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

Yukihide Takayama

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

Abstract

This paper presents a formulation of program extraction system from constructive proofs. It is designed as a simple system and is very similar to the ordinary Gentzen style of natural deduction as compared to other constructive logics to clarify the relationship between constructive proofs and programs. Proofs and λ -expressions are regarded as different things, and they are linked by the notion of proof compilation based on realizability.

The program extraction algorithm is given explicitly, and two simple examples, simulation of course of value induction by mathematical induction and, by using this as a user-defined rule of inference, extraction of a gcd program, are investigated to check how the well known method of proof normalisation works as an optimisation technique and what kinds of inefficiency remain after the proof normalisation. Subtle points about the execution of extracted codes are also discussed. Finally, modified V-code is introduced as an easy but powerful optimisation technique,

1. Introduction

Constructive proofs can be seen as a very high level description of algorithm, as has been discussed by many computer scientists and logicians in terms of the relationship between intuitionism and computation, and programming systems such as the Nuprl system [Constable 86] and the PX system [Hayashi 87] have been implemented. These systems have the facility to extract executable codes from constructive proofs, and are based on the notion of formulae-as-types or realizability interpretation. Their main purpose is to pursue high level expressive power of constructive logic. However, studying techniques to extract good code is also an important research problem.

This paper presents another formulation of the program extraction system which is obtained from a subset of Sato's constructive logic QJ [Sato 85]. It is quite simple compared to Martin-Löf's theory of types [Martin-Löf 84] and Hayashi's PX. Roughly, it is an intuitionistic version of a Gentzen-type system of natural deduction with a simple induction schema. It has some program constructs such as λ -expressions and primitive type structures. Primitive types such as natural number type and natural number list type are provided as built-in types, so that types cannot be manipulated as objects. This simple system, however, provides a very clear insight on the relationship between proofs and programs, and makes it much easier to work on the proof transformation technique and efficient code generation. In addition, a second order feature is introduced to write user-defined rules of inference, although this system does not pursue polymorphism as investigated in [Coquand 85].

Section 2 gives our formalism of programs and logic and the algorithm of proof compilation. Section 3 gives the method of user-defined rules description and program scheme extraction by introducing predicate variables and the second order universal quantifier, $\forall^2$. Section 4 shows the extraction of a gcd program by using a user-defined course of value induction rules. Section 5 outlines the operational semantics of the extracted codes. Section 6 works on the optimisation technique based on normalisation of proof trees, and introduces another powerful optimisation technique, the modified V-code. The conclusion and future problems are given in Section 7.

2. Proof Compiler

This section gives the formalization and the naive version of the program extraction algorithm based on the notion of *q-realizability*. They are based on a subset of the 1985 version of QJ [Sato 85], except for the higher order features, predicate variable and second order $\forall^2$ -I/E rules.

2.1 Notational Preliminaries

(1) Type expressions

- 1) *nat* ... Type expression of natural number.
- 2) *prop* ... Type expression of proposition. $P : \text{prop}$ means that P is a well formed formula.
- 3) $\text{Type}_1 \rightarrow \text{Type}_2$... The type of function from type Type_1 to type Type_2 .
- 4) $\text{Type}_1 \times \dots \times \text{Type}_n$... the Cartesian product type.

(2) Term expressions

- 1) $0, 1, 2, \dots$... Elements of type *nat*.
- 2) *left/right* ... Constants.
- 3) $x, y, z, \dots$ Individual variables are written in lower case letters. All the variables have types, and $x : \text{Type}$ is read as "variable x has type Type ".
Type declarations are usually omitted in the following description.
- 4) $(\text{Term}_0, \dots, \text{Term}_n)$ or simply $\text{Term}_0, \dots, \text{Term}_n$
Sequence of terms. If Term_i are of types Type_i ($0 \leq i \leq n$), then the above terms are seen to be of Cartesian product type: $\text{Type}_1 \times \dots \times \text{Type}_n$.
Sequences of variables are denoted as $\bar{x}, \bar{y}, \bar{z}, \dots$

- 5) $\lambda(x_0, \dots, x_n). \text{Term}$ $(0 \leq n)$ $\dots$ Lambda abstraction.
 - 6) *if A then Term₁ else Term₂* $\dots$ A is a higher order equation or inequation defined in (3) 1).
 - 7) $\mu(z_0, \dots, z_n). A(z_0, \dots, z_n)$ $(0 \leq n)$ $\dots$ μ is the fixed point operator.
 - 8) $\text{Term}(\text{Term}_0)(\text{Term}_1) \dots (\text{Term}_n)$ $\dots$ Application. Associate to left.
 - 9) $x \bmod y$ $\dots$ Residue of fraction of x by y .
 - 10) $\text{proj}(i)$ $\dots$ Projection function of type $\text{Type}_0 \times \dots \times \text{Type}_i \times \dots \times \text{Type}_n \rightarrow \text{Type}_i$ where $(0 < n, 0 \leq i \leq n)$
 - 11) $l(\text{SEQUENCE})$ $\dots$ Length of the sequence, SEQUENCE .
 - 12) $\text{any}[N]$ $\dots$ Sequence of arbitrary N codes. $(N > 0)$
 - 13) succ/pred $\dots$ Successor/predecessor function on nat
 - 14) $f(x_0, \dots, x_n)$ $(0 \leq n)$ $\dots$ Term which contains free variables $x_0, \dots, x_n$.
- (3) Formulae
- 1) $\text{Term}_1 = \text{Term}_2, \text{Term}_1 \leq \text{Term}_2, \text{Term}_1 < \text{Term}_2$
Higher order equation/inequation of terms.
 - 2) $P, Q, \dots$ $\dots$ Predicate variables. These are of type *prop*.
 - 3) $P(x), Q(x), \dots$ Predicates that have x as free variables. $P(x)$ is also denoted in an abstraction formula, $\lambda x. P$. Substitution of a term, t , to x that occurs free in P is defined as a kind of β -reduction: $(\lambda x. P)(t) \rightarrow P(t)$
 - 4) $\perp$ $\dots$ Contradiction. Note that $\neg A \stackrel{\text{def}}{=} A \supset \perp$.
 - 5) Definition of formulae
 1. Higher order equations and inequations of terms are atomic formulae.
 2. $\perp$ is an atomic formula.
 3. If P and Q are formulae, $P \wedge Q, P \vee Q, P \supset Q$ are formulae.
 4. If $P(x)$ is a formula containing x free, $\forall x : \text{Type}. P(x), \exists x : \text{Type}. P(x)$ are formulae where Type can be any type other than *prop*.
 5. If Q is a predicate variable $\forall^2 Q : \text{prop}. P(Q)$ is a formula.
- Formulae are regarded as of type *prop*.

(4) Proof trees

Proof trees are written in the ordinary natural deduction style as in [Prawitz 65]. Subtrees are often abbreviated as Π_i or, when the free variable, x , in the subtree should be stressed, $\Pi_i(x)$.

2.2 Inference Rules on Logical Constants and Equalities

The inference rules used in this paper are listed here.

(1) Introduction Rules

$$\frac{A \quad B}{A \wedge B} \quad \frac{A}{A \vee B} \quad \frac{B}{A \vee B} \quad \frac{[A]}{B} \quad \frac{[x : \text{Type}]}{A(x)} \quad \frac{t : \text{Type} \quad A(t)}{\exists x : \text{Type}. A(x)} \quad \frac{B}{A \supset B} \quad \frac{A(x)}{\forall x : \text{type}. A(x)}$$

(2) Elimination Rules

$$\frac{A \wedge B}{A} \quad \frac{A \wedge B}{B} \quad \frac{[A] \quad [B]}{A \vee B \quad C \quad C} \quad \frac{A \quad A \supset B}{B} \quad \frac{t : \text{Type} \quad \forall x : \text{Type}. A(x)}{A(t)} \quad \frac{\exists x : \text{Type}. A(x) \quad C}{C} \quad \frac{\perp}{A}$$

(3) Induction Rule

$$\frac{[x : \text{nat}, A(x), x > 0] \quad A(0) \quad A(\text{succ}(x))}{\forall x : \text{nat}. A(x)} \text{ (nat-ind)}$$

(4) Rules on Equalities

$$\begin{array}{c}
\frac{t = s \text{ (in Type)} \quad A(t)}{A(s)} (= 1) \qquad \frac{t : \text{Type}}{t = t \text{ (in Type)}} (= 2) \qquad \frac{t = s \text{ (in Type)}}{s = t \text{ (in Type)}} (= 3) \\
\frac{p = q \text{ (in Type)} \quad q = r \text{ (in Type)}}{p = r \text{ (in Type)}} (= 4)
\end{array}$$

QJ also has rules of arithmetic, formation of terms and formulae, and type inferences. However, these rules are not necessary as far as program extraction is concerned. The names of these rules are abbreviated to * in the following description.

The following additional rules are of higher order logic and are not contained in the original version of QJ.

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

2.3 Program Extraction Algorithm

The program extraction algorithm is given here. This algorithm performs q-realizability interpretation of QJ and the soundness of the extracted code, realizer, is proved in [Sato 85].

(1) Additional notations for algorithm description

• $Rv(A)$

Realizing variable sequence. Realizing variables are sequences of variables to which realizer codes for the formula are assigned. $Rv(A)$ is defined as follows:

1. $Rv(A) \stackrel{def}{=} (nil)$, if A is atomic.
2. $Rv(A \wedge B) \stackrel{def}{=} \text{concatenation of } Rv(A) \text{ and } Rv(B)$.
3. $Rv(A \vee B) \stackrel{def}{=} \text{concatenation of a new variable, } z, Rv(A), \text{ and } Rv(B)$.
4. $Rv(A \supset B) \stackrel{def}{=} Rv(B)$.
5. $Rv(\forall x : \text{Type}. A(x)) \stackrel{def}{=} Rv(A(x))$.
6. $Rv(\exists x : \text{Type}. A(x)) \stackrel{def}{=} \text{concatenation of a new variable, } z, \text{ and } Rv(A(x))$.
7. $Rv(P) \stackrel{def}{=} Rv(P(x))$, if $P : \text{prop}$ has x as free variables.

• $Ext(\frac{A}{B}(Rule))$

Top level procedure (function) of program extraction. It is often abbreviated to $Ext(B)$ in the situation where A and $Rule$ are clear. When a conclusion, B , depends on a list of formulae, Γ , this procedure is denoted by $Ext(\Gamma \vdash B)$. When $Ext(A)$, where A is a formula in Γ , is needed in $Ext(\Gamma \vdash B)$, $Rv(A)$ is used as the value of $Ext(A)$.

• $\sigma, \tau, \dots$

Substitution. These are defined as $\{x_0/Term_0, \dots, x_n/Term_n\}$, which means to substitute $Term_i$ for x_i ($0 \leq i \leq n$) that occurs free in a given expression. Application of σ is denoted as $Term\sigma$.

• (nil)

Denotes empty code.

(2) Definition of Ext procedure

$$\begin{array}{c}
Ext(\frac{A \quad B}{A \wedge B}(\wedge\text{-I})) \stackrel{def}{=} Ext(A), Ext(B) \\
Ext(\frac{A \wedge B}{A}(\wedge\text{-E})) \stackrel{def}{=} proj(0)(A \wedge B) \qquad Ext(\frac{A \wedge B}{B}(\wedge\text{-E})) \stackrel{def}{=} proj(1)(A \wedge B) \\
Ext(\frac{A}{A \vee B}(\vee\text{-I})) \stackrel{def}{=} left, Ext(A) \qquad Ext(\frac{B}{A \vee B}(\vee\text{-I})) \stackrel{def}{=} right, Ext(B)
\end{array}$$

$$\begin{array}{c}
[A] \quad [B] \\
Ext(\frac{A \vee B}{C} \frac{C}{C} (\vee\text{-E})) \\
\stackrel{def}{=} \begin{cases} Ext(A \vdash C)\sigma_0 & \text{if } proj(0)(Ext(A \vee B)) = left; \\ Ext(B \vdash C)\sigma_1 & \text{if } proj(0)(Ext(A \vee B)) = right; \\ \text{if } left = proj(0)(Ext(A \wedge B)) \text{ then } Ext(A \vdash C)\sigma_0 & \text{otherwise} \\ \text{else } Ext(B \vdash C)\sigma_1 & \end{cases}
\end{array}$$

where $\sigma_0 \stackrel{def}{=} \{Rv(A)/proj(1)(Ext(A \vee B))\}$ $\sigma_1 \stackrel{def}{=} \{Rv(B)/proj(1)(Ext(A \vee B))\}$

$$\begin{array}{c}
[A] \\
Ext(\frac{B}{A \supset B} (\supset\text{-I})) \stackrel{def}{=} \begin{cases} Ext(B) & \text{if } Rv(A) = (nil); \\ \lambda Rv(A). Ext(A \vdash B) & \text{otherwise} \end{cases}
\end{array}$$

$$\begin{array}{c}
A \quad A \supset B \\
Ext(\frac{B}{B} (\supset\text{-E})) \stackrel{def}{=} Ext(A \supset B)(Ext(A)) \\
[x : Type]
\end{array}$$

$$\begin{array}{c}
A(x) \\
Ext(\frac{A(x)}{\forall x : Type. A(x)} (\forall\text{-I})) \stackrel{def}{=} \lambda x. Ext(A(x))
\end{array}$$

$$\begin{array}{c}
t : Type \quad \forall x : Type. A(x) \\
Ext(\frac{A(x)}{A(t)} (\forall\text{-E})) \stackrel{def}{=} Ext(\forall x : Type. A(x))(t)
\end{array}$$

$$\begin{array}{c}
t : Type \quad A(t) \\
Ext(\frac{A(t)}{\exists x : Type. A(x)} (\exists\text{-I})) \stackrel{def}{=} t, Ext(A(t)) \\
[x : Type, A(x)]
\end{array}$$

$$\begin{array}{c}
\exists x : Type. A(x) \quad C \\
Ext(\frac{C}{C} (\exists\text{-E})) \stackrel{def}{=} Ext(x : Type, A(x) \vdash C)\sigma
\end{array}$$

where $\sigma \stackrel{def}{=} \{x/proj(0)(Ext(\exists x : Type. A(x))), Rv(A(x))/proj(1)(Ext(\exists x : Type. A(x)))\}$
 $[x : nat, A(x)]$

$$\begin{array}{c}
A(0) \quad A(succ(x)) \\
Ext(\frac{A(succ(x))}{\forall x : nat. A(x)} (nat\text{-ind}))
\end{array}$$

$$\stackrel{def}{=} \begin{cases} \mu \bar{z}. \lambda x. \text{if } x = 0 \text{ then } Ext(A(0)) \text{ else } Ext(x : nat, A(x) \vdash A(succ(x)))\sigma & \text{where } \bar{z} = Rv(A(x)), \text{ and } \sigma = \{\bar{z}/\bar{z}(pred(x))\} \\ \lambda x. \text{if } x = 0 \text{ then } Ext(A(0)) \text{ else } Ext(x : nat, A(x) \vdash A(succ(x))) & \text{if } Rv(A(x)) \text{ does not occur in } Ext(x : nat, A(x) \vdash A(succ(x))) \end{cases}$$

$$\begin{array}{c}
\perp \\
Ext(\frac{\perp}{A} (\perp\text{-E})) \stackrel{def}{=} \begin{cases} (nil) & \text{if } Rv(A) \text{ is } (nil); \\ any[l(Rv(A))] & \text{otherwise} \end{cases}
\end{array}$$

$$\begin{array}{c}
t = s \text{ (in Type)} \quad A(t) \\
Ext(\frac{A(t)}{A(s)} (=1)) \stackrel{def}{=} Ext(A(t))
\end{array}$$

$$\begin{array}{c}
t : Type \\
Ext(\frac{t : Type}{t = t \text{ (in Type)}} (=2)) \stackrel{def}{=} (nil) \quad Ext(\frac{t = s \text{ (in Type)}}{s = t \text{ (in Type)}} (=3)) \stackrel{def}{=} (nil)
\end{array}$$

$$\begin{array}{c}
p = q \text{ (in Type)} \quad q = r \text{ (in Type)} \\
Ext(\frac{q = r \text{ (in Type)}}{p = r \text{ (in Type)}} (=4)) \stackrel{def}{=} (nil) \quad Ext(\frac{A}{B} (*)) \stackrel{def}{=} (nil)
\end{array}$$

(3) The *Ext* procedure for ($\forall^2\text{-I/E}$) will be given in the next section.

3. Proof Schema Using Propositional Variables

3.1 Proof of Course of Value Induction

As is well known in mathematical logic, the course of value induction schema

$$\forall^2 P : prop. (\forall x : nat. (\forall y : nat. (y < x \supset P(y)) \supset P(x)) \supset \forall z : nat. P(z)) \quad [\text{COV-IND}]$$

can be proven by mathematical induction in second order logic. Any proof tree of the first order theorem that uses the course of value induction rule can be transformed into one that uses the mathematical induction rules as follows. Let Q be an individual predicate. The proof of $\forall x : \text{nat. } \forall y : \text{nat. } ((y < x \supset Q(y)) \supset Q(x))$ is called *course of value proof* in the following descriptions.

$$\frac{\Pi_0}{\frac{\forall x : \text{nat. } (\forall y : \text{nat. } (y < x \supset Q(y)) \supset Q(x))}{\forall z : \text{nat. } Q(z)}} \text{ (course of value induction)}$$

This tree is transformed into the following tree, called COV-TREE:

$$\frac{\frac{\Pi_0}{\forall x : \text{nat. } (\forall y : \text{nat. } (y < x \supset Q(y)) \supset Q(x))} \quad \Pi_0}{\forall z : \text{nat. } Q(z)} \quad [\text{COV-TREE}]$$

where Π_0 is as follows:

$$\frac{\frac{\Pi_1}{Q : \text{prop}} \quad \frac{\frac{\Pi_2}{\forall x : \text{nat. } (\forall y : \text{nat. } (y < x \supset P(y)) \supset P(x)) \supset \forall z : \text{nat. } P(z)}{\text{COV-IND}} (\forall^2\text{-I})}{\forall x : \text{nat. } (\forall y : \text{nat. } (y < x \supset Q(y)) \supset Q(x)) \supset \forall z : \text{nat. } Q(z)} (\forall^2\text{-E})$$

Π_2 is shown in detail Appendix 1.

3.2 Proof Compilation Algorithm for $(\forall^2\text{-I/E})$

The predicate variables and $\forall^2\text{-I/E}$ rules are used to handle user-defined rules of inference such as the course of value induction schema explained in 3.1.

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

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

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

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

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

$$\frac{\frac{\Pi_0}{Q : \text{prop}} \quad \frac{\frac{\Pi_1(P)}{F(P)}}{\forall^2 P : \text{prop. } F(P)}}{F(Q)} \xRightarrow{[P : \text{prop}]} \frac{\Pi_0}{\frac{[Q : \text{prop}]}{\Pi_1(Q)} F(Q)}$$

3.3 Proof Compilation of Course of Value Schema

The following code is generated from the proof of COV-IND given in 3.1 by the proof compilation algorithm given in 2.3 and 3.2. AA and $CODE_1$ are extracted from Π_{21} and *Theorem 1* in Appendix 1.

$$\lambda RV(P). \lambda RV(P(x)). \lambda z. RV(P(x))(z)(A_0(z)) \quad [\text{COV-CODE}]$$

where

$$A_0 \stackrel{\text{def}}{=} \mu RV(P(y)). \lambda x. \text{if } x = 0 \text{ then } \lambda y. \text{any}[l(RV(P(y)))] \\ \text{else } \lambda y. \text{if left} = AA(x)(y) \text{ then } RV(P(y))(pred(x))(y) \\ \text{else } RV(P(x))(x)(RV(P(y))(pred(x)))$$

$$AA \stackrel{\text{def}}{=} \mu z. \lambda x. \text{if } x = 0 \text{ then } \lambda y. \text{if } y = 0 \text{ then right else any}[1] \\ \text{else } \lambda y. \text{if left} = CODE_1(y) \text{ then left} \\ \text{else if left} = z(pred(x))(pred(y)) \text{ then left else right}$$

$$CODE_1 \stackrel{\text{def}}{=} \mu z. \lambda k. \text{if } k = 0 \text{ then left else if left} = z \text{ then right else right}$$

The code $\text{left} = AA(x)(y)$ in A_0 is the conditional equation which is logically equivalent to $y < x$. $RV(P(x))$ is equal to $RV(\forall x : \text{nat}. (\forall y : \text{nat}. (y < x \supset P(y)) \supset P(x)))$. The proof of this part must be given as a course of value proof, and the proof contains the computational meaning of how to construct the justification of $P(x)$ by using the justifications of $P(y)$ (for all y s.t. $y < x$). $RV(P(y))$ denotes the realizing variables of $\forall y : \text{nat}. (y < x \supset P(y))$.

4. Simple Example: GCD Program

The GCD program is taken as a simple example which uses COV-IND.

4.1 GCD Proof

The specification of GCD program is as follows:

$$\forall n : \text{nat}. \forall m : \text{nat}. \exists d : \text{nat}. (d \mid n \wedge d \mid m)$$

where for $p : \text{nat}$ and $q : \text{nat}$, $p \mid q \stackrel{\text{def}}{=} \exists r : \text{nat}. q = r \cdot p$.

The specification and proof that the constructed natural number is actually a maximal one which satisfies the specification is omitted here for simplicity. However, the natural number which satisfies this condition is constructed in the proof given below. The proof is called *proof of GCD program* or *GCD proof* in the following description.

The course of value proof of this specification is shown in Appendix 2.

4.2 Proof Compilation of GCD Proof

The proofs in Appendices 1 and 2 give a gcd program. The following code is obtained by the method given in 3.1 and the proof compilation.

$$gcd_0 \stackrel{\text{def}}{=} f(g)$$

where

$$f \stackrel{\text{def}}{=} \{\text{The code obtained from COV-CODE}(Rv(Q)) \text{ by applying } \beta\text{-reduction}\}$$

$$g \stackrel{\text{def}}{=} \lambda n. \lambda (z_0, z_1, z_2). \text{if left} = CODE_1(n) \text{ then } CODE_2 \text{ else } CODE_3$$

$$CODE_2 \stackrel{\text{def}}{=} \lambda m. (m, (\lambda p. 0)(m), (\lambda q. 1)(m))$$

$$CODE_{3,n} \stackrel{def}{=} \lambda m. (z_0(m \bmod n)(n), z_2(m \bmod n)(n), \\ z_1(m \bmod n)(n) + z_2(m \bmod n)(n) \cdot \frac{m - (m \bmod n)}{n})$$

Here, $Q \stackrel{def}{=} \Lambda x. \forall m : nat. \exists d : nat. (d \mid x \wedge d \mid m)$, and $(z_0, z_1, z_2) \stackrel{def}{=} Rv(\forall l. (l < n \supset Q(l)))$. Let $(w_0, w_1, w_2) \stackrel{def}{=} Rv(Q(x))$. Then

$$f \stackrel{def}{=} \lambda(w_0, w_1, w_2). \lambda z. (w_0, w_1, w_2)(z)(A_1(z))$$

where

$$A_1 \stackrel{def}{=} \mu(z_0, z_1, z_2). \lambda x. \text{if } x = 0 \text{ then } \lambda y. \text{any}[3] \\ \text{else } \lambda y. \text{if left} = AA(x)(y) \text{ then } (z_0, z_1, z_2)(pred(x))(y) \\ \text{else } (w_0, w_1, w_2)(x)((z_0, z_1, z_2)(pred(x)))$$

5. Execution of the Extracted Codes

5.1 Tiny Quty Interpreter

Tiny Quty is a subset of Quty [Sato 87]. It is a non-typed sequential functional language to describe executable codes extracted from constructive proofs. The syntax of **Tiny Quty** is given in Section 2 as the definition of term expressions and higher order (in)equations. For simplicity, the language presented here has no syntax for list structure. The chief difference from ordinal functional languages is that it allows sequences of variables as parameters of the fixed point operator, μ , i.e., multi-valued functions can be written.

The interpreter of **Tiny Quty** is an ordinal call-by-value evaluator of λ -expressions. However, the following features should be noted.

- (1) $f \stackrel{def}{=} \mu(z_0, \dots, z_n). t(z_0, \dots, z_n)$ is regarded as a sequence of single-valued functions $f_0, \dots, f_n$, and f_i ($0 \leq i \leq n$) is defined as $proj(i)(f)$. The following reduction can be performed on f :

$$\mu(z_0, \dots, z_n). t(z_0, \dots, z_n) \longrightarrow t(f_0, \dots, f_n)$$

- (2) $\lambda(x_0, \dots, x_n). t(x_0, \dots, x_n)$ is regarded as another description of $\lambda x_0 \dots \lambda x_n. t(x_0, \dots, x_n)$.

- (3) $(\lambda(x_0, \dots, x_n). t_0(x_0, \dots, x_n))(t_1)$ can be reduced to $t_0(A_0, \dots, A_n)$

where $(A_0, \dots, A_n) = SPLIT(t_1)$. The function, $SPLIT$, is the procedure that splits terms into sequences of terms if it is possible.

- (4) $\text{any}[N](Term)$ is reduced to $Term$.

5.2 Evaluation of the GCD Code

The code extracted in Section 4 is a program that takes two natural numbers as inputs and returns the sequence of three natural numbers. The first element of the pair is the gcd of the inputs, and the other two elements are verification information that can be seen as the decoded proof to show that the first element of the pair is actually the gcd. If one is interested in only the value of gcd, the extracted code should be properly transformed into a single-valued function.

5.3 Problem of $\perp$ -E Code

The code, $\lambda y. \text{any}[l(Rv(P(y)))]$, which occurs in [COV-CODE] and is extracted from the proof which uses $\perp$ -E rule, is evaluated when the whole code is executed. The reason is as follows: $\forall x. \forall y. (y < x \supset P(y))$ in Main Proof Tree of Appendix 1 is proved by (*nat-ind*), and the

base case is proved by $\perp$ -E rule. The base case proof corresponds to the termination condition in terms of proof compilation, so that it must always be evaluated just before the execution terminates.

The extraction algorithm for $\perp$ -E rule is defined as follows in 2.3:

$$Ext(\frac{\perp}{A}(\perp\text{-E})) \stackrel{def}{=} \begin{cases} (nil) & \text{if } Rv(A) \text{ is a nil sequence;} \\ any[l(Rv(A))] & \text{otherwise} \end{cases}$$

The operational semantics of $any[...]$ code is defined as $any[N](Term) \longrightarrow Term$ in 5.1. However, it is an experimental treatment.

The treatment around the realizer code of $\perp$ -E rule is clearer when the $\perp$ -contraction rules [Troelstra 83] are performed on proof trees using $\perp$ -E rule. The tree, Π_1 in Appendix 1, is the base case proof of $\forall x. \forall y. (y < x \supset P(y))$.

Let $P(x) \stackrel{def}{=} \forall m : nat. \exists d : nat. (d \mid x \wedge d \mid m)$ in Π_1 . After a sequence of $\perp$ -contractions, Π_1 is as follows. Note that d, p and q can be any elements of nat .

$$\begin{array}{c} \frac{\frac{\frac{p}{(*)} \quad \frac{\frac{[y : nat]}{[y < 0]} \quad \frac{[y : nat]}{\neg(y < 0)}(*)}{\perp}}{y = p \cdot d} \quad (\exists\text{-I}) \quad \frac{\frac{q}{(*)} \quad \frac{\frac{[y : nat]}{[y < 0]} \quad \frac{[y : nat]}{\neg(y < 0)}(*)}{\perp}}{m = q \cdot d} \quad (\exists\text{-I})}{\frac{d \mid y \quad d \mid m}{d \mid y \wedge d \mid m} \quad (\wedge\text{-I})} \quad (\exists\text{-I}) \\ \frac{\frac{\frac{\exists d. d \mid y \wedge d \mid m}{P(y)} \quad (\forall\text{-I})}{y < 0 \supset P(y)} \quad (\supset\text{-I})}{\forall y : nat. (y < 0 \supset P(y))} \quad (\forall\text{-I}) \end{array}$$

Any proof in which $\perp$ -E rule is used to derive a formula which contains at least one logical constant can be transformed into the proof in which the $\perp$ -rule is not used except to derive an atomic formula by using the same method. Consequently, the following *Ext* procedure suffices to extract codes from the $\perp$ -rule.

$$Ext(\frac{\perp}{A}(\perp\text{-E})) \stackrel{def}{=} (nil) \quad \text{where } l(A) = 0.$$

The extracted code with this procedure is $\lambda y. \lambda m. (d, p, q)$. d, p and q can be any natural number, so that they can be m because $m : nat$ is an assumption of the $(\forall\text{-I})$ rule application. Consequently, $\lambda y. any[l(Rv(P(y)))]$ is actually $\lambda y. \lambda m. (m, m, m)$.

This problem can be avoided if the mathematical induction on $x - 1$ is used in combination with divide and conquer strategy on $x = 0 \vee x > 1$. In this case, Π_1 should be as follows:

$$\begin{array}{c} \frac{\frac{[y : nat]}{y = 0} \quad \frac{[y < 1]}{P(0)}(*) \quad \Pi_0}{P(y)} \quad (=1) \\ \frac{y < 1 \supset P(y)}{\forall y : nat. (y < 1 \supset P(y))} \quad (\supset\text{-I}) \quad (\forall\text{-I}) \end{array}$$

This proof gives the termination condition of the recursive call program, and the code for $\perp$ -E rule which is extracted from the proof of the case $x = 0$ is never executed.

6. Optimisation Technique

In the PX system [Hayashi 87], the optimisation of extracted codes proceeds as follows: if the code of the form $(\lambda x. A)(B)$ is extracted in the process of proof compilation, then perform β -reduction of the code immediately. The optimisation technique of proof compilation can be presented more systematically.

6.1 Proof Normalisation and Partial Evaluation of Programs

Normalisation of proofs corresponds to partial evaluation of extracted codes from the proofs. The following are the normalisation rules given in [Prawitz 65].

(1) $\forall$ -normalisation

$$\frac{\frac{\Pi(a)}{P(a)}(\forall\text{-I})}{\frac{\forall x. P(x)}{P(t)}(\forall\text{-E})} \Rightarrow \frac{\Pi(t)}{P(t)}$$

(2) $\supset$ normalisation

$$\frac{\frac{\Pi_0}{A} \quad \frac{\frac{[A]}{\Pi_1} \quad \frac{B}{A \supset B}(\supset\text{-I})}{B}(\supset\text{-E})}{\frac{\Pi_0}{[A]} \quad \frac{B}{B}} \Rightarrow \frac{\Pi_0}{B}$$

There are other rules of normalisation such as the $\exists$, $\wedge$ and $\vee$ -normalisation rules; however, they are not effective for the optimisation in proof compilation as can be seen by the definition of the *Ext* procedure. Rules (1) and (2) correspond to β -reduction of λ -expressions.

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

$$\begin{array}{ccccccc} \text{Proof}_0 & \xrightarrow{\text{Norm}_0} & \text{Proof}_1 & \xrightarrow{\text{Norm}_1} & \dots & \xrightarrow{\text{Norm}_n} & \text{Proof}_n \\ \downarrow \text{Ext} & \curvearrowright & \downarrow \text{Ext} & & & \curvearrowright & \downarrow \text{Ext} \\ \text{Code}_0 & \xrightarrow{\beta\text{-red.}} & \text{Code}_1 & \xrightarrow{\beta\text{-red.}} & \dots & \xrightarrow{\beta\text{-red.}} & \text{Code}_n \end{array}$$

Following this diagram, optimisation facilities can be realized in either of two ways:

1. By implementing a proof normaliser. Proofs are normalized first, and then compiled.
2. By implementing a partial evaluator built in the proof compiler. Proofs are compiled first, then the partial evaluation method of the functional programming language is applied to the extracted codes.

From an aesthetic point of view, both proofs and codes should be transformed simultaneously to maintain a clear correspondence between proofs and programs in terms of proof compilation. The order of applying the normalisation rules can be arbitrary. If the normalisation rules are applied from the leaves of proof trees, this corresponds to call-by-value evaluation of the programs extracted from the proof trees. If they are applied from the bottom of proof trees, it means call-by-name evaluation. Tiny Quty programs are evaluated with call-by-value strategy. However, this is just for efficiency of runtime evaluation.

6.2 Example of Proof Normalisation

For the GCD proof given in 4.1, first, the $\forall$ -normalisation rule can be applied to the proof of $m \mid 0$ and $m \mid m$ in Π_1 in Appendix 2, as follows.

$$\frac{\frac{\frac{[m : \text{nat}]}{0 : \text{nat}}(*) \quad \frac{\frac{0 : \text{nat}}{0 = 0 \cdot p} \quad [p : \text{nat}]}{\exists d' : \text{nat}. 0 = d' \cdot p}}{\forall p : \text{nat}. p \mid 0}}{m \mid 0} \Rightarrow \frac{\frac{[m : \text{nat}]}{0 : \text{nat}}(*) \quad \frac{0 : \text{nat}}{0 = 0 \cdot m}}{m \mid 0}$$

$$\begin{array}{c}
\frac{\frac{\frac{1 : nat^{(*)}}{1 : nat^{(*)}} \quad \frac{1 : nat^{(*)}}{1 = 1 \cdot q} \quad [q : nat]}{\exists d'' : nat. q = d'' \cdot q}}{\frac{[m : nat] \quad \forall q : nat. q \mid q}{m \mid m}} \Rightarrow \frac{1 : nat^{(*)} \quad \frac{1 : nat^{(*)}}{m = 1 \cdot m} \quad [m : nat]}{m \mid m}
\end{array}$$

By this normalisation, $CODE_2$ is translated to the following $CODE_{22}$:

$$CODE_{22} \stackrel{def}{=} \lambda m. (m, 0, 1)$$

As the $(\supset -E)$ rule is used in the course of value induction schema given in 3.1, the $\supset$ -normalisation rule can be applied:

$$\begin{array}{c}
\frac{\frac{\Pi_0}{\forall x. (\forall y. (y < x \supset Q(y)) \supset Q(x))} \quad \frac{\frac{\Pi'}{\forall z. Q(z)}}{\forall x. (\forall y. (y < x \supset Q(y)) \supset Q(x)) \supset \forall z. Q(z)} (\supset -I)}{\forall z. Q(z)} (\supset -E) \\
\Rightarrow \\
\frac{\frac{\Pi_0}{\forall x. (\forall y. (y < x \supset Q(y)) \supset Q(x))} (\forall -I)}{\forall z. Q(z)}
\end{array}$$

where

Π_{11} is the course of value proof, and

$$\frac{\frac{\Pi'}{\forall z. Q(z)}}{\forall x. (\forall y. (y < x \supset Q(y)) \supset Q(x)) \supset \forall z. Q(z)}$$

is the proof of course of value schema by mathematical induction.

By this transformation, β -reduction of $f(g)$ is performed, and the $\forall$ -normalisation rule can also be applied to Π_{22} and Π_3 in Appendix 1 combined with the course of value proof given in Appendix 2, then the code is as follows:

$$gcd_1 \stackrel{def}{=} \lambda z. \lambda(z_0, z_1, z_2). \text{ if } left = CODE_1(z) \text{ then } CODE_{22} \text{ else } CODE_{3,z}(A_2(z))$$

$$\text{where } A_2 \stackrel{def}{=} A_1\{(w_0, w_1, w_2) / \lambda(z_0, z_1, z_2). \text{ if } left = CODE_1(x) \text{ then } CODE_{22} \text{ else } CODE_{3,x}\}$$

6.3 Modified V-Code

$left = CODE_1(y)$ in code AA corresponds to the proof of $y = 0 \vee y > 0$. However, none of the normalisation rules in 6.1 can be applied. Most of the execution of the gcd program is that of AA , the code extracted from the proof of $y < x + 1 \vdash y < x \vee y = x$, and $left = AA(x)(y)$ is logically equivalent to $y < x$, as explained in 4.2.

On the other hand, as given in 4.1, $n = 0 \vee n > 0$ ($n : nat$) and $y < x + 1 \vdash y < x \vee y = x$ are proved by mathematical induction. However, in practical situations, it is not efficient if we must always prove well known properties of natural numbers of this kind strictly by induction. For these reasons, the following modification is introduced in the proof compilation:

$$\text{Ext}(\frac{[A] \quad [B]}{A \vee B} \frac{C}{C} (V-E)) \stackrel{\text{def}}{=} \text{if } A \text{ then } \text{Ext}(A \vdash C) \text{ else } \text{Ext}(B \vdash C)$$

when A and B are equations or inequations of natural numbers.

In this case, the proof of $A \vee B$ can be omitted by declaring this formula as an axiom.

By this optimisation, the gcd code obtained in 6.2 is changed as follows:

$$\text{gcd}_2 \stackrel{\text{def}}{=} \lambda z. \lambda(z_0, z_1, z_2). \text{ if } z = 0 \text{ then } \text{CODE}_{22} \text{ else } \text{CODE}_{3,z}(A_3(z))$$

where

$$A_3 \stackrel{\text{def}}{=} \mu(z_0, z_1, z_2). \lambda x. \text{ if } x = 0 \text{ then } \lambda y. \text{ any}[3] \\ \text{ else } \lambda y. \text{ if } y < x \text{ then } (z_0, z_1, z_2)(\text{pred}(x))(y) \\ \text{ else } (\lambda(z_0, z_1, z_2). \text{ if } x = 0 \text{ then } \text{CODE}_{22} \\ \text{ else } \text{CODE}_{3,x}((z_0, z_1, z_2)(\text{pred}(x))))$$

7. Conclusion

This paper presented a simple proof compiler system based on the notion of realizability. A higher order feature was introduced to handle the description of user-defined rules of inference. Optimisation was handled with the notion of proof normalisation. Modified V-code was also introduced as a powerful technique of optimisation.

Acknowledgments

Thanks must go to Dr. Aiba, Dr. Murakami, and Mr. Sakai of ICOT, and to Mr. Kameyama and Professor Sato at Tohoku University, who gave me many useful suggestions.

REFERENCES

- [Constable 86] Constable, R. L. et al., "Implementing Mathematics with the Nuprl Proof Development System", Prentice-Hall, 1986
- [Coquand 85] Coquand, T. and Huet, G., "Constructions: A Higher Order Proof System for Mechanizing Mathematics", *EUROCAL 85*, Linz, Austria, 1985
- [Hayashi 87] Hayashi, S. and Nakano, H., "PX: A Computational Logic", RIMS-573, Research Institute for Mathematical Sciences, Kyoto University, 1987
- [Martin-Löf 84] Martin-Löf, P., "Intuitionistic Type Theory", Bibliopolis, Napoli, 1984
- [Prawitz 65] Prawitz, D., "Natural Deduction", Almqvist & Wiksell, 1965
- [Sato 85] Sato, M., "Typed Logical Calculus", TR-85-13, Department of Computer Science, University of Tokyo, 1985
- [Sato 87] Sato, M., "Quty: A Concurrent Language Based on Logic and Function", *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, 1987
- [Troelstra 73] A. S. Troelstra, "Mathematical Investigation of Intuitionistic Arithmetic and Analysis", *Lecture Notes in Mathematics Vol. 344*, Springer-Verlag, 1973

Appendix 1 Proof of COV-IND

Main Proof Tree:

$$\frac{\frac{[z : \text{nat}] \quad \frac{\frac{\Pi_1 \quad \Pi_2}{\forall x : \text{nat}. \forall y : \text{nat}. (y < x \supset P(y))}^{\text{(nat-ind)}}}{\forall y : \text{nat}. (y < z \supset P(y))}}{P(z)} \quad \Pi_3}{\forall z : \text{nat}. P(z)} \quad \Pi_3 (\supset -E)$$

$$\frac{\forall z : \text{nat}. P(z)}{\forall x : \text{nat}. (\forall y : \text{nat}. (y < x \supset P(y)) \supset P(x)) \supset \forall z : \text{nat}. P(z)}$$

Π_1 : (Base case proof of induction)

$$\frac{\frac{[y : nat] \quad \neg(y < 0) (*)}{[y < 0] \quad \neg(y < 0)} \quad \perp}{P(y)} \quad \frac{y < 0 \supset P(y)}{\forall y : nat. (y < 0 \supset P(y))}$$

Π_2 : (Step case proof of induction)

$$\frac{\frac{[x : nat] \quad [y : nat] \quad [y < succ(x)] \quad [y < x] \quad \frac{[y : nat] \quad [HYP] \quad (\forall-E)}{y < x \supset P(y)} \quad \frac{[HYP] \quad \Pi_{22} \quad (\supset-E)}{P(x)} \quad \frac{[y = x] \quad P(x)}{P(y)} \quad \Pi_{21}}{P(y)} \quad (\forall-E) \quad \frac{y < succ(x) \supset P(y)}{\forall y : nat. (y < succ(x) \supset P(y))}$$

where $HYP \stackrel{def}{=} \forall y : nat. (y < x \supset P(y))$

Π_{21} (Proof of $x : nat, y : nat, y < succ(x) \vdash y < x \vee y = x$)

$$\frac{\frac{\frac{\Pi_{212} \quad \Pi_{213} \quad \Pi_{214}}{y < x + 1 \vee y = x + 1} (\forall-E) \quad \frac{y < x + 2 \supset y < x + 1 \vee y = x + 1}{\forall y : nat. y < x + 2 \supset y < x + 1 \vee y = x + 1}}{\frac{[x : nat] \quad \Pi_{211} \quad \forall x : nat. \forall y : nat. y < x + 1 \supset y < x \vee y = x} (nat-ind)} \quad \frac{[y : nat] \quad \forall y : nat. y < x + 1 \supset y < x \vee y = x}{[y < succ(x)] \quad y < x + 1 \supset y < x \vee y = x} \quad y < x \vee y = x$$

Π_{211} (Base case proof of nat-ind)

$$\frac{\frac{\frac{\bar{0} (*)}{0 = 0} \quad [0 < 1] \quad \frac{0 = 0 \wedge 0 < 1}{0 = 0}}{0 < 0 \vee 0 = 0} \quad \frac{\frac{[y + 1 < 1] (*) \quad [y : nat] (*)}{y < 0 \quad \neg(y < 0)} \quad \perp}{y + 1 < 0 \vee y + 1 = 0} \quad \frac{0 < 1 \supset 0 < 0 \vee 0 = 0 \quad y + 1 < 1 \supset y + 1 < 0 \vee y + 1 = 0}{\forall y : nat. y < 1 \supset y < 0 \vee y = 0} (nat-ind)$$

Π_{212}

$$\frac{[y : nat] \quad Theorem \ 1}{y = 0 \vee y > 0} (\forall-E)$$

Theorem 1 is a simple theorem of number theory that states any natural number is equal to 0 or larger than 0. This can be proved by mathematical induction as follows:

$$\frac{\frac{\bar{0} (*)}{0 = 0} \quad \frac{[k = 0] \quad \frac{k + 1 = 1 (*)}{k + 1 > 0} \quad \frac{1 > 0 (*)}{k + 1 > 0} \quad \frac{[k > 0] \quad \frac{k + 1 > 1 (*)}{k + 1 > 0} \quad \frac{1 > 0 (*)}{k + 1 > 0}}{[k = 0 \vee k > 0] \quad \frac{k + 1 = 0 \vee k + 1 > 0 \quad k + 1 = 0 \vee k + 1 > 0}{k + 1 = 0 \vee k + 1 > 0}} \quad \frac{0 = 0 \vee 0 > 0}{\forall k : nat. k = 0 \vee k > 0} (nat-ind)$$

$$\begin{array}{c}
\Pi_{213} \quad \frac{\frac{[x : \text{nat}]}{[y = 0]} (*)}{\frac{0 < x + 1}{y < x + 1}} (*) \\
\frac{y < x + 1}{y < x + 1 \vee y = x + 1} \\
\Pi_{214} : (y > 0, y < x + 2, HYP'' \vdash y < x + 1 \vee y = x + 1) \\
\Pi_{2141} \quad \frac{\frac{[y - 1 < x]}{y < x + 1} (*)}{\frac{y < x + 1 \vee y = x + 1}{y < x + 1 \vee y = x + 1}} \quad \frac{\frac{[y - 1 = x]}{y = x + 1} (*)}{y < x + 1 \vee y = x + 1} \\
\Pi_{2141} : (y > 0, y < x + 2, HYP'' \vdash y - 1 < x \vee y - 1 = x) \\
\frac{\frac{[y > 0]}{y - 1 < x + 1} (*) \quad \frac{[y < x + 2]}{y - 1 < x + 1 \supset y - 1 < x \vee y - 1 = x} (*) \quad \frac{[HYP'']}{y - 1 < x \vee y - 1 = x} (\vee\text{-E})}{y - 1 < x \vee y - 1 = x}
\end{array}$$

where $HYP'' \stackrel{\text{def}}{=} \forall y : \text{nat}. (y < x + 1 \supset y < x \vee y = x)$.

Π_{22} : (Application of the hypothesis)

$$\frac{[x : \text{nat}] \quad [\forall x : \text{nat}. (\forall y : \text{nat}. (y < x \supset P(y)) \supset P(x))]}{\forall y : \text{nat}. (y < x \supset P(y)) \supset P(x)} (\vee\text{-E})$$

Π_3 : (Application of the hypothesis)

$$\frac{[z : \text{nat}] \quad [\forall x : \text{nat}. (\forall y : \text{nat}. (y < x \supset P(y)) \supset P(x))]}{\forall y : \text{nat}. (y < z \supset P(y)) \supset P(z)} (\vee\text{-E})$$

Appendix 2: The course of value proof of the GCD Proof

Let $Q(x) \stackrel{\text{def}}{=} \forall m : \text{nat}. \exists d : \text{nat}. (d \mid x \wedge d \mid m)$.

Main Proof Tree:

$$\begin{array}{c}
\frac{[n : \text{nat}] \quad \text{Theorem 1}}{n = 0 \vee n > 0} (\vee\text{-E}) \quad \Pi_1 \quad \Pi_2 (\vee\text{-E}) \\
\frac{Q(n)}{\forall l : \text{nat}. (l < n \supset Q(l)) \supset Q(n)} \\
\frac{\forall n : \text{nat}. (\forall l : \text{nat}. (l < n \supset Q(l)) \supset Q(n))}{\quad}
\end{array}$$

$\Pi_1 : (n = 0 \vdash Q(n))$

$$\begin{array}{c}
\frac{[m : \text{nat}] \quad \text{Theorem 2} \quad [m : \text{nat}] \quad \text{Theorem 3}}{m \mid 0 \quad m \mid m} \\
\frac{[m : \text{nat}] \quad m \mid 0 \wedge m \mid m}{\exists d : \text{nat}. (d \mid 0 \wedge d \mid m)} \\
\frac{[n = 0] \quad Q(0)}{Q(n)} (\vee\text{-I}) \\
Q(n) (=E)
\end{array}$$

Theorem 2 and 3 is simple theorems of number theory:

Theorem 2 :

(0 can be divided by any natural number)

$$\begin{array}{c}
\frac{[0 : \text{nat}] (*) \quad \frac{[p : \text{nat}]}{0 = 0 \cdot p} (*)}{\exists d' : \text{nat}. 0 = d' \cdot p} (\vee\text{-I}) \\
\forall p : \text{nat}. p \mid 0
\end{array}$$

Theorem 3 :

(any natural number can divides itself)

$$\begin{array}{c}
\frac{[1 : \text{nat}] (*) \quad \frac{[q : \text{nat}]}{q = 1 \cdot q} (*)}{\exists d'' : \text{nat}. q = d'' \cdot q} (\vee\text{-I}) \\
\forall q : \text{nat}. q \mid q
\end{array}$$

$\Pi_2 : (\forall l : \text{nat}. (l < n \supset Q(l)), n > 0, m : \text{nat}, n : \text{nat} \vdash Q(n))$

$$\frac{\frac{[n : \text{nat}] \quad \Pi_{21}}{\exists d : \text{nat}. d \mid (m \bmod n) \wedge d \mid n} (\forall\text{-E}) \quad \frac{[d : \text{nat}] \quad \Pi_{22}}{\exists d : \text{nat}. d \mid n \wedge d \mid m} (\exists\text{-I})}{\exists d : \text{nat}. d \mid n \wedge d \mid m} (\exists\text{-E}) \quad \frac{}{Q(n)} (\forall\text{-I})$$

$\Pi_{21} : (\forall l. (l < n \supset Q(l)), n > 0, m : \text{nat}, n : \text{nat} \vdash Q(m \bmod n))$

$$\frac{\frac{[n > 0] \quad [n : \text{nat}] \quad [m : \text{nat}]}{(m \bmod n) < n} (*) \quad \frac{\frac{[n > 0], \quad [n : \text{nat}] \quad [m : \text{nat}]}{(m \bmod n) : \text{nat}} (*) \quad [\forall l : \text{nat}. (l < n \supset Q(l))]}{(m \bmod n) < n \supset Q(m \bmod n)} (*)}{Q(m \bmod n)}$$

$\Pi_{22} : (d : \text{nat}, d \mid (m \bmod n) \wedge d \mid n, n > 0, m : \text{nat}, n : \text{nat} \vdash d \mid n \wedge d \mid m)$

$$\frac{\frac{[d \mid (m \bmod n) \wedge d \mid n]}{d \mid n}}{d \mid n \wedge d \mid m} \Pi_{221} (\wedge\text{-I})$$

$\Pi_{221} : (d : \text{nat}, d \mid (m \bmod n) \wedge d \mid n, n > 0, m : \text{nat}, n : \text{nat} \vdash d \mid m)$

$$\frac{\frac{[m : \text{nat}] \quad [n : \text{nat}] \quad [n > 0]}{(m - (m \bmod n))/n : \text{nat}} \quad \frac{[m : \text{nat}] \quad [n : \text{nat}] \quad [n > 0]}{m = n \cdot ((m - (m \bmod n))/n) + (m \bmod n)}}{\frac{\exists d_0 : \text{nat}. m = n \cdot d_0 + (m \bmod n)}{d \mid m}} \Pi_{2211} (\exists\text{-E})$$

$\Pi_{2211} : (\Gamma_1, d_0 : \text{nat}, m = n \cdot d_0 + (m \bmod n) \vdash d \mid m, \text{ where } \Gamma_1 \text{ is the hypothesis of } \Pi_{221}).$

$$\frac{\frac{[m = n \cdot d_0 + (m \bmod n)] \quad \frac{[d \mid (m \bmod n) \wedge d \mid n]}{d \mid (m \bmod n)} \quad \frac{\Pi_{22111} \quad \Pi_{22112}}{d \mid n \cdot d_0 + (m \bmod n)} (\exists\text{-E})}{d \mid n \cdot d_0 + (m \bmod n)} (\exists\text{-E})}{d \mid m} (= 1)$$

where $d \mid (m \bmod n) \stackrel{\text{def}}{=} \exists d_1 : \text{nat}. (m \bmod n) = d_1 \cdot d$

$\Pi_{22111} : (\Gamma_2, d_1 : \text{nat}, (m \bmod n) = d_1 \cdot d \vdash d \mid n \cdot d_0, \text{ where } \Gamma_2 \text{ is the hypothesis of } \Pi_{2211})$

$$\frac{\frac{[d \mid (m \bmod n) \wedge d \mid n]}{d \mid n} \quad \frac{[d_0 : \text{nat}] \quad [d_3 : \text{nat}]}{d_3 \cdot d_0 : \text{nat}} \quad \frac{[n = d_3 \cdot d] \quad [d_0 : \text{nat}] \quad [d_1 : \text{nat}]}{n \cdot d_0 = d_3 \cdot d_0 \cdot d}}{d \mid n \cdot d_0} (\exists\text{-E})$$

where $d \mid n \cdot d_0 \stackrel{\text{def}}{=} \exists d_2 : \text{nat}. n \cdot d_0 = d_2 \cdot d \quad d \mid n \stackrel{\text{def}}{=} \exists d_3 : \text{nat}. n = d_3 \cdot d$

$\Pi_{22112} : (\Gamma_3, d_2 : \text{nat}, n = d_3 \cdot d \vdash d \mid n \cdot d_0 + (m \bmod n), \text{ where } \Gamma_3 \text{ is the hypothesis of } \Pi_{22111})$

$$\frac{\frac{[d_1 : \text{nat}] \quad [d_2 : \text{nat}]}{d_1 + d_2} \quad \frac{[n \cdot d_0 = d_2 \cdot d] \quad [(m \bmod n) = d_1 \cdot d]}{n \cdot d_0 + (m \bmod n) = (d_1 + d_2) \cdot d}}{d \mid n \cdot d_0 + (m \bmod n)} (\exists\text{-I})$$