

Streams from a type-theoretic perspective

Peter Hancock

jww Neil Ghani, Dirk Pattinson

Strathclyde University

18 May 2010, Nijmegen

Infinite objects

A stream is a paradigm for an ‘infinite object’ in a certain sense. There are two ‘poles’ in the notion of infinity:

- μ transfinite entities, having a wellfounded structure, typified by the ordinal ω . We define functions *on* such: folds, primitive recursion,
- ν coinductive entities, explorable ‘in perpetuity’, typified by streams B^ω . We define functions *to* such: unfolds,

Infinite objects

A stream is a paradigm for an ‘infinite object’ in a certain sense. There are two ‘poles’ in the notion of infinity:

- μ transfinite entities, having a wellfounded structure, typified by the ordinal ω . We define functions *on* such: folds, primitive recursion,
- ν coinductive entities, explorable ‘in perpetuity’, typified by streams B^ω . We define functions *to* such: unfolds,

The work presented here (joint with Ghani and Pattinson) concerns an interplay between initiality μ and finality ν .

Infinite objects

A stream is a paradigm for an ‘infinite object’ in a certain sense. There are two ‘poles’ in the notion of infinity:

- μ transfinite entities, having a wellfounded structure, typified by the ordinal ω . We define functions *on* such: folds, primitive recursion,
- ν coinductive entities, explorable ‘in perpetuity’, typified by streams B^ω . We define functions *to* such: unfolds,

The work presented here (joint with Ghani and Pattinson) concerns an interplay between initiality μ and finality ν . How to handle coinduction in dependent type theory (and tools based on it) is not a settled matter. I’m not sure there’s much consensus.

What I use

μ

$$\text{in} \quad : F(\mu F) \rightarrow \mu F$$

$$\text{fold}_C \quad : (F C \rightarrow C) \rightarrow \mu F \rightarrow C$$

$$f \cdot \text{in} = \gamma \cdot F f \quad : F(\mu F) \rightarrow C$$

$$\iff f = \text{fold } \gamma \quad : \mu F \rightarrow C$$

ν

$$\text{out} \quad : F \rightarrow F(\nu F)$$

$$\text{unfold}_C \quad : (C \rightarrow F C) \rightarrow C \rightarrow \nu F$$

$$\text{out} \cdot f = F f \cdot \gamma \quad : C \rightarrow F(\nu F)$$

$$\iff f = \text{unfold } \gamma \quad : C \rightarrow \nu F$$

What I use

μ

$$\text{in} \quad : F(\mu F) \rightarrow \mu F$$

$$\text{fold}_C \quad : (F C \rightarrow C) \rightarrow \mu F \rightarrow C$$

$$f \cdot \text{in} = \gamma \cdot F f \quad : F(\mu F) \rightarrow C$$

$$\Leftarrow f = \text{fold } \gamma \quad : \mu F \rightarrow C$$

ν

$$\text{out} \quad : F \rightarrow F(\nu F)$$

$$\text{unfold}_C \quad : (C \rightarrow F C) \rightarrow C \rightarrow \nu F$$

$$\text{out} \cdot f = F f \cdot \gamma \quad : C \rightarrow F(\nu F)$$

$$\Leftarrow f = \text{unfold } \gamma \quad : C \rightarrow \nu F$$

On the practical side

asynchronous

non-reordering buffers

pipe-lining for throughput

flow-control

deadlock

stream processors

OS6, unix, virtual devices, tcp

synchronous

unbuffered

hand-shaken

low-latency

simulation

rpc

What is to come

streams, stream processors, and continuous functions

Simple types

one cycle $A^\omega \rightarrow B$ $T_A B$.

many cycles $A^\omega \rightarrow B^\omega$ $P_A B$

composition $(H^\omega \rightarrow B^\omega) \times (A^\omega \rightarrow H^\omega)$
 $\rightarrow (A^\omega \rightarrow B^\omega)$

automata, automata processors, ...

A fancier (more general) version of same thing with dependent types.

Stream processing programs

Given $A : \text{Set}$.

$$T_A : \text{Set} \rightarrow \text{Set}$$
$$T_A B \triangleq (\mu X) \underbrace{B}_! + \underbrace{(A \rightarrow X)}_?$$

Stream processing programs

Given $A : \text{Set}$.

$$T_A : \text{Set} \rightarrow \text{Set}$$
$$T_A B \triangleq (\mu X) \underbrace{B}_{!} + \underbrace{(A \rightarrow X)}_{?}$$

$$e : T_A B \rightarrow A^\omega \rightarrow B$$

$$e(!b) \alpha = b$$

$$e(?f) \alpha = e(f \alpha_0) \alpha'$$

[In [Tait](#) 1967 “Constructive Reasoning” (LMPS 3), modulo notation.]

Stream processing programs

Given $A : \text{Set}$.

$$T_A : \text{Set} \rightarrow \text{Set}$$
$$T_A B \triangleq (\mu X) \underbrace{B}_! + \underbrace{(A \rightarrow X)}_?$$

$$e : T_A B \rightarrow A^\omega \rightarrow B$$

$$e(!b) \alpha = b$$

$$e(?f) \alpha = e(f \alpha_0) \alpha'$$

[In [Tait 1967](#) “Constructive Reasoning” (LMPS 3), modulo notation.]

$$\tilde{e} : T_A B \rightarrow (A^\omega \rightarrow B \times A^\omega)$$

$$\tilde{e}(!b) \alpha = (b, \alpha)$$

$$\tilde{e}(?f) \alpha = \tilde{e}(f \alpha_0) \alpha'$$

T_A is the free monad over $X \mapsto X^A : \text{Set} \rightarrow \text{Set}$.

Brouwer's ordinals are represented by trees in $[\{0\}, N]$. $\langle 0, 0 \rangle$ is the least ordinal; $\langle 1, \pi r \rangle$ is the successor of r ; and if s is not of the form πr , $\langle 1, s \rangle$ is the ordinal sum $s0 + s1 + \dots$. BROUWER (e.g., [1927]) applied the theory of ordinals to his non-atomistic theory of non-discrete spaces such as Baire space and the continuum. But, it is just as convenient to work directly with trees, rather than with their ordering as ordinals. In fact, it is simpler to use the trees in $[N, N]$, which I will call the *countable* trees.⁸ Let $\vec{t} = \lambda x. t(\pi x)$. There is a nt η such that, writings' t for ηst , $\langle 0, s \rangle$ ' $t \equiv s$ and $\langle 1, s \rangle$ ' $t \equiv (s(t0))$ ' \vec{t} (when s and t are nt and the right hand sides are defined). Let p range over

⁷ BROUWER (cf. [1927]) does want to regard proofs as infinite structures. But, if the proof of $T_0\omega$ is so regarded, we see that it has the same structure as ω ; and so it would be circular to infer the well-foundedness of ω (i.e., $T_0\omega$) from the well-foundedness of this proof.

⁸ Let $<$ denote the partial ordering of $[U, V]$ generated by $sb^u < s$. The trees s in $[N, N]$ are countable in the sense that the $t < s$ can be enumerated. Of course, the species of all nt is countable; and so, classically, every $[U, V]$ is countable. But, I am using this term in its constructive sense.

Stream processors : $A^\omega \rightarrow B^\omega$

Run Tait in a loop

$$P_A : \text{Set} \rightarrow \text{Set}$$

$$P_A B \triangleq (\nu X) T_A(B \times X)$$

$$e_\infty : P_A B \rightarrow A^\omega \rightarrow B^\omega$$

Stream processors : $A^\omega \rightarrow B^\omega$

Run Tait in a loop

$$P_A : \text{Set} \rightarrow \text{Set}$$

$$P_A B \triangleq (\nu X) T_A(B \times X)$$

$$e_\infty : P_A B \rightarrow A^\omega \rightarrow \underbrace{B^\omega}_{(\nu X) B \times X}$$

Stream processors : $A^\omega \rightarrow B^\omega$

Run Tait in a loop

$P_A : \text{Set} \rightarrow \text{Set}$

$P_A B \triangleq (\nu X) T_A(B \times X)$

$e_\infty : P_A B \rightarrow A^\omega \rightarrow \underbrace{B^\omega}_{(\nu X) B \times X}$

$P_A B \times A^\omega = C$

$\downarrow \text{out} \times 1$

$T_A(B \times P_A B) \times A^\omega$

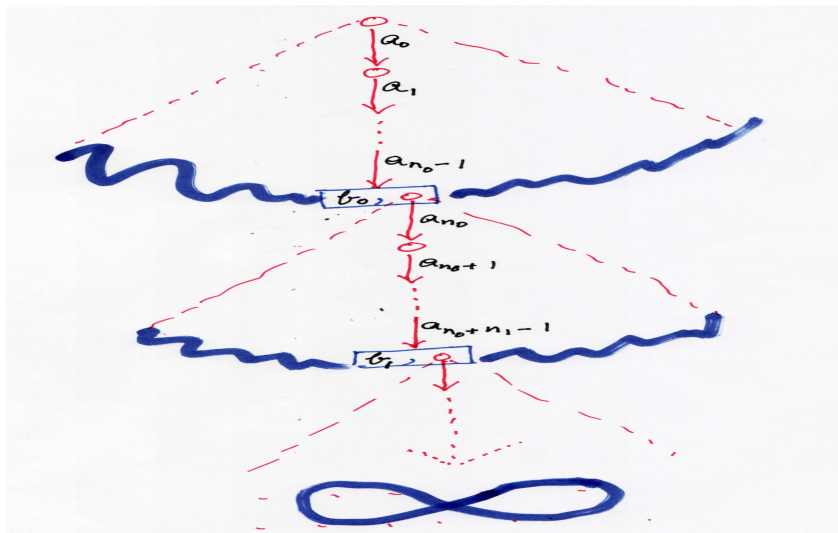
$\downarrow \tilde{e}$

$(B \times P_A B) \times A^\omega$

$\downarrow \text{assoc}$

$B \times (P_A B \times A^\omega) = B \times C$

Picture of a $P_A B$



Composing stream processors – pipes

compose : $P_A B \rightarrow P_C A \rightarrow P_C B$

Composing stream processors – pipes

$$\text{compose} : P_A B \rightarrow P_C A \rightarrow \underbrace{P_C B}_{(\nu X) T_C(B \times X)}$$

Composing stream processors – pipes

$$\text{compose} : P_A B \rightarrow P_C A \rightarrow \underbrace{P_C B}_{(\nu X) T_C(B \times X)}$$

$$P_A B \times P_C A$$

$$\downarrow \text{out} \times 1$$

$$T_A(B \times P_A B) \times P_C A$$

$$\downarrow \text{'c.o.b'} : T_A X \rightarrow (P_C A \rightarrow T_C(X \times P_C A))$$

$$T_C((B \times P_A B) \times P_C A)$$

$$\downarrow \text{fmap assoc}$$

$$T_C(B \times (P_A B \times P_C A))$$

Composing stream processors – pipes

$$\text{compose} : P_A B \rightarrow P_C A \rightarrow \underbrace{P_C B}_{(\nu X) T_C(B \times X)}$$

$$\begin{aligned} P_A B \times P_C A \\ \downarrow \text{out} \times 1 \\ T_A(B \times P_A B) \times P_C A \\ \downarrow \text{'c.o.b'} : T_A X \rightarrow (P_C A \rightarrow T_C(X \times P_C A)) \\ T_C((B \times P_A B) \times P_C A) \\ \downarrow \text{fmap assoc} \\ T_C(B \times (P_A B \times P_C A)) \end{aligned}$$

$$\begin{aligned} \text{cob}(!x) &= (\lambda p) !\langle x, p \rangle \\ \text{cob}(?f) &= \text{mul} \cdot \text{fmap}((\lambda \langle a, p \rangle) \text{cob}(f a)p) \cdot \text{out} \end{aligned}$$

Uses that T_C is a functor (fmap), and moreover a monad (mul).

Scheduler

$$?f \odot ?g \mapsto ?(a \mapsto (?f) \odot g a)$$

$$?f \odot !(m, p) \mapsto f m \odot (out p) \quad \text{internal comms}$$

$$!(c, p) \odot ?g \mapsto \begin{cases} ?(a \mapsto !(c, p)) \odot g a \\ !(c, (out p)) \odot g a \end{cases}$$

$$!(c, p) \odot !(m, q) \mapsto !(c, (out p)) \odot !(m, q)$$

Scheduler

$$\begin{aligned} ?f \quad \odot \quad ?g &\mapsto ?(a \mapsto (?f) \odot g a) \\ ?f \quad \odot \quad !(m, p) &\mapsto f m \odot (out p) && \text{internal comms} \\ !(c, p) \odot ?g &\mapsto \begin{cases} ?(a \mapsto !(c, p)) \odot g a \\ !(c, (out p)) \odot g a \end{cases} \\ !(c, p) \odot !(m, q) &\mapsto !(c, (out p)) \odot !(m, q) \end{aligned}$$

There are at least \aleph_1 behaviourally different implementations of the identity function : $\omega^\omega \rightarrow \omega^\omega$, i.e. buffers.

Summary, streams

$$\nu((A \rightarrow)^* \cdot (B \times))$$

$((? A)^* \circ ; (! B))^\omega$ (Kleene $_*$, Park $_^\omega$)

always eventually
infinitely often

$\square \diamond$

or

$$(\nu X) (\mu Y) B \times X + Y^A$$

The outer 'polarity' ν binds X which occurs in the scope of the variable bound by the inner polarity μ .

Generalisation, automata

Let $B : A \rightarrow \text{Set}$ be a family of sets. The elements of A are output 'requests'. If a is a request, then $B a$ is the set of input 'responses' that may be returned.

By an (A, B) automaton (Moore-style), I mean a tuple (S, δ, s_0) where

$$S : \text{Set}$$

$$\delta : S \rightarrow (\sum a : A) S^{B a}$$

$$s_0 : S$$

Generalisation, automata

Let $B : A \rightarrow \text{Set}$ be a family of sets. The elements of A are output 'requests'. If a is a request, then $B a$ is the set of input 'responses' that may be returned.

By an (A, B) automaton (Moore-style), I mean a tuple (S, δ, s_0) where

$$S : \text{Set}$$

$$\delta : S \rightarrow (\sum a : A) S^{B a}$$

$$s_0 : S$$

Such a thing implements an element of $(\nu X : \text{Set}) (\sum a : A) X^{B a}$. I will write $(A \triangleleft B)$ for the 'generalised polynomial' (a.k.a *container*) functor $X \mapsto (\sum a : A) X^{B a}$. An (A, B) automaton is an inhabitant of $\nu(A \triangleleft B)$.

Automata neighbourhoods

An element of the final coalgebra $\nu(A \triangleleft B)$ is given by a convergent stream of successive *observations* of that element, each refining the observations that have gone before.

As basic neighbourhoods we take complete explorations of the behaviour of the automaton, to a given depth/diameter. These are trees, but they grow in a layered fashion at their leaves, rather than at their roots.

Neighbourhoods and growth-points

By the magic of 'induction-recursion', I define simultaneously a set A_* and a family of sets $B_* : A_* \rightarrow \text{Set}$.

- ▶ A_* represents the set of (basic) neighbourhoods for the topology on $\nu(A \triangleleft B)$.
- ▶ B_* is the function that assigns to a neighbourhood the growth-locations/threads at which it may be refined.

A_* is defined inductively *while* B_* is defined recursively.

Neighbourhoods and growth-points

By the magic of 'induction-recursion', I define simultaneously a set A_* and a family of sets $B_* : A_* \rightarrow \text{Set}$.

- ▶ A_* represents the set of (basic) neighbourhoods for the topology on $\nu(A \triangleleft B)$.
- ▶ B_* is the function that assigns to a neighbourhood the growth-locations/threads at which it may be refined.

A_* is defined inductively *while* B_* is defined recursively.

$$() : A_* \quad ; B_*() = 1$$

$$\frac{\mathbf{a} : A_* \quad \sigma : B_* \mathbf{a} \rightarrow A}{\mathbf{a} \hat{\wedge} \sigma : A_*} \quad ; B_* (\mathbf{a} \hat{\wedge} \sigma) = (\Sigma b : B_* \mathbf{a}) B (\sigma b)$$

We slice an automaton into layers, like slices of salami.

$$\underbrace{(\cdots ((() \hat{\wedge} \sigma_0) \hat{\wedge} \sigma_1) \hat{\wedge} \cdots) \hat{\wedge} \sigma_{n-1}}_t \quad ((\cdots (((), b_0), b_1), \cdots), b_{n-1}) \mapsto \alpha_t$$

The covering relation

The set of neighbourhoods $A_* : Set$ and the function $B_* : A_* \rightarrow Set$ were introduced an initial algebra in a certain category with objects dependent families (I, J) .

We now make another 'initial' construction, of a closure operator, that takes a set valued function $Q : A_* \rightarrow Set$, and returns another $\overline{Q} : A_* \rightarrow Set$, which is the closure of Q under a certain ' Π -type'.

$$\overline{Q} \mathbf{a} = \underbrace{Q \mathbf{a}}_! + \underbrace{(\prod f : B_* \mathbf{a} \rightarrow A) \overline{Q}(\mathbf{a} \hat{=} f)}_?$$

$\overline{Q} \mathbf{a}$ means that \mathbf{a} is covered by neighbourhoods \mathbf{a}' such that $Q \mathbf{a}'$ is inhabited.

If Q is the predicate 'dead', then \overline{Q} is the predicate 'mortal'.

One cycle: finite reading, then one write

- ▶ Automata: $M \triangleq \nu(A \triangleleft B)$. $hd : M \rightarrow A$
 $tl : (\prod \alpha : M) B(hd \alpha) \rightarrow M$
- ▶ Container 'with coefficients':
 $(A \triangleleft_C B) X \triangleq (\sum a : A) C a \times (B a \rightarrow X)$

One cycle: finite reading, then one write

- ▶ Automata: $M \stackrel{\Delta}{=} \nu(A \triangleleft B)$. $hd : M \rightarrow A$
 $tl : (\prod \alpha : M) B(hd \alpha) \rightarrow M$
- ▶ Container 'with coefficients':
 $(A \triangleleft_C B) X \stackrel{\Delta}{=} (\sum a : A) C a \times (B a \rightarrow X)$
 $\tilde{e} : (A_* \triangleleft_{\overline{C}} B_*)(M) \rightarrow (A_* \triangleleft_C B_*)(M)$

One cycle: finite reading, then one write

- ▶ Automata: $M \triangleq \nu(A \triangleleft B)$. $hd : M \rightarrow A$
 $tl : (\prod \alpha : M) B(hd \alpha) \rightarrow M$

- ▶ Container 'with coefficients':
 $(A \triangleleft_C B) X \triangleq (\sum a : A) C a \times (B a \rightarrow X)$

$$\tilde{e} : (A_* \triangleleft_{\overline{C}} B_*)(M) \rightarrow (A_* \triangleleft_C B_*)(M)$$

$$\tilde{e}\langle \mathbf{a}, !b, \alpha \rangle = \langle \mathbf{a}, b, \alpha \rangle$$

$$\tilde{e}\langle \mathbf{a}, ?f, \alpha \rangle = \tilde{e}\langle \mathbf{a} \hat{\ } \sigma, f \sigma, \alpha' \rangle$$

where $\sigma : B_* \mathbf{a} \rightarrow A$

$$\sigma b \triangleq hd(\alpha b)$$

$$\alpha' : B_*(\mathbf{a} \hat{\ } \sigma) \rightarrow M$$

$$\alpha'\langle b, b' \rangle \triangleq tl(\alpha b) b'$$

How about when the *output* is infinite?

We will now be consuming entities in $\nu(A \triangleleft B)$, and producing entities in $\nu(C \triangleleft D)$.

This generalises our previous step from discrete output $A^\omega \rightarrow C$ to continuous output $A^\omega \rightarrow C^\omega$.

How about when the *output* is infinite?

We will now be consuming entities in $\nu(A \triangleleft B)$, and producing entities in $\nu(C \triangleleft D)$.

This generalises our previous step from discrete output $A^\omega \rightarrow C$ to continuous output $A^\omega \rightarrow C^\omega$.

The crucial thing is to do this in a *composable* way. This needs a little bit of thought.

How about when the *output* is infinite?

We will now be consuming entities in $\nu(A \triangleleft B)$, and producing entities in $\nu(C \triangleleft D)$.

This generalises our previous step from discrete output $A^\omega \rightarrow C$ to continuous output $A^\omega \rightarrow C^\omega$.

The crucial thing is to do this in a *composable* way. This needs a little bit of thought.

We need to generalise, and define a *doubly* indexed type of programs $Pos \mathbf{c} \mathbf{a}$ where $\mathbf{c} : C_*$, $\mathbf{a} : A_*$, and an interpreter (e_∞) of type:

$$(\prod \mathbf{c} : C_*, \mathbf{a} : A_*) Pos \mathbf{c} \mathbf{a} \rightarrow (B_* \mathbf{a} \rightarrow \nu(A \triangleleft B)) \rightarrow (D_* \mathbf{c} \rightarrow \nu(C \triangleleft D))$$

What's Pos ?

Some cogitation/calculation suggests:

$$Pos \stackrel{\Delta}{=} (\nu W : C_* \rightarrow A_* \rightarrow Set) \\ (\lambda \mathbf{c} : C_*) (\lambda \mathbf{a} : A_*) (\Sigma \tau : D_* \mathbf{c} \rightarrow C) W(\mathbf{c} \hat{\sim} \tau) \mathbf{a}$$

Which you may like to compare with

$$(\nu X) (A \rightarrow)^*(C \times X)$$

What's Pos ?

Some cogitation/calculation suggests:

$$Pos \stackrel{\Delta}{=} (\nu W : C_* \rightarrow A_* \rightarrow Set) \\ \overline{(\lambda \mathbf{c} : C_*) (\lambda \mathbf{a} : A_*) (\Sigma \tau : D_* \mathbf{c} \rightarrow C) W(\mathbf{c} \hat{\cap} \tau) \mathbf{a}}$$

Which you may like to compare with

$$(\nu X) (A \rightarrow)^*(C \times X)$$

The key isomorphism: for $\mathbf{c} : C_*$,

$$Pos \mathbf{c} \cong \overline{\left(\bigcup_{\tau : D_* \mathbf{c} \rightarrow C} Pos(\mathbf{c} \hat{\cap} \tau) \right)} : A_* \rightarrow Set$$

This supports a variety of operations that represent composition.

Standing Back

- ▶ representation of continuous functions between streams by coinductive-inductive stream processors.
- ▶ representation(s) of identity and composition, defined using unfolds.
- ▶ generalisation of this to 'automata'. (And beyond.)
- ▶ **NOT** done: proof that we have a category (*i.e.* associativity). This needs us to be precise about equality between stream processors, and adds another **unwelcome** dimension of complexity.

Standing Back

- ▶ representation of continuous functions between streams by coinductive-inductive stream processors.
- ▶ representation(s) of identity and composition, defined using unfolds.
- ▶ generalisation of this to ‘automata’. (And beyond.)
- ▶ **NOT** done: proof that we have a category (*i.e.* associativity). This needs us to be precise about equality between stream processors, and adds another **unwelcome** dimension of complexity.

The topology put on ‘automata’ is inspired by their representations as limits of ω^{op} -chains $1 \xleftarrow{!} F 1 \xleftarrow{F!} \dots$. Perhaps this is too crude, *e.g.* when F is ‘infinitary’ in some sense.

Standing Back

- ▶ representation of continuous functions between streams by coinductive-inductive stream processors.
- ▶ representation(s) of identity and composition, defined using unfolds.
- ▶ generalisation of this to ‘automata’. (And beyond.)
- ▶ **NOT** done: proof that we have a category (*i.e.* associativity). This needs us to be precise about equality between stream processors, and adds another **unwelcome** dimension of complexity.

The topology put on ‘automata’ is inspired by their representations as limits of ω^{op} -chains $1 \xleftarrow{!} F 1 \xleftarrow{F!} \dots$. Perhaps this is too crude, *e.g.* when F is ‘infinitary’ in some sense.

Schedulers and fairness. (Park, 80’s..., $\nu(F \cdot F^* \cdot G + G \cdot G^* \cdot F)$)

References



Peter Dybjer.

A general formulation of simultaneous inductive-recursive definitions in type theory.

JSL, 65(2):525–549, June 2000.



Peter Dybjer and Anton Setzer.

Induction-recursion and initial algebras.

APAL, 124(1-3):1–47, 2003.



Peter Dybjer and Anton Setzer.

Indexed induction-recursion.

JLAP, 66:1–49, 2006.



Peter Hancock, Neil Ghani, and Dirk Pattinson.

Continuous functions on final coalgebras.

In *MFPS 2009*, number 249 in ENTCS, pages 3–18, 2009.



Peter Hancock, Neil Ghani, and Dirk Pattinson.

Representations of stream processors using nested fixed points.

LMCS, 5(3):1–17, 2009.

Bar induction

Two notions of well-foundedness: **no inf. desc. chains** versus **inductive**. (Classically equivalent.)

Let A^* be the set of finite lists of A 's, $()$ the empty list, (\wedge) add an element to a list at the end, and $\bar{\alpha}n$ means $(\alpha_0, \alpha_1, \dots, \alpha_{n-1}) : A^*$.

$$\begin{aligned} & (\forall \alpha : A^\omega)(\exists n : \omega)B(\bar{\alpha}n) \rightarrow \\ & ((\forall \mathbf{a} : A^*)B\mathbf{a} \rightarrow C\mathbf{a}) \rightarrow \\ & ((\forall \mathbf{a} : A^*)((\forall x : A)C(\mathbf{a}\wedge x)) \rightarrow C\mathbf{a}) \rightarrow \\ & C() \end{aligned}$$

- ▶ $BI_D : (\forall \mathbf{a} : A^*)B\mathbf{a} + (B\mathbf{a} \rightarrow 0)$.
- ▶ $BI_M : (\forall \mathbf{a} : A^*, x : A)B\mathbf{a} \rightarrow B(\mathbf{a}\wedge x)$.

Some such necessary by 'Kleene-Vesley'. See Tait '68 too.