Proposition::** Let $X$ be an arbitrary list and $J$ be an arbitrary natural number such that $J \leq length(X)$. Then there exists a list $Y$ which satisfies the following condition;
a) $Y$ is partially sorted with regard to $J$
b) for all natural numbers $I$, $K$ such that $1 \leq I \leq J$ and $J \leq K \leq length(X)$ , $elem(I,Y) \leq elem(K,Y)$

```
theorem PROPOSITION :
   all X : L(nat), all J : nat.
     (
      J =< length(X)
      -> exist Y : L(nat).
           (
            partially_sorted(J,Y)
            &
            all I : nat, all K : nat.
              (
               1 =< I =< J
               & J =< K =< length(X)
               -> elem(I,Y) =< elem(K,Y)
              )
           )
     )
   proof
     ..........
   end_proof
end_theorem
```

≪ Informal Proof of proposition ≫

We prove this proposition by mathematical induction on $J$. If $J = 0$, it is sufficient that $Y$ is $X$ itself. Now assume that $1 < J$ and there exists list $Y_0$ which satisfies conditions a) and b). We will show the existence of Y which satisfies the following conditions;
a') $Y$ is partially sorted with regard to $J + 1$
b') for any natural numbers $I$,$K$ such that $1 \leq I \leq J+1 \leq K \leq length(X)$, $elem(I,Y) \leq elem(K,Y)$
Applying **Lemma 1** to $Y_0$ with $J + 1$, we can get a list $Y_1$ such that

$$elem(J+1, Y_1) \leq elem(I, Y_1)$$

for any natural number $I$,s.t,$J+1 \leq I \leq length(X)$ and by the definition of $Y_1$ in the proof of **Lemma 1** we also get

$$elem(K, Y_1) = elem(K, Y_0)$$

for any natural number $K$,s.t.,$1 \leq K \leq J$. Now from the induction hypothesis we can easily show that $Y_1$ satisfies a') and b').

q.e.d.

### 2.2.2 Quick Sort Algorithm

(1) Formal Specification

This is the same as 2.2.1 (1) Definition1 and Theorem 2.

(2) Proof Strategy

The same example is also demonstrated in [Sato 85] where the specification is proved by transfinite induction. We sketch here the proof by ordinary mathematical induction in PDL/QJ style description. The reason we use mathematical induction here is that in '86 version of QJ([Sato 86]), the transfinite induction schema has been omitted because it can be simulated by *induction on recursive type structures*. We use the strategy which is essentially the same as that in [Smith 82] where the proof is written in the Martin-Löf type theoretic style.

≪Informal Proof of Theorem 2≫

We prove the following proposition that is the slightly modified version of Theorem 2.

*Let $X$ and $X_1$ be any natural number lists. If $length(X_1) \leq length(X)$, then there exists $Y_1$, a permutation of $X_1$, that is sorted.*

We prove this by mathematical induction on $length(X)$

[Base Case : $length(X) = 0$]

In this case $X_1$ must be *nil* and as a required list $Y_1$, $Y_1 \equiv nil$ will suffice.

[Induction Step]

Let $X$ be an arbitrary list with length $N$, and this satisfies the proposition. Now consider any list $Y$ with length $N+1$. By **Lemma 3** we can get two lists $Y_1$, and $Y_2$ such that ;

1. for any natural number $I$, if $1 \leq I \leq length(Y_1)$ then $elem(I, Y_1) < hd(Y)$
2. for any natural number $J$, if $1 \leq I \leq length(Y_1)$ then $hd(Y) \leq elem(J, Y_2)$
3. $append(Y_1, Y_2)$ is a permutation of $tl(Y)$

The lengths of $Y_1$ and $Y_2$ are both equal to or less than $length(X)$ so that by the induction hypothesis, there are sorted versions of $Y_1$ and $Y_2$, say $W_1$ and $W_2$ . Now let $Z$ be $append(W_1, hd(Y).W_2)$ , then $Z$ is a sorted version of $Y$ whose length is $length(X)+1$. Thus the proposition is proved for the case $length(X)+1$.

q.e.d.

Notes::

If we write the above proof in PDL/QJ, the most crucial or tedious part resides in the proof that *$append(W_1, hd(Y).W_2)$ is a sorted version of $Z$*. We prove this by the case splitting strategy of

Case1:: $1 \leq I \leq J \leq length(W_1)$

Case2:: $length(W_1) + 1 \leq I \leq J$
and $J \leq length(append(W_1, hd(Y).W_2))$
Case3:: $1 \leq I \leq length(W_1)$
and $length(W_1) + 1 \leq J \leq length(append(W_1, hd(Y).W_2))$ where function **append** is defined as follows;

```
function append : L(nat)#L(nat). -> nat
   attain
      append = fun [X,Y].
                  if X is nil then Y
                  else hd(X).append(tl(X),Y)
   existence
         ..............
   uniqueness
         ..............
end_function
```

Now we must prove the following ;

1. If $1 \leq I \leq length(W_1)$ then
   $elem(I, append(W_1, hd(Y).W_2)) = elem(I, W_1)$
2. If $length(W_1) + 2 \leq I \leq length(append(W_1, hd(Y).W_2))$
   then
   $elem(I, append(W_1, hd(Y).W_2)) =$
   $elem(I - length(W_1 + 1), W_2)$
3. $elem(length(W_1) + 1, append(W_1, hd(Y).W_2)) = hd(Y)$

These are quite clear for the human mathematician but we must *prove* them using induction and inference rules on `if-then-else` terms with which the **append** function is defined.

**Lemma 3** *[Divide Lemma] Let $X$ be any list of natural number elements, and let $a$ be an arbitrary natural number. There is a permutation of elements of $X$; $\sigma$ such that*

$$\sigma(X) = append(S(a), L(a))$$

*where every element of $S(a)$ is less than $a$ and every element of $L(a)$ is equal to or greater than $a$*

≪Informal Proof of Lemma 3≫
This can be proved by induction on the recursive structure of the list. ( See [Sato 85] )

q.e.d.

## 3 Compilation

The following is an outline of proof compilation. First the proof described in PDL/QJ will be parsed into an (incomplete) proof tree. Secondly, the proof checker will check

the proof tree filling in the skipped inference steps automatically or through interaction with the human programmer. Up to this step everything is essentially the same as CAP system([Sakai 86]). In the last step, a complete proof tree will be analyzed generating the executable code. In this paper, only this step is refered to as *proof compilation*.

The proof compilation algorithm is given below and as an example the proof compilation of the proof in 2.1 is traced roughly.

### 3.1 Compilation Algorithm

The proof compilation algorithm is given as a PROLOG style program.

#### 3.1.1 Data Structure

Here are the BNF style definitions.

*TYPE* ::= `nat` | `L(nat)` | *Others*

*TERM* ::= *Variable* (Typed Variable) | $\bot$ (*abort*) | `fun` *Argument_list* , *TERM* ( $\lambda$-expression ) | *TERM(TERM)* ( application ) | `if` *FORMULA* `then` *TERM* `else` *TERM* | *Others*

*FORMULA* ::= *FORMULA* $\wedge$ *FORMULA* | *FORMULA* $\vee$ *FORMULA* | *FORMULA* $\supset$ *FORMULA* | $\forall$*Variable*(: *TYPE*) .*FORMULA* | $\exists$*Variable*(: *TYPE*) .*FORMULA* | *TERM* $\leq$ *TERM* | *TERM* = *TERM* (equality) | *TERM* | *Others*

*PROOF_TREE* ::= `p_tree(` *RULE, UPPER_SEQUENCE, CONCLUSION* )

*UPPER_SEQUENCE* ::= `[]` | `[` *PROOF_TREE* | *UPPER_SEQUENCE* `]`

*RULE* ::= *FORMULA_RULE* | *TERM_RULE* | *Others*

*FORMULA_RULE* ::= $\forall$-*elm* | $\forall$-*int* | $\exists$-*elm* | $\exists$-*int* | $\wedge$-*elm* | $\wedge$-*int* | $\vee$-*elm* | $\vee$-*int* | $\supset$-*elm* | $\supset$-*int* | $\bot$-*elm* | `nat`-*induction* | `L(list)` -*induction* | *Others*

*TERM_RULE* ::= `if`-*elim* | `if`-$=_1$ | `if`-$=_2$ | *Others*

*CONCLUSION* ::= *FORMULA* | *TERM*

```
   P1, .. , Pn                ...   upper sequence
  --------------(Rule)
       C                      ...   conclusion
```

#### 3.1.2 Realizer and Realizing variable of expression

In the following X,Y,.. denote sequences of variables and x,y,.. denote variables.

**Definition::(length and realizing variable)**

**1:** Length of *TERM*, *TERM*=*TERM* and *TERM* $\leq$ *TERM* is 0

**2:** Length of $F \wedge G$ , where $F$ is of length $m$ and $G$ is of length $n$, is $m + n$

**3:** Length of $F \vee G$ , where $F$ is of length $m$ and $G$ is of length $n$, is $m + n + 1$

**4:** Length of $F \supset G$ , where $F$ is of length $m$ and $G$ is of length $n$, is $n$

**5:** Length of $\forall x.F(x)$, where $F$ is of length $n$, is $n$

**6:** Length of $\exists x.F(x)$, where $F$ is of length $n$, is $n + 1$

*Length* means the length of the *realizer* sequence of the expression. Let $F$ be a formula and $X$ be a sequence of variables, then we define the special abstraction $(qF)[X]$, read $X$ *realizes* $F$ or $X$ *is a constructive meaning of* $F$. $X$ is called a *realizing variable* into which realizer code will be bound.

**Definition::(q-realizer)**

**1:** If $F$ is a $TERM$, $TERM{=}TERM$ or $TERM{\leq}TERM$ , then $(qF)[] \equiv F$

**2:** If $F$, $G$ are formulas, then $(qF \wedge G)[X,Y] \equiv (qF)[X] \wedge (qG)[Y]$

**3:** If $F$, $G$ are formulas, then $(qF \vee G)[z,X,Y] \equiv (outl(z) \wedge F \wedge (qF)[X] \wedge Y) \vee (outr(z) \wedge X \wedge G \wedge (qG)[Y])$

**4:** If $F$, $G$ are formulas, then $(qF \supset G)[Y] \equiv Y \wedge mono(Y) \wedge \forall X.(F \wedge (qF)[X] \supset (qG)[Y(X)])$

**5:** If $F$, $G$ are formulas, then $(q\forall x.F)[Y] \equiv \forall x.((qF)[Y(x)])$

**6:** If $F$, $G$ are formulas, then $(q\exists x.F)[z,Y] \equiv z \wedge F\{x \leftarrow z\} \wedge (qF\{x \leftarrow z\})[Y]$

*mono()* means monotonic function and *outl(a)* and *outr(a)* are the canonical elements of disjoint sum type data structures. (See [Sato 85]) $F\{x \leftarrow z\}$ means substitution.

### 3.1.3 Basic procedure

The following procedures will be used in **3.1.4.**

*realizing_vars(Formula, Variable_list)*

Calculate the length of *Formula* and create a list of new variables (realizing variables) with the same length as *Formula*.

*and_elm_substr(And_form_1, And_form_2, Substr)*

*And_form_1* is a conjunction of formulas and *And_form_2* is also a conjunction each of component formulas occurs in *And_form_1*. This procedure returns a natural number list which indicates the substructure of *And_form_2* in*And_form_1*. If, for example, *And_form_1* is $P \wedge Q \wedge R$ and *And_form_2* is $P \wedge R$, structure of*And_form_1/2* is [1,2,3] and [1,3], respectively, and this procedure returns [1,3].

*or_intr_substr(Or_form, Formula, Substr)*

This procedure is almost the same as *and_elm_subst* . But imput formula *Or_form* is a disjunction of formulas and *Substr* is not a list, but the index integer of *Formula* in *Or_form*.

*and_elm_filter(Code_list_1, Substructure, Code_list_2)*

*Code_list_2* corresponds to *Substructure* under the assumption that the structure of *Code_list_1* is [1,2,...,n]. If, for example, *Code_list_1* is $[Code_1, Code_2, Code_3]$ and *Substructure* is [1,3], then this procedure returns as a value of *Code_list_2* $[Code_1, Code_3]$.

*par_extraction(Upper_sequence, Var_table, Code_list)*
This procedure extracts intermediate code of each proof tree in *Upper_sequence* , and combine them into a PROLOG-list *Code_list* . *Var_table* is is the form of [] or $[[Var_1, Formula_1], .. ,[Var_N, Formula_N]]$. When code has to be extracted from particular formulas listed in *Var_table* , the corresponding variables will be attached instead of the required code .

*discharged(Rule, Upper_sequence, Dsch_form_list)*
This procedure extracts the discharged formulas with regard to *Rule* in *Upper_sequence*.

*srch_key(Item, Table, Key)*
*Table* is in the form of $[[Key_1,Item_1],..,[Key_N,Item_N]]$. This procedure searches a *Key* in *Table* whose corresponding item is *Item*. If there is no such key, it fails.

*attach_vars( Items_list, VarItem_table)*
This procedure attaches a newly generated variable for each item in *Items_list*, and returns the correspondence table of the form $[[Var_1, Item_1], .. ,[Var_N, Item_N]]$ as *VarItem_table*.

*change_items(VarItem_table, NewItems_list, New_table)*
Items in *VarItem_table* will be exchanged for those in *NewItems_list*. If, for example, *VarTem_table* is [[X,a],[Y,b],[Z,c]] and *NewItems_list* is [1,2,3] then *New_table* will be [[X,1],[Y,2],[Z,3]].

*substitute(Code_list1, VarCode_table, Code_list2)*
Corresponding code will be substituted, into every variable in the formulas in *Code_list1* listed in *VarCode_table*.

### 3.1.4 Algorithm

The idea is that the proof compiler will analyze a given proof tree from bottom to top extracting code step by step for the inference rule attached to each node of the proof tree. A similar algorithm which generates LISP code is given in [Hayashi 86].

Our compiler is a 2-pass compiler. In the first pass, intermediate code defined as follow is generated, and in the second pass, that intermediate code will be translated into PROLOG code.

*INTERMEDIATE_CODE* ::=
`$and$(` *INTERMEDIATE_CODE* `)`
| `$or$(` *CONDITION, INTERMEDIATE_CODELIST* `)`
| `$apply$(` *TERM, TERM* `)`
| `$fun$(`*ARGUMENT_LIST,*
*INTERMEDIATE_CODELIST* `)`
| `$case$(`*INDEX, INTERMEDIATE_CODELIST* `)`
| `$dummy_code$`
| `$rec_fun$([$zero$,`*INTERMEDIATE_CODELIST* `],`
`[$nat$,`*INTERMEDIATE_CODELIST,*
*VARIABLE*`])`
| `$rec_fun$([$nil$,`*INTERMEDIATE_CODELIST* `],`

```
        [$lnat$, INTERMEDIATE_CODELIST,
                VARIABLE])
    | $null$


≪ TopLevel ≫
compile( PROOF_TREE, PROLOG_CODE ) :-
    1st_pass([], PROOF_TREE, INTERMEDIATE_CODE ),
    2st_pass( INTERMEDIATE_CODE, PROLOG_CODE).

≪ dischargedformula ≫
1st_pass(VarForumula_table,p_tree(_,_,Formula),
          Variable ) :-
    srch_key(Formula, VarForumua_table, Variable ).

≪ ∀ − elm ≫
1st_pass(A, p_tree(∀-elm ,[_,p_tree(R,Us,∀x.A[x])],
                  A[Term] ),
          $apply$(Code, Term) ) :-
    1st_pass(A, p_tree(R,Us, ∀x. A[x] ), Code ).

≪ ∀ − int ≫
1st_pass(A, p_tree(∀-int, [P_TREE],_),
          $fun$( [Variable], Code ) ) :-
    discharged( ∀-int, [P_TREE],[ Variable:TYPE ] ),
    1st_pass(A, P_TREE, Code).


≪ ∃ − elm ≫
1st_pass(A, p_tree(∃-elm, [p_tree(R1,U1,∃x.A[x] ),
                  P_TREE],
                  CONCLUSION),
                  Code3 ) :-
    1st_pass(A, p_tree(R1,U1,∃x.A[x]),
          $exist$(Code1, Code2)),
    realizing_vars(∃x.A[x],
          List_of_realizing_vars),
    discharged(∃-elm,
          P_TREE,
          [Term:TYPE, A[Term]),
    1st_pass([[List_of_realizing_vars,
          [Term:Type, A[Term]]]|A],
          P_TREE, Code4),
    substitute(Code4,
          [[List_of_realizing_vars,
          [Code1,Code2]|A],
          Code3).
```

```
≪ ∃ − int ≫
1st_pass(A,p_tree(∃-int
          , [p_tree(_,_,TERM), P_TREE],∃x.A[x] ),
          $exist$(TERM,Code) ) :-
   1st_pass(A,P_TREE, Code).

≪ ∧ − elm ≫
1st_pass(A, p_tree(∧-elm , UPPER_SEQUENCE,
                  CONCLUSION),
          Code ) :-
   1st_pass(A, UPPER_SEQUENCE, $and$(Code_list)),
   and_elm_substr(UPPER_SEQUENCE, CONCLUSION,
                  Substr ),
   and_elm_filer(Code_list, Substr, Code_list1),
   (Code_list1 = [X|Y], Code = $and$(Code_list1)
   ; Code_list1 = [X] , Code = X
   ; Code_list1 = [], Code = $null$ ).

≪ ∧ − int ≫
1st_pass(A, p_tree(∀-int , UPPER_SEQUENCE,_),
          $and$(Code_list) ) :-
   par_extraction(UPPER_SEQUENCE, A,
                  $and$(Code_list) ).


≪ ∨ − elm ≫
1st_pass(A,  p_tree(∨-elm ,[OR_PART|UPPER_SEQUENCE],
                  CONCLUSION),
          $case$(Or_index, Code_list3 ) :-
   discharged(∨-elm , UPPER_SEQUENCE,
                  Discharged_formula_list),
   1st_pass(A, OR_PART, $or$(Or_index,Code_list1) ),
   attach_vars( Discharged_formula_list,
                  VarFormula_table ),
   change_items( VarFormula_table, Code_list1,
                  VarCode_table ),
   append(VarFormula_table, A, Table),
   par_extraction(UPPER_SEQUENCE,Table,
                  Code_list2),
   substitute(Code_list2,VarCode_table,Code_list3).


≪ ∨ − int ≫
1st_pass(A,p_tree(∨-int
                  ,[p_tree(RULE,UP_SEQ,CONC)],
                  CONCLUSION),
          $or$( Index, Code_list) :-
   or_intr_substr( CONCLUSION, CONC, Index ),
```

```
    1st_pass(A,p_tree(RULE, UP_SEQ, CONC), Code_list).

≪⊃ −elm ≫
1st_pass(A,p_tree(⊃-elm , [P_TREE,IMP_P_TREE],
                  CONCLUSION ),
          $apply$( Code_1, Code_2 ) :-
    1st_pass(A,P_TREE, Code_1 ),
    1st_pass(A,IMP_P_TREE, Code_2).


≪⊃ −int ≫
1st_pass(A,p_tree(⊃-int ,[P_TREE],Imp_formula),
          $fun$( Variable_list, Code)):-
    discharged(⊃-int,
          [P_TREE], [Discharged_formula] ),
    realizing_var(Discharged_formula,Variable_list),
    (Variable_list =[], X = $null$
    X = Variable_list
    1st_pass([[Discharged_formula,X]|A],
          P_TREE,Code).

≪ ⊥ − elm ≫
1st_pass(_, p_tree(⊥-elm , [P_TREE], CONCLUSION),
                  $dummy_code$ ).


≪ nat-induction ≫
1st_pass(A,p_tree( nat-induction,
                  [P_TREE_1,P_TREE_2], ∀x :nat.A[x]),
          $rec_fun$([$zero$, Code1],
                  [$nat$, Code2, New_var])) :-
    1st_pass(A, P_TREE_1, Code1),
    discharged( nat-induction,
          [P_TREE_2],
          [Induction_hypothesis])
    1st_pass([[New_var, Induction_hypothesis]|A],
                  P_TREE_2, Code2).

≪ L(nat)-induction ≫
1st_pass(A, p_tree( L(nat)-induction,
                  [P_TREE_1, P_TREE_2],
                  ∀x :L(nat).A[x]),
          $rec_fun$([$nil$, Code1],
          [$Lnat$, Code2, New_var] ) ):-
    1st_pass(A, P_TREE_1, Code1 ),
    discharged( L(nat)-induction,
                  [P_TREE_2],
                  [Induction_hypothesis]),
```

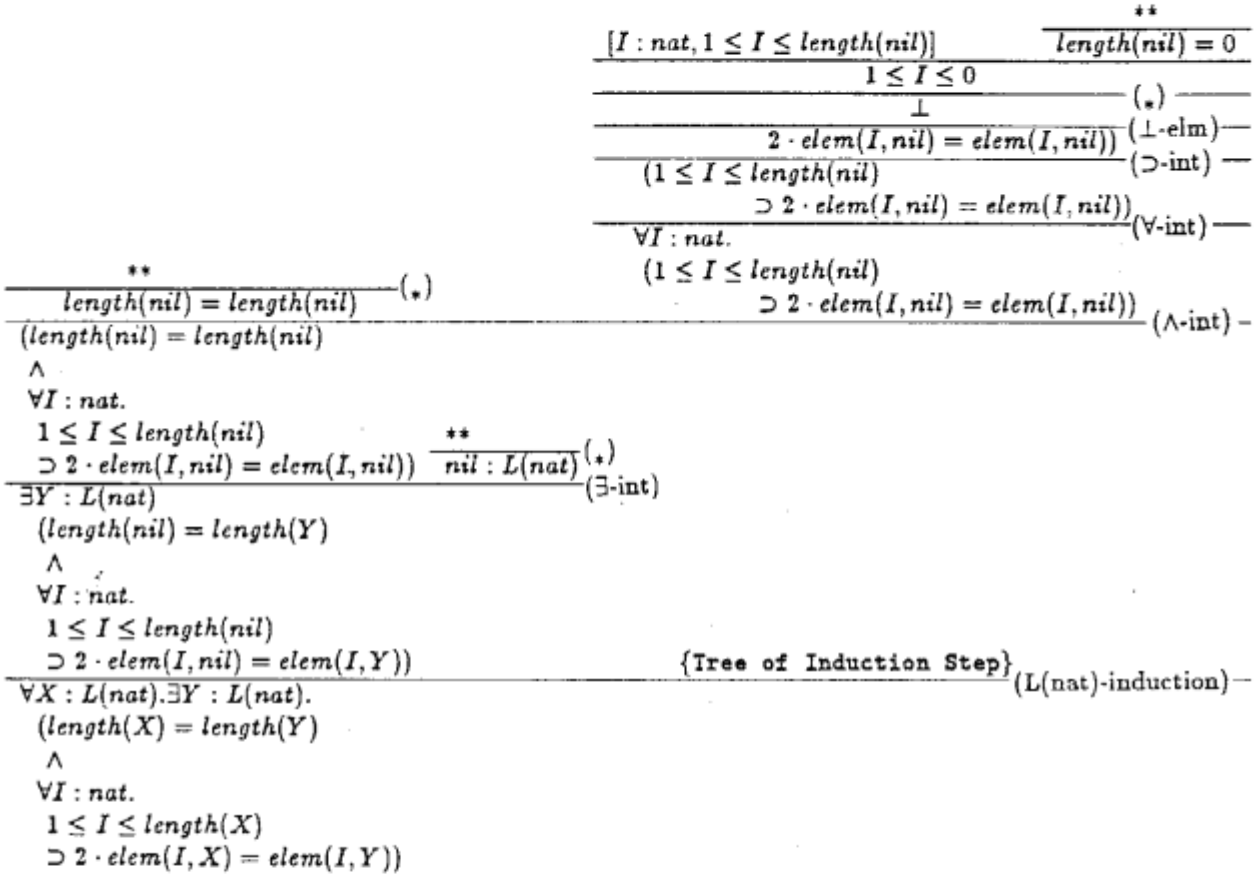


Figure 1: Example Proof Tree

```
1st_pass([[New_var,Induction_hypothesis]|A],
                P_TREE_2, Code2).

≪ OtherCases ≫
1st_pass(_,p_tree(_,_,_),$null$).
```

### 3.2 Example

(1) Proof Tree

**Figure 1-4** show the proof tree corresponding to the proof in 2.1. As we have shown in 4.1, the inference with term rule will be skipped in the proof compilation algorithm. Thus we omit the subtrees of term rule inference abbreviating them with (*) and ** .

(2) Compiling Procedure

1) The bottom of the example proof tree is an inference by L(nat)-induction rule. So that the extracted intermediate code is ;

$$\dfrac{\dfrac{\dfrac{\dfrac{[t : L(nat)\ ,\ length(X) = length(t)\ \wedge\ \forall I : nat.(1 \le I \le length(X) \supset 2 \cdot elem(I, X) = elem(I, t))]}{length(X) = length(t)}(\wedge\text{-elm})}{length(X) + 1 = length(t) + 1}(*)}{length(a.X) = length(2 \cdot a.t)}(*) \qquad \dfrac{\dfrac{\dfrac{\dfrac{[I : nat, 1 \le I length(a.X)]}{I = 1 \vee (2 \le I \wedge I < length(a.X))}(\vee\text{-int}) \quad \{Tree1\} \quad \{Tree2\}}{2 \cdot elem(I, a.X) = elem(I, 2 \cdot a.t)}}{1 \le I \le length(a.X) \supset 2 \cdot elem(I, a.X) = elem(I, 2 \cdot a.t)}(\supset\text{-int})}{\forall I : nat.(1 \le I \le length(a.X) \supset 2 \cdot elem(I, a.X) = elem(I, 2 \cdot a.t))}(\forall\text{-int})}{length(a.X) = length(2 \cdot a.t) \wedge \forall I : nat.(1 \le I \le length(a.X) \supset 2 \cdot elem(I, a.X) = elem(I, 2 \cdot a.t))}(\wedge\text{-int})$$

$$\dfrac{2 \cdot a.t : L(nat) \qquad [\exists Y_1 : L(nat).\ (length(X) = length(Y_1) \wedge \forall I : nat.(1 \le I \le length(X) \supset 2 \cdot elem(I, X) = elem(I, Y_1)))]}{\exists Y : L(nat).\ (length(a.X) = length(Y) \wedge \forall I : nat.(1 \le I \le length(a.X) \supset 2 \cdot elem(I, a.X) = elem(I, Y)))}(\exists\text{-elm}) \qquad \dfrac{\exists Y : L(nat).\ (length(a.X) = length(Y_1) \wedge \forall I : nat.(1 \le I \le length(a.X) \supset 2 \cdot elem(I, a.X) = elem(I, Y_1)))}{}(\exists\text{-int})$$

Figure 2: Tree of Induction Step

$$\dfrac{\dfrac{[I = 1]}{2 \cdot elem(I, a.X) = 2 \cdot a}(*) \qquad \dfrac{[I = 1]}{elem(I, 2 \cdot a.X) = 2 \cdot a}(*)}{2 \cdot elem(I, a.X) = elem(I, 2 \cdot a.X)}(*)$$

Figure 3: Tree1

$$\dfrac{\dfrac{\dfrac{[t : L(nat),\ length(X) = length(t) \wedge \forall I : nat\ (1 \le I \wedge I \le length(X) \supset 2 \cdot elem(I,X) = elem(I,t))]}{\forall I : nat.\ (1 \le I \wedge I \le length(X) \supset 2 \cdot elem(I,X) = elem(I,t))}(\wedge\text{-elm}) \quad \dfrac{[2 \le I]}{I - 1 : nat}(*)}{1 \le I - 1 \wedge I - 1 \le length(X) \supset 2 \cdot elem(I-1,X) = elem(I-1,t)}(\forall\text{-elm})$$

$$\dfrac{\dfrac{[2 \le I \wedge I \le length(a.X)]}{2 \cdot elem(I-1,X) = elem(I-1,t)}(\supset\text{-elm}) \quad \dfrac{\dfrac{[2 \le I \vee I \le length(X)]}{2 \le I}(\wedge\text{-elm})}{2 \cdot elem(I,a.X) = 2 \cdot elem(I-1,X)}(*)}{2 \cdot elem(I,a.X) = elem(I-1,t)}(*) \quad \dfrac{\dfrac{[2 \le I \wedge I \le length(a.X)]}{2 \le I}(\wedge\text{-elm})}{elem(I,2 \cdot a.t) = elem(I-1,t)}(*)$$

$$2 \cdot elem(I,a.X) = elem(I,2 \cdot a.t)\ (*)$$

Figure 4: Tree2

```
$rec_fun$([$nil$, Code1],
          [$Lnat$,Code2,New_var])
```

Code1 and Code2 are the extracted intermedicate codes from the tree in **Figure 1** of the base case and the tree in **Figure 2** of the induction case, respectively.

2) Code1 can be extracted by analysing through $(\exists - int)$, $(\wedge - int)$, $(*)$( Some inference rule on term ), $(\forall - int)$, $(\supset - int)$, $(\bot - elm)$ inferences from the bottom to the top of the tree, and we actually get the following;

```
$exist$( nil,
       $and$([$null$,
            $fun$([I],$fun$([],$dummy_code$))]))
```

3) Code2 can be extracted by analysing through $(\exists - elm)$, $(\exists - int)$, $(\wedge - int)$, $(*)$, $(\forall - int)$, $(\supset - int)$, $(*)$. In this procedure, a variable sequence *Rel_var_seq* will be attached instead of calculating the realizer of the induction hypothsis. We actually get the following intermediate code;

```
$exist$( 2*a.New_var,
       $and$([$null$,$fun$([I],
                $fun$([],
                $case$(Or_index,
                [$null$,$null$]))])))
```

4) From the above procedure and some suitable rearrangement, we can get the following intermedicate code ;

```
$rec_fun$([ $nil$, nil],
          [$Lnat$, 2*a.New_var, New_var] )
```

and from this we can easily get the following PROLOG code;

```
new_pred( nil, nil).
new_pred( a.X, 2*a.New_var ) :-
          new_pred(X, New_var).
```

## 4 Subsequent research

### 4.1 Optimization

There are two kinds of optimization. One is the local optimization technology adopted in most of the commercial compiler systems. The other is global or logic level optimization. If we can write programs in very high-level programming languages or at the natural language level, trivial parts of the algorithm will be abstracted and we can concentrate on the essential part of algorithm. That will help us produce efficient algorithms. We have shown two extreme proof examples of list sort specification. This case study shows how different proof strategies correspond to different algorithms. But more case studies are required to think about the problem of global level optimization.

As a systematic method for proof optimization, which will be categorized as the intermediate level optimization between local and global types, the proof normalization method([Prawitz 65]) may be effective. This is a topic for subsequent research.

### 4.2 Proof Checker

As mentioned in Section 2, a programming paradigm based on QJ entails an intelligent compiler which has both a code generating facility and a proof checking facility. As for the proof checking technique see [Sakai 87].


### Acknowledgments

The author thanks Mr.Sakai of ICOT 2nd Laboratory who was kind enough to read early drafts of this and to make necessary repairs to the example codings in PDL/QJ. Special thanks to Dr.Sato of Tohoku University, Dr.Hayashi of Kyoto University and many other members of CAP working group at ICOT for many useful suggestions.


### References


[Aho 74] Aho,A.V.,Hopcroft,J.E. and Ullman,J.D., "*The Design and Analysys of Computer Algorithms* ", Addison-Wesley, 1974.

[Beeson 83] Beeson,M.,"Proving programs and programming proofs", In *International Congress on Logic,Methodology, and Philosophy of Science* , Salzburg, Austria, North-Holland, Amsterdam, 1983.

[Beeson 85] Beeson,M.,"*Foundation of Constructive Mathematics* ", Springer, 1985.

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

[de Brujin 80] de Brujin,N.D.,"A survey of the project AUTOMATH", in *Essays in Combinatory Logic, Lambda Calculus and Formalism* , pp.589-606, Academic Press, New York, 1980.

[Hirose 86] Hirose,K.,"An approach to proof checker", *Lecture Notes on Computer Science 233* , Springer 1986.

[Hayashi 86] Hayashi,S.,"PX:a system extracting programs from proofs", *3rd Working Conference on the Formal Description of Programming Concepts* , Ebberup, Denmark, 1986.

[Kleene 45] Kleene,S.C.,"On the interpretation of intuitionistic number theory", *Journal of Symbolic Logic. Volume 10* . pp.109-124, 1945.

[Martin-Löf 82] Martin-Löf,P.,"Constructive mathematics and computer programm- ing", in *Logic, Methodology, and Philosophy of Science VI* , North-Holland, pp.153-179, 1982.

[McCarty 84] McCarty,D.C,."*Realizability and Recursive Mathematics* ", CMU-CS-84 -131, Carnegie-Mellon University, 1984.

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

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

[Sakai 87] Sakai,K.and Takayama.Y.,"QJ proof checker",(to appear)

[Sato 84] Sato,M and T,Sakurai,"Qute: A functional Language based on Unification", *Proceeding of the International Conference on Fifth Generation Computer System* , pp157-165, 1984.

[Sato 85] Sato,M.,"*Typed logical calculus*", TR-85-13, Department of Computer Science, University of Tokyo, 1986.

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

[Smith 82] Smith,J.,"The identification of propositions and types in Martin- Löf's type theory: a programming example", *Lecture Notes in Computer Science 158* , Springer, 1982.

[Uchida 83] S.Uchida, M.Yokota, A.Yamamoto, K.Taki and H.Nishikawa,"Outline of the Personal Sequential Inference Machine: PSI", *New Generation Computing Vol.1, No.1* , Ohmsha and Springer-Verlag, 1983.

Appendix : Inference rules of QJ

These are the inference rules of our logic, a slightly modified subset of original QJ([Sato 85]).

$$\ll \exists - elm \gg \quad \frac{\exists X : Type.A[X] \quad X : Type, A(X) \vdash C}{C}$$

$$\ll \exists - int \gg \quad \frac{Term : Type \qquad\qquad A(Term)}{\exists X.A(X)}$$

$$\ll \wedge - int \gg \quad \frac{F_1, .., F_n}{F_1 \wedge .. \wedge F_n}$$

$$\ll \wedge - elm \gg \quad \frac{F_1 \wedge .. \wedge F_m}{F_1 \wedge .. \wedge F_n}$$

where $n < m$

$$\ll \vee - elm \gg \quad \frac{F_1 \vee .. \vee F_n \quad F_1 \vdash C, \quad .., \quad F_n \vdash C}{C}$$

( We call $F_1 \vee .. \vee F_n$ OR_PART )

$$\ll \vee - int \gg \quad \frac{F_i}{F_1 \vee .. \vee F_i \vee .. \vee F_n}$$

$$\ll \supset - elm \gg \quad \frac{A \quad A \supset B}{B}$$

$$\ll \supset - int \gg \quad \frac{A \vdash B}{A \supset B}$$

$$\ll \bot - elm \gg \quad \frac{\bot}{F}$$

$$\ll \texttt{nat}\text{-}induction \gg \quad \frac{A[0] \qquad A[X] \vdash A[X+1]}{\forall x : nat.A[x]}$$

$$\ll \texttt{L(nat)}\text{-}induction \gg \quad \frac{A[nil] \qquad A[X] \vdash A[a..X]}{\forall x : L(nat).A[x]}$$

$$\ll if - elm \gg \quad \frac{\textbf{if } A \textbf{ then } B \textbf{ else } C \quad A, B \vdash F \quad \neg A, C \vdash F}{F}$$

$$\ll if - =_1 \gg \quad \frac{A}{\textbf{if } A \textbf{ then } B \textbf{ else } C \simeq B}$$

( $\simeq$ is Kleene equality )

$$\ll if - =_2 \gg \quad \frac{B}{\textbf{if } A \textbf{ then } B \textbf{ else } C \simeq C}$$

Figure 5: Inference Rule