ICOT Technical Report: TR-515

TR-515

Higher Order Programming in QPC^2-A Case Study of Map-Function

by
Y. Takayama

November, 1989

ICOT

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

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Submitted for publication in TCS

HIGHER ORDER PROGRAMMING IN QPC^2 – A CASE STUDY OF MAP-FUNCTION

YUKIHIDE TAKAYAMA

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

ABSTRACT

This paper describes the technique of defining programs in second order intuitionistic logic and generating efficient programs. **QPC^2**, a second order constructive logic, is presented and a variant of realizability, proof normalization, extended projection method, and various other techniques are used to generate programs from the definitions. The examples of *map*, *ap*(*map*, *succ*), and *ap*(*map*, *even_odd*), where *even_odd* returns boolean values according to input natural numbers, are investigated, and quite natural and efficient programs are extracted.

Keywords: Constructive Logic; Program Extraction; Extended Projection; Realizability.

1. Introduction

There are two main approaches to constructive logic: one is type theoretic formulation of constructive logic in which types and propositions are partly or totally identified following the principle of Curry-Howard isomorphism [1], the other follows the traditional formulation of intuitionistic logic [2, 3, 4] in which types and propositions are strictly separated and realizability interpretation of logical constants [5, 6] is the comparative notion to Curry-Howard isomorphism.

From the viewpoint of programming, programming methodology in constructive logic is an interesting research theme: how specifications are written and how to generate efficient programs from the theorems and proofs in various formulations of constructive logic. N. G. de Bruijn claimed that identification of propositions and types is a rather natural style of reasoning in mathematics and it can also be used in reasoning about programs [7,8] J. Smith, B. Nordström, K. Peterson and P. Chisholm developed programming techniques in Martin-Löf's theory of types [9] such as sorting, Ackerman function and simple parser algorithm [10, 11, 12]. C. Mohring showed how to develop programs in the calculus of constructions [13, 14]. J. Bates and R. Constable also gave a lot of examples in constructive type theory and program extractor system which generates efficient programs [15, 16,

17]. C. Mohring also defined the calculus of constructions with a new constant, *Spec*, and showed a technique to generate redundancy-free second order lambda terms from constructive proofs [18]. A similar technique was also introduced by B. Nordström and K. Peterson [11]. For the traditional formulation of intuitionistic logic, C. Goad defined a formulation of intuitionistic logic and a lambda calculus. He gave a technique called *pruning* to detect and eliminate redundant case branches and shows an example of an efficient bin packing algorithm [19]. S. Hayashi introduced a powerful schema of inductive definition called CIG, in **PX** [4], a Feferman-style theory of functions and classes, and showed various techniques to define and generate efficient programs using CIG. The author defined a formulation of constructive logic, QPC, and a type-free λ-calculus and a few simple examples were presented with several techniques to extract efficient programs [20]. The problem of redundancy in the extracted code was also specified in the formulation of QPC. The author also developed a technique to analyze and remove the redundancy called extended projection [21].

This paper works on the technique of higher order programming in $\mathbf{QPC}^2$, a second order version of QPC, with extended projection. Goad's system is a first order theory, and PX does not use second order universal quantification although that can be simulated by class variables. Introducing predicate variables and universal quantification over them ($\forall^2$) allows a kind of parametric programming and improves the expressive power of the formal system. Second order theory is quite common in type theory and typed lambda calculus. The chief difference from the type theoretic approach is that the second order feature itself does not induce polymorphism because propositions are not identified with types. However, polymorphism can be introduced by making the type theory in $\mathbf{QPC}^2$ second order.

One of the critical issues in second order constructive logic is how to define realizability of $\forall^2 P.A(P)$. One approach is Kreisel-Troelstra's realizability [22], which does not give any constructive interpretation to second order universal quantification. Similar realizability was introduced by J. Krivine [23]. Second order typed lambda calculus [24] can be seen as giving a formulation of realizability in which a second order lambda term is given as realizer of a second order universal quantification. $\mathbf{QPC}^2$ takes a slightly different approach. Realizability is defined only on first order formulas, and a program schema which resembles second order lambda term is extracted from a proof of $\forall^2 P.A(P)$ type formula. The schema is instantiated to a variant of type-free lambda term by proof normalization when $\forall^2$ is eliminated in the proof by the elimination rule of second order universal quantifiers.

Chapter 2 outlines QPC, extended projection method and a few extended features. Chapter 3 introduces $\mathbf{QPC}^2$. Chapter 4 gives the general strategy to define and extract *map*-function and list processing functions made of *map*. The program extraction algorithm for second order proofs is given. Also, ML-polymorphism is introduced to obtain an extended system, $\mathbf{QPC}^2_2$. Chapter 5 investigates the example of $map : (nat \to \sigma) \to L(nat) \to L(\sigma)$, where $\sigma = nat$ or $bool$, and $ap(map, f)$ where $f : nat \to \sigma$ to show how the native extraction based on realizability, proof normalization, and extended projection method are applied. Chapter 6 points out inefficiency of the program obtained in chapter 5, and shows a new technique to improve it. Chapter 7 gives the conclusion.

2. Outline of QPC

QPC is a first order intuitionistic Gentzen style of natural deduction with a variant of lambda calculus as its terms. A sort of q-realizability [6, 25] is defined, and it is used for

the naive version of program extractor from QPC proofs. Three notions, modified V-code, logical terms and *let*-construct are also introduced to improve the program extractor. Also, extended projection method is used to detect and eliminate redundancy in the program extracted by the naive version of extractor. Most part of the contents in this chapter are presented in [20, 21], but the notion of logical terms, *let*-construct and structural induction on lists are newly introduced.

2.1 Core part of QPC

The core part of QPC is a modified subset of QJ [3] and Quty [26]. The point of the modification is that Quty, which is the target language of the programming logic QJ, is restricted to a set which is related to program extraction, and, while Quty is a strongly typed language, our language is a variant of type-free lambda calculus.

2.1.1 Terms and Formulas

The terms of QPC are a variant of type-free λ-expressions.

Definition 1: Terms:
1) Individual variables, x, y, $\cdots$, are terms;
2) Natural numbers, $0, 1, \cdots$, and booleans, t and f, are terms;
3) Special constants $left$, $right$ and nil are terms;
4) If $t_0, \cdots . t_{n-1}$ $(n > 1)$ are terms, then the sequence of terms, $(t_0, \cdots, t_{n-1})$, is also a term. $()$ denotes the nil sequence;
5) $any[n]$, which denotes a sequence of any n terms, is a term;
6) If $\overline{x}$ is a sequence of individual variables and t is a term, then $\lambda\overline{x}.t$ is a term (λ-term);
7) If s and t are terms, then $ap(s, t)$ is a term (application);
8) If $\overline{z}$ is a sequence of variables and t is a term, then $\mu\overline{z}.t$ is a term (μ-term);
9) If L is an equation or inequation of terms and s and t are terms, then
$if\ bval(L)\ then\ s\ else\ t$ is a term;
10) Primitive functions. hd (car), tl (cdr), :: (cons), $proj$, $proj\_h$, $proj\_t$, $tseq$ and $ttseq$ are terms.

$bval$ is a function which returns t if a given formula is true, and returns f otherwise. $bval(A)$ is abbreviated A in the following description. A μ-term means a multi-valued recursive call function which is a code extracted from a proof in induction. nil and :: are list constructors.

The following is equivalence of terms:

$$\lambda\overline{x}.(t_1, \cdots, t_k) \equiv (\lambda\overline{x}.t_1, \cdots, \lambda\overline{x}.t_k)$$

$$ap((a_1, \cdots, a_k), b) \equiv (ap(a_1, b), \cdots, ap(a_k, b))$$

$$if\ A\ then\ (b_1, \cdots, b_k)\ else\ (c_1, \cdots, c_k) \equiv (if\ A\ then\ b_1\ else\ c_1, \cdots, if\ A\ then\ b_k\ else\ c_k)$$

The type structure of QPC is rather simple. Some of the terms such as μ-term cannot be typed, but the termination property of typing holds, i.e., if $t : \sigma$ for some type σ, then the computation of t terminates.

Definition 2: Types:
1) *nat* and *bool* are primitive types;
2) If σ and τ are types, then $\sigma \times \tau$ is a type;
3) If σ and τ are types, then $\sigma \to \tau$ is a type;
4) If σ is a type, then $L(\sigma)$ is also a type (σ-list type).

The typing rules are as follows:

$$n : nat \quad (n = 0, 1, \cdots) \qquad t : bool \qquad f : bool$$

$$\frac{\bar{x} : \sigma \quad t : \tau}{\lambda \bar{x}.t : \sigma \to \tau} \qquad \frac{x : \sigma \quad y : \tau}{(x, y) : \sigma \times \tau}$$

$$nil : L(\sigma) \quad (\sigma \text{ is any type}) \qquad \frac{a : \sigma \quad b : L(\sigma)}{a :: b : L(\sigma)}$$

Definition 3: Atomic formula:
1) $\bot$ is atomic;
2) For a term, t, and a type, σ, $t : \sigma$ is atomic;
3) For terms s and t, $s = t$, $s \leq t$ and $s < t$ are atomic.

$\bot$ means abort, $t : \sigma$ yields type checking, equalities are solved if both terms have the save normal forms, and inequalities are solved if both terms are reduced to natural numbers.

Definition 4: Formula:
1) An atomic formula is a formula;
2) If A and B are formulas, then $A \wedge B$, $A \vee B$ and $A \supset B$ are formulas;
3) If A is a formula and σ is a type, then $\forall x \in \sigma.A$ and $\exists x \in \sigma.A$ are formulas.

$A[a/x, b/y]$ denotes simultaneous substitution of a and b into any free occurrences of x and y in the expression, A. Also, $A(x)$ denotes a formula A which contains x free. $A(x)[a/x]$ is denoted $A(a)$.

A Harrop formula is a formula which has no constructive information.

Definition 5: Harrop formula:
1) If A is atomic, then A is Harrop;
2) If A and B are Harrop, then $A \wedge B$ is Harrop;
3) If A is Harrop, then $\forall x \in \sigma.A$ is Harrop;
4) If A is a formula, and B is Harrop, then $A \supset B$ is Harrop.

2.1.2 Rules of inference

The rules of inference are I-rules and E-rules on $\vee$, $\wedge$, $\supset$, $\exists$, and $\forall$ of natural deduction, equality rules including reduction rules of λ-term, *if-then-else*, and μ-terms, and induction

rules on natural numbers and lists:

$$\frac{A(0) \qquad \overset{[x-1:nat,\,A(x-1)]}{A(x)}}{\forall x \in nat.\ A(x)} \qquad\qquad \frac{A(nil) \qquad \overset{[hd(x):\sigma,\,tl(x):L(\sigma),\,A(tl(x))]}{A(x)}}{\forall x \in L(\sigma).\ A(x)}$$

2.1.3 Realizability

Realizability is defined in QPC$^+$, a minor extension of QPC, and is a base for the naive version of program extractor called *Ext*.

Definition 6: QPC$^+$
QPC$^+$ is obtained from QPC by adding the following clause to definition 4:
4) If e is a term and A is a formula, then e **qpc** A is a formula.

Definition 7: **qpc**-realizability

1) If A is Harrop, then $()$ **qpc** $A \stackrel{\text{def}}{=} A$;

2) a **qpc** $A \supset B \stackrel{\text{def}}{=} \forall b \subset \sigma.(A \wedge b$ **qpc** $A \supset ap(a,b)$ **qpc** $B)$
where σ is such that $a : \sigma \to \tau$ for some τ

3) (a,b) **qpc** $\exists x \in \sigma.A \stackrel{\text{def}}{=} a : \sigma \wedge A[a/x] \wedge b$ **qpc** $A[a/x]$

4) a **qpc** $\forall x \in \sigma.A \stackrel{\text{def}}{=} \forall x \in \sigma.(ap(a,x)$ **qpc** $A)$

5) (a,b,c) **qpc** $A \vee B \stackrel{\text{def}}{=} (a = left \supset A \wedge b$ **qpc** $A) \wedge (a = right \supset B \wedge c$ **qpc** $B)$

6) (a,b) **qpc** $A \wedge B \stackrel{\text{def}}{=} a$ **qpc** $A \wedge b$ **qpc** B

The chief differences from the standard **q**-realizability are seen in 1) and 5). For 1), the realizer of atomic formula is any term in the standard **q**-realizability, but **qpc**-realizability gives $()$ to reduce the redundancy in the extracted code. **qpc**-realizability is similar to **px**-realizability [4] in this respect. For 5), the realizer of $A \vee B$ type formula is the concatenation of three kinds of terms: $left$ or $right$ showing which formula really holds in the disjunction of A and B, the realizer of A and realizer of B. Therefore, for example, if A holds, the realizer of $A \vee B$ is $(left, \overline{t}, \overline{s})$ where $\overline{t}$ is the realizer of A and $\overline{s}$ is a dummy code. On the other hand, the standard **q**-realizability gives the concatenation of two kinds of information: $left$ or $right$ and the realizer of A or B. This idea leads to the notion of realizing variable sequences and lengths of formulas which are the base of extended projection method explained in 2.1.5.

Theorem 1: Soundness of **qpc** realizability
If A is provable in QPC, then there exists a term called realizer, e, such that e **qpc** *A is provable in QPC$^+$. Also, $FV(e) \subseteq FV(A)$.*

Ext is the procedure which extracts the realizer of the theorem from its proof, and the form of the realizer is given in the proof of the theorem.

Definition 8: Realizing variables: $Rv(A)$

1) $Rv(A) \stackrel{\text{def}}{=} ()$ $\cdots$ if A is Harrop

2) $Rv(A \wedge B) \stackrel{\text{def}}{=} (Rv(A), Rv(B))$

3) $Rv(A \vee B) \stackrel{\text{def}}{=} (z, Rv(A), Rv(B))$

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

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

6) $Rv(\exists x \in \sigma.\ A) \stackrel{\text{def}}{=} (z, Rv(A))$

z is a fresh variable, and $(z, Rv(A), Rv(B))$, for example, means the concatenation of a variable, z, $Rv(A)$ and $Rv(B)$. Therefore, $Rv(A)$ is a sequence of fresh variables introduced in 3) and 6).

Realizing variables are used in Ext as the code extracted from discharged formulas. For example, Ext for ($\supset$-I) application is defined as follows:

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

Definition 9: Length of a formula
The length of a formula A, $l(A)$, is the length of $Rv(A)$ as a sequence of variables.

2.1.4 μ-term

μ-term is the realization of induction proofs: $\mu\ (z_0, \cdots, z_{n-1}).\ F(z_0, \cdots, z_{n-1})$ $(n \geq 1)$ which is also called n-dimensional multi-valued recursive call functions. The intentional meaning of μ-term is the minimum solution of the following fixed point equation in some suitable domain.

$$(z_0, \cdots, z_{n-1}) = F(z_0, \cdots, z_{n-1})$$

If $F(z_0, \cdots, z_{n-1})$ is expanded into sequences of terms, $(F_0(\vec{z}), \cdots, F_{n-1}(\vec{z}))$, the equation can be solved: $(f_0, \cdots, f_{n-1}) = F(f_0, \cdots, f_{n-1})$, i.e.,

$$\mu\ (z_0, \cdots, z_{n-1}).\ F(z_0, \cdots, z_{n-1}) \equiv (f_0, \cdots, f_{n-1})$$

where

$$f_i = \mu z_i.\ F_i(f_0, \cdots, f_{i-1}, z_i, f_{i+1}, \cdots, f_{n-1}) \qquad (0 \leq i \leq n-1)$$

The program extraction procedure for structural induction on lists is as follows:

$$Ext\left(\begin{array}{c}\frac{\Sigma_0}{A(nil)} \qquad \frac{[hd(x) : \sigma, tl(x) : L(\sigma), A(tl(x))]}{\frac{\Sigma_1}{A(x)}}\\ \hline \forall x \in L(\sigma).\ A(x)\ (L(\sigma)\text{-}ind)\end{array}\right)$$

$$\stackrel{\text{def}}{=} \mu\ \vec{z}.\ \lambda\ x.\ if\ x = nil\ then\ Ext\left(\frac{\Sigma_0}{A(0)}\right)$$

$$else\ Ext\left(\frac{\Sigma_1}{A(x)}\right)[ap(\vec{z}, tl(x))/Rv(A(tl(x)))]$$

where $\overline{z}$ is a new sequence of variables of length $l(A(tl(x)))(= l(A(nil)) = l(A(x)))$

Ext for mathematical induction is defined similarly.

2.1.5 Extended projection

(1) Ordinary projection
$proj(i)$ is the function which takes ith projection of the given sequence of terms. For a finite set, $I \stackrel{\text{def}}{=} \{i_1, \cdots, i_k\} \subset \{0, 1, \cdots, n-1\}$, of natural numbers, $proj(I)$ is the function which takes a subsequence, $(t_{i_1}, \cdots, t_{i_k})$, of a given sequence of terms, $(t_0, \cdots, t_{n-1})$. Also, $proj\_h(i)$ (or $proj\_t(i)$) takes first (or last) i elements of the given sequence of terms. $tseq(i)$ takes ith to the last elements of the given sequence, and $ttseq(i, j)$ takes ith to $i+j-1$th elements of the given sequence.

Theorem 2:
The term extracted by Ext procedure from a QPC proof can be expanded to a sequence of terms.

Note that in the definition of Ext for ($L(\sigma)$-*ind*) in 2.1.4, $Ext(\Sigma_0/A(0))$ and $Ext(\Sigma_1/A(x))$ are expanded into sequences of terms of length $l(A(0))(= l(A(x)))$. Therefore, the μ-term in the definition of Ext can be expanded into a sequence of terms of the same length as explained in 2.1.4.

As can be seen from the definition of **qpc**-realizability, each element of the sequence corresponds to an $\exists$ or $\vee$ occurrence in the theorem. Also, the $\exists$ and $\vee$ occurrences in the theorem which correspond to part of the realizer can be pointed with the positions of variables in the realizing variables of the theorem.

Therefore, projection functions can be applied to the extracted code to obtain the part of the realizer which is really needed as the program. However, it does not always work well for induction proofs. Assume the following μ-term:

$$T \stackrel{\text{def}}{=} \mu(z_0, z_1).\lambda x.\mathit{if}\ x = 0\ \mathit{then}\ (0,0)\ \mathit{else}\ (ap(z_1, x-1)+1, ap(z_0, x-1)+2)$$

It can be expanded to $\mu z_0.\lambda x.\mathit{if}\ x = 0\ \mathit{then}\ 0\ \mathit{else}\ ap(z_1, x-1)+1$ and $\mu z_1.\lambda x.\mathit{if}\ x = 0\ \mathit{then}\ 0\ \mathit{else}\ ap(z_0, x-1)+2$. Then, the code, $proj(0)(T)$ alone is not a reasonable program because it calls $proj(1)(T)$ from inside.

(2) Extended projection
Extended projection is the projection applied on proofs instead of extracted codes. As explained above, any part of the extracted code can be specified with a set of positions in the realizing variables. Let I be the set of positions in $Rv(A)$, and let Π be a proof of A. Then, the procedure $Mark$ attaches the finite set on each occurrences of formulas in Π. For example, let $A \stackrel{\text{def}}{=} (P \wedge Q) \vee R$, $l(P) = 2$, $l(Q) = 1$, $l(R) = 3$ and $I = \{0, 2, 5\}$. 0 points the $\vee$ occurrence in A, and 2 and 5 point $\exists$ or $\vee$ occurrences in P and R.

$$Mark\left(\{0,2,5\}, \frac{\dfrac{\dfrac{\Sigma_0}{P} \quad \dfrac{\Sigma_1}{Q}}{P \wedge Q}(\wedge\text{-}I)}{(P \wedge Q) \vee R}(\vee\text{-}I)\right) - \frac{Mark\left(\{1\}, \dfrac{\dfrac{\Sigma_0}{P} \quad \dfrac{\Sigma_1}{Q}}{P \wedge Q}\right)}{\{(P \wedge Q) \vee R\}_{\{0,2,5\}}}$$

$$= \frac{Mark\left(\{1\}, \dfrac{\Sigma_0}{P}\right) \quad Mark\left(\varphi, \dfrac{\Sigma_1}{Q}\right)}{\dfrac{\{P \wedge Q\}_{\{1\}}}{\{(P \wedge Q) \vee R\}_{\{0,2,5\}}}}$$

$\{0,2,5\}$ and $\{1\}$ attached to the nodes show which parts of the code extracted from the subtrees are needed to extract the code specified by I. I is called the *declaration* to A.

The program extractor is modified to handle the tree, $Mark(I, \Pi)$, and it is called $NExt$ procedure.

The advantage of extended projection is that $Mark$ procedure can detect the situation explained in the example in (1). T is extracted from a proof in induction of a formula $\forall x.A(x)$ of length 2. If one tries to take the 0th projection, take $\{0\}$ as the declaration. Then, $Mark$ attaches $\{1\}$ to the occurrences of induction hypothesis, $A(x-1)$, which means the 0th element of T recursively calls the 1th element. Therefore, it turns out that both 0th and 1st elements must be extracted to generate reasonable program. The number 1 in $\{1\}$ is called *overflowed marking number*, and this can be eliminated by enlarging the declaration. If the declaration is enlarged to $\{0,1\}$, $Mark$ attaches $\{1\}$ to the induction hypothesis. However, 1 is not an overflowed marking number because it is contained in the new declaration.

2.1.6 Other notions and terminologies

A proof tree is often denoted as (Σ/B) instead of

$$\frac{\Sigma}{B}(Rule)$$

The proof theoretic terminologies used in this paper follow Prawitz [27].

Normalization procedure is the set of following reduction rules of proof trees:

$$\frac{\dfrac{\Sigma_0}{t \in \sigma} \quad \dfrac{\begin{array}{c}[x:\sigma] \\ \Sigma_1 \\ P(x)\end{array}}{\forall x \in \sigma.P(x)}(\forall\text{-I})}{P(t)}(\forall\text{-E}) \quad \forall\text{-}reduction \quad \Longrightarrow \quad \begin{array}{c}\Sigma_0 \\ \hline [t \in \sigma] \\ \Sigma_1[t/x] \\ P(t)\end{array}$$

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

$$\dfrac{\dfrac{\Sigma_0}{A}(\vee\text{-}I)}{\dfrac{A\vee B}{C}}\quad\dfrac{\overset{[A]}{\Sigma_1}}{C}\quad\dfrac{\overset{[B]}{\Sigma_2}}{C}(\exists\text{-}E) \quad \vee\textit{-reduction}_1 \quad \Longrightarrow \quad \dfrac{\dfrac{\Sigma_0}{[A]}}{\dfrac{\Sigma_1}{C}}$$

$$\dfrac{\dfrac{\Sigma_0}{B}(\vee\text{-}I)}{\dfrac{A\vee B}{C}}\quad\dfrac{\overset{[A]}{\Sigma_1}}{C}\quad\dfrac{\overset{[B]}{\Sigma_2}}{C}(\exists\text{-}E) \quad \vee\textit{-reduction}_2 \quad \Longrightarrow \quad \dfrac{\dfrac{\Sigma_0}{[B]}}{\dfrac{\Sigma_2}{C}}$$

$$\dfrac{\dfrac{\dfrac{\Sigma_0}{A}\quad\dfrac{\Sigma_1}{B}(\wedge\text{-}I)}{A\wedge B}(\wedge\text{-}E)}{A} \quad \wedge\textit{-reduction}_1 \quad \Longrightarrow \quad \dfrac{\Sigma_0}{A}$$

$$\dfrac{\dfrac{\dfrac{\Sigma_0}{A}\quad\dfrac{\Sigma_1}{B}(\wedge\text{-}I)}{A\wedge B}(\wedge\text{-}E)}{B} \quad \wedge\textit{-reduction}_2 \quad \Longrightarrow \quad \dfrac{\Sigma_1}{B}$$

$$\dfrac{\dfrac{\Sigma_0}{t:\sigma}\quad\dfrac{\Sigma_1}{A[t/x]}}{\exists x\in\sigma.\,A}\quad\dfrac{\overset{[x:\sigma]\ [A]}{\Sigma_2}}{C}(\exists\text{-}E) \quad \exists\textit{-reduction} \quad \Longrightarrow \quad \dfrac{\dfrac{\Sigma_0}{[t:\sigma]}\quad\dfrac{\Sigma_1}{A[t/x]}}{\dfrac{\Sigma_2[t/x]}{C}}$$

As is well known, normalization procedure corresponds to partial evaluation of extracted codes. $\forall$-reduction and $\supset$-reduction correspond to β-reduction of extracted codes. $\vee$-reduction simplifies *if-then-else* sentences. $\wedge$-reduction and $\exists$-reduction is implicitly embedded in Ext as will be explained in 2.2.2.

2.2 Some modified features

2.2.1 Modified $\vee$-code and logical terms

The definition of Ext for $(\vee\text{-}E)$ application is as follows:

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

$$\stackrel{\text{def}}{=} \mathit{if}\ \mathit{beval}(\mathit{proj}(0)(\mathit{Ext}(\Sigma_0/A \vee B)) = \mathit{left})$$
$$\mathit{then}\ \mathit{Ext}\left(\frac{\Sigma_1}{C}\right)[\mathit{ttseq}(1, l(A))(\mathit{Ext}(\Sigma_0/A \vee B))/\mathit{Rv}(A)]$$
$$\mathit{else}\ \mathit{Ext}\left(\frac{\Sigma_2}{C}\right)[\mathit{tseq}(l(A)+1)(\mathit{Ext}(\Sigma_0/A \vee B))]$$

The decision procedure extracted by Ext from ($\vee$-E) application can be simplified if $A \vee B$ itself is computable. Logical term is a class of formulas which are computable.

Definition 10: Logical terms
If $a_1 : \sigma_1, \cdots, a_n : \sigma_n$ and $b_1 : \sigma_1, \cdots, b_n : \sigma_n$ $(n \geq 1)$, and assume that R, which is $=$, $<$, $>$, $\leq$ or $\geq$, is well-defined on σ_i $(1 \leq i \leq n)$, then $a_1\ R\ b_1 \wedge \cdots \wedge a_n\ R\ b_n$ is a logical term.

Note that logical terms are Harrop formulas. Then, the modified Ext procedure is as follows:

$$\mathit{Ext}\left(\begin{array}{c} \\ \dfrac{\dfrac{\Sigma_0}{A \vee B}\quad \dfrac{\overset{[A]}{\Sigma_1}}{C}\quad \dfrac{\overset{[B]}{\Sigma_2}}{C}}{C}(\vee\text{-}F) \\ \\ \end{array}\right)$$

$$\stackrel{\text{def}}{=} \mathit{if}\ \mathit{beval}(A)$$
$$\mathit{then}\ \mathit{Ext}\left(\frac{\Sigma_1}{C}\right)$$
$$\mathit{else}\ \mathit{Ext}\left(\frac{\Sigma_2}{C}\right)[\mathit{tseq}(1)(\mathit{Ext}(\Sigma_0/A \vee B))/\mathit{Rv}(B)]$$

$\cdots$ if A is a logical term

$$\stackrel{\text{def}}{=} \mathit{if}\ \mathit{beval}(B)$$
$$\mathit{then}\ \mathit{Ext}\left(\frac{\Sigma_1}{C}\right)[\mathit{tseq}(1)(\mathit{Ext}(\Sigma_0/A \vee B))/\mathit{Rv}(A)]$$
$$\mathit{else}\ \mathit{Ext}\left(\frac{\Sigma_2}{C}\right)$$

$\cdots$ if B is a logical term

$$\stackrel{\text{def}}{=} \mathit{if}\ \mathit{beval}(\mathit{proj}(0)(\mathit{Ext}(\Sigma_0/A \vee B)) = \mathit{left})$$
$$\mathit{then}\ \mathit{Ext}\left(\frac{\Sigma_1}{C}\right)[\mathit{ttseq}(1, l(A))(\mathit{Ext}(\Sigma_0/A \vee B))/\mathit{Rv}(A)]$$
$$\mathit{else}\ \mathit{Ext}\left(\frac{\Sigma_2}{C}\right)[\mathit{tseq}(l(A)+1)(\mathit{Ext}(\Sigma_0/A \vee B))/\mathit{Rv}(B)]$$

$\cdots$ otherwise

2.2.2 *let*-construct

(1) Substitution in Ext procedure

In the program extraction from ($\vee$-E) application, the code extracted from the proof of $A \vee B$ is substituted to $Rv(A)$ and $Rv(B)$ in the code extracted from the proofs of C from

A and C from B. This sort of substitution is also performed in the handling of proofs in $(\exists\text{-}E)$ rule:

$$\dfrac{\dfrac{\Sigma_0}{\exists x \in \sigma.A} \quad \dfrac{[x:\sigma][A]}{\dfrac{\Sigma_1}{C}}}{C}(\exists\text{-}E)$$

Let $(t_0, t_1, \cdots, t_k)$ be the code extracted from $(\Sigma_0/\exists x \in \sigma.A)$, and let T be the code extracted from (Σ_1/C). The code extracted from the whole proof is $T[t_0/x, (t_1, \cdots, t_k)/Rv(A)]$. The substitution in the handling of $(\exists\text{-}E)$ application implicitly performs $\exists$-reduction if $\exists x \in \sigma.A$ is proved by $(\exists\text{-}I)$.
These substitution procedure in Ext are slightly different from the other extractors such as PX (Ref. 4) in which programs with let-construct or similar program constructs are generated instead of performing substitution in the program extraction. The substitution makes the extracted program a more efficient.

(2) Limitation of the substitution in Ext

If the major premises or formula occurrences above the major premises of $(\vee\text{-}E)$ applications or $(\exists\text{-}E)$ applications are proved in induction, the substitution of Ext is a little bit complicated. Assume, for example, a $(\exists\text{-}E)$ application is as follows:

$$\dfrac{\dfrac{\dfrac{\Sigma_0}{t:\sigma} \quad \dfrac{\dfrac{\Sigma_1}{\forall x \in \sigma.\exists y \in \tau.A}(ind)}{\exists y \in \sigma.A[t/x]}(\forall\text{-}E) \quad \dfrac{[x:\sigma][A[t/x]]}{\dfrac{\Sigma_2}{C}}}{C}(\exists\text{-}E)$$

The code extracted from the proof of the major premise is $ap(\mu(z_0, \cdots, z_{n-1}).F(z_0, \cdots, z_{n-1}), t)$. This code must be expanded into a sequence of terms to perform the substitution. As explained after theorem 2, there is a sequence of single-valued recursive call functions, $f_0, \cdots, f_{n-1}$, such that

$$\mu(z_0, \cdots, z_{n-1}).F(z_0, \cdots, z_{n-1}) \equiv (f_0, \cdots, f_{n-1})$$

Therefore,

$$ap(\mu(z_0, \cdots, z_{n-1}).F(z_0, \cdots, z_{n-1}), t) \equiv (ap(f_0, t), \cdots, ap(f_{n-1}, t))$$

However, this complicates handling of multi-valued recursive call functions, in particular implementation of the interpreter, and makes the extracted programs difficult to understand. Also, from the viewpoint of extraction of efficient programs, it is preferable to handle μ-terms without expanding them into sequences. This is explained in chapter 6.

(3) Introduction of let-construct

let-construct is introduced to handle μ-terms without expansion. For example, the code extracted from the last example of $(\exists\text{-}E)$ application is

$$let\ (y, Rv(A[t/x])) = ap(\mu(z_0, \cdots, z_{n-1}).F(z_0, \cdots, z_{n-1}), t)\ in\ Ext(\Sigma_1/C)$$

instead of

$$Ext(\Sigma_2/C)[ap(f_0, t)/y, (ap(f_1, t), \cdots, ap(f_{n-1}, t))/Rv(A[t/x])]$$

3. $\mathbf{QPC}^2$

$\mathbf{QPC}^2$ is a predicative extension of QPC which is obtained by introducing a new type constant, *prop*, predicate variables and introduction and elimination rules of second order universal quantifier.

3.1 Language of $\mathbf{QPC}^2$

The language of $\mathbf{QPC}^2$ is obtained as follows:

3.1.1 Types

New constants, $type_1$, $type_2$ and *prop*, are introduced.

Definition 11: Types in $\mathbf{QPC}^2$
1) If σ is a type defined in definition 2, then $\sigma : type_1$;
2) If $\sigma_1 : type_1, \cdots, \sigma_n : type_1$ $(n \geq 0)$, then $\sigma_1 \times \cdots \times \sigma_n \rightarrow prop : type_2$;

The types in $\mathbf{QPC}^2$ are those of type $type_1$ or $type_2$.

3.1.2 Formulas

Definition 12: Class 1 formulas:
1) The formulas constructed by the clause 1) and 2) in definition 4 are class 1 formulas;
2) If $\sigma : type_1$ and A is a class 1 formula, then $\forall x \in \sigma.A$ and $\exists x \in \sigma.A$ are class 1 formulas.
3) $P(a_1, \cdots, a_n)$ with a predicate variable P of type $\sigma_1 \times \cdots \times \sigma_n \rightarrow prop$ $(: type_2)$ and $a_i : \sigma_i$ $(0 \leq i \leq n)$ is a class 1 formula;

Definition 13: Class 2 formulas:
1) Class 1 formulas are class 2 formulas;
2) If A is a class 2 formula and $\sigma_1 \times \cdots \times \sigma_n \rightarrow prop : type_2$ and P is a predicate variable, then $\forall^2 P \in \sigma_1 \times \cdots \times \sigma_n \rightarrow prop.A$ is a class 2 formula.

The formulas of $\mathbf{QPC}^2$ are class 2 formulas.

3.1.3 Terms

The terms of $\mathbf{QPC}^2$ are those of QPC and the non-computable terms defined below:

Definition 14: Non-computables terms:
a) Predicate variables, P, Q, $\cdots$;
b) Abstracts $\lambda(x_1, \cdots, x_n).p$ where p is a class 1 formula and $\{x_1, \cdots, x_n\} \subset FV(p)$;
c) If P is a predicate variable and a_i $(1 \leq i \leq n)$ is a term,
then $RV(P(a_1, \cdots, a_n))$ is a term (Rv-schema);
d) If P is a predicate variable and T is a term, $\Lambda P.\ T$ is a term.
(program schema)

3.2 Rules on Formulas and Abstracts

$$\frac{A \text{ is a class 1 formula}}{A : prop}$$

The rules of inferences in QPC are, for example, written as follows:

$$\frac{A : prop \quad B : prop}{A \wedge B} \frac{A \quad B}{}(\wedge\text{-}I)$$

However, $A : prop$ and $B : prop$ is usually dropped.

$$\frac{\sigma_i : type_1 \ (0 \leq i \leq n) \quad P : \sigma_1 \times \cdots \times \sigma_n \to prop \quad (a_1, \cdots, a_n) : \sigma_1 \times \cdots \times \sigma_n}{P(a_1, \cdots, a_n) : prop}$$

$$\frac{\widetilde{q} : prop \quad \{x_1, \cdots, x_n\} \subset FV(\widetilde{q}) \quad x_i : \sigma_i \quad \sigma_i : type_1 \ (0 \leq i \leq n)}{\lambda(x_1, \cdots, x_n).\widetilde{q} : \sigma_1 \times \cdots \times \sigma_n \to prop}$$

$$\frac{\sigma_i : type_1 \ (0 \leq i \leq n) \quad \lambda\overline{x}.\widetilde{q} : \sigma_1 \times \cdots \times \sigma_n \to prop \quad \overline{a} : \sigma_1 \times \cdots \times \sigma_n}{\widetilde{q}[\overline{a}/\overline{x}] : prop}$$

3.3 Second Order Rules of Inference

$$\frac{\begin{array}{c}[P : \sigma_1 \times \cdots \times \sigma_n \to prop] \\ A(P)\end{array}}{\forall^2 P \in \sigma_1 \times \cdots \times \sigma_n \to prop.A(P)}(\forall^2\text{-}I)$$

$$\frac{\lambda\overline{x}.\widetilde{q} : \sigma_1 \times \cdots \times \sigma_n \to prop \quad \forall^2 P \in \sigma_1 \times \cdots \times \sigma_n \to prop.A(P)}{A(\lambda\overline{x}.\widetilde{q})}(\forall^2\text{-}E)$$

$A(\lambda\overline{x}.\widetilde{q})$ is obtained by translating all the occurrences of P as follows:

$$P(a_1, \cdots, a_n) \Rightarrow ap(\lambda(x_1, \cdots, x_n).\widetilde{q}, (a_1, \cdots, a_n)) \overset{\beta\text{-}red}{\longrightarrow} \widetilde{q}[a_1/x_1, \cdots, a_n/x_n]$$

3.4 Normalization rule

$$\frac{\dfrac{\Sigma_0}{\lambda(x_1, \cdots, x_n).\widetilde{q} : \sigma_1 \times \cdots \times \sigma_n \to prop} \quad \dfrac{\begin{array}{c}[P : \sigma_1 \times \cdots \times \sigma_n \to prop] \\ \dfrac{\Sigma_1(P)}{F(P)}\end{array}}{\forall^2 P \in \sigma_1 \times \cdots \times \sigma_n \to prop.F(P)}}{F(\lambda(x_1, \cdots, x_n).\widetilde{q})}$$

$$\forall^2\text{-}reduction \quad \Longrightarrow \quad \frac{\dfrac{\Sigma_0}{[\lambda(x_1, \cdots, x_n).\widetilde{q} : \sigma_1 \times \cdots \times \sigma_n \to prop]} \quad \Sigma_1(\lambda(x_1, \cdots, x_n).\widetilde{q})}{F(\lambda(x_1, \cdots, x_n).\widetilde{q})}$$

This rule is used in the program extraction from $\mathbf{QPC}^2$ proofs which will be shown with an example in the next chapter.

4. General Scheme of Specifications and Program Extraction

In this section, a method of higher order programming in QPC and $\mathbf{QPC}^2$ is presented through the example of *map*-function, and the naive version of program extractor, Ext, is extended to $\mathbf{QPC}^2$.

4.1 Specification

$\mathbf{QPC}^2$ does not handle polymorphic programming because, although it allows second order universal quantification on predicate variables, types and formulas are strictly separated and the type system in $\mathbf{QPC}^2$ is not polymorphic. However, it can handle a sort of parametric programming. Assume a recursive call function which takes an element, x, of $L(\sigma)$ and apply a given function, f, of type $\sigma \to \tau$ recursively on each element of x. This function, *map*, can be expressed as follows:

$$\lambda f.\mu z.\lambda x.if\ x = nil\ then\ nil\ else\ ap(f, hd(x)) :: ap(z, tl(x))$$

map-function can be extracted from $\mathbf{QPC}^2$ proofs by parameterizing f in the proofs.
The first idea of parameterizing f is to quantify it:

$$\begin{array}{l}\forall f \in \sigma \to \tau.Spec(f)\\ \text{where}\\ Spec(f) \stackrel{\text{def}}{=} \forall x \in L(\sigma).\exists y \in L(\tau).\\ \qquad\qquad length(x) = length(y)\\ \qquad\qquad \wedge(\forall i \in nat.1 \le i \le length(x) \supset f(elem(x,i)) = elem(y,i))\end{array}$$

$$length \stackrel{\text{def}}{=} \mu z.\lambda x.if\ x = nil\ then\ 0\ else\ ap(z, tl(x)) + 1$$
$$elem \stackrel{\text{def}}{=} \mu z.\lambda x.\lambda i.if\ i = 1\ then\ hd(x)\ else\ ap(ap(z, tl(x)), i-1)$$

This specification and the proof of it can be described in QPC, and the map function is extracted from the proof. The application of the map function to some particular function, g of type $\sigma \to \tau$, is performed by $(\forall\text{-}E)$ rule and $\forall$-reduction rule:

$$\dfrac{\dfrac{\Sigma_0}{g : \sigma \to \tau} \quad \dfrac{\dfrac{[f : \sigma \to \tau]}{\dfrac{\Sigma_1}{Spec(f)}}}{\forall f \in \sigma \to \tau.Spec(f)}(\forall\text{-}I)}{Spec(g)}(\forall\text{-}E) \quad \forall\text{-}red \Longrightarrow \quad \dfrac{\dfrac{\Sigma_0}{[g : \sigma \to \tau]} \quad \Sigma_1[g/f]}{Spec(g)}$$

However, if the function, g, is given as a specification $\forall p \in \sigma.\exists q \in \tau.\widetilde{q}(p,q)$ and its proof, the function application cannot be described in QPC.
The second idea of parameterizing f is to universally quantify the input-output relation of the function in $\mathbf{QPC}^2$:

$$\forall^2 P \in \sigma \times \tau \to prop.\forall p \in \sigma.\exists q \in \tau.\ P(p,q) \supset A(P)$$

where $A(P)$ is the relation of the list processing function in terms of P.

$$A(P) \stackrel{def}{=} \forall x \in L(\sigma).\exists y \in L(\tau).\\ length(x) = length(y)\\ \wedge(\forall i \in nat.(1 \leq i \leq length(x) \supset P(elem(x,i), elem(y,i))))$$

The specification is proved by $(\forall^2\text{-}I)$ rule.

4.2 Extraction of program schemata

The extraction rule for $(\forall^2\text{-}I)$ is as follows:

$$Ext\left(\frac{\begin{array}{c}[P:\sigma_1\times\cdots\times\sigma_n\rightarrow prop]\\ \Sigma\\ \hline F(P)\end{array}}{\forall^2 P\in\sigma_1\times\cdots\times\sigma_n\rightarrow prop.\ F(P)}(\forall^2\text{-I})\right) \stackrel{\text{def}}{=} \Lambda P.\ Ext\left(\frac{\Sigma}{F(P)}\right)$$

The definition of Rv is slightly changed: Add the following clause to definition 8:
7) $Rv(P(a_1,\cdots,a_n)) \stackrel{\text{def}}{=} RV(P(a_1,\cdots,a_n))$ where P is a predicate variable and $n \geq 0$

The extracted code from an application of $(\forall^2\text{-}I)$ rule has similarity to second order typed lambda term (Λ-calculus [24]), but it cannot be handled by β-conversion rule only. The code is not regarded as a program, but a P-abstracted program schema.

The program schema extracted from the proof of the second specification in 4.1 is in the form of $\Lambda P.\ T$. T may contain P in the Rv-schema, $RV(P(a,b))$, and $l(P(a,b))$ as in $any[l(P(a,b))]$, $proj\_h(l(P(a,b)),T)$ and $proj\_t(l(P(a,b)),T))$ because Ext try to determine the realizing variables if occurrences of assumptions are discharged by the $(\supset\text{-}I)$, $(\vee\text{-}E)$ and $(\exists\text{-}E)$ applications, and the length of formulas must be determined in the following procedure:

$$Ext\left(\frac{\begin{array}{c}\Sigma\\ \hline A\end{array}}{A\vee B}(\vee\text{-}I)\right) \stackrel{\text{def}}{=} (left, Ext(\Sigma/A), any[l(B)])$$

$$Ext\left(\frac{\begin{array}{c}\Sigma\\ \hline B\end{array}}{A\vee B}(\vee\text{-}I)\right) \stackrel{\text{def}}{=} (right, any[l(A)], Ext(\Sigma/B))$$

$$Ext\left(\frac{\begin{array}{c}\Sigma\\ \hline A\wedge B\end{array}}{A}(\wedge\text{-}E)\right) \stackrel{\text{def}}{=} proj\_h(l(A), Ext(\Sigma/A\wedge B))$$

$$Ext\left(\frac{\dfrac{\Sigma}{A \wedge B}}{B}(\wedge\text{-}E)\right) \stackrel{\text{def}}{=} proj\_t(l(B), Ext(\Sigma/A \wedge B))$$

When this schema is applied to an abstract of type $\sigma \times \tau \to prop$, it is converted to a program. This is performed as follows.
A function of type $\sigma \to \tau$ is expressed as the formula, $\forall p \in \sigma.\exists q \in \tau.\widetilde{q}(p,q)$ and proof of it where $\lambda(p,q).\widetilde{q}$ is some particular abstract of type, $\sigma \times \tau \to prop$. The program schema is applied to the abstract and the following reduction is performed:

$$\begin{array}{c} ap(\Lambda P.\ T, \lambda(p,q).\widetilde{q}) \\ \downarrow \beta\text{-reduction} \\ T[\lambda(p,q).\widetilde{q}/P] \\ \downarrow rv\ reduction \\ T'[\lambda(p,q).\widetilde{q}/P] \\ \downarrow len\text{-}reduction \\ T''[\lambda(p,q).\widetilde{q}/P] \end{array}$$

rv-reduction translates the occurrences of $RV(F[\lambda(p,q).\widetilde{q}/P])$, where F is a formula containing a predicate variable P free, into the value of $Rv(F[\lambda(p,q).\widetilde{q}/P])$. Also, len-reduction translates the occurrences of $l(F[\lambda(p,q).\widetilde{q}/P])$ into the value of it. $T''[\lambda(p,q).\widetilde{q}/P]$ is the obtained program.
Similar problem occurs for the induction proofs. As defined in 2.1.4, the parameter of the μ-term extracted from a proof of $\forall x.A(x)$ in induction is a sequence of new variables of length $l(\forall x.A(x))$. However, if $A(x)$ contains predicate variables, the length cannot be determined. In this case, $Rv(A(x-1))$ or $Rv(A(tl(x)))$ is used for the parameter, and the parameter may contain Rv-schemata.

The same extraction can be performed at proof level. $A(\lambda(p,q).\widetilde{q})$ is proved as follows:

$$\dfrac{\dfrac{\Sigma_0}{\forall p.\exists q.\widetilde{q}(p,q)} \quad \dfrac{\dfrac{\Sigma_1}{\lambda(p,q).\widetilde{q}} \quad \dfrac{\dfrac{\dfrac{[\forall p.\exists q.P(p,q)]}{\dfrac{\Sigma_2}{A(P)}}}{\forall p.\exists q.P(p,q) \supset A(P)}(\supset\text{-}I)}{\forall^2 P.(\forall p.\exists q.P(p,q) \supset A(P))}(\forall^2\text{-}I)}{\forall p.\exists q.\widetilde{q}(p,q) \supset A(\lambda(p,q).\widetilde{q})}(\forall^2\text{-}E)}{A(\lambda(p,q).\widetilde{q})}(\supset\text{-}E)$$

After performing $\forall^2$-reduction, the proof is as follows:

$$\dfrac{\dfrac{\Sigma_0}{\forall p.\exists q.\widetilde{q}(p,q)} \quad \dfrac{\dfrac{[\forall p.\exists q.\widetilde{q}(p,q)]}{\dfrac{\Sigma_2[\lambda(p,q).\widetilde{q}/P]}{A(\lambda(p,q).\widetilde{q})}}}{\forall p.\exists q.\widetilde{q}(p,q) \supset A(\lambda(p,q).\widetilde{q})}(\supset\text{-}I)}{A(\lambda(p,q).\widetilde{q})}(\supset\text{-}E)$$

Then perform the extraction procedure, Ext, to obtain the code:

$$ap(\lambda\overline{F}.T_{\overline{F}}, \overline{f})$$

where $T_{\overline{F}}$ is the code extracted from

$$\frac{[\forall p.\exists q.\widetilde{q}(p,q)]}{\dfrac{\Sigma_2[\lambda(p,q).\widetilde{q}/P]}{A(\lambda(p,q).\widetilde{q})}}$$

and is in the form

$$\mu z.\lambda x.if\ x = nil\ then\ nil\ else\ ap(proj(0)(\overline{F}), hd(x)) :: ap(z, tl(x))$$

if $A(\lambda(p,q).\widetilde{q})$ is proved in induction on lists. $\overline{F} \stackrel{\text{def}}{=} Rv(\forall p.\exists q.\widetilde{q}(p,q))$ and $proj(0)(F)$ is a variable for the value of q bound by $\exists$. Also, $\overline{f}$ is the code extracted from $(\Sigma_0/\forall p.\exists q.\widetilde{q}(p,q))$. If $\supset$-reduction is performed on the last proof, the proof is as follows and the extracted code is $T_{\overline{F}}[\overline{f}/\overline{F}]$.

$$\frac{\dfrac{\Sigma_0}{[\forall p.\exists q.\widetilde{q}(p,q)]}}{\dfrac{\Sigma_2[\lambda(p,q).\widetilde{q}/P]}{A(\lambda(p,q).\widetilde{q})}}$$

4.3 Redundancy in the Extracted Code

For some particular relation $\widetilde{q}$, $Rv(A(\lambda(p,q).\widetilde{q})) = (z, Rv(\widetilde{q}(elem(x,i), elem(y,i))))$ where z is the realizing variable for the value of $\exists y \in L(\tau)$ in $A(\lambda(p,q).\widetilde{q})$ and the value of z is the only code which needs to be extracted. Therefore, if the length of the relation, $\widetilde{q}$, is more than 1, the code extracted from the proof of $A(\lambda(p,q).\widetilde{q})$ has redundancy. Extended projection should be used in this case to eliminate the redundant part of the code.

4.4 Polymorphism

ML-polymorphism [28, 29, 30] can be introduced in the base type theory to obtain an extended system called $\mathbf{QPC}_2^2$. First add the following clause to the definition 11:

3) Type variables, $\alpha, \beta, \cdots$ are types of $type_1$;

Class 2 formulas defined in definition 12 are trivially extended to those containing type variables. The formulas of $\mathbf{QPC}_2^2$ are class 3 formulas defined below:

Definition 15: Class 3 formulas:
1) Class 2 formulas are class 3 formulas;
2) If A is a class 3 formula, then $\forall\alpha \in type_1$. A is a class 3 formula.

Also, the following rules are added:

$$\frac{\begin{array}{c}[\alpha : type_1]\\ A\end{array}}{\forall\alpha \in type_1.A} \qquad \frac{\sigma : type_1 \quad \forall\alpha \in type_1.A}{A[\sigma/\alpha]}$$

The program extraction procedure passes through type abstraction and instantiation because Ext generates type-free programs or program schemata, so that no-type information is needed.

$$Ext\left(\begin{array}{c}[\alpha : type_1]\\ \Sigma\\ \hline A\\ \hline \forall\alpha \in type_1.A\end{array}\right) \stackrel{\text{def}}{=} Ext\left(\begin{array}{c}\Sigma\\ \hline A\end{array}\right)$$

$$Ext\left(\begin{array}{c}\Sigma_0 \qquad \Sigma_1\\ \hline \sigma : type_1 \qquad \forall\alpha \in type_1.A\\ \hline A[\sigma/\alpha]\end{array}\right) \stackrel{\text{def}}{=} Ext\left(\begin{array}{c}\Sigma_1\\ \hline \forall\alpha \in type_1.A\end{array}\right)$$

In terms of realizability, this extraction is defined as follows if A is a class 1 formula without predicate variables:

$$a \ \mathbf{qpc}\ \forall\alpha \in type_1.A \stackrel{\text{def}}{=} \forall\alpha \in type_1.\ (a \ \mathbf{qpc}\ A)$$

This definition is similar to Kreisel-Troelstra realizability [22].

The specification given in 4.1 can be written in $\mathbf{QPC}_2^2$ as follows:

$$\forall\alpha \in type_1.\forall\beta \in type_1.\forall^2 P \in \alpha \times \beta \to prop.\forall p \in \alpha.\exists q \in \beta.\ P(p,q) \supset A'(P)$$

$$\begin{aligned} A'(P) \stackrel{\text{def}}{=} \forall x \in & L(\alpha).\exists y \in L(\beta).\\ & length(x) = length(y)\\ & \wedge(\forall i \in nat.(1 \leq i \leq length(x) \supset P(elem(x,i), elem(y,i)))) \end{aligned}$$

$\mathbf{QPC}_2^2$ will not be used in the following. However, the description in $\mathbf{QPC}^2$ can be trivially translated to that in $\mathbf{QPC}_2^2$.

5. Extraction of *map* function

In this section, the general schema is applied in two cases: $\sigma = \tau = nat$ and the function applied on each element of a σ-list is $succ \equiv \lambda x.x + 1$, and $\sigma = nat$, $\tau = bool$, and the function which returns $t : bool$ if the input natural number is even and $f : bool$ otherwise.

5.1 Specification and Proofs

The specification of *map* is, again, as follows:

$$Map:\ \forall^2 P \in nat \times \tau \to prop.(\forall p \in nat.\exists q \in \sigma.P(p,q) \supset A(P))$$

where

$$\begin{aligned} A(P) \stackrel{\text{def}}{=} & \forall x \in L(nat).\exists y \in L(\sigma). \\ & length(x) = length(y) \\ & \wedge(\forall i \in nat.1 \le i \le length(x) \supset P(elem(x,i), elem(y,i))) \end{aligned}$$

and $\sigma = nat$ or *bool*. It is proved by ($\forall^2$-I) followed by ($\supset$-I) and induction on $x : L(nat)$.

The specifications of successor function and the function which returns a boolean value according to whether the input natural number is even or not are as follows:

$$Succ:\ \forall p \in nat.\exists q \in nat.q = p + 1$$

This is proved by ($\forall$-I) and ($\exists$-I).

$$\begin{aligned} Even\_Odd:\ & \forall p \in nat.\exists q : bool. \\ & ((\exists x \in nat.p = 2 \cdot x \wedge q = t) \vee (\exists y \in nat.p = 2 \cdot y + 1 \wedge q = f)) \end{aligned}$$

This is proved by induction on $p : nat$.

Note that $l(q = p + 1) = 0$ and $l((\exists x \in nat.p = 2 \cdot x \wedge q = t) \vee (\exists y \in nat.p = 2 \cdot y + 1 \wedge q = f)) = 3$, therefore, according to 4.3, the extracted code have redundancy in the case of Map applied to $Even\_Odd$.

5.2 Extracted Codes from the proofs of Map, $Succ$, and $Even\_Odd$

The codes extracted from the proof of the specifications in the previous section are as follows:

$$\begin{aligned} & Map\text{-}term \\ & \equiv \Lambda P.\lambda(y, RV(P(p,q))). \\ & \quad \mu(z_0, RV(P(elem(tl(x),i), elem(y,i)))). \\ & \quad\quad \lambda x.if\ x = nil \\ & \quad\quad\quad then\ (nil, \lambda i.any[l(P(elem(nil,i), elem(nil,i)))]) \\ & \quad\quad\quad else\ (ap(y, hd(x)) :: ap(z_0, tl(x)), \\ & \quad\quad\quad\quad \lambda i.if\ i = 1 \\ & \quad\quad\quad\quad\quad then\ ap(RV(P(a,b)), hd(x)) \\ & \quad\quad\quad\quad\quad else\ ap(ap(RV(P(elem(tl(x),i), elem(y,i))), tl(x)), \\ & \quad\quad\quad\quad\quad\quad i - 1)) \end{aligned}$$

$$Succ\text{-}term \equiv \lambda p.p + 1$$

$Even\_Odd\text{-}term$
$= \mu(z_0, z_1, z_2, z_3)$
$\lambda p.if\ p = 0\ then\ (t, left, 0, any[1])$
$else\ if\ ap(z_1, p-1) = left\ then\ (f, right, any[1], ap(z_2, p \quad 1))$
$else\ (t, left, ap(z_3, p-1) - 1, any[1])$

The following functions can also be extracted by using the extended projection method on $Even\_Odd$:
i) Declaration = $\{1\}$:
$\mu z_1.\lambda p.if\ p = 0\ then\ left\ else\ if\ ap(z_1, p-1) = left\ then\ right\ else\ left$

ii) Declaration = $\{0, 1\}$
$\mu(z_0, z_1).\ \lambda p.\ if\ p = 0\ then\ (t, left)$
$else\ if\ ap(z_1, p-1) = left\ then\ (f, right)\ else\ (t, left)$

iii) Declaration = $\{1, 2, 3\}$
$\mu(z_1, z_1, z_3).\ \lambda p.\ if\ p = 0\ then\ (left, 0, any[1])$
$else\ if\ ap(z_1, p-1) = left\ then\ (right, any[1], ap(z_2, p-1))$
$else\ (left, ap(z_3, p-1) - 1, any[1])$

Other declarations cause overflow of marking numbers. The first program is the most efficient, but it returns the constants, $left$ and $right$, instead of boolean values. The third program returns the triple of boolean values, the constants $left$ or $right$ and $any[1]$.

5.3 Codes from Map applied to $Succ$

In this case, a fully efficient program can be generated by the naive extractor with $\forall^2$-reduction and proof normalization.

5.3.1 Naive Extractor with $\forall^2$-reduction

$$ap(\lambda y.\mu z_0.\lambda x.if\ x = nil\ then\ nil\ ap(y, hd(x)) :: ap(z_0, tl(x)), \lambda p.p + 1)$$

5.3.2 Extractor with Normalizer

$$\mu z.\lambda x.if\ x = nil\ then\ nil\ else\ (hd(x) + 1) :: ap(z, tl(x))$$

$\forall^2$-reduction, $\supset$-reduction, and $\forall$-reduction are performed.

5.4 Codes from Map applied to $Even\_Odd$)

5.4.1 Naive Extractor with $\forall^2$-reduction

$$
\begin{array}{l}
ap(\lambda(x_0,x_1,x_2,x_3).\ \mu(z_0,z_1,z_2,z_3) \\
\quad \lambda x.if\ x = nil \\
\qquad (nil, \lambda i.any[3]) \\
\qquad (ap(x_0, hd(x)) :: ap(z_0, tl(x)), \\
\qquad\ \lambda i.if\ i = 1\ then\ ap((x_1,x_2,x_3), hd(x)) \\
\qquad\qquad\qquad\qquad else\ ap(ap((z_1,z_2,z_3), tl(x)), i-1) \\
\quad \mu(w_0,w_1,w_2,w_3). \\
\qquad \lambda p.if\ p = 0 \\
\qquad\quad then\ (t, left, 0, any[1]) \\
\qquad\quad else\ if\ ap(w_1, p-1) = left \\
\qquad\qquad\quad then\ (f, right, any[1], ap(w_2, p-1)) \\
\qquad\qquad\quad else\ (t, left, ap(w_3, p-1) - 1, any[1])
\end{array}
$$

5.4.2 Naive extractor with normalizaton

$$
\begin{array}{l}
\mu(z_0,z_1,z_2,z_3).\ \lambda x.if\ x = nil \\
\qquad\qquad\qquad (nil, \lambda i.any[3]) \\
\qquad\qquad\qquad let\ (y_0,y_1,y_2,y_3) \\
\qquad\qquad\qquad\quad = ap(\mu(w_0,w_1,w_2,w_3) \\
\qquad\qquad\qquad\qquad\quad \lambda p.if\ p = 0 \\
\qquad\qquad\qquad\qquad\quad\ then\ (t, left, 0, any[1]) \\
\qquad\qquad\qquad\qquad\quad\ else\ if\ ap(w_1, p-1) = left \\
\qquad\qquad\qquad\qquad\qquad\quad then\ (f, right, any[1], ap(w_2, p-1)) \\
\qquad\qquad\qquad\qquad\qquad\quad else\ (t, left, ap(w_3, p-1) - 1, any[1]), \\
\qquad\qquad\qquad\qquad\quad hd(x)) \\
\qquad\qquad\qquad in\ (y_0 :: ap(z_0, tl(x)), \\
\qquad\qquad\qquad\quad \lambda i. \quad if\ i = 1 \\
\qquad\qquad\qquad\qquad\quad then\ (y_1,y_2,y_3) \\
\qquad\qquad\qquad\qquad\quad else\ ap(ap((z_1,z_2,z_3), tl(x)), i-1))
\end{array}
$$

This program still has redundancy, i.e., only the 0th term is needed. Therefore, extended projection should be used.

5.4.3 Extended projection method with normalization

Apply the extended projection method to the normalized proof of
Map applied to $Even\_Odd$ with the declaration $\{0\}$ because 0th code which is the value of $\exists y \in L(bool)$ in the specification is needed. However, overflowed marking numbers are found, so that the declaration must be enlarged. The following are the only cases in which no overflowed marking number occurs.

i) $I = \{0, 1\}$

$$
\begin{array}{l}
\mu(z_0, z_1). \\
\quad \lambda x. if\ x = nil \\
\quad then\ (nil, \lambda i. any[1]) \\
\quad else\ let\ (y_0, y_1) \\
\qquad = ap(\mu(w_0, w_1). \\
\qquad\quad \lambda p. if\ p = 0\ then\ (t, left) \\
\qquad\qquad else\ if\ ap(w_1, p-1) = left\ then\ (t, right)\ else\ (t, left), hd(x)) \\
\qquad in\ (y_0 :: ap(z_0, tl(x)). \\
\qquad\quad \lambda i. if\ i = 1\ then\ y_2\ else\ ap(ap(z_1, tl(x)). i - 1)
\end{array}
$$

ii) $I = \{1, 2, 3\}$

$$
\begin{array}{l}
\mu(z_1, z_2, z_3). \\
\quad \lambda x. if\ x = nil \\
\quad then\ \lambda i. any[3] \\
\quad else\ let\ (y_1, y_2, y_3) \\
\qquad = ap(\mu(w_1, w_2, w_3). \\
\qquad\quad \lambda p. if\ p = 0\ then\ (left, 0, any[1]) \\
\qquad\qquad else\ if\ ap(w_1, p-1) = left\ then\ (right, any[1], ap(w_2, p-1)) \\
\qquad\qquad\quad else\ (left, ap(w_3, p-1) - 1, any[1]), hd(x)) \\
\qquad in\ \lambda i. if\ i = 1\ then (y_1, y_2, y_3)\ else\ ap(ap((z_1, z_2, z_3), tl(x)), i - 1)
\end{array}
$$

6. Extended Projection with Ordinary Projection

The best program extracted extracted from Map applied to $Even\_Odd$ is that given in i) in 5.4.3. However, it is 2-dimensional multi-valued recursive call function, and it turns out that the calculation of the second element of the sequence of terms except in the inner multi-valued recursive call function $\mu(w_0, w_1). \cdots$ in *let* sentence is redundant. This can be made clearer if it is explained at proof tree level.
The structure of the normalized proof of Map applied to $Even\_Odd$ is as follows:

$$
\dfrac{\begin{array}{c}\Sigma_0 \\ \exists y.F(nil, y) \end{array} \quad INDUCTION\ STEP}{\forall x. \exists y. F(x, y)}(L(\sigma)\text{-}ind)
$$

$INDUCTION\ STEP$ is as follows:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\Sigma_1}{hd(x) : nat} \quad \dfrac{\Sigma_2}{\forall p. \exists q. \widetilde{q}(p, q)}}{\exists q. \widetilde{q}(hd(x), q)} \quad \dfrac{\dfrac{[y]^1}{q :: y : L(nat)} \quad \dfrac{\begin{array}{c}[\widetilde{q}(hd(x), q)]^2 \\ [F(tl(x), y)]^4 \end{array} \quad \dfrac{\Sigma_3}{F(x, q :: y)}}{\exists y. F(x, y)}}{\exists y. F(x, y)}(\exists\text{-}E)^2 \quad [\exists y. F(tl(x), y)]}{\exists y. F(x, y)}(\exists\text{-}E)^1
$$

let sentence is generated in the extraction at $(\exists\text{-}E)^2$ application: the code extracted from the tree, $(\Sigma_2/\forall p.\exists q.\tilde{q}(p,q)/\exists q.\tilde{q}(hd(x),q))$, is to be substituted in q and $Rv(\tilde{q}(hd(x),q))$. $\exists y.F(tl(x),y)$ is the induction hypothesis. If the declaration, $\{0\}$, is given to $\forall x.\exists y.F(x,y)$, the marking of the induction hypothesis is also $\{0\}$, in other words, there are no overflowed marking numbers in the outer induction. However, the marking of $\forall p.\exists q.\tilde{q}(p,q)$ also $\{0\}$, and Σ_2 is another induction proof. The marking procedure on Σ_2 finds an overflowed marking number, 1, at the induction hypothesis, $\exists q.\tilde{q}(p-1,q)$, then the marking of $\forall p.\exists q.\tilde{q}(p,q)$ must be enlarged to $\{0,1\}$. This leads the enlargement of the declaration to $\forall x.\exists y.F(x,y)$ into $\{0,1\}$. This is the reason why the code in 5.4.3 i) is 2-dimensional multi-valued recursive call function.
The extracted code can be improved by using the extended projection locally: inner induction proofs are linked with ordinary projection constructor. Then, the final program is as follows:

$$
\begin{array}{l}
\mu z_0. \\
\quad \lambda x.if\ x = nil \\
\qquad then\ nil \\
\qquad else\ let\ (y_0, y_1) \\
\qquad\qquad = ap(\mu(w_0, w_1). \\
\qquad\qquad\qquad \lambda p.if\ p = 0\ then\ (t, left) \\
\qquad\qquad\qquad\quad else\ if\ ap(w_1, p-1) = left\ then\ (t, right) \\
\qquad\qquad\qquad\qquad else\ (t, left).hd(x)) \\
\qquad\qquad in\ y_0 :: ap(z_0, tl(x))
\end{array}
$$

The ordinary projection, $(y_0, y_1) \to y_0$, is performed implicitly in *let*-sentence.

7. Conclusion

Techniques and examples of higher order programming in a second order intuitionistic logic, $\mathbf{QPC}^2$, were presented in this paper. The second order feature itself does not induce polymorphism as in the type theoretic formulation of constructive logic. However, it allows a sort of parametric programming, and by making the base type theory second order, it can be enhanced to a programming system with ML-polymorphism. Program extraction can be performed by using a variant of q-realizability for the first order part and second order normalization method which is comparable to β-reduction of Reynolds-style second order typed lambda calculus. Also, to generate a redundancy-free program, various techniques such as Harrop formula and modified V-code are used. In particular, good use of extended projection combined with ordinary projection is effective for the nested induction proofs. It turned out through the experimental study that extended projection should be used locally in the case of nested induction or in the situation where more than one induction subproof occurs in a whole proof, and the extracted codes from induction subproofs should be linked with *let*-construct.

References

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

Academic Press, 1980
[2] S. Feferman, "Constructive theory of functions and classes", In *Logic Colloquium '78*, North-Holland, Amsterdam, pp. 159-224, 1978
[3] M. Sato, "*Typed Logical Calculus*", Technical Report 85-13, Department of Information Science, Faculty of Science, University of Tokyo, 1985
[4] S. Hayashi and H. Nakano, "*PX – A Computational Logic*", The MIT Press, 1988
[5] S. C. Kleene, "On the interpretation of intuitionistic number theory", Journal of Symbolic Logic Vol. 10, pp.109-124, 1945
[6] A. S. Troelstra, "Mathematical investigation of intuitionistic arithmetic and analysis", Lecture Notes in Mathematics Vol. 344, Springer, 1973
[7] N. G. de Bruijn, "AUTOMATH – A Language for Mathematics", Les Presses de L'universite de Montréal, 1973
[8] N. G. de Bruijn, "A Survey of the Project AUTOMATH", in *Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, pp.579-606, 1980
[9] P. Martin-Löf, "*Intuitionistic Type Theory*", Bibliopolis, Napoli, 1984
[10] J. Smith, "The identification of propositions and types in Martin-Löf's type theory: a programming example", LNCS Vol. 155, 1982
[11] B. Nordström and K. Petersson, "Programming in constructive set theory: some examples", *Proceedings of 1981 Conference on Functional Programming Language and Computer Architecture*, ACM, pp.141-153, 1981
[12] P. Chisholm, "Derivation of Parsing Algorithm in Martin-Löf's Theory of Types", Science of Computer Programming Vol. 8, North-Holland, 1987
[13] C. Mohring, "Algorithm Development in the Calculus of Constructions", *Proceedings of 2nd Annual Symposium on Logic in Computer Science*, 1986
[14] T. Coquand and G. Huet, "The Calculus of Constructions", *Information and Computation*, 76, pp.95-120, 1988
[15] J. L. Bates, "*A logic for correct program development*", Ph.D. Thesis, Cornell University, 1979
[16] J. L. Bates and R. Constable, "Proofs as Programs", AMC Transaction on Programming Languages and Systems, Vol. 7, No. 1, 1985
[17] R. L. Constable, "*Implementing Mathematics with the Nuprl Proof Development System*", Prentice-Hall, 1986
[18] C. Paulin-Mohring, "Extracting F_ω's Programs from Proofs in the Calculus of Constructions", *16th Annual ACM Symposium on Principles of Programming Languages*, 1989
[19] C. A. Goad, "*Computational Uses of the Manipulation of Formal Proofs*", Ph.D. Thesis, Stanford University, 1980
[20] Y. Takayama, "QPC: QJ-Based Proof Compiler – Simple Examples and Analysis–", *Proceedings of 2nd European Symposium on Programming*, LNCS Vol. 300, 1988
[21]Y. Takayama, "Extended Projection – a new technique to extract efficient programs from constructive proofs", *Proceedings of 1989 Conference on Functional Programming Languages and Computer Architecture*, ACM, 1989
[22] G. Kreisel and A. S. Troelstra, "Formal systems for some branches of intuitionistic analysis", Annals of Math. Logic 1, pp229-387, 1979
[23] J. P. Krivine and M. Parigot, "Programming with proofs", Preprint, presented at 6th Symposium on Computation Theory, Wendish-Rietz, Germany, 1987
[24] J. C. Reynolds, "Three approaches to type structure", LNCS Vol. 185, Springer-Verlag, pp. 97-138, 1985

[25] M. J. Beeson, "*Foundation of constructive mathematics*", Springer-Verlag, 1985
[26] M. Sato, "Quty: A Concurrent Language Based on Logic and Function", *Proceedings of the Fourth International Conference on Logic Programming*, The MIT Press, 1987
[27] D. Prawitz, "*Natural Deduction*", Almquist and Wiksell, Stockholm, 1965
[28] M. J. Gordon, R. Milner and C. P. Wadsworth, "*Edinburgh LCF*", LNCS Vol. 78, 1979
[29] L. Dames and R. Milner, "Principal type-schemas for functional programming", Edinburgh University, 1982
[30] G. Huet, "*A Uniform Approach to Type Theory*, preprint, 1988

Appendix The complete proof of the specificaton.

The proof is written in the following backward reasoning style:

```
        < Statement >
        since by  < Rule >
                < Proof >
        end_since;
```

% Function definitions

```
function
    len(X:L(nat)) := if X = nil then 0 else len(tl(x)) + 1
end_function;
```

```
function
    elem(x:L(nat), i:nat)
        := if i = 1 then hd(x) else elem(tl(X), i-1)
end_function;;
```

% Map function

```
theorem /Map/
all P:nat # nat --> prop.
    (all p:nat.  some q:nat.  P(p,q)
     ->
     all x:L(nat).  some y:L(nat).
        (len(x) = len(y)
         & all i:nat.
              (1=<i & i=<len(x)
               -> P(elem(x,i), elem(y,i)))))
since by second_allI  % (∀²-I)
  let P:nat # nat --> prop be arbitrary;
  all p:nat.  some q:nat.  P(p,q)
  ->
  all x:L(nat).  some y:L(nat).
        (len(x)=len(y)
         & all i:nat.
              (1=<i & i=<len(x)
               -> P(elem(x,i), elem(y,i))))
  since  by impI  % (⊃-I)
    assume all p:nat.  some q:nat.  P(p,q);
    all x:L(nat).  some y:L(nat).
            (len(x) = len(y)
           & all i:nat.
                  (1=<i & i=<len(x)
                   -> P(elem(x,i), elem(y,i))))
```

```
since induction on x:L(nat)
  base  % base case of induction
    some y:L(nat).
      (len(nil) = len(y)
      &  all i:nat.
              (1=<i & i=<len(nil)
               -> P(elem(nil,i),elem(y,i))))
    since by exiI
      nil:L(nat) by axiom;
      len(nil) = len(nil)
      & all i:nat.
             (1=<i & i=<len(nil)
              -> P(elem(nil,i),elem(nil,i)))
      since by andI
        len(nil)=len(nil) by axiom;
        all i:nat.
            (1=<i & i=<len(nil)
             -> P(elem(nil,i),elem(nil,i)))
        since by allI   %  (∀-I)
          let i:nat be arbitrary;
          1=<i & i=<len(nil) -> P(elem(nil,i),elem(nil,i))
          since by impI
            assume 1=<i & i=<len(nil);
            P(elem(nil,i),elem(nil,i))
            since by botE
              contradiction
              since by axiom
                1=< i & i=<0 by assumption;
              end_since;
            end_since;
          end_since;
        end_since;
      end_since;
    end_since;
  step     % inducton step
    ind_hyp_is      % induction hypothesis
    some y:L(nat).
         (len(tl(x))=len(y)
         & all i:nat.
               (1=<i & i=<len(tl(x))
                -> P(elem(tl(x),i), elem(y,i))))
    some y:L(nat).
        (len(x)=len(y)
        & all i:nat.
              (1=<i & i=<len(x)
               -> P(elem(x,i), elem(y,i))))
    since by exiE      % (∃-E)
```

```
some y:L(nat).
    (len(tl(x))=len(y)
    & all i:nat.
           (1=<i & i=<len(tl(x))
            -> P(elem(tl(x),i), elem(y,i))))
by assumption;
let yy:L(nat) be such that
    len(tl(x))=len(yy)
   & all i:nat.
          (1=<i & i=<len(tl(x))
           -> P(elem(tl(x),i), elem(yy,i)));
some y:L(nat).  (len(x)=len(y)
             & all i:nat.
                   (1=<i & i=<len(x)
                    -> P(elem(x,i), elem(y,i))))
since by exiE
  some q:nat.  P(hd(x),q)
  since by allE     % (∀-E)
    hd(x) :  nat by axiom;
    all p:nat.  some q:nat.  P(p,q) by assumption;
  end_since;
  let qq:nat be such that P(hd(x),qq);
  some y:L(nat).  (len(x)=len(y)
               & all i:nat.
                      (1=<i & i=<len(x)
                       -> P(elem(x,i), elem(y,i))))
  since by exiI   % (∃-I)
    qq::yy:L(nat) by axiom;
    len(x)=len(qq::yy)
   & all i:nat.
          (1=<i & i=<len(x)
           -> P(elem(x,i), elem(qq::yy,i)))
    since by andI      % (∧-I)
      len(x)=len(qq::yy)
      since by axiom
        len(tl(x))=len(yy)
        since by andE   % (∧-E)
          len(tl(x))=len(yy)
          & all i:nat.
                (1=<i & i=<len(tl(x))
                 -> P(elem(tl(x),i),elem(yy,i)))
          by assumption;
        end_since;
      end_since;
      all i:nat.
          (1=<i & i=<len(x)
           -> P(elem(x,i), elem(qq::yy,i)))
      since by allI
```

```
let i:nat be arbitrary;
1=<i & i=<len(x) -> P(elem(x,i), elem(qq::yy,i))
since by impI
  assume 1=<i & i=<len(x);
  P(elem(x,i), elem(qq::yy,i))
  since divide and conquer   % (∨-E)
    i=1 | (2 =< i & i=<len(x))
    since by axiom
      1=<i & i=<len(x) by assumption;
    end_since;
   case i=1;
    P(elem(x,i),elem(qq::yy,i))
    since by eq_E  % (=-E) s = t ∧ A(s) ⊢ A(t)
      i=1 by assumption;
      P(elem(x,1),elem(qq::yy,1))
      since by eq_E
        elem(x,1)=hd(x) by axiom;
        P(hd(x), elem(qq::yy,1))
        since by eq_E
          elem(qq::yy,1)=qq by axiom;
          P(hd(x),qq) by assumption;
        end_since;
      end_since;
    end_since;
   case 2 =< i & i=<len(x);
    P(elem(x,i), elem(qq::yy,i))
    since by eq_E
      elem(x,i) = elem(tl(x), i-1) by axiom;
      P(elem(tl(x),i-1),elem(qq::yy,i))
      since by eq_E
        elem(qq::yy,i) = elem(yy,i-1) by axiom;
        P(elem(tl(x),i-1),elem(yy,i-1))
        since by impE  % (⊃-E)
          1=<i-1 & i-1=<len(tl(x))
          since by axiom
            2=<i & i=<len(x) by assumption;
          end_since;
          1=<i-1 & i-1=<len(tl(x))
          -> P(elem(tl(x),i-1),elem(yy,i-1))
          since by allE
            i-1:nat by assumption;
            all i:nat.
              (1=<i & i=<len(tl(x))
              -> P(elem(tl(x),i),elem(yy,i)))
            since by andE
              len(tl(x))=len(yy) &
              all i:nat.
```

```
                                    (1=<i & i=<len(tl(x))
                                    -> P(elem(tl(x),i),elem(yy,i)))
                                  by assumption;
                                end_since;
                              end_since;
                            end_since;
                          end_since;
                        end_since;
                      end_since;
                    end_since;
                  end_since;
                end_since;
              end_since;
            end_since;
          end_since;
        end_since;
      end_since;
    end_since;
end_since;
end_theorem;;


theorem /Succ/
   all p:nat.  some q:nat.  q = p + 1
since allI
  let p:nat be arbitrary;
  some q:nat.  q = p + 1
  since by exiI
    p+1:nat since by axiom
              p:nat by assumption;
            end_since;
    p+1 = p+1 by axiom;
  end_since;
end_since;
end_theorem;;


theorem /Even_Odd/
    all p:nat.  some q:bool.
        ((some x:nat.  p = 2*x      &  q = t)
          |(some y:nat.  p = 2*y + 1  &  q = f))
since induction on p:nat
  base
    some q:bool.
        ((some x:nat.  0 = 2*x      &  q = t)
          |(some y:nat.  0 = 2*y + 1  &  q = f))
    since by exiI
      t:bool by axiom;
         (some x:nat.  0=2*x  & t=t)
```

```
    | (some y:nat.  0=2*y+1 & t=f)
    since by orI1  % (∨-I) A ⊢ A ∨ B
      some x:nat.  0=2*x & t=t
      since by andI
        some x:nat.  0=2*x
        since by exiI
          0:nat by axiom;
          0 = 2*0 by axiom;
        end_since;
        t=t by axiom;
      end_since;
    end_since;
  end_since;
step
  ind_hyp_is
    some q:bool.
       ((some x:nat.  p-1 = 2*x & q=t)
       | (some y:nat.  p-1 = 2*y+1 & q=f))
  some q:bool.
    ((some x:nat.  p=2*x & q=t)
     | (some y:nat.  p=2*y+1 & q=f))
  since by exiE
    some q:bool.
       ((some x:nat.  p-1 = 2*x & q=t)
       | (some y:nat.  p-1 = 2*y+1 & q=f))
    by assumption;
    let qq:bool be such that
        (some x:nat.  p-1=2*x & qq=t)
      | (some y:nat.  p-1=2*y+1 & qq=f);
    some q:bool.
      ((some x:nat.  p=2*x & q=t)
       | (some y:nat.  p=2*y+1 & q=f))
    since divide and conquer
      (some x:nat.  p-1=2*x & qq=t)
      | (some y:nat.  p-1=2*y+1 & qq=f) by assumption;
     case some x:nat.  p-1=2*x & qq=t;
       some q:bool.
         ((some x:nat.  p=2*x & q=t)
          | (some y:nat.  p=2*y+1 & q=f))
       since by exiI
         f:bool by axiom;
         (some x:nat.  p=2*x & f=t)
         | (some y:nat.  p=2*y+1 & f=f)
         since by orI2  % (∨-I) B ⊢ A ∨ B
           some y:nat.  p=2*y+1 & f=f
           since by exiE
             some x:nat.  p-1 = 2*x
```

```
          since by andE
            some x:nat.  p-1=2*x & qq=t by assumption;
          end_since;
          let xx:nat be such that p-1 = 2 * xx;
          some y:nat.  p=2*y+1 & f=f
          since by andI
            some y:nat.  p = 2*y + 1
            since by exiI
              xx:nat by assumption;
              p=2 * xx + 1 by assumption;
            end_since;
            f=f by axiom;
          end_since;
        end_since;
      end_since;
    end_since;
  case some y:nat.  p-1=2*y+1 & qq=f;
    some q:bool.
      ((some x:nat.  p=2*x & q=t)
       | (some y:nat.  p=2*y+1 & q=f))
    since by exiI
      t:bool by axiom;
      (some x:nat.  p=2*x & t=t)
      | (some y:nat.  p=2*y+1 & t=f)
      since by oxiE
        some y:nat.  p-1 = 2*y + 1
        since by andE
          some y:nat.  p-1 = 2*y + 1
          & qq=f by assumption;
        end_since;
        let yy:nat be such that p-1=2*yy + 1;
        (some x:nat.  p=2*x & t=t)
        | (some y:nat.  p=2*y+1 & t=f)
        since by orI1
          some x:nat.  p=2*x & t=t
          since by andI
            some x:nat.  p=2*x
            since by exiI
               yy-1:nat by assumption;
               p=2*(yy-1) by assumption;
            end_since;
            t=t by axiom;
          end_since;
        end_since;
      end_since;
    end_since;
end_since;
```

```
    end_since;
end_since;
end_theorem;;


theorem /Ap(Map, Succ)/
all x:L(nat).  some y:L(nat).
      (len(x)=len(y)
      & all i:nat.
           (1=<i & i=<len(x)
            -> elem(y,i) = elem(x,i) + 1))
since by impE
  all p:nat.  some q:nat.  q=p+1  by /Succ/;
  all p:nat.  some q:nat.  q=p+1
  -> all x:L(nat).some y:L(nat).
        (len(x)=len(y)
        & all i:nat.  (1=<i & i=< len(x)
                       -> elem(y,i) = elem(x,i) + 1))
  since by second_allE   % (∀²-E)
       [p:nat,q:nat](q=p+1) :  nat # nat --> prop by axiom;
       all P:nat # nat --> prop.
         (all p:nat.  some q:nat.  P(p,q)
          ->
          all x:L(nat).  some y:L(nat).
             (len(x) = len(y)
             & all i:nat.
                    (1=<i & i=<len(x)
                     -> P(elem(x,i), elem(y,i)))))
       by /Map/;
  end_since;
end_since;
end_theorem;;


theorem /Ap(Map, Evan_Odd)/
all x:L(nat).  some y:L(bool).
      (len(x)=len(y)
      & all i:nat.
           (1=<i & i=<len(x)
            -> ((some a:nat.  elem(x,i)=2*a & elem(y,i)=t)
               |(some b:nat.  elem(x,i)=2*b+1 & elem(y,i)=f))
           ))
since by impE
  all p:nat.  some q:bool.
      ((some a:nat.  p = 2*a      &  q = t)
        |(some b:nat.  p = 2*b + 1  &  q = f))  by /Even_Odd/;
  all p:nat.  some q:bool.
      ((some a:nat.  p = 2*a      &  q = t)
```

```
       |(some b:nat.  p = 2*b + 1  &  q = f))
  -> all x:L(nat).some y:L(bool)
       (len(x)=len(y)
       & all i:nat.
              (1=<i & i=< len(x)
               ->
                ((some a:nat.  elem(x,i)=2*a & elem(y,i)=t)
                 |(some b:nat.  elem(x,i)=2*b+1 & elem(y,i)=f))
                ))
  since by second_allE
       [p:nat,q:bool]((some a:nat.  p=2*a & q=t)
                     | (some b:nat.  p=2*b+1 & q=f))
       :  nat # bool --> prop by axiom;
       all P:nat # nat --> prop.
         (all p:nat.  some q:nat.  P(p,q)
          ->
          all x:L(nat).  some y:L(bool).
             (len(x) = len(y)
             & all i:nat.
                    (1=<i & i=<len(x)
                     -> P(elem(x,i), elem(y,i)))))
       by /Map_1/;
% Obtained by making the type of  P nat × bool → prop
  end_since;
end_since;
end_theorem;;
```