Datatypes as Quotients of Polynomial Functors

Jeremy Avigad

Department of Philosophy and
Department of Mathematical Sciences
Carnegie Mellon University

joint work with Mario Carneiro and Simon Hudon
https://github.com/avigad/qpf

January 2019

Datatypes

Computer scientists love datatypes.

Datatypes

There are inductive datatypes.

```
inductive nat
| zero : nat
| succ : nat → nat

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

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

Datatypes

An inductive type is characterized by:

- The type:

  ```
  list α : Type
  ```

- The constructors:

  ```
  nil {α}  : list α
  cons {α} : α → list α → list α
  ```

- A recursor:

  ```
  list.rec {α β} : β → (α → list α → β → β) →
    list α → β
  ```

Datatypes

- defining equations:

```
list.rec b f nil          = b
list.rec b f (cons a l) = f a l (list.rec b f l)
```

- An induction principle:

```
∀ {α} (P : list α → Prop),
    P nil →
    (∀ a l, P l → P (cons a l)) →
  ∀ l, P l
```

Note: in Lean's dependent type theory,

- the recursor can map to a dependent type,
- the defining equation is a definitional equality, and
- induction is a special case of recursion.

Datatypes

There are also coinductive datatypes.

```
coinductive llist (α : Type)
| nil  : llist
| cons (head : α) (tail : llist) : llist

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

coinductive btree (α : Type)
| leaf (llabel : α) : btree
| node (nlabel : α) (left : btree) (right : btree) :
    btree
```

Datatypes

A coinductive type is characterized by:

- The type:

  ```
  stream α : Type
  ```

- The destructors:

  ```
  head {α} : stream α → α
  tail {α} : stream α → stream α
  ```

- A corecursor:

  ```
  stream.corec {α β} :
      (β → α × β) →
    β → stream α
  ```

Datatypes

- defining equations:

```
head (stream.corec f b) = (f b).1
tail (stream.corec f b) = stream.corec f (f b).2
```

- A coinduction principle:

```
∀ {α} (R : stream α → stream α → Prop),
    (∀ x y, R x y →
      head x = head y ∧
      R (tail x) (tail y)) →
  ∀ x y, R x y → x = y
```

Datatypes

Some datatypes are neither. For example:

- function types, like `a → b → c`
- subtypes, like `{x : nat // even x}`
- finite sets: `finset` α
- finite multisets: `multiset` α

In Lean, `multiset` is a quotient of `list`, and `finset` is a quotient of `multiset` (and a subtype is in fact an inductive type).

Datatypes

All these can be constructed in any reasonable foundation.

- Set theory: start with an infinite set, power set, separation, etc.
- Simple type theory: start with an infinite type, function types, and definition by abstraction.
- Dependent type theory: e.g. start with dependent function types, inductive types, and (maybe) quotient types.

Datatypes

Dataypes are intended for use in computation:

- Some constructive foundations come with a built-in computational interpretation.
- Even classical foundations can support code extraction, which is supposed to respect provable equalities.

Inductive definitions of the natural numbers go back to Frege and Dedekind, with important contributions from Tarski, Kreisel, Martin-Löf, Moschovakis, and others.

The theory of coinductive definitions was developed by Aczel, Mendler, Barwise, Moss, Rutten, Barr, Adámek, Rosický, and others.

An aside

Should mathematicians care?

- According to Kronecker, God created the natural numbers, and everything else is the work of humankind.
- Trees, finite lists (tuples), terms, formulas, etc. are combinatorial structures of interest.
- Substructures of algebraic structures are generated inductively, as is the collection of Borel sets.
- Escardó and Pavlović point out that analytic functions have a natural coinductive structure.
- Maybe a coinductive viewpoint is helpful for studying dynamical systems and processes?
- There is a nice mathematical theory of datatypes.

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

Constructors like `list α`, `finset α`, and α are *functorial*.

For example, a function `f : α → β` can be mapped over lists, giving rise to a function from `list α` to `list β`.

Category theorists write $F(\alpha)$ for the constructor and $F(f)$ for the mapping induced by f.

In Lean, to map `f` over `x` we write `f <$> x`.

This generalizes to multivariate functors $F(\alpha, \beta, \gamma, \ldots)$, like $\alpha \times \beta$.

Isabelle and BNFs

There is a literature as to the types of functors on set that have initial algebras (i.e. give rise to inductive definitions) and final coalgebras (i.e. give rise to coinductive definitions).

Not all do: for example, the powerset operation has neither.

The Isabelle group developed a notion of a *bounded natural functor* to support formalization in simple type theory.

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

This generalizes to multivariate functors.

Isabelle and BNFs

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

Examples:

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

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

Isabelle and BNFs

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

Examples:

- For stream β with head and tail, take $F(\alpha) = \beta \times \alpha$.
- For llist β, take $F(\alpha) = 1 + \beta \times \alpha$.

Coinductive definitions are *final* coalgebras.

Isabelle and BNFs

The class of multivariate BNFs is closed under:

- composition
- initial algebras
- final coalgebras

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

The modest goal of this talk: give a presentation that is

- more algebraic
- better suited to dependent type theory
- closer to computation
- pretty

Polynomial functors

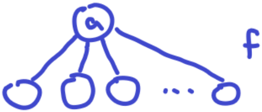
A *polynomial functor* P is one of the form

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

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, $\mathrm{list}\,\alpha \cong \Sigma\, n : \mathrm{nat},\ \mathrm{fin}\, n \to \alpha$.

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

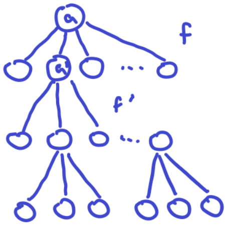
There is an obvious functorial action: $g : \alpha \to \beta$ maps (a, f) to $(a, g \circ f)$.

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 $B\ a$.



These are known as *W types*.

Polynomial functors

```
structure pfunctor :=
(A : Type u) (B : A → Type u)

def apply (α : Type*) := Σ x : P.A, P.B x → α

def map {α β : Type*} (g : α → β) :
  P.apply α → P.apply β :=
λ ⟨a, f⟩, ⟨a, g ∘ f⟩

inductive W (P : pfunctor)
| mk (a : P.A) (f : P.B a → W) : W
```

Polynomial functors

Every polynomial functor has a final coalgebra $\alpha \to P(\alpha)$.

The picture is the same, except now the trees do not have to be well-founded.

These are known as *M types*. We can construct them in Lean.

Polynomial functors

```
def M (P : pfunctor.{u}) : Type u

def M_dest : M P → P.apply (M P)

def M_corec : (α → P.apply α) → (α → M P)

def M_dest_corec (g : α → P.apply α) (x : α) :
  M_dest (M_corec g x) = M_corec g <$> g x

def M_bisim {α : Type*} (R : M P → M P → Prop)
    (h : ∀ x y, R x y → ∃ a f g,
      M_dest x = ⟨a, f⟩ ∧
      M_dest y = ⟨a, g⟩ ∧
      ∀ i, R (f i) (g i)) :
  ∀ x y, R x y → x = y
```

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

$$abs : P(\alpha) \to F(\alpha)$$

and

$$repr : F(\alpha) \to P(\alpha)$$

satisfying

$$abs(repr(x)) = x$$

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

Abstraction should be a natural transformation:

$$abs \circ P(f) = F(f) \circ abs$$

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.

What extra assumptions do we need to do the same constructions?

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.

We were able to define the equivalence relation from the bottom up, using an analogue of the BNF congruence axiom.

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.

We were able to define the equivalence relation from the bottom up, using an analogue of the BNF congruence axiom.

Then we found an alternative definition that avoids it.

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.

Preservation of weak pullbacks is needed to show that a composition is bisimulations is a bisimulation.

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.

Preservation of weak pullbacks is needed to show that a composition is bisimulations is a bisimulation.

But once again, using an alternative construction by Aczel and Mendler, we were able to find a construction that avoids the extra assumption.

Quotients of polynomial functors

The remarkable conclusion: we don't need any more assumptions.

The class of qpfs is closed under:

- composition
- quotients
- initial algebras
- final colagebras

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

The constructions are pretty.

Quotients of polynomial functors

```
structure pfunctor := (A : Type u) (B : A → Type u)

def apply (α : Type*) := Σ x : P.A, P.B x → α

def map {α β : Type*} (g : α → β) :
  P.apply α → P.apply β := λ ⟨a, f⟩, ⟨a, g ∘ f⟩

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)
```

Quotients of polynomial functors

```
def fix (F : Type u → Type u) [functor F] [qpf F]

def fix.mk : F (fix F) → fix F

def fix.rec {α : Type*} (g : F α → α) : fix F → α

theorem fix.rec_eq {α : Type*}
    (g : F α → α) (x : F (fix F)) :
  fix.rec g (fix.mk x) = g (fix.rec g <$> x)

theorem fix.ind {α : Type*} (g₁ g₂ : fix F → α)
    (h : ∀ x : F (fix F), g₁ <$> x = g₂ <$> x →
      g₁ (fix.mk x) = g₂ (fix.mk x)) :
  ∀ x, g₁ x = g₂ x
```

Quotients of polynomial functors

```
def cofix (F : Type u → Type u) [functor F] [qpf F]

def cofix.dest : cofix F → F (cofix F)

def cofix.corec {α : Type*} (g : α → F α) : α → cofix F

theorem cofix.dest_corec {α : Type u}
  (g : α → F α) (x : α) :
cofix.dest (cofix.corec g x) = cofix.corec g <$> g x

theorem cofix.bisim
    (r : cofix F → cofix F → Prop)
    (h : ∀ x y, r x y →
      quot.mk r <$> cofix.dest x =
      quot.mk r <$> cofix.dest y) :
  ∀ x y, r x y → x = y
```

Quotients of polynomial functors

```
def comp
    {G : Type u → Type u} [functor G] [qpf G]
    {F : Type u → Type u} [functor F] [qpf F] :
  qpf (functor.comp G F)
```

Quotients of polynomial functors

```
def quotient_qpf
    {F : Type u → Type u} [functor F] [qpf F]
    {G : Type u → Type u} [functor G]
    {abs  : Π {α}, F α → G α}
    {repr : Π {α}, G α → F α}
    (abs_repr : Π {α} (x : G α), abs (repr x) = x)
    (abs_map  : ∀ {α β} (f : α → β) (x : F α),
                  abs (f <$> x) = f <$> abs x) :
  qpf G
```

Quotients of polynomial functors

Mario, Simon, and I are working on a datatype package for Lean based on qpfs.

- The unary constructions are worked out.
- We are working on the multivariate ones.
- We are working on a parser and front end.

Other things to talk about:

- the constructions
- computational properties and quotients
- lifting relations and preservation under weak pullbacks
- the role of Fset
- multivariate constructions