Extraction of Redundancy-free Programs from Constructive Natural Deduction Proofs

YUKIHIDE TAKAYAMA

Institute for New Generation Computer Technology, 4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan

(Received 1 June 1988)

Executable codes can be extracted from constructive proofs by using realizability interpretation. However, realizability also generates redundant codes that have no significant computational meaning. This redundancy causes heavy runtime overheads and is one of the obstacles in applying realizability to practical systems that realize the mathematical programming paradigm. This paper presents a method to eliminate redundancy by analysing proof trees as pre-processing of realizability interpretation. According to the declaration given to the theorem that is proved, each node of the proof tree is marked automatically to show which part of the realizer is needed. This procedure does not always work well. This paper also gives an analysis of the procedure and techniques to resolve critical cases. The method is studied in simple constructive logic with primitive types, mathematical induction and its q-realizability interpretation. As an example, the extraction of a prime number checker program is given.

1. Introduction

Writing programs as constructive proofs of theorems is thought to be one good approach to automated programming and program verification (see, for example, Constable, 1986; Takayama, 1987). Executable codes can be extracted from constructive proofs by using the Curry–Howard isomorphism of formulas-as-types (Howard, 1980), or the notion of realizability (Troelstra, 1973). Here, it raises the problem of extracting efficient codes from proofs, or, in other words, optimization at proof level.

Bates (1979) applied a traditional syntactical optimization technique to the code extracted from proofs, which is a set of source-to-source transformation rules. Goad (1980) gave a technique to optimize programs at proof level called pruning. Generally, a proof contains a lot of information about the program that corresponds to the proof, and the pruning technique uses the information in optimization that drastically changes the strategies of algorithms. Sasaki (1986) introduced the technique called singleton justification to Bates' program extraction algorithm so that the trivial codes for formulas that have no computational meaning can be simplified. The basic idea is as follows: if $A$ and $B$ are atomic formulas, then the computational meaning is trivial, so that the code extracted from, for instance, $A \land B$, which is called singleton formula, is (trivial, trivial). The modified program extractor simplifies the code to trivial. The class of the singleton formulas is essentially equal to that of Harrop formulas (Troelstra, 1973). The QPC system (Takayama, 1988) uses a technique similar to singleton justification. Also, it uses proof normalization method to eliminate $\beta$-redex in the extracted codes, the modified $\nu$ code technique to simplify some classes of decision procedures, and a few other code simplification rules as in Bates' and Sasaki's program extractors. However, the code extracted from
constructive proofs still has redundancy, the redundant verification code, and it causes heavy runtime overheads. The formalization of the problem in this paper is as follows. If, for example, a constructive proof of the following formal specification is given:

$$\forall x : \sigma_0. \exists y : \alpha_1. A(x, y)$$

where $\sigma_0$ and $\sigma_1$ are types, and $A(x, y)$ is a formula with free variables, $x$ and $y$, a 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:

$$\begin{align*}
[x : \sigma_0] & \quad [x : \sigma_0] \\
\Sigma_0 & \quad \Sigma_1 \\
t_x : \sigma_1 & \quad A(x, t_x) \\
\exists y : \sigma_1. A(x, y) & \quad (\exists - I) \\
\forall x : \sigma_0. \exists y : \sigma_1. A(x, y) & \quad (\forall - I)
\end{align*}$$

where $\Sigma_0$ and $\Sigma_1$ 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, $(\Sigma_1/A(x, t_x))$, $t_x$ denotes a term which contains a free variable, $x$, and $(, )$ is the sequence constructor. In this paper, the executable code extracted from a constructive proof, which is called realizer code or simply realizer, is in the form of sequence of terms or a function which outputs a sequence of terms. The code contains verification information which is not necessary in practical computation. In this case, the expected code is:

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

so that $T$ is the redundant code.

The most reasonable way to overcome this problem would be to introduce suitable notation to specify which part of the proof is necessary in terms of computation. The set notation, $\{x : A \mid B\}$, is introduced in the Nuprl system (Constable, 1986) and ITT implemented by the Göteborg group (Nordström & Petersson, 1983) as a weaker notion of $\exists x : A. B$. This is used to skip the extraction of the justification for $B$. Paulin-Mohring (1989) modified the Calculus of Constructions (Coquand & Huet, 1988) by introducing two kinds of constants, Prop and Spec, to distinguish the formulas, 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 or higher order $\lambda$-calculus with dependent types. For the non-type-theoretic formulation of constructive logic, $\diamond$-bounded formulas introduced in PX (Hayashi & Nakano, 1988) play a similar role to the set notation from which no realizer code is extracted.

This paper presents another method for the program analysis at proof tree level, and for extraction of a redundancy-free realizer code in a non-type-theoretic formalization of constructive logic. In some cases, the redundancy can be removed easily by applying a simple operation to the extracted code. For example, if the $0$th element of the realizer code, $(t_0, t_1, \ldots, t_n)$, from a proof of $\exists x : \sigma. A(x)$ is needed, it is obtained by applying projection function: $proj(0)(t_0, t_1, \ldots, t_n) = t_0$. If the theorem is in the form of $\forall x : \sigma. A(x)$, the realizer from its proof is in the form of $\lambda x. (t_0, \ldots, t_n)$, so the procedure is a little
more complicated. Translate the realizer to \((\lambda x.t_0, \ldots, \lambda x.t_n)\) and apply projection function. However, the situation around the redundancy is more complicated when the program extraction is performed on proofs in induction, in other words, when recursive call programs are extracted. It needs rather sophisticated program analysis. For example, assume that the following recursive call program is extracted from a proof in induction:

\[
\mu(z_0, z_1). \lambda x. \text{if } x = 0 \text{ then } (0, 1) \text{ else } (f_{z_0, z_1}, g_{z_0, z_1})
\]

where \(\mu\) is a fixed point operator and both of the parameters, \(z_0\) and \(z_1\), of \(\mu\) actually occur in \(f\). This function calculates a sequence of terms of length 2, and both of the elements of the sequence calculated at the recursive call step are necessary to calculate the 0th element of the sequence. Therefore, it is impossible to extract only the 0th element of the realizer code. The program analysis of redundancy can be presented quite clearly and naturally if it is performed at the proof tree level because proofs are the logical description of programs and have a lot of information about them.

Section 2 defines a simple constructive logic used in this paper. This is basically an intuitionistic first order natural deduction with mathematical induction and a variant of type-free \(\lambda\)-calculus. The program extraction algorithm, \(\text{Ext}\), is defined here. \(\text{Ext}\) performs \(q\)-realizability interpretation of proofs. Section 3 introduces the notion of marking which is a basic tool for the analysis of proof trees. The marking procedure may fail if the proof uses mathematical induction. This is explained in section 4. The modified proof extraction algorithm, \(\text{NExt}\), is defined in section 5. \(\text{NExt}\) generates redundancy-free codes from the proof tree analysed by the marking procedure. Section 6 gives a few properties and characterization of marking and \(\text{NExt}\). A prime number checker program is investigated as an example in section 7. Section 8 gives some discussion and the concluding remark.

2. Simple Constructive Logic

The constructive logic used here is an intuitionistic version of first order natural deduction with mathematical induction. It has a type-free \(\lambda\)-calculus as terms, and equality and inequality between terms. It is a sugared subset of Sato's theory, \(\text{QJ}\) (Sato, 1985; Sato, 1986).

2.1. Expressions and Inference Rules

Only the part of the definitions which is sufficient to enable understanding of the contents of succeeding sections will be given. See Takayama (1988) for details.

Types are used as domains of quantified variables.

**Definition 1. Types.**

The primitive types are \(\text{nat}\), \(\text{2}\), and \(\text{bool}\). A type is constructed with primitive types and type constructors, \(\rightarrow\) (function type constructor) and \(\times\) (cartesian product constructor).

**Definition 2. Substitutions.**

A substitution is denoted \(\{X_0/T_0, \ldots, X_{n-1}/T_{n-1}\}\) which means substituting \(T_i\) for \(X_i\), and \(X_i\) is a variable or a sequence of variables. If \(X_i\) is a sequence of variables, \(T_i\) must be a sequence of terms. Application of a substitution, \(\theta\), to a term, \(T\), is denoted \(T\theta\).
DEFINITION 3. Terms (program constructs).

(1) Atoms:
   Elements of $\text{nat}$: $0, 1, 2, \ldots$
   Elements of $2$: left and right
   Elements of $\text{bool}$: $T$ and $F$

(2) Variables: $x, y, z, \ldots$

(3) Sequence:
   If $t_0, \ldots, t_n$ are terms, then sequence of the terms, $(t_0, \ldots, t_n)$ is also a term. A sequence of variables, $(x_0, \ldots, x_{n-1})$, will often be denoted $\overline{x}$. Nil sequence is denoted $(\ )$. $\text{any}[n]$ $(n \geq 0)$ denotes a sequence of any atoms, and $\text{any}[0] = (\ )$

(4) Abstraction:
   If $M$ is a term, and $X$ is a variable or a sequence of variables, then $\lambda X. M$ is a term;

(5) Application:
   If $M$ and $N$ are terms, then $M(N)$, or simply $MN$, is a term;

(6) If-then-else:
   If $A$ is an equation or inequation of terms and $\text{beval}$ is a function which determines whether $A$ is true or not and returns Boolean values, and if $S_0$ and $S_1$ are terms, then if $\text{beval}(A)$ then $S_1$ else $S_2$ is a term. $\text{beval}(A)$ is often abbreviated to $A$;

(7) Fixed point:
   If $M$ is a term which contains the variables, $z_0, \ldots, z_n$, free, then $\mu(z_0, \ldots, z_n). M$ is a term;

(8) Built-in functions:
   $\text{succ}$, $\text{pred}$, $+$, $-$, $/$.
   $\text{proj}(n) \cdots n$th projection function;
   $\text{proj}(I)$ where $I$ is a finite sequence of natural numbers. In the following, $M$ and $N$ are terms or sequences of terms and $X$ is a variable or a sequence of variables.

   (a) $\text{proj}(\langle i_0, \ldots, i_m \rangle)(S) \triangleq \langle \text{proj}(i_0)(S), \ldots, \text{proj}(i_m)(S) \rangle$
      where $S$ is a sequence of terms of length $n$ $(m < n)$;
   (b) $\text{proj}(I)(\lambda X. M) \triangleq \lambda X. \text{proj}(I)(M)$
   (c) $\text{proj}(I)(\text{any}[n]) \triangleq \text{any}[K] (k = (\text{length of } I) \leq n)$
   (d) $\text{proj}(I)(\text{if } \text{beval}(A) \text{ then } M \text{ else } N)$
      $\triangleq$ if $\text{beval}(A)$ then proj($I$)($M$) else proj($I$)($N$)
   (e) $\text{proj}(I)(M(N)) \triangleq \langle \text{proj}(I)(M)(N) \rangle$

   Note that $\text{proj}(I)(M\theta) = (\text{proj}(I)(M))\theta$ holds for a substitution $\theta$.

   For a sequence of terms, $S$, of length $n$,
   $\text{tseq}(i)(S) \triangleq \langle \text{proj}(i)(S), \text{proj}(i+1)(S), \ldots, \text{proj}(n-1)(S) \rangle$
   where $0 \leq i \leq n-1$
   $t\text{tseq}(i, l)(S) \triangleq \langle \text{proj}(i)(S), \text{proj}(i+1)(S), \ldots, \text{proj}(i+(l-1))(S) \rangle$
   where $0 \leq i \leq n-1$, $1 \leq l \leq n-i$.

Sequence of terms:
If $S_1, \ldots, S_2$ are sequences of terms, then their concatenation is denoted $(S_1, \ldots, S_2)$. $(S, ( ))$ and $(( ), S)$ are equal to $S$. Also, the following equivalence relations are given:
   if $\text{beval}(A)$ then $(M_0, \ldots, M_n)$ else $(N_0, \ldots, N_n)$
   $\equiv$ (if $\text{beval}(A)$ then $M_0$ else $N_0, \ldots$, if $\text{beval}(A)$ then $M_n$ else $N_n$)
   $\lambda X. \langle M_0, \ldots, M_n \rangle = \langle \lambda X. M_0, \ldots, \lambda X. M_n \rangle$
   $(M_0, \ldots, M_n)(N) = (M_0(N), \ldots, M_n(N))$. 

Fixed point:
A fixed point used here has a sequence of variables as parameters. A fixed point
\[ \mu(z_0, \ldots, z_{n-1}).M \]
denotes a solution of the following fixed point equation:
\[ (z_0, \ldots, z_{n-1}) = M. \]
If the term \( M \) is equivalent to a sequence of terms of length \( n \),
\[ M = (M_0, \ldots, M_{n-1}) \quad (*) \]
the fixed point equation can be solved, and the solution, \((f_0, \ldots, f_{n-1})\) is as follows:
\[ f_i = \mu z_i M_i[z_0/f_0, \ldots, z_{i-1}/f_{i-1}, z_{i+1}/f_{i+1}, \ldots, z_{n-1}/f_{n-1}] \quad (0 \leq i \leq n - 1). \]
Therefore, the following equivalence relation is introduced:
\[ \mu(z_0, \ldots, z_{n-1}).M = (f_0, \ldots, f_{n-1}) \quad \text{if} \quad (*) \text{ holds} \]

DEFINITION 4. Formula.
(1) \( \bot \) is an atomic formula;
(2) Equation and inequation of terms are atomic formulas;
(3) If \( M \) is a term and \( \sigma \) is a type, then \( M : \sigma \) is an atomic formula;
(4) If \( A \) and \( B \) are formulas, then \( A \land B, A \lor B \) and \( A \Rightarrow B \) are formulas;
(5) If \( x \) is a variable, \( \sigma \) is a type and \( A \) is a formula, then \( \exists x : \sigma. A \) and \( \forall x : \sigma. A \) are formulas.

Negation of a formula, \( \neg A \), is defined as \( \neg A \equiv A \Rightarrow \bot \). The type declarations of bound variables are often omitted. Also atomic formula \( M : \sigma \) is often denoted simply as \( M \).

Inference rules are as follows:
- Introduction and elimination rules on \( \land, \lor, \Rightarrow, \forall \) and \( \exists \);
- \( \bot \) elimination rule;
- Rules on equality and inequality of terms;
- Mathematical induction rule;
- Term construction rules

\( (*) \) is used as the abbreviation of the names of equality rules, term construction rules, and axioms.

2.2. PROOF THEORETIC TERMINOLOGY AND NOTATION

This section gives basic proof theoretic terminologies used in the following description. \( \Pi \) always stands for proof trees, and \( \Sigma \) for sequences of proof trees.
Assumptions discharged in the deduction are enclosed by square brackets: [ ]. Note that this is different from Prawitz's notation, in which both parentheses and square brackets are used: ( ) and [ ].

DEFINITION 5. Principal sign and \( C \) formula.
(1) Let \( F \) be a formula that is not atomic. Then, \( F \) has one of the forms \( A \land B, A \lor B, A \Rightarrow B, \forall x. A, \) and \( \exists x. A \); the symbol \( \land, \lor, \Rightarrow, \forall \) or \( \exists \) is called the principal sign of \( F \).
(2) A formula with the principal sign, \( C \), is called the \( C \) formula.

DEFINITION 6. Application and node.
In a proof tree as follows

\[
\begin{array}{c}
\Sigma_0 \ldots \Sigma_n \\
\hline
A_0 \ldots A_n (R) \\
\hline
B
\end{array}
\]

the formula occurrences, \( A_0, \ldots, A_n \) and \( B \), are called nodes, and the \( \frac{A_0 \cdots A_n}{B} (R) \) part is called application of rule \( R \), or \( R \) application.

**Definition 7. Subtree.**

If \( A \) is a formula occurrence in a proof tree \( \Pi \), the subtree of \( \Pi \) determined by \( A \) is the proof tree obtained from \( \Pi \) by removing all formula occurrences except \( A \) and the ones above \( A \).

When a proof tree

\[
\begin{array}{c}
\Sigma_0 \ldots \Sigma_i \ldots \Sigma_{n-1} \\
\hline
B_0 \ldots B_i \ldots B_{n-1} \\
\hline
C
\end{array}
\]

is given, the subtree determined by \( B_i \) will often be denoted as \( (\Sigma_i/B_i) \).

**Definition 8. Top- and end-formula.**

(1) A top-formula in a proof tree \( \Pi \), is a formula occurrence that does not stand immediately below any formula occurrence in \( \Pi \).

(2) An end-formula of \( \Pi \) is a formula occurrence in \( \Pi \) that does not stand immediately above any formula occurrence in \( \Pi \).

**Definition 9. Side-connected.**

Let \( A \) be a formula occurrence in \( \Pi \), let \( (\Pi_0, \Pi_1, \ldots, \Pi_{n-1}/A) \) be the subtree of \( \Pi \) determined by \( A \), and let \( A_0, A_1, \ldots, A_{n-1} \) be the end formulas of \( \Pi_0, \Pi_1, \ldots, \Pi_{n-1} \). Then, \( A_i \) is said to be side-connected with \( A_j \) \((0 \leq i, j < n)\).

**Definition 10. Minor and major premise.**

In the following rules, \( C \)'s as premises of the rules, \( C_0 \), and \( C_1 \) are said to be minor premises. A premise that is not minor is called a major premise.

\[
\frac{C \supset B}{B} \quad \frac{C}{C} (\supset -E) \\
\frac{\exists x A(x)}{C} \quad \frac{C}{C} (\exists -E)
\]

\[
\frac{A \lor B}{C_0} \quad \frac{C_0}{C} (\lor -E) \quad \text{where } C_0 = C_1 = C
\]

\( C_0 \) is called left minor premise, and \( C_1 \) is called right minor premise.
2.3. REALIZING VARIABLE SEQUENCES AND LENGTH OF FORMULAS

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

**Definition 11. $Rv(A)$**
1. $Rv(A) \overset{\text{def}}{=} ()$ if $A$ is atomic;
2. $Rv(A \land B) \overset{\text{def}}{=} (Rv(A), Rv(B))$;
3. $Rv(A \lor B) \overset{\text{def}}{=} (z, Rv(A), Rv(B))$ where $z$ is a new variable;
4. $Rv(A \rightarrow B) \overset{\text{def}}{=} Rv(B)$;
5. $Rv(\forall x.A(x)) \overset{\text{def}}{=} Rv(A(x))$;
6. $Rv(\exists x.A(x)) \overset{\text{def}}{=} (z, Rv(A(x)))$ where $z$ is a new variable.

**Example 1.**
$Rv(\forall x : \text{nat}.(x \geq 0 \Rightarrow (x = 0 \lor \exists y : \text{nat}.\text{succ}(y) = x))) = (z_0, z_1)$ where $z_0$ denotes the information that shows which subformula of the $\lor$ formula holds and $z_1$ denotes the realizing variables of $\exists y : \text{nat}.\text{succ}(y) = x$. Note that $Rv(\text{succ}(y) = x) = ()$.

**Definition 12.** Length of formulas.
$I(A)$, which is called the length of formula $A$, is the length of $Rv(A)$.

2.4. PROOF COMPILED (Ext procedure)

The realizability used in this paper is a variant of q-realizability defined by Sato (1985). The chief difference from the standard q-realizability as seen in chapter VII of Beeson (1985) is that the realizer code for an atomic formula is defined as nil sequence here while there is no such restriction in the standard q-realizability. px-realizability (Hayashi & Nakano, 1988) also has the same restriction. Another difference from the standard form is the definition of realizability of $\lor$ formulas. The standard q-realizability defines the realizer code of $A \lor B$ as $(\text{left}, a)$ or $(\text{right}, b)$ in which $\text{left}$ and $\text{right}$ are the flags to show which formula of the disjunction actually holds and $a$ and $b$ are realizer codes of $A$ and $B$. However, it is defined as $(\text{left}, a, \text{any}[l(B)])$ or $(\text{right}, \text{any}[l(A)], b)$ in this paper.

The realizability is reformulated here as the Ext procedure (Takayama, 1988) that takes proof trees as input and returns functional style programs as output.

1. For the realizer code of an assumption, the realizing variable sequence is used:
   $$\text{Ext}(A) \overset{\text{def}}{=} Rv(A).$$

2. No significant code is extracted from an atomic formula:
   $$\text{Ext} \left( \sum_A (\text{Rule}) \right) \overset{\text{def}}{=} ()$$ where $A$ is an atomic formula.

3. The realizer codes of $\land$ formulas and $\lor$ formulas are denoted as sequences. The constants $\text{left}$ and $\text{right}$ are used to denote the information indicating which of the formulas
connected by \( \lor \) actually holds.

\[
\begin{align*}
\text{Ext} \left( \frac{\Sigma_0}{A_0} \frac{\Sigma_1}{A_1} (\land - I) \right) & \overset{\text{def}}{=} \left( \text{Ext} \left( \frac{\Sigma_0}{A_0} \right), \text{Ext} \left( \frac{\Sigma_1}{A_1} \right) \right) \\
\text{Ext} \left( \frac{\Sigma}{A_0 \land A_1} (\land - E)_i \right) & (i = 0, 1) \\
\overset{\text{def}}{=} \text{ttseq}(p, q) \left( \text{Ext} \left( \frac{\Sigma}{A_0 \land A_1} \right) \right) \quad \text{where } (p, q) = \begin{cases} 
(0, l(A_0)) & \text{if } i = 0 \\
(l(A_0), l(A_1)) & \text{if } i = 1.
\end{cases}
\end{align*}
\]

(4) The realizer code extracted from the proof in the (\( \lor - E \)) rule is the \textit{if-then-else} program. If the decision procedure of \( A \lor B \) is simple (directly executable on computers), \( \text{Ext} \) generates the \textit{modified \( \lor \) code} (Takayama, 1988).

\[
\text{Ext} \left( \frac{\Sigma}{A \lor B} \frac{\Sigma}{B} \right) \overset{\text{def}}{=} \begin{cases} 
\{ [A] \ [B] \} & \text{if both } A \text{ and } B \text{ are equations or inequations of terms} \\
\left( \frac{\Sigma_0}{A_0} \frac{\Sigma_1}{A_1} \frac{\Sigma_2}{C} \right) & \text{otherwise}
\end{cases}
\]

is as follows:

(a) \textit{if \text{beval}(A) then Ext}(\Sigma_1/C) \text{ else Ext}(\Sigma_2/C) \quad \text{[modified \( \lor \) code]}

\( \ldots \text{when both } A \text{ and } B \text{ are equations or inequations of terms} \)

(b) \textit{if left} = \text{proj}(0)(\text{Ext}(\Sigma_0/A \lor B)) \text{ then Ext}(\Sigma_1/C)\theta \text{ else Ext}(\Sigma_2/C)\theta

\( \ldots \text{otherwise} \)

where \( \theta \overset{\text{def}}{=} \left\{ \begin{array}{l}
\text{Rv}(A)/\text{ttseq}(1, l(A))(\text{Ext}(\Sigma_0/A \lor B)), \\
\text{Rv}(B)/\text{ttseq}(l(A) + 1)(\text{Ext}(\Sigma_0/A \lor B))
\end{array} \right\} \).

(5) \( \lambda \) expressions are extracted from the proofs in (\( \rightarrow - I \)) and (\( \forall - I \)):

\[
\begin{align*}
\text{Ext} \left( \frac{\Sigma}{A(x)} \forall x : \sigma A(x) \right) & \overset{\text{def}}{=} \lambda x. \text{Ext} \left( \frac{\Sigma}{A(x)} \right) \\
\text{Ext} \left( \frac{\Sigma}{A \Rightarrow B} \right) & \overset{\text{def}}{=} \lambda \text{Rv}(A). \text{Ext} \left( \frac{\Sigma}{B} \right).
\end{align*}
\]
(6) The code that is in the form of a function application is extracted from the proofs in \((\Rightarrow-E)\) and \((\forall-E)\):

\[ \text{Ext} \left( \frac{\Sigma_0}{A \Rightarrow B} \frac{\Sigma_1}{A} \right) \overset{\text{def}}{=} \text{Ext} \left( \frac{\Sigma_0}{A \Rightarrow B} \left( \text{Ext} \left( \frac{\Sigma_1}{A} \right) \right) \right) \]

\[ \text{Ext} \left( \frac{\Sigma_0}{t : \sigma} \frac{\Sigma_1}{\forall x : \sigma . A(x)} \right) \overset{\text{def}}{=} \text{Ext} \left( \frac{\Sigma_1}{\forall x : \sigma . A(x)}(t) \right). \]

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

\[ \text{Ext} \left( \frac{\Sigma_0}{t : \sigma} \frac{\Sigma_1}{\exists x : \sigma . A(x)} \right) \overset{\text{def}}{=} \left( t, \text{Ext} \left( \frac{\Sigma_1}{A(t)} \right) \right) \]

\[ \text{Ext} \left( \frac{\Sigma_0}{\exists x : \sigma . A(x)} \frac{[x : \sigma, A(x)]}{C} \right) \overset{\text{def}}{=} \text{Ext} \left( \frac{\Sigma_1}{C} \right) \theta \]

where

\[ \theta \overset{\text{def}}{=} \{ Rv(A(x))/tseq(1)(\text{Ext}(\Sigma_0/\exists x : \sigma . A(x))), x/proj(0)(\text{Ext}(\Sigma_0/\exists x : \sigma . A(x))) \}. \]

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

\[ \text{Ext} \left( \frac{\Sigma_0}{\bot} \right) \overset{\text{def}}{=} \text{any}[l(A)]. \]

(9) The code extracted from \((=.-E)\) rule is as follows:

\[ \text{Ext} \left( \frac{\Sigma_0}{x = y} \frac{\Sigma_1}{A(x)} \right) \overset{\text{def}}{=} \text{Ext} \left( \frac{\Sigma_1}{A(x)} \right). \]

(10) Multi-valued recursive call functions are extracted from the proofs in mathematical induction.

\[ \text{Ext} \left( \frac{\Sigma_0}{A(0)} \frac{\Sigma_1}{\forall x : \text{nat} . A(x)} \right) \overset{\text{def}}{=} \mu \varepsilon . \lambda x . \text{if } x = 0 \text{ then Ext}(\Sigma_0/A(0)) \text{ else Ext}(\Sigma_1/A(x)) \sigma \]

where \(\varepsilon\) is a sequence of new variables whose length is \(l(A(\text{pred}(x)))\), and

\[ \sigma = \{ Rv(A(\text{pred}(x)))/\varepsilon(\text{pred}(x)) \}. \]
THEOREM 1. (Soundness of the Ext procedure).
Let A be a formula. If $\Pi_A$ is a proof of A, then $\neg \text{Ext}(\Pi_A) \vdash A$ where $a \vdash A$ means that a term, $a$, realizes the formula $A$, and $\text{FV}(A) \supset \text{FV}(\text{Ext}(\Pi_A))$.

PROOF. By straightforward conversion from the proof of the theorem on the soundness of realizability interpretation of QJ. (See Sato, 1985).

LEMMA 1. Let $A$ and $\Pi_A$ be a formula and its proof. Then the code, $\text{Ext}(\Pi_A)$, is equivalent to a sequence of terms of length $l(A)$.

PROOF. Induction on the construction of $\Pi_A$. The crucial point is that if $A$ is a $\forall$ formula, $\forall x. B(x)$, and proved in mathematical induction, and if $\text{Ext}(\Pi_A) = \mu(z_0, \ldots, z_{n-1}). M$, then $M$ is equivalent to a sequence of terms of length $n = l(B(\text{pred}(x))) = l(B(0)) = l(B(x))$. Then, $\text{Ext}(\Pi_A)$ is equivalent to a sequence of terms as explained in section 2.1.

The realizer code extracted by $\text{Ext}$ is equivalent to a sequence of terms, so that a realizer will also be called a realizer sequence.

3. Declaration and Marking of Proof Trees

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

The realizer of a formula, $A$, is a sequence of terms of length $l(A)$ according to Lemma 1 in the last section. Not all the elements of the sequence are always necessary but it is difficult to determine automatically which parts are and which parts are not; end users must specify which elements of the realizer codes of each node are needed. At the same time it is preferable to limit the information that end users must specify. The basic requirement is that end users should not need to understand how the proof compiler works in order to specify the redundant part of the proof in terms of computation.

The proof compiler does perform realizability 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 path of the proof tree analysis by the proof compiler is traced, the information given to the end-formula can be propagated 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 unnecessary code. Consequently, end users need not specify the information about redundancy at all the nodes in the proof tree; it is enough to specify them only at the conclusion of the proof.

3.1. Declaration to Specifications

(1) Declaration, $I$, of a specification, $A$, is a subset of the finite set of natural numbers, $\{0, 1, \ldots, l(A)-1\}$. $I$ is always assumed to be sorted. Assume $I = \{i_1, \ldots, i_n\}$, then $i_p < i_q$ if $p < q$. Therefore, $I$ is also regarded as a sorted sequence of natural numbers. A specification, $A$, with the declaration, $I$, is denoted $\{A\}_I$ or simply $A_I$. Elements of the declaration are called marking numbers.
(2) The empty set, \( \emptyset \), is called a *nil declaration*.
(3) The declaration, \( \{0, 1, \ldots, l(A) - 1\} \), is called a *trivial* declaration and denoted TRV.

In the following, a declaration to the conclusion of a proof, \( \Pi \), will often be called a *declaration to a proof*; \( \Pi \), or a *declaration given to \( \Pi \)*. A declaration is a set of the position numbers of the realizer sequence that specifies which elements of the realizer sequence are needed. It is the only information that end users of the system need to specify: the other part is performed automatically.

**Example 2.** Let \( \forall x_0, \ldots, x_{m-1}, \exists y_0, \ldots, \exists y_{n-1}. A(x_0, \ldots, x_{m-1}, y_0, \ldots, y_{n-1}) \) be the specification and assume that the values of \( y_0, \ldots, y_k, 0 \leq k \leq n - 1 \), are needed. It is declared with the set of the positions: \( \{0, \ldots, k\} \).

**Example 3.** A \( \exists r \forall x. (x < 3 \Rightarrow \exists y. \exists z. \exists w. x = y \cdot z + w) \land (0 \leq w < y) \) is a specification of division of natural numbers more than 3. \( Rv(A) = (z_0, z_1) \), where \( z_0 \) corresponds to \( \exists z \) and \( z_1 \) to \( \exists w \). In other words, a realizer of \( A \) is the sequence of a value of \( z \) and a value of \( w \). If the function that calculates the remainder of division of \( x \) by \( y \) is needed, the declaration of \( A \) is \( \{1\} \).

**Example 4.** \( B \{\exists x.(\exists y.x = 2 \cdot y) v (\exists z.x = 2 \cdot y + 1) \} \) is a specification of the program which checks whether the given natural number, \( x \), is even or odd. The program extracted by \( Ext \) from a proof of \( B \) calculates the triples \( (left, V_y, any[1]), (right, any[1], V_z) \), if \( x \) is even, and \( (right, any[1], V_z) \), if \( x \) is odd, in which \( V_y \) and \( V_z \) are the values of \( y \) and \( z \). The constants, left and right, indicate whether \( x \) is even or not. Therefore, the declaration should be \( \{0\} \) to generate redundancy-free program.

### 3.2. Marking

**Definition 14.** Marking.

(1) **Marking**, \( I \), of a node \( A \) in a proof tree is \( \{0\} \) or \( \emptyset \) if \( A \) is in the form of \( M : \sigma \).

Otherwise, marking of the node is a subset of the finite set of natural numbers, \( \{0, 1, \ldots, l(A) - 1\} \). As in the definition of declaration, \( I \) is also regarded as a sorted sequence of natural numbers. A node, \( A \), with the marking, \( I \), is denoted \( \{A\}_I \) or simply \( A_I \). Elements of a marking are called *marking numbers*.

(2) The empty set, \( \emptyset \), is called a *nil marking*.

(3) The marking, \( \{0, 1, \ldots, l(A) - 1\} \), is called a *trivial* marking and denoted TRV.

Note that declaration is a special case of marking; the marking of the end-formula of a proof tree is called a declaration. A marking of the conclusion of a subtree, \( \Pi \), of a tree will often be called a *marking of \( \Pi \)* or a *marking given to \( \Pi \)*.

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

\[
\begin{array}{c}
\Sigma \\
\sum^{(*)} \frac{\mathit{f}(s, t)}{\mathit{A}(x, s, t)} (\exists - I) \\
\sum^{(*)} \frac{\exists z.A(x, s, z)}{\exists z.A(x, y, z)} (\exists - I) \\
\frac{\forall x.3z.A(x, y, z)}{\forall x.3z.A(x, y, z)} (\forall - I).
\end{array}
\]

The code extracted by \(\mathit{Ext}\) is

\[
\lambda x. (s, t, \mathit{Ext}(\Sigma / A(x, s, t))) = (\lambda x. s, \lambda x. t, \lambda x. \mathit{Ext}(\Sigma / A(x, s, t))).
\]

However, only the 0th and 1st codes are needed here, so that the declaration is \(\{0, 1\}\). The marking of \(\exists y.3z.A(x, y, z)\), \(\{0, 1\}\), is determined according to the inference rule \((\forall - I)\) and the declaration. For the node, \(\exists z.A(x, s, z)\), the 0th code of the realizer sequence is the 1st code of \(\exists y.3z.A(x, y, z)\), so that the marking is \(\{0\}\). For \(A(x, s, t)\), no realizer code is necessary here, so the marking is \(\emptyset\). \(t\) and \(s\) should also be marked by \(\{0\}\), which indicates that \(s\) and \(t\) themselves are necessary. Consequently, the following tree is obtained:

\[
\begin{array}{c}
\{[x]\} \quad \Sigma \\
\frac{(s)}{\{[x]\}} (\times) \\
\frac{\{[x]\}}{\{[x]\}} (\times)
\end{array}
\]

**DEFINITION 15.** Marked proof tree.
The *marked proof tree* is a tree obtained from a proof tree and the declaration by the marking procedure.

The proof compilation procedure, \(\mathit{Ext}\), should be modified to take marked proof trees as inputs and extract part of the realizer code according to the marking. It will be defined later. The formal definition of the marking procedure, called \(\mathit{Mark}\), will also be given later, but before that, part of the definition will be given rather informally to make the idea clearer.

3.2.1. **MARKING OF THE \((\exists - I)\) RULE**

**By definition,** the 0th code of

\[
\mathit{Ext} \begin{pmatrix} \Sigma_0 & \Sigma_1 \\ \Sigma_0 & A(t) \end{pmatrix} (\exists - I)
\]

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 \(\emptyset\). The marking of \(A(t)\) is given as the marking numbers in \(I\) except \(0\). However, note that the \(i\)th 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, $K$, of natural numbers, $K - 1 \equiv \{a \mid a \in K, a - 1 \equiv 0\}$.

### 3.2.2. Marking of the (∃-E) Rule

By the definition of the Ext procedure, the realizer 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:

\[
\begin{array}{c}
[x, A(x)] \\
\Sigma_0 \\
\exists x. A(x) \\
C \\
\Sigma_1
\end{array} \quad \frac{C}{(∃-E)}.
\]

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, and let $J$ and $K$ be the unions of the markings of all occurrences of the two hypotheses, $x$ and $A(x)$. Note that $J$ is either $\{0\}$ or $\emptyset$.

\[
\begin{array}{c}
\{[x]\}_J, \{[A(x)]\}_K \\
\Sigma_0 \\
\exists x. A(x) \\
\{C\}_I \\
\Sigma_1
\end{array} \quad \frac{\{C\}_I}{(∃-E)}.
\]

The marking of the major premise, $∃x. A(x)$, is as follows:

**Case 1.** $J = \{0\}$.  
This means that the following reasoning is contained in the subtree determined by the minor premise in which $x$ occurs in $s$:

\[
\begin{array}{c}
[x] \\
\Sigma_2 \\
\Sigma_3 \\
s_x \frac{P(s_x)}{∃y. P(y)} (∃-I) \\
s_x \frac{∀y. P(y)}{P(s_x)} (∀-E)
\end{array}
\]

and the union of the marking of all the occurrences of $x$ in $Σ_2$ or $Σ_4$ is $\{0\}$ so that the value of $x$ should be extracted from the proof tree determined by the major premise. Consequently, the 0th element of the sequence of realizer codes of $∃x. A(x)$, 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 $\{0\} \cup (K + 1)$ where $K + 1 \equiv \{a + 1 \mid a \in K\}$.

**Case 2.** $J = \emptyset$.  
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 $K + 1$.

### 3.2.3. Marking of the (⇒-E) Rule

Let $I$ be the marking of the conclusion, $B$, of a $(⇒-E)$ application. The realizer of $A ⇒ B$ is a function that takes a realizer of $A$ as input and returns a realizer of $B$. Then,
\( \sum_0 \frac{A}{\{A \supset B\}}_1 \frac{\{B\}}_1 (\supset E) \).

The marking of \( A \) should be \( TRV \). The reason is as follows. The code extracted from \((\Sigma_0/A)\) is the input of the function extracted from \((\Sigma_1/A \supset B)\). However, the marking, \( I \), of \( A \supset B \) is only to restrict the output of the function.

The marking of the subtree determined by \( A \) is continued recursively.

3.2.4. DEFINITION OF THE Mark PROCEDURE

Notational preliminary.

\( Mark \) is defined in the following style:

\[
Mark \left( \frac{\sum_0 \ldots \sum_n}{B_0 \ldots B_n} \frac{\{A\}}_I \text{ (Rule)} \right) \triangleq Mark \left( \frac{\sum_0}{\{B_0\}_0} \right) \ldots Mark \left( \frac{\sum_n}{\{B_n\}_n} \right) \text{ (Rule)}.
\]

The following are the finite natural number set operations used in \( Mark \):

\( I + n \triangleq \{x + n \mid x \in I\} \)
\( I - n \triangleq \{x - n \mid x - n \geq 0, x \in I\} \)
\( I(<n) \triangleq \{x \in I \mid x < n\} \)
\( I(\geq n) \triangleq \{x \in I \mid x \geq n\} \).

Definition of \( Mark \)

\[
Mark \left( \frac{\sum_0 \ldots \sum_n}{B_0 \ldots B_n} \frac{\{A\}}_\emptyset \right) \triangleq Mark \left( \frac{\sum_0}{\{B_0\}_\emptyset} \right) \ldots Mark \left( \frac{\sum_n}{\{B_n\}_\emptyset} \right).
\]

Assume \( I \neq \emptyset \) in the following.

\[
Mark \left( \frac{\sum_0}{t} \frac{\sum_{1} A(t)}{\{\exists x. A(x)\}_I} \text{ (\( \exists I \))} \right) \triangleq Mark \left( \frac{\sum_0}{\{t\}_K} \right) Mark \left( \frac{\sum_{1} A(t)}{\{\exists x. A(x)\}_{I-1}} \right) \text{ (\( \exists I \))}
\]

where \( K \triangleq \begin{cases} \emptyset & \text{if } 0 \not\in I; \\ \{0\} & \text{otherwise}. \end{cases} \)

\[
Mark \left( \frac{\sum_0}{\exists x. A(x)} \frac{\sum_{1}}{C} \text{ (\( \exists E \))} \right) \triangleq Mark \left( \frac{\sum_0}{\{\exists x. A(x)\}_K} \right) Mark \left( \frac{\sum_{1}}{\{C\}_I} \right) \text{ (\( \exists E \))}
\]

where \( K = \begin{cases} M + 1 & \text{if } L = \emptyset; \\ \{0\} \cup (M + 1) & \text{if } L = \{0\} \end{cases} \).
and $L$ and $M$ are the unions of the markings of all the occurrences of $x$ and $A(x)$ as hypotheses obtained in $\text{Mark}(\Sigma_1/\{C\}_I)$.

\[
\text{Mark}(\Sigma_1/\{\forall x : \sigma. A(x)\}_I) \overset{\text{def}}{=} \text{Mark}(\Sigma/\{A(x)\}_I) \quad (\forall I)
\]

\[
\text{Mark}(\Sigma_0/\{A \land B\}_I) \overset{\text{def}}{=} \frac{\text{Mark}(\Sigma_0/\{A\}_I) \text{ Mark}(\Sigma_1/\{B\}_I)}{\{A \land B\}_I} \quad (\land I)
\]

\[
\text{Mark}(\Sigma_0/\{A \lor B\}_I) \overset{\text{def}}{=} \frac{\text{Mark}(\Sigma_0/\{A\}_I) \text{ Mark}(\Sigma_1/\{B\}_I)}{\{A \lor B\}_I} \quad (\lor I)_0
\]

\[
\text{Mark}(\Sigma_0/\{A \implies B\}_I) \overset{\text{def}}{=} \text{Mark}(\Sigma/\{A \implies B\}_I) \quad (\implies I)
\]

where $K = \{0\} \cup (J_0 + 1) \cup (J_1 + 1 + l(A))$, and $J_0$ and $J_1$ are the unions of the markings of all the occurrences of $A$ and $B$ as hypotheses. Note that for the case of the modified $\lor$-code, both $J_0$ and $J_1$ are $\emptyset$, so that $K = \{0\}$ if $I \neq \emptyset$.

\[
\text{Mark}(\Sigma_0/\{A \lor B\}_K) \overset{\text{def}}{=} \frac{\text{Mark}(\Sigma_0/\{A\}_I) \text{ Mark}(\Sigma_1/\{B\}_I)}{\{A \lor B\}_I} \quad (\lor I)_1
\]
\[
\begin{align*}
\text{Mark} \left( \frac{\Sigma_0}{A} \mid \frac{\Sigma_1}{\forall x.A(x)} \right)_{\{B\}_I} \quad \text{def} \quad \text{Mark} \left( \frac{\Sigma_0}{\forall x.A(x)} \right)_{\{B\}_I} \quad \text{Mark} \left( \frac{\Sigma_1}{A \supset B} \right)_{\{B\}_I} \quad (\supset -E) \\
\text{Mark} \left( \frac{\Sigma_0}{A(0)} \mid \frac{\Sigma_1}{A(x)} \right) \quad \text{Mark} \left( \frac{\forall x.A(x)}{\{A\}_I} \right) \quad (\forall x.A(x) \text{-ind}) \\
\text{Mark} \left( \frac{\Sigma_0}{x = y} \right) \quad \text{Mark} \left( \frac{\Sigma_0}{A(0)} \mid \frac{\Sigma_1}{A(x)} \right) \quad \text{Mark} \left( \frac{\forall x.A(x)}{\{A\}_I} \right) \quad (\forall x.A(x) \text{-ind}) \\
\text{Mark} \left( \frac{\Sigma}{\bot} \right) \quad \text{Mark} \left( \frac{\Sigma}{\forall x.A(x)} \right) \quad \text{Mark} \left( \frac{\forall x.A(x)}{\{A\}_I} \right) \quad (\text{-sub}) \\
\end{align*}
\]

Assumption.

\[
\text{Mark} \left( \{A\}_I \right) \quad \text{def} \quad \{A\}_I.
\]

Inference on terms.

\[
\text{Mark} \left( \frac{\Sigma}{\{s : \tau\}_{[0]} \ast} \right) \quad \text{def} \quad \text{Let the marking of the form } s : \tau \text{ in } \Sigma \text{ which } s \text{ occurs in } t \text{ be } \{0\} \text{ and the marking of other nodes in } \Sigma \text{ be } \emptyset.
\]

\[
\text{Mark} \left( \frac{\Sigma}{\{A\}} \mid \frac{\Sigma_0}{A(0)} \mid \frac{\Sigma_1}{A(x)} \right) \quad \text{Mark} \left( \frac{\forall x.A(x)}{\{A\}_I} \right) \quad (\forall x.A(x) \text{-ind}).
\]

4. Marking Procedure on Induction Proofs

4.1. MARKING CONDITION

The programs extracted from induction proofs are recursive call programs. Assume that the declaration, \(I\), is given to an induction proof and that \(\text{Mark}\) is performed with the declaration. Let \(J\) be the union of the markings of all the occurrences of induction hypothesis.

\[
\{[A(p\text{red}(x))], J \}
\]

\[
\text{Mark} \left( \frac{\Sigma_0}{[A(0)]_I} \mid \frac{\Sigma_1}{[A(x)]_I} \right) \quad \text{Mark} \left( \frac{\forall x.A(x)}{\{A\}_I} \right) \quad (\text{nat-ind}).
\]

The recursive call program, \(f\), extracted from the marked proof tree should calculate part of the realizer sequence of \(A(x)\) (conclusion of the induction step) of the positions specified by \(I\), if the input is not 0. At the recursive call step, it should calculate the
Redundancy-free Programs 45

realizer sequence of \( A(\text{pred}(x)) \) (induction hypothesis) of a set of positions which is included in \( I \). In other words, \( J \) must be a subset of \( I \), \( J \subseteq I \). This condition will be called the marking condition. This raises a question: does the marking condition always hold? In fact, the answer is not always affirmative. The next subsection gives the way to overcome the situation in which the marking condition does not hold, and proof theoretic characterization of the critical cases will also be given after the next subsection.

4.2. MARKING WITH BACKTRACKING

The basic idea to overcome the situation in which the marking condition does not hold is marking procedure controlled by backtracking. Let a marked induction proof tree be as in the previous subsection. If \( J \nsubseteq I \), then enlarge \( I \) to \( I \cup J \) and perform Mark again. Then, it may happen that \( J \) is enlarged to \( J' \) and \( J' \nsubseteq I \cup J \). In this case, \( I \cup J \) must be enlarged again to \( I \cup J \cup J'(=I \cup J') \). This procedure will be continued until the marking condition is satisfied, but the procedure always terminates because the declaration \( I \cup J \cup J' \cup \cdots \) is bound by \( TR_V \).

The situation is a little complex for the nested induction. Assume that an induction proof \( \Pi_0 \) contains another induction proof \( \Pi_1 \) in it. Let \( I \) be the declaration to \( \Pi_0 \), and perform the marking procedure. Let \( J, LL, \) and \( L \) be the unions of the markings of all the occurrences of the induction the hypotheses of \( \Pi_0 \) and \( \Pi_1 \), and the marking of the conclusion of \( \Pi_1 \). The marking conditions for the nested induction are \( J \subseteq I \) (condition for \( \Pi_0 \)) and \( LL \subseteq L \) (condition for \( \Pi_1 \)). \( I \) must be made sufficiently large to satisfy both of the conditions. Generally speaking, \( J, L, \) and \( LL \) are enlarged when \( I \) is enlarged. Suppose, for example, that \( LL \subseteq L \) and \( J \nsubseteq I \). Then, \( I \) must be enlarged to satisfy the condition for \( \Pi_0 \). However, this procedure may destroy the condition for \( \Pi_1 \) if \( LL \subseteq L' \) may hold for the new values, \( LL' \) and \( L' \), of \( LL \) and \( L \). Then, \( I \) must be enlarged again to satisfy the condition for \( \Pi_1 \), and that may destroy the condition for \( \Pi_0 \), and so on. Therefore, backtracking becomes rather complicated for the nested induction.

However, if the induction hypothesis of \( \Pi_0 \) is not used in \( \Pi_1 \), the backtrack can be made simpler by using a sort of projection function:

1. Let the declaration, \( I \), to \( \Pi_0 \) be sufficiently large to satisfy the marking condition for \( \Pi_0 \), and let \( L \) and \( LL \) be as above;
2. If \( LL \subseteq L \), the marking procedure on \( \Pi_0 \) is successful. Otherwise, go to (3);
3. Enlarge \( L \) (not \( I \)) to \( L' \) to satisfy the marking condition for \( \Pi_1 \), which is to say Mark(\( \Pi_1 \)) succeeds.

The modified proof compilation algorithm will become a little complex if it is to handle the marked proof tree obtained by the procedure (1) and (3). The proof compiler will generate the following program from the marked version of \( \Pi_0 \):

\[
\mu(z_{i_0}, \ldots, z_{i_k}).F_T
\]

where \( \{i_0, \ldots, i_k\} = I \), and \( F \) is the term in which \( T \), the code from the marked version of \( \Pi_1 \), occurs. \( T \) is obtained as follows: Let \( S \) be the realizer code extracted from the marked version of the subproof \( \Pi_1 \). \( S \) is the realizer codes of the conclusion of \( \Pi_1 \) of the positions specified by \( L' (\supseteq L) \). Then, \( T_{\text{def}} \text{proj}(L/L')(S) \) which works as follows: First, evaluate a value of \( S \). Let \( (s_{i_1}, \ldots, s_{i_k}) \) be the value, and \( L' = \{i_1, \ldots, i_k\} = \{j_1, \ldots, j_l\} = L \). Then the value of proj\((L/L')(S)\) is \((s_{j_1}, \ldots, s_{j_l})\).
4.3. PROOF THEORETIC CHARACTERIZATION OF CRITICAL APPLICATIONS

This subsection gives a proof theoretic characterization of the situation in which the marking condition does not hold. The results have no direct relation with the proof compilation algorithm that generates redundancy-free programs. However, the characterization gives a proof theoretic explanation of the phenomenon of marking of proof trees. Also, it could give a way of program analysis of recursive call programs at proof level.

4.3.1. CRITICAL SEGMENTS

(1) An example.
Let \( A(x) \) def \( \exists y. B(x, y) \vee C(x, y) \). Suppose that \( \forall x: \text{nat}. A(x) \) is proved by mathematical induction, and the induction step proceeds as follows:

\[
\frac{[\exists y. B(x-1, y) \vee C(x-1, y)]}{A(x)} \quad (\exists-E)
\]

where \( \exists y. B(x-1, y) \vee C(x-1, y) \) is the induction hypothesis, and \( \Pi \) is as follows:

\[
\frac{[y]}{B(x-1, y)} \quad [C(x-1, y)]
\]
\[\Sigma_0 \]
\[\Sigma_1 \]
\[
\frac{[B(x-1, y) \vee C(x-1, y)]}{A(x)} \quad A(x)
\]
\[\Sigma \quad \Sigma \]
\[\Sigma \quad \Sigma \]
\[
\frac{(\vee-E)}{A(x)}
\]

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

\[
\frac{[[\exists y. B(x-1, y) \vee C(x-1, y)]]}{\{A(x)\}_{(0)}} \quad (\exists-E)
\]

(2) Formal definition of critical segments.
The reason for this phenomenon is that the realizer code of \( A \vee B \) consists not only of the code of \( A \) and \( B \) but also of the code, \( \text{left} \) or \( \text{right} \). Therefore, the marking of \( A \vee B \) must contain 0 when the formula is the major premise of a \( (\vee-E) \) application.

The following proof theoretic terminologies are needed to formalize critical segments.

**Definition 16.** Thread.
A sequence \( A_1, A_2, \ldots, A_n \) of formula occurrences in a proof tree \( \Pi \) is a thread in \( \Pi \) if
(1) \(A_1\) is a top-formula in \(\Pi\), (2) \(A_i\) stands immediately above \(A_{i+1}\) in \(\Pi\) for each \(i < n\), and (3) \(A_n\) is the end-formula of \(\Pi\).

**Definition 17. Segment.**
The same formulas occur as minor premises and conclusions in \((\lor-E)\) and \((\exists-E)\) rules. Therefore, if there are successive applications of these rules in a proof tree, there are consecutive occurrences of the same formula in a thread. This sequence is called a *segment*. Any formula occurrence in a proof tree which is not a minor premise or a conclusion of these rules is also regarded as forming a trivial segment.

**Definition 18. Path.**
A *path* is the deduction sequence from a top-formula which is not discharged by \((\lor-E)\) or \((\exists-E)\) applications to the end-formula or to a minor premise of an application of the \((\rightarrow-E)\) rule. A path branches at an application of the \((\lor-E)\) rule or the \((\exists-E)\) rule:

\[
\frac{[A] [B]}{\Sigma_0} \quad \frac{\Sigma_1}{A \lor B} \quad \frac{\Sigma_2}{C} \quad \frac{(\lor-E)}{C} \quad \Pi
\]

In the \((\lor-E)\) rule application above, a path from a top-formula in \(\Sigma_0\) branches at \(A \lor B\). A branch passes through an occurrence of \(A\) or \(B\) as the discharged hypotheses, and goes down to the occurrence of \(C\) as the conclusion of the application. It is similar in the \((\exists-E)\) rule application. A path from a top-formula in \(\Sigma_0\) branches at \(\exists x.A(x)\), and a branch goes to the occurrence of \(C\) through one of the occurrences of \(A(x)\) as the discharged hypotheses. A path whose last element is the end-formula of the proof tree is called a *main path*.

See Prawitz (1965) for the formal definitions of thread, segment and path.

**Definition 19. Major premise attached to a formula.**
The major premise of the application of \((\lor-E)\) or \((\exists-E)\) that is side-connected with a formula, \(A\), in a segment is called the *major premise attached* to \(A\).

**Definition 20. Proper segment.**
The non-trivial segment in a marked proof tree, \(\Pi\), is called *proper* if every formula occurrence in the segment has non-nil marking.

**Definition 21. Critical segments.**
Let \(\Pi\) be an induction step proof in a proof tree. A proper segment, \(\sigma\), in \(\Pi\) is *critical* if there is a formula occurrence, \(A\), in \(\sigma\) such that the major premise, \(B\), attached to \(A\) is a formula occurrence in one of the main paths in \(\Pi\) from an occurrence of the induction hypothesis which also passes through \(\sigma\).

**Definition 22. Indispensable marking numbers.**
Assume an induction step proof, \(\Pi\). An *indispensable marking number* is a marking number of a node in \(\Pi\) which is obtained as follows:
(a) The node is along a main path in \(\Pi\) from an occurrence of the induction hypothesis;
(b) The marking number is propagated from the marking number, 0, of an occurrence of a \(\lor\) formula as the major premise of an \((\lor-E)\) application.
If there is a critical segment in an induction proof, there is a possibility that the marking condition is not satisfied because of the indispensable marking numbers of occurrences of the induction hypothesis.

Indispensable marking numbers can be calculated systematically in a restricted case as in the following lemma.

**Lemma 2.**

Let $\Pi$ be an induction step proof. Let $S \overset{\text{def}}{=} (A_1, A_2, \ldots, A_m)$ be a critical segment in $\Pi$, and $\pi$ be a main path in $\Pi$ from an occurrence of the induction hypothesis which passes through $S$ and a major premise, $F$, attached to $A_n$ (for some $n, 1 \leq n \leq m$) in $S$. Assume that there is a subsequence, $\pi_0 = \{B_1, B_2, \ldots, B_i\}$, of $\pi$ such that:

(a) $B_i = F$

(b) $B_i$ is a major premise attached to $A_{k(i)}$ in $S$ ($1 \leq i \leq l$, $k(1) = n$)

(c) $B_i$ ($i \geq 2$) is discharged by the ($\exists$-$E$) or ($\lor$-$E$) application whose major premise is $B_{i-1}$.

Then, the marking of $F$ contains the marking numbers $\varphi(j)$ ($1 \leq j \leq K$) defined as follows:

$$\varphi(i) \overset{\text{def}}{=} \sum_{p=1}^{\alpha(i)} \psi(p)$$

$$\psi(p) = \begin{cases} 0 & \text{if } p = 1 \\ 1 & \text{if } B_{p-1}(=C \lor D) \text{ is a major premise of } (\lor$-$E) \text{ and } B_p = D \\ 1 & \text{otherwise} \end{cases}$$

where $K$ and $\alpha(i)$ are as follows: Let $\pi_1 = \{B_{a(1)}, \ldots, B_{a(K)}\}$ is the subsequence of $\pi_0$ such that $B_{a(j)}$ ($1 \leq j \leq K$) is a major premise of an application of the ($\lor$-$E$) rule.

**Proof.** Let the occurrences of $A_i$ and $A_{i+1}$ be as follows:

$$\frac{[A] \quad [B]}{A_\lor B \quad A_i \quad A_i'} \quad (\lor$-$E) \quad (\text{where } A_i = A_i' = A_{i+1}).$$

Assume that $A \lor B$ is an element of $\pi_1$. As $S$ is a proper segment, the marking of $A \lor B$ contains 0.

**Case 1.** Assume that an element, $A_k$ ($k \geq i + 1$), in $S$ is a minor premise of an application of the ($\exists$-$E$) rule and $F_0$ in $\pi_0$ is a major premise attached to $A_k$. Assume also that $F_0$ is immediately before $A \lor B$ in $\pi_0$, that is, $A \lor B$ is discharged by the application of ($\exists$-$E$). Then, the marking number, 0, of $A \lor B$ becomes 1 in the marking of $F_0$:

$$\frac{[A] \quad [B]}{\Sigma_1 \quad \Sigma_2} \quad \{A_{i+1}\}_{I} \quad \{A_i'\}_{I}$$

$$\frac{\{A \lor B\}_{(0) \cup T_{\theta}} \quad \{A_i\}_{I}}{\{A_{i+1}\}_{I}} \quad \cdots$$

$$\frac{\{F_0\}_{(1) \cup T_{\theta}} \quad \{A_k\}_{I}}{\{A_{k+1}\}_{I}} \quad (\exists$-$E).$$
CASE 2. Assume that an element, $A_k (k \geq i + 1)$, in $S$ is a left minor premise of an $(\lor-E)$ application and $F_1$ in $\pi_0$ is a major premise attached to $A_k$. Assume also that $F_1$ is immediately before $A \lor B$ in $\pi_0$. Then, the marking number, 0, of $A \lor B$ become 1 in the marking of $F_1$.

\[
\begin{array}{c|c|c}
\{[A \lor B]\}_{(0) \cup \tau_0} & \{A_1\}_I & \{A_k\}_I \\
\{A_{i+1}\}_I & \cdots & \{A_k\}_I \\
\{F_1\}_{(1) \cup \tau_1} & \cdots & \{A_k\}_I \\
\{A_{k+1}\}_I & \cdots & \{A_k\}_I \\
\end{array}
\]

$((\lor-E)$.

CASE 3. Assume that an element, $A_k (k \geq i + 1)$, in $S$ is a right minor premise of a $(\lor-E)$ application and $F_2 (= C \lor D)$ in $\pi_0$ is a major premise attached to $A_k$. Assume also that $F_2$ is immediately before $A \lor B$ in $\pi_0$. Then, the marking number, 0, of $A \lor B$ becomes $l(C) + 1$ in the marking of $F_2$.

\[
\begin{array}{c|c|c}
\{[A \lor B]\}_{(0) \cup \tau_0} & \{A_1\}_I & \{A_k\}_I \\
\{A_{i+1}\}_I & \cdots & \{A_k\}_I \\
\{F_2\}_{(l(C) + 1) \cup \tau_2} & \cdots & \{A_k\}_I \\
\{A_{k+1}\}_I & \cdots & \{A_k\}_I \\
\end{array}
\]

$((\lor-E)$.

The lemma follows by continuing the discussion in a similar way.

EXAMPLE 5. There are many other cases of indispensable marking numbers. Assume the following induction step proof where $F = \exists x. A \wedge ((B \lor C) \lor D)$, $IH$ is an occurrence of the induction hypothesis and $I \neq \emptyset$.

\[
\begin{array}{c|c|c|c}
\{[A \wedge ((B \lor C) \lor D)]\}_M \wedge-E & \{B \lor C\}_K \lor-E & \{D\}_I \lor-E \\
\{A\}_I & \{A\}_I & \{A\}_I \\
\{F\}_N & \{A\}_I & \Pi \\
\end{array}
\]

As $0 \in K$, $1 \in L$ and $l(A) + 1 \in M$. Hence, $l(A) + 2 \in N$, $0$, $1$, $l(A) + 1$, and $l(A) + 2$ are indispensable marking numbers of $B \lor C$, $(B \lor C) \lor D$, $A \wedge ((B \lor C) \lor D)$, and $F$.

4.3.2. CRITICAL $((\exists-E)$ APPLICATION

(1) An example.

Assume that $\forall x. \exists y. \exists z. A(x, y, z)$ is proved in mathematical induction, and the declaration, $\{0\}$ is given to the conclusion. Also assume that the induction step part of the marked
proof tree is as follows:
\[
\frac{\{\exists y.\exists z. A(x-1, y, z)\}_{L}}{\exists y.\exists z. A(x, y, z)_{(0)}} (\exists-E)
\]
where \(\Pi\) is as follows:
\[
\frac{\{y\}_{(0)} \quad \{z\}_{0} \quad \{s_{y}z\}_{0}}{\exists z.A(x, s_{y}z, z)_{0} (\exists-I)}
\]
\[
\frac{\Sigma_{0} \quad \Sigma_{1}}{\exists y.\exists z. A(x, y, z)_{(0)}} (\exists-E).
\]
Note that both of the assumptions of the \((\exists-E)\) rules (eigen variables), \(y\) and \(z\), are used to construct \(s_{y}z\), so that \(0 \in K\) and \(\{0, 1\} \subseteq L\). Therefore, the marking condition does not hold: \(L \not\subseteq \{0\}\).

(2) Definition of critical \((\exists-E)\) applications.

\textbf{DEFINITION 23. Critical \((\exists-E)\) applications.}
If a \(\forall\) formula \(A \equiv \forall x.B(x)\) is proved in induction and \(A\) contains an \(\exists\) formula \(C(x)\). Assume that there is a main path from an occurrence of the induction hypothesis in which \(C(x-1)\) occurs as the major premise of an \((\exists-E)\) application and that there is an eigen variable of the application whose marking is \(\{0\}\). Let \(k\) be the position number of the principal sign, \(\exists\), of \(C(x)\) in \(A\). Then, if \(k\) is not contained in the declaration to \(A\), the \((\exists-E)\) application is said to be critical.

Note that, in the example (1), one of the \((\exists-E)\) applications is critical.

4.3.3. OTHER CRITICAL APPLICATIONS

The notion of critical segments and critical \((\exists-E)\) applications can only capture the situation of the marking along a main path from an occurrence of the induction hypothesis. However, there may be a path from an occurrence of the induction hypothesis which is not a main path. For example, assume a marked induction step proof is as follows:
\[
\frac{\{A(x-1)\}_{K}}{B_{TRV}} \quad \frac{\Sigma_{0} \quad \Sigma_{1}}{\exists y.\exists z. A(x, y, z)_{(0)}} (\exists-E) (J \neq \emptyset).
\]
\[
\frac{\{C\}_J}{\Pi} (\exists-E).
\]
The marking of \(B\) as a minor premise of the application of \((\exists-E)\) is always \(TRV\), so that \(K\) is always the same value whenever \(J\) is not nil marking. Therefore, \(I\) must be made sufficiently large to satisfy, \(K \subseteq I\).
5. Modified Proof Compilation Algorithm

The proof compilation should be modified to handle marked proof trees. The chief modification is:
(1) if the given formula, \( A_i \), is marked by \( \{i_0, \ldots, i_k\} \), extract the code for the \( i_l \)th \((0 \leq i \leq k)\) realizing variable in \( \text{rv}(A) \);
(2) if the formula, \( A_i \), is marked by \( \emptyset \), no code should be extracted and there is no need to analyse the subtree determined by \( A \).

The following is the definition of the modified version of the \( Ext \) procedure, \( NExt \). \(|I|\) denotes the number of elements in \( I \).

(1) Nil marking:
\[
NExt\left( \frac{\{A_0\}_{j_0} \cdots \{A_k\}_{j_k}}{\{B\}_I} (\text{Rule}) \right) \overset{\text{def}}{=} (\quad) \quad \text{where } I = \emptyset.
\]

In the following, \( I \) is assumed to be non-nil.

(2) Assumptions:
\[
NExt(\{A\}_I) \overset{\text{def}}{=} \text{proj}(I)(\text{rv}(A)).
\]

(3) \( \land \) and \( \lor \) formulas:
\[
NExt\left( \frac{\Sigma_0}{\{A_0\}_I} \frac{\Sigma_1}{\{A_1\}_I} (\land - I) \right) \overset{\text{def}}{=} \begin{cases} 
NExt\left( \frac{\Sigma_0}{\{A_0\}_I}, NExt\left( \frac{\Sigma_1}{\{A_1\}_I} \right) \right) & \text{if } I = \emptyset, \\
NExt\left( \frac{\Sigma}{\{A\}_I} \right) & \text{where } i = 0, 1.
\end{cases}
\]
\[
NExt\left( \frac{\Sigma}{\{A\}_I} (\lor - I) \right) \overset{\text{def}}{=} \begin{cases} 
\left( \text{left, } NExt\left( \frac{\Sigma}{\{A\}_I}, \text{any}[k] \right) \right) & \text{if } 0 \in I, \\
\left( \text{right, any}[I], NExt\left( \frac{\Sigma}{\{A\}_I} \right) \right) & \text{if } 0 \notin I.
\end{cases}
\]

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

(4) The code from \( (\lor - E) \) rule:
\[
NExt\left( \frac{\Sigma_0}{\{A \lor B\}_I} \frac{\Sigma_1}{\{C\}_I} (\lor - E) \right) \overset{\text{def}}{=} \begin{cases} 
\left( \text{left, any}[k], NExt\left( \frac{\Sigma}{\{B\}_I} \right) \right) & \text{if } 0 \in I, \\
\left( \text{right}[I], NExt\left( \frac{\Sigma}{\{B\}_I} \right) \right) & \text{if } 0 \notin I.
\end{cases}
\]
is as follows:

(a) if beval($A$) then \(N\text{Ext} \left( \frac{\Sigma_1}{\{C\}_1} \right)\) else \(N\text{Ext} \left( \frac{\Sigma_2}{\{C\}_1} \right)\) \([\text{modified \& code}]\)

when both $A$ and $B$ are equations or inequations of terms.

Note that, in this case, $J_1 = J_2 = \emptyset$.

(b) if left = proj(0) \(N\text{Ext} \left( \frac{\Sigma_0}{\{A \lor B\}_b} \right)\) then \(N\text{Ext} \left( \frac{\Sigma_1}{\{C\}_1} \right)\) else \(N\text{Ext} \left( \frac{\Sigma_2}{\{C\}_1} \right)\)

otherwise

where $\theta \triangleq \begin{cases} 
\text{proj}(J_1)(Rv(A))/tseq(1, |J_1|)(N\text{Ext}(\Sigma_0/\{A \lor B\}_b)), & \text{if } 0 \in J_1 \\
\text{proj}(J_2)(Rv(B))/tseq(|J_1|+1)(N\text{Ext}(\Sigma_0/\{A \lor B\}_b)), & \text{otherwise}
\end{cases}$

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

\[\text{NExt} \left( \frac{[x:\sigma]}{[\forall x: \sigma. A(x)]_1} \right) \overset{\text{def}}{=} \lambda x. \text{NExt} \left( \frac{\Sigma}{\{A(x)\}_1} \right)\]

(6) The codes from the proofs in ($\supset$-E) and ($\forall$-E):

\[\text{NExt} \left( \frac{\Sigma_0/\{A\}_1}{\{A \supset B\}_1} \right) \overset{\text{def}}{=} \text{NExt} \left( \frac{\Sigma_1/\{A \supset B\}_1}{\{A\}_1} \right) \left( \text{NExt} \left( \frac{\Sigma_0}{\{A\}_1} \right) \right)\]

\[\text{NExt} \left( \frac{\Sigma_0/\{t: \sigma\}_1}{\{A(t)\}_1} \right) \overset{\text{def}}{=} \text{NExt} \left( \frac{\Sigma_1/\{A(t)\}_1}{\{A(t)\}_1} \right) (t)\]

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

\[\text{NExt} \left( \frac{\Sigma_0/\{t: \sigma\}_1}{\exists x: \sigma. A(x)} \right) \overset{\text{def}}{=} \begin{cases} 
(t, \text{NExt} \left( \frac{\Sigma_1}{\{A(t)\}_1} \right)) & \text{if } J = \{0\} \\
\text{NExt} \left( \frac{\Sigma_1}{\{A(t)\}_1} \right) & \text{if } J = \emptyset
\end{cases}\]

\[\text{NExt} \left( \frac{\Sigma_0/\exists x: \sigma. A(x)}{\Sigma_1/\{C\}_1} \right) \overset{\text{def}}{=} \text{NExt} \left( \frac{\Sigma_1}{\{C\}_1} \right) \theta\]

where $\theta \triangleq \begin{cases} 
\text{proj}(L)(Rv(A(x)))/tseq(1)(N\text{Ext}(\Sigma_0/\exists x: \sigma. A(x))) & \text{if } 0 \in J,
\end{cases}$
and $\theta \overset{\text{def}}{=} \{\text{proj}(L)(Rv(A(x)))\}/\text{NExt}(\Sigma_0/\{\exists x : \sigma.A(x)\})$ if $0 \notin I$.

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

$$\text{NExt} \left( \frac{\Sigma}{\{A\}_I} (\bot-E) \right) \overset{\text{def}}{=} \text{any}[k] \quad \text{where} \quad k = |I|.$$ 

(9) The code extracted from $(=E)$ rule:

$$\text{NExt} \left( \frac{\Sigma_0 \{x = y\}_I \{A(x)\}_I}{\{A(y)\}_I} (=E) \right) \overset{\text{def}}{=} \text{NExt} \left( \frac{\Sigma_1}{\{A(x)\}_I} \right).$$

(10) The realizer code extracted from the proof by mathematical induction:

$$\text{NExt} \left( \frac{\Sigma_0 \{x : \text{nat}\} \{x > 0\}_I \{A(x-1)\}_I}{\{A(0)\}_I \{A(x)\}_I} (\forall x.A(x)) \right) \overset{\text{def}}{=} \mu z. \lambda x. \text{if} \ x = 0 \ \text{then} \ \text{NExt}(\Sigma_0/\{A(0)\}_I) \ \text{else} \ \text{NExt}(\Sigma_1/\{A(x)\}_I) \ \sigma$$

where $J \subseteq I$ and $\tilde{z}$ is a sequence of fresh variables of length $|I|$ and

$\sigma \overset{\text{def}}{=} \{\text{proj}(I)(Rv(A(x-1)))\}/\tilde{z} (\text{pred}(x)).$

### 6. Some Properties of Mark and NExt

#### 6.1. Normalization of Marked Proof Trees

Let $R$ be one of the logical connectives and quantifiers: $\rightarrow$, $\land$, $\lor$, $\forall$, and $\exists$. An application of $(R-I)$ succeeded by an application of $(R-E)$ is called an $R$-cut. Cuts can be eliminated by the $R$-reduction rules as defined in Prawitz (1965) and the rules are used in proof normalization. The rules will be denoted $\text{Red}_R$ in the following.

Cuts can also be defined on marked proof trees: a part of a marked proof tree is called an $R$-cut if it is an $R$-cut when all the markings of the nodes are removed. The $R$-reduction rules on marked proof trees, which will be referred to as $\text{Red}_R$ in the following, are defined as follows:

**Definition 24.** $R$-reduction rules on marked proof trees.

$\text{Red}_R$:

$$\begin{align*}
\frac{\Sigma_0}{\{B\}_I} (\rightarrow-I) & \quad \frac{\Sigma_1}{\{A\}_I} (\rightarrow-E) \quad \frac{\Sigma_1}{\{A\}_I} \\
\frac{\Sigma_0}{\{B\}_I} & \quad \frac{\Sigma_1}{\{B\}_I} \\
\frac{\{A\}_I}{\{A\}_I} & \quad \frac{\{B\}_I}{\{B\}_I} \\
\frac{\{A\}_I}{\{A\}_I} & \quad \frac{\{B\}_I}{\{B\}_I} \
\end{align*}$$
The meaning of $(\Sigma_{11}/[[A]]/\Sigma_0/\{B\})$ in the definition of $\text{Red}_\omega$ is as follows: Let $A_1, \ldots, A_k$ be the occurrences of $A$ as hypothesis in $(\Sigma_0/\{B\})$ and $J_1, \ldots, J_k$ be the marking of them such that $J_1 \cup \cdots \cup J_k = J$. Then, $(\Sigma_{11}/[[A]]/\Sigma_0/\{B\})$ is obtained by replacing the node $A_i$ in $(\Sigma_0/\{B\})$ by $\text{Mark}(\Sigma_j/\{A\})$ ($1 \leq i \leq k$). The meaning of the tree obtained by $\text{Red}_\nu$, $\text{Red}_\nu$, and $\text{Red}_\delta$ are defined similarly.

R-reduction of proof trees and the marking procedure commute in the following sense:
**Theorem 2. Normalization and Mark.**

Let $\Pi$ be a proof and $I$ be a declaration to $\Pi$, and let Mark$(I, \Pi)$ be the tree obtained by the marking procedure applied on $\Pi$ with the declaration $I$. Assume that the last two applications of rules in $\Pi$ form an R-cut. Then, if both Mark$(I, \Pi)$ and Mark$(I, \text{Red}_R(\Pi))$ succeed, $\text{Red}_R(\text{Mark}(I, \Pi)) = \text{Mark}(I, \text{Red}_R(\Pi))$, where $R$ is $\rightarrow$, $\land$, $\lor$, $\forall$, or $\exists$.

**Proof.** Straightforward.

Note that $\text{Mark}$ fails when the proof contains induction proofs and the marking condition is not satisfied. Also, even if the marking of a normalized proof satisfies the marking condition, the condition is not always satisfied for the original proof. For example, assume the following is a marked version of an induction step proof, $\Pi$, with marking $I$:

\[
\begin{array}{c}
\{[y]\}_0 \\
\Sigma_0 \\
\{t_\gamma\}_0 \\
\Sigma_1 \\
\{\exists y. B(x - 1, y)\}_K \\
\Pi_0 \\
\{\exists y. B(x, y)\}_I
\end{array}
\]

where $J \neq \emptyset$ and $0 \notin I$. Then, $0 \notin K$, so that the application of $(\exists - E)$ is critical and the marking condition is not satisfied. However, if $\text{Red}_v$ is applied to $\Pi$ and the marking procedure is performed with the marking $I$, the tree is as follows:

\[
\begin{array}{c}
\{[y]\}_0 \\
\Sigma_{00} \\
\{[t_\gamma]\}_0 \\
\Sigma_1(x/t_\gamma) \\
\{A(t_\gamma)\}_I \\
\{\exists y. B(x - 1, y)\}_K' \\
\Pi_0 \\
\{\exists y. B(x, y)\}_I
\end{array}
\]

In this case, $0 \notin K'$, so that the marking condition may be satisfied.

### 6.2. NExt Procedure and Projection

**Lemma 3.** Let the marked proof trees of $(\rightarrow - I)$, $(\lor - E)$, and $(\exists - E)$ applications and an induction step proof be as follows:

\[
\begin{array}{c}
\{[A]\}_I \\
\Sigma \\
\{B\}_I \\
\{A \rightarrow B\}_I
\end{array}
\]

(\rightarrow - I)
\[ \begin{array}{c}
\Pi_1 \\
\Sigma_0 \frac{\{[A]\}_{j_1} \quad \{[B]\}_{j_k}}{\{A \lor B\}_I} \\
\Sigma_1 \frac{\{C\}_I}{\{C\}_I} \quad (\lor\text{-}E).
\end{array} \]

\[ \begin{array}{c}
\Pi_2 \\
\Sigma_0 \frac{\{x\}_I \{[A(x)]\}_M}{\exists x. A(x)}_I \\
\Sigma_1 \frac{\{C\}_I}{\{C\}_I} \quad (\exists\text{-}E)
\end{array} \]

\[ \begin{array}{c}
\Pi_3 \\
\Sigma_0 \frac{\{A(x-1)\}_{j}}{\{A(0)\}_I} \\
\Sigma_1 \frac{\{A(x)\}_I}{\forall x. A(x)}_I \\
\end{array} \]

where \([A]_I\) in \(\Pi_0\), for example, means that \(J\) is the union of the marking of all the occurrences of \(A\) as the discharged hypothesis. Then,

1. For \(\Pi_0\), if an element, \(z\), of \(Rv(A)\) occurs free in \(NExt(\Sigma_1/\{B\}_I)\), then \(z\) is an element of \(proj(J)(Rv(A))\);
2. For \(\Pi_1\), if an element, \(z\), of \(Rv(A)\) (or \(Rv(B)\)) occurs free in \(NExt(\Sigma_1/\{C\}_I)\) (or \(NExt(\Sigma_2/\{C\}_I)\)), then \(z\) is an element of \(proj(J)(Rv(A))\) (or \(proj(J_2)(Rv(B))\));
3. For \(\Pi_2\), if an element, \(z\), of \(Rv(A(x))\) occurs free in \(NExt(\Sigma_1/\{A(x)\}_I)\), then \(z\) is an element of \(proj(J)(Rv(A(x)))\);
4. For \(\Pi_3\), if an element, \(z\), of \(Rv(A(x-1))\) occurs free in \(NExt(\Sigma_1/\{A(x)\}_I)\), then \(z\) is an element of \(proj(J)(Rv(A(x-1)))\).

**Sketch of the Proof.** It is sufficient to prove the following somewhat stronger statement:

Let a marked proof tree, \(\Pi\), be as follows, and let \(A\) be an arbitrary formula which is used in \(\Pi\) as a hypothesis and which is not discharged at any application of the rule in \(\Pi\).

\[ \begin{array}{c}
\{A\}_{j_1} \quad \{A\}_{j_k} \\
\Sigma_1 \ldots \Sigma_k \\
\{B\}_{l_1} \ldots \{B\}_{l_k} \quad (R) \\
\{C\}_I
\end{array} \]

\(J_I\) is the union of the markings of all the occurrences of \(A\) as a hypothesis in \(\Sigma_I\). Let

\(J \overset{\text{def}}{=} J_1 \cup \ldots \cup J_k\). Then,

(a) all the variables in \(proj(J)(Rv(A))\) occur free in \(NExt(\Pi)\);
(b) if \(z \in Rv(A)\) and \(z\) occurs free in \(NExt(\Pi)\), then \(z\) is an element of \(proj(J)(Rv(A))\).

The proof is continued in induction on the construction of \(\Pi\). If \(R\) is \((\land\text{-}I), (\land\text{-}E), (\lor\text{-}J), (\lor\text{-}E), (\exists\text{-}I), (\exists\text{-}E)\), or \((\exists\text{-}I)\), the proof is straightforward from the definition of
NExt. Assume that $R$ is $(\vee-E)$ and that $\Pi$ is as follows:

\[
\begin{array}{ccc}
\Sigma_0 & \{[P]_0\} & \{[Q]_0\} \\
{P \lor Q}_K & \Sigma_1 & \Sigma_2 \\
\{C\}_I & \{C\}_I & \{C\}_I \\
\end{array}
\]

By the induction hypothesis, $\text{NExt}(\Sigma_1/\{C\}_I)$ (or $\text{NExt}(\Sigma_2/\{C\}_I)$) contains all the variables in $\text{proj}(J_0)(Rv(P))$ (or $\text{proj}(J_1)(Rv(Q))$). Therefore, by the definition of NExt, the whole of $\text{tseq}(1)(\text{NExt}(\Sigma_0/\{P \lor Q\}_K))$ occur in $\text{NExt}(\Sigma_1/\{C\}_I)\theta$ and $\text{NExt}(\Sigma_2/\{C\}_I)\theta$ in $\text{NExt}(\Pi)$ where $\theta$ is the substitution. Also, $\theta$ does not instantiate any element of $Rv(A)$. Also, $\text{proj}(0)(\text{NExt}(\Sigma_0/\{P \lor Q\}_K))$ is used in the decision procedure of $\text{NExt}(\Pi)$. Then the proof of this case will be finished immediately. Other cases are similar.

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

**Theorem 3.** $\text{NExt}$ procedure and projection.

Let $A$ be a formula and $\Pi$ be its proof. Also, let $I$ be a declaration to $\Pi$ and $\text{Mark}(I, \Pi)$ be as in Theorem 2. Then, if $\text{Mark}(I, \Pi)$ succeeds, that is, if the marking condition is satisfied for any induction step proof contained in $\Pi$,

$\text{NExt}(\text{Mark}(I, \Pi)) = \text{proj}(I)(\text{Ext}(\Pi))$

holds.

**Proof.** By induction on the construction of the proof tree. Assume

$\text{NExt}(\text{Mark}(I_i, \Pi_i)) = \text{proj}(I_i)(\text{Ext}(\Pi_i))$ ($i = 1, \ldots, k$)

in the following proof tree

\[
\Pi \overset{\text{def}}{=} \Pi_1 \cdots \Pi_k(R)
\]

where $I_i$ is the marking of $\Pi_i$. In the following, the marked version of $\Sigma$ is also denoted $\Sigma$ for simplicity.

(1) Case $\Pi = \exists x.A(x)$:

$\text{NExt}(\text{Mark}(I, \Pi)) = \text{NExt}\left(\frac{\text{Mark}(\Sigma_0/\exists x.A(x))_K}{\{C\}_I} \cdot \frac{\text{Mark}(\Sigma_1/\{C\}_I)}{\{C\}_I} \cdot \text{(3-E)}\right)$

where

$\theta \overset{\text{def}}{=} \begin{cases} 
\text{proj}(L)(Rv(A(x))) / \text{tseq}(1)(\text{NExt}(\Sigma_0/\exists x.A(x)_K)), \\
\text{proj}(0)(\text{NExt}(\Sigma_0/\exists x.A(x)_K)) 
\end{cases}$ (if $0 \in K$)

$\theta \overset{\text{def}}{=} \begin{cases} 
\text{proj}(L)(Rv(A(x))) / \text{NExt}(\Sigma_0/\exists x.A(x)_K) 
\end{cases}$ (if $0 \not\in K$)
where $L$ is the union of the markings of all the occurrence of $A(x)$ as hypothesis. On the other hand, by the induction hypothesis,

$$N_{\text{Ext}}(\text{Mark}(\Sigma_0/\{\exists x . A(x)\}_K)) = \text{proj}(K)(\text{Ext}(\Sigma_0/\exists x . A(x)))$$

and $K = L + 1$ (if $0 \notin K$) or $\{0\} \cup (L + 1)$. Therefore,

$$\theta = \left\{ \frac{\text{proj}(L)(Rv(A(x)))}{\text{proj}(L)(tseq(1)(\text{Ext}(\Sigma_0/\exists x . A(x))))} \right\} \quad (\text{if } 0 \notin K)$$

$$\theta = \{ \text{proj}(L)(Rv(A(x))) / \text{proj}(L)(tseq(1)(\text{Ext}(\Sigma_0/\exists x . A(x)))) \} \quad (\text{if } 0 \notin K).$$

Then, by Lemma 3,

$$= N_{\text{Ext}}(\text{Mark}(\Sigma_1/\{C\}_I)) \theta_1$$

where $\theta_1 \stackrel{\text{def}}{=} \{Rv(A(x))/tseq(1)(\text{Ext}(\Sigma_0/\exists x . A(x))), x / \text{proj}(0)(\text{Ext}(\Sigma_0/\exists x . A(x)))\}$. Then, by the induction hypothesis,

$$= (\text{proj}(I)(\text{Ext}(\Sigma_1/C))) \theta_1 = \text{proj}(I)((\text{Ext}(\Sigma_1/C)) \theta_1) = \text{proj}(I)(\text{Ext}(\Pi)).$$

(2) Case $\Pi = \frac{\Sigma}{A \lor B} (\lor - l)_0$:

**case $0 \in I$:**

$$N_{\text{Ext}}(\text{Mark}(I, \Pi)) = N_{\text{Ext}}\left(\frac{\text{Mark}(\Sigma/\{A\}_{(I-1)(<l(A))})}{\{A \lor B\}_{I}} \frac{\Sigma}{A \lor B} (\lor - l)_0\right)$$

$$= (\text{left}, N_{\text{Ext}}(\text{Mark}(\Sigma/\{A\}_{(I-1)(<l(A))})), \text{any}[k])$$

where $k = |I| - (1 + |I - 1)(<I(A))|$. Then, by the induction hypothesis,

$$= (\text{left}, \text{proj}((I - 1)(<l(A)))(\text{Ext}(\Sigma/A)), \text{any}[k])$$

$$= \text{proj}((0) \cup ((I - 1)(<l(A)) + 1) \cup I (>l(A)))(\text{left}, \text{Ext}(\Sigma/A), \text{any}[I(B)])$$

$$= \text{proj}(I)(\text{Ext}(\Pi)).$$

**case $0 \notin I$:**

$$N_{\text{Ext}}(\text{Mark}(I, \Pi)) = N_{\text{Ext}}\left(\frac{\text{Mark}(\Sigma/\{A\}_{(I-1)(<l(A))})}{\{A \lor B\}_{I}} \frac{\Sigma}{A \lor B} (\lor - l)_0\right)$$

$$= (N_{\text{Ext}}(\text{Mark}(\Sigma/\{A\}_{(I-1)(<l(A))})), \text{any}[I])$$

where $l = |I| - |(I - 1)(<l(A))|$. Then, by the induction hypothesis,

$$= (\text{proj}((I - 1)(<l(A)))(\text{Ext}(\Sigma/A)), \text{any}[l])$$

$$= \text{proj}(((I - 1)(<l(A)) + 1) \cup I (>l(A)))(\text{left}, \text{Ext}(\Sigma/A), \text{any}[I(B)])$$

$$= \text{proj}(I)(\text{Ext}(\Pi)).$$
Redundancy-free Programs

\[
\begin{align*}
(\varepsilon -E) : C & = \text{Next}(\text{Mark}(I, \varepsilon)) \\
& = \text{Next}(\text{Mark}(E_0/\{A \lor B\}) + \text{Mark}(E_1/\{C\}) + \text{Mark}(E_2/\{C\})) \\
\text{where} & \\
J_1 & = \text{proj}(J_1)(Rv(A))/tseq(1, l(A))(\text{Ext}(E_0/A \lor B)), \\
J_2 & = \text{proj}(J_2)(Rv(B))/tseq(1, l(A)+1)(\text{Ext}(E_0/A \lor B)). \\
\text{Then, by Lemma 3,} & \\
& = \text{if left} = \text{proj}(0)(\text{Next}(\text{Mark}(E_0/A \lor B))) \\
& \quad \text{then Next(\text{Mark}(E_1/\{C\})) else Next(\text{Mark}(E_2/\{C\})))} \\
\text{where} & \\
\theta & = \begin{cases} \\
\text{proj}(J_1)(Rv(A))/tseq(1, l(A))(\text{Ext}(E_0/A \lor B)) & \\
\text{proj}(J_2)(Rv(B))/tseq(1, l(A)+1)(\text{Ext}(E_0/A \lor B)) \\
\end{cases} \\
& \text{As } 0 \notin K, \\
& = \text{if left} = \text{proj}(0)(\text{Ext}(E_0/A \lor B)) \\
& \quad \text{then proj}(I)(\text{Ext}(E_1/C)) else proj(1)(\text{Ext}(E_2/C)) \\
& \quad = \text{proj}(I)(\text{Ext}(\Pi)). \\
\text{In the case of modified } \lor \text{ code, left} = \text{proj}(0) \cdots \text{ part of the if-then-else construct is changed to } A, \text{ and the proof is similar.} \\
& \begin{align*}
[ x > 0, \ A(x) - 1 ] & \\
& = \text{Next}(\text{Mark}(E_0/A \lor B)) + \text{Mark}(E_1/\{C\}) + \text{Mark}(E_2/\{C\}) \\
& \quad \text{where} & \\
J_1 & = \text{proj}(J_1)(Rv(A))/tseq(1, l(A))(\text{Ext}(E_0/A \lor B)), \\
J_2 & = \text{proj}(J_2)(Rv(B))/tseq(1, l(A)+1)(\text{Ext}(E_0/A \lor B)). \\
\text{Then, by Lemma 3,} & \\
& = \text{if left} = \text{proj}(0)(\text{Next}(\text{Mark}(E_0/A \lor B))) \\
& \quad \text{then Next(\text{Mark}(E_1/\{C\})) else Next(\text{Mark}(E_2/\{C\})))} \\
\text{where} & \\
\theta & = \begin{cases} \\
\text{proj}(J_1)(Rv(A))/tseq(1, l(A))(\text{Ext}(E_0/A \lor B)) & \\
\text{proj}(J_2)(Rv(B))/tseq(1, l(A)+1)(\text{Ext}(E_0/A \lor B)) \\
\end{cases} \\
& \text{As } 0 \notin K, \\
& = \text{if left} = \text{proj}(0)(\text{Ext}(E_0/A \lor B)) \\
& \quad \text{then proj}(I)(\text{Ext}(E_1/C)) else proj(1)(\text{Ext}(E_2/C)) \\
& \quad = \text{proj}(I)(\text{Ext}(\Pi)). \\
\end{align*}

(4) Case \(\Pi = \begin{align*}
[ x > 0, \ A(x) - 1 ] & \\
& = \text{Next}(\text{Mark}(E_0/A \lor B)) + \text{Mark}(E_1/\{C\}) + \text{Mark}(E_2/\{C\}) \\
& \quad \text{where} & \\
J_1 & = \text{proj}(J_1)(Rv(A))/tseq(1, l(A))(\text{Ext}(E_0/A \lor B)), \\
J_2 & = \text{proj}(J_2)(Rv(B))/tseq(1, l(A)+1)(\text{Ext}(E_0/A \lor B)). \\
\text{Then, by Lemma 3,} & \\
& = \text{if left} = \text{proj}(0)(\text{Next}(\text{Mark}(E_0/A \lor B))) \\
& \quad \text{then Next(\text{Mark}(E_1/\{C\})) else Next(\text{Mark}(E_2/\{C\})))} \\
\text{where} & \\
\theta & = \begin{cases} \\
\text{proj}(J_1)(Rv(A))/tseq(1, l(A))(\text{Ext}(E_0/A \lor B)) & \\
\text{proj}(J_2)(Rv(B))/tseq(1, l(A)+1)(\text{Ext}(E_0/A \lor B)) \\
\end{cases} \\
& \text{As } 0 \notin K, \\
& = \text{if left} = \text{proj}(0)(\text{Ext}(E_0/A \lor B)) \\
& \quad \text{then proj}(I)(\text{Ext}(E_1/C)) else proj(1)(\text{Ext}(E_2/C)) \\
& \quad = \text{proj}(I)(\text{Ext}(\Pi)). \\
\end{align*}
\[ N \text{Ext}(\text{Mark}(I, \Pi)) = N \text{Ext} \left( \frac{\text{Mark} \left( \frac{\Sigma_0}{\{A(0)\}_I} \right) \text{Mark} \left( \frac{\Sigma_1}{\{A(x)\}_I} \right)}{\forall x. A(x)} \right) \]

\[ = \mu \bar{z} \lambda x. \text{if } x = 0 \text{ then } N \text{Ext} \left( \text{Mark} \left( \frac{\Sigma_0}{\{A(0)\}_I} \right) \right) \text{ else } N \text{Ext} \left( \text{Mark} \left( \frac{\Sigma_1}{\{A(x)\}_I} \right) \right) \theta \]

where \( \theta \overset{\text{def}}{=} \{ \text{proj}(I)(\text{Rv}(A(x-1))/\bar{z}(x-1)) \} \) and \( \bar{z} \) is a sequence of new variables of length \(|I|\).

By Lemma 3, \( (N \text{Ext}(\text{Mark}(\Sigma_0/A(x)))) \theta \overset{\text{def}}{=} (N \text{Ext}(\text{Mark}(\Sigma_1/A(x)))) \theta \) where \( \theta \overset{\text{def}}{=} \{ \text{Rv}(A(x-1))/\bar{z}(x-1) \} \) and \( \bar{z}' \) is a sequence of new variables of length \( |A(x)| \) such that \( \text{proj}(I)(\bar{z}') = \bar{z} \). Then, by the induction hypothesis,

\[ = \mu \bar{z} \lambda x. \text{if } x = 0 \text{ then } \text{proj}(I)(\text{Ext}(\Sigma_0/A(0))) \text{ else } \text{proj}(I)(\text{Ext}(\Sigma_1/A(x))) \theta_1 \]

\[ = \mu \bar{z}. \text{proj}(I)(\lambda x. \text{if } x = 0 \text{ then } \text{Ext}(\Sigma_0/A(0)) \text{ else } \text{Ext}(\Sigma_1/A(x))) \theta_1 \]

\[ = \mu \bar{z}. \text{proj}(I)(P) \]

where \( P \overset{\text{def}}{=} \lambda x. \text{if } x = 0 \text{ then } \text{Ext}(\Sigma_0/A(0)) \text{ else } \text{Ext}(\Sigma_1/A(x))) \theta_1 \). Assume that \( P \) is expanded to \( (M_0, \ldots, M_{n-1}) \) \( (n = |A(0)| = |A(x)|) \). Then,

\[ \mu \bar{z}'. P = (f_0, \ldots, f_{n-1}) \]

where \( f_i = \mu z_i. N_i, \bar{z}' = (z_0, \ldots, z_{n-1}) \) and \( N_i \) is obtained from \( M_i \) by substituting other \( f_j \)'s to free occurrences of \( z_j \)'s \( (j \neq i) \) as explained in section 2.1. Let \( I = \{i_1, \ldots, i_m\} \), then \( \mu \bar{z}. \text{proj}(I)(P) = \mu \bar{z}. (M_{i_1}, \ldots, M_{i_m}) \).

Note that by Lemma 3, any variable, \( z \), such that \( z \in \bar{z}' \) and \( z \notin \bar{z} \) does not occur in \( M_i \) \( (1 \leq p \leq m) \). Therefore, \( f_k \) \( (k \notin I) \) does not occur in \( f_k \) \( (1 \leq p \leq m) \). Hence, \( \mu \bar{z}. (M_{i_1}, \ldots, M_{i_m}) = (f_{i_1}, \ldots, f_{i_m}) = \text{proj}(I)(\mu \bar{z}'. P) = \text{proj}(I)(\text{Ext}(\Pi)) \).

(7) Other cases are similar or easy.

7. Example

Here, the example of a prime number checker program is investigated.

7.1. EXTRACTION OF A PRIME NUMBER CHECKER PROGRAM BY Ext

The specification of the program which takes any natural number as input and returns the Boolean value, \( T \), when the given number is prime, otherwise returns \( F \) is as follows:

\textbf{Specification}

\[ \forall p : \text{nat. } (p \geq 2 \Rightarrow \exists b : \text{bool. } ((\forall d : \text{nat. } (1 < d < p \Rightarrow \neg (d \mid p)) \land b = T) \lor (\exists d : \text{nat. } (1 < d < p \land (d \mid p)) \land b = F))) \]

where \( (x \mid y) \overset{\text{def}}{=} \exists z, y = x \cdot z. \)

This specification can be proved by using the following lemma:

\textbf{Lemma.} \( \forall p : \text{nat. } \forall z : \text{nat. } (z \geq 2 \Rightarrow A(p, z)) \)
where

\[ A(p, z) \triangleq \exists b : \text{bool. } (P_0(p, z, b) \lor P_1(p, z, b)) \]

\[ P_0(p, z, b) \triangleq \forall d : \text{nat. } (1 < d < z \Rightarrow \neg (d \mid p)) \land b = T \]

\[ P_1(p, z, b) \triangleq \exists d : \text{nat. } (1 < d < z \land (d \mid p)) \land b = F. \]

**Proof of Specification**

\[ \Sigma \]

\[ \forall p : \text{nat.} \forall z : \text{nat.} (z \geq 2 \Rightarrow A(p, z)) \]

\[ \forall p : \text{nat.} (p \geq 2 \Rightarrow A(p, p)) \]

The proof of the lemma, which will be denoted \( \Pi_{\text{Len}} \) in the following, is given in the Appendix, and the program extracted by \( \text{Ext} \) is as follows:

\[ \text{prime} \triangleq \lambda p. \text{Ext}(\Pi_{\text{Len}})(p)(p) \]

\[ \text{Ext}(\Pi_{\text{Len}}) \]

\[ \lambda p. \mu(z_0, z_1, z_2, z_3). \lambda z. \text{if } z = 0 \text{ then any}[4] \text{ else if } z = 1 \text{ then any}[4] \text{ else if } z = 2 \text{ then } (T, \text{left}, \text{any}[2]) \text{ else if } \text{proj}(0)((z_1, z_2, z_3)(z-1)) = \text{left} \text{ then } (T, \text{left}, \text{any}[2]) \text{ else } (F, \text{right}, z-1, \text{tseq}(1)(\text{Ext}(\text{prop})(p)(z-1))) \]

\( \text{Ext}(\text{prop}) \) is a function which takes natural numbers, \( m \) and \( n \), as input and returns \( (\text{right}, d) \) if \( m \) can be divided by \( n \) \( (m = d \cdot n) \) and \( (\text{left}, \text{any}[1]) \) otherwise.

\( \text{Ext}(\Pi_{\text{Len}}) \) is a multi-valued recursive call function which calculates a sequence of terms of length four. The Boolean value which denotes whether the given number is prime is the 0th element of the sequence.

7.2. Program Extraction by Declaration, Marking and \( \text{NExt} \)

(1) Declaration.

The realizing variables of the specification are sequence of variables of length four: \( (w_0, w_1, w_2, w_3) \). \( w_0, w_1, w_2, \) and \( w_3 \) are the variable for \( \exists \) symbol on \( b : \text{bool} \), the variable for \( \lor \) symbol which connects \( P_0 \) and \( P_1 \), the variable for \( \exists \) symbol on \( d : \text{nat} \), and the variable for \( \exists \) symbol in \( (d \mid p) \).

As the only information needed here is the value of \( b, w_0 \) should be specified, that is, the declaration is \{0\}. 

Redundancy-free Programs 61

where

\[ A(p, z) \triangleq \exists b : \text{bool. } (P_0(p, z, b) \lor P_1(p, z, b)) \]

\[ P_0(p, z, b) \triangleq \forall d : \text{nat. } (1 < d < z \Rightarrow \neg (d \mid p)) \land b = T \]

\[ P_1(p, z, b) \triangleq \exists d : \text{nat. } (1 < d < z \land (d \mid p)) \land b = F. \]
(2) Marking and Backtracking.
It turns out that the marked proof tree, which is obtained with the declaration \{0\} and \texttt{Mark}, does not satisfy the marking condition. The main part of the proof of the lemma is performed in mathematical induction. The marking of the conclusion of the induction proof is \{0\}, and the marking of an occurrence of the induction hypothesis (actually the induction hypothesis occurs only once in the proof) is \{1\}(\not\in\{0\}). Therefore, \texttt{Mark} fails. Then the declaration is enlarged to \{0, 1\} and the marking procedure is performed again. The marking of the occurrence of the induction hypothesis is \{1\} this time, and the marking condition is satisfied. Then, \texttt{NExt} is ready to extract the program.

(3) Program Extraction by \texttt{NExt}.
The \texttt{NExt} procedure extracts the following program from the marked proof tree obtained in (2).

\begin{verbatim}
prime' = \lambda p. \texttt{NExt}(\texttt{Mark}(\Pi_{len}))(p)(p)
NExt(\texttt{Mark}(\Pi_{len}))
  = \lambda p. \mu(z_0, z_1).
    \lambda z. \text{if } z = 0
    \text{then any}[2]
    \text{else if } z = 1
      \text{then any}[2]
    \text{else if } z = 2
      \text{then } (T, \text{left})
    \text{else if } z_1(z - 1) = \text{left} \cdots (*)
      \text{then if proj}(0)(\texttt{Ext}(\texttt{prop})(p)(z - 1)) = \text{left}
        \text{then } (T, \text{left})
      \text{else } (F, \text{right})
    \text{else } (F, \text{right}).
\end{verbatim}

Comparing the above code with \texttt{Ext}(\Pi_{len}), the reason why the declaration should be \{0, 1\} (not \{0\}) is as follows. To calculate the Boolean value which indicates whether the input natural number is prime, information as to whether the input can be divided by a natural number smaller than the input is necessary. That information is given as the 1st code, \texttt{left} or \texttt{right}, of the term sequence calculated by the main loop of the multi-valued recursive call function.

Note that only the 1st element of the sequence is calculated at the recursive call step (see \(*)\) part). This is what the marking of the induction hypothesis, \{1\}, means.

(4) Alternative Extraction.
The extracted program will be more efficient if the whole proof is normalized. In fact, \texttt{red}_\nu can be applied to the proof of the specification. If the declaration is \{1\}, a program which returns the constants, \texttt{left} and \texttt{right}, instead of Boolean values is extracted. The same program can also be extracted by changing the specification to the following and giving the declaration, \{0\}.

\begin{verbatim}
\forall p : \text{nat. } (p \geq 2 \Rightarrow ((\forall d : \text{nat. } (1 < d < p \Rightarrow \neg(d \mid p))) \vee (\exists d : \text{nat. } (1 < d < p \wedge (d \mid p)))))
\end{verbatim}
7.3. PROOF TREE ANALYSIS

By using the proof theoretic characterization of critical applications explained in section 4.3, the reason why the declaration should be enlarged to \( \{0, 1\} \) in the previous subsection can be explained in terms of the structure of the marked proof tree.

7.3.1. MAIN PATHS FROM INDUCTION HYPOTHESIS

The main part of the proof of the lemma is performed in mathematical induction, and Fig. 1 is the skeleton of the proof tree of the induction step. This is a part of the proof tree which is a collection of the formula occurrences along the main paths from an occurrence of the induction hypothesis which actually occurs only once in the proof. The formula occurrences in Fig. 1 with the index number, (1), (2), \ldots, can be found in the proof tree in Appendix with the same index numbers. The discharged hypotheses of some of \((\forall I)\) and \((\forall E)\) applications are not shown in the figure because they are not along the main paths.

Formulas \( A \) to \( F \) are in the following form:

\[
A(z) = z \geq 2 \Rightarrow A(p, z) = * \Rightarrow B(z)
\]

\[
B(z) = \exists b. P_0(p, z, b) \lor P_1(p, z, b) = \exists b. C(z, b)
\]

\[
C(z, B) = P_0(p, z, B) \lor P_1(p, z, B) = D_0(z, B) \lor D_1(z, B)
\]

\[
D_0(z, B) = (\forall d. (1 < d < z \Rightarrow \neg (d | p)) \land B = T) = E_0(z) \land *
\]

\[
D_1(z, B) = (\exists d. (1 < d < z \land (d | p)) \land B = F) = E_1(z) \land *
\]

\[
E_0(z) = \forall d. (1 < d < z \Rightarrow \neg (d | p)) = \forall d. F_0(z)
\]

\[
E_1(z) = \exists d. (1 < d < z \land (d | p)) = \exists d. F_1(z)
\]

\[
F_0(z) = 1 < d < z \Rightarrow \neg (d | p) = * \Rightarrow G_0
\]

\[
F_1(z) = 1 < d < z \land (d | p) = G_1(z) \land G_2
\]

\[
G_0 = \neg (d | p) \quad G_1(z) = 1 < d < z \quad G_2 = (d | p)
\]
where * is the abbreviation of some particular formula. \( C(z, b), D_0(z, b), \) and \( D_1(z, b) \) are abbreviated to \( C(z), D_0(z), \) and \( D_1(z) \).

There are three main paths from the occurrence of the induction hypothesis, \( A(x - 1) \):

\[
S_0 \overset{\text{def}}{=} (1), (2), (3), (4), (5), (6), (7), (8), (9), (10), (11), (12), (13), (14), (15), (16), (17), (18), (19)
\]

\[
S_1 \overset{\text{def}}{=} (1), (2), (3), (20), (21), (22), (23), (24), (25), (26), (27), (28), (29), (30), (15), (16), (17), (18), (19)
\]

and

\[
S_2 \overset{\text{def}}{=} (1), (2), (3), (20), (21), (31), (32), (25), (26), (27), (28), (29), (30), (15), (16), (17), (18), (19).
\]

There are five non-trivial segments along \( S_0, S_1 \) and \( S_2 \):

- \((a) (7), (8) (b) (13), (14), (15), (16), (17) (c) (30), (15), (16), (17) (d) (18), (19) (e) (26), (27)\).

Segments \((b) \) and \((c) \) will be critical after the marking.

### 7.3.2. Initial Marking

The marked proof tree initiated by the declaration, \( \{0\} \), is given in Fig. 2.

![Figure 2](image)

After the marking, the non-trivial segments, \((b) \) and \((c) \), become proper segments. Also, because the major premises of the \((\exists-E)\) and \((\forall-E)\) applications, \((2) \) and \((3) \), are along the main paths, \( S_1, S_2 \) and \( S_3 \), \((b) \) and \((c) \) are critical segments. The indispensable marking number of the occurrence of \( B(x - 1) \) is 1, so that the marking of the induction hypothesis contains 1, which is not contained in the declaration, \( \{0\} \).
7.3.2. RE-MARKING

The marking of the induction hypothesis is \( \{1\} \) (\( \not\subseteq \{0\} \)), so that the declaration is enlarged to \( \{0,1\} \). Perform the marking again to obtain the marked proof tree given in Fig. 3. In the marked proof tree of Fig. 3, the marking number, 1, of the formula occurrence indexed by (3) is the indispensable marking number, but it is contained in the declaration.

\[
\begin{align*}
[D_0(z-1)]_{\phi} & \quad (\lambda \cdot E) \\
E_0(z-1)_{\phi} & \quad (\forall \cdot E) \\
F_0(z-1)_{\phi} & \quad (\Rightarrow \cdot E) \\
G_0_{\phi} & \quad (\forall \cdot E) \\
G_0_{\phi} & \quad (\Rightarrow \cdot I) \\
F_0(z)_{\phi} & \quad (\forall \cdot I) \\
E_0(z)_{\phi} & \quad (\lambda \cdot I) \\
D_0(z,T)_{\phi} & \quad (\forall \cdot I) \\
C(z,T)_{[0]} & \quad (\exists \cdot I) \\
B(z)_{[0,1]} & \quad (v \cdot E) \\
B(z)_{[0,1]} & \quad (\exists \cdot I) \\
A(z)_{[0,1]} & \quad (\Rightarrow \cdot I) \\
A(z)_{[0,1]} & \quad (v \cdot E) \\
\end{align*}
\]

**Figure 3**

8. Conclusion

A method to extract redundancy-free realizer codes from constructive proofs was presented in this paper. The method allows fine-grained specification of redundancy, and most of the analysis of redundancy is performed automatically. The set notation in the Nuprl and ITT by Göteborg group and \( \diamond \)-bounded formulas in PX are also the notations to specify the redundancy. For example, by transforming a specification, \( \forall x. \exists y. \exists z. \exists w. A(x, y, z, w) \), to \( \forall x. \exists y. \exists z. \diamond \exists w. A(x, y, z, w) \), a function that calculates the values of \( y \) and \( z \) can be extracted in PX. However, a new proof must be given when the specification is changed. Also, if a function that calculates only the values of \( y \) and \( w \) are needed, \( \diamond \)-notation cannot handle it. The set notation is similar in this respect. On the other hand, one should just declare \( \{0,2\} \) to the specification in the method presented in the paper.

Paulin-Mohring's version of the Calculus of Constructions also allows specification of redundancy, but it is as fine-grained as the set notation and \( \diamond \)-bounded formulas. Her idea is to make a copy of the calculus with the constant \( Prop \) replaced by a new constant \( Spec \), and the theorems and proofs are described in a mixture of the original calculus and the copy of it. The program extraction is performed only on the copy of the calculus. Our method uses a system of notations called declaration and marking instead of a copy of the original formal system. The basic idea is to perform the program analysis of redundancy at proof level, and the metalogical system of notations is sufficient for the analysis. The analysis of redundancy is performed by the marking procedure, which may fail if the marking condition is not satisfied. However, the marking condition can be satisfied by implementing the backtracking mechanism given in section 4.
References


Appendix. Proof of Lemma ($\Pi_{Leu}$)

MAIN PROOF

\[
\frac{\Sigma_0}{\forall z. (\forall z \geq 2 \Rightarrow A(p, z))} \quad (\forall-I)
\]

Extracted Code by Ext:

\[
\lambda p. \mu(z_0, z_1, z_2, z_3). \lambda z.
\]

if $z = 0$ then Ext($\Sigma_0/z = 2 \Rightarrow A(p, 0)$) else Ext($\Sigma_1/z = 2 \Rightarrow A(p, z)$) $\sigma_0$

where $\sigma_0 \equiv \{ Rv(z - 1 \geq 2 \Rightarrow A(p, z - 1))/\{z_0, z_1, z_2, z_3\}(z - 1) \}$

PROOF of $p \vdash \forall z \geq 2 \Rightarrow A(p, z)$ ($\Sigma_0$)

\[
\frac{[0 \geq 2]}{A(p, 0)} \quad (\bot-E)
\]

PROOF OF $p, z - 1, z - 1 \geq 2 \Rightarrow A(p, z - 1) \leftarrow z \geq 2 \Rightarrow A(p, z)$ ($\Sigma_i$)

\[
\begin{array}{c}
\begin{array}{c}
\frac{[z \geq 2]}{z = 1 \lor z \geq 2} \quad (\ast) \\
\Sigma_{11}
\end{array}
\frac{A(p, z)}{1 < d < 2} (1.\text{-E})
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\frac{[z \geq 2, p, z - 1]}{z - 1 \geq 2 \Rightarrow A(p, z - 1)}
\end{array}
\frac{\frac{[z = 1]}{\Rightarrow I}}{z \geq 2 \Rightarrow A(p, z)} (18)
\frac{\frac{z \geq 2 \Rightarrow A(p, z)}{(\lor-E)}}{z \geq 2 \Rightarrow A(p, z)(19)}
\end{array}
\]

Extracted Code by Ext (modified $\land$-code):

\[
\begin{array}{c}
\begin{array}{c}
\text{if } z = 1 \text{ then } \text{any}[4] \text{ else } \text{Ext}(\Sigma_{11}/z \geq 2 \Rightarrow A(p, z))
\end{array}
\end{array}
\]

PROOF OF $p, z - 1, z \geq 2, z - 1 \geq 2 \Rightarrow A(p, z - 1) \leftarrow z \geq 2 \Rightarrow A(p, z)$ ($\Sigma_{11}$)

\[
\begin{array}{c}
\begin{array}{c}
\frac{[z = 2]}{z = 2 \lor z \geq 3} \quad (\ast)
\end{array}
\frac{\frac{[z \geq 3, z - 1]}{\Sigma_{110}}}{\frac{A(p, z)(17)}{A(p, z)(16)}} (\lor-E)
\end{array}
\]

\[
\begin{array}{c}
\frac{\frac{[z \geq 2]}{\Sigma_{111}}}{\frac{A(p, z)(18)}{A(p, z)(18) (\Rightarrow I)}}
\end{array}
\]

Extracted Code by Ext (modified $\land$ code):

\[
\begin{array}{c}
\begin{array}{c}
\text{if } z = 2 \text{ then Ext}(\Sigma_{110}/A(p, z)) \text{ else Ext}(\Sigma_{111}/A(p, z))
\end{array}
\end{array}
\]

PROOF OF $\Sigma_{110}$

\[
\begin{array}{c}
\begin{array}{c}
\frac{[d : \text{nat}] [1 < d < 2]}{1 < d < 2 \Rightarrow \neg(d \mid p)} (\ast)
\end{array}
\frac{\neg(d \mid p)}{(\land-\text{E})}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\frac{\frac{\forall d. (1 < d < 2 \Rightarrow \neg(d \mid p))}{\neg T} (\Rightarrow I)}{T = T (\lor-I)}
\end{array}
\frac{\neg T (\ast)}{P_0(p, 2, T) \lor P_1(p, 2, T)}
\end{array}
\]

\[
\begin{array}{c}
\frac{[z = 2]}{A(p, 2) \Rightarrow A(p, z)} (\Rightarrow I)
\end{array}
\]

Extracted Code by Ext: ($T$, left, any[2])

PROOF OF $\Sigma_{111}$

\[
\begin{array}{c}
\begin{array}{c}
\frac{[z \geq 3]}{z - 1 \geq 2} (\ast)
\end{array}
\frac{\exists b. P_0(p, z - 1, b) \lor P_1(p, z - 1, b)(22)}{A(p, z)(16) \Rightarrow A(p, z)} (\Rightarrow I)
\end{array}
\]

\[
\begin{array}{c}
\frac{[z \geq 3]}{\exists b. P_0(p, z - 1, b) \lor P_1(p, z - 1, b)(22) (\Rightarrow I)}\Pi_0 (\exists-\text{E})
\end{array}
\]

Extracted Code by Ext: ($T$, left, any[2])
where \( \Pi_0 \) is as follows:

\[
\begin{bmatrix}
\{ b, z \geq 3, z - 1 \} & \{ b, z - 1 \} \\
P_0(p, z - 1, b) & P_1(p, z - 1, b)
\end{bmatrix}
\]

\[
\frac{\Sigma_{1110}}{\Sigma_{1111}}
\]

\[
\frac{A(p, z)}{A(p, z)} \quad (\lor-E)
\]

Extracted Code by Ext:

\[
\begin{cases}
\text{if } \text{proj}(0)(w_1, w_2, w_3) = \text{left then } \text{Ext}(\Sigma_{1110}/A(p, z)) \\
\text{else } \text{Ext}(\Sigma_{1111}/A(p, z))
\end{cases}
\]

where \((w_0, w_1, w_2, w_3) \overset{\text{def}}{=} \text{Rv}(z - 1 \geq 2 \Rightarrow A(p, z - 1)) \) and \( \sigma_1 \overset{\text{def}}{=} \{ b/ w_0, \text{Rv}(P_0(p, z - 1, b) \lor P_1(p, z - 1, b))/ (w_1, w_2, w_3) \} \).

**Proof of \( \Sigma_{1110} : b, z \geq 3, z - 1, P_0(p, z - 1, b) \vdash A(p, z) \)**

\[
\begin{bmatrix}
\{ z - 1 \} \\
\{ \neg(z - 1 | p) \} \\
\{ (z - 1 | p) \}
\end{bmatrix}
\]

\[
\frac{\Sigma_{1100}}{\Sigma_{1101}}
\]

\[
\frac{A(p, z)}{A(p, z)} \quad (\lor-E)
\]

where \( \Pi_1 \) is as follows:

Extracted Code by Ext:

\[
\begin{cases}
\text{if } \text{proj}(0)(\text{Prop.})(p)(z - 1) = \text{left then } \text{Ext}(\Sigma_{1100}/A(p, z)) \\
\text{else } \text{Ext}(\Sigma_{1101}/A(p, z)) \sigma_2
\end{cases}
\]

where \( \sigma_2 \overset{\text{def}}{=} \{ \text{Rv}((z - 1 | p))/ \text{tseq}(1)(\text{Prop.})(p)(z - 1) \} \}

The proof of Prop is skipped.

**Proof of \( \Sigma_{1100} : \neg(z - 1 | p), P_0(p, z - 1, b) \vdash A(p, z) \)**

\[
\frac{\overline{T}^{(*)}}{P_0(p, z, T) \quad \left( \land-I \right)}
\]

\[
\frac{\overline{T}^{(*)}}{P_0(p, z, T) \lor P_1(p, z, T) \quad \left( \lor-I \right)}
\]

\[
\exists b, P_0(p, z, b) \lor P_1(p, z, b) \quad \left( \lor-I \right)
\]
where $\Pi_2$ is as follows:

\[
\begin{array}{c}
1 < d < z \\
1 < d < z - 1 \\
\forall d \neq z - 1
\end{array}
\]

\[
\begin{array}{c}
\vdash \neg((d | p) \\ [d] \forall d.1 < d < z - 1 \vdash \neg((d | p) \\ [d] \forall d.1 < d < z - 1 \vdash \neg((d | p) \\ [d] \forall d.1 < d < z - 1 \vdash \neg((d | p) \\
\end{array}
\]

Extracted Code by $Ext: (T, \text{left}, \text{any}[2])$

**Proof of $\Sigma_{1101}: z - 1, z \geq 3, (z - 1 | p), \vdash A(p, z)$**

\[
\begin{array}{c}
[z - 1: \text{nat}] \\
1 < z - 1 < z \\
\exists d.1 < d < z \land (z - 1 | p) \\
\end{array}
\]

\[
\begin{array}{c}
\vdash F \equiv F \\
\exists b. P_0(p, z, b) \lor P_1(p, z, b)
\end{array}
\]

Extracted Code by $Ext: (F, \text{right}, z - 1, Rv((z - 1 | p))$

**Proof of $b, z - 1, P_1(p, z - 1, b) \vdash A(p, z) (\Sigma_{1111})$**

\[
\begin{array}{c}
[z - 1: \text{nat}] \\
1 < z - 1 < z \\
\exists d.1 < d < z \land (d | p) \\
\end{array}
\]

\[
\begin{array}{c}
\vdash F \equiv F \\
\exists b. P_0(p, z, b) \lor P_1(p, z, b)
\end{array}
\]

where $\Pi_3$ is as follows:

\[
\begin{array}{c}
1 < d \\
\land (d | p) \\
1 < d < z - 1 \\
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

Extracted Code by $Ext: (F, \text{right}, (d, Rv((d | p))))\sigma_3$ where $\sigma_3 \equiv (d | w_2, Rv((d | p)) / w_3)$. 

\[
\begin{array}{c}
1 < d \\
\land (d | p) \\
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]

\[
\begin{array}{c}
\vdash (z - 1: \text{nat}) \\
\vdash (d | p) \\
\vdash (d | p) \\
\vdash (d | p)
\end{array}
\]