Non-wellfounded trees in Homotopy Type Theory

Benedikt Ahrens [1] Paolo Capriotti [2] Régis Spadotti [1]

[1]Institut de Recherche en Informatique de Toulouse, Université Paul Sabatier
[2]School of Computer Science, University of Nottingham

Types 2015

Summary

Main result

Coinductive types—in the form of M-types—are **derivable** in HoTT.

- Details in an article with same name (arXiv:1504.02949)
- Agda code on `https://hott.github.io/M-types/`

Outline

1. What are non-wellfounded trees?
2. Overview of coinductive types in extensions of type theory
3. Outline of the construction of a type of streams
4. The construction in HoTT: difficulties compared to ETT

Outline

Coinductive types via M-types

Coinductive types

- types of potentially infinite data structures
- only a finite part is "visible" at any moment

M-types

- a particular class of coinductive types
- elements can be visualized as **trees** of a given shape
- typical examples of coinductive types are M-types:
  - streams
  - colists

Non-wellfounded trees in pictures

Specification of an M-type

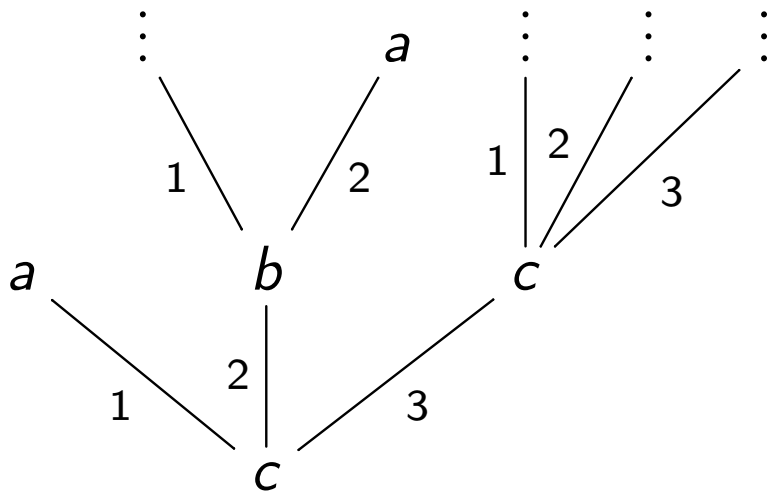
- a type A of **labels** for nodes
- a family of types $\left(B(a)\right)_{a \in A}$ specifying **branching**

$a, b, c \in A$

$B(a) = \{\}$

$B(b) = \{1, 2\}$

$B(c) = \{1, 2, 3\}$



Example: streams

Take A a type and $B := \lambda(a : A).1$. A tree of that shape is

$$a_0 \xrightarrow{1} a_1 \xrightarrow{1} a_2 \xrightarrow{1} \dots$$

Example: streams

Take A a type and $B := \lambda(a : A).1$. A tree of that shape is

$$a_0 \xrightarrow{1} a_1 \xrightarrow{1} a_2 \xrightarrow{1} \dots$$

Given a stream a, we can decompose it:

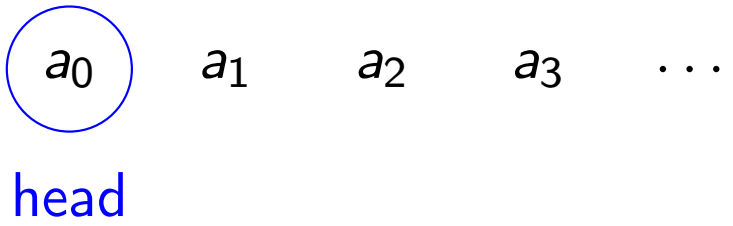
$$a_0 \qquad a_1 \qquad a_2 \qquad a_3 \qquad \dots$$

Example: streams

Take A a type and $B := \lambda(a : A).1$. A tree of that shape is

$$a_0 \xrightarrow{1} a_1 \xrightarrow{1} a_2 \xrightarrow{1} \dots$$

Given a stream a, we can decompose it:

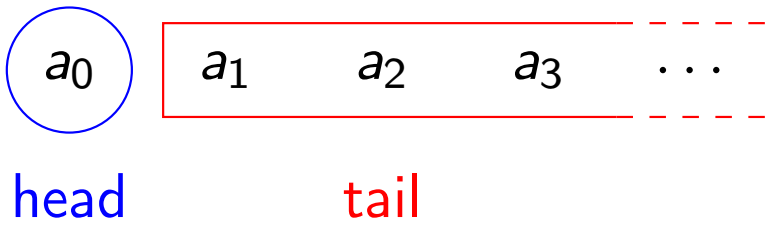


Example: streams

Take A a type and $B := \lambda(a : A).1$. A tree of that shape is

$$a_0 \xrightarrow{1} a_1 \xrightarrow{1} a_2 \xrightarrow{1} \dots$$

Given a stream a, we can decompose it:



Example: streams

Take A a type and $B := \lambda(a : A).1$. A tree of that shape is

$$a_0 \xrightarrow{1} a_1 \xrightarrow{1} a_2 \xrightarrow{1} \dots$$

Given a stream a, we can decompose it:

The tail of a stream is again a stream.

What is an M-type, formally?

Two possible ways to formally specify the type of streams:

externally: via typing rules

$$\frac{A : \mathcal{U}}{\mathsf{Stream}A : \mathcal{U}} \qquad \frac{t : \mathsf{Stream}A}{\mathsf{head}_A\ t : A} \qquad \frac{t : \mathsf{Stream}A}{\mathsf{tail}_A\ t : \mathsf{Stream}A} \qquad \dots$$

internally: via a universal property

- as a "terminal coalgebra": as the pair

$$(\mathsf{Stream}A, \langle \mathsf{head}, \mathsf{tail} \rangle : \mathsf{Stream}A \to A \times \mathsf{Stream}A)$$

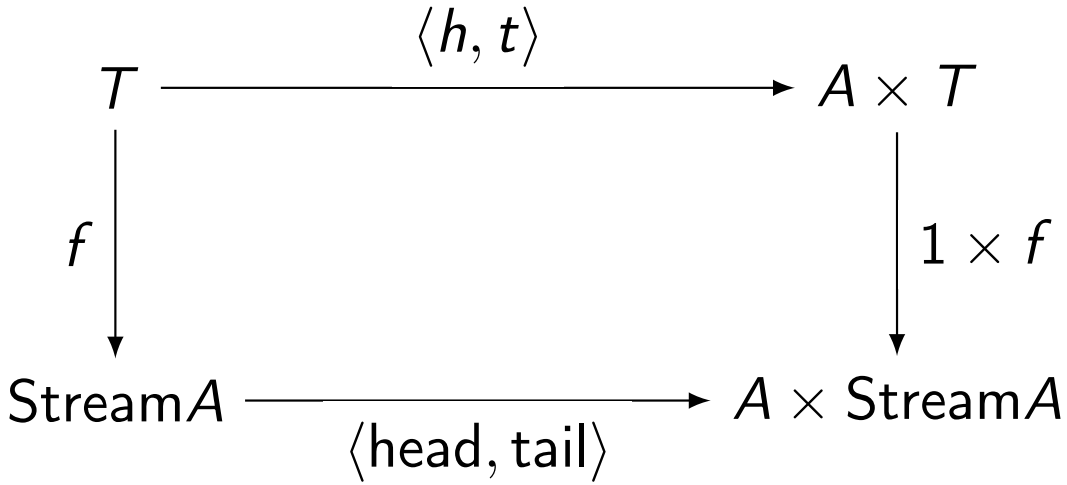
such that ...

Universal property of streams

. . . for any pair

$$(T, \langle h, t \rangle : T \to A \times T)$$

there is exactly one $f : T \to \mathsf{Stream} A$ such that



- Call this characterisation "internal", because is expressible within type theory.
- For any two $(\mathsf{Stream} A, \langle \mathsf{head}, \mathsf{tail} \rangle)$ and $(\mathsf{Stream}' A, \langle \mathsf{head}', \mathsf{tail}' \rangle)$, the carriers are isomorphic.

Constructing coinductive types

- In type theory with enough extensionality, the type $\mathbb{N} \to A$ satisfies the universal property of $\mathsf{Stream}A$.
- More generally, can M-types be built from other type constructors?

Outline

Martin-Löf type theory (MLTT)

Type theory	Notation
Inhabitant	$a : A$
Dependent type	$x : A \vdash B(x)$
Sigma type	$\sum_{(x:A)} B(x)$
Product type	$\prod_{(x:A)} B(x)$
Identity type	$a =_A b$
Natural numbers	$\mathbb{N}$

Construction of coinductive types in extensions of MLTT

with Identity Reflection (extensional TT)

- *Non-wellfounded trees in categories*, v. d. Berg & De Marchi 2004
- *Containers - constructing strictly positive types*, Abbott, Altenkirch & Ghani 2005

with Uniqueness of Identity Proofs & FunExt

- *Indexed containers*, Altenkirch + 4, unpublished

Coinductive types in MLTT + something

something = Identity Reflection or UIP

- a type A looks like a (constructive) set
- a universe $\mathcal{U}$ of types has the structure of a topos (roughly)
- construction of terminal coalgebras for polynomial functors on a topos well-known

something = Univalence

- a tower $(A, =_A, =_{=_A}, \ldots)$ looks like an ∞-groupoid
- a universe $\mathcal{U}$ has the structure of an "∞-topos"
- **this work:** construction of terminal coalgebras for polynomial functors on ∞-toposes

POST-TALK EDIT: the last point is to be read as an analogy, not as a precise statement. It should have been more clearly marked as such.

Remarks about extensions of MLTT

- For this talk, HoTT = MLTT + Univalence
- No HITs employed in this work
- UIP (as an axiom for all types) and Univalence are incompatible
- Univalence entails function extensionality
- Types satisfying UIP form a "subuniverse" of HoTT

Outline

How to construct streams

In this section

Sketch of classical construction of streams

- applies to ETT (mentioned work) and to HoTT (our work)
- differences between ETT and HoTT are discussed later

How to construct streams

In this section

Sketch of classical construction of streams

- applies to ETT (mentioned work) and to HoTT (our work)
- differences between ETT and HoTT are discussed later

Goal

Using the type constructors of MLTT, define a type $\mathsf{Stream}A$ and maps

$$\langle \mathsf{head}, \mathsf{tail} \rangle : \mathsf{Stream}A \to A \times \mathsf{Stream}A$$

s.t. for any $(T, \langle h, t \rangle : T \to A \times T)$, there is a unique "good"

$$f : T \to \mathsf{Stream}A$$

Approximating streams by a sequence of growing lists

A stream over A is one of...

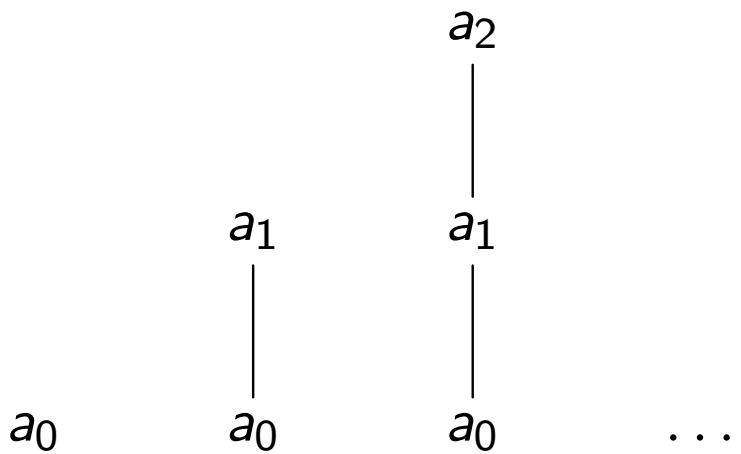
- a map $\mathbb{N} \to A$,

$$a_0, a_1, a_2, \ldots$$

- the "limit" of a sequence of finite approximations, i.e., of elements of A^n

$$(a_0), (a_0, a_1), (a_0, a_1, a_2), \ldots$$

in tree form



Approximating trees levelwise

The latter point of view generalizes from streams to arbitrary trees:

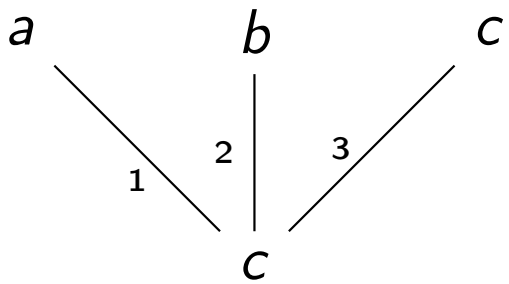
Approximating trees levelwise

The latter point of view generalizes from streams to arbitrary trees:

C

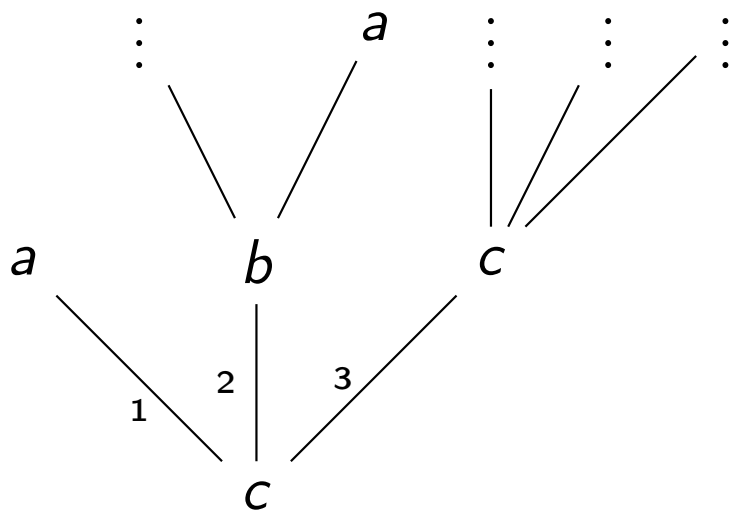
Approximating trees levelwise

The latter point of view generalizes from streams to arbitrary trees:



Approximating trees levelwise

The latter point of view generalizes from streams to arbitrary trees:



Sequence of lists and projections

$$1 \xleftarrow{!} A \xleftarrow{l_1} A^2 \xleftarrow{l_2} A^3 \xleftarrow{l_3} \dots$$

$$tt \longleftarrow\!\shortmid a_0 \longleftarrow\!\shortmid (a_0, a_1) \longleftarrow\!\shortmid (a_0, a_1, a_2) \longleftarrow\!\shortmid \dots$$

Sequence of lists and projections

$$1 \xleftarrow{!} A \xleftarrow{l_1} A^2 \xleftarrow{l_2} A^3 \xleftarrow{l_3} \dots$$

$$tt \longleftarrow a_0 \longleftarrow (a_0, a_1) \longleftarrow (a_0, a_1, a_2) \longleftarrow \dots$$

Definition

A **stream over** A consists of

- $(x_n : A^n)_{n:\mathbb{N}}$ with
- $l_n(x_{n+1}) = x_n$

$$\mathsf{Stream}A := \sum_{(x:\prod_{(n:\mathbb{N})} A^n)} \prod_{(n:\mathbb{N})} l_n(x_{n+1}) = x_n$$

Universal property for streams

Definition

$$\mathsf{Stream}A := \sum_{(x:\prod_{(n:\mathbb{N})} A^n)} \prod_{(n:\mathbb{N})} l_n(x_{n+1}) = x_n$$

Theorem

$\mathsf{Stream}A$ *has the universal property of streams. I.e. we have*

coalgebra structure

maps $\langle \mathsf{head}, \mathsf{tail} \rangle : \mathsf{Stream}A \to A \times \mathsf{Stream}A$

universal property

for any $(T, h : T \to A, t : T \to T)$, *there is a unique "good"* $f : T \to \mathsf{Stream}A$

Proof of the theorem

Coalgebra structure

- $\mathsf{head}\Big((tt, (a_0), (a_0', a_1), \ldots), p\Big) := a_0 : A$
- $$\mathsf{tail}\Big((tt,\ (a_0),\ (a_0', a_1),\ ((a_0', a_1)', a_2),\ \ldots),\ p\Big) := \Big((tt,\ (a_1),\ (a_1', a_2),\ ((a_1', a_2)', a_3),\ \ldots),\ \bar{p}\Big)$$
- $\langle \mathsf{head}, \mathsf{tail} \rangle$ is an equivalence

Universal property

Given $T, h : T \to A$ and $t : T \to T$, define $f : T \to \mathsf{Stream} A \ldots$

Outline

Differences between the constructions in ETT and HoTT

- In general, terms of identity type are non-trivial in HoTT
- cannot "ignore" them or equate them to reflexivity

When are two coalgebra morphisms equal?

in ETT	when the carriers are equal—use UIP to equate second components
in HoTT	also need the respective diagrams to commute **in the same way**

Proof-relevant identities

Proving two morphisms into streams equal involves comparing them pointwise, i.e., comparing streams:

When are two streams (x, p) and (x', p') equal?

in ETT when $x_n = x'_n$ for any n

in HoTT also need that $p : \prod_{(n:\mathbb{N})} l_n(x_{n+1}) = x_n$ is equal to $p' : \prod_{(n:\mathbb{N})} l_n(x'_{n+1}) = x'_n$ modulo transport

Proof-relevant identities

Proving two morphisms into streams equal involves comparing them pointwise, i.e., comparing streams:

When are two streams (x, p) and (x', p') equal?

in ETT when $x_n = x'_n$ for any n

in HoTT also need that $p : \prod_{(n:\mathbb{N})} l_n(x_{n+1}) = x_n$ is equal to $p' : \prod_{(n:\mathbb{N})} l_n(x'_{n+1}) = x'_n$ modulo transport

Calculations on paths are made feasible by

- decomposing them into small steps
- steps are often instances of general lemmas about paths, sigma types, etc.

A typical proof involving many paths

$$
\begin{aligned}
& L^P \\
\simeq & \sum_{(w:\prod_{(n:\mathbb{N})} \sum_{(a:A)} B(a)\to X_n)} \prod_{(n:\mathbb{N})} (P\pi_n)w_{n+1} = w_n \\
\simeq & \sum_{(a:\prod_{(n:\mathbb{N})} A)} \sum_{(u:\prod_{(n:\mathbb{N})} B(a_n)\to X_n)} \prod_{(n:\mathbb{N})} (a_{n+1}, \pi_n \circ u_{n+1}) = (a_n, u_n) \\
\simeq & \sum_{(a:\prod_{(n:\mathbb{N})} A)} \sum_{(p:\prod_{(n:\mathbb{N})} a_{n+1}=a_n)} \sum_{(u:\prod_{(n:\mathbb{N})} B(a_n)\to X_n)} \prod_{(n:\mathbb{N})} (p_n)_*(\pi_n \circ u_{n+1}) = u_n \\
\simeq & \sum_{(a:A)} \sum_{(u:\prod_{(n:\mathbb{N})} B(a)\to X_n)} \prod_{(n:\mathbb{N})} \pi_n \circ u_{n+1} = u_n \\
\simeq & \sum_{a:A} B(a) \to L \quad \simeq \quad PL
\end{aligned}
$$

Concluding remarks

Main result

Coinductive types—in the form of M-types—are **derivable** in HoTT.

Computation rule

only holds propositionally in the Agda implementation, due to lack of computational interpretation of UA/FunExt

Extensionality

bisimilarity is identity for the constructed M-types

Concluding remarks

Main result

Coinductive types—in the form of M-types—are **derivable** in HoTT.

Computation rule

only holds propositionally in the Agda implementation, due to lack of computational interpretation of UA/FunExt

Extensionality

bisimilarity is identity for the constructed M-types

The end