ICOT Technical Report: TR-684

TR-684

Defining Concurrent Processes Constructively

by
Y. Takayama

September, 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

Defining Concurrent Processes Constructively *

Yukihide Takayama
Kansai Laboratory, OKI Electric Industry Co., Ltd.
2-27 Shiromi 1-chome, Chuo-ku, Osaka 540, Japan
takayama@kansai.oki.co.jp

August 13, 1991

Abstract

This paper proposes a constructive logic in which a concurrent system can be defined as a proof of a specification. The logic is defined by adding stream types and several rules for them to an ordinary constructive logic. The unique feature of the obtained system is in the rule ($MPST$) which is a kind of structural induction on streams. Apart from other approaches in treating streams such as largest fixed point inductions and lazy type systems in typed functional languages, ($MPST$) is formulated as a purely logical rule in the natural deduction formalism which defines a concurrent process as a Burge's mapstream function. This formulation is possible when streams are viewed as sequences not infinite lists. Also, our logic has explicit nondeterminacy but we do not introduce any extralogical device. Our nondeterminacy rule, ($NonDet$), is actually a defined rule which uses inherent nondeterminacy in the traditional intuitionistic logic. Several techniques of defining stream based concurrent programs are also presented through various examples.

1 Introduction

Constructive logics give a method for formal development of programs, e.g., [8, 11]. 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. This functional interpretation of formulas is realized mechanically. Namely, if a constructive proof of the formula is given, the function, f, is extracted from the proof with q realizability interpretation

*This work was supported by the Institute for New Generation Computer Technology as a joint research project on theorem proving and its application.

[1] or with Curry-Howard correspondence of types and formulas [13]. This programming methodology will be referred to as *constructive programming* in the following.

Although constructive programming has been studied by many researchers, the constructive systems which can handle concurrency are rather few. This is mainly because most of the constructive logics have been formalized as intuitionistic logics, and the intuitionism itself does not have explicit concurrency besides proof normalization corresponding to the execution of programs [10]. For example, QJ [5] is an intuitionistic programming logic for a concurrent language, Quty. However, when we view QJ as a constructive programming system, concurrency only appears in the operational semantics of the programming language.

Linear Logic [4] gives a new formulation of constructive logic which is not based on intuitionism. This is the first constructive logic which can handle concurrency at the level of logic. In Linear Logic, formulas are regarded as processes or resources and every rule of inference defines the behavior of a concurrent operation. Linear Logic is similar to Milner's SCCS [9] in this respect, and the meaning of logical connectives are quite different from that of intuitionistic logics.

We take intermediate approach between QJ and Linear Logic, not throwing away but extending intuitionistic logic. The advantage of this approach is that the functional interpretation of logical connectives in the traditional constructive programming is preserved, and that both the sequential and concurrent parts of programs are naturally described as constructive proofs. To this end, we take the stream based concurrent programming model [2, 3]. We introduce stream types and quantification over stream types. A formula is regarded as a specification of a process when it is a universal or an existential formula over stream types, and otherwise it represents a specification of a sequential function, properties of processes or linkage relation between processes. A typical process, $\forall X.\exists Y.A(X,Y)$ where X and Y are stream variables, is regarded as a stream transformer. Most of the rules of inference are those of ordinary constructive programming systems, but rules for nondeterminacy and for stream types are also introduced. Among them, a kind of structural induction on stream types called $(MPST)$ is the hart of our extended system: With $(MPST)$, stream transformers can be defined as Burge's mapstream functions [14].

Reasoning about stream transformer can be handled by P. Dybjer and H. P. Sander [7] with a largest fixed point induction. However, their system is designed as a program verification system not as a program derivation system (constructive programming system), and even if it is modified to a program derivation system, the largest fixed point induction diminishes the advantage of constructive programming, namely, control mechanism in programs can be described as natural deduction style reasoning such as the structural induction for defining recursive call programs. T. Hagino [12] gave a clear categorical formalization of stream types (infinite list types or lazy types) whose canonical elements are given by a schema of mapstream functions, but relation between his formulation and logic is not investigated. N. Mendler and others [6] introduced lazy types and the rules for them into a constructive type theory. Their lazy types have a categorical semantics theory very similar to Hagino's one. However, Curry-Howard correspondence between types and formulas does not hold for the lazy types and their system is more like a lazy

functional programming system than a constructive programming system. On the other hand, $(MPST)$ is formulated as a purely logical rule in natural deduction which is possible when we view streams as sequences, not infinite lists.

Section 2 explains how a concurrent system is specified in logic. A process is specified by the $\forall x.\exists y.A(x,y)$ type formula as in the traditional constructive programming. The rest of the sections focus on the problem of defining processes which meet the specifications. Section 3 formulates streams and stream types. Streams are viewed as infinite lists or programs which generate infinite lists at the level of underlying programming language. At the logical reasoning level, streams are sequences, namely, total functions on natural numbers. This two level formulation of streams enables to introduce a natural deduction style structural induction rule on streams, $(MPST)$, which will be given in section 4. Section 5 presents the rest of the formalism of the whole system. The realizability interpretation which gives the program extraction algorithm from proofs will be defined. Several examples will be given in section 6 to demonstrate how stream based concurrent programming is performed in our system.

Notational preliminary: We assume first order intuitionistic natural deduction. Equalities of terms, typing relations $(M:\sigma)$, and $\top$ (true) are atomic formulas. The domain of the quantification is often omitted when it is clear from the context. Sequences of variables are denoted as $\overline{x}$ or $\overline{X}$. $M_x[N]$ 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. $(::)$ denotes the (infinite) list constructor. Function application is denoted $ap(M,N)$ or $M(N)$.

2 Specifying Concurrent Systems in Logic

The model of concurrent systems in this paper is as follows: A concurrent system consists of processes linked with streams. A process interacts with other processes only through input and output streams. The configuration of processes in a concurrent system is basically static and finite, but in some cases, which will be explained later, infinitely many new processes may be created by an already existing process. A process 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. I_σ is the type of streams over the type σ, but its definition will be given later.
The combination of two processes, $\forall X.\exists Y.\ A(X,Y)$ and $\forall P.\exists Q.\ B(P,Q)$, is described by the following proof procedure:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Sigma_1}{\forall X.\exists Y.\ A(X,Y)}}{\exists Y.\ A(X,Y)}(\forall E) \quad \Pi}{\exists Y.\exists \alpha.\ A(X,\alpha)\ \&\ B(\alpha,Y)}(\exists E)^{(1)}}{\forall X.\exists Y.\exists \alpha.\ A(X,\alpha)\ \&\ B(\alpha,Y)}(\forall I)$$

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

$$\dfrac{\dfrac{\dfrac{\Sigma_2}{\forall P.\exists Q.\ B(P,Q)}}{\exists Q.\ B(Y',Q)}(\forall E) \quad \dfrac{\dfrac{[A(X,Y')]^{(1)}\ [B(y,Q')]^{(2)}}{A(X,Y')\ \&\ B(Y',Q')}(\&\ I)}{\exists \alpha.\ A(X,\alpha)\ \&\ B(\alpha,Q')}(\exists I)}{\exists Y.\exists \alpha.\ A(X,\alpha)\ \&\ B(\alpha,Y)}(\exists I)}{\exists Y.\exists \alpha.\ A(X,\alpha)\ \&\ B(\alpha,Y)}(\exists E)^{(2)}$$

and Σ_1 and Σ_2 are the definition of process $\forall X.\exists Y.A(X,Y)$ and $\forall P.\exists Q.B(P,Q)$.
This is a typical proof style to define a composition of two functions. 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, $\dot{\exists}$, introduced by the author and S. Hayashi [16] or Hayashi's $\diamondsuit$ operator [11].
In the following, we focus on the problem of how to define a process (stream transformer) as a constructive proof.

3 Formulation of Streams

As explained in the previous section, a formula $\forall X : I_\sigma.\exists Y : I_\tau.A(X,Y)$ can be regarded as a specification of a stream transformer but we have not yet given the definition of stream types, I_σ, nor the rules for them. We must make clear what is the definition of the stream type I_σ and the semantics of quantification over I_σ.

3.1 Two Level Stream Types

A stream can be viewed at least in three ways: an infinite list, an infinite process, and an output sequence of an infinite process, namely, a total function on natural numbers. The formal theories of lazy functional programming such as [6] and [12] can be regarded as the theories of concurrent programming based on the first two points of view on streams. Our system uses a lazy typed lambda calculus as the underling programming languages and has lazy types as *computational stream types*. Computational stream types are only used as the type system for the underlying typed programming language. In proving specifications of stream transformers, we use *logical stream types* which is based on the third point of view on streams. In other words, we have two kinds of streams: computational streams at the programming language level, and logical streams at the logical reasoning level. We denote a computational stream type C_σ and a logical stream type I_σ. The following is the basic rules for computational stream types which are actually almost like those listed

in [6]. We confuse the meaning of the infinite list constructor, (::), and will use this as an infinite cartesian product constructor.

$$C_\sigma = \sigma \times C_\sigma$$

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

$$\frac{\Gamma \vdash M = N \ in \ \sigma \quad \Gamma \vdash S = T \ in \ C_\sigma}{\Gamma \vdash (M :: S) = (N :: T) \ in \ C_\sigma}$$

$$\frac{\Gamma \vdash (M :: S) = (N :: T) \ in \ C_\sigma}{\Gamma \vdash M = N \ in \ \sigma}$$

$$\frac{\Gamma \vdash (M :: S) = (N :: T) \ in \ C_\sigma}{\Gamma \vdash S = T \ in \ C_\sigma}$$

$$\frac{\Gamma, z : C_\sigma \vdash M : C_\sigma}{\Gamma \vdash \nu z.\ M : C_\sigma}$$

ν is the fixed point operator only used for describing a stream as an infinite process (infinite loop program). The typing rules for ν are defined as expected.

$$\frac{\Gamma \vdash M : C_\sigma}{\Gamma \vdash hd(M) : \sigma} \qquad \frac{\Gamma \vdash M : C_\sigma \qquad \Gamma \vdash n : nat}{\Gamma \vdash tl^n(M) : C_\sigma}$$

$$\frac{\Gamma \vdash X : C_\sigma}{\Gamma \vdash X = (hd(X) :: tl(X)) \ in \ C_\sigma}$$

$$\frac{\Gamma \vdash (M :: S) : C_\sigma}{\Gamma \vdash hd((M :: S)) = M \ in \ \sigma} \qquad \frac{\Gamma \vdash (M :: S) : C_\sigma}{\Gamma \vdash tl((M :: S)) = S \ in \ C_\sigma}$$

$$\frac{\Gamma, n : nat, tl^n(S) = tl^n(T) \ in \ C_\sigma \vdash S = T \ in \ \sigma^n \times C_\sigma}{\Gamma \vdash S = T \ in \ C_\sigma}(EXTE)_1$$

$$\frac{\Gamma, n : nat \vdash hd(tl^n(S)) = hd(tl^n(T)) \ in \ \sigma}{\Gamma \vdash S = T \ in \ C_\sigma}(EXTE)_2$$

$$\frac{\Gamma \vdash M_z = N_w[z] \ in \ C_\sigma}{\Gamma \vdash \nu z.\ M_z = \nu w.\ N_w \ in \ C_\sigma}(INTE)$$

Before giving the definition of logical stream types, note that the type, $nat \rightarrow \sigma$, is isomorphic C_σ, namely,

Proposition 1: *Let σ be any type, then Let $\varphi : (nat \rightarrow \sigma) \rightarrow C_\sigma$ be $\varphi(M) \equiv ap(\nu z.\lambda n.\ (M(n) :: z(n+1)), 0)$ for arbitrary $M : nat \rightarrow \sigma$, and let $\psi(N) \equiv \lambda n.\ hd(tl^n(N))$*

for arbitrary $N : C_\sigma$. Then, the following holds:
(1) For arbitrary $M : nat \to \sigma$. $\varphi(M) : C_\sigma$ and $\psi(\varphi(M)) = M$ in I_σ;
(2) For arbitrary $N : C_\sigma$, $\psi(N) : I_\sigma$ and $\varphi(\psi(N)) = N$ in C_σ where $=$ is the extensional equality.

A logical stream type, I_σ, is defined to be $nat \rightarrowtail \sigma$, and is regarded as a subtype of C_σ through the isomorphism.

$$\frac{\Gamma \vdash M : nat \to \sigma}{\Gamma \vdash M : I_\sigma} \qquad \frac{\Gamma \vdash M : I_\sigma}{\Gamma \vdash M : nat \to \sigma}$$

This means that any (total) function on the natural number type *nat* definable in the underlying programming language is regarded as a stream. A similar idea is formulated with regard to formulas:

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

The equality between streams is extensional. That is

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

The following rule characterizes a kind of continuity of stream transformers and is used for justifying $(MPST)$ rule given later.

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

where $A(n, \overline{X})$ is a rank 0 formula[11].

A logical stream also has, *hd*, *tl* and (::), which simulate those accompanied with C_σ:

$$\begin{aligned} hd(X) &\stackrel{\text{def}}{=} X(0) \text{ for } X : I_\sigma \\ tl^n(X) &\stackrel{\text{def}}{=} \lambda n.X(m+n) \text{ for } X : I_\sigma \\ (M :: S)(0) &\stackrel{\text{def}}{=} M \\ (M :: S)(n) &\stackrel{\text{def}}{=} S(n-1) \text{ for } n > 0 \end{aligned}$$

Note that $X(n) = hd(tl^n(X))$ for arbitrary $X : I_\sigma$ and $n : nat$. All the rules for *hd*, *tl* and (::) in computational streams also hold for these defined functions and the constructor for logical streams.

3.2 Quantification over Logical Stream Types

There is a difficulty in defining the meaning of quantification over (logical) stream types. The standard intuitionistic interpretation of, say, existential quantification over a type, σ, $\exists x : \sigma.A(x)$ is that "we can explicitly give the object, a, of type σ such that $A(a)$ holds". However, as a stream is a partial object we can only give an approximation of the complete object at any moment. Therefore we need to extend the familiar interpretation of quantification over types. In fact, Brouwer's theory of choice sequences [1] in intuitionism provides us with the meaning of quantification over infinite sequences. As we defined the logical streams as sequences, we can borrow the Brouwer's theory for the meaning of quantification.

There are two principles in Brouwer's theory, the principle of open data and the principle of function continuity.
The principle of open data, which informally states that for independent sequences any property which can be asserted must depend on initial segments of those sequences only, gives the meaning of the quantification of type, $\forall X.\exists y.A(X, y)$. That is, for an arbitrary sequence, X, there is a suitable initial finite segment, X_0, of X such that $\exists y.\ A(X_0, y)$ holds.
The principle of function continuity gives the meaning of the quantification of type, $\forall X.\exists Y.A(X, Y)$. Assume the case of natural number streams (total functions between natural number types). The function continuity is stated 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.

This semantics meets our intuition on functions of streams and stream transformer quite well. $\forall X : I_\sigma.\exists y : \tau.A(X, y)$ represents a function on streams over σ, but we would hardly ever try to define a function which returns a value after taking *all* the elements of an input stream. Also, we would expect a stream transformer, $\forall X : I_\sigma.\exists Y : I_\tau.A(X, Y)$, calculate the elements of the output stream, Y, gradually by taking finitely many elements of the input stream, X, at any step of the calculation.

Note that this semantics also meets the proof method used in [2]: To prove a property $P(X)$ on a stream X, we first prove P for an initial finite subsequence, X_0, of X ($\vdash P(X_0)$) and define $\vdash P(X)$ to be $\lim_{X_0 - X} P(X_0)$.

4 Structural Induction on Logical Streams

As streams can be regarded as infinite lists, we would expect to extend the familiar structural induction on (finite) lists to streams. However, a naive extension of the structural induction on finite lists does not work well. If we allow the rule below,

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

the following wrong theorem can be proved:

WrongTheorem: $\forall X : I_{nat}.\ B(X)$ *where* $B(X) \stackrel{\text{def}}{=} \exists n : nat.\ X(n) = 100$.
Proof: By (SI) *on* $X : I_{nat}$. *Assume* $B(tl(X))$. *Then, there is a natural number* k *such that* $tl(X)(k) = X(k+1) = 100$. *Then* $B(X)$.∎

This proof would correspond to the following meaningless program:

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

This is because the naive extension of the structural rule on finite lists does not maintain the continuity of the function on streams. On the other hand, as suggested by Hagino [12], an infinite list type can be obtained as the dual of a finite list type, and the dual of the structural induction on finite lists is formulated as a coinduction rule. However, as opposed to category theory, dual notions are rather difficult to formulate in the natural deduction formalism. We can formulate a coinduction as the axiom of the largest fixed-point induction as in [7], but the meaning of the rule is rather difficult. Therefore, we will take different approach.

4.1 Mapstream Functions as Stream Transformers

Recall that the motivation of pursuing a kind of structural induction on streams is to define stream transformers as proofs, and stream transformers can be realized as Burge's mapstream functions. A schema of mapstream functions is described in typed lambda calculus as follows:

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

If we give the procedures M and N, we obtain a mapstream function. Note that these procedures can be interpreted as follows:

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

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

This suggests that if a way to define M, N, and P as proof procedures is given, one can define stream transformers as constructive proofs.

4.2 A Problem of Empty Stream

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

$$\begin{aligned} flt_a &\equiv \lambda X.\ if\ (a|hd(X))\ then\ flt_a(tl(X))\ else\ (hd(X) :: flt_a(tl(X))) \\ &\equiv \lambda X.((M\ X) :: (((P\ M)\ N)(N X))) \end{aligned}$$

where $(a|hd(X))$ is true when $hd(X)$ can be divided by a (a natural number) and

$$\begin{aligned} M &\equiv \lambda X.\ if\ (a|hd(X))\ then\ M(tl(X))\ else\ hd(X) \\ N &\equiv \lambda X.\ if\ (a|hd(X))\ then\ N(tl(X))\ else\ tl(X) \end{aligned}$$

For example, $flt_5((5 :: 5 :: 5 :: 5 :: \ldots))$ is an empty sequence. This is because in the execution of $flt_5((5 :: 5 :: \ldots))$ $M(5 :: 5 :: 5 :: 5 :: \ldots)$ does not terminate. This contradicts the principle of open data explained in the previous subsection. To handle such a case, we introduce the notion of complete stream. The idea is to regard flt_5, for example, always generating some elements even if the input stream is $(5 :: 5 :: \ldots)$.

Def. 1: Complete types
Let σ be any type other than a stream type, then $\sigma_\perp$ denotes a type σ together with the bottom element $\perp_\sigma$ (often denoted just $\perp$) and it is called a *complete type*.

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

flt_5 is easily modified to a function from C_{nat} to $C_{nat_\perp}$, and then $flt_5((5 :: 5 :: \ldots))$ will be $(\perp :: \perp :: \ldots)$ which is practically a empty stream.

4.3 The $(MPST)$ rule

Based on the observations in the previous sections, we introduce a rule $(MPST)$ 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 a fairly large class of specifications of stream transformers as will be demonstrated later.
The rule is as follows:

$$\frac{\begin{array}{l}(a)\ \forall X : I_\sigma.\exists a : \tau.\ M(X,a) \\ (b)\ \forall X : I_\sigma.\forall a : \tau.\ \forall S : I_\tau.\ (M(X,a) \Rightarrow A(0,X,(a :: S))) \\ (c)\ \exists f : I_\sigma \rightarrow I_\sigma.\ \forall X : I_\sigma.\forall Y : I_\tau.\forall n : nat. \\ \qquad\qquad (A(n,f(X),tl(Y)) \Rightarrow A(n+1,X,Y))\end{array}}{\forall X : I_\sigma.\exists Y : I_\tau.\forall n : nat.\ A(n,X,Y)}(MPST)$$

where M is a suitable predicate and $A(n,X,Y)$ must be a rank 0 formula [11]. We do not give the definition of rank 0 formulas, but the intention is that we should not expect

to extract any computational meaning from $A(n, X, Y)$ part. This restriction comes from purely technical reason, but does not degenerate the expressive power of the rule from the practical point of view because we usually need only to define a stream transformer program but not the verification code corresponding to $A(n, X, Y)$ part. We can easily extend the rule to the multi input stream version.

The intuitive meaning of $(MPST)$ is as follows. As explained in 4.1, a mapstream function is defined when M and N procedure are given. (a) is the specification of the M procedure, f_M, and (b) means that f_M actually generates the right elements of the output stream. The N procedure, f_N, is defined as the value of existentially quantified variable, f, in (c). (c) together with (b) intuitively means the following: for $X : I_\sigma$ (input stream) and $Y : I_\tau$ (output stream), let us call a pair, $(f_N^n(X), tl^n(Y))$, the *nth f_N-descendant of* (X, Y). Then, for arbitrary $n : nat$, $A(n, X, Y)$ speaks about nth f_N descendant of (X, Y), and $A(n, f_N(X), tl(Y))$ actually speaks about $n + 1$th f_N-descendant of (X, Y).
If f_N is a stream transformer, this means that the process (stream transformer) defined by $(MPST)$ generates another processes dynamically.

Note that, as we must give a suitable formula, M, to prove the conclusion, $(MPST)$ is essentially a second order rule.

5 The Formal System

This section presents the rest of the formal account of our system briefly. First of all, the non-deterministic λ-calculus is defined as the underlying programming language. The calculus has a special constant called *coin flipper*, to simulate nondeterminacy, and computational stream types. Secondly, several rules of inference which have not been explained will be presented. Finally, the realizability interpretation of the system is defined, and this gives the program extraction algorithm from proofs.

5.1 Non-deterministic λ-calculus

The non-deterministic λ-calculus is a typed concurrent calculus based on parallel reduction. The core part is almost the same as that given in [15]. It has natural numbers, booleans (T and F), L and R as constants. Individual variables, lambda abstractions, application, sequences of terms ($(M_1, ..., M_n)$ where M_i are terms), *if-then-else*, and a fixed point operator (μ) are used as terms and program constructs. The 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 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, computational stream types and a special term called *coin flipper* is introduced to describe concurrent computation of streams.

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

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

That is, $\bullet$ is L or R in a nondeterministic way when it is executed. This is like flipping a coin, or can be regarded as hiding some particular decision procedure whose execution may not always be explained by the reduction mechanism.

$\bullet$ is regarded as an element of $\mathbf{2}^+$, a super type of $\mathbf{2}$. The elements of $\mathbf{2}$ have been used to describe the decision procedure of *if-then-else* programs in the program extraction from constructive proofs in [15] as $if\ T = L\ then\ M\ else\ N$. Nondeterminacy arises when T is replaced by $\bullet$. The intentional semantics of $\bullet$ is *undefined*. The type $\mathbf{2}^\dagger$ will be used instead of $\mathbf{2}$ in this paper with the following typing rules:

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

5.2 Rules of Inference

(1) Logical Rules
The rules for logical connectives and quantifiers are those of first order intuitionistic natural deduction with mathematical induction. See [16] for the complete account of the logical rules.

(2) Rules for Nondeterminacy

$$\bullet = L \vee \bullet = R \qquad \frac{A \quad A}{A}(NonDet)$$

$(NonDet)$ is actually a derived rule: This is obtained by proving A by divide and conquer on $\top \vee \top$. $(NonDet)$ 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) Auxiliary Rules

$$\frac{M : \sigma \to \sigma \quad a : \sigma \quad n : nat}{ap(M^n, a) : \sigma}(exp) \qquad \frac{f : \sigma_1 \to \tau_1 \quad g : \sigma_2 \to \tau_2}{f \times g : \sigma_1 \times \sigma_2 \to \tau_1 \times \tau_2}$$

5.3 Realizability Interpretation

The realizability defined in this section is a variant of **q**-realizability, and is obtained by modifying the realizability given in [15].

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

Def. 3: 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. 4: $type(A)$
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. 5: **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** $A \,\&\, \overline{b} : type(B)) \vee (z = R \,\&\, B \,\&\, \overline{b}$ **q** $B \,\&\, \overline{a} : type(A))$ provided that A and B are distinct or $A = B$ with A and B 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 $\overline{a} : type(A)$.*

Theorem: *Soundness of realizability:*
Assume that A is a formula. If A is proved, then there is a term, T, such that T **q** *A can be proved and $FV(T) \subset FV(A)$.*

The proof of the theorem gives the algorithm of program extraction from constructive proofs. The program extracted from $(NonDet)$ is $if\ \bullet = L\ then\ M\ else\ N$ where M and N are the program extracted from the subproofs of two premises. From a proof by $(MPST)$, the program $\lambda X.\lambda m.ap(f_M, f_N^m(X))$ is extracted where f_M and f_N are as explained in section 4.3.

6 Examples

The basic programming technique with $(MPST)$ is demonstrated in this section. In the following, we write X_n for $X(n)$ when X is a stream.

6.1 Simple Examples

A process which doubles each element of the input natural number stream is defined as follows:

SPEC 1: $\forall X : I_{nat}.\exists Y : I_{nat}.\forall n : nat.\ Y_n = 2 \cdot X_n$

Proof: The proof is continued by $(MPST)$. Let $M(X, a) \stackrel{\text{def}}{=} a = 2 \cdot hd(X)$, and (a) and (b) are easily proved. (c) is proved by letting $f = \lambda X.\ tl(X)$. ∎

The program extracted from the proof is $\lambda X.\lambda m.\ 2 \cdot hd(tl^m(X))$ which is, by the isomorphism φ, extensionally equal to $\nu z.\lambda X.\ (2 \cdot hd(X) :: z(tl(X)))$.

A process which takes two elements at once from the input stream and outputs the some of them is defined as follows:

SPEC 2: $\forall X : I_\sigma.\exists Y : I_\sigma.\forall n : nat.\ Y_n = X_{2\cdot n} + X_{2\cdot n+1}$

Proof: By $(MPST)$. Let $M(X, a) \stackrel{\text{def}}{=} a = hd(X) + hd(tl(X))$ and (a) and (b) are easily proved. (c) is proved by letting $f \stackrel{\text{def}}{=} \lambda X.\ tl^2(X)$. ∎

The program extracted from the proof is $\lambda X.\lambda m.\ hd(tl^{2\cdot m}(X)) + hd(tl^{2\cdot m+1}(X))$ which is extensionally equal to $\nu z.\lambda X.\ (hd(X) + hd(tl(X)) :: z(tl^2(X)))$.

6.2 Parameterized Processes and Complete Stream Types

A filter process defined below removes all the elements of the input stream, X, which can be divided by a fixed natural number p. This process is an example of parameterized processes. The definition uses the complete stream type and the rank 0 formula technique.

SPEC 3: $\forall p : nat.\forall X : I_{nat}.\exists Y : I_{nat_\perp}.\forall n : nat.\ \Diamond A(p, n, X, Y)$
where $A(p, n, X, Y) \stackrel{\text{def}}{=} ((p|X_n)\ \&\ Y_n = \perp) \vee (\neg(p|X_n)\ \&\ Y_n = X_n)$

Proof: Let $p : nat$ be arbitrary, and $\forall X.\exists Y.\forall n.\ \Diamond A(p, n, X, Y)$ will be proved by $(MPST)$. Let $M(X, a) \stackrel{\text{def}}{=} ((p|hd(X))\ \&\ a = \perp) \vee (\neg(p|hd(X))\ \&\ a = hd(X))$. (a) is proved by divide and conquer with regard to $(p|hd(X)) \vee \neg(p|hd(X))$. (b) is proved easily, and (c) is proved by letting $f = \lambda X.\ tl(X)$. ∎

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 $f_N \stackrel{\text{def}}{=} \lambda X.\ tl(X)$. Precisely, $(p|hd(X))$ should be a decision procedure for $(p|hd(X))$.

6.3 Dynamic Invocation of Processes

The following example, a program which extracts only prime numbers in the input stream, is one of the typical examples of dynamic creation of new processes.

SPEC 4: $\forall X : I_{nat}.\exists Y : I_{nat_\perp}.\forall n : nat.\ \Diamond A(n, X, Y)$

$$\text{where } A(n, X, Y) \stackrel{\text{def}}{=} (PR(X_n)\ \&\ Y_n = X_n) \vee (\neg PR(X_n)\ \&\ Y_n = \perp)$$
$$\text{and } PR(m) \stackrel{\text{def}}{=} \forall n : nat.\ (2 \leq n < m \Rightarrow \neg(\exists d : nat.\ m = d \cdot n)).$$

Proof: By $(MPST)$. Let $M(X, a) \stackrel{\text{def}}{=} (PR(hd(X))\ \&\ a = hd(X)) \vee (\neg PR(hd(X))\ \&\ a = \perp)$. (a) is proved by divide and conquer with regard to $PR(hd(X)) \vee \neg PR(hd(X))$. (b) is proved easily. The proof of (c) is a little complex.
Let $f \equiv \lambda X.\ if\ PR(hd(X))\ then\ flt(hd(X), tl(X))\ else\ tl(X)$ where $flt(p, X)$ is the filter process developed in the previous subsection. Then, for arbitrary $X : I_{nat_\perp}$ and $n : nat$ the following hold: 1. $PR(f(X)_n) \Rightarrow PR(tl(X)_n)$; 2. $\neg PR(f(X)_n) \Rightarrow \neg PR(tl(X)_n)$; 3. $PR(f(X)_n) \Rightarrow f(X)_n = tl(X)_n$. These can be proved by divide and conquer on $PR(hd(X)) \vee \neg PR(hd(X))$. Then, as $A(n, f(X), tl(Y)) \equiv (PR(f(X)_n)\ \&\ Y_{n+1} = f(X)_n) \vee (\neg PR(f(X)_n)\ \&\ Y_{n+1} = \perp)$, $A(n + 1, X, Y)$ holds. Then, (c) is proved. ∎

The program extracted from this proof is $\lambda X.\lambda m.\ ap(f_M, f_N^m(X))$ where
$f_M \stackrel{\text{def}}{=} \lambda X.\ if\ PR(hd(X))\ then\ hd(X)\ else\ \perp$
and $f_N \stackrel{\text{def}}{=} \lambda X.\ if\ PR(hd(X))\ then\ flt(hd(X), tl(X))\ else\ tl(X)$.
This program performs load distribution is the following way. When a prime number, p, is found in the input stream, X, this program invokes a filter process, flt_p making X as the input stream of flt_p and take the output stream of flt_p as the new input stream.

6.4 Nondeterminacy

The stream merge operation is a typical example of nondeterminacy which can also be defined by $(MPST)$. However, because of the condition (c) on $A(n, (X, Y), Z)$, our

specification is weaker than that of the merge operation. It specifies that all the elements of the output stream come from the input streams and nothing else. The rest of the criteria for a merge operation, namely, all the elements of the input streams occur in the output stream without repetition and loss, depends on how the formula M is defined in (a) and how f is defined for (c) in the premises of $(MPST)$.

SPEC 5: $\forall(X,Y) : I_{\sigma,\sigma}.\exists Z : I_\sigma.\forall n : nat.\ \Diamond A(n,(X,Y),Z)$
where $A(n,(X,Y),Z) \stackrel{\text{def}}{=} (\exists m : nat.\ Z_n = X_m) \vee\ (\exists l : nat.\ Z_n = Y_l)$

Proof: By $(MPST)$. Let $M((X,Y),a) \stackrel{\text{def}}{=} a = hd(X)$, then the proofs of (a) and (b) are straightforward. (c) is proved as follows: Let $(X,Y) : I_{\sigma,\sigma}$, $Z : I_\sigma$ and $n : nat$ be arbitrary. Then, $A(n,(tl(X),Y),tl(Z)) \Leftrightarrow (\exists m.\ tl(Z)_n = tl(X)_m) \vee (\exists l.\ tl(Z)_n = Y_l) \Leftrightarrow (\exists m.\ Z_{n+1} = X_{m+1}) \vee (\exists l.\ Z_{n+1} = Y_l)$. Hence, if $A(n,(tl(X),Y),tl(Z))$, then $(\exists m'.\ Z_{n+1} = X_{m'}) \vee (\exists l.\ Z_{n+1} = Y_l) \equiv A(n+1,(X,Y),Z)$ holds. Similarly, $A(n,(Y,tl(X)),tl(Z)) \Rightarrow A(n+1,(X,Y),Z)$ is proved. Then, two distinct proofs of (c) are given. Then, by $(NonDet)$, (c) 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 $f_N \stackrel{\text{def}}{=} \lambda(X,Y).\ if\ \bullet - L\ then\ (tl(X),Y)\ else\ (Y,tl(X))$.

7 Conclusion and Future Works

An extension of constructive programming to stream based concurrent programming was proposed in this paper. The system has lazy types at the level of programming language and logical stream types, which are types of sequences viewed as streams, at the level of logic. This two level formulation of streams enables to formulate a purely natural deduction style of structural induction on streams $(MPST)$ in which concurrent processes (stream transformers) are defined as proofs. Also, nondeterminacy was introduced at the level of logic using the inherent nondeterminacy of proof normalization in intuitionistic logic.
For the future work, as seen in the example of a merger process, the side condition for $(MPST)$ should be relaxed to handle larger varieties of concurrent processes.

References

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

[2] G. Kahn and D. B. MacQueen. The Semantics of a Simple Language for Parallel Programming. In *IFIP Congress 74.* Noth-Holland, 1974.

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

[4] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50, 1987. North–Holland.

[5] M. Sato. Quty: A Concurrent Language Based on Logic and Function. In *Fourth International Conference on Logic Programming*, pages 1034–1056. The MIT Press, 1987.

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

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

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

[9] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[10] S. Goto. Concurrency in proof normalization and logic programming. In *Internatioinal Joint Conference on Artificial Intelligence '85*, 1985.

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

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

[13] 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.

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

[15] Y. Takayama. **QPC**^{2}: A Second Order Logic for Higher Order Programming. TR 539, ICOT, April 1990.

[16] Y. Takayama and S. Hayashi. Extended Projection Method and Realizability Interpretation. TR 573, ICOT, July 1990.

Appendix 1: Proof of Proposition 1

Proposition : *Let σ be any type, 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 $M : nat \to \sigma$, and let $\psi(N) \equiv \lambda n.\, hd(tl^n(N))$ for arbitrary $N : C_\sigma$. Then, the following holds:*
(1) For arbitrary $M : nat \to \sigma$, $\varphi(M) : C_\sigma$ and $\psi(\varphi(M)) = M$ in I_σ;
(2) For arbitrary $N : C_\sigma$, $\psi(N) : I_\sigma$ and $\varphi(\psi(N)) = N$ in C_σ where $=$ is the extensional equality.

Proof: (1) Let $M : nat \to \sigma$ be arbitrary. $\psi(\varphi(M)) = \psi(ap(\nu z.\lambda n.(M(n) :: z(n+1)), 0)) = \lambda n.hd(tl^n(ap(L, 0)))$ where $L \stackrel{\text{def}}{=} \nu z.\lambda n.(M(n) :: z(n+1))$. Then, by the lemma below, the last term in the equality is equal to $\lambda n.hd(ap(L, n))$. As $ap(L, n) = (M(n) :: L(n+1))$, this is equal to $\lambda n.M(n)$. So by the η-conversion, $\psi(\varphi(M)) = M$.

Lemma: $\forall n.tl^n(ap(L, 0)) = ap(L, n)$
Proof of lemma: By mathematical induction on n. Base case is trivial. Assume $L(n) = tl^n(L(0))$. As $L(n) = (M(n) :: L(n+1))$ and $tl^n(L(0)) = (hd(tl^n(L(0))) :: tl^{n+1}(L(0)))$, $L(n+1) = tl^{n+1}(L(0))$. ∎

(2) Let $N : C_\sigma$ be arbitrary. $\varphi(\psi(N)) = ap(\nu z.\lambda n.(ap(\lambda m.hd(tl^m(N)), n) :: z(n+1)), 0)$. Let $L \stackrel{\text{def}}{=} \nu z.\lambda n.(ap(\lambda m.hd(tl^m(N)), n) :: z(n+1))$. Then, we will prove $L(0) = N$ by $(EXTE)_2$. As $L(n) = (hd(tl^n(N)) :: L(n+1))$, it suffices to prove the following:

$$\forall n.L(n) = tl^n(L(0))$$

We prove this by mathematical induction on n. The base case is trivial. Assume that $L(n) = tl^n(L(0))$. Then, as $L(n) = (hd(tl^{n+1}(N)) :: L(n+1))$ and $tl^n(L(0)) = (hd(tl^n(L(0))) :: tl^{n+1}(L(0)))$, $L(n+1) = tl^{n+1}(L(0))$. ∎

Appendix 2: Soundness Proof of the Realizability Interpretation

Theorem: *Assume that A is a formula. If A is proved, then there is a term, T, such that T $\mathbf{q}$ A can be proved and $FV(T) \subset FV(A)$.*

Proof: By induction on the construction of the proof of A. We prove here for the cases that the last rules in the proofs are (ST) , $(NonDet)$ and $(MPST)$. The remainder part of the proof is rather standard.
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, M, such that the following proof can be constructed from Σ:

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

By the definition of $\mathbf{q}$-realizability and equality rules,

$$\begin{aligned}
M\ \mathbf{q}\ \forall n.\exists x.\ A(n,x) &\equiv \forall n.\ M(n)\ \mathbf{q}\ \exists x.\ A(n,x) \\
&\equiv \forall n.\ tseq(M(n))\ \mathbf{q}\ A(n,\mathbf{p}_0(M(n))) \\
&= \forall n.\ ap(\lambda n.\ tseq(M(n)),n)\ \mathbf{q}\ A(n,ap(\lambda n.\ \mathbf{p}_0(M(n)),n)) \\
&\equiv \lambda n.\ tseq(M(n))\ \mathbf{q}\ \forall n.\ A(n,ap(\lambda n.\ \mathbf{p}_0(M(n)),n)) \\
&\equiv (\lambda n.\ \mathbf{p}_0(M(n)),\lambda n.\ tseq(M(n)))\ \mathbf{q}\ \exists Y.\forall n.\ A(n,Y(n)) \\
&= \lambda n.\ M(n)\ \mathbf{q}\ \exists Y.\forall n.\ A(n,Y(n))
\end{aligned}$$

where $\mathbf{p}_0$ is the 0th projection function and $tseq((x_0,x_1,\cdots,x_n)) \equiv (x_1,\cdots,x_n)$.
By η-rule, $\lambda n.\ M(n) = M$. Therefore,

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

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

$$\frac{\begin{array}{cc}\Sigma_1 & \Sigma_2 \\ A & A\end{array}}{A}(NonDet)$$

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

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

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

$$\begin{array}{cc} \Sigma_1' & \Sigma_2' \\ M\ \mathbf{q}\ A & N\ \mathbf{q}\ A \end{array}$$

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

$$\dfrac{\bullet = L \vee \bullet = R \qquad \dfrac{\dfrac{[\bullet = L] \quad \begin{array}{cc} \Sigma_3 & \Sigma_1' \\ T = M & M\ \mathbf{q}\ A \end{array}}{T\ \mathbf{q}\ A}(-E) \qquad \dfrac{[\bullet = R] \quad \begin{array}{cc} \Sigma_4 & \Sigma_2' \\ T = N & N\ \mathbf{q}\ A \end{array}}{T\ \mathbf{q}\ A}(=E)}{T\ \mathbf{q}\ A}(\vee E)$$

Case $(MPST)$: Assume that there is a proof by $(MPST)$. 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, $\Sigma'_{(a)}$ and $\Sigma'_{(c)}$, of $f_M\ \mathbf{q}\ \forall X.\exists a.\ M(X,a)$ and $f_N\ \mathbf{q}\ \exists f.\forall X.\forall Y.\forall n.\ (A(n, f(X), tl(Y)) \Rightarrow A(n+1, X, Y))$ for some f_M and f_N. As $A(n, X, Y)$ is a rank 0 formula, $\Sigma'_{(b)} = \Sigma_{(b)}$. Also, $f_M : I_\sigma \to (\tau \times type(M(X,a)))$ by proposition 3. By the definition of $\mathbf{q}$-realizability.
$f_N\ \mathbf{q}\ \exists f.\forall X.\forall Y.\forall n.\ (A(n, f(X), tl(Y)) \Rightarrow A(n+1, X, Y))$
$\equiv f_N : I_\sigma \to I_\sigma\ \&\ \forall X.\forall Y.\forall n.\ (A(n, f_N(X), tl(Y)) \Rightarrow A(n+1, X, Y))$. Using them, the conclusion of $(MPST)$ can be proved without $(MPST)$ as follows:

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

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

where $\exists Y.\forall m.\forall S.\ A(0, f_N^m(X), (Y(m) :: S))$ is proved by Σ_0 from $[X]^{(1)}$, and $\exists Y.\forall n.\ A(n, X, Y)$ is proved by Σ_1 from $[X]^{(1)}[Y]^{(2)}[\forall m.\forall S.\ A(0, f_N^m(X), (Y(m) :: S))]^{(2)}$.

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

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

$$\exists Y.\forall m.\forall S.\ A(0, f_N^m(X), (Y(m) :: S))$$

$\Sigma_{00} \stackrel{\text{def}}{=}$

$$\dfrac{\dfrac{\dfrac{[M(f_N^m(X), a)]^{(4)} \quad M(f_N^m(X), a) \Rightarrow A(0, f_N^m(X), (a :: S))}{A(0, f_N^m(X), (a :: S))}(\Rightarrow E) \quad [X]^{(1)}[m]^{(3)}[a]^{(4)}[S]^{(5)} \quad \Sigma_{000}}{\forall S.\ A(0, f_N^m(X), (a :: S))}(\forall I)^{(5)}}{\exists a.\forall S.\ A(0, f_N^m(X), (a :: S))}(\exists I)$$

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

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

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

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

where

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

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

$$\dfrac{[X]^{(1)}[Y]^{(2)}[n]^{(6)} \quad \dfrac{\dfrac{f_N \quad tl}{f_N \times tl} \quad \forall X.\forall Y.\forall n.\ (A(n,f_N(X),tl(Y)) \Rightarrow A(n+1,X,Y))}{\forall X.\forall Y.\forall n.\ (A(0,f_N^n(X),tl^n(Y)) \Rightarrow A(n,X,Y))}(R)}{A(0,f_N^n(X),tl^n(Y)) \Rightarrow A(n,X,Y)}(\forall E)$$

By the induction hypothesis, there is a term, T, such that a proof of $T\ \mathbf{q}\ \forall X.\exists Y.\forall n.\ A(n,X,Y)$ can be constructed from the proof, Π. The term, T, is $\lambda X.\lambda m.\ \mathbf{p}_0(ap(f_M, f_N^m(X)))$. ∎