# Datatypes as Quotients of Polynomial Functors

Simon Hudon

September 2019

Department of Philosophy and
Department of Mathematical Sciences
Carnegie Mellon University

joint work with Jeremy Avigad and Mario Carneiro
https://github.com/avigad/qpf

# Datatypes

These are inductive datatypes:

```
inductive list (α : Type)
| nil  : list
| cons : α → list → list

inductive btree (α : Type)
| leaf : btree
| node : α → btree → btree → btree
```

# Datatypes

These are inductive datatypes:

```
inductive list (α : Type)
| nil  : list
| cons : α → list → list

inductive btree (α : Type)
| leaf : btree
| node : α → btree → btree → btree
```

Lean supports these ...

# Datatypes

… but not these:

# Datatypes

… but not these:

```
coinductive stream (α : Type)
| cons (head : α) (tail : stream) : stream
```

## Datatypes

… but not these:

```
coinductive stream (α : Type)
| cons (head : α) (tail : stream) : stream

inductive tree (α β : Type)
| node (head : α) (children : multiset tree) : tree
  -- `multiset` is defined as a quotient over lists
```

# Datatypes

… but not these:

```
coinductive stream (α : Type)
| cons (head : α) (tail : stream) : stream

inductive tree (α β : Type)
| node (head : α) (children : multiset tree) : tree
  -- `multiset` is defined as a quotient over lists

inductive free_monad (F : Type → Type) (α : Type)
| pure : α → free_monad
| intro : F free_monad → free_monad
```

# Problem

Problem:

# Problem

Problem:

- Lean does not have coinductive types

# Problem

Problem:

- Lean does not have coinductive types
- Lean cannot nest inductive types and quotient types

# Problem

Problem:

- Lean does not have coinductive types
- Lean cannot nest inductive types and quotient types

Solution:

# Problem

Problem:

- Lean does not have coinductive types
- Lean cannot nest inductive types and quotient types

Solution:

- write a formal theory of (co)inductive types

# Problem

Problem:

- Lean does not have coinductive types
- Lean cannot nest inductive types and quotient types

Solution:

- write a formal theory of (co)inductive types
- write a parser for datatype specification

# Problem

Problem:

- Lean does not have coinductive types
- Lean cannot nest inductive types and quotient types

Solution:

- write a formal theory of (co)inductive types
- write a parser for datatype specification
- write an equation compiler for definitions (in progress)

# Problem

Problem:

- Lean does not have coinductive types
- Lean cannot nest inductive types and quotient types

Solution:

- write a formal theory of (co)inductive types
- write a parser for datatype specification
- write an equation compiler for definitions (in progress)
- (all in Lean)

# Isabelle and BNFs

Isabelle has a remarkable datatype package, developed by Julian Biendarra, Jasmin Christian Blanchette, Martin Desharnais, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel.

It supports:

- inductive definitions
- coinductive definitions
- nested definitions, with other constructions (like finite sets and finite multisets)
- mutual definitions

## Isabelle and BNFs

The Isabelle solution is based on:

- initial F-algebra

# Isabelle and BNFs

The Isabelle solution is based on:

- initial F-algebra
- final F-coalgebra

# Isabelle and BNFs

The Isabelle solution is based on:

- initial F-algebra
- final F-coalgebra
- composition

# Isabelle and BNFs

The Isabelle solution is based on:

- initial F-algebra
- final F-coalgebra
- composition
- of *bounded natural functors*

# Isabelle and BNFs

An *$F$-algebra* is a set $\alpha$ with a function $F(\alpha) \to \alpha$.

Examples:

- For nat with $0 : \text{nat}$ and $S : \text{nat} \to \text{nat}$, take $F(\alpha) = 1 + \alpha$.
- For list $\beta$ with nil and cons, take $F(\alpha) = 1 + \beta \times \alpha$.

Inductive definitions are *initial* algebras, in the sense of category theory.

If we reverse the arrows, we get a *$F$-coalgebra*: $\alpha \to F(\alpha)$

# Isabelle and BNFs

```
inductive X
| intro : tree (lazy_list X) → X
```

# Isabelle and BNFs

```
inductive X
| intro : tree (lazy_list X) → X
```

is defined as $X \triangleq \mathit{fix}(\mathit{tree} \circ \mathit{lazy\_list})$

# Isabelle and BNFs

The class of multivariate BNFs is closed under:

- composition
- initial algebras
- final coalgebras

They include `finset` and `multiset` and others.

## Polynomial functors

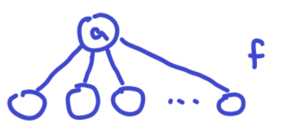
A *polynomial functor* $P$ is one of the form

$$P(\alpha) = \Sigma\, x : A,\ B\, a \to \alpha$$

for a fixed type $A$ and a fixed family of types $B : A \to \mathrm{Type}$.

Given $(a, f) \in P(\alpha)$, think of

- $a : A$ as the *shape*, and
- $f : B\, a \to \alpha$ as the *contents*



# Polynomial functors

Many common datatypes are (isomorphic to) polynomial functors.

For example, $\mathsf{list}\,\alpha \cong \Sigma\, n : \mathsf{nat},\ \mathsf{fin}\, n \to \alpha$.

Similarly, an element of $\mathsf{btree}\,\alpha$ has a shape, and nodes labeled by elements.

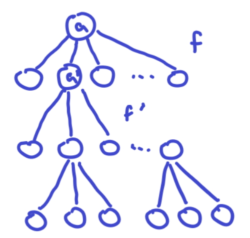
There is an obvious functorial action: $g : \alpha \to \beta$ maps $(a, f)$ to $(a, g \circ f)$.

# Polynomial functors

Every polynomial functor $P(\alpha)$ has an initial algebra $P(\alpha) \to \alpha$.

Think of elements as well-founded trees.

- Nodes have labels $a : \alpha$.
- Children are indexed by $B\,a$.




They are called *W types* and $P$'s final coalgebra yields *M types*.

# W and M types

$W$ types are easy in Lean using inductive types.

# W and M types

$W$ types are easy in Lean using inductive types.

```
inductive W (F : pfunctor) : Type
| intro {n} : ∀ a, (F.B a → W) → W
```

# W and M types

$M$ types are harder because Lean has no coinductive types.

# W and M types

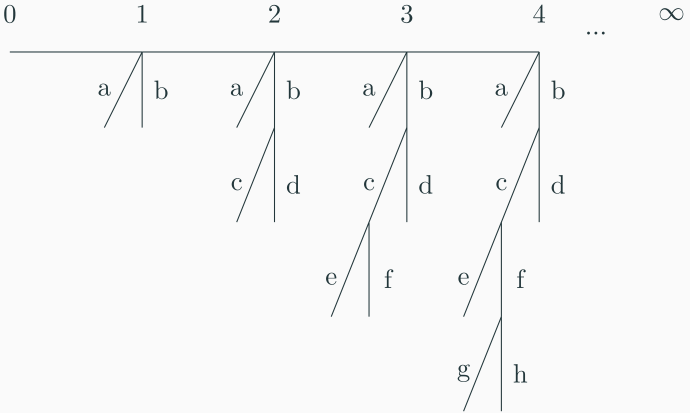
Solution?

# W and M types

Solution? Infinite series of finite approximations.

# W and M types

Solution? Infinite series of finite approximations.



# W and M types

Solution? Infinite series of finite approximations.

```
inductive cofix_a (F : pfunctor) : ℕ → Type u
| continue : cofix_a 0
| intro {n} : ∀ a, (F.B a → cofix_a n) → cofix_a (n+1)

inductive agree (F : pfunctor) :
  ∀ {n : ℕ}, cofix_a F n → cofix_a F (n+1) → Prop
| ...

structure M (F : pfunctor) :=
(approx : ∀ n, cofix_a F n)
(consistent : ∀ n, agree (approx n) (approx (n+1)))
```

# Polynomial functors

It is easy to show that polynomial functors are closed under composition.

So why not use them in place of BNFs?

# Polynomial functors

It is easy to show that polynomial functors are closed under composition.

So why not use them in place of BNFs?

The problem: constructors like finset and multiset are not polynomial functors.

For example, if $f(1) = f(2) = 3$, then $f$ maps $\{1, 2\}$ to $\{3\}$, which has a different shape.

# Polynomial functors

It is easy to show that polynomial functors are closed under composition.

So why not use them in place of BNFs?

The problem: constructors like finset and multiset are not polynomial functors.

For example, if $f(1) = f(2) = 3$, then $f$ maps $\{1, 2\}$ to $\{3\}$, which has a different shape.

The solution: use *quotients* of polynomial functors.

# Quotients of polynomial functors

$F(\alpha)$ is a *quotient of a polynomial functor (qpf)* if there are families

$$\mathit{abs} : P(\alpha) \to F(\alpha)$$

and

$$\mathit{repr} : F(\alpha) \to P(\alpha)$$

satisfying

$$\mathit{abs}(\mathit{repr}(x)) = x$$

for every $x$ in $F(\alpha)$.

Abstraction should be a natural transformation:

$$\mathit{abs} \circ P(f) = F(f) \circ \mathit{abs}$$

for every $f : \alpha \to \beta$.

## Quotients of polynomial functors

```
class qpf (F : Type u → Type u) [functor F] :=
(P          : pfunctor.{u})
(abs        : Π {α}, P.apply α → F α)
(repr       : Π {α}, F α → P.apply α)
(abs_repr   : ∀ {α} (x : F α), abs (repr x) = x)
(abs_map    : ∀ {α β} (f : α → β) (p : P.apply α),
                abs (f <$> p) = f <$> abs p)
```

Every BNF gives rise to a qpf.

# Quotients of polynomial functors

Let $W_P$ be the initial $P$-algebra.

Every element of $F(W_P)$ can have multiple representatives in $P(W_P)$.

So, to construct the intial $F$-algebra, we need to quotient out equivalent representations.

# Quotients of polynomial functors

The story for final coalgebras is more complicated.

We can analogously construct the greatest fixed point of $F(\alpha)$ by a suitable quotient of $M_P$.

The theory tells us to quotient by the greatest bisimulation of $M_P$.

## Quotients of polynomial functors

The remarkable conclusion: we end up using fewer assumptions that BNFs

The class of qpfs is closed under:

- composition
- quotients
- initial algebras
- final colagebras

In particular, `finset` and `multiset` are qpfs.

The constructions are pretty.

## Lean Formalization vs Isabelle Formalization

Lean

- No extension to the trusted code base
- Formalizes (co)fixed point of multivariate functors

# Lean Formalization vs Isabelle Formalization

Lean

- No extension to the trusted code base
- Formalizes (co)fixed point of multivariate functors

Isabelle

- No extension to the trusted code base
- For every natural number $n$, $n$-ary functors have their own theory

# Lean Formalization vs Coq Formalization

Lean

- No extension to the trusted code base
- Pattern matching is based on the use of recursors
- Allows recursive occurrences in parameters (when the parameter is a qpf)
- Supports Quotients

# Lean Formalization vs Coq Formalization

Lean

- No extension to the trusted code base
- Pattern matching is based on the use of recursors
- Allows recursive occurrences in parameters (when the parameter is a qpf)
- Supports Quotients

Coq

- (Co)Inductive types are part of the kernel
- Pattern matching is a language feature
- Do not allow recursive occurrences in parameters
- Do not support Quotients

# Implementation

Syntax looks native:

```
data tree (α β : Type) : Type
| leaf : tree
| node : α → (β → tree) → tree
```

# Implementation

Generated code:

```
inductive tree.shape
  (α : Type) (β : Type) (X : Type) : Type
| nil : tree.shape
| cons : α → (β → X) → tree.shape

def tree.shape.internal
  (β : Type) : typevec 2 → Type
| ⟨α,X⟩ := shape α β X

instance : mvfunctor (tree.shape.internal β) := ...
instance : mvqpf (tree.shape.internal β) := ...
```

# Implementation

Generated code (cont.):

```
def tree.internal (β : Type) (v : typevec 1) : Type
   := fix (list.shape.internal β) v
def tree (α β : Type) : Type := tree.internal β ⌈α⌉

instance : mvfunctor (tree.internal β) := ...
instance (β : Type) : mvqpf (tree.internal β) := ...
```

# Custom Equation Compiler

(Expected)

```
codef map {α β} (f : α → β) : stream α → stream β
| (cons x xs) := cons (f x) (map xs)

codef nats : stream ℕ :=
cons 0 (map nat.succ nats)
```

# Implementation

Generates also:

- Constructors (or destructors)
- (Co)recursors
- Induction principle (or bisimulation principle)
- Predicate and Relation lifting

## Implementation

Generates also:

- Constructors (or destructors)
- (Co)recursors
- Induction principle (or bisimulation principle)
- Predicate and Relation lifting

Limitation

- nesting not implemented yet
- no equation compiler
- no mutual (co)induction
- no (co)inductive families
- the desired computation rules require changes to Lean

Questions?

# Isabelle and BNFs

A functor $F(\alpha)$ is a *bounded natural functor* provided:

1. $F$ is a functor.
2. There is a natural transformation Fset from $F(\alpha)$ to set $\alpha$, such that the value of $F(f)(x)$ only depends on $f$ restricted to $\mathrm{Fset}(x)$.
3. $F$ preserves weak pullbacks.
4. There is a cardinal $\lambda$ such that
   4.1 $|\mathrm{Fset}(x)| \leq \lambda$ for every $x$
   4.2 $|\mathrm{Fset}^*(A)| \leq (|A| + 2)^\lambda$ for every set $A$.

This generalizes to multivariate functors.