TR-417

A Programming System Based on QJ

by
Y. Takayama

July, 1988

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

A Programming System Based on QJ

Yukihide Takayama

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

This paper presents a programming environment which is based on a constructive logic, QJ. This programming environment provides uniformity to work on problem solving, document preparation, and automatic coding. Simple programming developments are presented to demonstrate the mathematical programming in the environment. The programming logic behind the system and the automatic coding algorithm based on q-realisability are given. Finally, techniques to improve the efficiency of the program are investigated.

1. Introduction

The current improvement of programming and document preparation environment on computers is remarkable. However, there are extremely few satisfactory computer system that allow a computer scientist to solve problems, write mathematical papers, and implement the algorithms which he or she has developed all in uniform way and in uniform environment. Most people communicate with other researchers by distributing papers, documents or reports, not by distributing source programs even if they are written in high level programming languages. Papers can be prepared on computers, distributed through computer networks, and used as good materials to convey the contents of research achievement. Computers are becoming more important as a medium for computer scientists in this respect. However, computers do not assure the correctness of the contents of one's papers, so that proof reading takes a lot of time. When one implements the algorithm which is presented in a paper, one has to use a sort of programming system. However, no matter how good the programming environment is, one has to translate the definition of the algorithm given in that paper into a suitable programming language manually as long as the programming environment is separated from the document preparation environment. This is not efficient. Also, maintaining the consistency between the algorithms presented in the paper and the programs which implement the algorithms is difficult in most cases. This can be seen as one of the reasons why programs tend to be distributed separately even if they are describing the same mathematical contents. In other words, it is one of the bottle necks to making computers a better medium among computer scientists.

The aim of this research is to develop an environment on computers in which mathematics is mechanised to help its correct development, good document processing facilities are available, and implementation of the algorithms can be performed automatically when one finishes the preparation of the documents. Suitable programming logic, in which formal specification, reasoning about correctness and execution of programs can be described, is indispensable in realising this sort of environment. The constructive logic or constructive theory of types is thought to be a good candidate for programming logic [Beeson 83] [Martin-Löf 82] [Nordström 83a] [Smith 82] [Bates 79] [Huet 86] [Huet 88]. The chief reason for this is that, in intuitionistic

logic, logical constants are interpreted as sorts of operation [Martin-Löf 85], so that computation and reasoning can be described in uniform way.

Constructive logic is a research theme in itself. Various formulations of the logic have been developed [Beeson 85] [Coquand 86] [Feferman 78] [Martin-Löf 84], and the evaluation of the formulations is performed both from the theoretical and practical points of view. The work presented in this paper is an implementation of a constructive logic, QJ [Sato 85] [Sato 86a] [Sato 86b] [Sato 87], on a proof assistant system dedicated to linear algebra, CAP-LA [Hirose 86] [Sakai 86], and the first step of the trial of evaluating the formulation of the logic mainly from the practical point of view. This paper also works on the program extraction technique: a technique to link documents and programs. It is well known that executable codes can be extracted from constructive proofs by using the notion of realisability [Kleene 45] [Beeson 85] [McCarty 84], Gödel interpretation [Goto 79], or proof normalization [Goad 80]. The method of the extraction by using realisability is demonstrated in detail in [Bates 79] and [Hayashi 87]. However, the technique to extract efficient executable codes has not yet been fully investigated although there have been a few notable works [Goad 80] [Sasaki 86] [Mohring 88]. This paper also looks at this theme.

This paper is structured as follows. Section 2 demonstrates the mathematical programming where algorithms are described in the theorem proving style. Here, it is considered to be the description style of documents to be distributed, and it makes it clear that our system is not a coding support system, but an environment where mathematicians and designers of algorithms work. The document description language, PDL-QJ, is introduced in this section. Some problems in realizing this programming paradigm are also discussed. Section 3 presents the programming logic, QPC logic, of our system. The logic is a sugared version of a subset of QJ. The proof compilation algorithm, an algorithmic version of realisability interpretation of QJ, is also given here. Section 4 discusses the problems of the proof compilation algorithm in Section 3 from the point of view of runtime efficiency of extracted codes. The proof normalization technique, the modified $\vee$ code, and the extended projection method are presented to improve the efficiency of the extracted codes. Section 5 is the conclusion.

2. Mathematical Programming

This section overviews the image of the environment, which is a combination of document processing facilities and programming facilities [Takayama 87a].

2.1 Programming as Mathematical Documentation

Programs can be written as proofs of specification in the form of

$$\forall x_1.\cdots\forall x_m.\exists y_1.\cdots\exists, y_n.A(x_1,\cdots,x_m,y_1,\cdots,y_n)$$

where $x_1,\cdots,x_m$ and $y_1,\cdots y_n$ are input and output variables, and $A(x_1,\cdots,x_m,y_1,\cdots,y_n)$ is the logical description of the input-output relation [Nodström 83a] [Nodström 83b] [Smith 82] [Bates 85]. This is based on the following analogy between mathematics and programming.

Mathematics	Programming
Theorem	Problem or specification
Proof	Program + verification information
Proof strategy	Algorithm
Reference to other theorems	Subroutine call
Normalization	Execution
Definition	Definition of functions or predicates

Here, a few simple examples are shown in the ideal way as documentation to be distributed among computer scientists, i.e., in natural language, to demonstrate how the documentation of algorithms is described.

Theorem A: *For any natural number list, X, there exists a natural number list, Y, such that;*
a) the length of X is equal to the 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.

Proof: The proof proceeds by induction on the structure of list X.
1) Base step: Assume that $X = nil$, then $Y = nil$ is the list satisfying conditions a) and b).
2) Induction step: Suppose that for list X, list Y, satisfying conditions a) and b), has been obtained. Then a list, Ls, must be constructed from Y satisfying the following conditions. For an arbitrary natural number, a
a') the 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 $1 \leq N \leq length(a.X)$.
$Ls = 2a.Y$. Conditions a') and b') are checked.
a') $length(a.X) = length(X) + 1$, $length(2a.Y) = length(Y) + 1$ and because of condition a) for X and Y, a conclusion is obtained;
b') Here, $elem(I, X)$ denotes the I-th element of list X. Now, let $1 \leq N \leq length(X)$, then $elem(N+1, Ls) = elem(N+1, 2a.Y) = elem(N, Y)$ and because of condition b) for X and Y, $elem(N, Y) = 2 \cdot elem(N, X)$, so that $elem(N+1, Ls) = 2 \cdot elem(N, X)$. Now let $M = N + 1$, then $elem(M, Ls) = 2 \cdot elem(M, a.X)$ because $2 \leq M \leq length(X) + 1 = length(a.X)$, so that $elem(M, a.X) = elem(N, X)$. In addition, $elem(1, Ls) = 2 \cdot a = 2 \cdot elem(1, a.X)$, so that $elem(N, Ls) = 2 \cdot elem(N, a.X)$ where $1 \leq N \leq length(a.X)$. ∎

It is easy to see that the proof describes a program which takes a natural number list as input and doubles each element of the list, and it also gives the justification of the algorithm. Note that the chief proof strategy, induction of the list structure, corresponds to the recursive call algorithm on list structure.

The next example, the sorting problem, is well known.

Definition B: Let X be a list of length N. Then,
X is *sorted* $\stackrel{def}{=}$ For any natural numbers, I and J,
if $1 \leq I \leq J \leq N$ then (I-th element of X) $\leq$ (J-th element of J)

Proposition C: *Let X be an arbitrary list of length N, and J be an arbitrary natural number such that $J \leq N$. Then, there exists a list, Y, satisfying the following conditions:*
a) The length of Y is equal to the length of X, and every element occurring in X also occurs in

Y *(permutation condition);*
b) If $J \neq 0$, *for every natural number,* I, *such that* $J \leq I \leq N$

$$(J\text{-}th\ element\ of\ Y) \leq (I\text{-}th\ element\ of\ Y)$$

Proof: 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 mathematical induction is used for* M. *If* $M = 0$, *it is sufficient that* Y *is just* X *itself. Now assume that* $1 \leq M \leq N$ *and 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* $J \leq I \leq N$, $elem(J, Y_0) \leq elem(I, Y_0)$. *Assume also that* $M + 1 < N$, *otherwise there is nothing to prove. Now consider the case when* $M + 1$ *and define a list,* Y, *as follows: if* $elem(J-1, Y_0) \leq elem(J, Y_0)$, *then let* $Y \equiv Y_0$. *Otherwise, i.e,* $elem(J, Y_0) < elem(J-1, Y_0)$, Y *is defined 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, it can be checked easily that this Y *satisfies the following:*
a') Y is a permutation of X;
b') For all natural number, I, such that $J - 1 \leq I \leq N$ $elem(J-1, Y) \leq elem(I, Y)$. ∎

Definition D: Let $X \equiv a_1 \ldots . a_N.nil$ be a list of length N and J be a natural number. Then, X *is partially sorted with regard to* $J \stackrel{\text{def}}{=}$ If $J \leq N$ then list $a_1 \ldots . a_J.nil$ is sorted.

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

Proof: Trivial ∎

Proposition F: *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 conditions:*
a) Permutation condition with regard to X;
b) Y is partially sorted with regard to J;
c) For all natural numbers, I *and* K, *such that* $1 \leq I \leq J$ *and* $J \leq K \leq length(X)$, $elem(I, Y) \leq elem(K, Y)$.

Proof: It is proved by mathematical induction on J. *If* $J = 0$, *it is sufficient that* Y *is* X *itself. Now assume that* $0 < J$ *and there exists list* Y_0 *which satisfies conditions a), b) and c). The existence of Y, which satisfies the following conditions, is shown.*
a') Permutation condition with regard to Y_0;
b') Y is partially sorted with regard to $J + 1$;
c') For any natural numbers, I *and* K, *such that* $1 \leq I \leq J+1 \leq K \leq length(X)$, $elem(I, Y) \leq elem(K, Y)$.
Applying Proposition C to Y_0 *with* $J + 1$, *a list,* Y_1, *can be obtained 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 Proposition C*

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

is obtained for any natural number, K,s.t.,$1 \leq K \leq J$. Now from the induction hypothesis, it can be easily shown that Y_1 is such a Y. ∎

Theorem G: Sorting problem
For any natural number list, X, there exists a natural number list Y such that
a) Y satisfies the permutation condition;
b) Y is sorted.

Proof: Straightforward from Proposition E and F. ∎

The above proof of Theorem G corresponds to the bubble sort algorithm. It is also possible to write a proof which corresponds to the quick sort algorithm. The same example is also demonstrated in [Sato 85] where the specification is proved by transfinite induction, [Smith 82] where the proof is written in Martin-Löf's theory of types, and in [Mohring 86] where the proof is written in the calculus of constructions [Coquand 86]. The proof by ordinary mathematical induction is given here.

Proposition H: 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, σ, 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.

Proof: This can be proved by induction on the recursive structure of the list. (See [Sato 85].) ∎

Another proof of Theorem G: *The following proposition, a slightly modified version of Theorem G, is proved here.*
"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."
This is proved by mathematical induction on $length(X)$.
Base case ($length(X) = 0$): X_1 must be nil and $Y_1 \equiv nil$.
Induction step: Let X be an arbitrary list with length N, and this satisfies the proposition. Now consider any list, Y, with length $N + 1$.
With the divide lemma, two lists, Y_1, and Y_2, can be obtained 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 $1 \leq J \leq length(Y_2)$ 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 less than or equal to $length(X)$ so that by the induction hypothesis, there are sorted versions of Y_1 and Y_2, say W_1 and W_2. Now let $Z \equiv append(W_1, hd(Y).W_2)$, then Z is a sorted version of Y whose length is $length(X) + 1$. Thus, the proposition is proved for the case $length(X) + 1$. ∎

As can be seen from the three examples illustrated so far, constructive proofs of formal specifications have very similar structures to algorithms which realise the specifications. More precisely, a constructive proof of a specification can be seen as the description of an algorithm plus its mathematical meaning, in other words, justification of the algorithm.

2.2 Documentation Language: PDL-QJ

It is necessary to use a suitable description language to mechanise the handling of the mathematical documents as demonstrated in the previous section. PDL-QJ is the language in our system. A proof description language, PDL, has been designed and implemented in the CAP-LA project, which is developing a proof assistant system dedicated to linear algebra [Hirose 86] [Sakai 86]. The design goal of the language is similar to that of AUTOMATH [de Bruijn 80] or the Prl [Constable 86]; however, the approach is a little different. The basic idea of AUTOMATH and the Prl is to describe mathematics in typed lambda calculus by using the notion of formulae-as-types [Howard 80]. On the other hand, the design of PDL started by investigating quite a number of proof descriptions in ordinary textbooks of mathematics to find a good syntax to mechanise mathematics on computers. Therefore, PDL allows, in a way, a more natural proof description style than other languages in the type theoretic approach. The structure of the syntax is basically natural deduction [Prawitz 65] with additional expressions to describe mathematical objects such as natural numbers, scalars, matrices, indexed sums and products, and vectors. PDL-QJ is one of the PDL families. The chief purpose of the language is to describe algorithms on natural numbers and lists. It is an intuitionistic version of PDL with additional syntax for program constructs, such as if-then-else and lambda expressions.

The following is part of the basic syntax of PDL-QJ.

(0) Expressions

- `fun [X1,...,Xn]` ··· function with parameter X
- `if then else` ··· if-then-else construct
- `mu [Z1,...,Zn]` ··· fixed point operator
- Type expressions:

`nat` ··· natural number type
`L(nat)` ··· natural number list type
<TYPE> # <TYPE> ··· cartesian product type
<TYPE> --> <TYPE> ··· function type

- Logical constants: `&` ($\wedge$), `|` ($\vee$), `->` ($\supset$), `all` ($\forall$), and `some` ($\exists$) (negation)
- `=` ··· equality symbol
- <LABEL> : <FORMULA>¿··· labeled formula
- `$abort$` ··· abort

(1) Definition of functions

```
function   <FUNCTION NAME> :  <TYPE>
attain   <DEFINITION>
end_function
```

(2) Reasoning on logical constants

($\wedge$-I rule)	($\wedge$-E rule)	($\vee$-I rule)
.	.	.
.	.	.
.	.	.

```
A
.
.
.
B
hence A & B
```

```
A & B
hence A [B]
```

```
A [B]
hence A | B
```

(∨-E rule)

```
.
.
.
A | B
hence C
since divide and conquer A | B
case A
.
.
C
case B
.
.
C
end_since
```

(⊃-I rule)

```
A -> B
since
assume A
.
.
B
end_since
```

(⊃-E rule)

```
.
.
.
A -> B
.
.
.
A
hence B
```

(∀-I rule)

```
all x:t.  A(x)
since
let a:t be arbitrary
.
.
A(a)
end_since
```

(∀-E rule)

```
.
.
.
all x:t.  A(x)
.
.
.
a:t
hence A(a)
```

(∃-I rule)0

```
.
.
.
A(a)
.
.
.
some x:t.  A(x)
hence B
since
let a:t be such_that A(a)
```

(∃-E rule)

```
.
.
.
```

(⊥-E rule)

```
.
.
.
contradiction
hence A
```

```
a:t                              .
hence some x:t.  A(x)            .
                                 .
                                 end_since
```

(3) Induction schema

```
(nat-induction rule)
all m:nat.  A(m)
since induction on m
base
.
.
A(0)
step
let n:nat be arbitrary
[ind_hyp_is A(n)]
.
.
A(n+1)
end_since
```

```
(L(nat)-induction rule)
all m:L(nat).  A(m)
since induction on m
base
.
.
A(nil)
step
let a:nat.  x:nat be arbitrary
[ind_hyp_is A(x)]
.
.
A(cons(a.x))
end_since
```

(4) Description of equalities

```
(=-ref rule)
.
.
.
a = b
hence b = a
```

```
(=-trans rule)
.
.
.
a = b
.
.
.
b = c
hence a = c
```

```
(=-E rule)
.
.
.
a = b
.
.
.
A(a)
hence A(b)
```

(5) Reasoning about programs

```
(if =1 rule)
.
.
.
A
hence if A then B else C = B
```

```
(if =2 rule)
.
.
.
 A
hence if A then B else C = C
```

```
(β-red)
(fun [X]. F(X))(a) = F(a)
```

```
(μ-red)
mu [Z]. F(Z) = F(mu [Z]. F(Z))
```

The proof of Theorem A in PDL-QJ is given in the Appendix.

2.3 Proof Checker with Proof Finding Facility

One of the important research problems in developing good proof assistant systems is how to realise proof finding facilities. There are a lot of logical gaps in proofs written by humans; they usually skip the proofs of trivial facts which might cost hundreds of steps if they are proved in formal logic, and this makes proofs more readable for human beings. Therefore, the facility to fill the logical gaps is indispensable although a full automatic theorem proving facility is not always necessary.

The tactic facility of ML, which enables end users of the system to program proof finding strategies that they developed, is used in the proof finding package of the LCF [Gordon 79] and the Nuprl system [Constable 86]. The proof finding facility of the CAP-LA system uses the theorem proving mechanism of PROLOG in the first order logic part and the term rewriting system based theorem prover in the equational logic part [Sakai 86]. One observation of mathematical reasoning which the design of the proof finding facility is based on is that most of the logical gaps in mathematical reasoning by humans are in the equational logic part; they usually skip tedious reasoning about manipulation of equations and inequations of mathematical expressions. Therefore, the application of the rewriting technique is mostly elaborated. This observation basically holds for the reasoning of algorithms in computer science, so that this sort of proof checker system can also be used to realise our programming environment.

2.4 Expressive Power of Constructive Logic and Automatic Coding

The examples given in the previous section are examples of constructive proofs [Beeson 85]. It is well known that if a proof of the theorem is written constructively, the computational meaning of the proof can be analysed to extract a program automatically by using the notion of realisability [Kleene 45] [McCarty 84] [Hayashi 86]. In other words, some class of proofs can be compiled, so that there is no need for end users of the environment to be involved in the coding of programs if only the document in the form which is shown in the previous section is well prepared.

This raises a question: which class of problems can be solved as constructive reasoning, and which class of algorithms can be extracted automatically? It is basically possible to handle most of the elementary fundamental algorithms as given in [Aho 74]. For more complicated algorithms, it depends on the expressive power of the type system and how powerful the induction schema of the constructive logic involved is. For example, the notion of ordering, the induction schema on ordering structure, and the definition of the polynomial ring structure have to be described in constructive formal logic to extract the Gröbner base algorithm [Buchberger 83] [Takayama 87b]. Most of the definitions and proofs in linear algebra seem to be described in constructive ways. However, describing linear algebra in constructive logic and extracting programs of matrix calculation is also difficult [Takayama 87c]. The chief difficulty comes from the fact that matrices are described at various levels from vector spaces to ordered sets of scalars. If linear algebra is described at the level where matrices are handled as sequences of scalar valued functions, it is easy to prove the existence of products, sums, inverses, and determinants of

matrices. However, defining the type of, say, (m, n) matrices over the type of natural numbers, *nat*, is a little difficult because functions of type, $nat \times nat \rightarrow nat$, have to be handled generally. In other words, no specific function can be used in discussing matrix types, and the general strategy of type checking of $nat \times nat \rightarrow nat$ is unlikely to be defined in a constructive way. In addition, there is a difficulty in defining the type of regular matrices at this level. Linear independency of vector spaces can be defined by using the *apartness axiom* [Beeson 85]. Therefore, it is necessary to harmonise the vector space approach to linear algebra with the axiom and the scalar valued function approach.

Investigating and extending the expressive power of constructive type theory and the induction schema are ongoing research themes. Many mathematics libraries have been developed on the Nuprl system [Constable 86]. The extraction of a simple parser system is given in [Chisholm 87]. A constructive logic which has an array structure and the notion of call-by-reference is given in [Sasaki 86]. For induction schema, CIG recursion, which is an extended schema of that given in [Feferman 78], is given in [Hayashi 86]. A powerful induction schema in the calculus of construction [Coquand 86] is given in [Huet 87]. They aim to realise induction reasoning on general domains with particular postulates.

For non-deterministic or parallel algorithms, it is necessary to extend the programming logic from intuitionistic logic. The intuitionistic interpretation of the logical constant, $\vee$, prevents non-determinism [Takayama 87c]; to prove, for example, $A \vee B$, it is necessary to give the proof of A or B and the information of which of the formulae actually holds. Also, stream structure or infinite objects must be handled to describe parallelism such as generate and test style algorithms [Takayama 87d] An approach to handle parallel algorithms is presented in [Goto 85].

The second question is why the programming logic should be intuitionistic. The chief reason is that programs cannot be extracted from some proofs in classical logic at least by the notion of realisability. However, not all parts of the proof are related to the algorithm; the reasoning about the correctness of the algorithm given implicitly, or sometimes explicitly, in the proof may be written in classical logic. PX has a modal operator, $\Diamond$, to eliminate double negation [Hayashi 87]. However, it is not always clear which part of the proof is not related to algorithm, and the extraction of the program which is not needed to implement the necessary algorithm causes inefficiency of the extracted code. This problem is investigated in the later sections.

Here, the constructive logic is restricted to handling natural numbers, natural number lists and the induction schema on them. The program extraction algorithm is made clear and the technique to improve the efficiency of the extracted codes is elaborated.

3. Formal Structure of the System

The chief design feature of our programming environment is utilizing the fundamental facilities of the CAP-LA system. It is necessary to implement a sort of constructive logic in the proof assistant environment to realise the program extraction facility. However, as explained in the previous section, the CAP-LA system does not take the type theoretic approach. Therefore, it is desirable for the constructive logic used in our environment to be a different formulation from the constructive type theory; type structure and logic should be strictly separated. One suitable formulation for the criteria is Feferman-Beeson-Hayashi's constructive theory of functions and classes [Beeson 83] [Feferman 78] [Hayashi 87]. It is implemented in the PX system [Hayashi 86].

Another candidate is **QJ** [Sato 85] [Sato 86b], which is also a non type theoretic formulation of constructive logic. The chief difference between **QJ** and Feferman-Beeson-Hayashi's theory is that types can be treated as objects in the latter formulation; new data structure can be defined in the formal language. However, the formulation of **QJ** is much simpler, so that it is easier to work on the proof finding mechanism and extraction of an efficient programs from proofs.

3.1 Outline of QJ and Quty

3.1.1 QJ and Quty

QJ is an extension of the typed logical language, **Quty**, in the sense that a program in **Quty** is just a term of **QJ**. It is therefore possible to reason about **Quty** programs formally within **QJ**.

The logic of **QJ** is based on the *logic of partial terms* [Beeson 83] [Beeson 85] [Hayashi 86] where term expressions can fail to be denoted. This is a suitable framework for reasoning about partial recursive functions. Inference rules of **QJ** are basically an intuitionistic version of the Gentzen style of natural deduction with additional rules for terms such as construction and reduction rules of λ-expressions and if-then-else terms, special functions, **left**, **right**, **inl**, **inr** **outl** and **outr**, and induction schema on recursive type structures. The notion of q-*realisability* [Beeson 85] [Hayashi 86] is adopted as the fundamental facility to extract programs from proofs. For the semantics theory of **QJ**, a domain theoretic semantics is given [Sato 85]. In this model, a type structure is interpreted as a domain.

The type system of **QJ** does the typing of **Quty** programs, and types are not regarded as propositions as in constructive type theory.

Definition 1: *Types*
1. A type variable, s, is a type.
2. **1** is a type.
3. If σ and τ are types, then $\sigma \times \tau$ is a type.
4. If σ and τ are types, then $\sigma + \tau$ is a type.
5. If σ and τ are types, then $\sigma \rightarrow \tau$ is a type.
6. **recursive type:** If s is a type variable and σ is a type such that any subtype of σ which is of the form $\tau \rightarrow \rho$ or $\mu t.\tau$ does not contain free occurrence of s, then $\mu s.\sigma$ is a type.

Type **1** denotes a singleton poset (partially ordered set). Type $\sigma \times \tau$ denotes the cartesian product of the posets denoted by σ and τ. Type $\sigma + \tau$ denotes the disjoint sum of the posets denoted by σ and τ. Type $\sigma \rightarrow \tau$ denotes the set of partial functions from σ to τ. The poset denoted by $\mu s.\sigma$ is the smallest poset denoted by s such that s becomes the same as σ.

Definition 2: *Typed variables*
$V(\sigma)$ denotes the infinite set of variables whose type is σ. $V(\sigma) \cap V(\tau) = \phi$ if $\sigma \not\sim \tau$. $\sim$ is the equivalent relation of types. Type expressions which have the same denotation are regarded as equivalent.

Definition 3: *Typing rules*

$$\frac{a : \sigma \quad \sigma \sim \tau}{a : \tau} \qquad \frac{x \in V(\sigma)}{x : \sigma}$$

$$\frac{}{\bot_\sigma : \sigma} \qquad \frac{}{\top : \mathbf{1}}$$

$$\frac{a : \sigma \quad b : \tau}{a = b : \mathbf{1}} \qquad \frac{a : \sigma \quad b : \tau}{a, b : \sigma \times \tau}$$

$$\frac{a : \mathbf{1} \quad b : \mathbf{1}}{a \wedge b : \mathbf{1}} \qquad \frac{a : \sigma \times \tau}{\mathbf{left}a : \sigma}$$

$$\frac{a : \sigma \times \tau}{\mathbf{right}a : \tau} \qquad \frac{a : \sigma}{\mathbf{inl}_\tau a : \sigma + \tau}$$

$$\frac{a : \tau}{\mathbf{inr}_\sigma a : \sigma + \tau} \qquad \frac{a : \sigma + \tau}{\mathbf{out}a : \sigma}$$

$$\frac{a : \sigma \rightarrow \tau \quad b : \sigma}{a(b) : \tau} \qquad \frac{x : \sigma \quad a : \tau}{\lambda x.a : \sigma \rightarrow \tau}$$

$$\frac{x : \sigma \rightarrow \tau \quad a : \sigma \rightarrow \tau \quad \{x \text{ is not critical}\}}{\mu x.a : \sigma \rightarrow \tau}$$

$$\frac{a : \mathbf{1} \quad b : \tau \quad c : \tau}{\mathit{if}\ a\ \mathit{then}\ b\ \mathit{else}\ c : \tau} \qquad \frac{a : \mathbf{1} \quad b : \mathbf{1}}{a \vee b : \mathbf{1}}$$

$$\frac{x : \sigma \quad a : \mathbf{1}}{\exists x.a : \mathbf{1}}$$

The notion of *critical variables* makes the treatment of μ-expressions safe.

Note that the definition above is the typing rules of terms, and it shows that terms of type **1**, i.e, $\wedge$, $\vee$ and $\exists$ formulae and equalities of terms, are regarded as terms. They are also regarded as formulae. The operational semantics of terms, i.e, the **Quty** program, are given by regarding the terms as a term rewriting system; in particular, terms of type **1** are executed as logic programs [Sato 87].

Definition 4: *Formula*
1. Terms of type 1 are formulae.
2. If $a : \sigma$ and $b : \sigma$, then $a \sqsubseteq b$ is a formula.
3. If A and B are formulae, then $A \wedge B$ is a formula.
4. If A and B are formulae, then $A \vee B$ is a formula.
5. If A and B are formulae, then $A \supset B$ is a formula.
6. If x is a variable and A is a formula, then $\forall x.A$ is a formula.
7. If x is a variable and A a formula, then $\exists x.A$ is a formula.

$\sqsubseteq$ is the ordering of terms. The rules on $\sqsubseteq$ are also given. Equality $a = b$ is defined as $a \sqsubseteq b \wedge b \sqsubseteq a$.

3.1.2 Induction on the recursive type

Let $\tau \stackrel{\text{def}}{=} \mu s.\sigma$ be a recursive type and A be a formula. Then, the induction rule on the recursive type, τ, with regard to A is as follows:

$$\frac{\begin{matrix}[\sigma^*_A[x]] \\ A[x]\end{matrix}}{\forall x.A[x]}(ind)$$

where $\sigma^*_A[x]$, which is the induction hypothesis made from σ, is defined as follows.

Definition 5: *Induction hypothesis*
Let A be a formula and $\mu s.\sigma$ be a recursive type. The *induction hypothesis* σ^*_A is defined as follows by the induction on the construction of σ. $\sigma^*_A[x]$ is abbreviated to $\sigma^*[x]$ in the following description.

1. $s^*[x] \stackrel{\text{def}}{=} A[x]$
2. $1^*[x] \stackrel{\text{def}}{=} \top$
3. $(\rho_1 \times \rho_2)^*[x] \stackrel{\text{def}}{=} \rho_1^*[\mathbf{left}\ x] \wedge \rho_2^*[\mathbf{right}\ x]$
4. $(\rho_1 + \rho_2)^*[x] \stackrel{\text{def}}{=} ((\mathbf{outl}\ x)\downarrow \wedge \rho_1^*[\mathbf{outl}\ x]) \vee ((\mathbf{outr}\ x)\downarrow \wedge \rho_2^*[\mathbf{outr}\ x])$
5. $(\rho_1 \rightarrow \rho_2)^*[x] \stackrel{\text{def}}{=} \top$
6. $(\mu t.\rho)^*[x] \stackrel{\text{def}}{=} \top$

The form of type σ in recursive type τ must be $1 + *$. There are four recursive types depending on the form of the $*$ part:
$\mu s.(1 + s)$ $\cdots$ *natural numbers*
$\mu s.(1 + s \times s)$ $\cdots$ *S-expressions*
$\mu s.(1 + \sigma \times s)$ $\cdots$ *σ-list* (σ is a type)

According to the rules on the partial order, $\sqsubseteq$, the natural number type has flat ordering, and a function of type $\sigma \rightarrow \mu s.\ (1 + s)$ has almost flat ordering, i.e., functions such as $\lambda x.\ \bot$ exist. The natural number list type also has flat partial ordering. A function of type $\sigma \rightarrow \mu s.\ (1 + (\mu t.\ (1 + t)) \times s)$ also has almost flat ordering because the natural number list type has the bottom element, $\bot$.

As a theoretical result, the following holds.

Theorem 1: [Sato 85]
Every partial recursive function is representative in **QJ**.

3.1.3 Realisability interpretation

In the following, t q A reads "*t realised A*". This can also be read "*program t realises specification A*".

Definition 6: q*-realisability*
Let $x, y, \cdots$ denote variables, and $\overline{x}, \overline{y}, \cdots$ denote sequences of variables. $\overline{x}$ means $x_1 \wedge \cdots \wedge x_m$.

1. If A is an atomic formula, then a q $A \stackrel{\text{def}}{=} A \wedge a = nil$
2. $(\overline{x}, \overline{y})$ q $A \wedge B \stackrel{\text{def}}{=} \overline{x}$ q $A \wedge \overline{y}$ q B
3. $(z, \overline{x}, \overline{y})$ q $A \vee B \stackrel{\text{def}}{=} (\mathbf{outl}(z) \wedge A \wedge \overline{x}$ q $A \wedge \overline{y}\downarrow) \vee (\mathbf{outr}(z) \wedge \overline{x}\downarrow \wedge B \wedge \overline{y}$ q $B)$
4. $\overline{y}$ q $A \supset B \stackrel{\text{def}}{=} \overline{y}\downarrow \wedge \mathbf{mono}(\overline{y}) \wedge \forall \overline{x}.(A \wedge \overline{x}$ q $A \supset \overline{y}(\overline{x})$ q $B)$
5. $\overline{y}$ q $\forall x.A \stackrel{\text{def}}{=} \forall x.\ (\overline{y}(x)$ q $A)$
6. $(z, \overline{y})$ q $\exists x.A \stackrel{\text{def}}{=} z \wedge A_x[z] \wedge \overline{y}$ q $A_x[z]$

where $\mathbf{mono}(\overline{y}) \stackrel{\text{def}}{=} \forall \overline{x}_0.\forall \overline{x}_1.(\overline{x}_0 \sqsubseteq \overline{x}_1 \supset \overline{y}(\overline{x}_0) \sqsubseteq \overline{y}(\overline{x}_1))$. $A_x[z]$ means the substitution of z to x which occurs freely in A.

Note that by induction on the construction of A, $\vdash \bar{b}\ \mathrm{q}\ A \supset \bar{b}\downarrow$. In ordinal q-realisability, $a\ \mathrm{q}\ A \stackrel{\mathrm{def}}{=} A$ if A is atomic. However, the realisers for atomic formulae are restricted to nil in **QJ**. This is the same as **PX**-realisability [Hayashi 87]. The realisability here is not the standard q-realisability in this respect.

3.2 QPC Logic

QPC logic is a sugared version of a subset of **QJ**. It is the subset of **QJ** which is related to program extraction from proofs, and the primitive data structure is restricted to natural numbers and lists of natural numbers. By this restriction of data types, handling the $\sqsubseteq$ relation is easier. QPC is, roughly, an intuitionistic version of natural deduction with mathematical induction and induction on natural number lists plus higher order equality and inequality. The program part of QPC is also a subset of **Quty**, which is called *Tiny Quty*, and the operational semantics is given as ordinary lambda calculus with sequences of terms and multi-valued recursive call programs.

3.2.1 Expressions

Definition 7: *Types*

1. Primitive types: nat (the set of natural numbers), $L(nat)$ (natural number list), $bool$ (boolean type), $atom$, $lterm$ (logical term)
2. Function types: If σ_0 and σ_1 are types, then $\sigma_0 \rightarrow \sigma_1$ is also a type.
3. Type of sequence: If $\sigma_0 \cdots \sigma_{n-1}$ are types, then $\sigma_0 \times \cdots \times \sigma_{n-1}$ is also a type.

Definition 8: *Terms*

The terms of QPC are defined as typed terms.

1. Natural numbers

$0, 1, 2, \cdots$ are terms of type nat.

$$\frac{}{succ : nat \rightarrow nat} \qquad \frac{}{pred : nat \rightarrow nat} \qquad \frac{}{+ : nat \times nat \rightarrow nat}$$

2. Atoms

$$\frac{}{left : atom} \qquad \frac{}{right : atom}$$

3. Boolean

$$\frac{}{T : bool} \qquad \frac{}{F : bool}$$

4. Natural number list

$$\frac{}{NIL : L(nat)} \qquad \frac{}{cons : nat \times L(nat) \rightarrow L(nat)}$$

$$\frac{}{hd : L(nat) \rightarrow nat} \qquad \frac{}{tl : L(nat) \rightarrow L(nat)}$$

5. Typed variables

$$\frac{x \in V(\sigma)}{x : \sigma}$$

6. λ-expression

$$\frac{x : \sigma \quad t : \sigma \rightarrow \tau}{\lambda x.\, t : \tau}$$

7. Application

$$\frac{a : \sigma \to \tau \quad b : \sigma}{a(b) : \tau}$$

$a(b)$ is also denoted $(a\ b)$, $(a)b$ and $a\ b$.

8. Logical terms

$$\frac{a_1 : \sigma_1, \cdots, a_n : \sigma_n \quad b_1 : \sigma_1, \cdots, b_n : \sigma_n \quad R \textit{ is well defined on } \sigma_i (1 \leq i \leq n)}{a_1 R b_1 \wedge \cdots \wedge a_n R b_n : lterm}$$

where R is $<$, $\leq$ or $=$.

9. if-then-else

$$\frac{a : lterm \quad b : \sigma \quad c : \sigma}{if\ a\ then\ b\ else\ c : \sigma}$$

10. Sequence

$$\frac{a_1 : \sigma_1, \cdots a_n : \sigma_n}{(a_1, \cdots, a_n) : \sigma_1 \times \cdots \times \sigma_n}$$

If $n = 0$, the sequence is denoted nil. It is sometimes denoted $a_1, \cdots, a_n$. If x_is are typed variables, $\overline{x}$ denotes $(x_1, \cdots, x_n)$.

11. Any

$any[n] : \sigma_1 \times \cdots \times \sigma_n$ $(n : nat)$ denotes a sequence of arbitrary terms of suitable types.

12. Fixed point

$$\frac{\overline{x} : \sigma \to \tau \quad t : \sigma \to \tau}{\mu \overline{x}.\ t : \sigma \to \tau}$$

Definition 9: *Equivalence relations on terms*

(1) $\lambda nil.\ t \equiv t$
(2) $\lambda \overline{x}.\ nil \equiv nil$
(3) $\lambda \overline{x}.\ t \equiv \lambda x_1. \cdots \lambda x_n.\ t$
(4) $t(nil) \equiv t$
(5) $if\ a\ then\ (b_1, \cdots, b_n)\ else\ (c_1, \cdots, c_n) \equiv (if\ a\ then\ b_1\ else\ c_1, \cdots, if\ a\ then\ b_n\ else\ c_n)$
(6) $if\ a\ then\ nil\ else\ nil \equiv nil$
(7) $\lambda \overline{x}.\ (a_1, \cdots, a_m) \equiv (\lambda \overline{x}.\ a_1, \cdots, \lambda \overline{x}.\ a_m)$

Definition 10: *Formulae*

1. $\bot$ is an atomic formula.
2. Equation and inequation of terms are atomic formulae.

Note that if t is a term and σ is a type, then $t : \sigma$ is an abbreviation of $t = t$.

3. If A and B are formulae, then $A \wedge B$, $A \vee B$ and $A \supset B$ are formulae.
4. If x is a variable of type σ and A is a formula, then $\exists x : \sigma.A$ and $\forall x : \sigma.A$ are formulae.
5. If A is a formula, $\neg A \stackrel{\text{def}}{=} A \supset \bot$ is a formula.

The type declarations of bound variables are often omitted. Also, atomic formula $t : \sigma$ is often denoted simply t.

Note that inequality is defined as follows in the standard formulation of number theory: $a \leq b \stackrel{\text{def}}{=} \exists x : nat.\ a + x = b$ and $a < b \stackrel{\text{def}}{=} a \leq b \wedge \neg(a = b)$. However, they are treated as atomic in QPC.

3.2.2 Inference rules

The inference rules of QPC are as follows:

(1) Introduction rules

$$\frac{A \quad B}{A \wedge B} \qquad \frac{A}{A \vee B} \qquad \frac{B}{A \vee B} \qquad \frac{\begin{matrix}[A]\\ B\end{matrix}}{A \supset B} \qquad \frac{\begin{matrix}[x:\sigma]\\ A(x)\end{matrix}}{\forall x:\sigma.\ A(x)} \qquad \frac{t:\sigma \quad A(t)}{\exists x:\sigma.\ A(x)}$$

(2) Elimination rules

$$\frac{A \wedge B}{A} \qquad \frac{A \wedge B}{B} \qquad \frac{A \vee B \quad \begin{matrix}[A]\\ C\end{matrix} \quad \begin{matrix}[B]\\ C\end{matrix}}{C} \qquad \frac{A \quad A \supset B}{B}$$

$$\frac{t:\sigma \quad \forall x:\sigma.A(x)}{A(t)} \qquad \frac{\exists x:\sigma.A(x) \quad \begin{matrix}[t:\sigma, A(t)]\\ C\end{matrix}}{C} \qquad \frac{\bot}{A}$$

(3) Induction rule

$$\frac{A(0) \quad \begin{matrix}[x:nat, A(x)]\\ A(succ(x))\end{matrix}}{\forall x:nat.\ A(x)}(nat\text{-}ind) \qquad \frac{A(NIL) \quad \begin{matrix}[a:nat, x:L(nat), A(x)]\\ A(cons(a.x))\end{matrix}}{\forall x:L(nat).\ A(x)}(L(nat)\text{-}ind)$$

(4) Rules on equalities

$$\frac{t:\sigma}{t = t\ (in\ \sigma)}(=1) \qquad \frac{t = s\ (in\ \sigma)}{s = t\ (in\ \sigma)}(=2) \qquad \frac{p = q\ (in\ \sigma)\ q = r\ (in\ \sigma)}{p = r\ (in\ \sigma)}(=3)$$

$$\frac{t = s\ (in\ \sigma)\ A(t)}{A(s)}(=E)$$

$$\frac{b:\sigma \quad \lambda x.a:\sigma \to \tau}{(\lambda x.a)(b) = a_x[b]\ in\ \tau}(\lambda=) \qquad \frac{}{\mu\vec{z}.a = a_{\vec{z}}[\mu\vec{z}.a]}(\mu=)$$

$$\frac{a}{if\ a\ then\ b\ else\ c = b}(if=_1) \qquad \frac{\neg a}{if\ a\ then\ b\ else\ c = b}(if=_2)$$

(5) Axioms

Inference, for example, on inequalities is performed by giving axioms.

Definition 11: *Axiom*

1. Atomic formulae are axioms.
2. If A and B are axioms, then $A \wedge B$ and $A \supset B$ are axioms.
4. If A is a formula, then $\neg A$ is an axiom.

5. If A is an axiom and $\overline{x} : \sigma$ is a sequence of typed variables, then $\forall \overline{x}.\ A$ is an axiom.

Note that axioms are self realizing formulae [Beeson 85], and also the interpretation of the set of type 0 formulae in PX [Hayashi 87] and QPC.

3.2.3 Second order QPC

It is not permitted to quantify over propositions in **QJ**. However, propositional variables and universal quantification over them are very useful, particularly in manipulating user defined rules of inference or second order axioms such as course of value induction [Takayama 88a]. Second order QPC is a conservative extension of first order QPC.

Definition 12: *Types*

1. Types of the first order QPC.
2. *prop* (type of propositions and predicates)

Note that types such as $prop \rightarrow prop$ and $prop \times prop$ are not allowed.

Definition 13: *Terms*

The same as the first order part of QPC.

The definition of formulae is given as the typing rule of *prop*.

Definition 14: *Formulae*

$V(\sigma)$ denotes the infinite set of typed variables.

$$\frac{}{\bot : prop} \qquad \frac{P \in V(prop)}{P : prop} \qquad \frac{t_1 : \sigma \quad t_2 : \sigma \ (\sigma \neq prop)}{t_1 = t_2 : prop}$$

$$\frac{t_1 : \sigma \quad t_2 : \sigma \ (\sigma \neq prop)}{t_1 < t_2 : prop} \qquad \frac{t_1 : \sigma \quad t_2 : \sigma \ (\sigma \neq prop)}{t_1 \leq t_2 : prop} \qquad \frac{A : prop \quad B : prop}{A \wedge B : prop}$$

$$\frac{A : prop \quad B : prop}{A \vee B : prop} \qquad \frac{A : prop \quad B : prop}{A \supset B : prop} \qquad \frac{x \in V(\sigma) \ (\sigma \neq prop) \quad A : prop}{\forall x : \sigma.\ A : prop}$$

$$\frac{x \in V(\sigma) \ (\sigma \neq prop) \quad A : prop}{\exists x : \sigma.\ A : prop} \qquad \frac{A : prop}{\neg A : prop}$$

The inference rules of second order QPC are those of first order QPC and the following two rules on second order universal quantifier $\forall^2$:

$$\frac{\begin{array}{c}[P : prop] \\ F(P)\end{array}}{\forall^2 P : prop.\ F(P)}(\forall^2\text{-I}) \qquad \frac{Q : prop \quad \forall^2 P : prop.F(P)}{F(Q)}(\forall^2\text{-E})$$

3.3 Proof Compilation

The realisability is reformulated here as the Ext procedure [Takayama 88], which takes proof trees as input and returns functional style programs as output. The realiser code extracted by Ext is in the form of a sequence of terms.

3.3.1 Realizing variable sequence and length of formulae

The *realizing variable sequence* (or simply *realizing variables*) for a formula, A, which is denoted $Rv(A)$, is a sequence of variables to which realiser codes for the formula are assigned. Realizing variable sequences are used as the realiser code for assumption in the reasoning of natural deduction.

Definition 15: $Rv(A)$

1. $Rv(A) \stackrel{\text{def}}{=} (nil)$, if A is atomic.
2. $Rv(A \wedge B) \stackrel{\text{def}}{=} (Rv(A), Rv(B))$.
3. $Rv(A \vee B) \stackrel{\text{def}}{=} (z, Rv(A), Rv(B))$ where z is a new variable.
4. $Rv(A \supset B) \stackrel{\text{def}}{=} Rv(B)$.
5. $Rv(\forall x : \sigma.\ A(x)) \stackrel{\text{def}}{=} Rv(A(x))$.
6. $Rv(\exists x : \sigma.\ A(x)) \stackrel{\text{def}}{=} (z, Rv(A(x)))$ where z is a new variable.

Example:
$Rv(\forall x : nat.\ ((x \geq 0) \supset (x = 0 \vee \exists y : nat.\ succ(y) = x))) = (z_0, z_1)$
where z_0 denotes the information that shows which subformula of $\vee$ formula holds and z_1 denotes the realizing variables of $\exists y : nat.\ succ(y) = x$. Note that $Rv(succ(y) = x) = (nil)$.

Definition 16: *Length of formulae*
$l(A)$, which is called the *length of formula* A, is the length of $Rv(A)$.

3.3.2 Definition of the Ext procedure

In the following description, a substitution is denoted $\{X_0/T_0, \cdots, X_{n-1}/T_{n-1}\}$, which means substituting T_i for X_i, and X_i may be both a variable and a sequence of variables. When X_i is a sequence of variables, T_i must also be a sequence of terms. Application of a substitution, θ, to a term, T, is denoted $T\theta$. The following functions are also used in defining Ext:

- $proj(n)$ $\cdots$ function that projects the nth element of a sequence of terms
- $proj(I)$ $\cdots$ I is a finite set of natural numbers. If S is a sequence of terms of length n and $m < n$, then

$$proj(\{i_0, \cdots, i_m\}) \stackrel{\text{def}}{=} (proj(i_0)(S), \cdots, proj(i_m)(S))$$

- $tseq(n)$ $\cdots$ function that returns the subsequence of a given sequence: if S is a sequence of length n, then

$$tseq(i) = (proj(i)(S), proj(i+1)(S), \cdots, proj(n-1)(S))$$

• $ttseq(n,m)$ ··· function that returns the subsequence of a given sequence: if S is a sequence of length n, then

$$ttseq(i,l) = (proj(i)(S), proj(i+1)(S), \cdots, proj(i+(l-1))(S))$$

In the following, Π always stands for proof trees, and Σ for the sequence of proof trees.

(1) For the realiser codes of assumptions, realizing variable sequences are used:

$Ext([A]) \stackrel{\text{def}}{=} Rv(A)$

(2) No significant code is extracted from atomic formulae and axioms:

$Ext\left(\dfrac{\Sigma}{A}(Rule)\right) \stackrel{\text{def}}{=} nil$

where A is an atomic formula or an axiom.

(3) The realiser codes for the $\wedge$ and $\vee$ formulae are denoted as sequences. Atoms $left$ and $right$ are used to denote the information indicating which of the formulae connected by $\vee$ actually holds.

- $Ext\left(\dfrac{\dfrac{\Sigma_0}{A_0} \cdots \dfrac{\Sigma_{n-1}}{A_{n-1}}}{A_0 \wedge \cdots \wedge A_{n-1}}(\wedge\text{-}I)\right) \stackrel{\text{def}}{=} \left(Ext\left(\dfrac{\Sigma_0}{A_0}\right), \cdots, Ext\left(\dfrac{\Sigma_{n-1}}{A_{n-1}}\right)\right)$

- $Ext\left(\dfrac{\dfrac{\Sigma}{A_0 \wedge \cdots \wedge A_{n-1}}}{A_i}(\wedge\text{-}E)\right) \stackrel{\text{def}}{=} ttseq\left(\sum_{k=0}^{i-1} l(A_k), l(A_i)\right)\left(Ext\left(\dfrac{\Sigma}{A_0 \wedge \cdots \wedge A_{n-1}}\right)\right)$

- $Ext\left(\dfrac{\dfrac{\Sigma}{A}}{A \vee B}(\vee\text{-}I)\right) \stackrel{\text{def}}{=} \left(left, Ext\left(\dfrac{\Sigma}{A}\right), any[l(B)]\right)$

- $Ext\left(\dfrac{\dfrac{\Sigma}{B}}{A \vee B}(\vee\text{-}I)\right) \stackrel{\text{def}}{=} \left(right, any[l(A)], Ext\left(\dfrac{\Sigma}{B}\right)\right)$

(4) The realiser code extracted from the proofs by using the $\vee$-E rule is the if-$then$-$else$ program.

$$Ext\left(\dfrac{\dfrac{\Sigma_0}{A \vee B} \quad \dfrac{[A]\ \Sigma_1}{C} \quad \dfrac{[B]\ \Sigma_2}{C}}{C}(\vee\text{-}E)\right)$$

$$\stackrel{\text{def}}{=} if\ lcft = proj(0)\left(Ext\left(\dfrac{\Sigma_0}{A \vee B}\right)\right)\ then\ Ext\left(\dfrac{[A]\ \Sigma_1}{C}\right)\theta\ else\ Ext\left(\dfrac{[B]\ \Sigma_2}{C}\right)\theta$$

where

$$\theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} Rv(A)/ttseq(1, l(Rv(A)))\left(Ext\left(\dfrac{\Sigma_0}{A \vee B}\right)\right), \\ Rv(B)/tseq(l(Rv(A))+1)\left(Ext\left(\dfrac{\Sigma_1}{A \vee B}\right)\right) \end{array} \right\}$$

(5) λ expressions are extracted from the proofs in $(\supset\text{-}I)$ and $(\forall\text{-}I)$:

- $$Ext\left(\dfrac{\begin{array}{c}[x:\sigma]\\ \dfrac{\Sigma}{A(x)}\end{array}}{\forall x:\sigma.\ A(x)}(\forall\text{-}I)\right) \stackrel{\text{def}}{=} \lambda x.\ Ext\left(\dfrac{\begin{array}{c}[x:\sigma]\\ \Sigma\\ \hline A(x)\end{array}}{}\right)$$

- $$Ext\left(\dfrac{\begin{array}{c}[A]\\ \dfrac{\Sigma}{B}\end{array}}{A \supset B}(\supset\text{-}I)\right) \stackrel{\text{def}}{=} \lambda Rv(A).\ Ext\left(\dfrac{\begin{array}{c}[A]\\ \Sigma\\ \hline B\end{array}}{}\right)$$

(6) The code that is in the form of a function application is extracted from the proofs in $(\supset\text{-}E)$ and $(\forall\text{-}E)$:

- $$Ext\left(\dfrac{\dfrac{\Sigma_0}{A} \quad \dfrac{\Sigma_1}{A \supset B}}{B}(\supset\text{-}E)\right) \stackrel{\text{def}}{=} Ext\left(\dfrac{\Sigma_1}{A \supset B}\right)\left(Ext\left(\dfrac{\Sigma_0}{A}\right)\right)$$

- $$Ext\left(\dfrac{\dfrac{\Sigma_0}{t:\sigma} \quad \dfrac{\Sigma_1}{\forall x:\sigma.\ A(x)}}{A(t)}(\forall\text{-}E)\right) \stackrel{\text{def}}{=} Ext\left(\dfrac{\Sigma_1}{\forall x:\sigma.\ A(x)}\right)(t)$$

(7) The codes extracted from proofs in $(\exists\text{-}I)$ and $(\exists\text{-}E)$ are as follows:

- $$Ext\left(\dfrac{\dfrac{\Sigma_0}{t:\sigma} \quad \dfrac{\Sigma_1}{A(t)}}{\exists x:\sigma.\ A(x)}(\exists\text{-}I)\right) \stackrel{\text{def}}{=} \left(t, Ext\left(\dfrac{\Sigma_1}{A(t)}\right)\right)$$

- $$Ext\left(\dfrac{\dfrac{\Sigma_0}{\exists x:\sigma.\ A(x)} \quad \dfrac{\begin{array}{c}[x:\sigma, A(x)]\\ \Sigma_1\\ \hline C\end{array}}{}}{C}(\exists\text{-}E)\right) \stackrel{\text{def}}{=} Ext\left(\dfrac{\begin{array}{c}[x:\sigma, A(x)]\\ \Sigma_1\\ \hline C\end{array}}{}\right)\theta$$

where $\theta \stackrel{\text{def}}{=} \left\{ Rv(A(x))/tseq(1)\left(Ext\left(\dfrac{\Sigma_0}{\exists x:\sigma.\ A(x)}\right)\right), x/proj(0)\left(Ext\left(\dfrac{\Sigma_0}{\exists x:\sigma.\ A(x)}\right)\right)\right\}$.

(8) Any code is extracted from a proof in the $(\bot\text{-}E)$ rule:

- $Ext\left(\dfrac{\dfrac{\Sigma}{\perp}}{A}(\perp\text{-}E)\right) \stackrel{\text{def}}{=} any[l(A)]$.

(9) The multi-valued recursive call function is extracted from the induction proofs [Takayama 88c].

- $$Ext\left(\frac{\dfrac{\Sigma_0}{A(0)} \quad \dfrac{[x : nat, A(x)]}{\dfrac{\Sigma_1}{A(succ(x))}}}{\forall x : nat.\ A(x)}(nat\text{-}ind)\right)$$

$$\stackrel{\text{def}}{=} \mu\ \overline{z}.\ \lambda\ x.\ if\ x = 0\ then\ Ext\left(\frac{\Sigma_0}{A(0)}\right)\ else\ Ext\left(\frac{[x : nat, A(x)]}{\dfrac{\Sigma_1}{A(succ(x))}}\right)\sigma$$

where $\overline{z} = Rv(A(x))$, and $\sigma = \{\overline{z}/\overline{z}(pred(x)), x/pred(x)\}$.

- $$Ext\left(\frac{\dfrac{\Sigma_0}{A(NIL)} \quad \dfrac{[a : nat.x : L(nat), A(x)]}{\dfrac{\Sigma_1}{A(cons(a,x))}}}{\forall x : L(nat).\ A(x)}(L(nat)\text{-}ind)\right)$$

$$\stackrel{\text{def}}{=} \mu\ \overline{z}.\ \lambda\ x.\ if\ x = NIL\ then\ Ext\left(\frac{\Sigma_0}{A(NIL)}\right)\ else\ Ext\left(\frac{[a : nat.x : L(nat), A(x)]}{\dfrac{\Sigma_1}{A(cons(a,x))}}\right)\sigma$$

where $\overline{z} = Rv(A(x))$, and $\sigma = \{\overline{z}/\overline{z}(tl(x)), x/tl(x)\}$.

Note that $\overline{z}$ denotes a sequence of variables, so that $\mu\overline{z}.\ \cdots$ is a multi-valued recursive call function. The multi-valued recursive call function of *degree* n, $\mu\ (z_0, \cdots, z_{n-1}).\ F(z_0, \cdots, z_{n-1})$ where $n \geq 1$ and $F(z_0, \cdots, z_{n-1})$ is a term with free variables, $z_0, \cdots, z_{n-1}$, is defined as follows:

1) Assume that $F(z_0, \cdots, z_{n-1})$ is equivalent to the following sequence of functions:

$$(F_0(z_0, \cdots, z_{n-1}), \cdots, F_{n-1}(z_0, \cdots, z_{n-1}))$$

2) Let $G_i \stackrel{\text{def}}{=} \mu z_i.\ F_i(z_0, \cdots, z_{n-1})$, where $0 \leq i \leq n-1$.
3) Define H_i, where $0 \leq i \leq n-1$, inductively as follows:
(a) $H_0 \stackrel{\text{def}}{=} G_0$;
(b) For $1 \leq i \leq n-1$, let $H_i \stackrel{\text{def}}{=} G_i\{z_0/H_0, \cdots, z_{n-1}/H_{n-1}\}$;
(c) Redefine H_k, where $0 \leq k \leq i$, as follows: $H_k' \stackrel{\text{def}}{=} H_k\{z_i/G_i\}$.
4) $\mu(z_0, z_1, z_2).F(z_0, \cdots, z_{n-1}) \stackrel{\text{def}}{=} (H_0(z_0, \cdots, z_{n-1}), \cdots, H_{n-1}(z_0, \cdots, z_{n-1}))$.

Example:
Let $F(z_0, z_1, z_2) \stackrel{\text{def}}{=} (\lambda x.\ p(z_0, z_1, z_2),\ \lambda y.\ q(z_0, z_1, z_2),\ \lambda z.\ r(z_0, z_1, z_2))$. By the definition and

$(\mu\text{-}=)$ rule:

$$\frac{}{\mu z.\ F(z) = F(\mu z.\ F(z))}(\mu\text{-}=)$$

$\mu(z_0, z_1, z_2).F(z_0, z_1, z_2) = (H_0(z_0, z_1, z_2), H_1(z_0, z_1, z_2), H_2(z_0, z_1, z_2))$
where

$$H_0 \stackrel{\text{def}}{=} \mu z_0.\lambda x.p(z_0,\ \mu z_1.\lambda y.q(z_0, z_1, \mu z_2.\lambda z.r(z_0, z_1, z_2)),\ \mu z_2.\lambda z.r(z_0, \mu z_1.\lambda y.q(z_0, z_1, z_2), z_2))$$

$$H_1 \stackrel{\text{def}}{=} \mu z_1.\lambda y.q(\mu z_0.\lambda x.p(z_0, z_1, \mu z_2.\lambda z.r(z_0, z_1, z_2)),\ z_1,\ \mu z_2.\lambda z.r(\mu z_0.\lambda x.p(z_0, z_1, z_2), z_1, z_2))$$

$$H_2 \stackrel{\text{def}}{=} \mu z_2.\lambda z.r(\mu z_0.\lambda x.p(z_0, \mu z_1.\lambda y.q(z_0, z_1, z_2), z_2),\ \mu z_1.\lambda y.q(\mu z_0.\lambda x.p(z_0, z_1, z_2), z_1, z_2),\ z_2)$$

The execution of multi-valued recursive functions is quite expensive, so that making the degree smaller is an effective way of generating an efficient realiser code.

Theorem 2: Soundness of the Ext procedure
Let A be a sentence. If $\vdash_{\mathbf{QPC}_2} A$ and P is its proof tree, then $\vdash_{\mathbf{QPC}} Ext(P)$ q A.

Proof: By straightforward conversion from the proof of the theorem on the soundness of the realisability interpretation of QJ. See [Sato 85]. ∎

Definition 17: *Principal sign and C-formula*
(1) Let A be a formula that is not atomic. Then, A has exactly one of the forms $A \wedge B$, $A \vee B$, $A \supset B$, $\forall x.A$, and $\exists x.A$; the symbol $\wedge$, $\vee$, $\supset$, $\forall$, or $\exists$, respectively, is called the *principal sign* of A.
(2) A formula with the principal sign, C, is called the *C formula.*

3.3.3 Proof compilation of second order QPC

The proof compilation algorithm for $\forall^2$-I/E rules is a native extension of that for first order logic.

$$Ext(\frac{\begin{array}{c}[P : prop]\\ F(P)\end{array}}{\forall^2 P : prop.\ F(P)}(\forall^2\text{-I})) \stackrel{\text{def}}{=} \Lambda RV(P).\ Ext(P/RV(P) \vdash F(P))$$

$RV(P)$ is the variable to which $Rv(Q)$, where Q is a particular first order formula, is to be substituted. The intentional meaning of Λ is similar to ordinal lambda notation. Λ is used simply to distinguish the above case. $Ext(P/RV(P) \vdash F(P))$ means that if $Ext(P)$ is needed in the procedure of proof compilation of $F(P)$, $RV(P)$ should be used as the value of $Ext(P)$.

$$Ext(\frac{Q : prop_1 \quad \forall^2 P : prop.\ F(P)}{F(Q)}(\forall^2\text{-E})) \stackrel{\text{def}}{=} \beta\text{-reduction on } Ext(\forall^2 P : prop.\ F(P))(Rv(Q))$$

$Ext(\forall^2 P : prop.\ F(P))$ must be of the form $\Lambda RV(P).Ext(P/RV(P) \vdash F(P))$, so that by β-reduction of the Λ-expression, the above code is $Ext(Q \vdash F(Q))$. This corresponds to the following normalisation of second order logic:

$$\frac{\begin{array}{c}\Pi_0\\ Q : prop\end{array} \quad \frac{\begin{array}{c}[P : prop]\\ \Pi_1(P)\\ F(P)\end{array}}{\forall^2 P : prop.F(P)}}{F(Q)} \Longrightarrow \begin{array}{c}\Pi_0\\ [Q : prop]\\ \Pi_1(Q)\\ F(Q)\end{array}$$

4. Optimization Technique

There are two kinds of optimization technique in terms of proof complication. One is optimization of algorithms at proof tree level. For example, as explained in section 2, the extracted sorting algorithms vary from the bubble sort algorithm to the quick sort algorithm according to the proof strategies. It needs quite a drastic change of proof strategy, so that it is difficult to perform this sort of change of proofs automatically. However, a much smaller change of proofs is possible. A technique to optimise programs at proof level, *pruning*, is given in [Goad 80]. Generally, proofs contain a lot of information about the programs that correspond to the proofs, and the pruning technique uses the information in optimization to change the strategies of algorithms. Goad also investigated an application of the proof normalization method to partial evaluation of proofs. [Bates 79] applied a traditional syntactical optimization technique on the code extracted from proofs which might destroy the clear correspondence between proofs and program through realisability.

Another kind of optimization is reducing the redundancy of the codes extracted by the Ext procedure, in other words, realiser codes. Realiser codes generally contain a lot of codes which are irrelevant to the algorithm. [Sasaki 86] improved the program extraction algorithm based on realisability so that the trivial code for formulae that have no computational meaning can be simplified. The basic idea is as follows: if A and B are atomic formulae, then the computational meaning is trivial, so that the code extracted from, for instance, $A \wedge B$ is $(trivial, trivial)$. The modified program extractor simplifies the code to *trivial*. A similar technique is used in the PX system [Hayashi 86] as *type 0 formulae*. However, the code extracted from constructive proofs still has redundancy, and it causes heavy runtime overhead.

If a constructive proof of the following formal specification is given:

$$\forall x : \sigma_0.\ \exists y : \sigma_1.\ A(x, y)$$

where σ_0 and σ_1 are types, and $A(x, y)$ is a formula with free variables, x and y, the function, f, which satisfies the following condition can be extracted by q-realisability:

$$\forall x : \sigma_0. A(x, f(x)).$$

For example, if the proof is as follows:

$$\dfrac{\dfrac{\dfrac{[x:\sigma_0] \quad [x:\sigma_0]}{\dfrac{\Sigma_0}{t_x : \sigma_0} \quad \dfrac{\Sigma_1}{A(x,t_x)}(\exists\text{-}I)}}{\exists y : \sigma_1.\ A(x,y)}}{\forall x : \sigma_0.\ \exists y : \sigma_1.\ A(x,y)}(\forall\text{-}I)$$

where Σ_0 and Σ_1 denote sequences of subtrees, the extracted code can be expressed as:

$$\lambda x.\ (t_x, T)$$

where T is the code extracted from the subtree determined by $A(x, t_x)$, and t_x denotes a term which contains a free variable, x. In this paper, the executable codes extracted by the Ext procedure are in the form of sequences of terms or functions which output a sequence of terms. The codes contain verification information which is not necessary in practical computation. In this case, the expected code is:

$$f \stackrel{\text{def}}{=} \lambda x.\ t_x$$

so that T is the redundant code.

The most reasonable idea to overcome this problem would be introducing suitable notation to specify which part of the proof is necessary in terms of computation. The set notation, $\{x : A|B\}$, is introduced in the Nuprl system [Constable 86] as a weaker notion of $\exists x : A.\ B$. This is done to skip the extraction of the justification for B. [Mohring 88] modified the calculus of constructions [Coquand 86][Huet 86][Huet 88] by introducing two kinds of constants, *Prop* and *Spec*, to distinguish the formulae in proofs whose computational meaning is not necessary. These works are performed in the type theoretic formulation of constructive logic in the style of Martin-Löf.

In the following, the inefficiency of the extracted codes is demonstrated through examples, and three optimization techniques are given. The first two techniques, normalization and the modified $\vee$ code are well known. The third one is called the *extended projection method* [Takayama 88b], and is a proof theoretic method in the style of D. Prawitz to perform the program analysis at proof tree level, and to generate a redundancy-free realiser code.

4.1 Redundancy in Realiser Codes

The extraction of a gcd program is given in [Takayama 88a]. The specification is

$$\forall x : nat.\forall y : nat.\exists z : nat.((z \mid x) \wedge (z \mid y) \wedge (\forall w : nat.((w \mid x) \wedge (w \mid y)) \supset w \leq z))$$

where $(p \mid q) \stackrel{\text{def}}{=} \exists r : nat.q = r \cdot p$ and the outline of the proof is as follows:

$$\dfrac{\dfrac{\Sigma_0}{\forall x : nat.(\forall y : nat.(y < x \supset Q(y)) \supset Q(x))}\ \Pi_1}{\forall z : nat.Q(z)}$$

where Π_1 is as follows:

$$\dfrac{\dfrac{\Sigma_2}{Q : prop}\quad \dfrac{\dfrac{\Sigma_3}{\forall x : nat.(\forall y : nat.(y < x \supset P(y)) \supset P(x)) \supset \forall z : nat.P(z)}(\forall^2\text{-I})}{\text{COV-IND}}}{\forall x : nat.(\forall y : nat.(y < x \supset Q(y)) \supset Q(x)) \supset \forall z : nat.Q(z)}(\forall^2\text{-E})$$

where COV-IND is as follows:

$$\forall^2 P : prop.\ (\forall x : nat.(\forall y : nat.(y < x \supset P(y)) \supset P(x)) \supset \forall z : nat.P(z))$$

The extracted code by Ext is as follows:

$GCD \equiv (f\,g)$

$$
\begin{array}{l}
f \equiv \lambda\,(w_0, w_1, w_2).\\
\quad \lambda z.(w_0, w_1, w_2)\ z\\
\quad\quad ((\mu\ (z_0, z_1, z_2).\lambda x.\\
\quad\quad\quad if\ x = 0\ then\ \lambda y.any[3]\\
\quad\quad\quad else\ if\ left = (\mu z.\lambda k.\ if\ k = 0\ then\ left\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad else\ if\ left = z\ then\ right\ else\ right)\ pred(x)\\
\quad\quad\quad\quad then\ \lambda y.\ (((w_0, w_1, w_2)\ 0)\ \lambda y.any[3])\\
\quad\quad\quad\quad else\ \lambda y.if\ left = (AA\ pred(x)\ y)\\
\quad\quad\quad\quad\quad\quad then\ ((z_0, z_1, z_2)\ pred(x)\ y)\\
\quad\quad\quad\quad\quad\quad else\ (w_0, w_1, w_2)\ pred(x)((z_0, z_1, z_2)\ pred(x)))\ z)
\end{array}
$$

$$
\begin{array}{l}
AA \equiv \mu z.\lambda x.\ if\ x = 0\ then\ \lambda\ y.\ if\ y = 0\ then\ right\ else\ any[1]\\
\quad\quad\quad\quad else\ \lambda y.\ if\ left = (\mu z.\ \lambda k.\ \ if\ k = 0\ then\ left\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad else\ if\ left = z\ then\ right\ else\ right)\ y\\
\quad\quad\quad\quad\quad\quad\quad\quad then\ left\\
\quad\quad\quad\quad\quad\quad\quad\quad else\ if\ left = (z\ pred(x)\ pred(y))\ then\ left\ else\ right
\end{array}
$$

$$
\begin{array}{l}
g \equiv \lambda n.\ \lambda(z_0, z_1, z_2).\\
\quad\quad if\ left = (\mu z.\ \lambda k.\ if\ k = 0\ then\ left\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad else\ if\ left = z\ then\ right\ else\ right)\ n\\
\quad\quad then\ \lambda m.(m, (\lambda p.0)\ m, (\lambda q.1)\ m)\\
\quad\quad else\ \lambda m.\ (\ (z_0\ (m\ mod\ n)\ n), (z_2\ (m\ mod\ n)\ n),\\
\quad\quad\quad\quad\quad\quad (z_1\ (m\ mod\ n)\ n) + (z_2\ (m\ mod\ n)\ n) \cdot \dfrac{m - (m\ mod\ n)}{n})
\end{array}
$$

This code contains three kinds of inefficiency:
(1) β redex: The code, GCD, is an application of f to g (λ terms). Also, $(\lambda p.1)m$ and $(\lambda q.0)m$ are β redex. These parts can be optimised by performing the partial evaluation of β reduction.
(2) Simple decision procedures in complicated algorithms
The code $left = (AA\ x\ y)$ in f and the code $left = (\mu z.\lambda k.if\ k = 0\ then\ left\ else\ if\ left = z\ then\ right\ else\ right)n$ are logically equivalent to the simple decision procedures, $y < x$ and $n = 0$. They are the codes extracted from the induction proofs of $\forall x : nat.\forall y : nat.(y < x + 1 \supset y < x \vee y = x)$ and $\forall x : nat.x = 0 \vee x > 0$. These have to be proved by mathematical induction as long as constructive way of reasoning is maintained.
(3) Verification information
The whole program calculates the sequence of three natural numbers: the gcd of two given natural numbers and two other natural numbers. The latter natural numbers are redundant. This can be observed by the $\mu(z_0, z_1, z_2)$ part in f. The reason why the multi-valued function is extracted is as follows. The realizing variable sequence of the specification is (z_0, z_1, d_2). z_0 corresponds to the variable in the specification, z, which is quantified by $\exists$, and z_1 and z_2 are realizing variables of $(z \mid x)$ and $(z \mid y)$. Therefore, the two redundant natural numbers which the program calculates are the justification that z actually divides x and y.

4.2 Proof Normalization

It is well known that proof normalization [Prawitz 65] [Troelstra 73] corresponds to the evaluation of λ expressions. In particular, the following two rules correspond to β reduction of the extracted code by the Ext procedure, so that the inefficiency of β redex can be removed by this method.

(1) $\forall$-normalisation

$$\dfrac{\dfrac{\dfrac{\Sigma(a)}{P(a)}(\forall\text{-I})}{\forall x.P(x)}(\forall\text{-E})}{P(t)} \quad \Longrightarrow \quad \dfrac{\Sigma(t)}{P(t)}$$

(2) $\supset$ normalisation

$$\dfrac{\dfrac{\Sigma_0}{A} \quad \dfrac{\begin{matrix}[A]\\ \Sigma_1\\ B\end{matrix}}{A \supset B}(\supset\text{-I})}{B}(\supset\text{-E}) \quad \Longrightarrow \quad \dfrac{\begin{matrix}\Sigma_0\\ [A]\\ \Sigma_1\end{matrix}}{B}$$

By applying these rules, the proof of the specification is made simpler and the following program is extracted by the Ext procedure:

$$\begin{array}{l}
\lambda z.\ \lambda(z_0, z_1, z_2).\\
\quad g_z\ z\ ((\mu(z_0, z_1, z_2).\ \lambda x.\\
\qquad if\ x = 0\ then\ \lambda y.any[3]\\
\qquad else\ if\ left = (\mu z.\lambda k.\ if\ k = 0\ then\ left\\
\qquad\qquad\qquad\qquad else\ if\ left = z\ then\ right\ else\ right)\ pred(x)\\
\qquad\quad then\ \lambda y.\ g_0\ \lambda y.any[3]\\
\qquad\quad else\ \lambda y.\ if\ left = (AA\ pred(x)\ y)\\
\qquad\qquad\quad then\ ((z_0, z_1, z_2)\ pred(x)\ y)\\
\qquad\qquad\quad else\ g_z\ ((z_0, z_1, z_2)\ pred(x)))\ z)
\end{array}$$

where

$$\begin{array}{l}
g_z \stackrel{\text{def}}{=} \lambda(z_0, z_1, z_2).\\
\quad (\ if\ left = (\mu z.\ \lambda k.\ \ if\ k = 0\ then\ left\\
\qquad\qquad\qquad\qquad else\ if\ left = z\ then\ right\ else\ right)\ z\\
\qquad then\ \lambda m.(m, 0, 1)\\
\qquad else\ \lambda m.\ (\ (z_0\ (m\ mod\ z)\ z), (z_2\ (m\ mod\ z)\ z),\\
\qquad\qquad\qquad (z_1\ (m\ mod\ z)\ z)\\
\qquad\qquad\qquad +(z_2\ (m\ mod\ z)\ z) \cdot \dfrac{m - (m\ mod\ z)}{z}))
\end{array}$$

4.3 Modified $\vee$-code

This technique removes the inefficiency of simple decision procedures in complicated algorithms. The idea is to translate complex decision procedures into simple and equivalent decision procedures. A similar technique is used in the Nuprl program extractor system [Bates 79] [Constable 86] [Sasaki 86].

The definition of the Ext procedure on the $\vee$-E rule is modified as follows. Note that logical terms are the formulae which can be executed as programs.

$$Ext\left(\dfrac{\dfrac{\Sigma_0}{A \vee B} \quad \dfrac{\begin{matrix}[A]\\ \Sigma_1\\ C\end{matrix}}{C} \quad \dfrac{\begin{matrix}[B]\\ \Sigma_2\\ C\end{matrix}}{C}}{C}(\vee\text{-E})\right)$$

$\stackrel{\text{def}}{=}$

a) $if\ A\ then\ Ext\left(\frac{[A]}{\frac{\Sigma_1}{C}}\right)\ else\ Ext\left(\frac{[B]}{\frac{\Sigma_2}{C}}\right)$ [modified $\vee$ code]

$\cdots$ when A is a logical term

b) $if\ B\ then\ Ext\left(\frac{[B]}{\frac{\Sigma_2}{C}}\right)\ else\ Ext\left(\frac{[A]}{\frac{\Sigma_1}{C}}\right)$ [modified $\vee$ code]

$\cdots$ when A is not a logical term, but B is.

c) $if\ left = proj(0)(Ext\left(\frac{\Sigma_0}{A \vee B}\right))\ then\ Ext\left(\frac{[A]}{\frac{\Sigma_1}{C}}\right)\theta\ else\ Ext\left(\frac{[B]}{\frac{\Sigma_2}{C}}\right)\theta$

$\cdots$ otherwise

where

$$\theta \stackrel{\text{def}}{=} \left\{ \begin{array}{l} Rv(A)/ttseq(1, l(Rv(A)))\left(Ext\left(\frac{\Sigma_0}{A \vee B}\right)\right), \\ Rv(B)/tseq(l(Rv(A))+1)\left(Ext\left(\frac{\Sigma_1}{A \vee B}\right)\right) \end{array} \right\}$$

The following code is extracted by applying the modified Ext procedure to the proof obtained in the previous section:

$$\begin{array}{l} \lambda\ z.\lambda(z_0, z_1, z_2). \\ \quad g_z^1\ z\ ((\mu(z_0, z_1, z_2).\ \lambda x.\ if\ x = 0\ then\ \lambda y.any[3] \\ \qquad\qquad else\ if\ pred(x) = 1\ then\ \lambda y.\ g_0^1\ \lambda y.any[3] \\ \qquad\qquad\quad else\ \lambda y.\ if\ y < pred(x) \\ \qquad\qquad\qquad then\ ((z_0, z_1, z_2)\ pred(x)\ y) \\ \qquad\qquad\qquad else\ g_x^1\ ((z_0, z_1, z_2)\ pred(x)))\ z) \end{array}$$

where

$$\begin{array}{l} g_z^1 \stackrel{\text{def}}{=} \lambda(z_0, z_1, z_2). \\ \quad (\ if\ z = 0 \\ \quad\ then\ \lambda m.(m, 0, 1) \\ \quad\ else\ \lambda m.\ (\ (z_0\ (m\ mod\ z)\ z), (z_2\ (m\ mod\ z)\ z), \\ \qquad\qquad (z_1\ (m\ mod\ z)\ z) \\ \qquad\qquad +(z_2\ (m\ mod\ z)\ z) \cdot \frac{m - (m\ mod\ z)}{z})) \end{array}$$

4.4 Extended Projection Method

The proof trees are a clear description of the logical meaning of programs, so that the analysis to detect the redundancy of realiser codes is much easier to perform if it is performed at the proof tree level.

The realiser of a formula, A, is a sequence of codes of length $l(A)$ as defined in the last section. However, not all the elements of the sequence are always necessary. In addition, it is generally

difficult to determine automatically which part of the realiser code is really necessary, so that it is necessary for end users to specify which elements of the realiser codes of each node are needed, but at the same time, it is preferable to limit the information that end users should specify.

On the other hand, the proof compiler performs realisability interpretation. It analyses a given proof tree from bottom to top, extracting the code step by step for the inference rule of each application in the proof tree, so that if the information is given in the end-formula, the information can be inherited from bottom to top of the proof tree being reformed according to the inference rule of each application. The proof compiler uses the information to refrain from generating code that is not necessary. Consequently, end users may not specify the nodes in the proof tree about the redundancy; it suffices to specify them only in the conclusion of the proof.

4.4.1 Proof theoretic terminology and notation

Definition 18: *Application and node*
In the following proof tree,

$$\dfrac{\dfrac{\Sigma_0}{A_0}\cdots\dfrac{\Sigma_n}{A_n}}{\dfrac{B}{\Pi}}(R)$$

the formula occurrences, A and B, are called *nodes*, and the $\dfrac{A}{B}(R)$ part is called the *application* of rule R, or the *R application*.

Definition 19: *Subtree*
If A is a formula occurrence in proof tree Π, *the subtree of Π determined by A* is the proof tree obtained from Π by removing all formula occurrences except A and the ones above A.

Definition 20: *Side-connected*
Let A be a formula occurrence in Π, let $(\Pi_0, \Pi_1, \cdots, \Pi_{n-1}/A)$ be the subtree of Π determined by A, and let $A_0, A_1, \cdots, A_{n-1}$ be the end formulae of $\Pi_0, \Pi_1, \cdots, \Pi_{n-1}$ respectively. Then, A_i is said to be *side-connected* with A_j $(0 \leq i, j < n)$.

Definition 21: *Top and end-formula*
(1) A *top-formula* in a formula tree, Π, is a formula occurrence that does not stand immediately below any formula occurrence in Π.
(2) An *end-formula* of Π is a formula occurrence in Π that does not stand immediately above any formula occurrence in Π.

Definition 22: *Minor and major-premise*
In the following rules, C, C_0, C_1 and C_2 are said to be *minor premises*. A premise that is not minor is a *major premise*.

$$\dfrac{C \quad C \supset B}{B}(\supset\text{-}E) \qquad \dfrac{\exists x.\, A(x) \qquad \overset{[A(x)]}{C}}{C}(\exists\text{-}E)$$

$$\dfrac{A \vee B \quad \overset{[A]}{C_0} \quad \overset{[B]}{C_1}}{C_2}(\vee\text{-}E) \qquad C_0, C_1, C_2 \text{ are of the same form.}$$

C_0 is called the *left minor premise*, and C_1 is called the *right minor premise*.

Definition 23: *Cut*

- An application of $(\supset\text{-}I)$ succeeded by an application of $(\supset\text{-}E)$ is called a *cut*.

$$\dfrac{\dfrac{\Sigma_0}{B} \quad \dfrac{\overset{[B]}{\Sigma_1}}{B \supset A}(\supset\text{-}I)}{A}(\supset\text{-}E)$$

4.4.2 Declaration to specifications

Definition 24: *Declaration*

(1) A *declaration* of a specification, A, is the finite set, I, of offsets of $Rv(A)$. It is a subset of the set of natural numbers totally ordered by $\leq$. A specification, A, with the declaration, I, is denoted $\{A\}_I$. Elements of the declaration are called *marking numbers*.

(2) The empty set, ϕ, is called the *nil declaration*.

(3) The declaration, $\{0, 1, \cdots, l(A) - 1\}$, is called *trivial*.

The declaration indicates which values of the existentially quantified variables of a given theorem are needed. It is the only information that end users of the system need to specify; the other part is performed automatically. Suppose, for simplicity, that the given theorem is of the following canonical form:

$$\forall x_0. \cdots \forall x_{m-1}. \exists y_0. \cdots \exists y_{n-1}. A(x_0, \cdots, x_{m-1}, y_0, \cdots, y_{n-1}),$$

and the values of $y_0, \cdots, y_k$, $0 \leq k \leq n-1$, are needed. It is declared with the set of the positions:

$$\{0, \cdots, k\}$$

Example:

$A \stackrel{\text{def}}{=} \forall x.\ (3 \leq x \supset \forall y. \exists z. \exists w.\ x = y \cdot z + w)$ a specification of division of natural numbers more than 3. $Rv(A) = \{z_0, z_1\}$, where z_0 corresponds to z and z_1 to w. If the function that calculates the remainder of division of x by y is needed, the declaration of A is $\{1\}$.

The following restriction assures a sort of soundness.

Restriction: The marking numbers of a declarations cannot specify realizing variables of more than two subformulae of the specification which are separated by $\wedge$. For example, if the specification is of the form $A \wedge B$ and $l(A) = 2$ and $l(B) = 3$, marking such as $\{0, 3\}$ is thought to be illegal because 0 specifies a variable in $Rv(A)$ and 3 specifies a variable in $Rv(B)$.

4.4.3 Marking

Definition 25: *Marking*

(1) *Marking* of a node, A, in a proof tree, Π, is the finite set, I, of offsets of $Rv(A)$. It is a subset of the set of natural numbers totally ordered by $\leq$. A node, A, with the marking, I, is denoted $\{A\}_I$. Elements of the marking are called *marking numbers*.

(2) The empty set, ϕ, is called *nil marking*.

(3) The marking, $\{0, 1, \cdots, l(A) - 1\}$, is called *trivial marking*.

Note that declaration is a special case of marking; the marking of the end-formula of the proof tree is the declaration.

Marking means to attach to each node of given proof trees the information that indicates which codes among the realiser sequence of a given formula are needed. The marking can be determined according to the inference rule of each node and the declaration. For example, let $\forall x.\ \exists y.\ \exists z.\ A(x,y,z)$ be the specification of a program and a function from x to y, and z be the expected code from the proof of this specification. Assume the proof to be as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{s}(*) \quad \dfrac{\dfrac{}{t}(*) \quad \dfrac{[x]}{\dfrac{\Sigma}{A(x,s,t)}}}{\exists z.\ A(x,s,z)}(\exists\text{-}I)}{\exists y.\ \exists z.\ A(x,y,z)}(\exists\text{-}I)}{\forall x.\ \exists y.\ \exists z.\ A(x,y,z)}(\forall\text{-}I)$$

The code extracted by Ext is

$$\lambda x.\ (s, t, Ext(A(x,s,t))$$

or equivalently

$$(\lambda x.s,\ \lambda x.t,\ \lambda x.Ext(A(x,s,t))).$$

However, only the 0th and 1st codes are needed here, so that the declaration is $\{0,1\}$. The marking of $\exists y.\exists z.\ A(x,y,z)$, $\{0,1\}$, is determined according to the inference rule $(\forall\text{-}I)$ and the declaration. For the node, $\exists z.\ A(x,s,z)$, the 0th code of the realiser sequence is the 1st code of $\exists y.\exists z.A(x,y,z)$, so that the marking is $\{1\}$. For $A(x,s,t)$, no realiser code is necessary here, so that the marking is ϕ. t and s should also be marked by $\{0\}$, which indicates that s and t themselves are necessary. Consequently, the following tree is obtained:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{s\}}(*)_{\{0\}} \quad \dfrac{\dfrac{}{\{t\}}(*)_{\{0\}} \quad \dfrac{[x]}{\dfrac{\Sigma}{\{A(x,s,t)\}_{\phi}}}}{\{\exists z.\ A(x,s,z)\}_{\{1\}}}(\exists\text{-}I)}{\{\exists y.\ \exists z.\ A(x,y,z)\}_{\{0,1\}}}(\exists\text{-}I)}{\{\forall x.\ \exists y.\ \exists z.\ A(x,y,z)\}_{\{0,1\}}}(\forall\text{-}I)$$

Definition 26: *Marked proof tree*
A *marked proof tree* is a tree obtained from a proof tree and the declaration by the marking procedure.

The marking procedure continues from the bottom of proof trees to the tops. The proof compilation procedure, Ext, should be modified to take marked proof trees as inputs and extract part of the realiser code according to the marking. It will be defined later. The formal definition of the marking procedure, called $Mark$, is given in [Takayama 88b], but here, part of the definition will be given rather informally to make the idea clearer.

(1) Marking of the $(\exists\text{-}I)$ application

By definition, the 0th code of

$$Ext\left(\frac{\overset{-}{t}^{(*)} \quad \dfrac{\Sigma}{A(t)}}{\exists x.A(x)}(\exists\text{-}I)\right)$$

is the term which is the value of x bound by $\exists$. Let I be the marking of the conclusion, then t should be marked $\{0\}$ if $0 \in I$, otherwise the marking is ϕ. The marking of $A(t)$ is given as all marking numbers in I except 0. However, note that the ith code $(0 < i)$ of $\exists x.A(x)$ corresponds to the $i-1$th code of $A(t)$. Consequently, the marking of $A(t)$ is $(I - \{0\}) - 1$ where, for any finite set of natural numbers, K, and any natural number, n, $K - n \stackrel{\text{def}}{=} \{a - n | a \in K \wedge n \leq a\}$.

(2) Marking of the $(\exists\text{-}E)$ application

By the definition of the Ext procedure, the realiser code of C concluded by the following inference is obtained by instantiating the code from the subtree determined by the minor premise by the code from the subtree determined by the major premise:

$$\frac{\dfrac{\Sigma_0}{\exists x.\ A(x)} \qquad \dfrac{[x, A(x)]}{\dfrac{\Sigma_1}{C}}}{C}(\exists\text{-}E)$$

where $A(x)$ contains x as free variables.
Hence, both the marking of C as the conclusion of the above tree and the marking of C as the minor premise are the same. The marking of the subtree determined by the minor premise can be performed inductively. Let J and K be the union of the marking of all occurrences of the two hypotheses, x and $A(x)$. Note that J is either $\{0\}$ or ϕ.

$$\frac{\dfrac{\Sigma_0}{\exists x.\ A(x)} \qquad \dfrac{[\{x\}_J, \{A(x)\}_K]}{\dfrac{\Sigma_1}{\{C\}_I}}}{\{C\}_I}(\exists\text{-}E)$$

The marking of the subtree determined by the major premise is as follows:
Case 1: $J = \{0\}$
This means that the following reasoning is contained in the subtree determined by the minor premise:

$$\frac{(x) \quad P(x)}{\exists y.\ P(y)}(\exists\text{-}I)$$

and the marking of (x) is $\{0\}$, so that x should be extracted from the proof tree determined by the minor premise, C. Consequently, the 0th element of the sequence of realiser codes of $\exists x.\ A(x)$, which is the value of x in $A(x)$, is necessary to instantiate the code from the subtree determined by the minor premise, so that the marking is:

$$\frac{\Sigma_0}{\{\exists x.\ A(x)\}_{\{0\} \cup (K+1)}}$$

Case 2: $J = \phi$
This means that the value of x is not necessary to instantiate the code from the subtree determined by the minor premise, so that the marking is:

$$\frac{\Sigma_0}{\{\exists x.\ A(x)\}_{K+1}}$$

(3) Marking of the ($\vee$-E) application

The realiser code of C concluded by the following inference

$$\dfrac{\dfrac{\Sigma_0}{A \vee B} \quad \dfrac{[A]}{\dfrac{\Sigma_1}{C}} \quad \dfrac{[B]}{\dfrac{\Sigma_2}{C}}}{C}(\vee\text{-}E)$$

is an *if* T_0 *then* T_1 *else* T_2 type code where T_1 and T_2 are sequences of the same length (because both are the codes of C), so that C as the conclusion and two Cs as minor premises should have the same marking. T_1 and T_2 are obtained by instantiating $Rv(A)$ and $Rv(B)$ in the code extracted from the subtrees determined by the minor premise. The code extracted from the subtree determined by the major premise is used both to make T_0 and for the instantiation of $Rv(A)$ and $Rv(B)$. Let I be the marking of the conclusion, then the marking of the subtrees determined by the minor premises can be determined inductively. Let J_0 and J_1 be the unions of markings of all As and Bs as hypotheses:

$$\dfrac{\dfrac{\Sigma_0}{A \vee B} \quad \dfrac{\{[A]\}_{J_0}}{\dfrac{\Sigma_1}{\{C\}_I}} \quad \dfrac{\{[B]\}_{J_1}}{\dfrac{\Sigma_2}{\{C\}_I}}}{\{C\}_I}(\vee\text{-}E)$$

The marking of the subtree determined by $A \vee B$ is as follows:

Case 1: $I = \phi$

This means that it is not necessary to extract any code from this proof tree, so that, of course, no code from the subtree is necessary:

$$\dfrac{\Sigma_0}{\{A \vee B\}_\phi}$$

Case 2: $I \neq \phi$

Code T_0 is the decision procedure that decides which formula in A and B actually holds. This is obtained in the 0th code of the sequence of realiser codes of the subtree determined by $A \vee B$. Also, the codes to be assigned to $\{[A]\}_{J_0}$ and $\{[B]\}_{J_1}$ are obtained in the remainder of the code from the subtree, so that the marking is:

$$\dfrac{\Sigma_0}{\{A \vee B\}_{\{0\} \cup J_0' \cup J_1'}}$$

where $J_0' = J_0 + 1$ and $J_1' = J_1 + l(A)$.

(4) Marking of the ($\supset$-E) rule

The realiser code of $A \supset B$ is of the following form:

$$\lambda \overline{x}.\ (t_0, \cdots, t_k) \equiv (\lambda \overline{x}.t_0, \cdots, \lambda \overline{x}.t_k)$$

and $(t_0, \cdots, t_k)$ is the code of $A \supset B$ which contains the variable sequence $\overline{x}(= Rv(A))$ as free variables, so that the length of the code from $A \supset B$ is the same as that of B. Let I be the marking of the conclusion. Then, the marking of $A \supset B$ should also be I:

$$\dfrac{\dfrac{\Sigma_0}{A} \quad \dfrac{\Sigma_1}{\{A \supset B\}_I}}{\{B\}_I}(\supset\text{-}E)$$

The marking of the subtree determined by A is as follows.
Case 1: The application of $(\supset\text{-}E)$ is part of the cut:
The realiser code of A as the minor premise is restricted by the marking of A as a hypothesis of the subtree determined by $A \supset B$. Let I be the marking of B, and let J be the union of the marking of As as a hypothesis:

$$\dfrac{\Pi_0 \quad \dfrac{\{[A]\}_J \quad \dfrac{\Sigma_1}{B}}{\{A \supset B\}_I}(\supset\text{-}I)}{\{B\}_I}(\supset\text{-}E)$$

Hence, the marking of the subtree is:

$$\frac{\Sigma_0}{\{A\}_J}$$

Case 2: Cut-free proof
The marking of $A \supset B$ restricts only the length of output sequence $\lambda\overline{x}.\ (t_0, \cdots, t_k)$, and, for the input, all the values of the variable sequence $\overline{x}$ are necessary. Specifically, it may happen that some variables in $\overline{x}$ are not used in a particular output subsequence, $\lambda\overline{x}.(t_{i_0}, \cdots, t_{i_l})$, $\{t_{i_0}, \cdots, t_{i_l}\} \subset \{t_0, \cdots, t_k\}$. These redundant variables cannot be detected by the proof theoretic method. However, this cannot always be seen as redundancy; $\lambda(x, y).x$ and $\lambda x.x$ is to be seen as a different function. Consequently, the marking of the subtree determined by the minor premise is trivial.

4.4.4 Critical applications

(1) Induction hypothesis and marking

The programs extracted from induction proofs are recursive call programs. For simplicity, it is assumed in the following description that induction steps are proved without any application of another induction, and induction always means mathematical induction here. If the recursive call program, f, extracted from the induction proof

$$\dfrac{\dfrac{\Sigma_0}{A(0)} \quad \dfrac{[A(x)] \quad \Sigma_1}{A(x+1)}}{\forall x.A(x)}(nat\text{-}ind)$$

is a program that calculates a sequence of terms of length $n(= l(\forall x.A(x)))$, every recursive call of f must calculate the sequence of realiser codes of the same positions, so that the marking of not only $A(0)$, $A(x+1)$ (conclusion of the induction step) and $\forall x.A(x)$ but also $A(x)$ (induction hypothesis) should be the same. This raises a question: are the markings of $A(x+1)$ (conclusion of the induction step) and $A(x)$ (hypothesis of induction) by the $Mark$ procedure always the same? Actually, if the $(\vee\text{-}E)$, $(\exists\text{-}E)$, $(\supset\text{-}E)$ and $(\wedge\text{-}I\&E)$ rules are used in the proof of induction step, the answer is not always affirmative.
The rest of this section is dedicated to the analysis of these critical applications of the rules.

(2) Critical $(\vee\text{-}E)$ and $(\exists\text{-}E)$ applications

Let $A(x) \stackrel{\text{def}}{=} \exists x : nat.\ B(x) \vee C(x)$ where $B(x)$ and $C(x)$ are some formulae with x as free variables. Suppose that $\forall x : nat.\ A(x)$ is proved by mathematical induction, and the induction step proceeds as follows. $\exists x.\ B(x) \vee C(x)$ is the induction hypothesis.

$$\dfrac{[\exists x.\ B(x) \vee C(x)] \quad \dfrac{[B(x) \vee C(x)] \quad \dfrac{\begin{matrix}[x]\\ [B(x)]\\ \Sigma_0\end{matrix}}{A(x+1)} \quad \dfrac{\begin{matrix}[x]\\ [C(x)]\\ \Sigma_1\end{matrix}}{A(x+1)}}{A(x+1)}(\vee\text{-}E)}{A(x+1)}(\exists\text{-}E)$$

If the declaration of $\forall x.\ A(x)$ is $\{0\}$, the marked proof tree is as follows:

$$\dfrac{\{[\exists x.\ B(x) \vee C(x)]\}_L \quad \dfrac{\{[B(x) \vee C(x)]\}_K \quad \dfrac{\{[x]\}_P\{[B(x)]\}_I}{\{A(x+1)\}_{\{0\}}} \quad \dfrac{\{x\}_Q\{[C(x)]\}_J}{\{A(x+1)\}_{\{0\}}}}{\{A(x+1)\}_{\{0\}}}(\vee\text{-}E)}{\{A(x+1)\}_{\{0\}}}(\exists\text{-}E)$$

where Σ_{00} and Σ_{11} are the suitably marked versions of Σ_0 and Σ_1. I and J are the union of the markings of $B(x)$ and $C(x)$, and P and Q are the union of the markings of x as hypotheses. Note that P and Q are either $\{0\}$ or ϕ. Then K and L are as follows:

Case 1: $P \cup Q = \{0\}$

$$K = \{0\} \cup (I+1) \cup (J + l(B(x))$$
$$L = \{0\} \cup (K+1) = \{0,1\} \cup (I+2) \cup (J + l(B(x)) + 1)$$

Case 2: $P \cup Q = \phi$

$$K = \{0\} \cup (I+1) \cup (J + l(B(x))$$
$$L = K + 1 = \{1\} \cup (I+2) \cup (J + l(B(x)) + 1)$$

On the other hand, because $\exists x.\ B(x) \vee C(x)$ is the induction hypothesis, it should have the same marking as $\forall x.\ A(x)$, i.e., $\{0\}$. However, the marking of the induction hypothesis, L, contains a 1 that is not contained in the marking of $\forall x.\ A(x)$. This indicates the fact that it is necessary to specify more codes in the realiser sequences than one expects when $(\vee\text{-}E)$ and $(\exists\text{-}E)$ is used below the deduction sequence down from the induction hypotheses.
The reason for this phenomenon is that the realiser code of $A \vee B$ consists not only of the code of A and B but also of the *left* or *right* code, so that the marking of $A \vee B$ must contain 0 except in a few special situations. A similar thing can be said about the marking of $\exists x.A(x)$ type formulae.

Definition 27: *Thread*
Let $S \stackrel{\text{def}}{=} (A_1, A_2, \cdots, A_n)$ be a sequence of proof occurrences in a formula tree, Π. Then S is a *thread* iff
(1) A_1 is a top-formula in Π;
(2) A_i stands immediately above A_{i+1} in Π for each $i < n$;
(3) A_n is the end-formula of Π.

Definition 28: *Segment*
Let $S \stackrel{\text{def}}{=} (A_1, A_2, \cdots, A_n)$ be a sequence of consecutive formula occurrences in a thread in a

proof tree, Π. Then S is a *segment* iff
(1) A_1 is not a conclusion of the application of ($\vee$-E) or ($\exists$-E).
(2) For arbitrary i ($< n$), A_i is a minor premise of an application of ($\vee$-E) or ($\exists$-E).
(3) A_n is not a minor premise of any application of ($\vee$-E) or ($\exists$-E).

Note that all formula occurrences in a segment are of the same form. Any formula occurrence A in a proof tree Π that is not a conclusion or a minor premise of the application of ($\vee$-E) or ($\exists$-E) is a segment by (1) and (3) of the definition. This kind of segment will be called a *trivial segment* in the following description.

Definition 29: *Major premise attached to a formula*
The major premise of the application of ($\vee$-E) or ($\exists$-E) that is side-connected with a formula A in a segment is, if it exists, called the *major premise attached to* A.

Definition 30: *Proper segment*
The segment in a marked proof tree Π is called *proper* iff every formula occurrence in the segment has non-trivial marking.

Definition 31: *Path*
Let $S \stackrel{\text{def}}{=} (A_1, A_2, \cdots, A_n)$ be a sequence in a deduction, Π. S is a *path* iff
(1) A_1 is a top-formula in Π that is not discharged by an application of ($\vee$-E) and ($\exists$-E).
(2) A_i, for each $i < n$, is not the minor premise of an application of ($\supset$-E), and either (a) A_i is not the major premise of ($\vee$-E) or ($\exists$-E), and A_{i+1} is the formula occurrence immediately below A_i, or (b) A_i is the major premise of an application of ($\vee$-E) or ($\exists$-E), and A_{i+1} is an assumption discharged by the application in Π.
(3) A_n is either a minor premise of ($\supset$-E), the end-formula of Π, or a major premise of an application of ($\vee$-E) or ($\exists$-E) such that any assumptions are not discharged by the application.

Definition 32: *Main path*
The *main path* in a proof tree, Π, is the path whose last formula is the end-formula of Π.

Definition 33: *Critical segment*
Let Π be a subtree of the induction step proof in a proof tree in induction. A proper segment, σ, in Π is *critical* iff there is a formula occurrence, A, in σ such that the major premise, B, attached to A is a formula occurrence in one of the main paths of Π from the induction hypothesis.

(3) Critical ($\supset$-E) applications

Suppose that the induction hypothesis is used as a hypothesis above a minor premise of ($\supset$-E) and the proof is cut-free:

$$\dfrac{\dfrac{[A(x)]}{\dfrac{\Sigma_0}{B} \quad \dfrac{\Sigma_1}{B \supset C}(\supset\text{-}E)}{C}}{\begin{matrix}\Pi \\ A(x+1)\end{matrix}}$$

Then the marking of B is trivial, so that $[A(x)]$ has trivial marking. In this case, the correspondence between the markings of induction hypotheses and the conclusions of the induction step holds only if the marking of $A(x+1)$ is trivial.

Definition 34: *Critical* ($\supset$-E) *application*
If there is a path from the induction hypothesis to a minor premise, A, of an application of ($\supset$-E), A is called the *critical* ($\supset$-E) *premise*, and the application is called the *critical* ($\supset$-E) *application.*

(4) Critical ($\wedge$-I&E) applications

Assume that the induction hypothesis is of the form $A \wedge B$ and the end-formula of the proof is $A' \wedge B'$. A and A' are of the same construction and differ at most in some atomic formulae. B and B' are of the same relation. Assume that the proof is as follows:

$$\dfrac{\dfrac{\dfrac{[A \wedge B]}{A}}{\begin{matrix}\Pi_0 \\ A'\end{matrix}} \quad \begin{matrix}\Sigma_1 \\ B'\end{matrix}}{A' \wedge B'}$$

Let I be the non-nil marking of $A' \wedge B'$, and assume that $\{a | a \in I \wedge l(A') \leq a\} = I$. Then the marking of A' is ϕ so that the marking of the induction hypothesis, $A \wedge B$, is also ϕ, i.e., different from I. This situation is problematic in terms of the correspondence of markings of induction hypotheses and conclusions of the induction steps. The restriction on declarations in Section 4.4.2 prevents this sort of situation.

(5) Soundness of the marking procedure

Theorem 3: [Takayama 88b]
Suppose that a formula, $\forall x.A(x)$, *is proved by mathematical induction, and that* I *is an arbitrary declaration of the conclusion. Let* Π *be a normal deduction of* $A(x) \vdash A(x+1)$, *and assume that there is no critical* ($\wedge$-I&E) *application in* Π:

$$\dfrac{A(0) \quad \dfrac{\dfrac{[A(x)]}{\Sigma}}{A(x+1)}}{\forall x.\ A(x)}(\textit{nat-ind})$$

(1) If Π *has a critical* ($\supset$-E) *application in one of the main paths from the induction hypothesis,* $[A(x)]$, *its marking is nil.*
(2) If Π *has no critical* ($\supset$-E) *application or critical segment, the marking of the induction hypothesis by Mark,* $[A(x)]$, *is trivial.*
(3) Otherwise, the marking of $[A(x)]$ *is* I.

According to the theorem, the declaration of the conclusion is as follows:
Case 1: If the proof tree of the induction step has the critical ($\supset$-E) application in one of the main paths from the induction hypothesis, the declaration must be trivial.
Case 2: If the proof tree of the induction step has no critical ($\supset$-E) application or critical segment, the declaration may be arbitrary.
Case 3: If the proof tree of the induction step has no critical ($\supset$-E) application but has critical segments, the declaration must be enlarged to eliminate critical segments. In this case, the marking of the induction hypothesis, S, and the initial declaration are different, so that the declaration should be same as S and the marking be performed again.

4.4.5 Modified proof compilation: $NExt$

The proof compilation should be modified to handle marked proof trees. The chief modifications are:

1) if the given formula, A, is marked by $\{i_0, \cdots, i_k\}$, extract the code for the i_lth $(0 \leq l \leq k)$ realizing variable in $Rv(A)$;
2) if formula A is marked by ϕ, no code should be extracted and there is no need to analyse the subtree determined by A;
3) if formula A is trivially marked, all the codes for $Rv(A)$ should be extracted.

The following is the definition of the modified version of the Ext procedure, $NExt$.

(1) Assumptions:

$NExt(\{[A]\}_I) \stackrel{\text{def}}{=} proj(I)(Rv(A))$

(2) Atomic formulae or axioms:

$$NExt\left(\frac{\{A_0\}_{J_0} \cdots \{A_k\}_{J_k}}{\{B\}_I}(Rule)\right) \stackrel{\text{def}}{=} nil$$

where B is an atomic formula or an axiom

(3) $\wedge$ and $\vee$ formulae:

- $$NExt\left(\frac{\dfrac{\Sigma_0}{\{A_0\}_{I_0}} \cdots \dfrac{\Sigma_{n-1}}{\{A_{n-1}\}_{I_{n-1}}}}{\{A_0 \wedge \cdots \wedge A_{n-1}\}_I}(\wedge\text{-}I)\right) \stackrel{\text{def}}{=} (NExt\left(\frac{\Sigma_0}{\{A_0\}_{I_0}}\right), \cdots NExt\left(\frac{\Sigma_{n-1}}{\{A_{n-1}\}_{I_{n-1}}}\right))$$

Note that if $I_i = \phi$, $NExt\left(\dfrac{\Sigma_i}{\{A_i\}_{I_i}}\right) = (nil) \quad i = 0 \cdots 1$.

- $$NExt\left(\frac{\dfrac{\Sigma}{\{A_0 \wedge \cdots \wedge A_{n-1}\}_J}}{\{A_i\}_I}(\wedge\text{-}E)\right) \stackrel{\text{def}}{=} NExt\left(\frac{\Sigma}{\{A_0 \wedge \cdots \wedge A_{n-1}\}_J}\right)$$

where $i = 0 \cdots n-1$.

- $$NExt\left(\frac{\dfrac{\Sigma}{\{A\}_J}}{\{A \vee B\}_I}(\vee\text{-}I)\right) \stackrel{\text{def}}{=} \begin{cases} (left, NExt\left(\dfrac{\Sigma}{\{A\}_J}\right), any[k]) & \text{if } 0 \in I \\ (NExt\left(\dfrac{\Sigma}{\{A\}_J}\right), any[l]) & \text{if } 0 \notin I \end{cases}$$

- $$NExt\left(\frac{\dfrac{\Sigma}{\{B\}_J}}{\{A \vee B\}_I}(\vee\text{-}I)\right) \stackrel{\text{def}}{=} \begin{cases} (right, any[k], NExt\left(\dfrac{\Sigma}{\{B\}_J}\right)) & \text{if } 0 \in I \\ (any[l], NExt\left(\dfrac{\Sigma}{\{B\}_J}\right)) & \text{if } 0 \notin I \end{cases}$$

where $k = \mid I \mid -(1 + \mid J \mid)$ and $l = \mid I \mid - \mid J \mid$.

(4) The code from the ($\vee$-E) rule:

$$NExt\left(\dfrac{\dfrac{\Sigma_0}{\{A \vee B\}_{J_0}} \quad \dfrac{\{[A]\}_{J_1}}{\dfrac{\Sigma_1}{\{C\}_I}} \quad \dfrac{\{[B]\}_{J_2}}{\dfrac{\Sigma_2}{\{C\}_I}}}{\{C\}_I}(\vee\text{-}E)\right)$$

is as follows:

a) $if\ A\ then\ NExt\left(\dfrac{\{[A]\}_{J_1}}{\dfrac{\Sigma_1}{\{C\}_I}}\right)\ else\ NExt\left(\dfrac{\{[B]\}_{J_2}}{\dfrac{\Sigma_2}{\{C\}_I}}\right)$ **[modified $\vee$ code]**

when both A and B are equations or inequations of terms
Note that, in this case, $J_1 = J_2 = \phi$.

b) $if\ left = proj(0)\left(NExt\left(\dfrac{\Sigma_0}{\{A \vee B\}_{J_0}}\right)\right)\ then\ NExt\left(\dfrac{\{[A]\}_{J_1}}{\dfrac{\Sigma_1}{\{C\}_I}}\right)\theta\ else\ NExt\left(\dfrac{\{[B]\}_{J_2}}{\dfrac{\Sigma_2}{\{C\}_I}}\right)\theta$

otherwise

where $\theta \stackrel{\text{def}}{=} \left\{\begin{array}{c} proj(J_1)(Rv(A))/proj(J_0+1)\left(NExt\left(\dfrac{\Sigma_0}{\{A \vee B\}_{J_0}}\right)\right), \\ proj(J_2)(Rv(B))/proj(J_1 + (1+ \mid J_0 \mid))\left(NExt\left(\dfrac{\Sigma_0}{\{A \vee B\}_{J_0}}\right)\right) \end{array}\right\}$

(5) The codes from the ($\supset$-I) and ($\forall$-I) rules:

- $NExt\left(\dfrac{\dfrac{[x:\sigma]}{\dfrac{\Sigma}{\{A(x)\}_I}}}{\{\forall x:\sigma.\ A(x)\}_I}(\forall\text{-}I)\right) \stackrel{\text{def}}{=} \lambda x.\ NExt\left(\dfrac{[x:\sigma]}{\dfrac{\Sigma}{\{A(x)\}_I}}\right)$

- $NExt\left(\dfrac{\dfrac{\{[A]\}_J}{\dfrac{\Sigma}{\{B\}_I}}}{\{A \supset B\}_I}(\supset\text{-}I)\right) \stackrel{\text{def}}{=} \lambda\ proj(J)(Rv(A)).\ NExt\left(\dfrac{\{[A]\}_J}{\dfrac{\Sigma}{\{B\}_I}}\right)$

J is the union of all the markings of A used as assumptions.

(6) The code that is in the form of a function application is extracted from the proofs in ($\supset$-E) and ($\forall$-E):
Note that proofs must be cut-free.

- $NExt\left(\dfrac{\dfrac{\Sigma_0}{A} \quad \dfrac{\Sigma_1}{\{A \supset B\}_I}}{\{B\}_I}(\supset\text{-}E)\right) \stackrel{\text{def}}{=} NExt\left(\dfrac{\Sigma_1}{\{A \supset B\}_I}\right)\left(NExt\left(\dfrac{\Sigma_0}{A}\right)\right)$

$$\bullet\ NExt\left(\frac{\overline{t:\sigma}^{(*)}\quad\dfrac{\Sigma}{\{\forall x:\sigma.\ A(x)\}_I}}{\{A(t)\}_I}(\forall\text{-}E)\right) \stackrel{\text{def}}{=} NExt\left(\frac{\Sigma}{\{\forall x:\sigma.\ A(x)\}_I}\right)(t)$$

(7) The codes from the $(\exists\text{-}I)$ and $(\exists\text{-}E)$ rules:

$$\bullet\ NExt\left(\frac{\overline{\{t:\sigma\}_J}^{(*)}\quad\dfrac{\Sigma}{\{A(t)\}_K}}{\{\exists x:\sigma.\ A(x)\}_I}(\exists\text{-}I)\right) \stackrel{\text{def}}{=} \begin{cases} (t, NExt\left(\dfrac{\Sigma}{\{A(t)\}_K}\right)) & \text{if } J \neq \phi \\ NExt\left(\dfrac{\Sigma}{\{A(t)\}_K}\right) & \text{if } J = \phi \end{cases}$$

$$\bullet\ NExt\left(\frac{\overline{\{\exists x:\sigma.\ A(x)\}_J}^{(*)}\quad\begin{matrix}[\{x:\sigma\}_K,\{A(x)\}_L]\\ \dfrac{\Sigma}{\{C\}_I}\end{matrix}}{\{C\}_I}(\exists\text{-}E)\right)$$

$$\stackrel{\text{def}}{=} NExt\left(\begin{matrix}[\{x:\sigma\}_K,\{A(x)\}_L]\\ \dfrac{\Sigma}{\{C\}_I}\end{matrix}\right)\theta$$

$$\text{where } \theta \stackrel{\text{def}}{=} \left\{\begin{matrix} proj(L)(Rv(A(x)))/tseq(1)\left(NExt\left(\overline{\{\exists x:\sigma.\ A(x)\}_J}^{(*)}\right)\right), \\ x/proj(0)\left(NExt\left(\overline{\{\exists x:\sigma.\ A(x)\}_J}\right)\right)\end{matrix}\right\}.$$

(8) The code extracted from a proof in $(\bot\text{-}E)$ rule:

$$\bullet\ NExt\left(\frac{\Sigma}{\dfrac{\bot}{\{A\}_I}(\bot\text{-}E)}\right) \stackrel{\text{def}}{=} any[k] \qquad \text{where } k = |I|$$

(9) The realiser code extracted from the proof from induction proofs:

$$\bullet\ NExt\left(\frac{\Sigma_0\quad\dfrac{[x:nat,\{A(x)\}_I]}{\{A(succ(x))\}_I}}{\{\forall x:nat.\ A(x)\}_I}(nat\text{-}ind)\right)$$

$$\stackrel{\text{def}}{=} \mu\,\overline{z}.\ \lambda\,x.\ if\ x=0\ then\ NExt\left(\frac{\Sigma_0}{\{A(0)\}_I}\right)\ else\ NExt\left(\frac{[x:nat,\{A(x)\}_I]}{\{A(succ(x))\}_I}\right)\sigma$$

$$\text{where } \overline{z} = proj(I)(Rv(A(x))), \text{and } \sigma = \{\overline{z}/\overline{z}(pred(x)), x/pred(x)\}$$

$$\bullet\ NExt\left(\frac{\dfrac{\Sigma_0}{\{A(NIL)\}_I}\quad\dfrac{[a:nat.x:L(nat),\{A(x)\}_I]}{\dfrac{\Sigma_1}{\{A(cons(a,x))\}_I}}}{\{\forall x:L(nat).\ A(x)\}_I}(L(nat)\text{-}ind)\right)$$

$$\stackrel{\text{def}}{=} \begin{array}{l}\mu\ \overline{z}.\ \lambda\ x.\ if\ x = NIL\ then\ NExt\left(\dfrac{\Sigma_0}{\{A(NIL)\}_I}\right)\\ \qquad else\ NExt\left(\dfrac{[a:nat.x:L(nat),\{A(x)\}_I]}{\dfrac{\Sigma_1}{\{A(cons(a,x))\}_I}}\right)\sigma\end{array}$$

where $\overline{z} = proj(I)(Rv(A(x)))$, and $\sigma = \{\overline{z}/\overline{z}(tl(x)), x/tl(x)\}$

(10) Trivial marking:

$$NExt\left(\frac{A_0\cdots A_k}{B}(Rule)\right) \stackrel{\text{def}}{=} Ext\left(\frac{A_0\cdots A_k}{B}(Rule)\right)$$

The following theorem shows that $Mark$ and $NExt$ can be seen as an extension of the projection function on the extracted codes.

Theorem 4: Soundness of the $NExt$ procedure
Let A be a sentence and D be the declaration. If $\vdash_{\mathbf{QPC}} A$ and Π is its proof tree, then

$$proj(D)(Ext(\Pi)) = NExt(Mark(\Pi))$$

Proof: Straightforward ∎

The following program is extracted by declaring $\{0\}$ to the specification and applying the $NExt$ procedure accompanied by $Mark$ to the proof obtained in the previous section.

$$\begin{array}{l}\lambda\ z.\lambda z_0.\\ \quad g_z^2\ z\ ((\mu z_0.\ \lambda x.\ if\ x = 0\ then\ \lambda y.any[1]\\ \qquad\qquad else\ if\ pred(x) = 1\ then\ \lambda y.\ g_0^2\ \lambda y.any[1]\\ \qquad\qquad\quad else\ \lambda y.\ if\ y < pred(x)\\ \qquad\qquad\qquad\qquad then\ (z_0\ pred(x)\ y)\\ \qquad\qquad\qquad\qquad else\ g_z^2\ (z_0\ pred(x)))\ z)\end{array}$$

where

$$g_z^2 \stackrel{\text{def}}{=} \lambda\ z_0.if\ z = 0\ then\ \lambda m.m\ else\ \lambda m.\ (z_0\ (m\ mod\ z)\ z)$$

5. Conclusion

This paper presented a programming system based on a constructive logic, **QJ**. It is not a coding support system, and the goal of the design is to provide an environment for researchers in which document processing and implementation of the algorithms can be performed in uniform way. A programming logic, QPC, a sugared subset of **QJ**, was defined, and q-realisability

interpretation was reformulated as the program extraction procedure, *Ext*. The document description language, PDL-QJ, which is the mathematics vernacular of QPC, allows us to write in the natural style as seen in ordinary mathematics textbooks. The *Ext* procedure does not always generate efficient executable codes. Three kinds of redundancy in the code extracted by *Ext* were pointed out, and the proof normalization method and the modified V code method were investigated to reduce the redundancy. Finally, a technique to generate redundancy free codes, the extended projection method, was given.

Acknowledgment

Thanks must go to Mr. K. Sakai of ICOT first laboratory with whom I discussed the basic design of the programming environment on the CAP-LA system. My thanks must also go to Dr. C. Mohring and Dr. G. Huet of INRIA, Professor M. Beeson at San Jose State University, Professor J. Bates at Carnegie Mellon University, Mr. S. Goto of NTT Lab., Professor S. Hayashi at Kyoto University, Mr. T. Sakurai at Tokyo Metropolitan University, and Mr. Y. Kameyama at Tohoku University who game me many useful suggestions. I also extend special thanks to Professor T. Itoh and Professor M. Sato at Tohoku University for encouraging me in my work and giving useful suggestions.

REFERENCES

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

[Bates 79] Bates, J.L., "*A logic for correct program development*", Ph.D. Thesis, Cornell University, 1979

[Bates 85] Bates, J. L. and Constable, R., "Proofs as Programs", AMC Transaction on Programming Languages and Systems, Vol. 7, No. 1, 1985

[Beeson 83] Beeson, M.J., "Proving Programs and Programming Proofs", in *Logic, Methodology, and Philosophy of Science VII*, Barcan Marcus, R. and Dorn, G.J.W., Weingartner, eds., North-Holland, Amsterdam, pp.51-82, 1983

[Beeson 85] Beeson, M.J., "*Foundation of constructive mathematics*", Springer-Verlag, 1985

[Buchberger 83] Buchberger, B. "Gröbner bases: An Algorithmic Method in Polynomial Ideal Theory", Technical Report, CAMP-LINZ, 1983

[Chisholm 87] Chisholm, P., "Derivation of Parsing Algorithm in Martin-Löf's Theory of Types", Science of Computer Programming Vol. 8, North-Holland, 1987

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

[Coquand 86] Coquand, T. and Huet, G., "*The Calculus of Constructions*", Rapports de Recherche N° 530, INRIA, 1986

[de Bruijn 80] de Bruijn, N.G., "A Survey of the Project AUTOMATH", in *Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, pp.579-606, 1980

[Feferman 78] Feferman, S., "Constructive theory of functions and classes", in *Logic Colloquium '78*, North-Holland, Amsterdam, pp.159-224, 1978

[Goad 80] Goad, C.A., "*Computational Uses of the Manipulation of Formal Proofs*", Ph.D. Thesis, Stanford University, 1980

[Gordon 79] Gordon, M. J., Milner, A. J., and Wadsworth, C. P., "*Edinburgh LCF*", LICS Vol. 78, 1979

[Goto 79] Goto, S., "Program synthesis through Gödel interpretation", LNCS Vol. 75, Springer-Verlag, 1979

[Goto 85] Goto, S., "Concurrency in Proof Normalization", IJCAI-85, 1985

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

[Hayashi 87] Hayashi, S. and Nakano, H., "*PX: a computational logic*", RIMS-573, RIMS, Kyoto University, 1987

[Hirose 86] Hirose, K., "An Approach to a Proof Checker", LNCS Vol. 233, Springer, 1986

[Howard 80] Howard, W.A., "The formulae-as-types notion of construction", in *Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, pp.479-490, 1980

[Huet 86] Huet, G., "*Formal Structure for Computation and Deduction*", lecture given at CMU, 1986

[Huet 87] Huet, G., "Induction Principle Formalized in the Calculus of Constructions", TAPSOFT'87, LNCS Vol. 250, Springer-Verlag, 1987

[Huet 88] Huet, G., "*A Uniform Approach to Type Theory*, (to appear)

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

[McCarty 84] McCarty, D.C., "*Realizability and Recursive Mathematics*", Ph.D. Thesis, Oxford, 1984

[Martin-Löf 82] Martin-Löf, P., "Constructive mathematics and computer programming", in *Logic, Methodology, and Philosophy of Science VI*, Cohen, L.J. et al, eds., North-Holland, pp.153-179, 1982

[Martin-Löf 84] Martin-Löf, P., "*Intuitionistic Type Theory*", Bibliopolis, Napoli, 1984

[Martin-Löf 85] Martin-Löf, P., "On the meanings of the logical constants and the justifications of the logical laws" , lecture given at Siena, included in Proceedings of the Third Japanese-Swedish Workshop, Institute of New Generation Computing Technology, 1985

[Mohring 86] Mohring-Paulin, C., "Algorithm Development is the Calculus in the Constructions", *Proceedings of Symposium on Logic in Computer Science*, 1986

[Mohring 88] Mohring-Paulin, C., 1988, personal communication

[Nordström 83] Nordström, B. and Petersson, K., "Types and Specifications", Information Processing 83, pp.915-920, 1983

[Nordström 83] Nordström, B. and Petersson, K., "Programming in constructive set theory: some examples", in Proceedings of 1981 Conference on Functional Programming Language and Computer Architecture, pp.141-153, 1983

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

[Sakai 86] Sakai, K., "Toward Mechanization of Mathematics - Proof Checker and Term Rewriting System", France-Japan Artificial Intelligenece and Computer Science Symposium '86, ICOT, 1986, also to appear in *Programming of Future Generation Computers*, ed; Fuchi, K., and Nivat, M., North-Holland, 1988

[Sasaki 86] Sasaki, J., "*Extracting Efficient Code From Constructive Proofs*", Ph.D. Thesis, Cornell University, 1986

[Sato 85] Sato, M., "*Typed Logical Calculus*", Technical Report 85-13, Department of Information Science, Faculty of Science, University of Tokyo, 1985

[Sato 86a] Sato, M., Lecture given at the University of Tokyo, 1986

[Sato 86b] Sato, M., "QJ: A Constructive Logical System with Types", France-Japan Artificial Intelligence and Computer Science Symposium 86, Tokyo, 1986

[Sato 87] Sato, M., "Quty: A Concurrent Language Based on Logic and Function", *Proceedings of the Fourth International Conference*, MIT Press, pp. 1034-1056, 1987

[Smith 82] Smith, J., "The identification of propositions and types in Martin-Löf's type theory: a programming example", LNCS Vol. 158, Springer-Verlag, 1982

[Takayama 87a] Takayama, Y., "Writing Programs as QJ-Proofs and Compiling into PROLOG Programs", *Proceedings of 4th Symposium on Logic Programming*, 1987, also published as Technical Report TR-244, ICOT

[Takayama 87b] Takayama, Y., "On the Extraction of Gröbner Base Algorithm from the Constructive Proof", unpublished manuscript, 1987

[Takayama 87c] Takayama, Y., "On the extraction of matrices calculation programs from constructive proofs", unpublished manuscript, 1987

[Takayama 87d] Takayama, Y., "Proof Parameterization Method in Constructive Logic", Technical Report TR-245, ICOT, 1987

[Takayama 88a] Takayama, Y., "QPC: QJ-based Proof Compiler –Simple Examples and Analysis", *Proceeding of European Symposium on Programming '88* LNCS Vol. 300, Springer-Verlag, 1988, also published as Technical Report TR-296, ICOT

[Takayama 88b] Takayama, Y., "Proof Theoretic Approach to the Extraction of Redundancy-free Realizer Code", Technical Report, ICOT, 1988, (to appear)

[Takayama 88c] Takayama, Y., "Proof Compilation of Induction Proofs in the QPC System", unpublished manuscript, 1988

[Troelstra 73] Troelstra, A.S. "*Mathematical investigations of intuitionistic arithmetic and analysis*", Springer Lecture Notes in Mathematics, Vol. 344, 1973

Appendix: Sample Coding in PDL-QJ

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

function elem:  nat # L(nat) --> nat
   attain
       elem = mu [z].  fun [I, X]. if I = 0 then $abort$
                          else if I = 1 then hd(X0 else z(I-1, tl(X)))
end_function

theorem SIMPLE_EXAMPLE:
  all X:L(nat).  some Y:L(nat).
          (length(X) = length(Y)
          & all I:nat.  (1 =< I =< length(X) -> 2*elem(I, X) = elem(I,Y)))
proof
since induction on 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:nat 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 z:L(nat).
      (HYP_1:  length(X) = length(z)
     & HYP_2:  all I:nat.  (1=<I=<length(X) -> 2*elem(I,X) = elem(I,z))
   length(a.X) = length(X) + 1
   length(2*a.X) = length(X) + 1
   hence length(a.X) = length(2*a.z) by HYP_1
   all I:nat.  (1=<I=<length(a.X) -> 2*elem(I,a.X)=elem(I,2*a.z))
   since
      let I:nat be such that
      1=<I=<length(a.X)
      then 2*elem(I,a.X) = elem(I,2*a.z)
      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.z)
            == if I=0 then $abort$
               else if I=1 then hd(2*a.z) else elem(I-1,tl(2*a.z))
            = hd(2*a.z) = 2*a
            hence 2*elem(I,a.X) = elem(I,2*a.z)
         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,z) by HYP_2
            hence 2*elem(I,a.X) = elem(I-1,z)  by LABEL_1,LABEL_2
            elem(I,2*a.z)
            == if I=0 then $abort$
               else if I-1 then hd(2*a.X) else elem(I-1,tl(2*a.z))
            = elem(I-1,tl(2*a.z)) by assumption
            = elem(I-1,z)
            hence 2*elem(I,a.X) = elem(I,2*a.z)
      end_since
   end_since
   hence concluded
end_since
end_proof
```