PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

Defining Concurrent Processes Constructively *

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

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 $(MPST)$ rule which is a kind of structural induction on streams. The $(MPST)$ rule is based on the idea of largest fixed point inductions, but the formulation of the rule is quite different and it allows to define a concurrent process as a Burge's mapstream function with a good intuition on computation. 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., [C+86, HN89]. 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 [TvD88] or with Curry-Howard correspondence of types and formulas [How80]. This programming methodology will be referred to as *constructive programming* [SK90] 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 [Got85]. For example, QJ [Sat87] 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 Quty.

Linear Logic [Gir87] 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. The logic was obtained by refining logical connectives of traditional intuitionistic or classical logic to introduce drastically new connectives with the meaning of parallel execution. In Linear Logic, formulas are regarded as processes or resources and every rule of inference defines the behavior of a concurrent operation. Linear Logic resembles Milner's SCCS [Mil89] in this respect.

We take intermediate approach between QJ and Linear Logic in the sense of 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 based on intuitionism 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 [KM74]. We introduce stream types and quantification over stream types. A formula is regarded as a specification of a process when it is a universally or an existentially quantified 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 heart of our extended system: With $(MPST)$, stream transformers can be defined as Burge's mapstream functions [Bur75].

*This work was supported by ICOT as a joint research project on theorem proving and its application.

T. Hagino [Hag87] gave a clear categorical formalization of stream types (infinite list types or lazy types) whose canonical elements are given by a schema of map-stream functions, but relation between his formulation and logic is not investigated. N. Mendler and others [PL86] introduced lazy types and the type checking rules for them into an intuitionistic type theory preserving the propositions-as-types principle in the sense that an empty type can exist even in the extended type theory. However, they do not give sufficient rules of inference for proving specification of stream handling programs. Reasoning about stream transformer can be handled with a largest fixed point induction as was demonstrated by P. Dybjer and H. P. Sander [DS89]. However, their system is designed as a program verification system not as a constructive programming system. Although q-realizability interpretation for program extraction can be defined for the coinduction rule [KT91], the rule seems rather difficult to use for proving specifications. The reason is that the coinduction rule deeply depends on the notion of bisimulation, so that in the proof procedure one must find a stronger logical relation included in the more general logical relation and that is not always an easy task.

The $(MPST)$ rule is based on a similar idea to the coinduction rule: one must find a new logical relation and a new function to prove the conclusion. However, what one must find has a clear intuitive meaning as the components of a concurrent process. Therefore, the $(MPST)$ rule shows an intuitive guideline on how to construct a concurrent process.

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 $(MPST)$ 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. $M_{\overline{x}}[\overline{N}]$ 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 $M(N)$. $M^n(N)$ denotes $\underbrace{M(\cdots M(}_{n} N)\cdots)$.

2 Specifying Concurrent Systems in Logic

The model of concurrent computation 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 already existing processes. A process is regarded as a transformer (stream transformer) of input streams to an output stream, and it is specified by the $\forall \overline{X} : I_{\sigma_1,\cdots,\sigma_n}.\ \exists Y : I_\tau.\ A(\overline{X},Y)$ type of formula where I_σ denotes the type of streams over the type σ, but its definition will be given later. $I_{\sigma_1,\cdots,\sigma_n}$ 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.

The combination of two processes, $\forall X.\exists Y.\ A(X,Y)$ and $\forall P.\exists Q.\ B(P,Q)$, by linking the stream Y and P is described by the following proof procedure:

$$\dfrac{\dfrac{\dfrac{\Sigma_1}{\forall X.\exists Y.\ A(X,Y)}(\forall E)}{\exists Y.\ A(X,Y)} \quad \Pi_0}{\dfrac{\exists Y.\exists\alpha.\ A(X,\alpha)\ \&\ B(\alpha,Y)}{\forall X.\exists Y.\exists\alpha.\ A(X,\alpha)\ \&\ B(\alpha,Y)}(\forall I)}(\exists E)^{(1)}$$

where $\Pi_0 \stackrel{\text{def}}{=}$

$$\dfrac{\dfrac{\dfrac{\Sigma_2}{\forall P.\exists Q.\ B(P,Q)}(\forall E)}{\exists Q.\ B(Y',Q)} \quad \Pi_1}{\exists Y.\exists\alpha.\ A(X,\alpha)\ \&\ B(\alpha,Y)}(\exists E)^{(2)}$$

and $\Pi_1 \stackrel{\text{def}}{=}$

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

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.

All these things just realize the idea that functions can be viewed as a special case of processes. In the following, we focus on the problem of how to define a process (stream transformer) as a constructive proof.

3 Formulation of Streams

We give in this section the definition of the stream types C_σ and I_σ, and consider 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 [PL86] and [Hag87] can be regarded as the theories of concurrent functional programming based on the first two points of view on streams. Our system uses a lazy typed lambda calculus as the underlying programming language and has lazy types as *computational stream types*. Computational stream types are only used as the type system for the underlying language. In proving specifications of stream transformers, we use *logical stream types* which are 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. The idea behind them is similar to that behind the lazy type rules in [PL86]. We confuse the meaning of the infinite list constructor, (::), and will use this also as an infinite cartesian product constructor. We abbreviate $M \stackrel{\sigma}{=} N$ for $M = N$ *in* σ in the following.

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

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

$$\frac{\Gamma \vdash (M :: S) \stackrel{C_\sigma}{=} (N :: T)}{\Gamma \vdash S \stackrel{C_\sigma}{=} T} \qquad \frac{\Gamma, z : T \vdash M : T}{\Gamma \vdash \nu z.\, M : T}$$

where T is C_σ or $\tau \to C_\sigma$.

ν is the fixed point operator only used for describing a stream as an infinite process (infinite loop program). The reduction rule for ν-terms is defined as expected. hd and tl are the primitive destructor functions on streams.

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

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

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

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

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

$$\frac{\Gamma \vdash M_z \stackrel{C_\sigma}{=} N_w[z]}{\Gamma \vdash \nu z.\, M_z \stackrel{C_\sigma}{=} \nu w.\, N_w}$$

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

Proposition 1: *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,*
(1) For arbitrary $M : nat \to \sigma$, $\varphi(M) : C_\sigma$ and $\psi(\varphi(M)) = M$ in $nat \to \sigma$;
(2) For arbitrary $N : C_\sigma$, $\psi(N) : nat \to \sigma$ and $\varphi(\psi(N)) = N$ in C_σ.

A logical stream type, I_σ, has the same elements as $nat \to \sigma$, but the elements are viewed differently, namely, viewed as streams:

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

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

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

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

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

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

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

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 intuition-

istic 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 [TvD88] in intuitionism provides us with the meaning of quantification over infinite sequences.

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 $f|X = Y$ is an abbreviation of $\forall x : nat.\ f(x :: X) = Y(x)$ and K is the class of functions that 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.

These principles meet our intuition of functions on streams and stream transformers very 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 [KM74]: 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 $\lim_{X_0 \to X} P(X_0)$.

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,

$$\frac{\Gamma, A(tl(X)) \vdash A(X)}{\Gamma \vdash \forall X : I_\sigma.\ A(X)}(SI)$$

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 uninteresting 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. Therefore, we need a drastically different idea in the case of infinite lists. One candidate is the coinduction rule (a largest fixed point induction) as in [DS89]: $(B \Rightarrow \Phi_P[B]) \Rightarrow (B \Rightarrow \nu P.\Phi)$ where $\nu P.\Phi$ denotes the largest fixed point of $P = \Phi.\ \forall X : I_\sigma.A(X)$ part will be described with $\nu P.\Phi$ type formulas, and one must find a suitable logical relation B to prove the conclusion. But searching B will not always be an easy task: we wish the searching task decomposed into more than one smaller tasks each of which has clear and intuitive meaning of computation. Therefore, we take another approach: the $(MPST)$ rule.

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, from the viewpoint of continuity, these procedures should be as follows:

$M \equiv$ "Fetch initial segment, X_0, of the input stream, X, to generate the first element of the output stream. "

$N \equiv$ "Prepare for fetching the next finite segment input stream interleaving, if necessary, other stream transformer between the original input stream and the input port.

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 filter program on natural number streams realized as a mapstream function:

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

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

$$M \equiv \lambda X.\ if\ (a|hd(X))\ then\ M(tl(X))\ else\ hd(X)$$
$$N \equiv \lambda X.\ if\ (a|hd(X))\ then\ N(tl(X))\ else\ tl(X)$$

For example, $flt_5((5 :: 5 :: 5 :: 5 :: \ldots))$ is an empty sequence because the evaluation of $M(5 :: 5 :: 5 :: 5 :: \ldots)$ does not terminate. This contradicts the principle of open data explained in 3.2. 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 an 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:

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

where M is a suitable predicate and $A(n,X,Y)$ must be a rank 0 formula [HN89]. We can easily extend the rule to the multiple input stream version. We do not give the precise definition of rank 0 formulas here, 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. The technical reason for the side condition of $(MPST)$ is as follows: $(MPST)$ is in fact a derived rule with (ST) and (CON), so that q-realizability interpretation defined in the next section is carried out using the interpretation of those rules. The difficulty resides in the interpretation of the (CON) rule, but if we restrict the formula $A(n,\overline{X})$ in (CON) to be rank 0, the interpretation is trivial. This condition corresponds to to side condition of $(MPST)$.

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 certainly 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 formalization of our system briefly.

5.1 Non-deterministic λ-calculus

The non-deterministic λ-calculus is a typed concurrent calculus based on parallel reduction and this is used as the underlying programming language. The core part is almost the same as that given in [Tak91]. 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 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 $\to$ (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 streams, computational stream types and a special term called *coin flipper* is introduced to describe concurrent computation of streams. For the reduction strategy, ν-

terms in section 3.1 are lazily evaluated.

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, $\bullet$ reduces to L or R in a nondeterministic way. 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 [Tak91] as *if* $T = L$ *then* M *else* N. Nondeterminacy arises when T is replaced by $\bullet$. The intentional semantics of $\bullet$ is *undefined*. $\mathbf{2}^+$ enjoys the following typing rules:

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

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.

(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

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

5.3 Realizability Interpretation

The realizability defined in this section is a variant of q-realizability [TvD88].

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 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.

A type is assigned for each formula, which is actually the type of the realizer of the 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. $type(A \Rightarrow B) \stackrel{\text{def}}{=} type(A) \to type(B)$;
5. $type(\forall x : \sigma.\ A) \stackrel{\text{def}}{=} \sigma \to type(A)$;
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 in a trivially extended logic in which realizability relations are regarded as formulas, 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. Other part of the extraction algorithm can be seen in [Tak91].

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 the successive two elements at once from the input stream and outputs the sum 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 $A(p,n,X,Y) \stackrel{\text{def}}{=} ((p|X_n)\ \&\ Y_n = \perp) \vee (\neg(p|X_n)\ \&\ Y_n = X_n)$ and $\Diamond$ is the rank 0 operator.
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: $\forall X : I_{nat}.\exists Y : I_{nat_\perp}.\forall n : nat.\ \Diamond A(n,X,Y)$
where
$A(n,X,Y) \stackrel{\text{def}}{=} (PR(X_n)\ \&\ Y_n = X_n) \vee (\neg PR(X_n)\ \&\ Y_n = \perp)$
and $PR(m) \stackrel{\text{def}}{=} \forall n : nat.\ (2 \le n < m \Rightarrow \neg(\exists d : nat.\ m = d \cdot n))$.

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, from $A(n, f(X), tl(Y)) \Leftrightarrow (PR(f(X)_n)\ \&\ Y_{n+1} = f(X)_n) \vee (\neg PR(f(X)_n)\ \&\ Y_{n+1} = \perp)$, $A(n+1,X,Y)$ can be proved. 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 in 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 only specifies that all the elements of the output stream come from the input streams. The rest of the criteria for a merge operation, namely, all the elements of the input streams occur in the output stream preserving the order of the input elements 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 $A(n,(X,Y),Z) \stackrel{\text{def}}{\equiv} (\exists m : nat.\ Z_n = X_m) \vee (\exists l : nat.\ Z_n = Y_l)$

Proof: By $(MPST)$. Let $M((X,Y),a) \stackrel{\text{def}}{\equiv} a = hd(X)$, 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, $A(n,(tl(X),Y),tl(Z)) \equiv (\exists m.\ tl(Z)_n = tl(X)_m) \vee (\exists l.\ tl(Z)_n = Y_l) \Leftrightarrow (\exists m.\ Z_{n+1} = X_{m+1}) \vee (\exists l.\ Z_{n+1} = Y_l)$. This implies $(\exists m'.\ Z_{n+1} = X_{m'}) \vee (\exists l.\ Z_{n+1} = Y_l) \equiv A(n+1,(X,Y),Z)$. 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
$\lambda(X,Y).\lambda m.\ ap(f_M, f_N^m(X,Y))$
where $f_M \stackrel{\text{def}}{\equiv} \lambda X.hd(X)$ and $f_N \stackrel{\text{def}}{\equiv} \lambda(X,Y).\ if\ \bullet = L\ then\ (tl(X),Y)\ else\ (Y,tl(X))$.

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. The $(MPST)$ rule allows to develop the proof of a specification with a good intuition on the concurrent process to be defined, and the rule seems to be easier to handle than the largest fixed point induction. 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

[Bur75] W. H. Burge. *Recursive Programming Techniques.* Addison-Wesley, 1975.

[C+86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[DS89] 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.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50, 1987. North-Holland.

[Got85] S. Goto. Concurrency in proof normalization and logic programming. In *Internatioinal Joint Conference on Artificial Intelligence '85*, 1985.

[Hag87] T. Hagino. A Typed Lambda Calculus with Categorical Type Constructors. In *Category Theory and Computer Science, LNCS 283*, 1987.

[HN89] S. Hayashi and H. Nakano. *PX : A Computational Logic.* The MIT Press, 1989.

[How80] 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.

[KM74] G. Kahn and D. B. MacQueen. The Semantics of a Simple Language for Parallel Programming. In *IFIP Congress 74.* North-Holland, 1974.

[KT91] S. Kobayashi and M. Tatsuta. private communication. 1991.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[PL86] N. Mendler P. Panangaden and R. L.Constable. Infinite Objects in Type Theory. In *Symposium on Logic in Computer Science'86*, 1986.

[Sat87] 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.

[SK90] M. Sato and Y. Kameyama. Constructive Programming in SST. In *Proceedings of the Japanese-Czechoslovak Seminar on Theoretical Foundations of Knowledge Information Processing*, pages 23–30, INORGA, 1990.

[Tak91] Y. Takayama. Extraction of Redundancy-free Programs from Constructive Natural Deduction Proofs. *Journal of Symbolic Computation*, 12(1):29–69, 1991.

[TvD88] 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.