Data Types as Quotients of Polynomial Functors

Jeremy Avigad
Department of Philosophy, Carnegie Mellon University, Pittsburgh, PA, USA
http://www.andrew.cmu.edu/user/avigad/
avigad@cmu.edu

Mario Carneiro
Department of Philosophy, Carnegie Mellon University, Pittsburgh, PA, USA
di.gama@gmail.com

Simon Hudon
Department of Philosophy, Carnegie Mellon University, Pittsburgh, PA, USA
https://www.cmu.edu/dietrich/philosophy/people/postdoc-fellows/simon-hudon%20.html
simon.hudon@gmail.com

Abstract

A broad class of data types, including arbitrary nestings of inductive types, coinductive types, and quotients, can be represented as quotients of polynomial functors. This provides perspicuous ways of constructing them and reasoning about them in an interactive theorem prover.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory; Theory of computation → Data structures design and analysis

Keywords and phrases data types, polynomial functors, inductive types, coinductive types

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.6

Supplement Material Lean formalizations are online at `https://github.com/avigad/qpf`.

Funding Work partially supported by AFOSR grant FA9550-18-1-0120 and the Sloan Foundation.

Acknowledgements We are grateful to Andrei Popescu, Dmitriy Traytel, and Jasmin Blanchette for extensive discussions and very helpful advice.

1 Introduction

Data types are fundamental to programming, and theoretical computer science provides abstract characterizations of such data types and principles for reasoning about them. For example, an *inductive type*, such as the type of lists of elements of type α, is freely generated by its constructors:

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

Such a declaration gives rise to a type constructor, `list`, constructors `nil` and `cons`, and a recursor:

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

The recursor satisfies the following equations:

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

We also have an induction principle:

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

In words, to prove that a property holds of all lists, it is enough to show that it holds of the empty list and is preserved under the cons operation. Here we have adopted the syntax of the Lean theorem prover [19], so the curly braces around the type arguments α and β indicate that these arguments are generally left implicit and inferred from context. The type of the variable `P`, namely `list` $\alpha \to$ `Prop`, indicates that it is a predicate on lists.

Dual to the notion of an inductive type is the notion of a *coinductive* type, such as the type of streams of elements of α:

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

Our syntax is similar to that used for the inductive declaration, but the fundamental properties of such a type can be expressed naturally in terms of the *destructors* rather than the constructors. Roughly speaking, when one observes a stream of elements of α, one sees an element of α, the *head*, and another stream, the *tail*. In addition to the type constructor `stream` and the destructors `head` and `tail`, one obtains a corecursor:

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

It satisfies these defining equations:

```
head (stream.corec f b) = fst (f b)
tail (stream.corec f b) = stream.corec f (snd (f b))
```

We also have 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
```

Intuitively, the corecursor allows one to construct a stream from an element b of β by giving an element of α, the head, and another element of β to continue the construction. The coinductive principle says we can prove two streams are equal by showing that they satisfy a relation on streams that implies the heads are the same and the tails are again related.

Inductive definitions date to the beginning of modern logic, with inductive characterizations of the natural numbers by Dedekind [20] and Frege [24], and generalizations by Knaster [31], Tarski [39], Kreisel [32], and many others (e.g. [3]). The general study of coinductive definition seems to have originated with Aczel [3, 4], and has since been extended by many others (e.g. [6, 9, 37]).

The algebraic study of data types begins with the observation that many constructions are *functorial* in their arguments. For example, an element x of $\mathsf{list}(\alpha)$ and a function $f : \alpha \to \beta$ give rise to an element of $\mathsf{list}(\beta)$, the result of applying f to each element in the list. Similarly, products $\alpha \times \beta$ and sums $\alpha + \beta$ are functorial in either argument. In category-theoretic notation, given a functor $F(\alpha)$, one would write $F(f)$ to denote the map from $F(\alpha)$ to $F(\beta)$. In Lean, given a functor F, we can write `f <$> x` to denote $F(f)(x)$, since the system can infer `F` from the type of `x`, namely, `F` α. In this paper, we will generally use category-theoretic notation and the language of set-valued functors when talking about the general constructions, to be consistent with the general literature. When we focus on dependent type theory and our formalizations, however, we will resort to type-theoretic syntax and take α to be a type rather than a set.

For any functor F from sets to sets, a function $F(\alpha) \to \alpha$ is known as an *F-algebra*, and specifying an inductive definition amounts to specifying such an algebra. For example, the natural numbers are defined by a constant $0 \in \mathbb{N}$ and a function from $\mathbb{N}$ to $\mathbb{N}$. Putting

these together yields a function $1 + \mathbb{N} \to \mathbb{N}$, where 1 denotes a singleton set and $+$ denotes a disjoint union. So the constructors for $\mathbb{N}$ amount to a function $F(\mathbb{N}) \to \mathbb{N}$, where $F(\alpha)$ is the functor $1 + \alpha$. The inductive character of the natural numbers means that $\mathbb{N}$ is an *initial* F-algebra, in the sense that for any F-algebra $F(\alpha) \to \alpha$, there is a function rec from $\mathbb{N}$ to α such that the following square commutes:

$$\begin{array}{ccc} F(\mathbb{N}) & \xrightarrow{F(\mathsf{rec})} & F(\alpha) \\ \downarrow & & \downarrow \\ \mathbb{N} & \xrightarrow{\mathsf{rec}} & \alpha \end{array}$$

Similarly, $\mathsf{list}(\alpha)$ is an initial algebra for the functor $F(\beta) = 1 + \alpha \times \beta$. Dually, a *coalgebra* for a functor $F(\alpha)$ is a function $\alpha \to F(\alpha)$. A coinductive definition corresponds to a *final coalgebra*, that is, a coalgebra $\lambda \to F(\lambda)$ with the property that for any colagebra $\alpha \to F(\alpha)$, there is a map from $\alpha \to \lambda$ making the corresponding square commute.

The question is then: Which set-valued functors have initial algebras and final coalgebras? Not all do: initial algebras and final coalgebras are both fixed points (that is, satisfy $F(\alpha) \simeq \alpha$), so the usual diagonalization argument shows that the power-set functor has neither. A sufficient condition for the existence of both is that the functor F is κ-*bounded* for some cardinal κ [6, 37].

To formalize these constructions in the context of simple type theory, developers of the *Isabelle* theorem prover [35] proposed the notion of a *bounded natural functor* [12, 15], or *BNF* for short. A functor $F(\alpha)$ is a BNF if it satisfies the following:

- There is a natural transformation set which for each α maps elements of $F(\alpha)$ to elements of the power set of α, such that for any pair of maps $f, g : \alpha \to \beta$ and any element of x of $F(\alpha)$, if f and g agree on $\mathsf{set}(x)$, then $F(f)(x) = F(g)(x)$.
- There is a fixed cardinal $\kappa \geq \aleph_0$ such that for every x, $|\mathsf{set}(x)| \leq \kappa$.
- F preserves weak pullbacks (see Section 5).

The generalization to multivariate functors is straightforward. BNFs are closed under composition and formation of initial and final coalgebras, and the class of BNFs includes data type constructions such as finite sets and finite multisets. This forms the basis for a powerful and extensible data type package [12, 15].

Here we present a variation on the BNF constructions based on the notion of a *quotient of a polynomial functor*, or QPF for short. Like BNFs, QPFs support definitions such as the following, which defines a well-founded tree on α to consist of a node labeled by an element of α together with a finite set of subtrees :

```
inductive tree (α : Type)
| mk : α → finset tree → tree
```

Here finset can be replaced by list, multiset, or stream, or, indeed, any QPF constructor. Moreover, replacing `inductive` by `coinductive` yields the corresponding coinductive type of arbitrary trees, not just the well-founded ones.

QPFs are more general than BNFs in a sense we will make precise in Section 5, but their main appeal is that they provide another perspective on the BNF constructions, and are amenable to formalization. Our approach is well-suited to dependent type theory, and the components of our constructions, including polynomial functors, W types, M types, and quotients, are familiar inhabitants of the type theory literature. At the same time, the use of dependent types is mild, and the constructions can easily be translated to the language of set theory or simple type theory.

We have found that working with QPFs is natural and intuitive. Indeed, after hitting upon the notion, we discovered that Adámek and Porst [5, Proposition 5.2] have shown that a functor is accessible if and only if it is a quotient of a polynomial functor (see also [6, Example 6.4]). But we have not seen constructions of initial algebras and final coalgebras carried out directly in these terms, and the approach via QPFs makes them easy to understand. We expect that the QPF perspective will also be conducive to generalizations, as discussed in Section 7.

The constructions described in this paper have been formalized in Lean and are available online, as indicated below the abstract to this paper. Lean's underlying logical framework, like that of Coq, is a variant of the *Calculus of Inductive Constructions*, which has a computational interpretation. This version provides non-nested inductive type families with only primitive recursors, following the specification of Dybjer [21]. On top of the core logic, Lean's library includes propositional extensionality, quotient types [18, 8], and a classical choice principle [8]. Our constructions make use of the first two, which imply function extensionality. We had to use the choice principle in only one place, when constructing the QPF instance for the final coalgebra of a multivariate constructor. We believe the use of Lean's built-in quotient types could be avoided, but we do not know whether it is possible to avoid propositional and function extensionality.

We are in the process of implementing a data type package for Lean based on these constructions. Lean currently supports nested inductive definitions, compiling them down to indexed inductive definitions and compiling recursive definitions down to well-founded recursion along a synthesized measure of size. Our approach, like the Isabelle approach, adds a wealth of new constructions: not only coinductive definitions, but also arbitrary nestings of inductive definitions, coinductive definitions, and quotient constructions. Moreover, it provides principles of recursion and corecursion based on the associated functorial map.

2 Polynomial functors

Let us start with the notion of a *polynomial functor*, also known as a *container* [1, 2, 25]. These are functors of the form $P(\alpha) = \Sigma_{x \in A} (B(x) \to \alpha)$, where B denotes a family of sets $(B(x))_{x \in A}$ and Σ denotes the *dependent sum*. Thus, an element of $P(\alpha)$ is a pair (a, f) with $a \in A$ and $f : B(a) \to \alpha$. Think of (a, f) as representing a structured object with data from α, where $a \in A$ specifies the *shape* of the element and $f \in B(a) \to \alpha$ specifies its *contents*. In the literature on containers, the polynomial functor given by the data A and B is usually denoted $A \triangleright B$. There is an obvious functorial action: given $g : \alpha \to \beta$, $P(g)$ maps (a, f) in $P(\alpha)$ to $(a, g \circ f)$ in $P(\beta)$, preserving the shape while transforming the contents. Below, we will say more generally that P is a polynomial functor if it is *isomorphic* to one of this form. It is easy to define these in Lean:

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

variable P : pfunctor.{u}

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

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

In these definitions, `Type u` denotes a fixed but arbitrary universe of types. Lean's projection notation is a convenient syntactic device: since `P` has type `pfunctor`, Lean interprets `P.apply` as `pfunctor.apply P`. Similarly, the corner brackets denote *anonymous constructors*: since

`P.apply` reduces to a sigma type, Lean interprets ⟨`a, f ∘ g`⟩ as `sigma.mk a (f ∘ g)`. We also make use of a pattern-matching lambda, which translates the variables in the bound pattern to applications of destructors of a single bound variable.

Many familiar data types are polynomial functors. For instance, any list of elements of α is given by its length, n, and a function from $\{0, \ldots, n-1\}$ to α. Streams of elements of α have only one shape, and the contents are functions from $\mathbb{N}$ to α. The type of lazy lists of elements of α can be seen as the disjoint union of these two, so the set of shapes is the disjoint union of $\mathbb{N}$ and a singleton. A tree with nodes labeled by α has as shape the unlabeled tree and as contents a map from the nodes to α.

It is not hard to show that if $P(\alpha)$ and $Q(\alpha)$ are polynomial functors, then so is their composition, $P(Q(\alpha))$. Moreover, every polynomial functor P has an initial algebra, a familiar construct in dependent type theory known as a W *type* [34]. Elements of the data type W_P corresponding to P can be viewed as well-founded trees in which every node has a label a from A and children indexed by $B(a)$. In other words, an element of W_P is given by an element a in A and a function $f : B(a) \to W_P$. Such inductive types are given axiomatically by Lean's type-theoretic foundation, and can be declared as follows:

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

The constructor forms an element of W from $a \in A$ and $f : B(a) \to W$. This is just a variant of the usual algebra map $P(W) \to W$ in which the argument, an element of $P(W)$, has been replaced by its two components. The built-in recursion principle for the type above says exactly that this map is the initial algebra.

Every polynomial functor P also has a final coalgebra, known as the associated M *type*. The data type M_P has the same description as above except that the trees are no longer required to be well founded. Abstractly, M types can be specified as follows:

```
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)

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

theorem M_bisim (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
```

The principle `M_bisim` is a coinduction principle for trees. Here the anonymous constructor $\langle a, f \rangle$ is used to represent an element of $P(M)$ in terms of $a : A$ and $f : B(a) \to M$. A *bisimulation* relation between trees is a relation r such that $r(x, y)$ holds if and only if x and y have the same branching type at the top node and the children at the top node are again pointwise related by r. Two trees are *bisimilar* if there is a bisimulation between them, and the principle `M_bisim` says that any two trees that are bisimilar are in fact equal.

M types are not given axiomatically by the Calculus of Inductive Constructions, but as the specification above suggests, they can be defined in Lean. One way to go about it is to define for each n the type of trees of depth at most n, using a special node to denote potential continuations at the leaves:

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

We can then say what it means for an approximation of depth n to agree with one of depth $n+1$, and define an element of the M type to be a sequence of approximations such that each is consistent with the next:

```
inductive agree : ∀ {n : ℕ}, M_approx P n → M_approx P (n+1) → Prop
| continue (x : M_approx P 0) (y : M_approx P 1) : agree x y
| intro {n} {a} (x : P.B a → M_approx P n) (y : P.B a → M_approx P (n+1)) :
  (∀ i, agree (x i) (y i)) → agree (M_approx.intro a x) (M_approx.intro a y)

structure M := (approx : Π n, M_approx P n) (agrees : ∀ n, agree (x n) (x (n+1)))
```

We will see in Section 4 that these considerations extend to the multivariate case. In other words, there is a natural notion of an n-ary polynomial functor, and if $P(\vec{\alpha}, \beta)$ is an $(n+1)$-ary polynomial functor, then for each fixed tuple $\vec{\alpha}$, it has an initial algebra $W(\vec{\alpha})$ and a final coalgebra $M(\vec{\alpha})$. Moreover, $W(\vec{\alpha})$ and $M(\vec{\alpha})$ are polynomial functors in $\vec{\alpha}$.

In short, polynomial functors are closed under composition, initial algebras, and final coalgebras, and so seem to have all the virtues of BNFs. Why not take them as the basis for a data type package?

The answer is that the class of polynomial functors is not as general as the class of BNFs. For example, although the type $\mathsf{finset}(\alpha)$ of finite sets of elements of α and the type $\mathsf{multiset}(\alpha)$ of finite multisets of elements of α are both BNFs, cardinality considerations can be used to show that they are not polynomial functors. For another view of what goes wrong, note that if we map the finite set $\{1, 2\}$ under a function which sends both 1 and 2 to 3, we get the set $\{3\}$, which does not have the same shape.

But we can view $\mathsf{finset}(\alpha)$ as a *quotient* of a polynomial functor, namely, the quotient of $\mathsf{list}(\alpha)$ that identifies any two lists that have the same elements. Similarly, we can view $\mathsf{multiset}(\alpha)$ as the quotient of $\mathsf{list}(\alpha)$ by equivalence up to permutation. This points the way to a solution: rather than consider only polynomial functors, we should consider their quotients as well.

3 Quotients of polynomial functors

A natural way to say that a functor $F(\alpha)$ is a quotient of the polynomial functor $P(\alpha)$ is to say that, for every α, there is a surjective function abs from $P(\alpha)$ to $F(\alpha)$. Think of elements of $P(\alpha)$ as being concrete representations of more abstract objects in $F(\alpha)$. We can express the fact that abs is surjective by supplying a right inverse, repr, which maps any element of $F(\alpha)$ to some representative in $P(\alpha)$. Finally, we should assert that abs is a natural transformation between P and F, which is to say, it respects their functorial behavior:

$$\begin{array}{ccc} P(\alpha) & \xrightarrow{P(f)} & P(\beta) \\ {\scriptstyle \mathsf{abs}_\alpha}\downarrow & & \downarrow{\scriptstyle \mathsf{abs}_\beta} \\ F(\alpha) & \xrightarrow{F(f)} & F(\beta) \end{array}$$

Remember that given $f : \alpha \to \beta$, there is a map $P(f)$ from $P(\alpha)$ to $P(\beta)$. We require this map to be preserved by abs, so that $\mathsf{abs}_\beta \circ P(f) = F(f) \circ \mathsf{abs}_\alpha$. In other words, mapping a representation x and then abstracting it should yield the same result as abstracting it and then mapping it, making the square above commute.

In Lean, we can specify that F is a quotient of a polynomial functor as follows:

```
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)
```

One can show that every BNF can be represented in this way. (Briefly, if κ is the relevant cardinal bound, we can take the shapes in the polynomial functor to be the set of pairs of the form $(I, F(I))$ with $I \subseteq \kappa$, and the contents of such a shape to be indexed by I.)

To see that every QPF has an initial algebra, suppose that F is a quotient of P in the sense above. Let W be the initial P-algebra. We want to construct the least fixed point fix of F. By initiality, there is an isomorphism between W and $P(W)$, so every tree in W can be viewed as consisting of a shape, $a \in A$, and a sequence $f : B(a) \to W$ of subtrees. Via the abs function, these represent an element of $F(W)$. The problem is that multiple elements of $P(W)$ can represent the same element of $F(W)$, so that multiple elements of W can represent the same element of $F(W)$. So already W looks like an overapproximation to fix. Moreover, recursively, under the functorial map for F, $F(W)$ is again an overapproximation to $F(\mathsf{fix})$.

The solution is to say what it means for two elements of W to represent the same element of fix, and then define fix to be the quotient of W by that equivalence relation. The relation we are after is the smallest equivalence relation closed under the following two rules:

$$\frac{\mathsf{abs}(a, f) = \mathsf{abs}(a', f')}{\langle a, f\rangle \equiv \langle a', f'\rangle} \qquad \frac{\forall x\ (f(x) \equiv f'(x))}{\langle a, f\rangle \equiv \langle a, f'\rangle}$$

The key condition is the first one, which says that two trees represented by (a, f) and (a', f') are equivalent if their abstractions are the same element of $F(W)$. The second clause extends the relation recursively to trees with the same shape and equivalent subtrees. The relation is defined inductively in Lean as follows:

```
inductive Wequiv : q.P.W → q.P.W → Prop
| abs (a : q.P.A) (f : q.P.B a → q.P.W) (a' : q.P.A) (f' : q.P.B a' → q.P.W) :
    abs ⟨a, f⟩ = abs ⟨a', f'⟩ → Wequiv ⟨a, f⟩ ⟨a', f'⟩
| ind (a : q.P.A) (f f' : q.P.B a → q.P.W) :
    (∀ x, Wequiv (f x) (f' x)) → Wequiv ⟨a, f⟩ ⟨a, f'⟩
| trans (u v w : q.P.W) : Wequiv u v → Wequiv v w → Wequiv u w
```

Notationally, here `q` is the relevant QPF structure, `q.P` is the polynomial functor, and `q.P.W` denotes the associated `W` construction, a function of `q.P`. The third clause ensures that the relation is transitive, and hence an equivalence relation. We then define fix to be the quotient:

```
def fix (F : Type u → Type u) [functor F] [q : qpf F] :=
quotient (Wsetoid : setoid q.P.W)
```

`Wsetoid` bundles `Wequiv` with a proof that the latter is an equivalence relation. Any function $g : F(W) \to \beta$ gives rise to a function $g' : P(W) \to \beta$ defined by $g' = g \circ \mathsf{repr}$, and so we can use g to define functions by recursion on W:

```
def recF {α : Type u} (g : F α → α) : q.P.W → α
| ⟨a, f⟩ := g (abs ⟨a, λ x, recF (f x)⟩)
```

This is just an ordinary recursion on W, using repr to mediate the difference between P and F. Moreover, any function defined by such a recursion will respect the equivalence relation `Wequiv`, and so lifts to a function from fix to α.

```
def fix.rec {α : Type u} (g : F α → α) : fix F → α :=
quot.lift (recF g) (recF_eq_of_Wequiv g)
```

We can map any tree W to a canonical representative, in such a way that any two equivalent trees are mapped to the same representative. This gives us a choice-free way of mapping fix back to W. Composing maps $F(\mathsf{fix}) \to P(\mathsf{fix}) \to P(W) \to W \to \mathsf{fix}$ gives us the desired constructor. With these definitions, we can prove:

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

theorem fix.ind_rec {α : Type u} (g g' : fix F → α)
    (h : ∀ x : F (fix F), g <$> x = g' <$> x → g (fix.mk x) = g' (fix.mk x)) :
  ∀ x, g x = g' x

theorem fix.ind (p : fix F → Prop)
    (h : ∀ x : F (fix F), liftp p x → p (fix.mk x)) :
  ∀ x, p x
```

The last theorem above expresses the induction principle, defined in terms of a predicate lifting operation defined in Section 5. The second-to-last theorem implies the uniqueness of functions satisfying the defining equations for the recursor.

With the recursor, we can then define a destructor from fix to $F(\mathsf{fix})$ and prove that it is an inverse to the constructor. This completes the construction of the initial algebra for any unary quotient of polynomial functors.

We can analogously construct the greatest fixed point of $F(\alpha)$ as a suitable quotient of M_P. Remember that the M-type analogue of the principle of induction on a W type is the bisimulation principle, `M_bisim`, presented at the end of the last section. The corresponding version for the final algebra should look like this:

```
theorem cofix.bisim (r : cofix F → cofix F → Prop)
    (h : ∀ x y, r x y → liftr r (cofix.dest x) (cofix.dest y) :
  ∀ x y, r x y → x = y
```

The function liftr in the statement of the principle refers to the canonical method of lifting a binary relation r on α to a relation $\hat{r}$ on $F(\alpha)$, described in Section 5. One strategy of constructing the final coalgebra cofix, familiar from the literature on set-valued functors (e.g. [37]), is to define the relation R to be the union of all bisimulation relations on the underlying M type, and then define cofix to be the the quotient M/R. For that proof to go through, we need to know that the union of all bisimulation relations is an equivalence relation, which in turn requires showing that the composition of bisimulation relations is again a bisimulation. And *that* can be shown as a consequence of the fact that the function $F(\alpha)$ preserves weak pullbacks. This explains why this assumption appears in Isabelle's definition of BNFs.

Going back to an early paper by Aczel and Mendler [4], however, we were able to find a construction that avoids the additional assumption. The trick is to use an alternative notion of lift for binary relations on a set α. Given a binary relation r on α, let q_r be the quotient map corresponding to the least equivalence relation on α that includes r. We can then define the alternative notion of lift which holds of x and y in $F(\alpha)$ if and only if $F(q_r)(x) = F(q_r)(y)$. In other words, rather than lift the relation, we map the quotient function. We now define two elements of M to bear this lifted version of r if their F-abstractions do, and define cofix to be the quotient under the union of all such relations.

```
def is_precongr (r : q.P.M → q.P.M → Prop) : Prop :=
∀ {x y}, r x y → abs (quot.mk r <$> M_dest x) = abs (quot.mk r <$> M_dest y)

def Mcongr : q.P.M → q.P.M → Prop := λ x y, ∃ r, is_precongr r ∧ r x y

def cofix (F : Type u → Type u) [functor F] [q : qpf F]:= quot (@Mcongr F _ q)
```

It is especially convenient that Lean's fundamental quotient construction, `quot.mk r`, does not require `r` to be an equivalence relation. (The axioms governing quotients in Lean imply that the result is equivalent to quotienting by the equivalence relation generated by r.) We can show that quotient by a finer relation factors through the quotient by a coarser one:

```
def factor {α : Type*} (r s : α → α → Prop) (h : ∀ x y, r x y → s x y) :
  quot r → quot s :=
quot.lift (quot.mk s) (λ x y rxy, quot.sound (h x y rxy))

def factor_mk_eq {α : Type*} (r s: α → α → Prop) (h : ∀ x y, r x y → s x y) :
  factor r s h ∘ quot.mk r = quot.mk s := rfl
```

With this fact, we can use the bisimulation principle on M to derive the bisimulation principle on the quotient. When the dust settles, we have all the desired functions and properties:

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

def cofix.corec {α : Type u} (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_rel (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
```

Since identity under the mapped quotients of r is implied by the lift of r, this formulation of the bisimulation principle implies cofix.bisim above, as well as the following variation:

```
theorem cofix.bisim' {α : Type u} (q : α → Prop) (u v : α → cofix F)
    (h : ∀ x, q x → ∃ a f f',
      cofix.dest (u x) = abs ⟨a, f⟩ ∧
      cofix.dest (v x) = abs ⟨a, f'⟩ ∧
      ∀ i, ∃ x', q x' ∧ f i = u x' ∧ f' i = v x') :
  ∀ x, q x → u x = v x
```

It is, moreover, straightforward to show that quotients of polynomial functors are closed under composition and quotients:

```
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)

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
```

In short, we have shown that unary QPFs support the same constructions as unary BNFs. We now turn to the multivariate case.

4 Multivariate constructions

A ternary functor on $F(\alpha, \beta, \gamma)$ on sets is one that is functorial in each argument. Our goal is to extend the notion of a QPF to such functors, and, indeed, functors of arbitrary arity. In this respect, dependent type theory offers a distinct advantage over simple type theory: whereas Isabelle's BNF package has to synthesize definitions of n-ary functors dynamically for each n, in dependent type theory we can treat an n-tuple of types as a first-class object parameterized by n. This facilitates the implementation of a data type package, as discussed in Section 6.

Formally, we define an n-tuple of types to be a function from a canonical finite type $\mathsf{fin}(n)$ of elements to an arbitrary type universe:

```
def typevec (n : ℕ) := fin n → Type*
```

We can then define the usual morphisms on the category of n-tuples, namely, n-tuples of functions, with composition and identity.

```
def arrow (α β : typevec n) := Π i : fin n, α i → β i

infixl ` ⟹ `:40 := arrow

def id {α : typevec n} : α ⟹ β := λ i x, x

def comp {α β γ : typevec n} (g : β ⟹ γ) (f : α ⟹ β) : α ⟹ γ :=
λ i x, g i (f i x)

infixr ` ⊙ `:80 := typevec.comp
```

Lean's notions of *functor* (a type constructor with a map function) and *lawful functor* (a functor satisfying the usual laws) carry over straightforwardly to the multivariate setting:

```
class mvfunctor {n : ℕ} (F : typevec n → Type*) :=
(map : Π {α β : typevec n}, (α ⟹ β) → (F α → F β))

infixr ` <$$> `:100 := mvfunctor.map

class is_lawful_mvfunctor {n : ℕ} (F : typevec n → Type*) [mvfunctor F] :=
(id_map   : Π {α : typevec n} (x : F α), id <$$> x = x)
(comp_map : Π {α β γ : typevec n} (g : α ⟹ β) (h : β ⟹ γ) (x : F α),
                (h ⊙ g) <$$> x = h <$$> g <$$> x)
```

Notice that we use the notation `f <$$> x` to denote the functorial map of the n-tuple of functions f on the element x, where x is an element of the multivariate $F(\vec{\alpha})$. With these definitions and notation, the definition of a multivariate QPF is almost exactly the same as the definition of a unary one:

```
class mvqpf {n : ℕ} (F : typevec.{u} n → Type*) [mvfunctor F] :=
(P         : mvpfunctor.{u} n)
(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)
```

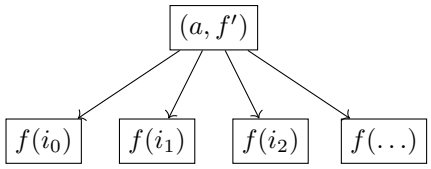
We need to show that if $F(\vec{\alpha}, \beta)$ is an $(n+1)$-ary QPF, then for each tuple $\vec{\alpha}$ it has both an initial algebra $\mathsf{fix}(\vec{\alpha})$ and a final coalgebra $\mathsf{cofix}(\vec{\alpha})$, and, moreover, that the latter are n-ary functors in $\vec{\alpha}$. The constructions require operations $\mathsf{append1}(\vec{\alpha}, \beta)$ for extending an n-tuple of types $\vec{\alpha}$ by a single type β, and operations $\mathsf{drop}(\vec{\alpha})$ and $\mathsf{last}(\vec{\alpha})$ that return the initial n-tuple and final elements of such an $(n+1)$-tuple. Similarly, we need an operation $\mathsf{append\text{-}fun}(f, g)$ that appends a function to an n-tuple of functions, and operations $\mathsf{drop\text{-}fun}$ and $\mathsf{last\text{-}fun}$ that destruct the resulting $(n+1)$-tuple. One minor problem is that constructions like these sometimes give rise to types that are provably but not definitionally equal. For example, $\mathsf{append1}(\mathsf{drop}(\vec{\alpha}), \mathsf{last}(\alpha))$ is provably equal to $\vec{\alpha}$, but we need an explicit cast from one to the other if we want expressions to type check. Such difficulties were mild, and they were a small price to pay for the benefits of being able to reason about arbitrary tuples uniformly.

With a formal theory of tuples of types and maps between them, unary notions carry over nicely to the multivariate setting. The definition of a multivariate polynomial functor $P(\vec{\alpha})$ is straightforward:

```
structure mvpfunctor (n : ℕ) := (A : Type.{u}) (B : A → typevec.{u} n)

variables {n : ℕ} (P : mvpfunctor.{u} n)

def apply (α : typevec.{u} n) : Type u := Σ a : P.A, P.B a ⟹ α

def map {α β : typevec n} (f : α ⟹ β) : P.apply α → P.apply β :=
λ ⟨a, g⟩, ⟨a, f ⊙ g⟩
```

As before, we can think of an element of $P(\vec{\alpha})$ as consisting of a shape, a, and a map $f : B(a) \Longrightarrow \vec{\alpha}$. All that has changed is that the contents now consist of tuples of functions.

Given an $(n+1)$-ary polynomial functor $P(\vec{\alpha}, \beta)$, we need to construct its initial algebra, $W(\vec{\alpha})$, and show that it is again a polynomial functor. Intuitively, each element of $W(\vec{\alpha})$ is a well-founded tree, in which each node is labeled by an element of A together with a function $f' : \mathsf{drop}(B(a)) \Longrightarrow \vec{\alpha}$, and the children of that node are given by a function $f : \mathsf{last}(B(a)) \to W(\vec{\alpha})$:



There are various ways to view such a tree. One is as an ordinary unary W type, where the set of shapes at each node is given by $A' = \Sigma_{a \in A}(\mathsf{drop}(B(a)) \to \vec{\alpha}$. Given an element $p = (a, f')$ of A', the set of indices $B'(p) = \mathsf{last}(B(p.\mathsf{fst}))$ depends only on the first component. This, however, introduces an artificial dependency of the index set of the branches on the contents f'. A slightly modified description is to view the $W(\vec{\alpha})$ as given inductively by the following constructor and recursion principle:

```
def W_mk {α : typevec n} (a : P.A) (f' : P.drop.B a ⟹ α)
  (f : P.last.B a → P.W α) : P.W α

def W_rec {α : typevec n} {C : Type*}
  (g : Π a : P.A, ((P.drop).B a ⟹ α) → ((P.last).B a → P.W α) →
    ((P.last).B a → C) → C) : P.W α → C
```

In words, an element of $W(\vec{\alpha})$ is given inductively by a triple (a, f', f) where a is in A, f' is a tuple of functions from $\mathsf{drop}(B(a))$ to $\vec{\alpha}$, and f' is a function from $\mathsf{last}(B(a))$ to $W(\vec{\alpha})$. The induction principle and defining equations for the recursor are as expected.

An alternative view of $W(\vec{\alpha})$ makes it clear that it is a polynomial functor. As the picture suggests, we can think of an element of $W(\vec{\alpha})$ as having the shape of a well-founded tree with labels from A, with children at a node labeled a indexed by the set $\mathsf{last}(B(a))$. In other words, the shape is just the ordinary W type given by these data. The *contents* of the tree amount to the sum total of all the functions f' at each node. We can combine these into one big function from the disjoint union of all the index sets $\mathsf{drop}(B(a))$ at all the nodes. This disjoint union can be conveniently described by an inductive definition:

```
inductive W_path : P.last.W → fin n → Type u
| root (a : P.A) (f : P.last.B a → P.last.W) (i : fin n) (c : P.drop.B a i) :
    W_path ⟨a, f⟩ i
| child (a : P.A) (f : P.last.B a → P.last.W) (i : fin n) (j : P.last.B a)
    (c : W_path (f j) i) : W_path ⟨a, f⟩ i
```

Here, `P.last` denotes the polynomial functor just described, and `P.last.W` is the corresponding W type, which we take to be the shape of $W(\vec{\alpha})$. The type `W_path` describes the index set associated to this shape as the sum of the index sets `P.drop.B a i` for each $i < n$, together with the index sets assigned to all the children. This gives us the desired representation of $W(\vec{\alpha})$ as a multivariate polynomial functor:

```
def Wp : mvpfunctor n := { A := P.last.W, B := P.W_path }
```

There are two things worth noting here. First, the type of `W_path` is equivalent to `P.last.W` $\implies$ `typevec n`. This makes use of the specific representation of `typevec n`, but this use is not essential; with another representation, we could still define `W_path` as above, and then compose it with the relevant isomorphism. The second thing to note is that the analogous inductive definition works just as well for M types, since it does not require the tree to be well founded.

Both characterizations of $W(\vec{\alpha})$ are essential. The first description, the inductive one, allows us to carry out the construction of the initial algebra. The second description of $W(\vec{\alpha})$, as a polynomial functor, enables us to show that the initial algebra of a QPF is again a QPF. Rather than define both objects and prove them isomorphic, we found it more convenient to take the second description to be the official definition of $W(\vec{\alpha})$ and use that to define the constructor and recursor specified by the first description.

Coordinating the different notions of a polynomial functor was the most difficult part of extending the constructions from the unary to the multivariate setting. With these characterizations of $W(\vec{\alpha})$, the construction of the initial algebra $\mathsf{fix}(\vec{\alpha})$ of a multivariate QPF $F(\vec{\alpha}, \beta)$ is almost line-by-line the same as the construction in the unary case, replacing unary primitives with their multivariate counterparts. Suppose $F(\vec{\alpha}, \beta)$ is a quotient of the polynomial functor of $P(\vec{\alpha}, \beta)$. The associated $W(\vec{\alpha})$ is again a polynomial functor, and $\mathsf{fix}(\vec{\alpha})$ is defined as a quotient of that. It is not hard to define the map on $\mathsf{fix}(\vec{\alpha})$ in terms of the map on $W(\vec{\alpha})$, and then use the QPF property of F to show that the maps commute with the abstraction function from $W(\vec{\alpha})$ to $\mathsf{fix}(\vec{\alpha})$. In short, we have that if $F(\vec{\alpha}, \beta)$ is a multivariate QPF, then so is $\mathsf{fix}(\vec{\alpha})$.

The construction of the final coalgebra $\mathsf{cofix}(\vec{\alpha})$ is similar: the approach above can be used to construct the M types $M(\vec{\alpha})$ as polynomial functors, and, once again, the unary construction carries over. Showing that multivariate QPFs are closed under compositions and quotients is once again straightforward.

5 Lifting predicates and relations

Let F be any set-valued functor. By definition, F allows us to map any function $f : \alpha \to \beta$ to a function from $F(\alpha) \to F(\beta)$, enabling us to reason about the behavior of f under F. For instance, $\mathsf{list}(f)$ applies f to every element of a list, and $\mathsf{finset}(f)$ maps any finite set s to $f[s]$, the image of s under f.

Sometimes it is useful to reason about the behavior of predicates and relations as well. A standard way of doing that is to consider their *lifts* [33, 36, 38], defined as follows. Let p be any predicate on α. Then there is an inclusion map $\iota : \{u \in \alpha \mid p(u)\} \to \alpha$, and saying that $p(u)$ holds is equivalent to saying that u is in the image of ι. To lift p to $F(\alpha)$, consider the map $F(\iota) : F(\{u \mid p(u)\}) \to F(\alpha)$, and given any element x of $F(\alpha)$, say that the lift $\hat{p}$ holds of x if and only if there is an element z of $F(\{u \mid p(u)\})$ such that $F(\iota)(z) = x$.

Similarly, if $r(u, v)$ is a binary relation between α and β, we can lift r to a relation $\hat{r}$ between $F(\alpha)$ and $F(\beta)$ as follows. Consider the set $\{(u, v) \in \alpha \times \beta \mid r(u, v)\}$ of pairs, and the two projections π_0 and π_1. Given x in $F(\alpha)$ and y in $F(\beta)$, say that $\hat{r}(x, y)$ holds if there is an element z of $F(\{(u, v) \mid r(u, v)\})$ such that $F(\pi_0)(z) = x$ and $F(\pi_1)(z) = y$.

It is straightforward to define these notions in the type-theoretic setting, with types and subtypes in place of sets and subsets.

```
def liftp {α : Type u} (p : α → Prop) : F α → Prop :=
λ x, ∃ z : F (subtype p), subtype.val <$> z = x

def liftr {α : Type u} (r : α → β → Prop) : F α → F β → Prop :=
λ x y, ∃ z : F {p : α × β // r p.fst p.snd},
  (λ t, t.val.fst) <$> z = x ∧ (λ t, t.val.snd) <$> z = y
```

If P is a polynomial functor, p is a predicate on α, and x is in $P(\alpha)$, it is easy to check that $\hat{p}(x)$ holds if and only if x is of the form (a, f) and for every $i \in B(a)$, $p(f(i))$. Similarly, for a relation r between α and β, x in $P(\alpha)$, and $y \in P(\beta)$, $\hat{r}(x, y)$ if and only if there are a, f, and f' such that x is of the form (a, f), y is of the form (a, f') and for every i, $r(f(i), f'(i))$. In words, $\hat{r}(x, y)$ holds if x and y have the same shape and their contents are pointwise related. If F is a *quotient* of a polynomial functor, the statements are the same up to a choice of representative.

▶ **Theorem 1.** *Let F be a QPF, let p be a predicate on α, and r be a binary relation between α and β.*

- *$\hat{p}(x)$ holds if and only if there are a and f such that $x = \mathsf{abs}(a, f)$ and, for every i, $p(f(i))$.*
- *$\hat{r}(x, y)$ holds if and only if there are a, f, f' such that $x = \mathsf{abs}(a, f)$, $y = \mathsf{abs}(a, f')$, and for every i, $r(f(i), f'(i))$.*

Lifting extends straightforwardly to multivariate QPFs: if $F(\vec{\alpha})$ is an n-ary QPF, we can lift n-ary tuples of predicates and n-ary tuples of relations analogously, and the corresponding version of Theorem 1 holds.

We can use these notions to clarify the additional structure that comes with the Isabelle formulation of a BNF. If F is a QPF and x is an element of $F(\alpha)$, intuitively, $\hat{p}(x)$ says that p holds of the contents of x. When F is a polynomial functor, this is literally true, but the possibility of multiple representations in a QPF muddies the waters. We would like to have a function $\mathsf{supp}(x)$, the "support" of x, such that for every predicate p on α and x in $F(\alpha)$, we have $\hat{p}(x) \leftrightarrow \forall u \in \mathsf{supp}(x)\ p(u)$. Call this condition $(*)$.

▶ **Theorem 2.** *Let F be any set-valued functor. If* supp *satisfies* $(*)$*, then for any* $x \in F(\alpha)$ *we have* $\mathsf{supp}(x) = \{u \mid \forall p\ (\hat{p}(x) \to p(u))\} = \bigcap\{\beta \mid \beta \subseteq \alpha \wedge x \in \mathsf{lm}\ F(\iota_{\beta\to\alpha})\}$*, where* $\iota_{\beta\to\alpha}$ *is the inclusion map from* β *to* α*.*

Proof. We check the first equation, and leave it to the reader to verify the second. Suppose $u \in \mathsf{supp}(x)$ and $\hat{p}(x)$. Then $(*)$ implies that $p(u)$ holds. For the converse, note that taking $p(u)$ to be $u \in \mathsf{supp}(x)$ in $(*)$, it is immediate that $\hat{p}(x)$ holds. So any u satisfying $\forall p\ (\hat{p}(x) \to p(u))$ is an element of $\mathsf{supp}(x)$. ◀

▶ **Theorem 3.** *Let F be a QPF, and let* $\mathsf{supp}(x) = \{u \mid \forall p\ (\hat{p}(x) \to p(u))\}$.

- *For every* x, $\mathsf{supp}(x) = \bigcap\{\mathsf{lm}\ f \mid \mathsf{abs}(a, f) = x\}$.
- *Condition* $(*)$ *holds at* x *for every* p *if and only if there are* a, f *such that* $\mathsf{abs}(a, f) = x$ *and for every* a', f' *such that* $\mathsf{abs}(a', f') = x$, $\mathsf{lm}\ f \subseteq \mathsf{lm}\ f'$.

Proof. For the first clause, let x be arbitrary, and suppose $\hat{p}(x)$ implies $p(u)$ for every p. If $x = \mathsf{abs}(a, f)$, let p be the predicate $u \in \mathsf{lm}\ f$. Then it is easy to check that $\hat{p}(x)$ holds, and hence $p(u)$. Conversely, suppose u is an element of the right-hand side and p is a predicate such that $\hat{p}(x)$ holds. Then there are a and f such that $\mathsf{abs}(a, f) = x$ and such that $p(f(i))$ holds for every i. Hence $p(u)$.

For the forward direction of the second clause, note that if $p(u)$ is the predicate $u \in \mathsf{supp}(x)$, then, by $(*)$, we have $\hat{p}(x)$. The conclusion follows from Theorem 2 and the first clause. Using the first clause, the converse direction of the second clause is also straightforward. ◀

Theorem 3 says that condition $(*)$ holds for a QPF F if and only if every element x of $F(\alpha)$ has a representation (a, f) whose contents are minimal, and these contents determine which lifted predicates hold. Unfortunately, there is nothing in the definition of a QPF that rules out representations having superfluous elements, but the next theorem shows that adding this as an additional assumption has pleasant consequences.

▶ **Theorem 4.** *Let F be a QPF satisfing the additional property that for every a, f, a', and f', if* $\mathsf{abs}(a, f) = \mathsf{abs}(a', f')$*, then* $\mathsf{lm}\ f = \mathsf{lm}\ f'$*. Then:*

- supp *satisfies* $(*)$*, and whenever* $x = \mathsf{abs}(a, f)$, $\mathsf{supp}(x) = \mathsf{lm}\ f$.
- *For every* x *in* $F(\alpha)$ *and* $g : \alpha \to \beta$, $\mathsf{supp}(F(g)(x)) = g[\mathsf{supp}(x)]$.

In other words, with the additional assumption, our function supp has the same properties as the function set associated to Isabelle's BNFs.

BNFs have one additional property, which can also conveniently be expressed in terms of lifts. If r is a relation between α and β and s is a relation between β and γ, the composition $r \circ s$ is defined by $(r \circ s)(u, w) \equiv \exists v\ (r(u, v) \wedge s(v, w))$. It is straightforward to show from the definition of lifting that for every x in $F(\alpha)$ and z in $F(\gamma)$, $\widehat{r \circ s}(x, z)$ implies $(\hat{r} \circ \hat{s})(x, z)$. But the converse does not necessarily hold, and the special case where F is a QPF gives an inkling of what can go wrong: the fact that there is a shape that relates x to y by $\hat{r}$ and another shape that relates y to z by $\hat{s}$ does not necessarily mean there is a single shape that does both, and hence relates x and z by $\widehat{r \circ s}$.

When the converse does hold for every r and s, F is said to *preserve weak pullbacks*. This is a useful property: it implies that the composition of bisimulation relations relative to F is again a bisimulation relation. There are, however, interesting examples of QPFs that do not preserve weak pullbacks, such as a bounded finite powerset, which for some fixed k returns the collection of finite subsets with at most k elements. For details and alternative characterizations of preservation of weak pullbacks, see [6, 33, 36, 38], and for more instances of QPFs that do not preserve them, see [4, Section 6] and [28, Section 6.4].

6 Implementation

We are currently writing a data type compiler for Lean that builds on the formal constructions just described. The compiler, which is implemented entirely in Lean's metaprogramming framework [22], introduces the keywords `data` and `codata` into Lean's normal syntax and translates each data type specification into a number of definitions. Whereas the Isabelle implementation has to construct n-ary instances of the constructions for each fixed n, the uniform theory of multivariate constructions described in Section 4 simplifies the expressions we need to construct, and therefore reduces the likelihood of failure at compile time.

The commands `data` and `codata`, respectively, declare the initial algebra and final coalgebra of a multivariate QPF $F(\vec{\alpha}, \beta)$. In our implementation, we refer to F as the *shape* of the declaration. The key insight is that both the functor and its representation as a QPF can be synthesized from the syntactic specification. Consider the following input:

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

This describes the type of trees in which every internal node has a label from α and a sequence of children indexed by β. Since β occurs in a negative position, our compiler interprets that as a dead parameter. It then replaces `tree` with a parameter X and interprets the resulting shape as a binary functor $F_\beta(\alpha, X)$.

```
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
```

Note that the internal version bundles α and X together into a vector of length 2. The next task is to synthesize a QPF instance. In general, the arguments to each constructor are compositions of QPFs, so the entire shape, a sum of products of QPFs, is again a QPF.

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

We then use the generic QPF fix construction to define the initial fixed point.

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

We can then define the constructors, destructors, recursor, and so on:

```
def tree.nil (α β : Type) : tree α β := ...

def tree.cons (α β : Type) (x : α) (xs : β → tree α β) : tree α β := ...

def tree.cases_on {α β : Type} {C : tree α β → Sort u_1} (n : tree α β) :
  C (tree.nil α β) →
```

```
    (Π (a : α) (a_1 : β → tree α β), C (tree.cons α β a a_1)) →
    C n := ...

def tree.rec {α β X : Type} : X → (α → (β → X) → X) → tree α β → X := ...
```

If we replace `data` by `codata`, we get the corresponding coinductive type. It has same constructors and destructors, but, instead, the following corecursor and bisimulation principle:

```
def tree'.corec :
  Π (α β α_1 : Type), (α_1 → shape α β α_1) → α_1 → tree' α β := ...

def tree'.bisim : ∀ (α β : Type) (r : tree' α β → tree' α β → Prop),
  (∀ (x y : tree' α β),
      r x y → mvfunctor.liftr (typevec.rel_last ⌈α⌉ r) (mvqpf.cofix.dest x)
    (mvqpf.cofix.dest y)) →
  ∀ (x y : tree' α β), r x y → x = y := ...
```

Our work on the compiler is still in progress: we do not yet handle nested data types in the specification of the shape or present lifted predicates and relations in a user-friendly way. We also intend to write an equation compiler to support more natural ways to define functions.

7 Conclusions and related work

We have shown that the representation of data types as quotients of polynomial functors is natural, and facilitates important data type constructions. Surprisingly, the bulk of our formalization deals with constructions that are intuitively straightforward, like the representations of multivariate W and M types as polynomial functors as described in Section 4. It is notable that, with this infrastructure, our constructions of the initial algebras and final coalgebras require only a few hundred lines of code.

Other theorem provers such as Coq [26] and Agda[1] support coinductive types and corecursion by extending the trusted kernel. Here we have followed Isabelle's approach by constructing such data types explictly, without extending the axiomatic framework. We made use of a quotient construction that is given axiomatically in Lean, though other libraries, including Isabelle's, take a definitional approach to quotients as well [17, 27, 29]. Tassi has recently developed methods for generating induction principles and other theorems to support the use of inductive types in Coq [40]. In a sense, this serves to recover some of the benefits of the more modular approaches given by BNFs and QPFs.

Abbot et al. [2] have considered quotients of polynomial functors by equivalence with respect to sets of permutations of the indices associated to each shape, and they have shown that these have nice computational properties. Such quotients are special cases of QPFs. Polynomial functors are closely related to *species* [30], but, as noted by Yorgey [41, Section 8], the precise relationship between polynomial functors and species is not yet well understood.

There are a number of ways that our work can be extended. Our constructions currently yield nondependent types and nondependent recursion and corecursion principles, so an obvious task is to work out and formalize the semantics of indexed inductive and coinductive data types. To that end, work by Altenkirch et al. [7] on indexed polynomial functors provides a good starting point. We are grateful to an anonymous referee for pointing out that there is nothing special about $\mathsf{fin}(n)$ in the definition of mvfunctor in Section 4, and so replacing

[1] `https://agda.readthedocs.io/en/latest/language/coinduction.html`

$\mathsf{fin}(n)$ by an arbitrary index type I is an easy first step towards handling families of types. The latter would also require generalizing our constructions to handle functors on categories other than the category of types (in this case, categories of indexed families). The paper by Blanchette et al. [16] shows how to achieve nonuniform forms of recursion and corecursion with BNFs, which can be seen as a step towards handling such dependencies. Dependent families would provide us with a shortcut to defining mutual inductive and coinductive definitions, currently handled by Isabelle's BNF package but not ours. Blanchette et al. [14] have shown that restricting morphisms to permutations can be used to model data types with binders.

We have not dealt with the computational interpretation of corecursion or code extraction at all. Even though most of our formalization is constructive, the defining equations for corecursion do not correspond to computational reductions in our underlying definitions. Firsov and Stump [23] show how to model inductive types in a computational type theory extending the calculus of constructions with implicit products, heterogeneous equality, and intersection types. It would be interesting to know whether coinductive types can be modeled in a similar way. Blanchette et al. [13] provide a nice overview of various approaches to computational intepretation of corecursion, and Basold and Geuvers [10, 11] provide a computational analysis of dependent versions of a type theory with both recursion and corecursion. We are hopeful that quotients of polynomial functors can provide insight into the semantics of such a system.

References

1 Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of Containers. In Andrew D. Gordon, editor, *Foundations of Software Science and Computational Structures (FOSSACS) 2003*, pages 23–38. Springer, 2003. doi:10.1007/3-540-36576-1_2.

2 Michael Gordon Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Constructing Polymorphic Programs with Quotient Types. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction (MPC) 2004*, pages 2–15. Springer, 2004. doi:10.1007/978-3-540-27764-4_2.

3 Peter Aczel. *Non-well-founded sets*. Stanford University, Center for the Study of Language and Information, Stanford, CA, 1988.

4 Peter Aczel and Nax Mendler. A final coalgebra theorem. In *Category theory and computer science*, pages 357–365. Springer, Berlin, 1989. doi:10.1007/BFb0018361.

5 Jiri Adámek and H.-E. Porst. On tree coalgebras and coalgebra presentations. *Theoret. Comput. Sci.*, 311(1-3):257–283, 2004. doi:10.1016/S0304-3975(03)00378-5.

6 Jiří Adámek, Stefan Milius, and Lawrence S. Moss. Fixed points of functors. *J. Log. Algebr. Methods Program.*, 95:41–81, 2018. doi:10.1016/j.jlamp.2017.11.003.

7 Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. *J. Funct. Programming*, 25:e5, 41, 2015. doi:10.1017/S095679681500009X.

8 Jeremy Avigad, Soonho Kong, and Leonardo de Moura. Theorem Proving in Lean. Online documentation. URL: https://leanprover.github.io/theorem_proving_in_lean/.

9 Michael Barr. Terminal coalgebras in well-founded set theory. *Theor. Comput. Sci.*, 114(2):299–315, 1993. doi:10.1016/0304-3975(93)90076-6.

10 Henning Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic*. PhD thesis, Radboud Universiteit Nijmegen, 2018.

11 Henning Basold and Herman Geuvers. Type Theory based on Dependent Inductive and Coinductive Types. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Logic in Computer Science (LICS) 2016*, pages 327–336. ACM, 2016. doi:10.1145/2933575.2934514.

12 Julian Biendarra, Jasmin Christian Blanchette, Aymeric Bouzy, Martin Desharnais, Mathias Fleury, Johannes Hölzl, Ondrej Kuncar, Andreas Lochbihler, Fabian Meier, Lorenz Panny, Andrei Popescu, Christian Sternagel, René Thiemann, and Dmitriy Traytel. Foundational (Co)datatypes and (Co)recursion for Higher-Order Logic. In Clare Dixon and Marcelo Finger, editors, *Frontiers of Combining Systems (FroCoS) 2017*, pages 3–21. Springer, 2017. doi:10.1007/978-3-319-66167-4_1.

13 Jasmin Christian Blanchette, Aymeric Bouzy, Andreas Lochbihler, Andrei Popescu, and Dmitriy Traytel. Friends with Benefits: Implementing Corecursion in Foundational Proof Assistants. In Hongseok Yang, editor, *Programming Languages and Systems (ESOP) 2017*, pages 111–140. Springer, 2017. doi:10.1007/978-3-662-54434-1_5.

14 Jasmin Christian Blanchette, Lorenzo Gheri, Andrei Popescu, and Dmitriy Traytel. Bindings as bounded natural functors. *PACMPL*, 3(POPL):22:1–22:34, 2019. doi:10.1145/3290335.

15 Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly Modular (Co)datatypes for Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving (ITP) 2014*, pages 93–110. Springer, 2014. doi:10.1007/978-3-319-08970-6_7.

16 Jasmin Christian Blanchette, Fabian Meier, Andrei Popescu, and Dmitriy Traytel. Foundational nonuniform (Co)datatypes for higher-order logic. In *Logic in Computer Science (LICS) 2017*, pages 1–12. IEEE Computer Society, 2017. doi:10.1109/LICS.2017.8005071.

17 Cyril Cohen. Pragmatic Quotient Types in Coq. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving (ITP) 2013*, pages 213–228. Springer, 2013. doi:10.1007/978-3-642-39634-2_17.

18 Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986. URL: http://dl.acm.org/citation.cfm?id=10510.

19 Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In Amy P. Felty and Aart Middeldorp, editors, *Conference on Automated Deduction (CADE) 2015*, pages 378–388. Springer, 2015. doi:10.1007/978-3-319-21401-6_26.

20 Richard Dedekind. *Was sind und was sollen die Zahlen?* Vieweg, Braunschweig, 1888. Translated by Wooster Beman as "The nature and meaning of numbers" in *Essays on the theory of numbers*, Open Court, Chicago, 1901; reprinted by Dover, New York, 1963.

21 Peter Dybjer. Inductive Families. *Formal Asp. Comput.*, 6(4):440–465, 1994. doi:10.1007/BF01211308.

22 Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *PACMPL*, 1(ICFP):34:1–34:29, 2017. doi:10.1145/3110278.

23 Denis Firsov and Aaron Stump. Generic derivation of induction for impredicative encodings in Cedille. In June Andronick and Amy P. Felty, editors, *Certified Programs and Proofs (CPP) 2018*, pages 215–227. ACM, 2018. doi:10.1145/3167087.

24 Gottlob Frege. *Grundgesetze der Arithmetik*. H. Pohle, Jena, Volume 1, 1893, Volume 2, 1903.

25 Nicola Gambino and Martin Hyland. Wellfounded Trees and Dependent Polynomial Functors. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs (TYPES) 2003*, pages 210–225. Springer, 2003. doi:10.1007/978-3-540-24849-1_14.

26 Eduardo Giménez. Codifying Guarded Definitions with Recursive Schemes. In Peter Dybjer, Bengt Nordström, and Jan M. Smith, editors, *Types for Proofs and Programs (TYPES) 1994*, pages 39–59. Springer, 1994. doi:10.1007/3-540-60579-7_3.

27 Martin Hofmann. A Simple Model for Quotient Types. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *Typed Lambda Calculi and Applications (TLCA) 1995*, pages 216–234. Springer, 1995. doi:10.1007/BFb0014055.

28 Johannes Hölzl, Andreas Lochbihler, and Dmitriy Traytel. A Formalized Hierarchy of Probabilistic System Types (Proof Pearl). In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving (ITP) 2015*, pages 203–220. Springer, 2015. doi:10.1007/978-3-319-22102-1_13.

29 Peter V. Homeier. A Design Structure for Higher Order Quotients. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLs) 2005*, pages 130–146. Springer, 2005. doi:10.1007/11541868_9.

30 André Joyal. Une théorie combinatoire des séries formelles. *Adv. in Math.*, 42(1):1–82, 1981. doi:10.1016/0001-8708(81)90052-9.

31 Bronisław Knaster. Un théorème sur les fonctions d'ensembles. *Ann. Soc. Polon. Math.*, 6:133–134, 1928.

32 Georg Kreisel. Generalized Inductive Definitions. Stanford Report on the Foundations of Analysis (mimeographed), CH. III, Stanford, 1963.

33 Alexander Kurz and Jiri Velebil. Relation lifting, a survey. *J. Log. Algebr. Meth. Program.*, 85(4):475–499, 2016. doi:10.1016/j.jlamp.2015.08.002.

34 Per Martin-Löf. An intuitionistic theory of types: predicative part. In *Logic Colloquium 1973*, pages 73–118. North-Holland, Amsterdam, 1975.

35 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL. A proof assistant for higher-order logic*. Springer Verlag, Berlin, 2002.

36 Jan J. M. M. Rutten. Relators and Metric Bisimulations. *Electr. Notes Theor. Comput. Sci.*, 11:252–258, 1998. doi:10.1016/S1571-0661(04)00063-5.

37 Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoret. Comput. Sci.*, 249(1):3–80, 2000. doi:10.1016/S0304-3975(00)00056-6.

38 Sam Staton. Relating Coalgebraic Notions of Bisimulation. In Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki, editors, *Algebra and Coalgebra in Computer Science (CALCO) 2009*, pages 191–205. Springer, 2009. doi:10.1007/978-3-642-03741-2_14.

39 Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955. URL: http://projecteuclid.org/euclid.pjm/1103044538.

40 Enrico Tassi. Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq. This *Proceedings*, 2019.

41 Brent A. Yorgey. Species and functors and types, oh my! In Jeremy Gibbons, editor, *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell*, pages 147–158. ACM, 2010. doi:10.1145/1863523.1863542.