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
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
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:

\texttt{coinductive stream (α : Type)}

| \texttt{cons (head : α) (tail : stream) : stream}
... but not these:

coinductive stream \((\alpha : \text{Type})\)
| cons (head : \(\alpha\)) (tail : stream) : stream

inductive tree \((\alpha \beta : \text{Type})\)
| node (head : \(\alpha\)) (children : multiset tree) : tree

-- `multiset` is defined as a quotient over lists
... but not these:

```plaintext
coinductive stream (\alpha : Type)
| cons (head : \alpha) (tail : stream) : stream

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

inductive free_monad (F : Type \to Type) (\alpha : Type)
| pure : \alpha \to free_monad
| intro : F free_monad \to free_monad
```
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)
Problem:

- Lean does not have coinductive types
Problem:

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

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

Solution:
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:

- 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:

- 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:

- 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 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
The Isabelle solution is based on:

- initial F-algebra
- final F-coalgebra
The Isabelle solution is based on:

- initial F-algebra
- final F-coalgebra
- composition
The Isabelle solution is based on:

- initial F-algebra
- final F-coalgebra
- composition
- of *bounded natural functors*
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: $\alpha \to F(\alpha)$.
inductive $X$

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

is defined as $X \triangleq \text{fix}(\text{tree} \circ \text{lazy_list})$
The class of multivariate BNFs is closed under:

- composition
- initial algebras
- final coalgebras

They include finset and multiset and others.
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 $B : A \to \text{Type}$.

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

- $a : A$ as the *shape*, and
- $f : B \; a \to \alpha$ as the *contents*
Many common datatypes are (isomorphic to) polynomial functors.

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

Similarly, an element of 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.
$W$ types are easy in Lean using inductive types.
$W$ types are easy in Lean using inductive types.

```lean
inductive W (F : pfunctor) : Type
| intro {n} : ∀ a, (F.B a → W) → W
```
$M$ types are harder because Lean has no coinductive types.
Solution?
W and M types

**W and M types**


```
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)))
```
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.
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

$\text{abs} : P(\alpha) \rightarrow F(\alpha)$

and

$\text{repr} : F(\alpha) \rightarrow P(\alpha)$

satisfying

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

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

Abstraction should be a natural transformation:

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

for every $f : \alpha \rightarrow \beta$. 
Quotients of polynomial functors

```lean
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.
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 initial $F$-algebra, we need to quotient out equivalent representations.
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 $M_P$. 
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

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

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

```lean
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) := ...
```
Generated code (cont.):

```python
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) := ...
```
codef map \{\alpha \beta\} (f : \alpha \to \beta) : stream \alpha \to stream \beta
| (cons x xs) := cons (f x) (map xs)

codef nats : stream \mathbb{N} :=
cons 0 (map nat.succ nats)
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
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?
A functor $F(\alpha)$ is a \textit{bounded natural functor} provided:

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

This generalizes to multivariate functors.