ICOT Technical Report: TR-461

TR-461

Extended Projection
— A New Method to Extract Efficient Programs from Constructive Proofs —

by
Y. Takayama

March, 1989

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

Extended Projection
– A New Method to Extract Efficient Programs from Constructive Proofs

Yukihide Takayama

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

Abstract

This paper gives a method to extract redundancy free programs from constructive proofs, using the realizability interpretation. The proof trees are analyzed in the style of program analysis, and they are mechanically translated into trees with additional information, called marked proof trees. The program extractor takes marked proof trees as input, and generates programs in a type-free lambda calculus with sequences.

1. Introduction

Constructive type theories and constructive logics are used in the formal development of functional programs as in Nuprl [Constable 86], the calculus of construction [Coquand 88] [Huet 88], and PX [Hayashi 88]. Programmers formalize the problem or specification as a theorem, and the programming is performed in the style of theorem proving. The correctness of the program is assured by the proof checker in constructive logic, and the theorem and the proof prepared are used as the documentation.

The program extractor is one of the building blocks of the programming environment. It analyzes theorems and their proofs and extracts the programs. The realizability interpretation of logical constants in intuitionistic logic, such as the Curry-Howard isomorphism [Howard 80] and q-realizability [Beeson 85], is used as the basic mechanism of the program extractor. Also, various techniques, such as proof normalization and Harrop formulas, are used to generate efficient programs [Goad 80] [Bates 79] [Sasaki 86]. This paper works on the problem of redundant code, which is one of the main problems in terms of efficient code generation with q-realizability. However, the problem is not always inherent to a particular formulation of realizability.

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-realizability:

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

For example, if the proof is as follows:

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

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, $([x : \sigma_0]/\Sigma_1/A(x, t(x)))$, $t(x)$ denotes a term which contains the variable x, free, and $(t_1, t_2, \cdots)$ means the sequence of terms. In this case, the expected code is:

$$f \stackrel{\text{def}}{=} \lambda x.\ t(x)$$

so that T is redundant.

The traditional solution to this problem has been to introduce suitable constructs to specify which part of the proof is necessary in terms of computation. The set notation is introduced in Nuprl and ITT [Nordström 83]. This is done to skip part of the extraction. [Paulin-Mohring 89] modified the calculus of construction by introducing two kinds of constants, *Prop* and *Spec*. The PX system introduced $\Diamond$-bounded formulas from which no realizer code is extracted. These constructs are obtained by extending the underlying constructive logics, and can also be seen as the control commands to program extractors. However, to extract an efficient program from a proof, the specification and the proof must be translated into a specification and its proof with control commands. This transformation is generally difficult and sometimes impossible to mechanize.

This paper presents a new construct called *marking*. It is independent from the formulation of underlying constructive logic, and allows much more fine-grained specification of redundancy than traditional systems. It also enables mechanization of the transformation of ordinary proofs into proofs with constructs. In other words, the analysis of which part of the proof is actually relevant to the algorithm is performed mechanically by giving a simple declaration to the specification. This method is called the *extended projection method*, EPM for short. EPM can be regarded as a method to analyze functional programs which uses proof trees as the enriched formal description about programs. EPM also enables the extraction of multiple programs from a single proof just by changing the declaration to the specification, letting the programmer select a suitable program.

Section 2 overviews the functional language, Tiny Quty, and the constructive logic in which Tiny Quty programs are developed. It also briefly explains the realizability of the logic. Section 3 introduces the notion of declaration to formal specifications and the marking procedure which analyzes the proof trees to specify the redundancy in them. The crucial part of the proof tree analysis resides in the proofs in induction. Section 4 elaborates on this issue. Section 5 gives the definition of the program extractor. A simple example of a prime number checker program is given in section 6. It demonstrates how the redundancy free program is extracted and how multiple programs are extracted from a single proof. Section 7 works on an analysis of proofs in induction again. It explains the phenomena in section 4 from a proof theoretic viewpoint. Section 8 gives the conclusion.

2. The Language and Constructive Logic

2.1 Tiny Quty

Tiny Quty is a sort of type-free λ-calculus obtained as a sugared subset of Quty [Sato 87]. The only essential difference between it and the standard type-free λ-calculus [Barendregt 81] is that it has sequences of terms as its data structure and a fixed point operator for multi-valued recursive call functions.

(1) Sequence of terms

A sequence of terms is denoted $(t_0, \cdots, t_{n-1})$ or $t_0, \cdots, t_{n-1}$ where t_i is a term, and a term is regarded as a sequence whose length is 1. The nil sequence is denoted (). The concatenation of sequences, $S_0 \cdots S_{k-1}$, is denoted $(S_0, \cdots, S_{k-1})$.

(2) Constants and variables

The language has integers, boolean values, T and F, special constants, $left$ and $right$, and $any[n]$, which denotes the sequence of any constants whose length is n. Variables are denoted in lower case letters: x, y, $\cdots$, and $\overline{x}$, $\overline{y}$ $\cdots$ denote sequences of variables.

(3) Abstraction and application

For a sequence of variables, $\overline{x}$, and a term, M, $\lambda\overline{x}.M$ denotes a lambda abstraction. For two terms, M and N, $M(N)$, or simply $M\ N$, denotes an application.

(4) If-then-else

If A is an equation or inequation of terms, and B and C are terms, then $if\ beval(A)\ then\ B\ else\ C$ is an if-then-else term. $beval$ is a function which evaluates the equation or inequation and returns a boolean value. The term is abbreviated as $if\ A\ then\ B\ else\ C$ in the following description.

(5) Term equivalence

$a)\ \lambda().\ Term \equiv Term$

$b)\ \lambda(x_0, \cdots, x_{n-1}).\ Term \equiv \lambda x_0. \cdots \lambda x_{n-1}.Term$

$c)\ if\ A\ then\ ()\ else\ () \equiv ()$

$d)\ if\ A\ then\ (a_0, \cdots, a_{n-1})\ else\ (b_0, \cdots, b_{n-1})$

$\equiv (if\ A\ then\ a_0\ else\ b_0, \cdots,\ if\ A\ then\ a_{n-1}\ else\ b_{n-1})$

$e)\ \lambda\overline{x}.\ (a_0, \cdots, a_{m-1}) \equiv (\lambda\overline{x}.\ a_0, \cdots, \lambda\overline{x}.\ a_{m-1})$

$f)\ Term\ () \equiv Term$

$g)\ (a_0, \cdots, a_{n-1})(b) \equiv (a_0(b), \cdots, a_{n-1}(b))$

(6) Fixed point operator

If $F(z_0, \cdots, z_{n-1})$ is a term which contains $\overline{z} \stackrel{\text{def}}{=} z_0, \cdots, z_{n-1}$ as free variables, and is equivalent to $(F_0(\overline{z}), \cdots, F_{n-1}(\overline{z}))$, then $\mu(z_0, \cdots, z_{n-1}).F(z_0, \cdots, z_{n-1})$ denotes the solution of the system of the fixed point equation $(z_0, \cdots, z_{n-1}) = (F_0(\overline{z}), \cdots, F_{n-1}(\overline{z}))$. The solution is also described as $(f_0, \cdots, f_{n-1})$ where $f_i \stackrel{\text{def}}{=} \mu z_i.F_i(f_0, \cdots, f_{i-1}, z_i, f_{i+1}, \cdots, f_{n-1})\ i = 0, \cdots, n-1$.

(7) Built-in functions

1) $succ$, $pred$, $+$, $-$, $/$, $\cdot$ $\cdots$ successor/predecessor functions and arithmetic functions

2) $proj(n)$ $\cdots$ strike out the n-th element of a given sequence of terms

3) $proj(I)$ $\cdots$ Let I be a finite sequence of natural numbers. For a sequence of terms, S, of length n $(m \leq n)$, $proj(\{i_0, \cdots, i_{m-1}\})\ (S) \stackrel{\text{def}}{=} (proj(i_0)(S), \cdots, proj(i_{m-1})(S))$

4) $tseq(i)$, $ttseq(i, l)$ are defined as follows. For a sequence, S, with length n

$$tseq(i)(S) \stackrel{\text{def}}{=} (proj(i)(S), proj(i+1)(S), \cdots, proj(n-1)(S))$$
$$ttseq(i,l)(S) \stackrel{\text{def}}{=} (proj(i)(S), proj(i+1)(S), \cdots, proj(i+(l-1))(S))$$

Theorem 1: Every Tiny Quty program is equivalent to a sequence of terms.

2.2 QPC

QPC used here is basically an intuitionistic first order Gentzen style of natural deduction [Prawitz 65] with mathematical induction, higher order equalities and inequality in the sense that functions can be handled, Tiny Quty as its terms and a simple type structure of terms. See [Takayama 88] for detailed definitions.

(1) Type structure

nat and *bool* are used as the primitive types. QPC also has the function type, Cartesian product type, and recursive type in which *nat* and mathematical induction are defined. See [Sato 87] for details of the type structure.

(2) Formulas

$\bot$, equations of terms, $t : \sigma$, where t is a term and σ is a type, are atomic formulas. Inequalities of terms are also regarded as atomic, which is a little different from their standard treatment. Other formulas and subformulas with logical constants, $\forall$, $\exists$, $\supset$, $\wedge$, and $\vee$, are defined in the standard way as in [Prawitz 65]. $t : \sigma$ is often abbreviated to t.

(3) Rules of inference

Besides the I-rules and E-rules of standard natural deduction, QPC has the $\bot$-E rule, mathematical induction, rules on equalities and inequalities of terms, and typed term construction rules.

Here, a few proof theoretic terminologies are defined as follows. See [Prawitz 65] for details.

Definition 1: Top-formula and end-formula

1) A *top-formula* in a proof 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 2: Major premise and minor premise

C, C_0, and C_1 as premises in the following rules of inferences are called *minor premises*. Other premises are called *major premises*.

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

$$\frac{A \vee B \quad \overset{[A]}{C_0} \quad \overset{[B]}{C_1}}{C_2}(\vee\text{-}E) \qquad C_0, C_1, C_2 \text{ are all equal.}$$

C_0 and C_1 are called the *left minor premise* and the *right minor premise*. $\exists x.A(x)$ and $A \vee B$ are called the major premises *connected horizontally to* the minor premises, C, C_0, and C_1.

Definition 3: Subtree determined by a formula
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 those above A.

2.3 q-realizability

The realizability used in this paper is slightly different from standard q-realizability [Beeson 85]. The nil term, (), is attached to an atomic formula as the realizer, while any terms can be used in atomic formulas in standard q-realizability. The realizability used here is essentially the same as that of px-realizability [Hayashi 88] in this respect. See [Sato 85] for details. The algorithmic version of the realizability is formalized as the Ext procedure [Takayama 88]. Ext calculates the realizer of the theorem; in other words, it extracts the Tiny Quty program that satisfies the specification from the proof of the specification. The realizer of a formula is defined as a sequence of terms in Tiny Quty.

3. Declaration and Marking

3.1 Realizing variables, length, and ∃-∨ information of a formula

The sequence of realizing variables of a formula is the sequence of variables to which the realizer of the formula is assigned. It is also used as the realizer of the formula which occurs as an assumption.

Definition 4: $Rv(A)$

(1) $Rv(A) \stackrel{\text{def}}{=} ()$ $\cdots$ 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 fresh variable

(4) $Rv(A \supset B) \stackrel{\text{def}}{=} Rv(B)$

(5) $Rv(\forall x : Type.\ A(x)) \stackrel{\text{def}}{=} Rv(A(x))$

(6) $Rv(\exists x : Type.\ A(x)) \stackrel{\text{def}}{=} (z, Rv(A(x)))$
where z is a fresh variable

Definition 5: Length of a formula
For a formula, A, the length of $Rv(A)$ as a sequence is called the length of the formula and denoted $l(A)$.

Note that the length of the realizer as a sequence of terms is determined according to the form of the formula.

Definition 6: ∃-∨ information
1) For a formula $A \vee B$, the ∨-information is *left* if A holds and *right* if B holds.
2) For a formula, $\exists x.A(x)$, the ∃-information is the term, t, for which $A(t)$ holds.
3) The ∃-∨ information of a formula is the collection of ∨-information and ∃-information of all the subformulas of the formula.

Ext can be roughly seen as the extraction of the ∃-∨ information of the specification, and the realizing variables are the variables to which the ∃-∨ information of all the subformulas of the specification is assigned.

Example 1:
$Rv(\forall x : nat.\ ((x \geq 0) \supset (x = 0 \vee \exists y : nat.\ succ(y) = x))) = (z_0, z_1)$ z_0 is the variable to which the ∨-information of which shows which in $x = 0$ and $\exists y.succ(y) = x$ holds is assigned. z_1 is the variable to which the value, t, for which $succ(t) = x$ holds (∃-information), is assigned.

3.2 Declaration

In the following description, the end-formula of a proof tree is called *specification*.

Definition 7: Declaration
1) For a specification, A, a subset of the finite set of natural numbers, $\{0, 1, \cdots, l(A) - 1\}$, is called the *declaration* to A. A specification with a declaration, I, is denoted $\{A\}_I$. Π_I denotes a proof tree, Π, whose end-formula has the declaration, I. Each element of a declaration is called the *marking number*;
2) The empty set, ϕ, is called *nil marking*;
3) $trivial \stackrel{\text{def}}{=} \{0, 1, \cdots, l(A) - 1\}$ is called *trivial marking*.

A declaration specifies which ∃-∨ information is needed for a given specification with the set of positions of the elements in the realizing variables of the formula. This is the only information which programmers have to give in our method.

Example 2: The following is a specification of the program, f, which tests whether the given natural number is odd or even: $B \stackrel{\text{def}}{=} \forall x.(\exists y.x = 2 \cdot y) \vee (\exists z.x = 2 \cdot z + 1)$. $Rv(B) = \{z_0, z_1, z_2\}$. z_0 is the variable for ∨-information, and z_1 and z_2 are the variables for ∃-information. Therefore, f is the program that calculates a sequence of terms of length 3: the first element is the constant $left$ or $right$ according to whether x is even or odd, and the second and third elements are integers. If one wishes f to return only $left$ or $right$, one should give the declaration, $\{0\}$, to B.

3.3 Marking

For each formula occurrence in a proof tree, information called *marking*, which is similar to the declaration, is attached. The definition of marking is obtained by changing "specification" to "a formula occurrence in a proof tree" in the definition of declaration. The only difference between declaration and marking is that marking can be determined mechanically by giving the declaration to the specification. In fact, the marking of a formula occurrence, A, is determined by the inference rule whose conclusion, B, is the formula occurrence immediately below A, and the marking or declaration of B. The terms "declaration" and "marking" will often be confused in the following description. The procedure is called the *marking procedure*, $Mark$.

The tree obtained by the procedure is called a *marked (proof) tree*. In the following, part of the definition of $Mark$ is given rather informally. It uses the following operations on finite sets of natural numbers:

$I + n \stackrel{\text{def}}{=} \{x + n \mid x \in I\}$ $\quad$ $I - n \stackrel{\text{def}}{=} \{x - n \mid x - n \geq 0, x \in I\}$

$I(< n) \stackrel{\text{def}}{=} \{x \in I \mid x < n\}$ $\quad$ $I(\geq n) \stackrel{\text{def}}{=} \{x \in I \mid x \geq n\}$

(1) Marking of ($\exists$-I) rule application

The 0th term in the following sequence is the term, t, according to the definition of Ext.

$$Ext\left(\frac{\dfrac{\Sigma_0}{t} \quad \dfrac{\Sigma_1}{A(t)}}{\exists x.A(x)}(\exists\text{-}I)\right)$$

Let I be the marking of $\exists x.A(x)$. If $0 \in I$, the marking of t is $\{0\}$, otherwise, ϕ. The marking of $A(t)$ is obtained from $I - \{0\}$. However, the ith ($i > 0$) term of the realizer of $\exists x.A(x)$ is the $i-1$ th term of the realizer of $A(t)$. Therefore, the marking of $A(t)$ is $(I - \{0\}) - 1 = I - 1$.

(2) Marking of ($\exists$-E) rule application

The realizer of the end-formula, C, is obtained by instantiating t and $Rv(A(t))$ in the realizer of the minor premise of the application by the realizer of the major premise of the application:

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

where $A(t)$ contains t free.

The marking of C as the minor premise is equal to that of C as the end-formula. $Mark$ on the subtree determined by the minor premise C is performed recursively. Let J and K be the sums of the markings of all the occurrences of t and $A(t)$, which is called the marking of $[t]$ and $[A(t)]$ in the following description. Note that J is either $\{0\}$ or ϕ. $Mark$ on the subtree determined by the major premise, $\exists x.A(x)$, goes as follows:

Case 1: $J = \{0\}$

The following forms of the subtrees must be contained in the subtree determined by the minor premise, and the marking of [t] is $\{0\}$.

$$\frac{\dfrac{\overset{[t]}{\Sigma_0}}{s} \quad \dfrac{\Sigma_1}{P(s)}}{\exists y.\ P(y)}(\exists\text{-}I) \qquad \text{or} \qquad \frac{\dfrac{\overset{[t]}{\Sigma_0}}{s} \quad \dfrac{\Sigma_1}{\forall z.P(z)}}{P(s)}(\forall\text{-}E)$$

Therefore, the 0th term in the realizer of $\exists x.A(x)$ has to be extracted to instantiate the realizer of the minor premise. Consequently, the marking of $\exists x.A(x)$ is $\{0\} \cup \{K + 1\}$.

Case 2: $J = \phi$

The 0th term in the realizer of $\exists x.A(x)$ is not needed, so that the marking of $\exists x.A(x)$ is $K + 1$.

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

The term *if* T_0 *then* T_1 *else* T_2 is extracted from the following proof tree where T_1 and T_2 are sequences of the same length.

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

The markings of Cs as the left and right minor premises are equal to that of the end-formula. T_1 and T_2 are obtained by instantiating $Rv(A)$ and $Rv(B)$ in the realizers extracted from the subtrees determined by the minor premises by the realizer, $\bar{t}$, of $A \vee B$. Also, the 0th term of $\bar{t}$ is used to make T_0, which is the decision procedure of which of A and B holds. The markings of the subtrees determined by the minor premises are determined recursively. Let J_0 and J_1 be the markings of $[A]$ and $[B]$. Then, the marking of the subtree determined by the major premise, $A \vee B$, is as follows:

Case 1: $I = \phi$

The realizer extracted from the proof tree will be in the following form: $if\ A\ then\ ()\ else\ () \equiv ()$. Therefore, the marking of $A \vee B$ is $\{0\}$.

Case 2: $I \neq \phi$

The 0th term of the realizer of the major premise, $A \vee B$, is used to make T_0. The other part of the realizer is used to make the realizers of $\{[A]\}_{J_0}$ and $\{[B]\}_{J_1}$. Therefore, the marking of $A \vee B$ is $\{0\} \cup (J_0 + 1) \cup (J_1 + 1 + l(A))$.

(4) Marking of $(\supset\text{-}E)$ rule application

The realizer of $A \supset B$ will be in the following form: $\lambda\overline{x}.\ (t_0, \cdots, t_k) \equiv (\lambda\overline{x}.t_0, \cdots, \lambda\overline{x}.t_k)$, where $(t_0, \cdots, t_k)$ is the realizer of B which contains $\overline{x}(= Rv(A))$ free. The marking of $A \supset B$ is equal to B.

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

The marking of the subtree determined by the minor premise, A, is as follows:

Case 1: $R \neq (\supset\text{-}E)$

The marking of $A \supset B$ only specifies the length and the positions of $\lambda\overline{x}.\ (t_0, \cdots, t_k)$, and, for the input of the function which is the realizer of A, there is no restriction and all of the variables in $\overline{x}$ are necessary. Therefore, the marking of A is trivial.

Case 2: $R = (\supset\text{-}E)$

The realizer of the minor premise, A, is restricted by the marking of $[A]$. Let J be the marking of $[A]$, then the marking of A as the major premise is J.

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

4. Marking of Proofs in Induction

4.1 Overflowed and missing marking numbers

The marking procedure for induction proofs is defined as follows:

$$Mark\left(\dfrac{\dfrac{\Sigma_0}{A(0)}\quad\dfrac{[x>0,A(x-1)]}{\dfrac{\Sigma_1}{A(x)}}}{\{\forall x.A(x)\}_I}(ind)\right) \stackrel{\text{def}}{=} \dfrac{Mark\left(\dfrac{\Sigma_0}{\{A(0)\}_I}\right)\quad Mark\left(\dfrac{[x>0,A(x-1)]}{\dfrac{\Sigma_1}{\{A(x)\}_I}}\right)}{\{\forall x.A(x)\}_I}(ind)$$

The realizer extracted from a proof in induction is generally the multi-valued recursive call function which calculates the sequence of terms, $\overline{t}$, with the sequence of the terms, $\overline{s}$, of the same length. The length of the sequences is equal to the length of the conclusion of the proof, the conclusion of the induction step proof, and the induction hypothesis. $\overline{t}$ is the $\exists$-$\vee$ information of the conclusion of the induction step proof, and $\overline{s}$ means the $\exists$-$\vee$ information of the induction hypothesis.

The multi-valued recursive function, f, extracted from an induction proof can be expressed as a sequence of single-valued recursive call functions, $f \equiv (f_0, \cdots, f_{n-1})$, where n is the length of the conclusion of the proof. Therefore, if one gives a declaration, say $I = \{i_0, \cdots, i_{k-1}\}$, to the conclusion of the proof, the program extractor would generate the code $(f_{i_0}, \cdots, f_{i_{k-1}})$. However, it may happen that one of the functions, f_j $(j \in I)$, calls a function, f_l $(l \notin I)$, recursively. This means that the lth $\exists$-$\vee$ information of the induction hypothesis must be used to calculate the jth $\exists$-$\vee$ information of the conclusion of the proof. $Mark$ can check which $\exists$-$\vee$ information in the induction hypothesis is used to construct the $\exists$-$\vee$ information of the conclusion of the induction proof.

Definition 8: Overflowed and missing marking numbers

Let I be a declaration to the conclusion of a proof in induction, and let J be the marking of the induction hypothesis determined by $Mark$.

1) If a marking number, i, is in I but not in J, then i is called a *missing marking number*.

2) If a marking number, j, is in J but not in I, then j is called an *overflowed marking number*.

If marking numbers overflow, the program extractor cannot generate the right program as explained above. However, a missing marking number does not prevent the extraction of the right program. For example, if the declaration contains the marking number, i, the extracted program contains the following code: $\mu z_i.\lambda x.F_i(x, z_i)$. If i is a missing marking number, z_i does not occur in $F_i(x, z_i)$. Consequently, if I is the declaration and J is the marking of the induction hypothesis determined by $Mark$, $J \subseteq I$ must be satisfied for the right program extraction.

4.2 Elimination of overflowed marking numbers

4.2.1 Enlarging the declaration

Basically, an overflowed marking number can be eliminated by enlarging the declaration to the conclusion of the proof. If I is the declaration and S is the set of overflowed marking numbers, then the marking procedure should be performed again with the new declaration, $I \cup S$. It may happen that a set of marking numbers, S', overflows when the declaration is enlarged, so that the procedure may have to be repeated. However, the procedure halts at most with the trivial declaration.

4.2.2 Nested inductions

The elimination procedure of overflowed marking numbers is a little more complex when an induction proof contains other induction proofs and the induction hypothesis of the outer induction is used in the inner induction proofs. For example, assume the following proof:

$$\dfrac{\dfrac{\Sigma_0}{A(0)} \qquad \dfrac{\dfrac{[A(x-1)]}{\Sigma_2} \quad \dfrac{\dfrac{[A(x-1)]}{\Sigma_3}}{B(0)} \quad \dfrac{\dfrac{[A(x-1)][B(y-1)]}{\Sigma_4}}{B(y)}\,(ind)}{\forall y.B(y)} \qquad \dfrac{\Sigma_1}{A(x)}}{\forall x.A(x)}\,(ind)$$

$\forall x.A(x)$ is proved by induction, and $\forall y.B(x)$ is also proved by induction in the proof of $A(x)$. Assume that the declaration, I, is given to $\forall x.A(x)$ and the marked proof tree is as follows:

$$\dfrac{\dfrac{\Sigma_0'}{\{A(0)\}_I} \qquad \dfrac{\dfrac{\{[A(x-1)]\}_{J_0}}{\Sigma_2'} \quad \dfrac{\dfrac{\{[A(x-1)]\}_{J_1}}{\Sigma_3'}}{\{B(0)\}_L} \quad \dfrac{\dfrac{\{[A(x-1)]\}_{J_2}\{[B(y-1)]\}_{LL}}{\Sigma_4'}}{\{B(y)\}_L}\,(ind)}{\{\forall y.B(y)\}_L} \qquad \dfrac{\Sigma_1'}{\{A(x)\}_I}}{\{\forall x.A(x)\}_I}\,(ind)$$

Let $J \stackrel{\text{def}}{=} J_0 \cup J_1 \cup J_2$, then both of the followings must be satisfied:

$$J \subseteq I \qquad \cdots\cdots(1)$$

$$LL \subseteq L \qquad \cdots\cdots(2)$$

If I is enlarged to assure (1), L is generally enlarged so that the procedure to assure (2) needs to be performed on the new L and LL. On the other hand, if L must be enlarged to assure (2), I should be enlarged to do that, then J is also enlarged, so that the procedure to assure (1) needs to be performed on the new values of I and J.

4.2.3 Reverse marking: $Mark^{-1}$

If induction proofs $\Pi_0^{ind}, \cdots, \Pi_{k-1}^{ind}$ occur in another proof, Π, which may also be an induction proof, the declaration to the end-formula of Π should be enlarged to enlarge the markings of Π_i^{ind} $(0 \leq i < k)$. The reverse marking, $Mark^{-1}$, is the procedure which is used in the operation. In other words, $Mark^{-1}$ determines which marking is needed for the consequence of the application of the rule to obtain the specified marking of a premise of the application. $Mark^{-1}$ is defined straightforwardly as a deterministic procedure for most rules of inference. However, non-determinism occurs in the following three cases because, when the marking of the major premise, S, is specified and the marking of the conclusion of the application of each rule must be determined, the marking, K_i, of each occurrence of the assumption discharged by the rule must be determined. In other words, the equation must be solved on finite sets of natural numbers: $SetOp(S) = \Sigma_i K_i$, where $SetOp$ is a suitable set operation.

Case 1:

$$Mark^{-1}\left(\dfrac{\dfrac{\Sigma_0}{\{\exists x.\, A(x)\}_S} \quad \dfrac{[t,\ A(t)]}{\dfrac{\Sigma_1}{C}}}{\Pi}(\exists\text{-}E)\right) \qquad \text{where } S \neq \phi$$

$A(t)$ occurs more than twice as a hypothesis in the subtree determined by C.

Case 2:

$$Mark^{-1}\left(\dfrac{\dfrac{\Sigma_0}{\{A \vee B\}_S} \quad \dfrac{[A]}{\dfrac{\Sigma_1}{C}} \quad \dfrac{[B]}{\dfrac{\Sigma_2}{C}}}{\Pi}(\vee\text{-}E)\right) \qquad \text{where } S \neq \phi \text{ and } S \neq \{0\}$$

Case 3:

$$Mark^{-1}\left(\dfrac{\dfrac{\Sigma_0}{\{A\}_S} \quad \dfrac{\dfrac{[A]}{\dfrac{\Sigma_1}{B}}}{A \supset B}(\supset\text{-}I)}{\Pi}(\supset\text{-}E)\right) \qquad \text{where } S \neq \phi$$

where A occurs as a hypothesis more than twice in the subtree determined by $A \supset B$.

The modified marking algorithm, $MARK$, that makes marked proof trees in which no marking numbers overflow is as follows. Let Π be a proof tree.

1) Perform $Mark(\Pi_I)$. Let Π_I' be the obtained marked proof tree.

2) Search for the subtrees in induction $\Pi_1^{ind}, \cdots, \Pi_k^{ind}$, in which some marking numbers overflow. If induction proofs are nested, the outermost (or innermost) proofs are always picked up. Let

S_i be the set of overflowed marking numbers in Π_i^{ind}, and go to 3). If there is no such subtree in induction, go to 4).

3) Let $I \cup \Sigma_{i=1}^{k} L_i$ be the new declaration to Π, and go to 1). L_i is the marking obtained as follows. Assume:

$$\Pi = \frac{\begin{matrix}\Pi_i^{ind}\\ \Sigma\end{matrix}}{A}$$

Then, L_i is the marking of A obtained by:

$$Mark^{-1}\left(\frac{\begin{matrix}\Pi_i^{ind}{}_{S_i}\\ \Sigma\end{matrix}}{A}\right)$$

4) Return the result of 1).

Theorem 2: Let Π be a proof tree which may contain subtrees in induction, and let I be an arbitrary declaration to the end-formula of Π. Then,

1) There are no overflowed marking numbers in $MARK(\Pi_I)$;

2) There exists a declaration, J, such that $I \subseteq J$ and $MARK(\Pi_I) = Mark(\Pi_J)$.

5. Program Extractor

The program extractor, $NExt$, is obtained by modifying the Ext procedure described in 2.3. The chief modification is as follows:

1) If the marking of a formula occurrence, A, is I, then extract the subsequence of the realizer of A which corresponds to $proj(I)(Rv(A))$.

2) For a formula occurrence whose marking is trivial, $NExt$ works in the same way as Ext.

The following is part of the definition of $NExt$:

- $NExt(\{A\}_I) \stackrel{\text{def}}{=} proj(I)(Rv(A))$ if A is used as a hypothesis

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

$$\stackrel{\text{def}}{=} if\ D\ then\ NExt\left(\frac{\begin{matrix}\{[A]\}_{J_1}\\ \Sigma_1\\ \{C\}_I\end{matrix}}{}\right)\theta\ else\ NExt\left(\frac{\begin{matrix}\{[B]\}_{J_2}\\ \Sigma_2\\ \{C\}_I\end{matrix}}{}\right)\theta$$

where D is A when A is an equation or inequation of terms, otherwise it is equal to $left = proj(0)\left(NExt\left(\frac{\Sigma_0}{\{A \vee B\}_{J_0}}\right)\right)$ and

$$\theta \stackrel{\text{def}}{=} \left\{ \begin{array}{c} proj(J_1)(Rv(A))/ttscq(1,|J_1|)\left(NExt\left(\frac{\Sigma_0}{\{A \vee B\}_{J_0}}\right)\right), \\ proj(J_2)(Rv(B))/tseq(|J_1|)\left(NExt\left(\frac{\Sigma_0}{\{A \vee B\}_{J_0}}\right)\right) \end{array} \right\}.$$

Note that $t\theta$ denotes the application of a substitution, θ, to a term, t, and $0 \in J_0$.

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

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

- $$NExt\left(\frac{\frac{\Sigma_0}{\{t:\sigma\}_J} \quad \frac{\Sigma_1}{\{A(t)\}_K}(\exists\text{-}I)}{\{\exists x:\sigma.\ A(x)\}_I}\right) \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} \left(t, NExt\left(\frac{\Sigma_1}{\{A(t)\}_K}\right)\right) & \cdots \text{if } J = \{0\} \\ NExt\left(\frac{\Sigma_1}{\{A(t)\}_K}\right) & \cdots \text{if } J = \phi \end{array} \right.$$

- $$NExt\left(\frac{\frac{\Sigma_0}{\{\exists x:\sigma.\ A(x)\}_J} \quad \frac{\{[t:\sigma_1]\}_K, \{[A(t)]\}_L}{\frac{\Sigma}{\{C\}_I}}(\exists\text{-}E)}{\{C\}_I}\right) \stackrel{\text{def}}{=} NExt\left(\frac{\{[t:\sigma]\}_K, \{[A(t)]\}_L}{\frac{\Sigma}{\{C\}_I}}\right)\theta$$

where $\theta \stackrel{\text{def}}{=} \left\{ \begin{array}{c} proj(L)(Rv(A(x)))/tseq(1)\left(NExt\left(\frac{\Sigma_0}{\{\exists x:\sigma.\ A(x)\}_J}\right)\right), \\ t/proj(0)\left(NExt\left(\frac{\Sigma_0}{\{\exists x:\sigma.\ A(x)\}_J}\right)\right) \end{array} \right\}$ if $K = \{0\}$, and

$\theta \stackrel{\text{def}}{=} \left\{ proj(L)(Rv(A(x)))/NExt\left(\frac{\Sigma_0}{\{\exists x.A(x)\}_J}\right) \right\}$ if $K = \phi$.

- $$NExt\left(\frac{\frac{\Sigma_0}{\{A(0)\}_I} \quad \frac{[x, x>0]\{[A(pred(x))]\}_J}{\frac{\Sigma_1}{\{A(x)\}_I}}(ind)}{\{\forall x:nat.\ A(x)\}_I}\right)$$

$$\stackrel{\text{def}}{=} \mu\, \overline{z}.\ \lambda\, x.\ if\ x = 0\ then\ NExt\left(\frac{\Sigma_0}{\{A(0)\}_I}\right)\ else\ NExt\left(\frac{[x, x>0]\{[A(pred(x))]\}_J}{\frac{\Sigma_1}{\{A(x)\}_I}}\right)\sigma$$

where $\sigma = \{Rv(A(pred(x)))/\overline{z}(pred(x))\}$ and $\overline{z}$ is the sequence of fresh variables with length $|I|$. Note that $J \subseteq I$ should hold.

The following theorem means that the combination of $MARK$ and $NExt$ is an extension of applying the projection on the program extracted by Ext.

Theorem 3:

Let I be any declaration to the end-formula, A, of a proof tree, Π. Then,

(1) When the $(\supset\text{-}I)$ rule is not used in Π, $NExt(MARK(\Pi_I)) = proj(I)(Ext(\Pi))$ holds.

(2) When A is the conclusion of the $(\supset\text{-}I)$ rule application and the rule is not used in the other part of Π, $T \stackrel{\text{def}}{=} NExt(MARK(\Pi_I))$ is the code which is equal to that obtained as follows:

a) Let $\lambda\overline{x}.\ t_{\overline{x}} = proj(I)(Ext(\Pi))$, where $\overline{x}$ represents the realizing variables of the assumption of the $(\supset\text{-}I)$ application;

b) Eliminate all variables from $\overline{x}$ which do not occur in $t_{\overline{x}}$ to obtain the subsequence of the variables, $\overline{y}$;

c) Let $T = \lambda\overline{y}.\ t_{\overline{x}}$.

6. An Example

The extraction of a prime number checker program is demonstrated in this section. $MARK$ and $NExt$ can be used not only to extract redundancy free programs but also to extract multiple programs from a single proof.

6.1 Extraction of a prime number checker program

The following is a specification of the program which returns the boolean value, T, if the given natural number, p, is prime, and returns F otherwise.

$$\forall p : nat.\ \ (p \geq 2 \supset \exists b : bool.\ \ (\ (\forall d : nat.\ (1 < d < p \supset \neg(d \mid p)) \wedge b = T) \\ \vee (\exists d : nat.\ (1 < d < p \wedge (d \mid p)) \wedge b = F)))$$

where $(d \mid p) \stackrel{\text{def}}{=} \exists r : nat.\ p = r \cdot d$

The outline of the proof is as follows:

$$\dfrac{\dfrac{[p : nat] \quad \dfrac{\dfrac{\Sigma}{LEMMA}}{\forall z : nat.\ (z \geq 2 \supset A(p,z))}(\forall\text{-}E)}{p \geq 2 \supset A(p,p)}(\forall\text{-}E)}{\forall p : nat.(p \geq 2 \supset A(p,p))}(\forall\text{-}I)$$

where LEMMA is $\forall p : nat.\ \forall z : nat.\ (z \geq 2 \supset A(p,z))$ and $A(p,z) \stackrel{\text{def}}{=} \exists b : nat.\ (P_0(p,z,b) \vee P_1(p,z,b))$, $P_0(p,z,b) \stackrel{\text{def}}{=} \forall d : nat.\ (1 < d < z \supset \neg(d \mid p)) \wedge b = T$, and $P_1(p,z,b) \stackrel{\text{def}}{=} \exists d : nat.\ (1 < d < z \wedge (d \mid p)) \wedge b = F$.

(1) Program extraction by Ext

Ext described in 2.3 generates the following program, where $Ext\left(\dfrac{\Sigma}{LEMMA}\right)$ is abbreviated to $Ext(\Sigma)$.

$$
\begin{aligned}
&prime = \lambda p.\ Ext(\Sigma)(p)(p)\\
&Ext(\Sigma) = \lambda p.\ \mu(z_0, z_1, z_2, z_3).\\
&\qquad \lambda z.\ if\ z = 0\ then\ any[4]\\
&\qquad else\ \ if\ z = 1\ then\ any[4]\\
&\qquad\quad else\ \ if\ z = 2\ then\ (T, left, any[2])\\
&\qquad\qquad else\ \ if\ proj(1)((z_0, z_1, z_2, z_3)(z-1)) = left \cdots (*)\\
&\qquad\qquad\quad then\ \ if\ proj(0)(Ext(\mathbf{prop})(p)(z-1)) = left\\
&\qquad\qquad\qquad then\ (T, left, any[2])\\
&\qquad\qquad\qquad else\ (F, right, z-1,\ proj(1)(Ext(\mathbf{prop})(p)(z-1)))\\
&\qquad\qquad\quad else\ (F, right, z_2(z-1), z_3(z-1))
\end{aligned}
$$

$$
\begin{aligned}
&Ext(\mathbf{prop})\\
&\stackrel{\text{def}}{=} \lambda m.\ \lambda n.\ \ (if\ proj(1)(Ext(\mathbf{Th.})) = 0\ then\ \ (right, proj(0)(Ext(\mathbf{Th.})))\ else\ (left, any[1])
\end{aligned}
$$

prop is the proof tree of the proposition in arithmetic: $\forall m.\forall n.\neg(n|m) \vee (n|m)$, and **Th.** is the proof tree of $\forall p.\forall q.\exists d.\exists r.(p = d \cdot q + r \vee 0 \leq r < q)$.

$Ext(\Sigma)$ is a multi-valued recursive call function which calculates sequences of terms of length 4. It can be observed from the program that the first element of the sequence is the boolean value, and the rest of the sequence is redundant. However, as can be seen from the part marked $(*)$, the second element of the sequence is used to calculate the first element. Consequently, the third and fourth elements are redundant.

(2) Program extraction by $MARK$ and $NExt$

• Declaration

The meaning of the realizing variables, (z_0, z_1, z_2, z_3), of the specification is as follows: z_0 is the variable for $\exists$-information of $\exists b$, z_1 is the variable for $\vee$ which connects P_0 and P_1, z_2 is the variable for $\exists$-information of $\exists d$, and z_3 is the variable for $\exists$-information of $(d \mid p)$. Let the declaration be $\{0\}$ to extract only the boolean value.

• Marking procedure

An overflowed marking number is found during the procedure. The main part of the proof of LEMMA is performed by mathematical induction, and the marking of the conclusion is $\{0\}$ and the marking of the induction hypothesis is $\{1\}$, so that 1 is the overflowed marking number. Therefore, the declaration is enlarged to $\{0,1\}$ by $MARK$.

• Code generation by $NExt$

The program extracted by $NExt$ from the marked proof tree is as follows, where Σ' is the marked proof tree of LEMMA. The chief part of the program is the recursive call function which calculates sequences of length 2.

$$
\begin{aligned}
&prime = \lambda p.NExt(\Sigma')(p)(p)\\
&NExt(\Sigma') = \lambda p.\ \mu(z_0, z_1).\\
&\qquad \lambda z.\ if\ z = 0\ then\ any[2]\\
&\qquad else\ \ if\ z = 1\ then\ any[2]\\
&\qquad\quad else\ \ if\ z = 2\ then\ (T, left)\\
&\qquad\qquad else\ \ if\ proj(1)((z_0, z_1)(z-1)) = left\\
&\qquad\qquad\quad then\ \ if\ proj(0)(Ext(\mathbf{prop})(p)(z-1)) = left\\
&\qquad\qquad\qquad then\ (T, left)\ else\ (F, right)\\
&\qquad\qquad\quad else\ (F, right)
\end{aligned}
$$

6.2 Extraction of multiple programs

Multiple programs can be extracted from the proof simply by changing the declaration, I.

Case 1: $I = \{1\}$

The extracted code is the program which returns the constants $left$ and $right$ instead of T and F.

$$
\begin{aligned}
&prime_1 \stackrel{\text{def}}{=} \lambda p.T_1(p)(p)\\
&T_1 \stackrel{\text{def}}{=} \lambda p.\ \mu z_1.\\
&\qquad \lambda z.\ if\ z = 0\ then\ any[1]\\
&\qquad else\ \ if\ z = 1\ then\ any[1]\\
&\qquad\quad else\ \ if\ z = 2\ then\ left\\
&\qquad\qquad else\ \ if\ z_1(z-1) = left\\
&\qquad\qquad\quad then\ \ if\ proj(0)(Ext(\mathbf{prop})(p)(z-1)) = left\\
&\qquad\qquad\qquad then\ left\ else\ right\\
&\qquad\qquad\quad else\ right
\end{aligned}
$$

This program can also be extracted from the proof of:

$$
\begin{aligned}
\forall p : nat.\ \ (p \geq 2 \supset (\ &(\forall d : nat.\ (1 < d < p \supset \neg(d \mid p)))\\
&\vee(\exists d : nat.\ (1 < d < p \wedge (d \mid p)))))
\end{aligned}
$$

and by giving the declaration, $\{0\}$.

Case 2: $I = \{2, 3\}$

The extracted program will return $any[2]$ when the given number, p, is prime, and otherwise returns the sequence of natural numbers (r, s), where $p = r \cdot s$ and r is the minimum number which satisfies this condition. However, 1 turns out to be an overflowed marking number in the marking procedure for the same reason in the extraction of $prime$, so that the declaration is enlarged to $\{1, 2, 3\}$. The program obtained is as follows:

$$prime_2 \stackrel{\text{def}}{=} \lambda p.\ T_2(p)(p)$$

$$T_2 \stackrel{\text{def}}{=} \lambda p.\ \mu(z_1, z_2, z_3).$$
$$\lambda z.\ if\ z = 0\ then\ any[3]$$
$$else\ \ if\ z = 1\ then\ any[3]$$
$$else\ \ if\ z = 2\ then\ (left, any[2])$$
$$else\ \ if\ proj(0)((z_1, z_2, z_3)(z-1)) = left$$
$$then\ \ if\ proj(0)(Ext(\mathbf{prop})(p)(z-1)) = left$$
$$then\ (left, any[2])$$
$$else\ (right, z-1,\ proj(1)(Ext(\mathbf{prop})(p)(z-1)))$$
$$else\ (right, z_2(z-1), z_3(z-1))$$

7. Proof Theoretic Analysis

The phenomena of overflowed and missing marking numbers were briefly explained in 4.1 from the viewpoint of the structure of recursive call functions. They can also be explained from the proof theoretic viewpoint. The forms of the marked induction step proof trees in which marking numbers overflow or miss can be specified for the normalized proofs [Prawitz 65], i.e., there are no redundant applications of rules of inferences. There are four patterns in which marking numbers may miss: critical ($\perp$-E) application, and critical ($\wedge$-$I\&E$) markings, critical ($\exists$-$I\&E$) markings, and critical ($\vee$-$I\&E$) markings. In addition, there are three patterns in which marking numbers may overflow: critical ($\exists$-E) and ($\supset$-E) applications and critical segments [Takayama 89]. Only the critical segment is presented here to illustrate how the proof theoretic analysis is performed.

7.1 Critical Segments

For example, assume that the induction hypothesis, $A(x-1)$, is in the form of $\exists y.\ B(x-1,y) \vee C(x-1,y)$ and the induction step proof is as follows:

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

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

$$\dfrac{\{[\exists y.\ B(x-1,y) \vee C(x-1,y)]\}_L \qquad \Pi}{\{A(x)\}_{\{0\}}}(\exists\text{-}E)$$

where the Π part is as follows:

$$\dfrac{\{[B(x-1,t) \vee C(x-1,t)]\}_K \qquad \dfrac{\begin{matrix}\{[t]\}_P \\ \{[B(x-1,t)]\}_I \\ \Sigma_{00} \\ \{A(x)\}_{\{0\}}\end{matrix} \qquad \dfrac{\begin{matrix}\{[t]\}_Q \\ \{[C(x-1,t)]\}_J \\ \Sigma_{11} \\ \{A(x)\}_{\{0\}}\end{matrix}}{\{A(x)\}_{\{0\}}}(\vee\text{-}E)}{\{A(x)\}_{\{0\}}}$$

P and Q are equal to $\{0\}$ or ϕ, and K and L are as follows:

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

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

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

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

Therefore, 1 is the overflowed marking number in both cases. This can be explained as follows. The realizer of $A \vee B$ is the concatenation of one of the constants, $left$ or $right$, and the realizers of A and B, and the constant is the first, i.e., the 0th element, in the realizer. Also, the constant is always used to generate the code from the proof in the ($\vee$-E) rule. Consequently, if $A \vee B$ is used as the major premise of an application of ($\vee$-E), 0 should always be contained in the marking of $A \vee B$, and it is incremented by 1 in the other major premise of ($\vee$-E) or ($\exists$-E) which contains $A \vee B$ as a subformula.

Another example is as follows. Assume that the induction hypothesis, $A(x-1)$, is in the form of $(B \vee \exists y.C(x-1,y)) \vee D$ and the induction step proof is as follows:

$$\dfrac{\dfrac{[(B \vee \exists y.C(x-1,y)) \vee D] \quad \dfrac{\dfrac{\dfrac{[B]}{B \vee \exists y.C(x,y)} \quad \dfrac{\dfrac{[\exists y.C(x-1,y)]}{\exists y.C(x,y)}\,\Sigma_1}{B \vee \exists y.C(x,y)}}{(B \vee \exists y.C(x,y)) \vee D}\,\Pi_0}{(B \vee \exists y.C(x,y)) \vee D}(\vee\text{-}E) \quad [D]}{(B \vee \exists y.C(x,y)) \vee D}\,\Pi_2\,(\vee\text{-}E)$$

Assume that the declaration, $\{k\}$ $(k > 1)$, is given to specify the $\exists$-$\vee$ information of $\exists y$. The marking of $[B \vee \exists y.C(x-1,y)]$ is $\{0\} \cup S_0$ for a finite set, S_0. Then, the marking of $[(B \vee \exists y.C(x-1,y)) \vee D]$ is $\{0,1\} \cup S_1$ for a finite set, S_1, and 0 and 1 overflow. These marking numbers correspond to the $\exists$-$\vee$ information of two $\vee$ connectives in $A(x)$. This can be explained, from the viewpoint of computation, that two decision procedures must be solved before calculating the value of y bounded by $\exists$.

Before the definition of the critical segment is given, the intuitive, but not always mathematically correct in minor respects, definitions of a few standard proof theoretic terminologies are given. See [Prawitz 65] for rigorous definitions.

Definition 9: Segment

The formula occurrences of minor premises in ($\vee$-E) and ($\exists$-E) rules are in the same form as their conclusions. Therefore, if there are successive applications of these rules in a proof tree, they form the sequence of formula occurrences in the same form. The sequence is called a *segment*.

Definition 10: Path

A sequence of formula occurrences, $(A_1, A_2, \cdots, A_n)$, in a proof tree, Π, is called a *path* iff:

1) A_1 is a top-formula of Π;
2) A_n is the end-formula of Π or a minor premise of an application of the $(\supset\text{-}E)$ rule;
3) A_i $(i < n)$ is not a minor premise of an application of the $(\supset\text{-}E)$ rule;
4) A_i $(i < n)$ is a formula immediately above A_{i+1}, or A_i is a major premise of an application of the $(\exists\text{-}E)$ or $(\vee\text{-}E)$ rule and A_{i+1} is the assumption discharged by the application in Π.

Definition 11: Critical segment
Let Π be a normalized induction step proof, and let σ be a segment in Π. Then, σ is called *critical* iff there is a formula occurrence, A, in σ such that the major premise, B, which is connected horizontally to A is on the path from an occurrence of the induction hypothesis in Π.

7.2 Overflowed marking number in the example in 6

In the program extraction example in section 6, $\{0\}$ is given as the declaration first and 1 turns out to be an overflowed marking number. This phenomenon can be explained by the notion of critical segments.

The part of the proof of the specification which corresponds to the *if-then-else* term containing the decision procedure annotated by $(*)$ in the prime number checker program in 6.1 is as follows. It is given as the marked proof tree with declaration $\{0\}$.

$$\dfrac{\{[z \geq 3]\}_\phi \quad \left\{\begin{bmatrix} z-1 \geq 2 \\ \supset A(p, z-1) \end{bmatrix}\right\}_{\{1\}} \quad \dfrac{\left\{\begin{bmatrix} P_0 \\ \vee P_1 \end{bmatrix}\right\}_{\{0\}} \quad \dfrac{\dfrac{\{[b, z-1]\}_\phi \quad \{[P_0]\}_\phi}{\Sigma_{1110}} \quad \{A(p,z)\}_{\{0\}}}{\{A(p,z)\}_{\{0\}}} \quad \dfrac{\dfrac{\{[b, z-1]\}_\phi \quad \{[P_1]\}_\phi}{\Sigma_{1111}} \quad \{A(p,z)\}_{\{0\}}}{\{A(p,z)\}_{\{0\}}} (\vee\text{-}E)}{\{\exists b.\ P_0 \vee P_1\}_{\{1\}}}}{\{A(p,z)\}_{\{0\}}} (\exists\text{-}E)$$

where $P_i \stackrel{\text{def}}{=} P_i(p, z-1, b)$, and $z - 1 \geq 2 \supset A(p, z-1)$ is the induction hypothesis. As can be seen from the tree, the sequence of $A(p, z)$ forms the critical segments, and the overflowed marking number, 1, is generated here.

8. Conclusion

The extended projection method, EPM, was presented in this paper. EPM analyzes and removes the redundancy in the program extracted from constructive proofs. It performs program analysis by using proof trees as an enriched description of the program. What is new in EPM compared to the traditional methods given in [Constable 86], [Nordström 83], [Hayashi 88], and [Paulin-Mohring 89] is:
1) the transformation from the ordinal proof trees to the annotated proof trees, which contain information about the redundant part of the proofs in terms of the program extraction, is mechanized;
2) it allows more fine-grained declaration about the redundant part of the proof;
3) the marking system in EPM is independent from the underlying constructive logic so that EPM gives a general method of program analysis.
EPM also enables the extraction of multiple programs from a single proof quite easily.

Acknowledgment

My special thanks must go to Professor Hayashi from RIMS of Kyoto University who read the early version of my paper and gave me several helpful comments. I also appreciate the encouragement of Professor Ito and Professor Sato of Tohoku University.

References

[Barendregt 81] Barendregt, H. P., *The Lambda Calculus, Its Syntax and Semantics*, North-Holland, 1981

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

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

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

[Coquand 88] Coquand, T. and Huet, G., "The Calculus of Constructions", Information and Computation, Vol. 76, pp.95-120, 1988

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

[Hayashi 88] Hayashi, S. and Nakano, H., "PX – A Computational Logic", The MIT Press, 1988

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

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

[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

[Paulin-Mohring 89] Paulin-Mohring, C., "Extracting F_ω's Programs from Proofs in the Calculus of Constructions", 16th Annual ACM Symposium on Principles of Programming Languages, 1989

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

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

[Takayama 88] Takayama, Y., "QPC: QJ-Based Proof Compiler – Simple Examples and Analysis –", *European Symposium on Programming '88*, Nancy, 1988

[Takayama 89] Takayama, Y., "Proof Theoretic Approach to the Extraction of Redundancy-free Realizer Codes", to appear in 1989