ICOT Technical Report: TR-539

TR-539

QPC^2: A Second Order Logic for High Order Programming

by
Y. Takayama (Oki)

February, 1990

To be submitted for publication in Theoretical Computer Science

$\mathbf{QPC}^2$: A Second Order Logic for Higher Order Programming

Yukihide Takayama

System Laboratory, OKI Electric Co. Ltd.
11-22 Sibaura 4 chome, Minato-ku, Tokyo, Japan
takayama@okilab.oki.co.jp

ABSTRACT

A predicative second order constructive logic for programming, $\mathbf{QPC}^2$ is presented in this paper. It is obtained by introducing predicate and type variables and universal quantification over them to a modified sugared subset of QJ. The form of second order formulas in $\mathbf{QPC}^2$ is restricted and the restriction is inspired by parametric polymorphism as in ML. The logic, although it has only a weak second order feature, allows various kinds of higher order programming, and extraction of programs in a variant of untyped λ-calculus from second order proofs is rather easy.
Keywords: second order logic, realizability, program extraction

1. Introduction

As is well known, programs can be developed in constructive logics. The basic idea is to regard a formula in the form of $\forall x \in \sigma.\exists y \in \tau.A(x,y)$ as a specification of a function, f, from the input of type σ to the output of type τ. A proof of the formula can be regarded as abstract presentation of the function, and can be translated to the function by realizability interpretation. The translation is called program extraction from constructive proofs. On the other hand, one of the chief feature of programming is higher order programming: application of a function to another function. Therefore, a constructive logic should also be higher order to capture higher order programming.
One way to realize a higher order constructive logic is to introduce function variables and quantification over them, in other words, a first order constructive logic with function variables such as QJ [1] [2] and QPC [3]. For example, a specification of map-function can be described as a formula in the form of $\forall f \in \sigma \to \tau.\forall x \in L(\sigma).\exists y \in L(\tau).A(f,x,y)$ where $\sigma \to \tau$ is the type of functions from σ to τ and $L(\sigma)$ and $L(\tau)$ are the types of lists over σ and τ. The specification and its proof can be applied to other functions, say successor function if both σ and τ are natural number type. The application can be described as a proof in the elimination rule of first order universal quantifier. However, if a function, g, is given as a specification, $\forall x.\exists y.B(x,y)$ type formula, and its proof, application of the map-function to g cannot be described as a proof procedure.

Another way is to introduce predicate (proposition) variables and quantification over them, i.e., second (or ω) order logics. This approach has been well developed as higher order typed lambda calculi or constructive type theories such as the Calculus of Constructions [4], second order lambda calculus [5], and F_ω [6]. For non type theoretic approach, an impredicative second order logic called AF_2 [7] has been given. However, the chief interest in AF_2 is to define various data structures as in [8].

This paper presents a predicative second order constructive logic for programming, $\mathbf{QPC^2}$. Predicate variables, type variables, and universal quantification over them are allowed in addition to quantification over function variables. Unlike type theoretic formulation of constructive logics based on Curry-Howard corresponding [9], types and formulas are clearly separated: types are only to categorize expressions such as terms, propositions and propositional functions (predicates). For program extraction from second order proofs extracted programs are usually described in second order typed lambda calculi as seen in the program extractor for the Calculus of Construction [10]. However, if the form of (second order) universally quantified formulas are restricted, program extraction from second order proofs can be rather simple and can generate programs in a variant of untyped lambda calculus in many cases. The restriction of second order formulas was inspired by parametric polymorphism of ML [11] in which type variables are interpreted as universally quantified and, if types are regarded as propositions, the type system represents a predicative second order logic. Also, higher order programming can well be described in spite of the restriction of the formulas.

The structure of the paper is as follows. Section 2 and 3 defines the language and rules of $\mathbf{QPC^2}$. Section 4 presents the method of program extraction. A variant of **q**-realizability for second order logic is given. The way how higher order programming can be performed in $\mathbf{QPC^2}$ is explained in section 5. Section 6 gives an example of program extraction. A map-function and its optimization are demonstrated. Section 7 gives final remarks and conclusion.

2. The Language of $\mathbf{QPC^2}$

2.1 Terms of $\mathbf{QPC^2}$

The terms of $\mathbf{QPC^2}$ denote programs in a variant of untyped λ-calculus. Sequences of terms, if-then-else, let-sentences and fixed point operator are available in addition to lambda abstraction and application.

Definition 1: terms

1) $0, 1, \cdots$ (natural numbers) are terms;

2) $left$, $right$, any, nil, t, f (constants) are terms;

3) $x, y, z, \cdots$ (individual variables) are terms;

4) If $M_1, \cdots, M_n$ are terms , then $(M_1, \cdots, M_n)$ (sequence of terms) is a term;
() denotes nil sequence. A sequence of variables is often denoted $\overline{x}$. $any[n]$ denotes the sequence $(any, \cdots, any)$ of length n;

5) If M is a term and X is a variable or a sequence of variables, then $\lambda X.M$ (λ terms) is a term;

6) If M and N are terms, then $ap(M, N)$ (application) is a term;

7) If M and N are terms, and if A is a (in)equality of terms, then $if\ beval(A)\ then\ M\ else\ N$ (if-then-else) is a term;
8) If M is a term and Z is a variable or a sequence of variables, then $\mu Z.M$ (μ-term) is a term;
9) If X is a variable or a sequence of variables, and if T and M are terms, then $let\ X = T\ in\ M$ (let sentence) is a term;
10) *succ*, *pred*, *tseq*, *ttseq*, *proj*, *beval*, ::, and *app* (built-in functions) are terms.

$if\ beval(A)\ then\ M\ else\ N$ will often be abbreviated to $if\ A\ then\ M\ else\ N$ in the following. μ-terms denote (multi-valued) recursive call functions. Let $\overline{s} \stackrel{\text{def}}{=} (s_1, \cdots, s_n)$ be any sequence of terms, then

$$proj(k)(\overline{s}) = s_k \quad (1 \leq k \leq n)$$
$$tseq(k)(\overline{s}) = (s_k, s_{k+1}, \cdots, s_n) \quad (1 \leq k \leq n)$$
$$ttseq(k,l)(\overline{s}) = (s_k, s_{k+1}, \cdots, s_{k+l-1}) \quad (1 \leq k \leq n, 1 \leq l \leq n-k+1)$$

beval decides whether input (in)equation holds and returns boolean values, t or f. :: and *app* are the cons and the append function of lists. *succ* and *pred* are successor and predecessor functions. It is also possible to extend the term structure by introducing arithmetic operators.
In the following, X, Y, Z, $\cdots$ denote variables or sequences of variables.

2.2 Types of $\mathbf{QPC}^2$

The types in $\mathbf{QPC}^2$ are separated in three categories: $type_1$, $type_2$ and $type_3$. $type_1$ types specify the domains of variables quantified by $\forall$ and $\exists$, and $type_2$ types specify the domains of $\forall^2$. $type_3$ types are to categorize the codes extracted from proofs.

Definition 2: $type_1$ types
1) *nat*, *bool*, and **2** (primitive types) are $type_1$ types;
2) $\alpha, \beta, \cdots$ (type variables) are $type_1$ types;
3) If σ and τ are $type_1$ types, then $\sigma \times \tau$ (Cartesian product) and $\sigma \rightarrow \tau$ (function space) are $type_1$ types;
4) If σ is a $type_1$ type, then $L(\sigma)$ (type of lists over σ) is a $type_1$ type.

Definition 3: $type_2$ types
1) $type_1$ and *prop* (proposition type) are $type_2$ types;
2) If $\sigma : type_1$, then $\sigma \rightarrow prop$ (predicate type) is a $type_2$ type.

As can be seem from the definition of predicate type, $\forall^2$ quantifies variables over formulas without $\forall^2$, in other words, $\mathbf{QPC}^2$ is a predicative second order logic. $\forall^2$ also quantifies type variables over $type_1$.

Definition 4: $type_3$ types
1) If σ is a $type_1$ type, then σ is a $type_3$ type;
2) If σ is a $type_1$ type and τ is a $type_3$ type, then $(\sigma \rightarrow prop) \rightarrow \tau$ is a $type_3$ type.

2.3 Formulas of $\mathbf{QPC}^2$

The formulas of $\mathbf{QPC}^2$ have three caregories: $class_1 \subset class_2$, and $class_3$. $class_1$ categorizes formulas which may contain predicate variables and type variables free, and $class_2$ formulas are closure of $class_1$ formulas, in other words, universally quantification of predicate variables and type variables in the given $class_1$ formulas. $class_3$ forumulas are typing relations.
In the following, P, Q, $\cdots$ denote predicate variables. Each predicate variable is assigned a natural number called *arity* which means the number of parameters. A predicate variable with arity 0 is called proposition variable.

Definition 5: $class_1$ formula
1) If M and N are terms, then $M = N$, $M < N$, $M \leq N$, and $\perp$ are $class_1$ formulas (atomic formulas);
2) If P is a n-ary predicate variable and $M_1 \cdots M_n$ are terms, then $P(M_1, \cdots, M_n)$ is a $class_1$ formula;
3) If A and B are $class_1$ formulas and σ is a $type_1$ type, then $A \wedge B$, $A \vee B$, $A \supset B$, $\forall x \in \sigma.A(x)$, and $\exists x \in \sigma.A(x)$ are $class_1$ formulas.

Negation of a formula, A, is defined as $\neg A \overset{\text{def}}{=} A \supset \perp$.

Definition 6: $class_2$ formula
1) If A is a $class_1$ formula, then A is a $class_2$ formula;
2) If A is a $class_1$ formula, then $\forall^2 X_1 \in T_1. \cdots \forall^2 X_n \in T_n.A$ is a $class_2$ formula where T_i is a $type_2$ type, and if $T_i = type_1$ then X_i is a type variable, otherwise, X_i is a predicate (proposition) variable.

Definition 7: Abstract
If A is a $class_1$ formula which contains variables $x_1, \cdots, x_n$, which are not predicate variables or type variables, free, then an expression, $\lambda(x_1, \cdots, x_n).A$ is called a predicate or an abstract.

Definition 8: $class_3$ formula
1) If M is a term and σ is a $type_1$ type, then $M : \sigma$ is a $class_3$ formula;
2) $\sigma : type_1$ is a $class_3$ formula;
3) If $\lambda\overline{x}.A$ is an abstract and $\sigma \rightarrow prop$ is a $type_2$ type, then $\lambda\overline{x}.A : \sigma \rightarrow prop$ is a $class_3$ formula.

2.4 Scheme

A scheme is roughly a mixture of terms and predicate variables. Unlike terms, schemes cannot be used in proofs, they are used only to describe the codes extracted from second order proofs.

Definition 9: Scheme
1) If M is a term, then M is a scheme;
2) If P is a predicate variable and M_i $(1 \leq i \leq n)$ is a term, then $any[L(P(M_1, \cdots, M_n))]$

(any code scheme) and $RV(P(M_1,\cdots,M_n))$ (Rv-scheme) are schemes;
3) If P is a predicate variable and T is a scheme, then $\Lambda P.T$ (program scheme) is a scheme;
4) If T is a scheme and if Q is a predicate variable or an abstract, then $ap(T,Q)$ is a scheme.

A Rv-scheme, $RV(P(M_1,\cdots,M_n))$, is an expression which contains a predicate variable P free. The meaning of Rv-scheme is that it denotes the realizing variable sequences, which will be explained in section 4, of a formula which is obtained by substituting a predicate to the free predicate variable. A Rv-scheme can, therefore, be regarded as a variable for the realizing variable sequences. The expression, $L(P(M_1,\cdots,M_n))$, in an any code scheme, $any[L(P(M_1,\cdots,M_n))]$, denotes the length of $P(M_1,\cdots,M_n)$ which will also be defined in section 4. The term, $any[n]$, can naturally be extended to the case in which $n = m + L(P(M_1,\cdots,M_n))$ and $n = L(P(M_1,\cdots,M_n)) + m$ (m is a natural number.):

$$any[m + L(P(M_1,\cdots,M_n))] \equiv (any[m], any[L(P(M_1,\cdots,M_n))])$$

$$any[L(P(M_1,\cdots,M_n)) + m] \equiv (any[L(P(M_1,\cdots,M_n))], any[m])$$

Both of them are regarded as sequences of length $m+1$.

In the following, substitution of a scheme, T, to a variable (or sequence of variables), X, which occurs free in an expression E is denoted $E_X[T]$. If X is a sequence of variables then T must also be a sequence of scheme of the same length. $E_{x_1,\cdots,x_n}[M_1,\cdots,M_n]$ denotes simultaneous substitution. If A is a formula, $A_x[M]$ is also denoted $A(M)$.

3. The Rules of **QPC^2**

3.1 Rules on Scheme Calculus

3.1.1 Term Equivalence Rules

$$ap((M_1,\cdots,M_n),N) \equiv (ap(M_1,N),\cdots,ap(M_n,N))$$

$$\lambda X.(M_1,\cdots,M_n) \equiv (\lambda X.M_1,\cdots,\lambda X.N_n)$$

$$\begin{aligned}&if\ A\ then\ (M_1,\cdots,M_n)\ else\ (N_1,\cdots,N_n)\\ &\quad\equiv (if\ A\ then\ M_1\ else\ N_1,\cdots,if\ A\ then\ M_n\ else\ N_n)\end{aligned}$$

$$let\ X = T\ in\ (M_1,\cdots,M_n)\ \equiv (let\ X = T\ in\ M_1,\cdots,let\ X = T\ in\ M_n)$$

$$\mu(z_1,\cdots,z_n).(M_1,\cdots,M_n) \equiv (f_1,\cdots,f_n)$$

where $f_i = \mu z_i.(M_i)_{z_1,\cdots,z_{i-1},z_{i+1},\cdots,z_n}[f_1,\cdots,f_{i-1},f_{i+1},\cdots,f_n]$

3.1.2 Reduction Rules

$M \longrightarrow N$ means that the term M is reduced to N.

$$\frac{M_1 \longrightarrow N_1 \quad M_2 \longrightarrow N_2}{(M_1,N_1) \longrightarrow (N_1,N_2)}$$

$$ap(\lambda x.M, N) \longrightarrow M_x[N]$$

$$\frac{N \equiv (N_1, \cdots, N_n)}{ap(\lambda(x_1, \cdots, x_n).M, N) \longrightarrow M_{x_1,\cdots,x_n}[N_1, \cdots, N_n]}$$

$$\frac{beval(A) = t}{if\ beval(A)\ then\ M\ else\ N \longrightarrow M}$$

$$\frac{beval(A) = f}{if\ beval(A)\ then\ M\ else\ N \longrightarrow N}$$

$$let\ x = T\ in\ M \longrightarrow M_x[T]$$

$$\frac{T \equiv (T_1, \cdots, T_n)}{let\ (x_1, \cdots, x_n) = T\ in\ M \longrightarrow M_{x_1,\cdots,x_n}[T_1, \cdots, T_n]}$$

$$\frac{M \equiv (M_1, \cdots, M_n) \quad \mu(z_1, \cdots, z_n).(M_1, \cdots, M_n) \equiv (f_1, \cdots f_n)}{\mu(z_1, \cdots, z_n).M \longrightarrow M_{z_1,\cdots,z_n}[f_1, \cdots, f_n]}$$

$$ap(\Lambda P.T, S) \longrightarrow T_P[S]$$

3.2 Type Rules

3.2.1 $type_1$ types

$$\frac{\sigma : type_1 \quad \tau_i : type_1\ (1 \le i \le n)}{\sigma \to (\tau_1 \times \cdots \times \tau_n) \equiv (\sigma \to \tau_1) \times \cdots \times (\sigma \to \tau_n)}$$

$$\frac{\sigma : type_1 \quad \tau : type_1 \quad M : \sigma \quad \sigma \equiv \tau}{M : \tau}$$

$$\frac{\sigma\ is\ a\ type_1\ type}{\sigma : type_1}$$

$$\frac{}{n : nat} \quad (n = 0, 1, \cdots) \qquad \frac{\sigma : type_1}{any : \sigma}$$

$$\frac{}{t : bool} \qquad \frac{}{f : bool} \qquad \frac{}{left : \mathbf{2}} \qquad \frac{}{right : \mathbf{2}}$$

$$\frac{\sigma : type_1}{nil : L(\sigma)} \qquad \frac{\sigma : type_1 \quad M : \sigma \quad N : L(\sigma)}{M :: N : L(\sigma)} \qquad \frac{\sigma : type_1 \quad M : L(\sigma) \quad N : L(\sigma)}{app(M, N) : L(\sigma)}$$

$$\frac{\sigma : type_1 \quad \tau : type_1 \quad M : \sigma \quad N : \tau}{(M, N) : \sigma \times \tau}$$

$$\frac{\sigma : type_1 \quad \tau : type_1 \quad \begin{matrix}[X : \sigma] \\ M : \tau\end{matrix}}{\lambda X.M : \sigma \to \tau}$$

$$\frac{\sigma : type_1 \quad \tau : type_1 \quad M : \sigma \to \tau \quad N : \sigma}{ap(M, N) : \tau}$$

$$\frac{\sigma : type_1 \quad \tau : type_1 \quad S : \sigma \quad T : \sigma \quad M : \tau \quad N : \tau}{if\ beval(S = T)\ then\ M\ else\ N : \tau}$$

$$\frac{\sigma : type_1 \quad S : nat \quad T : nat \quad M : \sigma \quad N : \sigma}{if\ beval(S\ R\ T)\ then\ M\ else\ N : \sigma} \quad (R = \leq, \geq, < or >)$$

$$\frac{\begin{array}{r}[X : \sigma]\\ \sigma : type_1 \quad T : \sigma \quad M : \sigma\end{array}}{let\ X = T\ in\ M : \sigma}$$

$$\frac{\begin{array}{r}[Z : \sigma \to \tau]\\ \sigma \to \tau : type_1 \quad T : \sigma \to \tau\end{array}}{\mu Z.T : \sigma \to \tau}$$

$$\frac{\sigma : type_1 \quad M : \sigma \quad N : L(\sigma)}{M :: N : L(\sigma)} \qquad \frac{\sigma : type_1 \quad M : L(\sigma) \quad N : L(\sigma)}{app(M, N) : L(\sigma)}$$

$$\frac{k : nat\ (1 \leq k \leq n) \quad N : \sigma_1 \times \cdots \times \sigma_n}{tseq(k)(N) : \sigma_k \times \cdots \times \sigma_n}$$

$$\frac{k : nat\ (1 \leq k \leq n) \quad l : nat\ (1 \leq l \leq (n - k + 1)) \quad N : \sigma_1 \times \cdots \times \sigma_n}{ttseq(k, l)(N) : \sigma_k \times \cdots \times \sigma_{k+l-1}}$$

$$\frac{k : nat\ (1 \leq k \leq n) \quad N : \sigma_1 \times \cdots \times \sigma_n}{proj(k)(N) : \sigma_k}$$

$$\frac{\alpha : type_1}{RV(P(M_1, \cdots, M_n)) : \alpha} \qquad \frac{\alpha : type_1}{any[L(P(M_1, \cdots, M_n))] : \alpha}$$

3.2.2 $type_2$ types

$$\frac{A\ is\ a\ class_1\ formula}{A : prop}$$

$$\frac{\sigma : type_1 \quad M : \sigma \quad N : \sigma}{beval(M = N) : bool} \qquad \frac{M : nat \quad N : nat}{beval(M\ R\ N) : bool} \quad (R \equiv \leq, \geq, < or >)$$

$$\frac{\begin{array}{l}[X : \sigma]\\ p : prop \quad \sigma : type_1\end{array}}{\lambda X.p : \sigma \to prop} \qquad \frac{\sigma : type_1 \quad \lambda X.p : \sigma \to prop \quad T : \sigma}{p_X[T] : prop}$$

3.2.3 $type_3$ types

$$\frac{\begin{array}{c}[P : \sigma \to prop]\\ T : \tau\end{array}}{\Lambda P.T : (\sigma \to prop) \to \tau} \qquad \frac{\Lambda P.T : (\sigma \to prop) \to \tau \quad S : \sigma \to prop}{ap(\Lambda P.T, S) : \tau}$$

3.3 Logical Rules

3.3.1 First Order Rules

In the following A, B, C, $A(0)$, $A(nil)$, $A(x)$, $A(pred(x))$ $A(tl(x))$ are $class_1$ formula, i.e., of type $prop$.

- Rules on $\wedge$, $\vee$ and $\supset$

$$\frac{A \quad B}{A\wedge B}(\wedge I) \qquad \frac{A\wedge B}{A}(\wedge E)_1 \qquad \frac{A\wedge B}{B}(\wedge E)_2$$

$$\frac{A}{A\vee B}(\vee I)_1 \qquad \frac{B}{A\vee B}(\vee I)_2 \qquad \frac{A\vee B \quad \begin{array}{c}[A]\\ C\end{array} \quad \begin{array}{c}[B]\\ C\end{array}}{C}(\vee E)$$

$$\frac{\begin{array}{c}[A]\\ B\end{array}}{A\supset B}(\supset I) \qquad \frac{A\supset B \quad A}{B}(\supset E)$$

- Induction

$$\frac{\begin{array}{c}[x=0\vee(x>0\wedge A(pred(x)))]\\ A(x)\end{array}}{\forall x\in nat.A(x)}(nat\text{-}ind) \qquad \frac{\begin{array}{c}[x=nil\vee(x\neq nil\wedge A(tl(x)))]\\ A(x)\end{array}}{\forall x\in L(\sigma).A(x)}(L(\sigma)\text{-}ind)$$

These rules are usually used in the following form:

$$\frac{A(0) \quad \begin{array}{c}[A(pred(x))]\\ A(x)\end{array}}{\forall x\in nat.A(x)} \qquad \frac{A(nil) \quad \begin{array}{c}[A(tl(x))]\\ A(x)\end{array}}{\forall x\in L(\sigma).A(x)}$$

- Rules on quantifiers

$$\frac{\sigma : type_1 \quad \begin{array}{c}[x:\sigma]\\ A(x)\end{array}}{\forall x\in\sigma.A(x)}(\forall I) \qquad \frac{\sigma : type_1 \quad M:\sigma \quad \forall x\in\sigma.A(x)}{A(M)}(\forall E)$$

$$\frac{\sigma : type_1 \quad M:\sigma \quad A(M)}{\exists x\in\sigma.A(x)}(\exists I) \qquad \frac{\exists x\in\sigma.A(x) \quad \begin{array}{c}[x:\sigma, A(x)]\\ C\end{array}}{C}(\exists E)$$

- Rule on absurdity

$$\frac{\perp}{A}(\perp E)$$

- Rules on equalities

In the following, M, N and S are terms, and A is a $class_1$ formula.

$$\frac{\sigma : type_1 \quad M:\sigma}{M=M} \qquad \frac{M=N}{N=M} \qquad \frac{M=N \quad N=S}{M=S}$$

$$\frac{M \equiv N}{M = N} \qquad \frac{M \longrightarrow N}{M = N} \qquad \frac{M = N \quad A_x[M]}{A_x[N]}(= E)$$

3.3.2 Second Order Rules

$$\frac{\begin{array}{c}[\alpha : type_1]\\ A\end{array}}{\forall^2 \alpha \in type_1.A}(\forall^2 I)_t \qquad \frac{\sigma : type_1 \quad \forall^2 \alpha \in type_1.A}{A_\alpha[\sigma]}(\forall^2 E)_t$$

$$\frac{\sigma : type_1 \quad \begin{array}{c}[P : \sigma \to prop]\\ A\end{array}}{\forall^2 P \in \sigma \to prop.A}(\forall^2\text{-}I)_p \qquad \frac{\lambda X.p : \sigma \to prop \quad \forall^2 P \in \sigma \to prop.A}{Conv(A_P[\lambda X.p])}(\forall^2\text{-}E)_p$$

$Conv(A_P[\lambda X.p])$, also denoted just $A_P[\lambda X.p]$, is obtained by performing β-reduction

$$(\lambda X.p)(M) \overset{\mathrm{def}}{=} ap(\lambda X.p, M) \longrightarrow p_X[M]$$

in $A_P[\lambda X.p]$.

3.4 Some Properties

The following properties are easily proved.

Proposition 1: *Assume that A is a $class_2$ formula in the form of $\forall^2 X \in T.F$. Then, if A is proved in $\mathbf{QPC}^2$, the last rule used in the proof is either $(\forall^2 I)_t$, $(\forall^2 I)_p$, $(\forall^2 E)_t$ or $(\forall^2 E)_p$*

This is because the form of $class_2$ formulas is restricted, and any first order rules are only for $class_1$ formulas.

Proposition 2: *Assume a $class_2$ formula in the form of $\forall^2 X_1 \in T_1.\cdots\forall^2 X_n \in T_n.A$ where A is a $class_1$ formula. If the formula is provable in $\mathbf{QPC^2}$, then there exists a proof in $\mathbf{QPC}^2$ in which last n rules applied are either $(\forall^2 I)_p$ or $(\forall^2 I)_t$, and these rules are not used in other part.*

This can be proved by using proof normalization on the second order universal quantifier [13].

Corollary: *Assume that A is a $class_1$ formula. Then, if A is proved in $\mathbf{QPC^2}$, there is a proof of A which does not use $(\forall^2 I)_t$, $(\forall^2 I)_p$, $(\forall^2 E)_t$, or $(\forall^2 E)_p$*

Proposition 3: *Assume that M and N are terms such that $M \equiv N$. If $M : \sigma$ $(\sigma : type_1)$, then $N : \sigma$.*

Proposition 4: *Assume that M and N are terms such that $M \longrightarrow N$. If $M : \sigma$ $(\sigma : type_1)$, then $N : \sigma$.*

Because μ-terms have types, typability does not mean termination property of the term. Moreover, it is possible to use non-terminating term, M, in $(\forall E)$ and $(\exists I)$:

$$\frac{\sigma : type_1 \quad M : \sigma \quad \forall x \in \sigma.A(x)}{A(M)}(\forall E) \qquad \frac{\sigma : type_1 \quad M : \sigma \quad A(M)}{\exists x \in \sigma.A(x)}(\exists I)$$

4. Program Extraction

4.1 qpc-realizability

Definition 10: Realizability relation
Assume that A is a $class_2$ formula and that $\overline{z} = (z_1, \cdots, z_n)$ is a sequence of variables which does not occur free in A. Then, $\overline{z}$ **qpc** A, which reads "$\overline{z}$ realizes A", is called a **qpc**-realizability relation, or **qpc**-realizability for short. $\overline{z}$ is called a sequence of realizing variables, or realizing variable sequence, of A.

Definition 11: QPC^{2+}
QPC^{2+} is a trivial extension of **QPC^2** by adding all the realizability relation as formulas.

Definition 12: qpc-realizability
1) If A is atomic, then () **qpc** $A \stackrel{\text{def}}{=} A$;
2) If P is a predicate variable,
then $\overline{a}$ **qpc** $P(M_1, \cdots, M_n) \stackrel{\text{def}}{=} \overline{a} = RV(P(M_1, \cdots, M_n)) \wedge P(M_1, \cdots, M_n)$
3) $\overline{a}$ **qpc** $A \supset B \stackrel{\text{def}}{=} \forall b \in \sigma.(A \wedge b$ **qpc** $A \supset ap(\overline{a}, b)$ **qpc** $B)$
where σ is such that $\overline{a} : \sigma \to \tau$ for some τ
4) $(\overline{a}, \overline{b})$ **qpc** $\exists x \in \sigma.A \stackrel{\text{def}}{=} \overline{a} : \sigma \wedge A_x[\overline{a}] \wedge \overline{b}$ **qpc** $A_x[\overline{a}]$
5) $\overline{a}$ **qpc** $\forall x \in \sigma.A \stackrel{\text{def}}{=} \forall x \in \sigma.(ap(\overline{a}, x)$ **qpc** $A)$
6) $(z, \overline{a}, \overline{b})$ **qpc** $A \vee B \stackrel{\text{def}}{=} (z = left \wedge A \wedge \overline{a}$ **qpc** $A) \vee (z = right \wedge B \wedge \overline{b}$ **qpc** $B)$
7) $(\overline{a}, \overline{b})$ **qpc** $A \wedge B \stackrel{\text{def}}{=} \overline{a}$ **qpc** $A \wedge \overline{b}$ **qpc** B
8) $\overline{a}$ **qpc** $\forall^2 P \in \sigma.A \stackrel{\text{def}}{=} \forall^2 P \in \sigma.(ap(\overline{a}, P)$ **qpc** $A)$
where σ is a $type_2$ type other than $type_1$
9) $\overline{a}$ **qpc** $\forall^2 \alpha \in type_1.A \stackrel{\text{def}}{=} \forall^2 \alpha \in type_1.(\overline{a}$ **qpc** $A)$

Note that the interpretation of $\forall^2$ formulas varies according to which types second order variables are quantified on: the clause 8) is similar to the typing rules of second order typed lambda terms, and the clause 9) is the same as Kreisel-Troelstra realizability [12] which is also used in AF_2 [7]. The intention of clauses 8) and 9) is that predicates which have computational meaning should be substituted to predicate variables while type information should be removed in the program extraction.

Proposition 5: *For a realizability relation $\overline{z}$* **qpc** *A and a scheme e, e* **qpc** *$A \supset\subset (\overline{z}$* **qpc** *$A)_{\overline{z}}[e]$ can be provable in* **QPC^{2+}**

From the definition of **qpc**-realizability, a sequence of realizing variables can be determined by the structure of the given formula as follows:

Definition 13: $Rv(A)$ (sequence of realizing variables)
1) $Rv(A) = ()$ if A is atomic;
2) $Rv(P(M_1, \cdots, M_n)) = RV(P(M_1, \cdots, M_n))$ if P is a predicate variables;
3) $Rv(A \wedge B) = (Rv(A), Rv(B))$;
4) $Rv(A \supset B) = Rv(B)$;
5) $Rv(\forall x \in \sigma.A) = Rv(A)$;
6) $Rv(A \vee B) = (z, Rv(A), Rv(B))$ (z is a new variable);
7) $Rv(\exists x \in \sigma.A) = (z, Rv(A))$ (z is a new variable);
8) $Rv(\forall^2 \alpha \in type_1.A) = Rv(A)$;
9) $Rv(\forall^2 P \in \sigma \to prop.A) = Rv(A)$.

Definition 14: Length of formula
Assume that A is any $class_2$ formula in $\mathbf{QPC}^{2+}$. Then, the length of A, $l(A)$, is the length of $Rv(A)$ as a sequence of variables.

Note that $l(A)$ is the number of $\exists$, $\vee$ and $P(M_1, \cdots, M_n)$ type formulas that occur in the strictly positive part of A, and these logical connectives and formulas can be pointed by a position number i ($1 \leq i \leq l(A)$).

Typing of realizing variables can be performed in $type_3$ types from the structure of the given formula as follows:

$$\frac{Rv(A):\sigma \quad Rv(B):\tau}{Rv(A \wedge B):\sigma \times \tau} \qquad \frac{Rv(A):\sigma \quad Rv(B):\tau}{Rv(A \supset B):\sigma \to \tau} \qquad \frac{Rv(A):\tau}{Rv(\forall x \in \sigma.A):\sigma \to \tau}$$

$$\frac{Rv(A):\sigma \quad Rv(B):\tau}{Rv(A \vee B):\mathbf{2} \times \sigma \times \tau} \qquad \frac{Rv(A):\tau}{Rv(\exists x \in \sigma.A):\sigma \times \tau}$$

$$\frac{Rv(A):\sigma}{Rv(\forall^2 \alpha \in type_1.A):\sigma} \qquad \frac{Rv(A):\tau}{Rv(\forall^2 P \in \sigma \to prop.A):(\sigma \to prop) \to \tau}$$

4.2 Properties of **qpc** realizability

Proposition 6: *Let A be a $class_1$ formula which contains no predicate variables. If A is realizable, i.e, there is a scheme, M, such that M **qpc** A, then A is provable in $\mathbf{QPC}^2$.*

Proof: By induction on the construction of A and the definition of **qpc**-realizability. ∎

Theorem 1: Soundness of **qpc**-realizability
*Assume that A is a $class_2$ formula in $\mathbf{QPC}^2$. If A is proved in $\mathbf{QPC}^2$, then (1) there is a scheme, e, such that e **qpc** A can be proved in $\mathbf{QPC}^{2+}$;* (2) *e is typed by the type of $Rv(A)$, i.e., if $Rv(A):\sigma$ then $e:\sigma$; (3) $FV(e) \subset FV(A)$*

Proof: The proof of (1) is performed by induction on the structure of proof trees. (2) and (3) are easily checked.

If the proof tree is A, then let $e = Rv(A)$. Let R be the name of the last rule which is used in the proof tree of A. If A is atomic, then let $e = ()$. Other cases are as follows:

case $R = (\wedge I)$: Assume that $A \equiv B \wedge C$. Let a and b be schemes such that a **qpc** B and b **qpc** C. Then, let e be (a, b). e **qpc** A can be proved by $(\wedge I)$.

case $R = (\wedge E)_1$: Assume that $A \wedge B$ is the premise of the R application. Let a be a scheme such that a **qpc** $A \wedge B$, then let $e = ttseq(1, l(A))(a)$. e **qpc** A can be proved by $(\wedge E)_1$.

case $R = (\wedge E)_2$: Assume that $B \wedge A$ is the premise. Let a be a scheme such that a **qpc** $B \wedge A$, then let $e = tseq(l(B) + 1)(a)$. e **qpc** A can be proved by $(\wedge E)_2$.

case $R = (\vee I)_1$: Assume that $A \equiv B \vee C$ and B be the premise. Let a be a scheme such that a **qpc** B, then let $e = (left, a, any[len(C)])$ where $len(C)$ is defined as follows:

1) $len(A) = 0$ if A is atomic;
2) $len(P(M_1, \cdots, M_n)) = L(P(M_1, \cdots, M_n))$ if P is a predicate variable;
3) $len(A \wedge B) = len(A) + len(B)$;
4) $len(A \supset B) = len(B)$
5) $len(\forall x \in \sigma.A) = len(A)$;
6) $len(A \vee B) = 1 + len(A) + len(B)$;
7) $len(\exists x \subset \sigma.A) = 1 + len(A)$.

Note that $len(A) = l(A)$ if A does not contain any predicate variable because the only difference between $l(A)$ and $len(A)$ is $l(P(M_1, \cdots, M_n)) = 1$. c **qpc** A can be proved by $(\vee I)$.

case $R = (\vee I)_2$; Assume that $A \equiv B \vee C$ and C be the premise. Let b be a scheme such that b **qpc** C, then let $e = (right, any[len(B)], b)$. e **qpc** A can be proved by $(\vee I)$.

case $R = (\vee E)$; Assume that $B \vee C$ is the first premise of the R application. Let a be a scheme such that a **qpc** $B \vee C$, and b and c be the scheme which realize A as the second and the third premises of the R application. Note that b and c may contain $Rv(B)$ and $Rv(C)$. Then let $e = if\ proj(1)(a) = left\ then\ (let\ Rv(B) = ttseq(2, l(B))(a)\ in\ b)\ else\ (let\ Rv(C) = tseq(l(B) + 2)(a)\ in\ c)$. e **qpc** A can be proved by $(\vee E)$ and $(= E)$.

case $R = (\supset I)$; Assume that $A \equiv B \supset C$ and C be the premise. Let a be a scheme such that a **qpc** C. a may contains $Rv(B)$. Then, let $e = \lambda Rv(B).a$. e **qpc** A can be proved by $(\forall I)$, $(\supset I)$ and $(= E)$.

case $R = (\supset E)$; Assume that $B \supset A$ and B are the premises. Let a and b be schemes such that a **qpc** $B \supset A$ and b **qpc** B. Then, let $e = ap(a, b)$. e **qpc** A can be proved by $(\supset E)$, $(\wedge I)$ and $(\forall E)$.

case $R = (\forall I)$; Assume that $A \equiv \forall x \in \sigma.B$. Let a be a scheme such that a **qpc** A, then let $e = \lambda x.a$. e **qpc** A can be proved by $(\forall I)$ and $(= E)$.

case $R = (\forall E)$; Assume that $M : \sigma$ and $\forall x \in \sigma.B(x)$ are the premises. Let a be a scheme such that a **qpc** $\forall x \in \sigma.B(x)$. Then, let $e = ap(a, M)$. e **qpc** A can be proved by $(\forall E)$.

case $R = (\exists I)$; Assume that $M : \sigma$ and $B(M)$ be the premises. Let a be a scheme such that a **qpc** $B(M)$. Then, let $e = (M, a)$. e **qpc** A can be proved by $(\wedge I)$.

case $R = (\exists E)$; Assume that $\exists x \in \sigma.B(x)$ is the first premise. Let a be a scheme such that a **qpc** $\exists x \in \sigma.B(x)$ and b be a scheme which realizes A as the second premise. Then,

let $e = let\ (x, Rv(B)) = a\ in\ b$. The proof of e **qpc** A can be obtained from the proof of b **qpc** A by substituting $(proj(1)(a), tseq(1)(a))$ to $(x, Rv(B))$, and replacing $proj(1)(a) : \sigma$ and $tseq(1)(a)$ **qpc** $B(x)$ as the discharged hypotheses by the proof of a **qpc** $\exists x \in \sigma.B(x)$.
case $R = (= E)$; Assume that $A \equiv A_x[N]$ and that $M = N$ and $A_x[M]$ be the premises. Let a be a scheme such that a **qpc** $A_x[M]$. Then, let $e = a$. e **qpc** A can be proved by $(= E)$.
case $R = (\bot E)$; Let $e = any[len(A)]$. e **qpc** A can be proved by $(\bot E)$.
case $R = (L(\sigma)\text{-}ind)$; Assume that $A \equiv \forall x \in L(\sigma).B(x)$, and A is proved as follows:

$$\dfrac{\dfrac{\Sigma_0}{B(nil)} \qquad \dfrac{\genfrac{}{}{0pt}{}{[x \neq nil \wedge B(tl(x))]}{\Sigma_1}}{B(x)}}{\forall x \in L(\sigma).B(x)}$$

This proof can be translated to the following:

$$\dfrac{\dfrac{[x = nil \vee (x \neq nil \wedge B(tl(x)))] \qquad \dfrac{[x = nil] \quad \dfrac{\Sigma_0}{B(nil)}}{B(x)}(= E) \qquad \dfrac{\genfrac{}{}{0pt}{}{[x \neq nil \wedge B(tl(x))]}{\Sigma_1}}{B(x)}}{B(x)}(\vee E)}{\forall x \in L(\sigma).B(x)}(L(\sigma)\ ind)$$

By the induction hypothesis, following proofs in $\mathbf{QPC}^2$ exist:

$$\dfrac{\Sigma_0'}{a\ \mathbf{qpc}\ B(nil)} \qquad \dfrac{\Sigma_1'}{b\ \mathbf{qpc}\ B(x)}$$

Also, $x = nil \vee (x \neq nil \wedge B(tl(x)))$ is realized by its realizing variables: $(z, \overline{x}) \overset{\text{def}}{=} (z, Rv(B(tl(x))))$, and $(z, \overline{x})$ **qpc** $x = nil \vee (x \neq nil \wedge B(tl(x))) \equiv (z = left \wedge x = nil) \vee (z = right \wedge x \neq nil \wedge B(tl(x)) \wedge \overline{x}$ **qpc** $B(tl(x)))$. Let $e' = if\ z = left\ then\ a\ else\ b$. Then, the the following proof can be constructed:

$$\dfrac{\dfrac{H_0 \qquad \dfrac{\dfrac{\genfrac{}{}{0pt}{}{[H_1]}{\Sigma_3}}{e' = a} \quad \genfrac{}{}{0pt}{}{[H_1]}{\Pi_0}}{e'\ \mathbf{qpc}\ B(x)}(= E) \qquad \dfrac{\dfrac{\genfrac{}{}{0pt}{}{[H_2]}{\Sigma_4}}{e' = b} \quad \genfrac{}{}{0pt}{}{[H_2]}{\Pi_1}}{e'\ \mathbf{qpc}\ B(x)}(= E)}{e'\ \mathbf{qpc}\ B(x)}(\vee E)}{\forall x \in L(\sigma).e'\ \mathbf{qpc}\ B(x)}(L(\sigma)\text{-}ind)$$

where $H_0 \overset{\text{def}}{=} (z, \overline{x})$ **qpc** $(x = nil \vee (x \neq nil \wedge B(tl(x))))$, $H_1 \overset{\text{def}}{=} z = left \wedge x = nil$, $H_2 \overset{\text{def}}{=} z = right \wedge x \neq nil \wedge B(tl(x)) \wedge \overline{x}$ **qpc** $B(tl(x))$ and Π_0 and Π_1 are as follows:

$$\dfrac{\dfrac{[H_1]}{x = nil}(\wedge E) \qquad \dfrac{\Sigma_0'}{a\ \mathbf{qpc}\ B(nil)}}{a\ \mathbf{qpc}\ B(x)}(= E) \qquad\qquad \dfrac{\genfrac{}{}{0pt}{}{[H_2]}{\Sigma_1'}}{b\ \mathbf{qpc}\ B(x)}$$

Let $\overline{z}$ be a new sequence of variables of length $l(B)$, and assume the equation: $\overline{z} = \lambda x.e'_{\overline{x}}[ap(\overline{z}, tl(x))]$. The solution of this equation is the following recursive call function:

$f \overset{\text{def}}{=} \mu\overline{z}.\lambda x.if\ z = left\ then\ a\ else\ b_{\overline{x}}[ap(\overline{z}, tl(x))]$. f is equivalent to a sequence of schemes $f_1, \cdots, f_n$ $(n = l(B))$. Note that $z = left \supset\subset x = nil$ can be proved in $\mathbf{QPC}^2$, so that f can be regarded as $\mu\overline{z}.\lambda x.if\ x = nil\ then\ a\ else\ b_{\overline{x}}[ap(\overline{z}, tl(x))]$. Note also that $ap(f,x) = e^{l}_{\overline{x}}[ap(f, tl(x))]$ can be proved. By substituting $ap(f, tl(x))$ to $\overline{x}$ in the above proof, $\forall x \in L(\sigma).(e^{l}_{\overline{x}}[ap(f, tl(x))]$ **qpc** $B(x)) \equiv \forall x \in L(\sigma).(ap(f,x)$ **qpc** $B(x)) \equiv f$ **qpc** $\forall x \in L(\sigma).B(x)$ can be proved. Therefore, let e be f.
case $R = (nat\text{-}ind)$; Similar to the previous case.
Let $e = \mu\overline{z}.\lambda x.if\ x = 0\ then\ a\ else\ b_{\overline{z}}[ap(\overline{z}, x-1)]$ where $\overline{z}$ is a new sequence of variables of length $l(A)$.
case $R = (\forall^2 I)_p$: Assume that $A \equiv \forall^2 P \in \sigma \to prop.B$ and the proof is as follows:

$$\frac{\begin{array}{c}[P : \sigma \to prop] \\ \Sigma \\ \overline{B}\end{array}}{\forall^2 P \in \sigma \to prop.B}(\forall^2\text{-}I)_p$$

Let a be the scheme such that a **qpc** B which is constructed from the subproof (Σ/B). Then, $\Lambda P.a$ **qpc** $\forall^2 P \in \sigma \to prop.B \overset{\text{def}}{=} \forall^2 P \in \sigma \to prop.(ap(\Lambda P.a, P)$ **qpc** $B) = \forall^2 P \in \sigma \to prop.(a$ **qpc** $B)$. Therefore, let $e = \Lambda P.a$. e **qpc** A can be proved by $(\forall^2 I)_p$
case $R = (\forall^2 I)_t$: Assume that $A \equiv \forall^2 \alpha \in type_1.B$ and that the proof is as follows:

$$\frac{\begin{array}{c}[\alpha : type_1] \\ \Sigma \\ \overline{B}\end{array}}{\forall^2 \alpha \in type_1.B}(\forall^2 I)_t$$

Let a be a scheme such that a **qpc** B. Then let $e = a$. e **qpc** A can be proved by $(\forall^2 I)_t$.
case $R = (\forall^2 E)_p$: Assume that $A \equiv B_p[\lambda\overline{x}.p]$ and that the proof is as follows:

$$\frac{\dfrac{\Sigma_0}{\lambda\overline{x}.p \in \sigma \to prop} \quad \dfrac{\Sigma_1}{\forall^2 P \in \sigma \to prop.B}}{B_P[\lambda\overline{x}.p]}(\forall^2 E)_p$$

By proposition 2, this proof can be normalized to reduce the case of $R = (\forall^2 I)_p$ or a first order rule.
case $R = (\forall^2 E)_t$ Assume that $A \equiv A_\alpha[\sigma]$ and that the proof is as follows:

$$\frac{\dfrac{\Sigma_0}{\sigma : type_1} \quad \dfrac{\Sigma_1}{\forall^2 \alpha \in type_1.A}}{A_\alpha[\sigma]}(\forall^2 E)_t$$

By proposition 2, the above proof can be normalized to reduce the case of $R = (\forall^2 I)_t$ or a first order rule.

∎

Corollary:
Assume that A is a $class_1$ formula which does not contain any predicate variables. If A is proved in $\mathbf{QPC}^2$, then there is a term, e, which realizes A.

The proof of the theorem can be formalized as a program extractor procedure, *Ext*, in a straightforward way as in [2] and [3].

4.3 Optimization

Three kinds of optimization technique can be used for first order proofs.

(1) Proof normalization
As is well known, some of the proof normalization rules (See [13] or [14]) correspond to partial evaluation of the extracted programs. For example, if the hypothesis A occurs only once in Σ_0 in the following proof,

$$\dfrac{\dfrac{\dfrac{\begin{matrix}[A]\\ \Sigma_0\end{matrix}}{B}}{A \supset B}(\supset I) \qquad \dfrac{\Sigma_1}{A}}{B}(\supset E)$$

then $\supset$-reduction optimizes the extracted code by performing β-reduction:
$ap(\lambda Rv(A).a, b) \longrightarrow a_{Rv(A)}[b]$ where $\lambda Rv(A).a$ and b are terms extracted from the subproofs of $A \supset B$ and A. If A occurs more than twice in Σ_0, the reduction rule substitutes copes of b to the occurrences of $Rv(A)$ in a. Therefore, proof normalization does not actually optimize the extracted code in this case if the program is executed in call-by-value strategy.

(2) Modified $\vee$ code
The decision procedure in *if-then-else* code which is extracted from a proof in the $(\vee E)$ rule can be simplified in some cases. If A and B are both (in)equalities of terms in the major premise, $A \vee B$, of a $(\vee E)$ application, the extracted decision procedure may just A. This optimization can also be extended to the case in which either A or B is a (in)equality of terms. See [3] for the detail.

(3) Extended projection
This technique is to remove redundancy in the extracted codes. For example, from a proof of $\exists x \in \sigma.A(x)$, **qpc**-realizability extracts a code in the form of (t, s) in which t is a term such that $A(t)$ holds, and s is a term extracted from the subproof of $A(t)$. The code s is often redundant. In the extended projection technique, if the position number of $\exists$ and $\vee$ in the specification which corresponds to redundant code is given, a procedure which is similar to strictness analysis with abstract interpretation propagates the information to each node of the proof tree. The code extractor generates redundancy free code from the annotated proof tree. Similar method is given in [15], [16], [17], and [10], but the extended projection allows more fine-grained specification of redundancy and an algorithm of semi-automatic analysis of redundancy is given. See [18] for the detail.

5. Writing Specifications

Three examples, map-function, sorting, and user defined rules of inference, are investigated to demonstrate the expressive power of $\mathbf{QPC}^2$.

5.1 Map Function

Two kinds of specification of map-function is possible.

(1) Specification using a function variable
The function which the map-function takes as input is described in a function variable and universal quantification.

$$\begin{array}{l}
\forall^2\alpha \in type_1.\forall^2\beta \in type_1.\forall f \in \alpha \to \beta.\\
\qquad \forall x \in L(\alpha).\exists y \in L(\beta)\\
\qquad\qquad length(x) = length(y)\\
\qquad\qquad \wedge(\forall i \in nat.1 \leq i \leq length(x) \supset f(elem(i,x)) = elem(i,y))
\end{array}$$

where $length(x)$ and $elem(i,x)$ are functions which calculates the length of x and ith element of x. They can be defined in $\mathbf{QPC}^2$.
The specification can be proved by $(\forall^2 I)_t$, $(\forall I)$, and $(L(\alpha)\text{-}ind)$. The extracted code will be as follows:

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

Let the specification be, for simplicity, $\forall^2\alpha.\forall^2\beta.\forall f.MAP(\alpha,\beta,f)$. Application of map-function to other function, say $succ \overset{\text{def}}{=} \lambda x.x+1$, can be described as follows as a proof procedure: Because $nat : type_1$, $\forall f.MAP(nat,nat,f)$ can be proved by $(\forall E)_t$. Also, because $\lambda x.x+1 : nat \to nat$, then $MAP(nat,nat,succ)$ can be proved by $(\forall E)$.
This schema has a problem: the map-function cannot be applied to a function which is defined as a specification and its proof.

(2) Specification using a predicate variable
The input function of the map function can be described as $\forall x.\exists y.P(x,y)$ with a predicate variable, P, which is universally quantified.

$$\begin{array}{l}
\forall^2\alpha \in type_1.\forall^2\beta \in type_1.\forall^2 P \in \alpha\times\beta \to prop.\\
\qquad (\forall p \in \alpha.\exists q \in \beta.P(p,q)\\
\qquad\quad \supset \forall x \in L(\alpha).\exists y \in L(\beta).\\
\qquad\qquad\qquad length(x) = length(x)\\
\qquad\qquad\qquad \wedge\forall i \in nat.(1 \leq i \leq length(x)\\
\qquad\qquad\qquad\qquad\qquad \supset P(elem(i,x), elem(i,y))))
\end{array}$$

This specification can be proved by two applications of $(\forall^2 I)_t$ followed by $(\forall^2 I)_p$, $(\supset I)$ and $(L(\alpha)\text{-}ind)$, and from which a program scheme can be extracted. Let the specification be, for simplicity, $\forall^2\alpha.\forall^2\beta.\forall^2 P.(\forall p.\exists q.P(p,q) \supset SPEC(\alpha,\beta,P))$. Application of the map function to *even_odd* given as a specification and its proof can be described as a proof procedure. A specification of *even_odd* can be described as follows:

$$\begin{array}{l}
\forall p \in nat.\exists q \in bool.\\
\qquad ((\exists x \in nat.p = 2\cdot x \wedge q = t) \vee (\exists y \in nat.p = 2\cdot y + 1 \wedge q = f))
\end{array}$$

This can be proved by $(nat\text{-}ind)$. Let the specification be $\forall p.\exists q.EVOD(p,q)$. Then, the application is as follows:

$$\dfrac{\dfrac{\Sigma_0}{\forall p.\exists q.EVOD(p,q)} \qquad \dfrac{\dfrac{\Sigma_1}{\lambda(p,q).EVOD} \quad \dfrac{\Sigma_2}{\forall^2 P.(\forall p.\exists q.P(p,q) \supset SPEC(nat,bool,P))}}{\forall p.\exists q.EVOD(p,q) \supset SPEC(nat,bool,\lambda(p,q).EVOD)}(\forall^2 E)_p}{SPEC(nat,bool,\lambda(p,q).EVOD)}(\supset E)$$

where Σ_0 is a proof of the specification of *even_odd*, Σ_1 is a proof that $\lambda(p,q).EVOD$ is an abstract of type $nat \times bool \to prop$, and Σ_2 is a proof of elimination of $\forall^2$ quantifiers on α and β followed by the proof of the specification of map-function. This proof can be normalized by using one $\supset$-reduction and three $\forall^2$-reductions (one is for $\forall^2 P$ and others are for $\forall^2\alpha$ and $\forall^2\beta$). The normalized proof does not contain the second order rules or predicate variables, so that a term can be extracted.

5.2 Sorting Program

Total order relation on the elements of the lists to be sorted can be generalized by using predicate variables.

SORTING:

$$\begin{aligned}&\forall^2\alpha \in type_1.\forall^2 P \in \alpha \times \alpha \to prop.\\ &\quad (REL(P,\alpha)\\ &\qquad \supset \forall x : L(\alpha).\exists y : L(\alpha).PERM(x,y) \wedge SORTED(y,P))\end{aligned}$$

where $REL(P,\alpha)$ is a formula which means that P is a total order on α, $PERM(x,y)$ means that y can be obtained by some permutation of x, $SORTED(y,P)$ means that y is sorted with the order relation, P:

$$SORTED(y,P) \overset{\text{def}}{=} \forall i \in nat.\forall j \in nat.(1 \le i < j < length(y) \supset P(elem(i,y), elem(j,y)))$$

The specification can be proved by $(\forall^2 I)_t$, $(\forall^2 I)_p$, and $(\supset I)$.
This specification can be applied, for example, to the following total order relation on $nat \times nat$ type.

$$lex(x,y) \overset{\text{def}}{=} proj(1)(x) < proj(1)(y) \vee (proj(1)(x) = proj(1)(y) \wedge proj(2)(x) \le proj(2)(y))$$

By using $(\forall^2 E)_t$, $(\forall^2 E)_p$, and two $\forall^2$-reductions, the following first order proof can be obtained:

$$\dfrac{\Sigma}{\begin{aligned}&REL(lex, nat \times nat)\\ &\quad \supset \forall x : L(nat \times nat).\exists y : L(nat \times nat).PERM(x,y) \wedge SORTED(y, lex)\end{aligned}}\,(\supset I)$$

The term extracted from this proofs takes a justification of $REL(lex, nat \times nat)$ and any list, x, of type $L(nat \times nat)$ and returns the sorted version of x.

5.3 Use Defined Rules of Inference

As is well known, well-founded induction must be used to write a proof which corresponds to quick sort algorithm. The induction is available after proving the following formula:

$$\begin{aligned}&\forall^2\alpha \in type_1.\forall^2 F \in L(\alpha) \to prop.\forall^2 P \in \alpha \times \alpha \to prop.\\ &\quad (REL(P)\\ &\qquad \supset (\forall x \in L(\alpha).\forall y \in L(\alpha).(P(y,x) \supset F(y)) \supset F(x)\\ &\qquad\qquad \supset \forall z \in L(\alpha).F(z)))\end{aligned}$$

This is proved by $(\forall^2 I)_t$, $(\forall^2 I)_p$, $(\supset I)$ and $(L(\alpha)\text{-}ind)$.

6. Example of Program Extraction

The code extracted from the second specification of map-function and its application to other function will be given.

(1) The code extracted form a proof of the specification is as follows:

```
MAP ≡ $\Lambda P.\lambda(x_0, RV(P(p,q)))$.
          $\mu(z_0, RV(P(elem(i,tl(x)), elem(i,x_0))))$.
               $\lambda x.if\ x = nil$
                  $then\ (nil, \lambda i.any[L(P(elem(i,nil), elem(i,nil)))])$
                  $else\ (ap(x_0,hd(x)) :: ap(z_0,tl(x))$,
                       $\lambda i.if\ i = 1$
                          $then\ ap(RV(P(p,q)), hd(x))$
                          $else\ ap(ap(RV(P(elem(i,tl(x)), elem(i,x_0))), tl(x))$,
                               $i - 1))$
```

This is not a program but a program scheme because it contains an abstraction (ΛP), Rv-scheme ($RV(P(p,q))$, $RV(P(elem(i,tl(x)),elem(i,x_0)))$), and any code scheme ($any[L(P(elem(i,nil),elem(i,nil)))]$). $(x_0, RV(P(p,q)))$ is $Rv(\forall p \in \alpha.\exists q \in \beta.P(p,q))$ which is extracted from the assumption $\forall p \in \alpha.\exists q \in \beta.P(p,q)$ discharged in the application of ($\supset I$), and $(z_0, RV(P(elem(i,tl(x)),elem(i,x_0))))$ is the parameter of recursive call function which is extracted from the ($L(\alpha)$-*ind*) proof on $x \in L(\alpha)$. Informally, the recursive call function is to calculates a sequence of terms: the first element is the value of $y \in L(\beta)$ (nil and $ap(x_0,hd(x)) :: ap(z_0,tl(x))$) and the rest of the sequence is the justification of $length(x) = length(y) \wedge \forall i \in nat.(1 \leq i \leq length(x) \supset P(elem(i,x),elem(i,y)))$ ($\lambda i.any[L(P(elem(i,nil),elem(i,nil)))]$ and
$\lambda i.if\ i = 1\ then\ ap(RV(P(p,q)),hd(x))$
$else\ ap(ap(RV(P(elem(i,tl(x)),elem(i,x_0))),tl(x)),i-1))$.
The justification part is redundant in many cases. More specifically, if the predicate which will be substituted to P has any computational contents, the justification part causes redundant code. For example, if the map function is applied to the successor function, $\forall p \in nat.\exists q \in nat.q = p+1$ and its proof, i.e., $q = p+1$ is substituted to P, no redundancy occurs.

(2) The following program is generated from the proof describing application of map-function to *even_odd*. The specification of *even_odd* is substituted to the predicate variable,

P, in the proof of the second specification of map-function.

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

The function, $\mu(w_0, w_1, w_2, w_3).\lambda p.\cdots$, is the code extracted from a proof of the specification of *even_odd*. The top level application, $ap(\_, \_)$, corresponds to $(\supset E)$ rule. If $\supset$-reduction is performed before code extraction, the following code is obtained:

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

This program has a lot of redundancy. It can be removed by using extended projection method. Therefore, the final code obtained is as follows:

```
$\mu z_0.\lambda x.if\ x = nil$
        $then\ nil$
        $else\ let\ (y_0, y_1)$
                $= ap(\mu(w_0, w_1).$
                        $\lambda p.if\ p = 0\ then\ (t, left)$
                            $else\ if\ ap(w_1, p - 1) = left\ then\ (t, right)$
                                 $else\ (t, left),$
                    $hd(x))$
             $in\ y_0 :: ap(z_0, tl(x))$
```

(3) Application of the map function with the specification in 5.1 (2) to other function can also be performed at the extracted code level. The scheme, MAP, obtained in (1) can be applied to a specification of a function of type $\alpha \to \beta$ by introducing the following additional reduction rules on schemes:

$$RV((\lambda X.p)(M)) \longrightarrow Rv(p_X[M])$$

$$any[L((\lambda X.p)(M))] \longrightarrow any[len[p_X[M]]]$$

where $\lambda X.p$ is an abstract and $(\lambda X.p)(M)$ is obtained from $P(X)$ by substitution.

For example, the code obtained by applying MAP to the input-output relation of *even_odd*, $\lambda(p,q).((\exists x \in nat.p = 2 \cdot x \wedge q = t) \vee (\exists y \in nat.p = 2 \cdot x + 1 \wedge q = f))$, is as follows:

```
$\lambda(x_0, w_0, w_1, w_2).$
    $\mu(z_0, w_0', w_1', w_2').$
        $\lambda x.if\ x = nil$
            $then\ (nil, \lambda i.any[3])$
            $else\ (ap(x_0, hd(x)) :: ap(z_0, tl(x)),$
                $\lambda i.if\ i = 1$
                    $then\ ap((w_0, w_1, w_2), hd(x))$
                    $else\ ap(ap((w_0', w_1', w_2'), tl(x)),$
                        $i - 1))$
```

This code can be applied to *even_odd* function extracted from a proof of the specification. However, it is better that the application is performed as a proof procedure because the optimization technique can be used at proof level.

7. Discussion and Conclusion

A predicative second order constructive logic, $\mathbf{QPC^2}$, was presented. It allows predicate variables which range over predicates without second order quantifiers and type variables which range over types constructed with elementary type constructors (cartesian product, arrow, and list) from a few primitive types. These variables are treated as universally quantified. This simple second order logic, however, enables a sort of higher order programming as demonstrated in this paper.

For the program extraction, **qpc**-realizability interpretation is defined. It looks like the formulation of second order typed lambda calculus for the quantified predicate variables, and Kreisel-Troelstra realizability for the quantified type variables. This distinction means that type information should be eliminated in the program extraction while a predicate is interpreted as containing computational meaning to be extracted. To assure the soundness theorem of **qpc**-realizability, a class of expressions called *scheme* is introduced in addition to a variant of untyped lambda expressions. Scheme is a comparative notion to second order lambda terms, but a scheme expression is not regarded as a program. The structure of schemes extracted from the second order proofs is simple because the form of the second order formulas is restricted. Schemes can be handled rather easily by using proof normalization, and in many cases second order proofs can be translated to first order proofs from

which lambda terms can be extracted.
The system presented in this paper does not have the notion of *non-informative proposition*([10]) except Harrop formulas [14] from which no computational contents should be extracted. Therefore, the extracted sorting programs from the proofs of the example specification need redundant codes, which justifies that the relation P is actually a total relation, as inputs. This problem will be overcome by introducing two kinds of constants *prop* (non-informative proposition) and *spec* (informative proposition) as in [10], and redefine **qpc**-realizability as follows:

- $e\ \mathbf{qpc}\ \forall^2\alpha \in type_1.A \stackrel{\text{def}}{=} \forall^2\alpha \in type_1.(e\ \mathbf{qpc}\ A)$
- $e\ \mathbf{qpc}\ \forall^2 P \in \sigma \to prop.A \stackrel{\text{def}}{=} \forall^2 P \in \sigma \to prop.(e\ \mathbf{qpc}\ A)$
- $e\ \mathbf{qpc}\ \forall^2 P \in \sigma \to spec.A \stackrel{\text{def}}{=} \forall^2 P \in \sigma \to spec.(ap(e, P)\ \mathbf{qpc}\ A)$

Acknowledgment

The main part of this work was elaborated when the author was in ICOT. The author is indebted to Dr. Hasegawa, chief of the first laboratory of ICOT, for the opportunity of the research.

References

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

[2] Y. Takayama, "Writing Programs as QJ-proofs and Compiling into PROLOG Programs", *Proceedings of Fourth IEEE Symposium on Logic Programming*, 1987

[3] Y. Takayama, "QPC: QJ-based proof compiler, – simple examples and analysis", *Proceedings of European Symposium on Programming 88*, LNCS 300, 1988

[4] T. Coquand and G. Huet, "The Calculus of Constructions", Information and Computation 76, 1988

[5] J.C. Reynolds, "Towards theory of type structure", *Proceedings of Programming Symposium*, LNCS 19, 1974

[6] J. Y. Girard, "System F of Variable Types – fifteen years later", Theoretical Computer Science 45, 1986

[7] M. Parigot, "Programming with Proofs: A Second-Order Type Theory", *Proceedings of European Symposium on Programming 88*, LNCS 300, 1988

[8] C. Böhm and A. Berarducci, "Automatic synthesis of typed λ-programs on term algebras", Theoretical Computer Science 39, 1985

[9] 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

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

[11] L. Dames and R. Milner, "*Principal type-schemas for functional programming*", Edinburgh University, 1982

[12] G. Kreisel and A. S. Troelstra, "Formal systems for some branches of intuitionistic analysis", Annals of Math. Logic 1, pp229-387, 1979

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

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

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

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

[17] 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

[18] 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

[19] M. Tatsuta, "Program Synthesis Using Realizability", Master's thesis, University of Tokyo, 1987