TR-624

Coinductive Constructive Programming for Concurrent Systems

by
Y. Takayama (Oki)

February, 1991

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

(03)3456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

Coinductive Constructive Programming for Concurrent Systems *

Yukihide Takayama
Electric System Laboratory, OKI Electric Industry Co., Ltd.
11-22 Sibaura 4 chome, Minato-ku, Tokyo 108, Japan
takayama@okilab.oki.co.jp, takayama@icot.or.jp

November 30, 1990

Abstract

This paper investigates problems that emerge when the paradigm of constructive programming (proofs-as-programs) is applied to concurrent programming, in particular Kahn-MacQueen-Keller style of stream based concurrent programming with nondeterminacy. The main interest in this paper is how streams are treated formally and how recursive programs on streams should be described as proof procedures. We regard streams as Brouwer's choice sequence, and introduce a natural deduction style of coinduction rule based on the categorical treatment of streams by T. Hagino to define recursive programs on streams. Also, several examples including Eratosthenes' sieve algorithm and a nondeterministic stream merger are presented.

1 Introduction

Constructive logic gives a formal method of developing programs. Suppose, for example, the following formula:

$$\forall x : D_1.\exists y : D_2.\ A(x, y)$$

This is regarded as a specification of a function, f, whose domain is D_1 and the codomain is D_2 satisfying the input-output relation, $A(x, y)$, that is, $\forall x : D_1.\ A(x, f(x))$ holds. If a constructive proof of the formula is given, the function, f, is extracted from the proof with q realizability interpretation in constructive logic [4, 29] or with Curry-Howard correspondence of types and formulas [12]. This realizes Bishop's idea of *mathematical*

*This work was supported by Institute for New Generation Computer Technology as a joint research project on paralell programming environment.

language as high level programming language [4, 5], and will be referred to as *constructive programming* in the following. Constructive programming has been studied by a lot of researchers as in [3, 8, 11, 17, 18, 20, 24, 26, 28]. One of the advantages of constructive programming is that control mechanism in programs can be described as logical reasoning. In particular, recursive call programs can be written as proofs in induction. For the recursive programming on a list type, L_σ, the structural induction rule is used:

$$\frac{A(nil) \qquad \overset{[x \neq nil,\, A(tl(x))]}{A(x)}}{\forall x : L_\sigma.\ A(x)}(st\text{-}ind)$$

The proof of $A(nil)$ defines the termination condition of the recursive program, and the occurrences of the induction hypothesis, $A(tl(x))$, correspond to the recursive calls of the program on $tl(x)$. Then, a proof in structural induction on lists

$$\frac{\Sigma_1 \quad A(nil) \qquad \overset{[x \neq nil,\, A(tl(x))]}{\Sigma_2 \quad A(x)}}{\forall x : L_\sigma.\ A(x)}(st\ ind)$$

defines the following recursive call program:

$$F(x) \equiv if\ x = nil\ then\ Code_1\ else\ Code_2$$

where $Code_1$ is the program defined by the subproof, $(\Sigma_1/A(nil))$, and $Code_2$, in which F may occur, is the program defined by the subproof, $(\Sigma_2/A(x))$.

On the other hand, the notion of streams is one of the key ideas in extending constructive programming to concurrent programming. Then, the problem is whether the structural induction can be extended to stream types or a comparative rule on streams can be formulated. The naive extension of $(st\text{-}ind)$ to regarding streams as infinite lists

$$\frac{\overset{[A(tl(x))]}{A(x)}}{\forall x : I_\sigma.\ A(x)}(SI)$$

where I_σ is a type of streams over σ, does not work well because, for example, the following wrong theorem can be proved:

Let nat be the natural number type and $elem(n, x)$ *denote the* n*th element of* x*. Then,*
$\forall x : I_{nat}.\ B(x)$ *where* $B(x) \overset{\text{def}}{=} \exists n : nat.\ elem(n, x) = 100$.
Proof: By (SI) *on* $x : I_{nat}$*. Assume* $B(tl(x))$*. Then, there exists a natural number* k *such that* $elem(k, tl(x)) = elem(k+1, x) = 100$*. Then* $B(x)$.∎

This proof would correspond to the following meaningless program:

$$foo = \lambda x.\ foo(tl(x))$$

Therefore, for the treatment of streams, drastically different idea is needed.
There have been several works for the formal treatment of streams. T. Hagino [10] gave a uniform categorical framework of both least and greatest fixed point types, and showed that a stream (infinite list) type can be defined as a greatest fixed point type while a (finite) list type is defined as a least fixed point type. P. Aczel and N. Mendler [1, 2] also gave a characterization of stream types in a categorical setting. Both works are along the similar line to the work by M. B. Smyth and G. Plotkin [22] and relevance to logic is not pursued. N. Mendler et al [15] introduced lazy types to the Nuprl proof development system [8], but Curry-Howard correspondence does not work well for the lazy types. That is, lazy types is not regarded as formulas, and recursive programming on streams with lazy types cannot be described as logical reasoning as in the sense of the structural induction for the recursive programming on finite lists. P. Dybjer and H. P. Sander [9] gave a verification system for concurrent systems based on the idea of constructive programming. Their system uses a domain theoretic axiom, a greatest fixedpoint induction rule (coinduction) introduced by D. Park [19], for the verification of recursive call programs on streams. Although streams are handled in a framework of logic, formalizing the coinduction in the style of natural deduction remains to be an open problem.

This paper extends constructive programming to Kahn-MacQueen-Keller style model of stream based concurrent programming with nondeterminacy [13, 14]. An computation agent of the model is regarded as a stream transformer as in [9], so that we assume a recursive program on streams to be a stream transformer that takes several streams as input and calculates a stream as output. The basic idea of defining a stream transformer is Burge's stream generator function [7] which was reformulated by T. Hagino as a categorical combinator, P, in the following commutative diagram of coinduction:

$$\begin{array}{ccccc} \sigma & \xleftarrow{hd} & I_\sigma & \xrightarrow{tl} & I_\sigma \\ \| & & \uparrow{\scriptstyle PMN} & & \uparrow{\scriptstyle PMN} \\ \sigma & \xleftarrow{M} & \tau & \xrightarrow{N} & \tau \end{array}$$

where I_σ is a stream type on a type σ, τ is an arbitrary type, and $M : \tau \to \sigma$ and $N : \tau \to \tau$ are arbitrary morphisms. P is interpreted as the following lambda term:

$$P \equiv \lambda M^{\tau\to\sigma}.\lambda N^{\tau\to\tau}.\lambda x^{\tau}.\ ((M\ x) :: (((P\ M)\ N)\ (N\ x)))$$

where (::) is the infinite list constructor. If τ is a cartesian product of stream types, $I_{\tau_1} \times \cdots \times I_{\tau_n}$, and the morphisms, M and N, are defined as constructive proofs, then the stream transformer, PMN, can be defined with a suitable rule of inference representing the combinator P. The rule is to prove the specification of the stream transformer in the form of $\forall x_1 : I_{\tau_1}.\cdots\forall x_n : I_\tau.\exists y : I_\sigma.\ A(x_1,\cdots,x_n,y)$. If the class of the formula, $A(x_1,\cdots,x_n,y)$, is suitably restricted, this idea works well and we obtain a natural deduction style of coinduction.

In our coinduction rule, streams are regarded as Brouwer's choice sequences: A stream type, I_σ, consists both with lawless and lawlike sequences. This means that our system takes the sequences such as keyboard input into account. This maintains the idea of concurrent system as open system [16]. From aesthetic point of view, the formulation of streams as choice sequences is sufficient, but we also define computational streams, which are represented by non-terminating recursive call functions, and computational stream types for practical reason. A computational stream type is an extension of lazy types with nondeterminacy, and the class of computational streams can be effectively embedded into the class of choice sequences.
A crucial design branch of the formal system resides in the treatment of the program such as the stream filter program whose output stream may be empty. One approach is to regard finite sequences, including empty sequences, as streams, but we introduce a notion of complete stream type in which finite sequences are also regarded as virtually infinite. That is to keep the uniformity of the formulation.
Nondeterminacy is simulated by a special term called the coin flipper at the programming language level, but can be represented rather implicitly at the proof level because of the inherent nondeterminacy in the proof normalization with regard to disjunction symbol.

The construction of the paper is as follows. Section 2 gives the basic idea for treating streams and stream transformers. A coinduction rule, $(Coind)$, in natural deduction style formalism is defined here. The rest of the formalization of the whole system is presented in section 3. A calculus called non-deterministic λ-calculus is defined as the program language. The rules for stream types, computational stream types, and effective embedding of computational streams into the class of choice sequences is established. Finally, a realizability interpretation of the formal system, which gives the program extraction algorithm from proofs, is defined. Several examples of concurrent programs are given to demonstrate the expressive power of the formal system, in particular $(Coind)$ rule in section 4. Comparison with other works and discussion are presented in the final section.

Notational preliminary: We assume first order intuitionistic logic. Equalities of terms, typing relation $(M : \sigma)$, $\bot$ (absurdity), and $\top$ (true) are atomic formulas. Logical connectives and quantifiers are $\&$, $\vee$, $\Rightarrow$, $\neg$, $\forall$ and $\exists$. $x, y, \cdots, X, Y, \cdots$ denote individual variables, and sequences of variables are denoted as $\overline{x}$ and $\overline{X}$. $M_x[N]$ denotes substitution of N to the variable, x, occurring freely in M. $M_{\overline{x}}[N]$ denotes simultaneous substitution. $FV(M)$ is the set of free variables in M. $\equiv$ and $\stackrel{\text{def}}{=}$ are definitional equalities, and $A \Leftrightarrow B$ is an abbreviation of $(A \Rightarrow B) \,\&\, (B \Rightarrow A)$.

2 Defining Stream Transformers

The model of concurrent systems, in this paper, is as follows: A concurrent system consists of computation agents linked with streams. A computation agent interacts with other computation agents or the outside world through the streams. The configuration of computation agents in a concurrent system is basically static, but in a exceptional case, which will be explained later, new agents may be created by an already existing agents.

A computation agent is regarded as a transformer (stream transformer) of input streams to an output stream, and it is specified by the following type of formula:

$$\forall \overline{x} : I_{\sigma_1,\cdots,\sigma_n}.\ \exists y : I_\tau.\ A(\overline{x}, y)$$

where $I_{\sigma_1,\cdots,\sigma_n}$ is an abbreviation of $I_{\sigma_1} \times \cdots \times I_{\sigma_n}$, $\overline{x}$ and y are input and output streams, and $A(\overline{x}, y)$ is the relation definition of input and output streams. Here, the number of output stream is restricted to one, but that is only for the convenience of the discussion. The combination of two computation agents, $\forall x.\exists y.\ A(x,y)$ and $\forall p.\exists q.\ B(p,q)$, is represented by the following proof procedure:

$$\dfrac{\dfrac{\dfrac{[X]^{(1)} \quad \forall x.\exists y.\ A(x,y)}{\exists y.\ A(X,y)}(\forall E) \qquad \Pi}{\exists Y.\exists \alpha.\ A(X,\alpha)\ \&\ B(\alpha,Y)}(\exists E)^{(2)}}{\forall X.\exists Y.\exists \alpha.\ A(X,\alpha)\ \&\ B(\alpha,Y)}(\forall I)^{(1)}$$

where $\Pi \stackrel{\text{def}}{=}$

$$\dfrac{\dfrac{|y|^{(2)} \quad \forall p.\exists q.\ B(p,q)}{\exists q.\ B(y,q)}(\forall E) \qquad \dfrac{[q]^{(3)} \quad \dfrac{|y|^{(2)} \quad \dfrac{[A(X,y)]^{(2)}\ [B(y,q)]^{(3)}}{A(X,y)\ \&\ B(y,q)}(\&\ I)}{\exists \alpha.\ A(X,\alpha)\ \&\ B(\alpha,q)}(\exists I)}{\exists Y.\exists \alpha.\ A(X,\alpha)\ \&\ B(\alpha,Y)}(\exists I)}{\exists Y.\exists \alpha.\ A(X,\alpha)\ \&\ B(\alpha,Y)}(\exists E)^{(3)}$$

Thus, a concurrent system is also specified by $\forall x.\exists y.\ A(x,y)$ type formula. X and Y are input and output streams of the whole concurrent system, and α is an internal stream. The internal streams of a concurrent system can be hidden by using the checked existential quantifier, $\stackrel{\flat}{\exists}$, introduced by the author and S. Hayashi [27] or Hayashi's $\diamondsuit$ operator [11]. In the following, we focus on the problem of how to define a computation agent (stream transformer) as a constructive proof.

2.1 Categorical Characterization of Stream Transformers

The behavior of a stream transformer with input stream, X, and an output stream, Y, can be modeled as follows: it is an infinite sequence of two operations, M and N, occurring alternatively in it.

$$M \longleftrightarrow \left\{ \begin{array}{l} \text{Fetch initial segment, } X_0, \text{ of the input stream, } X, \text{ to} \\ \text{generate the first element of the output stream. (Fig.1)} \end{array} \right\}$$

$$N \longleftrightarrow \left\{ \begin{array}{l} \text{Prepare for fetching next elements from the input} \\ \text{stream interleaving, if necessary, other stream} \\ \text{transformer between the original input stream and} \\ \text{the input port. (Fig. 2)} \end{array} \right\}$$

This informal process model is well formalized by using Hagino's categorical formulation of fixed point types [10] which we will refer to as the categorical version of coinduction on streams in the following. A stream transformer, $tran : I_\tau \to I_\sigma$, is defined as the morphism, PMN, in the following commutative diagram:

$$\begin{array}{ccccc} \sigma & \xleftarrow{hd} & I_\sigma & \xrightarrow{tl} & I_\sigma \\ \| & & \uparrow{\scriptstyle PMN} & & \uparrow{\scriptstyle PMN} \\ \sigma & \xleftarrow{M} & I_\tau & \xrightarrow{N} & I_\tau \end{array}$$

A stream type is accompanied with two destructers, hd (head) and tl (tail). If the morphisms $M : I_\tau \to \sigma$ and $N : I_\tau \to I_\tau$ are given, the morphism $PMN : I_\tau \to I_\sigma$ is determined. P is a categorical combinator which can be interpreted in typed lambda calculus as follows:

$$P \equiv \lambda M^{I_\tau \to \sigma}.\lambda N^{I_\tau \to I_\tau}.\lambda x^{I_\tau}.\ ((M\ x) :: (((P\ M)\ N)\ (N\ x)))$$

This suggests that, in order to define a stream transformer as a constructive proof, one must at least give a way to defined M, N, and P as proof procedures.

2.2 Choice Sequences

Following [16], a concurrent system is an open system that interacts with the outside world. So that an input stream of a computation agent should not be restricted to lawlike sequence but should allow lawless sequences whose elements are defined, for example, by the keyboard inputs or casts of an (abstract) die. This feature is captured by Brouwer's theory of choice sequences [30]. Also, the theory provides us with the meaning of quantification over streams. Following the suggestion obtained in the previous subsection, M and N would be defined by constructive proofs of $\forall X : I_\sigma.\exists y : \tau.\ M(x, y)$ and $\forall X : I_\sigma.\exists Y : I_\sigma.\ N(x, y)$. In the following, stream variables will be denoted in uppercase letters: X, Y, $\cdots$. The principle of open data, which informally states that for independent lawless sequences any property which can be asserted must depend on initial segments of these sequences only, gives the meaning of the first type of quantification, $\forall X.\exists y$. That is, for arbitrary stream, X, there is a suitable initial finite segment, X_0, of X such that $\exists y : \tau.\ M(X_0, y)$ holds. This meets the process model of the operation M in the previous subsection. A generalized version of the principle, which is called function continuity, gives the meaning of the second type of quantification, $\forall X.\exists Y$. Assume the case of natural number streams (total functions between natural number types). The function continuity is as follows:

$$\forall X.\exists Y.\ A(X, Y) \Rightarrow \exists f : K.\ \forall X.\ A(X, f|X)$$

where $f|X = Y \stackrel{\text{def}}{=} \forall x.\ f(x :: X) = Y(x)$ and K is the class of neighborhood functions which take initial finite segment of the input sequences and return the values. This means that every element of Y is determined with a suitable initial finite segment of X, and this

also meets the process model of a stream transformer which successively takes elements of input streams generating gradually the elements of the output streams. We, then, regard streams as choice sequences and borrow the open data principle for the semantical explanation of quantification over streams.

2.3 A Problem of Infinity

Before giving the rule of inference for defining stream transformers, a little more observation of stream programming is needed. Assume a filer program on natural number streams:

$$flt_a \equiv \lambda X^{I_{nat}}.\ if\ (a|hd(X))\ then\ flt_a(tl(X))\ else\ (hd(X) :: flt_a(tl(X)))$$

where $(a|hd(X))$ is true when $hd(X)$ can be divided by a (a natural number). flt_a can also be defined with the Hagino's combinator, P, as follows:

PMN
where $M \equiv \lambda X^{I_{nat}}.\ if\ (a|hd(X))\ then\ M(tl(X))\ else\ hd(X)$
and $N \equiv \lambda X^{I_{nat}}.\ if\ (a|hd(X))\ then\ N(tl(X))\ else\ tl(X)$

For example, $flt_5((5 :: 5 :: 5 :: 5 :: \cdots))$ is empty sequence. However, it should be regarded as undefined for the following reason. If flt_5 is $P\ M\ N$ defined above, the execution of $M(5 :: 5 :: 5 :: 5 :: \cdots)$ does not terminate, and cannot even decide the value is to be empty from a suitable initial segment of $(5 :: 5 :: 5 :: 5 :: \cdots)$ unless the M know the justification that the elements of the input stream is always 5. One may define the subdomain of the input streams on which M always finds the values of output stream as has been done for recursive programs on finite lists with CIG mechanism of PX [11]. That may be possible when the streams are restricted to lawlike sequences. However, because we allow lawless sequences, the domains which ensure infinite output streams are generally undecidable. This contradicts the open data principle explained in the previous subsection. Therefore, we introduce the notion of complete stream for a uniform treatment of outputs of stream transformers.

Def. 1: Complete types
For any type, σ, $\sigma_\perp$ denotes a type σ together with the bottom element $\perp_\sigma$ (often denoted just $\perp$) and it is called a *complete type*. A type which is not complete is called an *ordinary type*.

Def. 2: Complete stream types
A stream type, I_σ is called complete when σ is a complete type.

A complete type contains a bottom element, $\perp$, so that $(\perp :: \perp :: \cdots)$ and $(1 :: 2 :: \perp :: \perp :: 3 :: \cdots)$ are elements of the generalized stream type, $I_{nat_\perp}$. It is also possible to speak of complete stream types which is necessary in handling a stream of streams.

Then, a specification of flt_a can be written as follows:

$$\forall X : I_{nat}.\exists Y : I_{nat_\perp}.\forall n : nat.$$
$$(((a|X(n)) \,\&\, Y(n) = \perp) \vee (\neg(a|X(n)) \,\&\, Y(n) = X(n)))$$

2.4 The Coinduction Rule

Based on the observations in the previous sections, we introduce a coinduction rule for defining stream transformers. The rule is formulated in natural deduction style, but the formula, A, in the specification of a stream transformer, $\forall X.\exists Y.\ A(X, Y)$, is restricted. In spite of the restriction, the rule can handle large class of specifications of stream transformers as will be demonstrated in section 4.
The rule is as follows:

$$\frac{\begin{array}{l}(a)\ \forall \overline{X} : I_{\sigma_1,\cdots,\sigma_n}.\exists a : \tau.\ M(\overline{X}, a)\\ (b)\ \forall \overline{X} : I_{\sigma_1,\cdots,\sigma_n}.\forall a : \tau.\ \forall S : I_\tau.\ (M(\overline{X}, a) \Rightarrow A(0, \overline{X}, (a :: S)))\\ (c)\ \exists f : I_{\sigma_1,\cdots,\sigma_n} \to I_{\sigma_1,\cdots,\sigma_n}.\ \forall \overline{X} : I_{\sigma_1,\cdots,\sigma_n}.\forall Y : I_\tau.\forall n : nat.\\ \qquad\qquad (A(n, f(\overline{X}), tl(Y)) \Rightarrow A(n+1, \overline{X}, Y))\end{array}}{\forall \overline{X} : I_{\sigma_1,\cdots,\sigma_n}.\exists Y : I_\tau.\forall n : nat.\ A(n, \overline{X}, Y)}\ (Coind)$$

where M is a suitable predicate and $A(n, \overline{X}, Y)$ is a rank 0 formula defined below:

Def. 3: Rank 0 formulas [11]

1. If M is a term and σ is a type, then $M : \sigma$ is rank 0;
2. If M and N is rank 0 then, $M = N$ is rank 0;
3. $\top$ and $\perp$ is rank 0;
4. If A is a formula and G, G_1 and G_2 are rank 0, then $G_1 \,\&\, G_1$, $A \Rightarrow G$, $\neg A$, $\Diamond A$, and $\forall x : \sigma.\ G$ are rank 0.

The rules for rank 0 formulas are as follows:

($\Diamond$1) $\qquad \neg\neg A \Leftrightarrow \Diamond A$

($\Diamond$2) $\qquad \forall x : \sigma.\ \Diamond A \to \Diamond\forall x : \sigma.\ A$

($\Diamond$3) $\qquad A \Leftrightarrow \Diamond A \quad$ (if A is a rank 0 formula)

Theorem 1: (S. Hayashi)
Let A be a formula. If $\Gamma, (DNE) \vdash A$ is proved, then $\Gamma \vdash \Diamond A$ can be proved, where (DNE) is the double negation elimination rule.

This theorem means that $\Diamond A$ can be proved by proving A classically.
This restriction comes from a technical reason related to realizability interpretation, and does mean that no computational meaning is extracted from $A(n, \overline{X}, Y)$ part. This is actually rather convenient because, in most cases, we are only interested in the value of

Y. We can also use the logic of checked existential quantifier, $\tilde{\exists}$, for rank 0 formulas developed in [27], but that is besides the point of the paper and rank 0 formulas are much easier to handle in discussing $(Coind)$.

The intuitive meaning of $(Coind)$ is as follows. This rule represents the following categorical version of coinduction:

$$\begin{array}{ccccc} \sigma & \xleftarrow{hd} & I_\sigma & \xrightarrow{tl} & I_\sigma \\ \| & & \uparrow P & & \uparrow P \\ \sigma & \xleftarrow{f_M} & I_{\sigma_1,\cdots,\sigma_n} & \xrightarrow{f_N} & I_{\sigma_1,\cdots,\sigma_n} \end{array}$$

f_M and f_N are the programs corresponding to the proofs of (a) and (c). P is the stream transformer corresponding to the whole proof in $(Coind)$. (c) together with (b) intuitively means the following: For $\overline{X} : I_{\sigma_1,\cdots,\sigma_n}$ and $Y(= P(\overline{X})) : I_\tau$, let us call a pair, $(f^n(\overline{X}), tl^n(Y))$, the nth f-descendant of $(\overline{X}, Y)$. Then, for arbitrary $n : nat$, $A(n, \overline{X}, Y)$ speaks about nth f-descendant of $(\overline{X}, Y)$, and $A(n, f(\overline{X}), tl(Y))$ actually speaks about $n+1$th f-descendant of $(\overline{X}, Y)$.
If f_N (the value of $\exists f$ in (c)) is a stream transformer program, this means that the process (stream transformer) defined by $(Coind)$ generates another processes dynamically.

3 The Formal System

This section presents the rest of the formal account of our system. First of all, as a programming language non-deterministic λ-calculus is defined. The calculus has a special constant called the *coin flipper*, to simulate non-determinism, and computational stream types (lazy types). The calculus is almost similar to that developed for the Nuprl [8] except the coin flipper. Secondly, the rules of inference including rules for streams other than those for computational streams are given. The canonical embedding of computational streams to the class of choice sequences is also given. Finally, the realizability interpretation of the system is defined, and this gives the program extraction algorithm from proofs.

3.1 Non-deterministic λ-calculus

Non-deterministic λ-calculus is a typed concurrent calculus based on parallel reduction. The core part is almost similar to that given in [24, 25, 26]. It has natural numbers, booleans (T and F), and L and R as constants. Individual variables, lambda-abstractions, application ($M(N)$ or $ap(M, N)$ where M and N are terms), sequences of terms ($(M_1, ..., M_n)$ where M_is are terms), *if-then-else*, and a fixed point operator (μ) are used as terms and program constructs. Parallel Reduction rules for terms are defined as expected, and if a term, M, is reducible to a term, N, then M and N are regarded as equal. Also, several primitive functions are provided for arithmetic operations and

for the handling of sequences of terms such as projection of elements ($\mathbf{p}_i(\overline{(x)}) = x_i$) or subsequences from a sequence of terms. The type structure of the calculus is almost that of simply typed λ-calculi. *nat* (natural number type), *bool* (boolean types), and **2** (type of L and R) are primitive types and $\times$ (cartesian product) and $\rightarrow$ (arrow) are used as type constructors. The type inference rules for this fragment of the calculus are defined as expected. In addition to them, stream types and a special term called *coin flipper* is introduced to describe concurrent computation of streams. These additional notions will be explained in the sequel.

3.1.1 Coin Flipper

The coin flipper is a device for describing non-determinacy. It is a term, $\bullet$, whose computational meaning is given by the following reduction rule:

$$\bullet \triangleright L \text{ or } R$$

That is, $\bullet$ returns L or R in non-deterministic way when it is executed. This is like flipping a coin, and need not always be executed by reduction mechanism. Therefore, $\bullet$ is like *oracles* in recursive function theory.

Example 1: (A non-deterministic program)
Let $M_x \stackrel{\text{def}}{=} \mathit{if}\ x = 1\ \mathit{then}\ 2\ \mathit{else}\ 3$ and $N_x \stackrel{\text{def}}{=} \mathit{if}\ x = 1\ \mathit{then}\ 3\ \mathit{else}\ 4$. Then, $\lambda x.\ \mathit{if}\ \bullet = L\ \mathit{then}\ M_x\ \mathit{else}\ N_x$ is a function that returns 2 or 3 when 1 is given as a input and returns 3 or 4 otherwise.

$\bullet$ is regarded as an element of $\mathbf{2}^+$, a super type of **2**. The element of **2** have been used to describe the decision procedure of *if-then-else* programs in the program extraction from constructive proofs in [24], [25] and [26] as $\mathit{if}\ T = L\ \mathit{then}\ M\ \mathit{else}\ N$. Non-determinism arises when T is replaced by $\bullet$. The intentional semantics of $\bullet$ is *undefined*. The type $\mathbf{2}^+$ will be used instead of **2** in this paper with the following typing rules:

$$L : \mathbf{2}^+ \qquad R : \mathbf{2}^+ \qquad \bullet : \mathbf{2}^+$$

3.1.2 Computational Streams

As is well known in functional programming, a stream can be represented by a non-terminating recursive call function. We will call thus represented streams *computational streams*. Instead of μ, the symbol ν will be used as fixed point operator to denote non-terminating recursive call functions representing streams. This syntactic convention is to distinguish terms which should be handled by lazy evaluation and terms which need not. The computation rule for ν-terms is the same as that of μ-terms:

$$\nu z.\ M \triangleright M_z[\nu z.\ M]$$

Example 2: A stream, $(0 :: 0 :: 0 :: \ldots)$, can be defined as $\nu z.\ (0 :: z)$. Precisely, for any natural number, n, this stream is represented as $(\underbrace{0 :: 0 :: \ldots :: 0}_{n} :: \nu z.\ (0 :: z))$.

Example 3: (Stream of natural numbers)
A stream of natural numbers, $(0 :: 1 :: 2 :: 3 :: \ldots)$, can be defined as $nst(0)$ where $nst \stackrel{\text{def}}{=} \nu z.\ \lambda n.\ (n :: z(n+1))$. Note that $nst(1)$, $nst(2)$, $\cdots$, are also streams of natural numbers starting from 1,2, ... respectively.

3.1.3 Computational Stream Types

The type of computational streams over a type σ will be denoted C_σ. First of all, as pointed out in [1] the domain, X, of streams over a domain A can be regarded as the (largest) fixed point of the domain map $\Phi : X \to A \times X$, so that C_σ can be characterized by the following type equality:

$$C_\sigma = \sigma \times C_\sigma \qquad (STTE)$$

A computational stream is regarded as an element of infinite cartesian product of σ, so that one of the typing rules is:

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash S : C_\sigma}{\Gamma \vdash (M :: S) : C_\sigma}(ST1)$$

We confuse the meaning of the infinite list constructor, $(::)$, and will use this as a cartesian product constructor. Note that ST1 is obtained by a straightforward extension of the cartesian product rule

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash (M :: N) : \sigma \times \tau}$$

and STTE.
ST1 is not enough because type checking by the rule will not terminates as a stream is an infinite object. The following rule is also needed.

$$\frac{\Gamma, z : \sigma \vdash M : \sigma}{\Gamma, z : \sigma \vdash \nu z.\ M : \sigma}(ST2)$$

Example 4: Type checking of the natural number stream, $(\nu z.\lambda n.\ (n :: z(n+1)))(0)$, goes as follows:

$$\frac{\Gamma \vdash 0 : nat \quad \Gamma \vdash \nu z.\lambda n.\ (n :: z(n+1)) : nat \to C_{nat}\ \Sigma_0}{\Gamma \vdash (\nu z.\lambda n.\ (n :: z(n+1)))(0) : C_{nat}}$$

where Σ_0 is as follows:

$$\dfrac{\dfrac{\dfrac{\Sigma_1}{\Delta \vdash z : nat \to C_{nat} \quad \Delta \vdash n+1 : nat} \quad \Delta \vdash n : nat}{\Delta \vdash z(n+1) : C_{nat}}(ST1)}{\dfrac{\Delta \vdash (n :: z(n+1)) : C_{nat}}{\Gamma, z : nat \to C_{nat} \vdash \lambda n.\,(n :: z(n+1)) : nat \to C_{nat}}(ST2)}}{\Gamma \vdash \nu z.\lambda n.\,(n :: z(n+1)) : nat \to C_{nat}}$$

where $\Delta \stackrel{\text{def}}{=} \Gamma, z : nat \to C_{nat}, n : nat$

Equality of stream can be characterized in two ways. One is by viewing streams as elements of an infinite cartesian product type, and this entails extensional equality because there is no normal forms of the streams. Another one is by viewing streams as recursive call functions, and this can be handled by intentional equality.

$$\dfrac{\Gamma, n : nat, tl^n(S) = tl^n(T)\ in\ C_\sigma \vdash S = T\ in\ \sigma^n \times C_\sigma}{\Gamma \vdash S = T\ in\ C_\sigma}(EXTE)$$

where $tl^n(T)$ means that $tl$is applied to T for n times.

$$\dfrac{\Gamma \vdash M_z \to N_w[z]\ in\ \sigma}{\Gamma \vdash \nu z.\ M_z \to \nu w.\ N_w \in \sigma}(INTE)$$

Example 5: $\nu z.\ (ap(\lambda x.\ 5, 1) :: z)$ and $\nu w.\ (5 :: w)$ are intentionally equal.

Example 6: $(\nu z.\lambda n.\ (5 :: z(n+1)))(0)$ and $\nu z.\ (5 :: z)$ both represent the same stream, $(5 :: 5 :: 5 :: \ldots)$, and they are extensionally equal programs, but not equal intentionally. Let $F \stackrel{\text{def}}{=} \nu z.\lambda n.\ (5 :: z(n+1))$ and $T \stackrel{\text{def}}{=} F(0)$. Also let $S \stackrel{\text{def}}{=} \nu z.\ (5 :: z)$. Then, $T = S\ in\ C_{nat}$ can be proved by proving $\forall k : nat.\ (tl^k(T) = tl^k(S)\ in\ C_{nat} \Rightarrow S = T\ in\ \sigma^k \times C_{nat})$ by mathematical induction on $k : nat$. Base case $(k = 0)$ is obvious. Then, assume $tl^{k-1}(T) = tl^{k-1}(S)\ in\ C_{nat} \Rightarrow S = T\ in\ \sigma^{k-1} \times C_{nat}$ as the induction hypothesis. $\forall l : nat.\ tl^l(T) = F(l)\ in\ C_{nat}$ is proved by mathematical induction on $l : nat$. Also, $\forall l : nat.\ tl^l(S) = S\ in\ C_{nat}$ is proved by mathematical induction on $l : nat$. Assume that $tl^k(T) = S\ in\ C_{nat}$ holds. $F(k-1) = (5 :: F(k))\ in\ C_{nat}$ can be proved, so that by the assumption and STTE, $F(k-1) = (5 :: S)\ in\ C_{nat}$. Also, by $(STTE)$ and $\forall l.\ tl^l(S) = S\ in\ C_{nat}$, $(5 :: S) = tl(S) = tl^{k-1}(S)\ in\ C_{nat}$. Therefore, by the induction hypothesis and $\forall l.\ tl^l(T) = F(l)$, $S = T\ in\ \sigma^{k-1} \times C_{nat}$. So that by STTE, $S = T\ in\ \sigma^k \times C_{nat}$. Then, $S = T\ in\ C_{nat}$ by EXTE.

3.2 Rules for Streams

$$\dfrac{M : nat \to \sigma}{M : I_\sigma}$$

This means that any (total) function on nat typable in the non deterministic λ-calculus is regarded as a stream. As the calculus has the coin flipper term, some of the lawless

sequences can be represented. A stream type, I_σ, is regarded as containing other lawless sequences which cannot be represented in the calculus. Therefore, we do not assume the rule such as follows:

$$\frac{M : I_\sigma}{M : nat \to \sigma}(Wrong\ Rule)$$

This is the openness of our system. For example, if we are to handle the keyboard input, and it can be coded as a sequence, the following rule can be added:

$$\frac{M\ is\ a\ keyboard\ input}{M : nat \rightsquigarrow \sigma} \qquad \frac{M : nat \rightsquigarrow \sigma}{M : I_\sigma}$$

where $nat \rightsquigarrow \sigma$ denotes a type of total functions on nat which cannot be typed in the non-deterministic λ-calculus.

$$\frac{\forall n : nat.\exists x : \sigma.\ A(n,x)}{\exists Y : I_\sigma.\forall n : nat.\ A(n,Y(n))}(ST)$$

The equality between streams is extensional. That is

$$\frac{X : I_\sigma \quad Y : I_\sigma \quad \forall n : nat.\ X(n) = Y(n)}{X = Y\ in\ I_\sigma}$$

The following rule is used for justifying (*Coind*).

$$\frac{F : I_{\sigma_1,\cdots,\sigma_k} \to I_{\sigma_1,\cdots,\sigma_k} \quad \forall \overline{X} : I_{\sigma_1,\cdots,\sigma_k}.\forall n : nat.\ A(0,F(\overline{X})) \Rightarrow A(n+1,\overline{X})}{\forall \overline{X} : I_{\sigma_1,\cdots,\sigma_k}.\forall n : nat.\ A(0,F^n(\overline{X})) \Rightarrow A(n,\overline{X})}(R)$$

where $A(n,\overline{X})$ is a rank 0 formula.

There are other rules for handling elements of streams:

$$\frac{X : I_\sigma}{hd(X) : \sigma} \qquad \frac{X : I_\sigma \quad n : nat}{tl^n(X) : I_\sigma} \qquad \frac{X : I_\sigma \quad n : nat}{elem(n,X) = hd(tl^n(X))}$$

$$\frac{X : I_\sigma \quad n : nat}{X(n) = hd(tl^n(X))} \qquad \frac{X : I_\sigma \quad n : nat}{tl^n(X) = (hd(tl^n(X)) :: tl^{n+1}(X))}$$

3.3 Other rules of inference

3.3.1 Logical Rules

The rules for logical connectives and quantifiers are those of first order intuitionistic natural deduction with mathematical induction. See [23, 24, 26] for the complete account of the logical rules.

3.3.2 Rules for Nondeterminacy

$$\bullet = L \vee \bullet = R$$

$$\frac{A \qquad A}{A}(NonDet)$$

This is actually a derived rule:

$$\dfrac{\dfrac{\dfrac{\top}{\top \vee \top}(\vee I) \qquad [\top] \quad [\top]}{A} \qquad A \qquad A}{A}(\vee E)$$

$(NonDet)$ means that if two distinct proof of A are given, one of them will be chosen in a nondeterministic way. This is the well-known nondeterminacy both in classical and intuitionistic natural deduction.

3.3.3 Auxiliary Rules

$$\frac{M : \sigma \to \sigma \quad a : \sigma \quad n : nat}{ap(M^n, a) : \sigma}(exp)$$

$$\frac{f : \sigma_1 \to \tau_1 \quad g : \sigma_2 \to \tau_2}{f \times g : \sigma_1 \times \sigma_2 \to \tau_1 \times \tau 2}$$

3.4 Embedding of Computational Streams into Stream Types

There are two layers of streams: the layer of choice sequences and that of computational streams. The latter is embedded into the former.
Let $M : nat \to \sigma$, then $M : I_\sigma$. Then, let $\varphi : (nat \to \sigma) \to C_\sigma$ be $\varphi(M) \equiv ap(\nu z.\lambda n.\ (M(n) :: z(n+1)), 0)$ for arbitrary $M : nat \to \sigma$. Also, let $\psi : C_\sigma \to (nat \to \sigma)$ be $\psi(N) \equiv \lambda n.\ hd(tl^n(N))$ for arbitrary $N : C_\sigma$. Then, the following holds:

Proposition 1: *(Canonical Embedding) $nat \to \sigma (\subset I_\sigma)$ and C_σ are isomorphic:*
(1) For arbitrary $M : nat \to \sigma$, $\psi(\varphi(M)) = M$ in I_σ;
(2) For arbitrary $N : C_\sigma$, $\varphi(\psi(N)) = N$ in C_σ where $=$ is the extensional equality.

Proof: (1) First prove $\forall n : nat.\ tl^n(ap(L, 0)) = ap(L, n)$ where $L \stackrel{\text{def}}{=} \nu z.\lambda n.\ (M(n) :: z(n+1))$. Then the rest of the proof is easy. (2) Straightforward.∎

This embedding can be naturally extended to $C_{\sigma_1, \cdots, \sigma_n}$.

3.5 Realizability Interpretation

The realizability defined in this section is a variant of q-realizability, and is obtained by modifying the realizability given in [21] and [25].

A new class of formulas called realizability relations is introduced to define q-realizability.

Def. 4: Realizability relation
A *realizability relation* is an expression in the form of $\overline{a}$ q A, where A is a formula defined and $\overline{a}$ is a finite sequence of variables which does not occur in A. $\overline{a}$ is called a *realizing variables* of A. For a term, M, M q A, which reads "a term, M, realizes a formula, A", denotes $(\overline{a}$ q $A)_{\overline{a}}[M]$, and M is called a *realizer* of A.

In the following, a formula means one other than realizability relation. A type is assigned for each formula.

Def. 5: $type(A)$
Let A be a formula. Then, a type of A, $type(A)$, is defined as follows:

1. $type(A)$ is empty, if A is rank 0;
2. $type(A \,\&\, B) \stackrel{\text{def}}{=} type(A) \times type(B)$;
3. $type(A \vee B) \stackrel{\text{def}}{=} \mathbf{2}^+ \times type(A) \times type(B)$;
4. $type(A \Rightarrow B) \stackrel{\text{def}}{=} type(A) \rightarrow type(B)$;
5. $type(\forall x : \sigma.\ A) \stackrel{\text{def}}{=} \sigma \rightarrow type(A)$;
6. $type(\exists x : \sigma.\ A) \stackrel{\text{def}}{=} \sigma \times type(A)$;

Proposition 2: *Let A be a formula with a free variable x. Then, $type(A) = type(A_x[M])$ for any term M of the same type as x.*

Def. 6: q-realizability

1. If A is a rank 0 formula, then () q $A \stackrel{\text{def}}{=} A$;
2. $\overline{a}$ q $A \Rightarrow B \stackrel{\text{def}}{=} \forall b : type(A).\ (A \,\&\, b$ q $A \Rightarrow \overline{a}(b)$ q $B)$;
3. $(a, \overline{b})$ q $\exists x : \sigma.\ A \stackrel{\text{def}}{=} a : \sigma \,\&\, A_x[a] \,\&\, \overline{b}$ q $A_x[a]$;
4. $\overline{a}$ q $\forall x : \sigma.\ A \stackrel{\text{def}}{=} \forall x : \sigma.\ (\overline{a}(x)$ q $A)$;
5. $(z, \overline{a}, \overline{b})$ q $A \vee B \stackrel{\text{def}}{=} (z = L \,\&\, A \,\&\, \overline{a}$ q $A) \vee (z = R \,\&\, B \,\&\, \overline{b}$ q $B)$
   if A and B are distinct or $A = B$ and A is not rank 0;
6. $\bullet$ q $A \vee A \stackrel{\text{def}}{=} A$ if A is rank 0;
7. $(\overline{a}, \overline{b})$ q $A \,\&\, B \stackrel{\text{def}}{=} \overline{a}$ q $A \,\&\, \overline{b}$ q B.

Proposition 3: *Let A be any formula. If $\overline{a}$ q A, then $\overline{a} : type(A)$.*

From the definition of realizability, realizing variables can be determined from the construction of the formulas as follows:

Def. 7: Realizing variables: $Rv(A)$

1. $Rv(A) \stackrel{\text{def}}{=} () \quad \cdots$ if A is rank 0;
2. $Rv(A \,\&\, B) \stackrel{\text{def}}{=} (Rv(A), Rv(B))$;
3. $Rv(A \vee B) \stackrel{\text{def}}{=} (z, Rv(A), Rv(B))$ (z is a fresh variable);
4. $Rv(A \Rightarrow B) \stackrel{\text{def}}{=} Rv(B)$;
5. $Rv(\forall x : \sigma.\ A) \stackrel{\text{def}}{=} Rv(A)$;
6. $Rv(\exists x : \sigma.\ A) \stackrel{\text{def}}{=} (z, Rv(A))$ (z is a fresh variable).

Def. 8: Length of a formula
The length of a formula A, $l(A)$, is the length of $Rv(A)$ as a sequence of variables.

Proposition 4: *A formula A is rank 0 if and only if $l(A) = 0$*

Theorem 2: Soundness of **q**-realizability
*Assume that A is a formula. If A is proved without $(Coind)$, then there is a term, T, such that T **q** A can be proved, $FV(T) \subset FV(A)$ and T is equal to a sequence of terms of length $l(A)$.*

Proof: By induction on the construction of the proof of A. We prove here for the cases that the last rule in the proof is (ST) or $(NonDet)$. The remainder par of the proof will be found in [25] or [27].
Case (ST): Assume that the following proof is given:

$$\frac{\begin{array}{c}\Sigma \\ \forall n.\exists x.\ A(n,x)\end{array}}{\exists Y.\forall n.\ A(n,Y(n))}(ST)$$

Then, by the induction hypothesis, there is a term, e, such that the following proof can be constructed from Σ:

$$\begin{array}{c}\Sigma' \\ e\ \mathbf{q}\ \forall n.\exists x.\ A(n,x)\end{array}$$

By the definition of **q**-realizability,

$$\begin{aligned}
e\ \mathbf{q}\ \forall n.\exists x.\ A(n,x) &\equiv \forall n.\ e(n)\ \mathbf{q}\ \exists x.\ A(n,x) \\
&\equiv \forall n.\ tl(e(n))\ \mathbf{q}\ A(n, \mathbf{p_0}(e(n))) \\
&= \forall n.\ ap(\lambda n.\ tl(e(n)), n)\ \mathbf{q}\ A(n, ap(\lambda n.\ \mathbf{p_0}(e(n)), n)) \\
&= \lambda n.\ tl(e(n))\ \mathbf{q}\ \forall n.\ A(n, ap(\lambda n.\ \mathbf{p_0}(e(n)), n)) \\
&\equiv (\lambda n.\ \mathbf{p_0}(e(n)), \lambda n.\ tl(e(n)))\ \mathbf{q}\ \exists Y.\forall n.\ A(n, Y(n)) \\
&= \lambda n.\ c(n)\ \mathbf{q}\ \exists Y.\forall n.\ A(n, Y(n))
\end{aligned}$$

By η-rule, $\lambda n.\ e(n) = e$. Therefore,

$$\begin{array}{c}\Sigma' \\ e\ \mathbf{q}\ \exists Y.\forall n.\ A(n,Y(x))\end{array}$$

Case $(NonDet)$: Assume that A is proved as follows:

$$\dfrac{\begin{array}{cc}\Sigma_0 & \Sigma_1 \\ A & A\end{array}}{A}(NonDet)$$

As $(NonDet)$ is a derived rule, this can be translated to the following proof:

$$\dfrac{\dfrac{}{\top \vee \top}(\vee I) \quad \begin{array}{cc}[\top] & [\top] \\ \Sigma_1 & \Sigma_2 \\ A & A\end{array}}{A}(\vee E)$$

By the induction hypothesis, the following proofs can be constructed:

$$\begin{array}{cc}\Sigma_1' & \Sigma_2' \\ e_1 \ \mathbf{q} \ A & e_2 \ \mathbf{q} \ A\end{array}$$

Then, let $c \stackrel{\text{def}}{=} if\ L = \bullet\ then\ e_1\ else\ e_2$, and the following proof can be constructed:

$$\dfrac{\bullet = L \vee \bullet = R \qquad \dfrac{\begin{array}{cc}[\bullet = L] & \\ \Sigma_3 & \Sigma_1' \\ e = e_1 & c_1 \ \mathbf{q} \ A\end{array}}{e \ \mathbf{q} \ A}(= E) \qquad \dfrac{\begin{array}{cc}[\bullet = R] & \\ \Sigma_4 & \Sigma_2' \\ e = e_2 & c_2 \ \mathbf{q} \ A\end{array}}{e \ \mathbf{q} \ A}(= E)}{e \ \mathbf{q} \ A}(\vee E)$$

∎

3.6 Realizability Interpretation of $(Coind)$

Theorem 3: *$(Coind)$ has* **q***-realizability interpretation. More precisely, theorem 2 also holds when $(Coind)$ is used for proving a formula.*

Proof: The proof is performed as continuation of that of theorem 2. Assume that there is a proof by $(Coind)$. Let $\Sigma_{(a)}$, $\Sigma_{(b)}$ and $\Sigma_{(c)}$ denote the subproofs of the premises (a), (b) and (c). Then by the induction hypothesis, there are proofs, $\Sigma'_{(a)}$, $\Sigma_{(b)}$, and $\Sigma'_{(c)}$, of $f_M \ \mathbf{q}\ \forall \overline{X}.\exists a.\ M(\overline{X}, a)$ and $f_N \ \mathbf{q}\ \exists f.\forall \overline{X}.\forall Y.\forall n.\ (A(n, f(\overline{X}), tl(Y)) \Rightarrow A(n+1, \overline{X}, Y))$. Note that $A(n, \overline{X}, Y)$ is a rank 0 formula so that $\Sigma'_{(b)} = \Sigma_{(b)}$. Also, $f_M : I_{\sigma_1,\cdots,\sigma_n} \rightarrow (\tau \times type(M(\overline{X}, a)))$ by proposition 2. By the definition of **q**-realizability, $f_N \ \mathbf{q}\ \exists f.\forall \overline{X}.\forall Y.\forall n.\ (A(n, f(\overline{X}), tl(Y)) \Rightarrow A(n+1, \overline{X}, Y))$
$\equiv f_N : I_{\sigma_1,\cdots,\sigma_n} \rightarrow I_{\sigma_1,\cdots,\sigma_n}\ \&\ \forall \overline{X}.\forall Y.\forall n.\ (A(n, f_N(\overline{X}), tl(Y)) \Rightarrow A(n+1, \overline{X}, Y))$. Then, the following proof the conclusion of $(Coind)$ can can be constructed:

$\Pi \stackrel{\text{def}}{=}$

$$\dfrac{\dfrac{\dfrac{[\overline{X}]^{(1)}}{\Sigma_0}}{\exists Y.\forall m.\forall S.\ A(0, f_N^m(\overline{X}), (Y(m) :: S))} \qquad \dfrac{[\overline{X}]^{(1)}[Y]^{(2)}[\forall m.\forall S.\ A(0, f_N^n(\overline{X}), (Y(m) :: S))]^{(2)}}{\Sigma_1}}{\dfrac{\exists Y.\forall n.\ A(n, \overline{X}, Y)}{\forall \overline{X}.\exists Y.\forall n.\ A(n, \overline{X}, Y)}(\forall I)^{(1)}}(\exists E)^{(2)}$$

$\Sigma_0 \stackrel{\text{def}}{=}$

$$\dfrac{\dfrac{\dfrac{\dfrac{[\overline{X}]^{(1)}[m]^{(3)} f_N}{f_N^m(\overline{X})}(exp) \qquad \Sigma_{(a)}}{\forall \overline{X}.\exists a.\ M(\overline{X}, a)}}{\exists a.\ M(f_N^m(\overline{X}), a)}(\forall E) \qquad \dfrac{[a]^{(4)}[\overline{X}]^{(1)}[m]^{(3)}[M(f_N^m(\overline{X}), a)]^{(4)}}{\Sigma_{00}}}{\dfrac{\exists a.\forall S.\ A(0, f_N^m(\overline{X}), (a :: S))}{\dfrac{\exists a.\forall S.\ A(0, f_N^m(\overline{X}), (a :: S))}{\forall m.\exists a.\forall S.\ A(0, f_N^m(\overline{X}), (a :: S))}(\forall I)^{(3)}}(ST)}(\exists E)^{(4)}$$

$$\Sigma_{00} \stackrel{\text{def}}{=} \dfrac{\dfrac{[M(f_N^m(\overline{X}), a)]^{(4)} \quad M(f_N^m(\overline{X}), a) \Rightarrow A(0, f_N^m(\overline{X}), (a :: S))}{A(0, f_N^m(\overline{X}), (a :: S))}(\Rightarrow E)}{\dfrac{[a]^{(4)} \quad \forall S.\ A(0, f_N^m(\overline{X}), (a :: S))}{\exists a.\forall S.\ A(0, f_N^m(\overline{X}), (a :: S))}(\exists I)}(\forall I)^{(5)}$$

$$\Sigma_{000} \stackrel{\text{def}}{=} \dfrac{\dfrac{\dfrac{[\overline{X}]^{(1)}[m]^{(3)} f_N}{f_N^m(\overline{X})}(exp) \quad [a]^{(4)}}{[S]^{(5)}} \qquad \Sigma_{(b)}}{M(f_N^m(\overline{X}), a) \Rightarrow A(0, f_N^m(\overline{X}), (a :: S))}(\forall E)$$

$$\Sigma_1 \stackrel{\text{def}}{=} \dfrac{\dfrac{\dfrac{[\overline{X}]^{(1)}[Y]^{(2)}[n]^{(6)}}{[\forall m.\forall S.A(0, f_N^n(\overline{X}), (Y(m) :: S))]^{(2)}}}{\Sigma_{10}} \qquad \dfrac{[\overline{X}]^{(1)}[Y]^{(2)}[n]^{(6)}}{\Sigma_{11}}}{\dfrac{A(0, f_N^n(\overline{X}), tl^n(Y)) \qquad A(0, f_N^n(\overline{X}), tl^n(Y)) \Rightarrow A(n, \overline{X}, Y)}{A(n, \overline{X}, Y)}(\Rightarrow E)}$$

$$\dfrac{\dfrac{[Y]^{(2)} \quad \forall n.\ A(n, \overline{X}, Y)}{\exists Y.\forall n.\ A(n, \overline{X}, Y)}(\exists I)}(\forall I)^{(6)}$$

$\Sigma_{10} \stackrel{\text{def}}{=}$

$$\dfrac{\begin{array}{cc} [Y]^{(2)} & [Y]^{(2)} \\ [n]^{(6)} & [n]^{(6)} \\ \Pi_{100} & \Pi_{101} \end{array} \qquad \dfrac{[n]^{(6)} \quad \dfrac{[n]^{(6)}[Y]^{(2)}}{tl^{n+1}(Y)} \quad [\forall m.\forall S.\ A(0, f_N^m(\overline{X}), (Y(m) :: S))]^{(2)}}{A(0, f_N^n(X), (Y(n) :: tl^{n+1}(Y)))}(\forall E)}{A(0, f_N^n(\overline{X}), tl^n(Y))}(= E)$$

where

$$\Pi_{100} \stackrel{\text{def}}{=} \dfrac{[n]^{(6)}[Y]^{(2)}}{Y(n) - hd(tl^n(Y))} \qquad \Pi_{101} \stackrel{\text{def}}{=} \dfrac{[n]^{(6)}[Y]^{(2)}}{tl^n(Y) = (hd(tl^n(Y)) :: tl^{n+1}(Y))}$$

$\Sigma_{11} \stackrel{\text{def}}{=}$

$$\dfrac{\dfrac{[\overline{X}]^{(1)}\,[Y]^{(2)}\,[n]^{(0)} \qquad \dfrac{f_N \;\; tl}{f_N \times tl} \quad \forall\overline{X}.\forall Y.\forall n.\ (A(n, f_N(\overline{X}), tl(Y)) \Rightarrow A(n+1, \overline{X}, Y))}{\forall\overline{X}.\forall Y.\forall n.\ (A(0, f_N^n(\overline{X}), tl^n(Y)) \Rightarrow A(n, \overline{X}, Y))}(R)}{A(0, f_N^n(\overline{X}), tl^n(Y)) \Rightarrow A(n, \overline{X}, Y)}(\forall E)$$

By the induction hypothesis, there is a term, T, such that a proof of $T\ \mathbf{q}\ \forall\overline{X}.\exists Y.\forall n.\ A(n, \overline{X}, Y)$ can be constructed from the proof, Π. The term, T, is $\lambda\overline{X}.\lambda m.\ ap(f_M, f_N^m(\overline{X}))$. ∎

The proof of the theorem means that the program extracted from a proof by (*Coind*) is in the form of $\lambda\overline{X}.\lambda m\ ap(f_M, f_N^m(\overline{X}))$. By the embedding, φ, of $C_{\sigma_1,\cdots,\sigma_m}$ into $I_{\sigma_1,\cdots,\sigma_m}$, the program is $\lambda\overline{X}.\ ap(\nu z.\lambda k.\ (ap(\lambda m.\ ap(f_M, f_N^m(\overline{X})), k) :: ap(z, k+1)), 0)$, and this is extensionally equal to $\nu z.\lambda\overline{X}.\ (ap(f_M, X) :: ap(z, ap(f_N, \overline{X})))$. This is essentially the same as the PMN combinator in Hagino's categorical typed lambda calculus.

4 Examples

4.1 Double$^\omega$

Specification: $\forall X : I_{nat}.\exists Y : I_{nat}.\forall n : nat.\ A(n, X, Y)$
where $A(n, X, Y) \stackrel{\text{def}}{=} elem(n, Y) = 2 \cdot elem(n, X)$.

Proof: Let $M(X, a) \stackrel{\text{def}}{=} a = 2 \cdot hd(X)$, and $\forall X : I_{nat}.\exists a : nat.\ M(X, a)$ can be proved. $\forall X : I_{nat}.\forall a : nat.\forall S : I_{nat}.\ (M(X, a) \Rightarrow A(0, X, (a :: S)))$ holds because $A(0, X, (a :: S)) \equiv M(X, a)$. Finally, by letting $f = \lambda X.\ tl(X)$, $\exists f : I_{nat} \ \rightarrow I_{nat}.\forall X : I_{nat}.\exists Y : I_{nat}.\forall n : nat.\ (A(n, f(X), tl(Y)) \Rightarrow A(n+1, X, Y))$ can be proved. Then, by (*Coind*), the specification is proved. ∎

The program extracted from the proof is $\lambda X.\lambda m.\ 2 \cdot hd(tl^m(X))$.

4.2 Step Filter

The process which has one input stream and filters out the $2 \cdot n + 1$ th $(n = 0, 1, 2, \cdots)$ elements in the input stream can be specified as follows:

Specification: $\forall X : I_\sigma.\exists Y : I_\sigma.\forall n : nat.\ A(n, X, Y)$
where $A(n, X, Y) \stackrel{\text{def}}{=} elem(n, Y) = elem(2 \cdot n, X)$

Proof: Let $M(X, a) \stackrel{\text{def}}{=} a = hd(X)$, and $\forall X : I_\sigma.\exists a : \sigma.\ M(X, a)$ can be proved. $\forall X : I_\sigma.\exists a : \sigma.\forall S : I_\sigma.\ (M(X, a) \Rightarrow A(0, X, (a :: S)))$ holds because $A(0, X, (a :: S)) \equiv$

$M(X,a)$. Finally, by letting $f \stackrel{\text{def}}{=} \lambda X.\ tl^2(X)$, $A(n, f(X), tl(Y)) \equiv A(n+1, X, Y)$ can be proved, so that $\exists f : I_\upsilon \to I_\sigma . \forall X : I_\sigma . \forall Y : I_\sigma . \forall n : nat.\ (A(n, f(X), tl(Y)) \Rightarrow A(n+1, X, Y))$ holds. Then, by $(Coind)$, the specification is proved. ∎

The program extracted from the proof is $\lambda X.\lambda m.\ hd(tl^{2\cdot m}(X))$.

4.3 Stream Filter

Let $p : nat$ be arbitrary parameter. A parameterized process with a natural number stream, X, as input stream which filters out all the element of X which can be divided by p can be defined as follows:

Specification: $\forall p : nat. \forall X : I_{nat_\perp} . \forall Y : I_{nat_\perp} . \forall n : nat.\ \Diamond A(p, n, X, Y)$
where $A(p, n, X, Y) \stackrel{\text{def}}{=}$
$((p|elem(n, X))\ \&\ elem(n, Y) - \perp) \vee (\neg(p|elem(n, X))\ \&\ elem(n, Y) = elem(n, X))$

Proof: Let $p : nat$ be arbitrary, and $\forall X . \forall Y . \forall n.\ \Diamond A(p, n, X, Y)$ will be proved by $(Coind)$. Let $M(X, a) \stackrel{\text{def}}{=} ((p|elem(n, X))\ \&\ elem(n, Y) = \perp) \vee (\neg(p|elem(n, X))\ \&\ elem(n, Y) = elem(n, X))$. $\forall X : I_{nat_\perp} . \exists a : I_{nat_\perp} .\ M(X, a)$ is proved as follows. Let $X : I_{nat_\perp}$ be arbitrary. As $(p|hd(X)) \vee \neg(p|hd(X))$ is decidable – note that $\neg(p|hd(X))$ holds if $hd(X) = \perp$, $\exists a : I_{nat_\perp} .\ M(X, a)$ can be proved by divide and conquer: if $(p|hd(X))$, let $a = \perp$ and otherwise let $a = hd(X)$. For arbitrary $X : I_{nat_\perp}$, $a : nat_\perp$, and $S : I_{nat_\perp}$, $A(p, 0, X, (a :: S)) = M(X, a)$ because $elem(0, X) = hd(X)$ and $elem(0, (a :: S)) = a$. Hence, $\forall X : I_{nat_\perp} . \forall a : nat_\perp . \forall S : I_{nat_\perp} .\ (M(X, a) \Rightarrow A(p, 0, X, (a :: S)))$ is proved. Finally, by letting $f = \lambda X.\ tl(X)$, $\exists f : I_{nat_\perp} \to I_{nat_\perp} . \forall X : I_{nat_\perp} . \forall Y : I_{nat_\perp} . \forall n : nat.\ (A(p, n, f(X), tl(Y)) \Rightarrow A(n+1, X, Y))$ is proved. Then, by $(Coind)$ and $(\forall I)$ the specification is proved. ∎

The program extracted from the proof is $\lambda p.\lambda X.\lambda m.\ ap(f_M, f_N^m(X))$ where
$f_M \stackrel{\text{def}}{=} \lambda X.\ if\ (p|hd(X))\ then\ \perp\ else\ hd(X)$ and $f_N \stackrel{\text{def}}{=} \lambda X.\ tl(X)$.

4.4 Eratosthenes' Sieve Algorithm

Specification: $\forall X : I_{nat_\perp} . \exists Y : I_{nat_\perp} . \forall n : nat.\ \Diamond A(n, X, Y)$
where $A(n, X, Y)$
$\stackrel{\text{def}}{=} (PR(elem(n, X))\ \&\ elem(n, Y) = elem(n, X)) \vee (\neg PR(elem(n, X))\ \&\ elem(n, Y) = \perp)$
and $PR(m) \stackrel{\text{def}}{=} \forall n : nat.\ (2 \le n \le m \Rightarrow \neg(\exists d : nat.\ m = d \cdot n))\ \&\ m \neq \perp$.

Proof: The proof by $(Coind)$. Let $M(X, a) \stackrel{\text{def}}{=} (PR(hd(X))\ \&\ a = hd(X)) \vee (\neg PR(hd(X))\ \&\ a = \perp)$. As $PR(hd(X)) \vee \neg PR(hd(X))$ is decidable, $\forall X : I_{nat_\perp} . \exists a : nat_\perp .\ M(X, a)$ can be proved by $(\forall I)$ and $(\vee E)$. As $A(0, X, (a :: S)) = M(X, a)$ for arbitrary $X : I_{nat_\perp}$, $a : I_{nat_\perp}$ and $S : I_{nat_\perp}$, $\forall X : I_{nat_\perp} . \forall a : nat_\perp . \forall S : I_{nat_\perp} .\ (M(X, a) \Rightarrow A(0, X, (a :: S)))$ holds. For

the proof of (d), let $f = \lambda X.\ if\ PR(hd(X))\ then\ flt(hd(X), tl(X))\ else\ tl(X)$ where pr is an abbreviation of some suitable decision procedure to check whether X is prime or not, and $flt(p, X) \stackrel{\text{def}}{=} \lambda m.ap(\lambda X.\ if\ (p|hd(X))\ then\ \bot\ else\ hd(X), tl^m(X))$. Then, $(1)pr(elem(n, f(X))) \Rightarrow pr(elem(n, tl(X)))$, $(2)\neg pr(elem(n, f(X))) \Rightarrow \neg(elem(n, tl(X)))$ and $(3)pr(elem(n, f(X))) \Rightarrow elem(n, f(X)) = elem(n, tl(X))$ hold for arbitrary $X : I_{nat_\bot}$ and $n : nat$. This can be proved as follows. If $\neg pr(hd(X))$, $f(X) = tl(X)$ so that (1), (2) and (3) is trivial. Otherwise, note that for arbitrary $X : I_{nat_\bot}$ and $n : nat$, $elem(n, flt(hd(X), tl(X))) = ap(flt(hd(X), tl(X)), n)$
$= if\ (hd(X)|hd(tl^{n+1}(X)))\ then\ \bot\ else\ hd(tl^{n+1}(X))$
$= if\ (hd(X)|elem(n, tl(X)))\ then\ \bot\ else\ elem(n, tl(X))$ holds, so that $elem(n, f(X))$ is $elem(n, tl(X))$ if $elem(n, tl(X))$ is not prime and cannot be divided by $hd(X)$ or if $elem(n, tl(X))$ is prime, and $\bot$ otherwise. Hence, (1), (2) and (3) holds. Then, as $A(n, f(X), tl(Y))$
$\equiv (pr(elem(n, f(X))) \,\&\, elem(n+1, Y) = elem(n, f(X)))\vee(\neg pr(elem(n, f(X))) \,\&\, elem(n+1, Y) = \bot)$, $A(n+1, X, Y)$ holds. This proves (d). ∎

The program extracted from this proof is $\lambda X.\lambda m.\ ap(f_M, f_N^m(X))$ where
$f_M \stackrel{\text{def}}{=} \lambda X.\ if\ pr(hd(X))\ then\ hd(X)\ else\ \bot$
and $f_N \stackrel{\text{def}}{=} \lambda X.\ if\ pr(hd(X))\ then\ flt(hd(X), tl(X))\ else\ tl(X)$.

4.5 Nondeterministic Stream Merger

The stream merge operation is one of the typical example of nondeterminacy. A merge program can be defined by $(Coind)$ as follows, but, because of the condition (d) on $A(n, (X, Y), Z)$, the specification is much weaker than that of the merge operation. It depends on how M is defined and how the value of f is defined for (a) and (d) in the premises of $(Coind)$.

Specification: $\forall(X, Y) : I_{\sigma,\sigma}.\exists Z : I_\sigma.\forall n : nat.\ \Diamond A(n, (X, Y), Z)$
where $A(n, (X, Y), Z) \stackrel{\text{def}}{=} (\exists m : nat.\ elem(n, Z) = elem(m, X)) \vee (\exists l : nat.\ elem(n, Z) = elem(l, Y))$

The proof will continue as follows: Let $M((X, Y), a) \stackrel{\text{def}}{=} a = hd(X)$, and $\forall(X, Y) : I_{\sigma,\sigma}.\exists a : \sigma.\ M((X, Y), a)$ can be proved. For arbitrary $(X, Y) : I_{\sigma,\sigma}$, $a : \sigma$ and $S : I_\sigma$, assume $M((X, Y), a)$. Because $elem(0, X) = hd(X) = a$ and $elem(0, (a :: S)) = a$, $elem(0, (a :: S)) = elem(0, X)$. So that by $(\exists I)$ and $(\vee I)$ $A(0, (X, Y), (a :: S))$ is proved. Hence, by $(\Rightarrow I)$ and $(\forall I)$, $\forall(X, Y) : I_{\sigma,\sigma}.\exists a : \sigma.\forall S : I_\sigma.\ (M((X, Y), a) \Rightarrow A(0, (X, Y), (a :: S)))$ is proved. Finally, (d), which is $\exists f : I_{\sigma,\sigma} \to I_{\sigma,\sigma}.\forall(X, Y) : I_{\sigma,\sigma}.\forall Z : I_\sigma.\forall n : nat.\ (A(n, f(X, Y), tl(Z)) \Rightarrow A(n+1, (X, Y), Z))$, is proved as follows: Let $(X, Y) : I_{\sigma,\sigma}$, $Z : I_\sigma$ and $n : nat$ be arbitrary. Then, $A(n, (tl(X), Y), tl(Z)) = (\exists m.\ elem(n, tl(Z)) = elem(m, tl(X))) \vee (\exists l.\ elem(n, tl(Z)) = elem(l, Y)) \equiv (\exists m.\ elem(n+1, Z) = elem(m+1, X)) \vee (\exists l.\ elem(n+1, Z) = elem(l, Y)) \equiv (\exists m'.\ elem(n+1, Z) = elem(m', X)) \vee (\exists l.\ elem(n+1, Z) = elem(l, Y)) \equiv A(n+1, (X, Y), Z)$. Hence, a proof of (d) is obtained. Similarly, $A(n, (Y, tl(X)), tl(Z)) \equiv A(n+1, (X, Y), Z)$ holds. Hence, another proof of

(d) is obtained. Then, by $(NonDet)$, (d) is proved, and, by $(Coind)$, the specification is proved. ∎

The program extracted from this proof is $\lambda(X,Y) : I_{\sigma,\sigma}.\lambda m.\ ap(f_M, f_N^m(X,Y))$ where $f_M \stackrel{\text{def}}{=} \lambda X.hd(X)$ and $f_N \stackrel{\text{def}}{=} \lambda(X,Y).\ if\ \bullet = left\ then\ (tl(X),Y)\ else\ (Y,tl(X))$.

5 Discussion and Concluding Remarks

The system and method presented in this paper is an extension of the pioneering works on formal treatment of streams such as [10] and [15] in the sense that recursive call programs on streams, specification and verification of them can be uniformly treated in the framework of logic, and nondeterminacy is also taken into account. Similar work to ours was carried out by P. Dybjer and H. P. Sander [9] as a verification method of concurrent systems. We give here some comparison with their work. Some of the differences come from the differences of the purpose of the system: verification or program construction.

(1) μ-coinduction
They used a greatest fixedpoint induction in μ-calculus [19]

$$(R \subseteq \Phi[R/X]) \to (R \subseteq \nu X.\Phi) \qquad (\mu\text{-}coind)$$

for the verification of stream transformers defined as mutually recursive call functions on streams.
The reasoning they used in the verification of the alternating bit protocol – transferring a bit stream safely through unreliable channels –, can be presented in the natural deduction style as follows:

$$\frac{B(x,TR(x)) \quad \forall x : I_\sigma.\forall y : I_\sigma.(B(x,y) \Rightarrow hd(x) = hd(y)\ \&\ B(tl(x),tl(y)))}{\forall x : I_\sigma.\ x \approx TR(x)}(coind)$$

where B is suitably defined relation and TR is the mutually recursive function defining the concurrent system for the protocol. However, if one wish to specify various concurrent programs with formulas in the form of $\forall x : I_\sigma.\exists y : I_\tau.A(x,y)$ as in the traditional constructive programming, one need to use the logical connectives and the quantifiers. They do not give how to handle specifications with the logical connectives and the quantifier, but it may be possible if we introduce recursively defined predicates as used for finite lists in [28]. For example, a specification of $double^\omega$ will be

$$\forall x : I_{nat}.\exists y : I_{nat}.\ A(x,y)$$
$$\text{where } A(x,y) = (hd(y) = 2 \cdot hd(x))\ \&\ A(tl(x),tl(y))$$

The advantage of this idea is that it is unnecessary to regard a stream as a sequence (an element of type $nat \to \sigma$) so that specifications can be written in a more elegant way than

in our method. However, if we take not only lawlike sequences but also lawless sequences into account, our approach seems to be more natural.
On the other hand, (μ-*coind*) has direct connection to the notion of bisimulation [1, 16], but the relation between our coinduction and bisimulation is not always clear.

(2) Infinity Condition of Streams
They use (μ-*coind*) also for verifying infinity of streams. That is to keep the uniformity of the verification method. However, infinity condition is too strong to handle some class of programs such as the stream filer whose output can be empty stream. We introduced the notion of general streams in which empty or finite streams are virtually infinite to overcome this problem.

(3) Treatment of Nondeterminacy
They treat nondeterministic communication agency as incompletely specified deterministic one, and used *Fair* type for the verification of nondeterministic choice. We follow essentially the same idea, but introduced a coin flipper term in the programming language and hide it at the proof level by using the inherent nondeterminacy in the cut elimination of proofs in ($\vee E$).

There are other problems remaining as the future work. We borrowed the theory of choice sequences for the treatment of streams, and gave a coinduction rule, (*Coind*). There are other formulation of induction on choice sequences, bar induction. One variant of bar induction (monotone bar induction) is as follows:

$$\frac{\forall\alpha.\exists x.P(\overline{\alpha}(x)) \quad \forall m.\forall n.(P(n) \Rightarrow P(n*m)) \quad \forall n.(\forall y.P(n*(y)) \Rightarrow P(n))}{P(())}(BI)$$

where α, x (also y), and n (also m) range over choice sequences, natural numbers, and finite sequences, $\overline{\alpha}(x)$ denotes the initial segment of α up to the position x, $n*m$ denotes concatenation, and $()$ denotes the empty sequence.
The author does not know well the relation between (*Coind*) and (BI) or whether (BI) could be used for defining stream transformers. Bar induction is formulated for continuity of functionals of type $N^N \to N$ (N is the type of natural numbers), and if $N^N \to N^N$ version of bar induction (bar induction for function continuity) is formulated, this could be used for defining stream transformers. Another problem is that whether our system is powerful enough to define wide variety of concurrent systems. For example, whether the nondeterminacy problem presented in [6] can be handled in our system, or not.

Acknowledgments

I would like to thank Yasusi Fujiwara, Susumu Hayashi, Kuniaki Mukai, Peter Dybjer and the members of concurrent programming research group in ICOT for helpful discussions.

References

[1] Aczel, P. *Non-well-founded sets.* CSLI Lecture Notes 14. Stanford University, 1988.

[2] Aczel, P. and Mendler, N. A Final Coalgebra Theorem. In *Category Theory and Computer Science, LNCS 389*, 1989.

[3] Bates, J. I. *A logic for correct program development.* PhD thesis, Cornell University, 1979.

[4] Beeson, M. J. *Foundation of Constructive Mathematics.* Springer–Verlag, 1985.

[5] Bishop, E. *Foundation of Constructive Analysis.* McGraw-Hill, New York, 1967.

[6] Brock, J. D. and Ackerman, W. B. Scenarios: A Model of Non-determinate Computation. In *Lecture Notes in Computer Science, Vol. 107*, 1981.

[7] Burge, W. H. *Recursive Programming Techniques.* Addison-Wesley, 1975.

[8] Constable, R. et al. *Implementing Mathmatics with the NuPrl Proof Development System.* Prentice–Hall, 1986.

[9] Dybjer, P. and Sander, H. P. A Functional Programming Approach to the Specification and Verification of Concurrent Systems. *Formal Aspects of Computing*, 1:303 – 319, 1989.

[10] Hagino, T. A Typed Lambda Calculus with Categorical Type Constructors. In *Category Theory and Computer Science, LNCS 283*, 1987.

[11] Hayashi, S. and Nakano, H. *PX : A Computational Logic.* The MIT Press, 1988.

[12] Howard, W. A. The formulas-as-types notion of construction. In *Essays on Combinatory Logic, Lambda Calculus and Formalism, eds. J. P. Seldin and J. R. Hindley.* Academic Press, 1980.

[13] Kahn, G. and MacQueen, D. B. Coroutine and Networks of Parallel Processes. In *Proceedings of Information Processing 77*, pages 993 998. Noth-Holland, 1977.

[14] Keller, R. M. Denotational Models for Parallel Programs with Indeterminate Operators. In *Formal Description of Programming Concepts*, pages 337 – 366. North–Holland, 1978.

[15] Mendler, N., Panangaden, P. and Constable, R. L. Infinite Objects in Type Theory. In *Proceedings of Symposium on Logic in Computer Science'86*, 1986.

[16] Milnor, R. *A Calculus of Communicating Systems.* Lecture Note in Conputer Science, 92. Springer–Verlag, 1980.

[17] Mohning, C. Algorithm Development in the Calculus of Constructions. In *Proceedings of Symposium on Logic in Conputer Science*, pages 84–91, 1986.

[18] Nordström, B., Petersson, K. and Smith, J. *Programming in Martin-Löf's Type Theory, An Introduction.* International Series of Monographs on Computer Science 7. Oxford Science Publications, 1990.

[19] Park, D. Finiteness is Mu-Ineffable. *Theoretical Computer Science*, 3:173 – 181, 1976.

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

[21] Sato, M. Typed Logical Calculus. Department of Information Science 85-13, University of Tokyo, 1985.

[22] Smyth, M. B. and Plotkin, G. D. The Category Theoretic Solution of Recursive Domain Equations. *SIAM Journal of Computing*, 11, 1982.

[23] Takayama, Y. Writing Programs as QJ-Proofs and Compiling into PROLOG Programs. In *Proceedings of 4th Symposium on Logic Programming.* IEEE, 1987.

[24] Takayama, Y. QPC : QJ-Based Proof Compiler Symple Examples and Analysys. In *European Symposium on Programming '88,* Lecture Notes in Computer Science 300. Springer-Verlag, 1988.

[25] Takayama, Y. $\mathbf{QPC}^2$: A Second Order Logic for Higher Order Programming. Submitted to Theoretical Computer Science, March 1990.

[26] Takayama, Y. Extraction of Redundancy-free Programs from Constructive Natural Deduction Proofs. *Journal of Symbolic Computation*, (to appear).

[27] Takayama, Y. and Hayashi, S. Extended Projection Method and Realizability. Presented at ESPRIT LF Workshop in May 1990, Submitted to Information and Computation, July 1990.

[28] Tatsuta, M. Program Synthesis Using Realizability. *Theoretical Computer Science*, (to appear).

[29] Troelstra, A.S., editor. *Mathematical investigation of intuitionistic arithmetic and analysis.* Lecture Note in Mathematics. 344. Springer–Verlag, 1973.

[30] Troelstra, A.S. and van Dalen, D. *Constructivism in Mathematics, An Introduction.* Studies in Logic and the Foundation of Mathematics 121 and 123. North–Holland, 1988.

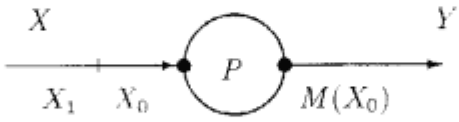

Figure 1: Fetch X_0 and Output the First Element $M(X_0)$

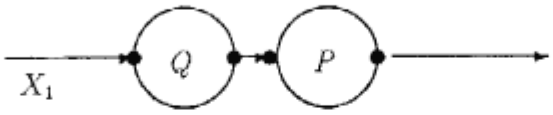

Figure 2: Interleaving a New Process Q