Implementing a definitional (co)datatype package in Lean 4, based on quotients of polynomial functors

MSc Thesis *(Afstudeerscriptie)*

written by

Alex C. Keizer
(born October 15th, 1998 in Alkmaar, The Netherlands)

under the supervision of **Dr. Jasmin Blanchette** and **Dr. Benno van den Berg**, and submitted to the Examinations Board in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam*.

Date of the public defense:	**Members of the Thesis Committee:**
February 22, 2023	Dr. Malvin Gattinger (chair)
	Dr. Tobias Kappé
	Dr. Jasmin Blanchette (supervisor)
	Dr. Benno van den Berg (supervisor)

Institute for Logic, Language and Computation

Abstract

Coinduction is the dual of induction. While inductive types are ubiquitous in functional programming languages, coinductive types, also known as codatatypes, are considerably less well-supported. Notably, Lean 4, the latest edition of an interactive theorem prover and dependently typed functional programming language developed at Microsoft Research, lacks support for coinduction.

I have implemented a (co)datatype package for Lean 4, which compiles high-level, definitional specifications for types, which may mix induction, coinduction and quotients, into a definition of that type in terms of quotients of polynomial functors. These QPFs were previously formalized in Lean 3, which I took as a starting point and ported to Lean 4. This thesis describes how the compilation procedure works, and dives into technical details of the Lean meta-programming system that influenced the implementation. The package adds **`data`** and **`codata`** commands for specifications of inductive, respectively coinductive, types. Additionally, a **`qpf`** command exposes a specific part of the procedure, called the *composition pipeline*, that is used to define new QPFs composed of other QPFs.

Acknowledgements

Naturally, I would like to thank my supervisors, Jasmin Blanchette and Benno van den Berg, for introducing me to the world of interactive theorem provers, their guidance, advice and patience. I would also like to thank Jeremy Avigad and Simon Huddon, for agreeing to meet with me and explain their work, which formed the inspiration of this thesis. Of course, I should also thank Malvin Gattinger and Tobias Kappé for agreeing to be on the committee. Finally, I want to thank my friends and family, too numerous to name, for their support and enduring my ramblings about the awesomeness of theorem provers and infinite data structures. Special thanks go out to Bram Honig, for some last minute proof-reading. Finally, I would like to thank my partner, Alice, in particular, for being an endless source of love and support when I needed it most.

Contents

Chapter 1

Introduction

Modern logic is primarily built on a framework of induction. It is no surprise, then, that Lean—an interactive theorem prover / dependently typed, functional programming language—prominently features induction.

In Lean, new datatypes are generally defined using the **`inductive`** keyword, which exposes a high-level, definitional syntax.

```
inductive List α
  | nil  : List α
  | cons : α → List α → List α
```

Read as follows: `List` has two *constructors*, i.e., ways to construct a list. `List.nil` is a constant representing the empty list; `List.cons head tail`, where `head` is of type `α` and `tail` is another list, representing the operation of adding a new element to the front of an existing list. Notice also that `α` is a type parameter, meaning that the list is generic over the type of its elements; `List Nat` is a list of natural numbers, while `List String` is a list of strings.

From this description, the datatype is freely generated, as codified by the characteristic principles that are automatically added by the **`inductive`** command. The *principle of no confusion* states that different combinations of constructors yield distinct elements of the inductive type, and the *recursion principle* embodies structural recursion on that type. For instance, the following is a simplified signature of a recursion principle for lists:

```
rec : β → (α → β → β) → (List α → β)
```

Where the function `rec b f` maps `List.nil` to the constant `b`, and `List.cons head tail` to `f head (rec f tail)`. Since Lean is dependently typed, the actual recursion principle is slightly more involved, but the core idea is the same. This principle also asserts that `List.nil` and `List.cons` are the only ways to obtain elements of type `List`. This extra assertion is also sometimes presented separately as the *principle of no junk*.

The inductive interpretation means that every element of `List` consists of only finitely many applications of its constructors. In particular, it is not possible to construct a list with an infinite chain of cons applications

```
cons 0 (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 ...)))))
```

The recursion must end in `nil` at some point; `List` represents only finite lists. That being said, (countably) infinite lists (also called streams) have a straightforward encoding in Lean, as functions from natural numbers to the parameter type.

```
def Stream α := Nat → α
```

The type of potentially infinite lists is simply the sum of `List` and `Stream`.

```
inductive CoList α
  | fin : List α      -- a finite list, or
  | inf : Stream α    -- an infinite list
```

However, Lean lacks a comprehensive, definitional solution to defining infinite data structures.

To address this, Avigad *et al.* describe a framework for defining *coinductive* types (also called codatatypes) in Lean, basing their constructions on the category theoretical notion of *quotients of polynomial functors* (QPF). They also provide a formalization of their work in Lean 3 [1]. However, to define a (co)datatype using this framework directly, a user has to hand-compile their specification into the provided fundamental operations, making it a not very user-friendly way to define a type.

The main contribution of this thesis is the description, and prototype implementation, of **`data`** and **`codata`** commands, enabling users to write a (co)datatype specification with the familiar, and more convenient, syntax of **`inductive`**, which is then automatically compiled into the appropriate constructions on QPFs. The implementation of these commands was done in Lean 4; I ported the existing formalization by Avigad *et al.* to this newer version of Lean.

For example, the following two **`codata`** specifications are compiled into coinductive types for infinite lists and potentially infinite lists, equivalent to the ad hoc definitions we saw above.

```
codata Stream' α
  | cons : α → Stream' α → Stream' α

codata CoList' α
  | nil  : CoList' α
  | cons : α → CoList' α → CoList' α
```

Similarly, we can redefine inductive (i.e., finite) lists in terms of **`data`**.

```
data List' α
  | nil  : List' α
  | cons : α → List' α → List' α
```

The QPF approach is *compositional*: The **data** and **codata** commands recognize when the newly defined (co)datatype is itself a QPF, and automatically register it as such, enabling its use in subsequent (co)datatype definitions.

This is not limited to just induction or coinduction; (co)datatypes may be defined with a nested mix of both. For example, the following defines a potentially infinitely branching (because children of a node are stored in a `CoList`) tree of finite depth (because of the inductive interpretation of **data**).

```
data RoseTree α
  | node : α → CoList' (RoseTree α) → RoseTree α
```

If, instead, we define a coinductive type, we would get trees of infinite depth.

```
codata CoRoseTree α β
  | node : α → CoList' (CoRoseTree α) → CoRoseTree α
```

We can also define finitely branching trees of infinite depth, by replacing `CoList'` with `List'`.

```
codata CoRoseTree₂ α β
  | node : α → List' (CoRoseTree₂ α) → CoRoseTree₂ α
```

Quotients

Besides **inductive** types, Lean also supports defining new types as quotients of other types. It is common, e.g., to define a multiset as the equivalence class of lists with respect to the relation that equates lists up to permutation.

```
/-- `List.perm as bs` holds iff `as` is a permutation of `bs` -/
def List.perm : List α → List α → Prop

def Multiset α := Quot.mk (@List.perm α)
```

We'd like to be able to use `Multiset` to define the type of unordered trees.

```
inductive UnorderedTree α
  | node : α →  Multiset (UnorderedTree α) → UnorderedTree α
```

Yet, this definition won't compile; it is not allowed to nest a recursive occurrence of `UnorderedTree` behind a quotient type. Lean's built-in inductive and quotient types do not compose well.

As the name suggests, QPFs support a notion of quotient types. Our datatype package does not yet provide user-friendly syntax for defining QPF-based quotient types, but suppose that `Multiset'` were manually defined in terms of QPFs, then we can define unordered trees as follows:

```
noncomputable data UnorderedTree' α
  | node : α →  Multiset' (UnorderedTree' α) → UnorderedTree' α
```

That is, the compositionality of QPFs extends to arbitrary mixes of inductive, coinductive and quotient types.

Limitations

The implementation is not complete yet, and in particular, doesn't yet generate nicely encapsulated no confusion and (co)recursion principles. The QPF framework certainly allows for these principles to be derived, and it is already possible to define (co)recursive functions on (co)datatypes. However, only by invoking the fundamental operation, exposing the supposedly internal QPF encoding.

A major opportunity for future work lies in improving this situation, bringing the experience of writing (co)recursive for **`data`** and **`codata`** types closer to the equational approach used to define recursive functions for standard **`inductive`** types.

Related Work

The QPF framework by Avigad *et al.*, and the algebraic study of datatypes in general, is based on the observation that datatypes are generally *functorial* [1].

Specifically, the QPF constructions are a variant of the notion of *bounded natural functors* (BNF), which dates back to an effort to add a definitional package for (co)datatypes to Isabelle—a different theorem prover [2, 3]. Inspiration has also gone the other direction, with Fürer *et al.* studying quotients of BNFs [4].

Lean is developed by de Moura *et al.* at Microsoft Research, based on the Calculus of Inductive Constructions [5, 6]. Agda and Coq, two more theorem provers, provide support for coinductive types and corecursion directly in their trusted kernel [7, 8]. Contrast this with the BNF or QPF approaches, which are implemented entirely as a library.

Basold and Geuvers [9, 10] studied a dependent type theory with coinductive types separate from any specific theorem prover. Their approach ensures a computational meaning of the terms.

Organization

Chapter 2 will explain what QPFs are, and illustrate Lean's syntax in the process. Chapter 3 will dig into the differences between Lean 3 and Lean 4, and detail the process of porting the QPF formalizations made by Avigad *et al.* Chapter 4 will describe enhancements made to the formalizations in the process, which go beyond just porting the existing behaviour. Chapter 5 will establish the procedure to translate the definitional syntax of **`data`** and **`codata`** into the proper constructions in the theory of QPFs. Chapter 6 will go into technical detail about the (proof of concept) implementation of these commands, and the Lean 4 meta-programming system.

Chapter 7 will discuss the limitations of the current implementation, and opportunities for improvement. Finally, chapter 8 concludes the thesis.

Accompanying code can be found at `https://github.com/alexkeizer/qpf4`. It provides the **`data`** and **`codata`** commands discussed so far, alongside a **`qpf`** command for the composition pipeline. A preconfigured environment is also made available at `https://gitpod.io#github.com/alexkeizer/qpf4-example/blob/main/Scratch/Examples.lean`, allowing readers to try out **`data`** and **`codata`** in their browser, with minimal configuration.

The code snippets in this thesis are tested with version `leanprover/lean4:nightly-2022-04-28`. The same version was used to develop the code in the linked repository and runs in the preconfigured environment. The thesis also contains code that (intentionally) does not typecheck. These snippets are typeset with a red line, like so:

```
def Foo := Bar -- whoops, Bar does not exist
```

Chapter 2

Background

A key concept for this thesis is the QPF (Quotient of Polynomial Functor), and how QPFs can be used to encode (co)inductive types. The current chapter serves to illuminate this notion, and explain relevant parts of the Lean system along the way.

We will assume as little background knowledge as possible, yet, some (minimal) exposure to category theory and functional programming concepts will be beneficial to understanding. Readers can find references at [11, 12] for category theory and [13] for functional programming.

> **Remark:** We will use remarks like this one below code snippets to explain Lean syntax and concepts that might not be familiar. We don't assume any knowledge of Lean, but the interested reader is invited to consult the online documentation, or *Functional Programming in Lean* for a more comprehensive introduction [14, 13].

In the interest of consistency, we will use Lean 4 syntax and naming conventions throughout the whole thesis, even in our current discussion of the general theory and formalizations as presented by Avigad *et al.* (in Lean 3). The differences between the Lean definitions presented here and their original definition in [1] are purely superficial.

The encoding of types as QPFs relies on a key observation; a lot of types are functorial in nature. Take, for example, the inductive type of lists, specified as

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

This defines a function `List` that takes a type, `α`, and returns a new type, whose elements represent lists of `α`. We call such functions *type functions*.

2.1 Type universes

Lean is a dependently typed language, so types *are* terms. In particular, things like `Nat` and `String` are types, but they are also *values* of type `Type`. The type of `List`, then, is `Type → Type`. In particular, there is no distinction between type functions and functions that operate on, e.g., natural numbers.

> **Remark:** We write `f : α → β` to say "`f` has type `α → β`". The arrow type `α → β` stands for the type of functions taking an argument of type `α` to produce a value of type `β`.

We call `Type` a *type universe*, i.e., a type whose elements are themselves types. However, `Type` is itself also a value, which needs to live in some universe. It cannot be the case that `Type : Type`; this leads to Girard's paradox, which is the type theory analogue of Russel's paradox [15].

Instead, Lean has an infinite sequence of increasing type universes, so that `Type : Type 1`, `Type 1 : Type 2`, and in general, `Type u : Type (u+1)`. In fact, `Type` is just a shorthand for `Type 0`, the smallest type universe.

Universe levels `u` are essentially natural numbers, but they are **not** first-class values, and are indeed very different from elements of `Nat`. In particular, when writing `Type u`, the universe level `u` is **not** a regular term. There is a separate grammar, with some minimal builtin operations, that defines valid universe levels.

It is possible, and indeed common to be generic over the universe of some type parameter, we call such definitions *universe polymorphic*.

For example, the actual definition of lists specifies that the parameter `α` should live in an arbitrary universe `Type u`.

```
inductive List (α : Type u)
  | nil  : List α
  | cons : α → List α → List α
```

The type of this version of `List` is `Type u → Type u`, for arbitrary universe levels `u`.

2.2 Functoriality of type functions

Returning to our discussion of `List` and its functoriality, there is an obvious mapping function to lift a function `f : α → β`, for arbitrary types `α` and `β`, into a function `List α → List β`, by applying `f` to each element of the argument list. Its signature is written as

```
map : (f : α → β) → List α → List β
```

Remark: If a function takes multiple arguments, it is idiomatic to write them in a *curried* style, so `f : α → β → γ` says that `f` is a function that takes two arguments, an `α` and a `β`, to produce a `γ`. Arrows are right-associative.

Furthermore, `(a : α) → β` is a *dependent* arrow; it is a function from `α` to `β`, with the possibility to make the resulting type depend on the *value* `a` of the first argument. We used this syntax in the definition of `map` above, even though it is a regular, non-dependent function, just to give a name to the first argument. It would have been equivalent to write `map : (α → β) → List α → List β`.

Notice that `α` and `β` are arbitrary and thus must also be arguments to `map`. Nevertheless, the value of `α` and `β` can be inferred from the other arguments and there is no need to supply values for them when calling `map`, hence, they can be *implicit arguments*.

It is possible to explicitly define implicit arguments, using curly brackets `{...}`.

```
map: {α β : Type u} → (α → β) → List α → List β
```

This allows us to call `map` as `map f as` and the values of `α` and `β` are inferred from the types of `f` and `as`.

Lean will automatically add implicit binders for free variables in a type signature, a feature called *auto-bound implicits*. We will generally rely on this feature, adopting the convention that `α` and `β` refer to types.

A type function `F : Type u → Type v` together with a mapping operation `map : (f: α → β) → F α → F β` form a *functor*, so long as they preserve:

- *Identity maps*, that is, `F (id α) = id (F α)`, with `id β` the identity function on arbitrary types `β`, and
- *Compositions*, that is, `(map f) ∘ (map g) = map (f ∘ g)`, where ∘ denotes function composition

A function $f : F(\alpha) \to \alpha$, for some fixed *carrier* type `α`, is called an *`F`-algebra*. Inductive types correspond to the carrier of an *initial* such algebra. To clarify, an `F`-algebra `f` is initial, if for every `F`-algebra $g : F(\beta) \to \beta$ there is exactly one arrow `rec` $: \alpha \to \beta$ that makes the left square of fig. 2.1 commute.

Consider the type of natural numbers, `Nat`; we'll show that the constructors for `Nat` constitute an initial algebra for functor the $F_{nat}(\alpha) =$ `Unit` $\oplus \alpha$, where `Unit` is the unit type with `(unit : Unit)` as its sole inhabitant and $\oplus$ denotes a sum. On functions $f : \alpha \to \beta$, the obvious choice of mapping operation $F(f) : F(\alpha) \to F(\beta)$ is such that $F(f)$(`unit`) = `unit` and $F(f)(a) = f(a)$ for all $a : \alpha$.

Figure 2.1: Commuting square for initial F-algebras f and final F-coalgebras c

Natural numbers are defined by a constant $0 : \mathtt{Nat}$ and a function $\mathrm{succ} : \mathtt{Nat} \to \mathtt{Nat}$. Let $f : F_{nat}(\mathtt{Nat}) \to \mathtt{Nat}$ be the F-algebra defined as $f(\mathtt{unit}) = 0$ and $f(i) = \mathrm{succ}(i)$ for all $i : \mathtt{Nat}$.

Then, let $g : F_{nat}(\alpha) \to \alpha$ be an arbitrary F-algebra, and define a function $\mathrm{rec} : \mathbb{N} \to \alpha$ such that $\mathrm{rec}(0) = g(\mathtt{unit})$ and $\mathrm{rec}(n+1) = \mathrm{rec}(g(n))$, for every $n \in \mathbb{N}$. The inductive properties of `Nat` are exactly what makes this algebra initial.

Conversely, initial algebras are unique up to isomorphism, so any other initial F_{nat}-algebra $h : F_{nat}(\beta) \to \beta$ is equivalent to f and its carrier, β, is equivalent to `Nat`.

Dually, there is a connection between coinductive types and *final coalgebras*. More concretely, a F-coalgebra is a function $c : \alpha \to F(\alpha)$, with α again an arbitrary carrier type, The coalgebra c is *final* if for every other F-coalgebra $d : \beta \to F(\beta)$ there is a unique arrow `corec: β → α` that makes the right square of fig. 2.1 commute. Finally, the carrier of a final coalgebra corresponds to a coinductive type.

Alternatively, we know that α is the carrier of an initial F-algebra (resp. final F-coalgebra) and thus corresponds to an inductive (resp. coinductive) type iff it is the least (resp. greatest) fixpoint of F. We will also call these the *fixpoint*, resp. *cofixpoint*, of F.

Not all functors, though, have initial algebras, or final coalgebras.

2.3 Polynomial functors

Of special interest are *polynomial functors*, which, intuitively, are created from just a few primitive operations (constants, sums, products, and exponentials). More formally, we say that a polynomial functor is defined by a set A and an A-indexed family of sets B_a as

$$P(X) = \Sigma_{a \in A} B_a \to X$$

That is, $P(X)$ is the disjoint union of all functions from B_a to X, for every $a \in A$. In the theory, we generally still refer to functors that are not defined in this form, but are isomorphic to a polynomial functor in the strict sense, as polynomial.

To encode this in Lean, we replace "set" with "type" and obtain the following.

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

Remark: `structure` is a simple wrapper around `inductive`, for when there is only one constructor. The above type is equivalent to

```
inductive PFunctor
| mk : (A : Type u) → (B : A → Type u) → PFunctor
```

Then, the operations on types and functions are straightforward:

```
/-- Applying `P` to an object of `Type` -/
def PFunctor.Obj (P : PFunctor) (α : Type u)
    := Σ x : P.A, P.B x → α

/-- Applying `P` to a morphism of `Type` -/
def PFunctor.map (P : PFunctor) (f : α → β) : P.Obj α → P.Obj β
    := fun ⟨a, g⟩ => ⟨a, f ∘ g⟩
```

Remark: Variable `P` is known to be of type `PFunctor`, so `P.Obj` is recognized as `PFunctor.Obj P`. Similarly, as `PFunctor` has only one constructor, `PFunctor.mk`, the anonymous constructor syntax `⟨a, g⟩` is translated to `PFunctor.mk a g`. Finally, `fun ⟨a, g⟩ => _` defines a function that takes a single argument `x : P.Obj α`, and immediately deconstructs it into the constituent elements `a : P.A` and `g : P.B a → α`.

So an element of `P.Obj α` is a (dependent) pair of a *shape* $a \in A$ and a function $g : B_a \to \alpha$, representing the *contents*. A mapped function `P.map f` then leaves the shape as is, and pre-composes f with the content g.

W-types

We already saw that inductive types are freely generated by the constructors. In a sense, an element of, e.g., `Nat` is a well-founded (i.e., finite) tree with two kinds of nodes: `zero` nodes are leaves, and `succ` nodes have exactly one child.

The *W-type* of a polynomial functor `P` is the type of exactly such trees: shapes $a \in A$ distinguish different kinds of nodes, and the cardinality of B_a determines the number of children nodes of type a have. Such trees are easily encoded by an `inductive` type.

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

By construction, the W-type of P is its fixpoint, with `W.mk` its initial algebra.

It is important to make a distinction between types that *are* polynomial functors, and types that are (equivalent to) W-types *of* polynomial functors. For example, `Nat` is not a polynomial functor, but it *is* the W-type of a polynomial functor (namely, $\mathsf{F}_{\mathsf{nat}}$).

Conversely, `List` is both. It is a polynomial functor: take $A = \mathbb{N}$ and $B_n = \{0, .., n-1\}$, then a list of n elements is encoded as a pair with shape n, and content $f : \{0, .., n-i\}$ mapping each $i < n$ to the i-th element of the list. Simultaneously, `List α`, for every `α` is also the W-type of a *different* polynomial functor, defined by $A = \mathtt{Unit} \oplus \alpha$, where B of the unit value is the empty type and $B_a = \mathtt{Unit}$ for every $a : \alpha$. The W-type of this polynomial functor consists of trees where leaves represent the empty list, and internal nodes are labelled with an element of type `α` (the head of the list) and have exactly one subtree (the tail of the list).

M-types

Coinductive types are similar, except that the trees they are represented by may be of infinite depth. Encoding these in Lean is quite a bit more involved.

An *M-type* of a polynomial functor `P` is the type of potentially infinite depth trees, where shapes $a \in A$ distinguish different kinds of nodes, and the cardinality of B_a determines the number of children nodes of type a have.

An *approximation* of an M-type up to depth n is the type of such trees of height at most n, where any required subtrees at depth $n+1$ are replaced with a special "continue" leaf.

```
/-- `CofixA P n` is an `n` level approximation of an M-type -/
inductive CofixA (P : PFunctor) : Nat → Type u
| continu : CofixA 0
| intro {n} : (a : P.A) → (P.B a → CofixA P n) → CofixA P (succ n)
```

Then, we define what it means for approximations *agree* — either the first approximation must be a "continue" leaf or both approximations have nodes of the same kind as root, and the corresponding subtrees recursively agree — whence an M-type is just an infinite series of approximations of increasing depth where every approximation agrees with the next.

```
inductive Agree : ∀ {n : Nat}, CofixA P n → CofixA P (n + 1) → Prop
| continu (x : CofixA P 0) (y : CofixA P 1) : Agree x y
| intro {n} {a}   (x  : F.B a → CofixA P n)
                  (x' : F.B a → CofixA P (n + 1))
                  : (∀ i : F.B a, Agree (x i) (x' i))
                  → Agree (CofixA.intro a x) (CofixA.intro a x')

structure M (P : PFunctor) :=
  (approx : ∀ n, CofixA P n)
  (consistent : ∀ n, Agree (approx n) (approx (n+1)))
```

The M-type of a polynomial functor is, by construction, its cofixpoint.

2.4 Quotients

As mentioned in the introduction, Lean also supports quotients of types. Recall the example of multisets, defined as a quotient on lists.

```
/-- `List.perm as bs` holds iff `as` is a permutation of `bs` -/
def List.perm : List α → List α → Prop

def Multiset α := Quot (@List.perm α)
```

Polynomial functors cannot represent such quotients. Thus, we generalize to *quotients* of polynomial functors, QPFs.

Intuitively, a functor F is the quotient of some polynomial functor P when there is a surjective *natural transformation* `abs` from P to F. We can think of $\texttt{abs}_\alpha$ as mapping abstract objects in $F(\alpha)$ to their concrete representations in $P(\alpha)$, for every α. Recall that every functor is equipped with a mapping operation on functions $f : \alpha \to \beta$. To say that `abs` is a natural transformation is to say that it respects this functorial structure. That is to say, the square of fig. 2.2 should commute for all arrows $f : \alpha \to \beta$.

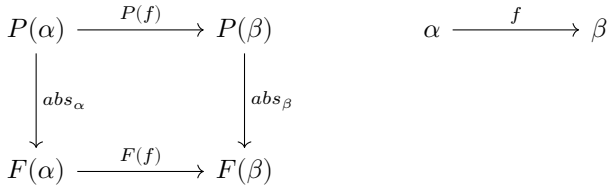

Figure 2.2: Commuting square for natural transformation $abs : P \Longrightarrow F$

Formally, we can show that `abs` is surjective by providing, for every α, a function $\texttt{repr} : F(\alpha) \to P(\alpha)$ and proving that it is a right-inverse. This is specified in Lean with the `QPF` typeclass.

```
class Qpf (F : Type u → Type u) [Functor F] where
  P        : PFunctor
  abs      : ∀ {α}, P.Obj α → F α
  repr     : ∀ {α}, F α → P.Obj α
  abs_repr : ∀ {α} (x : F α), abs (repr x) = x
  abs_map  : ∀ {α β} (f : α → β) (p : P.Obj α),
               abs (f <$> p) = f <$> abs p
```

Where `f <$> x` is Lean syntax for applying $\texttt{map}_G$ `f` to `x` (the functor `G` is inferred from the type of `x`), `abs_repr` shows that `repr` is a right-inverse of `abs`, and `abs_map` is the naturality condition of `abs`.

Fixpoint and cofixpoint

Crucially, every QPF still has an initial algebra; if F is the quotient of some polynomial functor P, then the least fixed point of F can be constructed as a quotient of P's W-type over a suitable relation. Similarly, the greatest fixed point (thus, the final coalgebra) of a QPF F can be constructed as a quotient of P's M-type.

The exact details of this construction are not relevant for the rest of this thesis, we merely need to know that there is some way to construct these fixed points. Nonetheless, the full construction can be found in the work of Avigad *et al.*. [1].

2.5 Multivariate functors

Other common examples of functors are sums (`α ⊕ β`) and products (`α × β`). These functors take two arguments (both `α` and `β`), yet, we've only discussed univariate functors (i.e., functors with a single argument).

One could see, e.g., the product functor as one of two univariate functors, `(α × ·)` or `(· × β)`, effectively choosing to fix one of the arguments. However, this limits compositionality. To illustrate, consider a type of binary trees.

```
data BinaryTree α
  | node : (BinaryTree α × BinaryTree α) → TreeOfProd α
  | leaf : α → TreeOfProd α
```

Because `BinaryTree α` is used as *both* arguments to the product, this specification is only well-formed if the product functor is functorial in both arguments (simultaneously). Thus, we generalize the definitions presented so far to *multivariate* functors.

It is difficult to reason about n-ary curried functions, where n is arbitrary, so instead an *uncurried* representation of multivariate, universe polymorphic, type functions is used.

```
def TypeFun (n : Nat) := (TypeVec.{u} n) → Type v
```

Conceptually, `TypeVec` (short for *type vector*) is a list of exactly n elements of `Type u`. We define type vectors as functions from a canonical finite type `Fin2 n`, which has exactly n inhabitants, to types.

```
def TypeVec (n : Nat) := (i : Fin2 n) → Type u
```

Notice that all arguments to a `TypeFun` live in the same universe u, but the result lives in a potentially different universe v. Most constructions require that these universes coincide, which can be written as `TypeFun.{u,u}`.

Suppose that `Sum'` is the uncurried version of sums, then `Sum' ![α₁, α₂]` would mean `α₁ ⊕ α₂`. A multivariate `map` operation now takes not one function, but a vector of n functions, each going from `αᵢ` to `βᵢ`. We define the type of such vectors of functions as,

```
def Arrow (v₁ v₂ : TypeVec n) := (i : Fin2 n) → (v₁ i → v₂ i)
```

We generally write `v₁ ⟹ v₂` instead of `Arrow v₁ v₂`, leading to the following signature for multivariate `map`.

```
map : {v₁ v₂ : TypeVec n} → (v₁ ⟹ v₂) → F v₁ → F v₂
```

Remark: Objects of type `TypeVec n`, for some fixed but arbitrary length n, and arrows $(\cdot \implies \cdot)$ form a category. Multivariate type functors are in fact nothing more than (univariate) functors from the category of type vectors to the category of types.

Thus, if `f₁ : α₁ → β₁` and `f₂ : α₂ → β₂`, then `Sum'.map ![f₁, f₂]` yields a function from `Sum' ![α₁, α₂]` to `Sum' ![β₁, β₂]`.

Finally, the functoriality constraints have an obvious generalization to the multivariate case.

Multivariate polynomial functors

An n-ary, polynomial functor is still defined with a shape A as before, but the content now maps $a \in A$ not to a single type, but to a vector of n types.

```
structure MvPFunctor (n : Nat) := (A : Type u) (B : A → TypeVec.{u} n)
```

The generalization proceeds relatively straightforwardly.

```
def MvPFunctor.Obj (P : MvPFunctor.{u} n) : TypeFun.{u,u} n
    := fun (α : TypeVec n) => Σ a : P.A, P.B a ⟹ α

def MvPFunctor.map  (P : MvPFunctor.{u} n)
                    (f : v₀ ⟹ v₁)
                        : P.Obj v₀ → P.Obj v₁
    := fun ⟨a, g⟩ => ⟨a, TypeVec.comp f g⟩
```

Where `TypeVec.comp` is the pointwise composition of two vectors of functions.

Remark: Implicit variables are not limited to types, we will use `n` and `m` for natural numbers and `v, v₀, v₁, ...` for type vectors.

Multivariate M- and W-types

The multivariate W-type is a fixed point with respect to only the last variable. Suppose P is an $n+1$-ary polynomial functor given by shapes A and contents B, then its W-type is itself an n-ary polynomial functor.

The shape of W is given by data-less, well-founded trees, whose nodes are labelled with $a \in A$ and whose children are indexed by $last(B_a)$, the last type in the vector of types B_a. That is, the shapes of W are elements of the univariate W-type given by A and $last(B_a)$.

The content for a given tree, then, is given by valid paths to the nodes in this tree.

```
/-- A path from the root of a tree to one of its nodes -/
inductive WPath : P.last.W → Fin2 n → Type u
| root (a : P.A) (f : P.last.B a → P.last.W) (i : Fin2 n) (c : P.drop.B a i) :
    WPath ⟨a, f⟩ i
| child (a : P.A) (f : P.last.B a → P.last.W) (i : Fin2 n) (j : P.last.B a)
        (c : W_path (f j) i) :
    WPath ⟨a, f⟩ i
```

Note that `P.last` denotes the (univariate) polynomial functor given by A and $last(B_a)$, the last type in vector B_a, and `P.drop` is the (multivariate) polynomial functor given by A and $drop(B_a)$, the result of removing the last type of the vector B_a.

To conclude, an element of the multivariate W-type is a pair of a tree and a function from (paths to) nodes in that tree to the data contained at that node.

```
def Wp (P : MvPFunctor (n+1)) : MvPFunctor n :=
  { A := P.last.W, B := P.WPath }
```

The M-type is defined analogously, except, the shapes are non-well-founded trees. That is, given by a univariate M-type.

```
def Mp (P : MvPFunctor (n+1)) : MvPFunctor n :=
  { A := P.last.M, B := P.WPath }
```

Multivariate QPFs

The definition of multivariate QPFs is not very different from the univariate case. We merely replace (polynomial) functors with multivariate (polynomial) functors.

```
class MvQpf {n : ℕ} (F : TypeFun.{u,_} n) [MvFunctor F]  where
  P         : MvPFunctor.{u} n
  abs       : ∀ {v}, P.Obj v → F v
  repr      : ∀ {v}, F v → P.Obj v
  abs_repr  : ∀ {v} (x : F v), abs (repr x) = x
  abs_map   : ∀ {v₀ v₁} (f : v₀ ⟹ v₁) (p : P.Obj v₀),
                abs (f <$$> p) = f <$$> abs p
```

Where `f <$$> p` is the multivariate analogue of `f <$> p`, i.e., applying `(F.map f)` to `p`.

Again, the initial algebra or final coalgebra of a multivariate QPF can be constructed as a quotient of the underlying polynomial functor's W- or M-type.

2.6 Inductive families

It is important to clarify that we're only considering inductive *types*, for which the recursive occurrences of the type being declared must not use other values for the type parameters. Lean also has inductive *families* of types for which this restriction does not hold.

Consider, for example, `Fin2`, the type of natural numbers less than n.

```
inductive BadFin2 (n : Nat)
| fz : BadFin2 (n+1)
| fs : BadFin2 n → BadFin2 (n+1)
```

This won't compile, because this defines an inductive type, and the constructors mention `BadFin2 (n+1)`, while they are only allowed to mention `BadFin2 n`.

The syntax for an inductive family is very similar; we can promote `n` from parameter to *index* by removing the binder, and giving a type signature.

```
inductive Fin2 : Nat → Type
| fz : Fin2 (n+1)
| fs : Fin2 n → Fin2 (n+1)
```

Inductive families do not correspond to initial algebras in the same way that inductive types do, and the constructions as QPFs *fundamentally* don't support inductive families, nor the coinductive analogue. Hence, we shall limit ourselves to just (co)inductive types.

Chapter 3

Porting the QPF formalization from Lean 3 to Lean 4

Development of Lean was started by Leonardo de Moura at Microsoft Research [5, 14]. As an in-progress research project, there explicitly are no stability guarantees. Indeed, Lean 4 is not backward compatible with Lean 3, meaning that the QPF formalizations by Avigad *et al.* ([1]) were not directly usable for this project. The current chapter details my efforts to port these formalizations to Lean 4.

Lean 3 has a comprehensive, community-maintained mathematics library, mathlib ([16]), which also functions as its unofficial, de-facto standard library, and to which Avigad *et al.* contributed their formalization of QPFs. Mathlib4 is the (in-progress) port of this library to Lean 4 ([17]), and I am currently in the process of contributing the ported QPF formalization back to mathlib4.

3.1 Mathport & changes from Lean 3

Lean 4 is not just a new version of the language; most of the code base has been rewritten. There were some changes in the syntax and naming convention, but most importantly, the meta-programming in Lean 4 has been completely reworked.

To help with porting code, the mathlib community developed the mathport tool ([18]), which (best-effort) translates Lean 3 source code into Lean 4 source code. This takes care of superficial syntax and naming changes, but the translation is far from perfect. In particular, mathport did reasonably well in translating declaration signatures, but the translated proofs were full of errors. This is primarily because mathport (intentionally) does not deal with meta-programming code at all and a lot of *tactics* (proof automation meta-programs) provided by mathlib were still missing from mathlib4.

3.2 Inference of implicit arguments

Lean relies heavily on inference to reduce verbosity, falling back on explicitly provided values or annotations when types or arguments are wrongly inferred. Lean 4 sometimes inferred something sufficiently different from Lean 3 that the result no longer typechecked.

For example, in the following theorem, Lean 4 couldn't figure out the right types for `f ::: g` and `f' ::: g'`. Explicit type annotations, which look like `(f ::: g : _)`, had to be added.

```
theorem append_fun_inj {α α' : Typevec n} {β β' : Type _}
                       {f f' : α ⟹ α'} {g g' : β → β'} :
  (f ::: g : (α ::: β) ⟹ _) = (f' ::: g' : (α ::: β) ⟹ _)
    → f = f' ∧ g = g'
```

Similarly, Lean has a different inference algorithm for so-called *eliminators*. Lean 3 (and more recent versions of Lean 4) support an `@[elab_as_eliminator]` attribute on user-defined functions that instructs Lean to use this specialized eliminator inference, but the version we're using does not have that. This came up, e.g., in the proof of `Cofix.bisim_aux`. The Lean 3 proof simply used

```
apply quot.induction_on c
```

Where `quot.induction_on` is marked with `elab_as_eliminator`. In Lean 4 we had to explicitly provide the *motive* (which determines the induction hypothesis), since it was being inferred incorrectly.

```
apply Quot.induction_on (motive := fun c => ∀b, r c b →
      Quot.lift (Quot.mk r') h₁ c = Quot.lift (Quot.mk r') h₁ b) c
```

The tricky part is that the proof generally doesn't throw an error at `Quot.induction_on` when it infers the wrong motive; it just produces a different proof state, tripping up later tactics. Catching this involved stepping through both the Lean 3 and Lean 4 proof states to find out where, exactly, the proof states diverged, and then explicitly providing the desired motive.

By default, the pretty printer used to show intermediate proof states will omit any implicit arguments in the term, to closely resemble whatever was written in the source code. However, we can instruct it to be more verbose (both in Lean 3 and in Lean 4):

```
set_option pp.implicit true -- print implicits, or
set_option pp.all true      -- set all options ()
```

This problem was hard to diagnose but relatively easy to remedy. Once the divergence was found, we can readily see from the Lean 3 proof state what the motive should be.

3.3 Missing tactics

Conversely, the issue of missing tactics was easy to diagnose, but harder to fix. The long-term best solution would have been to implement missing tactics. However, there were quite a few, and implementing them would probably have taken a long time.

Instead, proofs that used such missing tactics were changed to use the lower-level tactics that were available. For example, the `congr` tactic was missing, but easily replaced with either `apply congr`, `apply congrArg`, or `apply congrFun`.

Notably, mathlib4 has made much progress in implementing such missing tactics in more recent versions, so these proofs can now be changed back to use the higher-level tactics.

Chapter 4

Enhancing the QPF formalization

Besides directly porting the formalization of QPFs from Lean 3 to Lean 4, I also identified and implemented various enhancements. This chapter serves to elaborate on those changes which are too big to be considered just porting but don't fall strictly under the (co)datatype synthesis and meta-programming parts of the project.

4.1 Curried functions

Like most functional languages, it is idiomatic to write Lean functions in their curried form, so `f : Type → Type → Type`, rather than `f : (Type × Type) → Type`. However, the formalizations are done in terms of vectors of types and uncurried type functions.

```
def TypeFun (n : Nat) := TypeVec n → Type v
```

There is an obvious translation from `TypeFun` to a curried type function and, vice versa, from a curried function taking n types from the same universe and returning a type, to a `TypeFun n`. These conversions were implemented as `TypeFun.curried` and `TypeFun.ofCurried`, respectively, and it was proven that these functions are inverses.

To wit, they behave as expected:

```
variable (F : TypeFun 2) (F' : Type 1 → Type 1 → Type 2)
example : F.curried α β = F ![α, β]                  := by rfl
example : F' α β = (TypeFun.ofCurried F') ![α, β]    := by rfl
```

The type `CurriedFun α β n` is a recursively defined alias for `α → ... → α → β`, taking n arguments of type `α` to produce an element of `β`.

```
def CurriedFun (α : Type u) (β : Type v) : Nat → Type (max u v)
  | 0   => PUnit.{u+1} → β
```

```
  | 1   => α → β
  | n+1 => α → CurriedFun α β n
```

Intuitively one might expect a `CurriedFun` taking no arguments (so, $n = 0$) to be equal to just `β`, but that does not typecheck—`Type v` and `Type (max u v)` are not, in general, the same type.[1] In any case, functions without arguments are not particularly interesting for our purposes, so a simpler solution was chosen. Functions that take no arguments are seen as functions from the universe polymorphic unit type `PUnit`.

A curried type function is just an instance of `CurriedFun`.

```
def CurriedTypeFun := CurriedFun (Type u) (Type v)
```

Considering all this complexity, it is easy to see why Avigad *et al.* made all the formalizations and constructions in terms of uncurried functions. Still, uncurried functions feel very unidiomatic and users will rightfully expect their (co)datatypes to function as curried type functions. It would be interesting to see whether it is possible to reformulate the formalization of QPFs in terms of curried functions. For the time being, we'll satisfy ourselves with hiding these details through `TypeFun.curried` and `TypeFun.ofCurried` conversions.

4.2 Typeclass extensions

The following change might feel underwhelming, but it presents a considerable quality of life improvement for the `MvQpf` typeclass. The latter was originally defined as

```
class MvQpf {n : Nat} (F : TypeFun n) [MvFunctor F] where
    -- ...
```

This makes sense, since `F` can only be a QPF if it is a functor in the first place. However, when declared like this, the type of `MvQpf` is

```
{n : Nat} → (F : TypeFun n) → [MvFunctor F] → Type _
```

In particular, this means that `MvQpf F` does not name a type unless an instance of `MvFunctor F` can be inferred. For concrete QPFs this is generally not problematic, but when `F` is a variable, this restriction becomes annoying.

For example, if we wish to formalize "Let `F` be an n-ary QPF", we would like to simply write "`F` is an n-ary type function, and there is an instance of `MvQpf F`". Like so

```
variable (F : TypeFun n) [MvQpf F]
```

[1] Note that Lean's universes are *non-cumulative*, meaning that elements of `Type v` are not automatically part of higher universe `Type (v + 1)`.

This doesn't work, we have to explicitly assume a `[MvFunctor F]` bound as well. Even worse, in some situations, different mentions of `MvQpf F` could infer different `MvFunctor F` instances for the implicit argument, causing surprising type mismatches. So, the definition was changed to **`extend`** `MvFunctor`, rather than taking an argument.

```
class MvQpf {n : Nat} (F : TypeFun n) extends MvFunctor F where
    -- ...
```

Which roughly means that the `MvFunctor F` instance becomes one of the fields of the `MvQpf` typeclass. Hence, `MvQpf F` has no more implicit arguments (the value for `n` is fixed by `F`), which fixes these issues.

4.3 Typeclass inference for vectors

Composition of an n-ary functor `F` with m-ary functors $\texttt{G}_0, \texttt{G}_1, \ldots, \texttt{G}_{n-1}$ originally took the following variables.

```
variable {n m : Nat}
         (F : TypeVec.{u} n → Type _)
         [fF : MvFunctor F]
         [q : MvQpf F]
         (G : Fin2 n → TypeVec.{u} m → Type u)
         [fG : ∀ i, MvFunctor (G i)]
         [q' : ∀ i, MvQpf (G i)]
```

Firstly, by the preceding section, we can leave out the `MvFunctor` assumptions, since they are now part of the `MvQpf` assumptions.

Secondly, the last variable, `q'`, states that `G i`, for every `i` of type `Fin2 n`, is a QPF. The square brackets indicate that it is a *typeclass variable*, which should be filled in by typeclass *inference*.

As there are only n inhabitants of `Fin2 n`, the universally quantified inference problem `∀ i, MvQpf (G i)` neatly reduces to n non-qualified inference problems `MvQpf (G 0)`, `MvQpf (G 1)`, etc.

However, Lean's inference engine does not seem to be able to make this step by itself, failing to infer an instance for `∀ i, MvQpf (G i)` even if an instance of `MvQpf` can be inferred for each individual type function $\texttt{G}_\texttt{i}$.

So, we introduce a new typeclass, `VecMvQpf`, which wraps the universally quantified typeclass problem.

```
class VecMvQpf (G : Vec (TypeFun m) n) where
    prop : ∀ i, MvQpf (G i)
```

Then we can register instances by recursion on the size n of the vector G. For the base case $n = 0$, the vector `G` is empty, and it is vacuous to say all elements are QPFs.

```
instance instNil    (G : Vec (TypeFun m) 0) : VecMvQpf G
    := /- ... -/
```

For $n+1$, we recurse in the `succ` typeclass variable.

```
instance instSucc   (G : Vec (TypeFun m) (n + 1))
                    [zero : MvQpf (G .fz)]
                    [succ : VecMvQpf (fun i => G i.fs)] :
                        VecMvQpf G
    := /- ... -/
```

There is no need to write typeclass variables in terms of `VecMvQpf` because of the following instance.

```
instance instUnbox [inst : VecMvQpf G] :
    ∀i, MvQpf (G i)
```

Note that we could write `instNil` and `instSucc` directly in terms `MvQpf`, e.g.,

```
instance instSucc'  (G : Vec (TypeFun m) (n + 1))
                    [zero : MvQpf (G .fz)]
                    [succ : ∀ i, G i.fs] :
                        ∀ i, G i
    := /- ... -/
```

This is accepted, but won't actually help to derive instances for larger vectors `G`. It seems that Lean puts a limit on recursion depth when trying to infer a universally quantified typeclass problem, whereas it will recurse deeper for `VecMvQpf`.

4.4 Universe polymorphic finite type

Originally, `Fin2` was defined as the following, straightforward, inductive family, as seen in section 2.6.

```
inductive Fin2 : Nat → Type
| fz : Fin2 (n+1)
| fs : Fin2 n → Fin2 (n+1)
```

However, this definition forces `Fin2` to live in `Type`. During the project, a need arose for a finite type with exactly n inhabitants, but in arbitrary universes. Thus, the universe polymorphic `PFin2` type was added.

```
inductive PFin2 : Nat → Type u
| fz : PFin2 (n+1)
| fs : PFin2 n → Fin2 (n+1)
```

Whence `Fin2` was changed to just be an alias for `PFin2.{0}`. Most theorems and definitions for `Fin2` were easily restated in terms of `PFin2`.

Chapter 5

Designing a procedure for synthesizing QPFs from specifications

This chapter will establish a procedure that compiles a specification of a (co)datatype into the proper constructions on QPFs. It will do so in the abstract, focusing on the details of the procedure, rather than the implementation details of the Lean meta-programming system, which will be covered in the next chapter.

5.1 Shape types

Arguably the simplest, and most fundamental, inductive types are `Sum α β` and `Prod α β`, representing "either α or β" and "a pair of α and β", respectively. They can be defined as

```
inductive Sum α β                    inductive Prod α β
  | inl : α → Sum α β                  | mk : α → β → Prod α β
  | inr : β → Sum α β
```

They are also examples of what we will call *shape* types.

Definition: A *shape* type is an inductive type `Foo α_1 … α_n`, where each constructor takes only arguments of types in $\{\alpha_1, \dots, \alpha_n\}$.

That is, each constructor's arguments must be typed as one of the parameters of the shape type. Let's make this a bit clearer by looking at examples that are **not** shape types.

```
inductive List α
  | nil  : List α
```

```
  | cons : α → List α → List α

inductive ListWrapper α
  | mk : List α → ListWrapper α

inductive NatWrapper
  | mk : Nat → NatWrapper
```

The only parameter to `List` is `α`, but the `cons` constructor takes a `List α` as second argument, so `List` is not a shape type. Similarly, `ListWrapper.mk` (resp. `NatWrapper.mk`) takes an argument of type `List α` (resp. `Nat`), which are not type parameters, so these types are not shapes either.

Notice that shape types are non-recursive and do not depend on any other types, as a direct consequence of the definition. This makes them easy to translate into a polynomial functor.

> **Remark:** One way to do this translation is to realize that all shape functors can be defined as a composition of sums and products. This is similar to what the datatype package for Isabelle/HOL in [3] does. We'll use a different, slightly more monolithic approach.

Recall that (multivariate) polynomial functors are defined as

```
structure MvPFunctor (n : Nat) :=
  (A : Type u) (B : A → TypeVec n)
```

Let us return to the example of sum and product types. For the *head* type (`A` in the definition), we will take a type that has exactly as many constructors as the shape type, but such that each constructor is a constant (i.e., takes no arguments). Note that the head type does not take any type parameters.

```
inductive Sum.HeadT
  | inl : HeadT
  | inr : HeadT
```

```
inductive Prod.HeadT
  | mk : HeadT
```

The *child* family of types (`B` in the definition) maps each constructor c to a vector of types `α_c`. What is most important is the cardinality of each type `α_c i`, because that is what determines the number of arguments of the i-th type parameter needed to use constructor c.

The concrete structure of these types is not relevant, so we'll always use `PFin2 m` (see section 4.4), the type of natural numbers less than m, to construct a type with cardinality m.

Remark: We could have used `PFin2` for the head type as well, rather than generating bespoke inductive types. However, the current approach has the benefit that it's very clear which element of `HeadT` represents which constructor.

Let us start by counting for each constructor and each parameter type, how many times the constructor takes an argument of that type.

Constructor	α	β
`Sum.inl`	1	0
`Sum.inr`	0	1
`Prod.mk`	1	1

Figure 5.1: Constructor argument bookkeeping

Using the counts of fig. 5.1, we define the child family of types.

```
def Sum.ChildT : Sum.HeadT → TypeVec 2
  | .inl => ![PFin2 1, PFin2 0] -- the counts of α and β for Sum.inl
  | .inr => ![PFin2 0, PFin2 1] -- the counts of α and β for Sum.inr

def Prod.ChildT : Prod.HeadT → TypeVec 2
  | .mk  => ![PFin2 1, PFin2 1]
```

Remark: If Lean knows which type to expect, say `Sum.HeadT`, and we write an identifier with a leading dot, like `.inl`, then it will automatically add the type as namespace, concluding that we must mean `Sum.HeadT.inl`.

From here on, the construction is the same for both types; we'll show it just for `Sum`.

```
def Sum.P  : MvPFunctor 2 := MvPFunctor.mk Sum.HeadT  Sum.ChildT

def QpfSum.Uncurried : TypeFun 2 := MvPFunctor.Obj Sum.P
def QpfSum : Type → Type → Type  := TypeFun.curried QpfSum.Uncurried
```

And we're done! However, these QPF-based versions of the types are still not very nice to use. For example, if we want to construct a pair in `QpfProd`, we have to go through `MvPFunctor.mk`, which encodes its arguments in a not user-friendly way. Namely, to construct `Sum α β` from an `(a : α)`, i.e., use the `inl` constructor, it expects something of type

```
(Sum.ChildT .inl) ⟹ ![α, β]
```

Recalling that $\Longrightarrow$ is shorthand for a vector of functions, we get

```
(i : PFin 2) → (![PFin2 1, PFin 0] i → ![α, β] i)
```

That is, `inl` expects a function `f : PFin2 1 → α` together with a function `g : PFin2 0 → β`.

> **Remark:** Note that vectors are indexed right-to-left, so `![α, β] 0` is `β` and `![α, β] 1` is `α`

Recall that we defined vectors of size n as functions `PFin2 n → α`, so, equivalently, `f` is a vector of `α`s of length 1 and `g` is a trivial empty vector. So, really, the `inl` constructor only needs a single argument of type `α`, as desired. We can define a more convenient version of `inl` as follows:

```
def QpfSum.inl {α β} (a : α) : QpfSum α β :=
  MvPFunctor.mk .inl (fun (i : PFin 2) => match i with
    | 1 => ![a]
    | 0 => ![]
  )
```

Similarly, the `inr` constructor needs only a single argument of type `β`.

```
def QpfSum.inr {α β} (b : β) : QpfSum α β :=
  MvPFunctor.mk .inr (fun (i : PFin 2) => match i with
    | 1 => ![]
    | 0 => ![b]
  )
```

By defining these constructors, we can make `QpfSum` behave a bit more like the analogous **`inductive`** type. Nonetheless, it is not possible to do pattern matching with these constructor wrappers, making `QpfSum` slightly harder to use than the **`inductive`** version.

Shape types as inductive types

Because of the simplicity of shape types, i.e., their lack of any (co)induction or composition, they can just be defined as regular **`inductive`** types. However, to use shape types in subsequent constructions, e.g., taking a (co)fixpoint, needed to define more complicated (co)datatypes, we do need to show that it is a QPF.

It stands to reason that if `P` is a polynomial functor, and `F` is isomorphic to `P`, then `F` is at the very least a QPF.[1] This observation is formalized as `MvQpf.ofPolynomial`.

```
  def ofPolynomial {F : TypeFun n}
                   (P : MvPFunctor n)
                   (box    : ∀{α}, F α → P.Obj α)
                   (unbox  : ∀{α}, P.Obj α → F α)
                   (box_unbox_id : ∀{α} (x : P.Obj α), box (unbox x) = x)
                   (unbox_box_id : ∀{α} (x : F α), unbox (box x) = x)
                 : MvQpf F
```

In the case of our `Sum` example, we instantiate `ofPolynomial` with `F := TypeFun.ofCurried Sum` and `P := Sum.P`. To generate `box` and `unbox` we expand our bookkeeping a bit. For each constructor argument we generate a fresh identifier, and while doing so we keep two lists: a list of all identifiers, in the order they were introduced, and a separate list for each parameter type, with just the identifiers for that type.

To illustrate, consider the following, slightly artificial, type

```
inductive SumOfPairs α β γ where
  | pairA     : α → α → SumOfPairs α β γ
  | pairCandB : γ → β → SumOfPairs α β γ   -- note the order of β and γ
```

Following the procedure introduced at the start of this section, we obtain a corresponding polynomial functor.

```
inductive SumOfPairs.HeadT where
  | pairA     : HeadT
  | pairCandB : HeadT

def SumOfPairs.ChildT : SumOfPairs.HeadT → TypeVec 3
  | .pairA      => ![PFin2 2, PFin2 0, PFin2 0]
  | .pairCandB  => ![PFin2 0, PFin2 1, PFin2 1]

def SumOfPairs.P : MvPFunctor 3 :=
  ⟨SumOfPairs.HeadT, SumOfPairs.ChildT⟩
```

Let us use a_0 and a_1 for the arguments to the first constructor, and `c` and `b` for the second constructor. Then, we separate the arguments by their type.

The definition of `box` follows fairly directly from fig. 5.2, namely

```
def SumOfPairs.box : SumOfPairs α β γ → SumOfPairs.P.Obj ![α, β, γ]
  -- we pattern match using all variables for this constructor
  | .pairA a₀ a₁ => MvPFunctor.mk .pairA fun i => match i with
    | 2 => ![a₀, a₁]  -- α
```

[1] Informally, we would call `F` polynomial. But, recall that we formalized polynomial functors with a specific structure that might not be preserved by the isomorphism, so formally `F` might not be considered polynomial.

Type	pairA	pairCandB
all	[a_0, a_1]	[c, b]
α	[a_0, a_1]	[]
β	[]	[b]
γ	[]	[c]

Figure 5.2: Identifier bookkeeping for `SumOfPairs` constructors

```
  | 1 => ![]        -- β
  | 0 => ![]        -- γ
| .pairCandB c b => MvPFunctor.mk .pairCandB fun i => match i with
  | 2 => ![]        -- α
  | 1 => ![b]       -- β
  | 0 => ![c]       -- γ
```

Logically, `unbox` does the reverse, using the bookkeeping in fig. 5.2 to determine that for `pairA` its arguments are stored at `child 2 0` and `child 2 1`—where the first index corresponds to the type (α, β, or γ, indexed right-to-left) and the second number indexes the list of arguments specific to that type—while for `pairCandB` the arguments are found at `child 0 0` and `child 1 0`.

```
def SumOfPairs.unbox : SumOfPairs.P.Obj ![α, β, γ] → SumOfPairs α β γ
  := fun ⟨head, child⟩ => match head with
      | .pairA      => SumOfPairs.pairA (child 2 0) (child 2 1)
      | .pairCandB  => SumOfPairs.pairCandB (child 0 0) (child 1 0)
```

The proof that these two functions are inverses is a straightforward mix of case distinction and reflexivity. Finally, `ofPolynomial` can be used to show that `SumOfPairs` is a QPF.

5.2 Recursive and corecursive types

If a type is recursive but otherwise does not mention other types, it is not strictly a shape type. Nevertheless, by adding an extra type parameter ρ and substituting ρ for all (co)recursive occurrences of the type to be defined, we obtain a shape type.

For example, the shape corresponding to `List α` (as defined in section 5.1) is

```
inductive List.Shape α ρ
  | nil  : Shape α ρ
  | cons : α → ρ → Shape α ρ
```

To get rid of the extra variable ρ, we then take the fixpoint. Notice that it is required to show that `List.Shape` is a QPF before we can do this. `List.Shape` is a shape type, so we can derive an instance of `MvQpf` following the procedure of the last section, whence the following defines the type of finite lists:

```
def QpfList.Uncurried : TypeFun 2
  := MvQpf.Fix (TypeFun.ofCurried List.Shape)
```

Conversely, suppose we wish to define `CoList`, the *coinductive* type of potentially infinite lists. We introduced this type in the introduction with a specification that is similar to `List`, but mandates a corecursive interpretation.

```
codata CoList α
  | nil  : CoList α
  | cons : α → CoList α → CoList α
```

There is no difference in the procedure to obtain the corresponding shape, we just replace all occurrences of `CoList α` as constructor argument type with a new type parameter `ρ`, obtaining the exact same shape as for `List`.

```
inductive CoList.Shape α ρ
  | nil  : Shape α ρ
  | cons : α → ρ → Shape α ρ
```

But rather than taking the fixpoint, we take the *cofixpoint* for a coinductive interpretation.

```
def CoList.Uncurried : TypeFun 2
  := MvQpf.CoFix (TypeFun.ofCurried CoList.Shape)
```

5.3 Composition pipeline

The running example for this section will be the rose tree. Leaves of this tree are labelled with `α`, while internal nodes are labelled with `β` and can have a finite, non-zero number of children.

```
data QpfTree α β
  | leaf : α → QpfTree α β
  | node : β → QpfTree α β → QpfList (QpfTree α β) → QpfTree α β
```

The type is recursive, so we introduce the fresh parameter as before, taking care to also substitute it in subexpressions.

```
data QpfTree.Nonrecursive α β ρ
  | leaf : α → Nonrecursive α β ρ
  | node : β → ρ → QpfList ρ → Nonrecursive α β ρ
```

However, the result is not quite a shape type yet, we also have to get rid of `QpfList ρ` as an argument type. To do so, we simply introduce more parameters, while remembering which type these new parameters are supposed to stand for.

```
inductive QpfTree.Shape α β ρ σ₁
  | leaf : α → QpfTree α β ρ σ₁
  | node : β → ρ → σ₁ → QpfTree α β ρ σ₁
```

Then, proceed as in section 5.1 to show that `QpfTree.Shape` is a QPF.

Remark: When doing this substitution, we can reuse the same parameter for multiple occurrences of the same type. Suppose we had a constructor that takes two lists, like

```
| node₂ : β → ρ → QpfList ρ → QpfList ρ → QpfTree α β ρ
```

Then we can reuse the same fresh parameter σ_1 for both occurrences.

```
| node₂ : β → ρ → σ₁ → σ₁ → QpfTree α β ρ σ₁
```

On the other hand, if a non-parameter type also occurs as a subexpression of another type, then we will not substitute it with the same parameter. The example gets a bit contrived, but suppose nodes consist of both a list of children and a nested list of lists of children.

```
| node₃ : β → ρ → QpfList ρ → QpfList (QpfList ρ) → QpfTree α β ρ
```

This is translated to

```
| node₃ : β → ρ → σ₁ → σ₂ → QpfTree α β ρ σ₁ σ₂
```

Where σ_1 stands for `QpfList ρ`, as before, and σ_2 stands for `QpfList (QpfList ρ)`, *not* `QpfList` σ_1.

Do note that this does *not* apply when we are adding a new parameter for recursive occurrences in the very first step (like we did by adding `ρ` to obtain the `Nonrecursive` specification), such variables *do* get substituted in all subexpressions.

Returning to `QpfTree.Shape`, parameter σ_1 is supposed to stand for `QpfList ρ`, so we're aiming to define a QPF which satisfies:

```
Base α β ρ = QpfTree.Shape α β ρ (QpfList ρ)
```

The *composition pipeline* translates such an equation to a definition of `Base` in terms of the appropriate construction on QPF, such that: (a) `Base` is known to be a 3-ary QPF, and (b) the desired equality indeed holds. The composition pipeline is not just an internal detail of the procedure, we also expose it through the **`qpf`** command, whose syntax is inspired by **`def`**.

```
qpf Base α β ρ := QpfTree.Shape α β ρ (QpfList ρ)
```

As the name composition pipeline alludes, we are interested in defining compositions of QPFs. Formally, `MvQpf.Comp` has the signature:

```
TypeFun n → Vec (TypeFun m) n → TypeFun m
```

That is, an n-ary type function `F` is composed with an n-sized vector of m-ary type functions, resulting in an m-ary type function. The composition is essentially defined as

```
(Comp F ![G₁, ..., Gₙ]) ![α₁, ..., αₘ]
          = F ![G₁ ![α₁, ..., αₘ], ..., Gₘ ![α₁, ..., αₘ]]
```

All arguments `αᵢ` are broadcast to all functors `Gⱼ`, meaning we don't have to worry about argument duplication or reordering.

Continuing with the motivating example, we are going to define `Base` as `Comp QpfTree.Shape ![G₁, G₂, G₃, G₄]`, for some functors `G₁`, `G₂`, `G₃` and `G₄`, satisfying the following equalities:

```
qpf G₁ α β ρ := α                qpf G₃ α β ρ := ρ
qpf G₂ α β ρ := β                qpf G₄ α β ρ := (QpfList ρ)
```

Which we will recursively determine. Before we continue, it should be clarified that the composition pipeline is not able to satisfy all equations. For example, it is impossible to define a QPF that satisfies

```
qpf H α β := α → β
```

This is because the arrow type constructor `(· → ·)` is *not* functorial. At least, it is not functorial in both arguments. If instead, we fix any value `α` for the first argument, we obtain `(α → ·)`, which *is* a QPF. More generally, we say that in `α → β`, the variable `α` is *dead*, since it occurs as a non-functorial argument, while `β` is a *live* variable.

By default, the composition pipeline assumes arguments are live, but we can explicitly mark some as dead by giving a type ascription.

```
qpf H (α : Type u) β = α → β
```

The result is a family of QPFs, which means `H` is not a QPF, but `H a`, for arbitrary values `a : α` is.

Remark: Recall that the live parameters to a functor must all live in the same universe `Type u`, where `u` can be either some fixed level, or universe polymorphic. To make the implementation simpler, we don't allow any type ascription for live variables in the prototype implementation, instead relying on inference to determine what universe they should live in. This frees up the type ascription as a simple (to implement) way of distinguishing live and dead variables.

In a more polished future implementation, we should probably move to a more explicit syntax to mark dead variables, to minimize the risk of variables that could be live unintentionally being marked dead.

Continuing on, the composition pipeline does support these three kinds of functors:

- *Projections:* The target expression is just a parameter, as in `G₁ α β ρ = α`.
- *Constants:* The target expression does not mention the live parameters at all, e.g., `G α β = Nat` or `G (n : Nat) β = PFin2 n`.
- *Compositions:* The target expression is an application of a QPF.

This notably excludes dependent arrows `(a : α) → _`, anonymous functions `fun γ => _`, and applications that cannot be appropriately broken down (such as `α → β` when `α` is a live parameter).

Projections are trivial to identify and are represented with `MvQpf.Prj n i`, where n is the arity and i the index of the parameter to project to (counted right-to-left).

```
qpf G₁ α β ρ := α  --> def G₁ := @MvQpf.Prj 3 2
qpf G₂ α β ρ := β  --> def G₂ := @MvQpf.Prj 3 1
qpf G₃ α β ρ := ρ  --> def G₃ := @MvQpf.Prj 3 0
```

For constants, we only have to verify that the target expression does not depend on live parameters, then we translate them into `@MvQpf.Const n τ`, with n again the arity and `τ` the constant type.

```
qpf C₁ α β γ       := Nat     --> def C₁ := @MvQpf.Const 3 Nat

qpf C₂ (n : Nat) α := PFin2 n
    --> def C₂ (n : Nat) := @MvQpf.Const 1 (PFin2 n)
```

In the second example, `PFin2 n` is not strictly a constant—it depends on `n`—but it is still represented with a constant functor because n is marked a dead variable.

The last kind of supported expressions are applications `G e₁ ... eₖ` where `G` is a k-ary QPF that doesn't depend on live parameters, for arbitrary k. This is the kind of expression we discussed at the start of this section. We can represent them as `MvQpf.Comp` with recursively determined functors representing arguments `eᵢ`. Note that `G` is allowed to be an application itself, so long as it fits the set criteria (it is a QPF and does not depend on live variables), so this breakdown is not necessarily unique.

As an optimization, we don't have to generate `def` declarations for all intermediate functors, we can just inline them.

```
qpf G₄ α β ρ := QpfList ρ
  --> def G₄ α β ρ := MvQpf.Comp QpfList ![@MvQpf.Prj 3 0]

qpf Base α β ρ := QpfTree.Shape α β ρ (QpfList ρ)
  --> def Base α β ρ := MvQpf.Comp QpfTree.Shape ![
  -->     @MvQpf.Prj 3 2,                        -- G₁
  -->     @MvQpf.Prj 3 1,                        -- G₂
```

```
-->     @MvQpf.Prj 3 0,                          -- G₃
-->     MvQpf.Comp QpfList ![@MvQpf.Prj 3 0], -- G₄
--> ]
```

Finally, the desired type is obtained by taking the fixpoint. Here, too, we can inline the definition of `Base`.

```
def QpfTree α β : TypeFun 2 :=
  MvQpf.Fix (MvQpf.Comp QpfTree.Shape ![
      @MvQpf.Prj 3 2,                          -- G₁
      @MvQpf.Prj 3 1,                          -- G₂
      @MvQpf.Prj 3 0,                          -- G₃
      MvQpf.Comp QpfList ![@MvQpf.Prj 3 0], -- G₄
  ])
```

5.4 Auxiliary constructions

After constructing the (co)datatype, there are a few extra definitions we want to add to make the result more usable. To begin with, we want to generate a function for each constructor in the specification, with the same name and type. Generating these is relatively straightforward. For example, let us recall the specification for lists.

```
data List α
  | nil  : List α
  | cons : α → List α → List α
```

Following the steps described so far defines `List` in terms of a generated shape type `List.Shape` and a fixpoint. The constructors we want to generate are simple wrappers around these.

```
def List.nil : List α
  := MvQpf.Fix.mk List.Shape.nil

def List.cons : α → List α → List α
  := fun a as => MvQpf.Fix.mk (List.Shape.cons a as)
```

Effectively, we are just applying `Fix.mk` to the result of the corresponding constructor of `List.Shape`. The same method applies to codatatypes, using `CoFix.mk` instead.

At this point, we would also want to generate specialized (co)recursion principles and `noConfusion` theorems, but as mentioned in the introduction, we don't support doing so yet.

5.5 Final overview

To sum up, we can give a high-level overview of the procedure in the following steps.

1. Make the specification non-recursive by introducing a new parameter (section 5.2)
2. Introduce more parameters, to obtain a shape type `Shape` (section 5.3)
3. Show that `Shape` is a QPF, by (section 5.1)
   (a) Deriving an equivalent polynomial functor, and
   (b) Showing that this polynomial functor is isomorphic to `Shape`
4. Use the composition pipeline to solve for the intended values for parameters introduced in step 2, producing the `Base` functor (section 5.3)
5. Take the fixpoint (or cofixpoint) of `Base`, getting rid of the parameter introduced in step 1 and producing the desired functor (section 5.2)
6. Generate auxiliary constructions (section 5.4)

Steps 1 and 5 can be omitted if the specification is not (co)recursive, and steps 2 and 4 may be omitted if the input is already a shape type, but it is also fine to forgo such analysis and perform all steps anyway.

Chapter 6

Implementing the procedure as a proof of concept

In the preceding chapter, we explored how to translate definitional specifications of (co)datatypes to the fundamental constructions that can encode (co)datatypes as QPFs. In the current chapter, we will discuss the technical details of implementing this procedure in Lean 4, as **data** and **codata** commands.

To that end, we're going to dig deeper into parts of the meta-programming system. Nonetheless, we will focus on only the parts that are relevant to our implementation, for a more comprehensive introduction to meta-programming in Lean 4, the interested reader is invited to consult [19, 20].

We still won't assume knowledge of Lean specifics, but do require some more familiarity with functional programming (roughly, readers should be somewhat familiar with monads, and how they are used for side effects, as explained in, e.g., [13]).

6.1 Extending Lean's syntax

Like many modern theorem provers, Lean has facilities that allow us to register custom syntax for specific functions. For example, the following macro allows us to write `Γ ⊢ 0 : Nat` in the familiar syntax for some type checking relation `Typing Γ 0 Nat` we might define.

```
macro Γ:term " ⊢ " e:term " : " t:term : term => `(Typing $Γ $e $t)
```

Actually, this **macro** declaration is just syntactic sugar for the following two parts.

```
syntax term " ⊢ " term " : " term : term
macro_rules
  | `($Γ ⊢ $e : $t) => `(Typing $Γ $e $t)
```

Firstly, `syntax` defines a *parser extension*. Lean's grammar does not have a clearly delimited macro invocation syntax, instead, macros can freely extend Lean's syntax, influencing how the parser transforms source code into an *abstract syntax tree* (AST). The result of parsing, this AST, is represented in Lean as an object of type `Syntax`.

> **Remark:** In more recent versions of Lean, the `Syntax` type has been replaced by `TSyntax cat`, where `cat` is the (statically known) syntax category (e.g., `term` or `command`). Our code was written before this overhaul, thus we generally refer to the old way in this chapter.

In this example, the parser extension states that three terms (bound to `Γ`, `e` and `t`), interspersed with `⊢` and `:` symbols form a new term.

Then, the `macro_rules` part defines the *semantics* of the macro, as a substitution. We both match on, and create new syntax trees, using *syntax quotations* (written as `` `() ``). We can use a variable of type `Syntax` in a syntax quotation as, e.g, `` `($Γ) ``, this is called an *anti-quotation*. By substituting on the level of syntax trees, grouping and precedence are preserved. For example, `Γ ⊢ 1 + 2 * 3 : Nat` is correctly translated to `Typing Γ (1 + 2 * 3) Nat`.

Clearly, though, `data` and `codata` would be hard to define in terms of such substitutions. We could, in theory, replace the right-hand side of a `macro_rules` match arm with arbitrary code that produces a `Syntax` object, and in this way, define a *procedural macro*. This is, however, not idiomatic; complex commands are usually defined as an *elaborator*.

Data and codata Syntax

Before we get to what an elaborator is, we'll take a few steps back and consider how to define the parser extensions for `data` and `codata`.

The flexibility and extensibility of Lean are not just for users. The Lean 4 compiler is mostly written in Lean 4 itself, and it turns out that a lot of the language syntax is implemented with the meta-programming system. Amongst them, `inductive`, whose syntax is defined with the following parser extension.

```
def «inductive» := leading_parser "inductive " >> declId >> optDeclSig
                   >> optional (symbol " :=" <|> " where")
                   >> many ctor
                   >> optDeriving
```

> **Remark:** Notice how we wrote `«...»` around `inductive`; these brackets allow us to use reserved keywords as identifiers.

Where `leading_parser` is a still lower-level macro to define a parser extension than the **`syntax`** declaration we saw before, and `declId`, `optDeclSig`, etc. are all parsers, or parser combinators. Combinators `a >> b` and `a <|> b` mean "first parse `a`, then `b`" and "parse `a` *or* `b`", respectively.

This parser accepts an `"inductive"` *atom* (or, literal), followed by a declaration identifier (`declId`), an optional signature (`optDeclSig`), optionally a `" :="` or `"where"` atom, followed by zero or more (`many`) constructor specifications (`ctor`), and finally, an optional list of typeclasses to derive (`optDeriving`).

This `Parser` instance is then used in the `declaration` parser.

```
@[builtinCommandParser] def declaration := leading_parser
        declModifiers false >> (/- ... -/ <|> «instance» <|> /- ... -/)
```

The `builtinCommandParser` attribute registers `declaration` as a builtin syntax extension for the `command` category.

> **Remark:** There are many syntax categories, of which `term` and `command` are by far the most common. The former, `term`, is the syntax category for terms, e.g., `x + y` or **`Type`** → `Nat` → **`Type`**, whereas `command` is the category for top-level commands, such as **`inductive`**, but also definitions (**`def`**), theorems (**`theorem`**), etc.

We want **`data`** to be (mostly) suitable as a drop-in replacement for **`inductive`**, so we copy the latter's parser verbatim, replacing the `"inductive"` atom with `"data"`. Similarly, **`codata`** also uses the same syntax, except the command is now `"codata"`.

```
def inductive_like (cmd : String) : Parser
  := leading_parser cmd >> declId  >> optDeclSig
                        >> Parser.optional  (symbol " :=" <|> " where")
                        >> many ctor
                        >> optDeriving

def data   := inductive_like "data "
def codata := inductive_like "codata "
```

> **Remark:** The syntax highlighting is slightly misleading. At this point the parser does not know about **`data`** and **`codata`** as commands yet, so they don't need to be enclosed with `«...»` brackets to be used as identifiers.

Since we are not defining builtin syntax, we use the `commandParser` attribute to define the parser extension rather than `builtinCommandParser`.

```
@[commandParser] def declaration := leading_parser
    declModifiers false >> (data <|> codata)
```

After this definition, Lean knows how to successfully parse datatype specifications such as

```
data Foo α
  | foo : α → Foo α
```

However, we only defined the syntax, not the semantics. The code above still fails, complaining that we did not define an *elaboration function* for `declaration`.

Unsupported syntax

Note that parser extensions are not strictly about which syntax is *supported*. Instead, they declare which syntax this command is *responsible* for. Suppose, for example, that a user tried to derive a typeclass for their datatype.

```
data Foo α
  | foo : α → Foo α
  deriving BEq
```

Although this syntax is accepted by the **`data`** parser, we don't actually support deriving typeclasses yet.

Removing the `>> optDeriving` part from the parser definition would bring the accepted syntax closer to what is actually supported, making `Foo` as written above lead to a generic parse error.

Nevertheless, the user is clearly trying to invoke our **`data`** command, and **`inductive`** does support such a **`deriving`** statement, so it is to be expected that users will try to use it. The generic parse errors are quite obscure and bad at communicating to the user that **`deriving`** statements are not supported; a user might reasonably think the parse error is because of some (non-existent) typo.

So instead, we have the parser accept exactly the same syntax as **`inductive`**, and in the *elaborator* we then throw a custom, informative error when a user tries to use unsupported constructs. In this way, **`data`** is truly a drop-in replacement for **`inductive`**, syntax-wise, and any errors will guide the user and explain unsupported syntax.

6.2 Defining the semantics of a command

With the parser extension in place, the next step is to give our commands their semantics, by defining a corresponding *command elaborator*. That is, we have to define the Lean code that is run when a **`data`** or **`codata`** command is invoked.

We already mentioned that the two main syntax categories are terms and commands. Both have a slightly different kind of elaborator.

Elaborating a term means to translate it to an expression of the core logic of Lean (as encoded by the `Expr` type). A *term elaborator* is thus a function `Syntax → TermElabM Expr`, where `TermElabM` is the term elaboration monad. This monad gives (read-only) access to the *environment* (i.e., declarations and imports that the term can refer to) and *metavariable context* (roughly speaking, a metavariable represents a hole in the program that should be synthesized/inferred at some point during compilation).

Commands, on the other hand, are not translated but used for their side effect. Hence, the type of a *command elaborator* is `Syntax → CommandElabM Unit`, where the `CommandElabM` monad allows for four kinds of side effects:

1. Modifying the environment (e.g., **`inductive`**, **`def`**, **`theorem`**),
2. Logging messages to inform the user (e.g., `#check`, `#print`, `#eval`),
3. Performing IO, and
4. Throwing errors

Although `TermElabM` neither extends nor is extended by `CommandElabM`, it is common to elaborate terms as part of a command elaborator, e.g., by using `liftTermElabM` or `runTermElabM`.

Elaborator registration

Besides defining a function of the appropriate type, an elaborator also needs to be decorated with a `commandElab` attribute.

```
@[commandElab declaration] def elabData : Syntax → CommandElabM Unit
```

The argument to the attribute, `declaration`, refers to the parser we defined in the preceding section, which is how Lean knows to use this elaborator for **`data`** and **`codata`** commands. Before going further into the implementation of `elabData`, we'll go over some high-level choices that were made, and their trade-offs.

6.3 Elaborating inductive declarations

For the elaborator, too, we draw inspiration from how **`inductive`** is implemented. Elaboration starts in the generic declaration elaborator, which checks what kind of declaration it is given, and defers to `elabInductive` (in case of an inductive declaration). The latter, in turn, calls `inductiveSyntaxToView` to transform the `Syntax` object into a `InductiveView` and defers to `elabInductiveViews`.

This `InductiveView` is a thin wrapper around the syntax tree, allowing us to refer to the different parts of an inductive declaration by name, rather than offset, but crucially, still stores, e.g., constructor types as `Syntax` objects.

At this point, auto-bound implicit variables are added, the constructor type terms are elaborated, and type universes are inferred (if needed), to produce an elaborated `InductiveType` object. Finally, the `InductiveType` is wrapped in a `Declaration`, added to the environment, and auxiliary constructions are generated with `mkAuxConstructions`.

Most of this work, besides the steps that modify the environment, is also relevant for **`data`** and **`codata`** declarations.

An attempt was made to copy the code of `elabInductiveViews` (and all private or protected functions it called) and factor out all work that is relevant for both **`inductive`** and **`data`**/**`codata`** declarations into a common function. However, it turned out that some elaboration steps needed to produce the `InductiveType` object were not desirable for then running the procedure of chapter 5 on.

For example, type parameters are added to constructor types as implicit arguments during the elaboration steps.

```
inductive Foo α β
  | mk : α → β → Foo α β

inductive Bar α
  | mk : {β : Type} → α → β → Bar α

-- Foo.mk has type `{α: Type} → {β: Type} → α → β → Foo α β`
-- Bar.mk has type `{α: Type} → {β: Type} → α → β → Bar α`
```

However, we would like to give as input to the procedure just (expressions corresponding to) types `α → β → Foo α β` and `β : Type → α → β → Bar α`. It is, of course, possible to detect which of the implicit argument are type parameters, by looking at which variables are passed to `Foo`, resp. `Bar`, but this adds complexity.

In the end, the required changes were deemed to require too much engineering effort. Since the implementation is intended purely as a prototype, a simpler approach was taken: most steps of the procedure are performed on the level of syntax, rather than on elaborated expressions. The composition pipeline is an exception: for that step we do have to elaborate the provided terms.

Concretely, a `DataView` structure was defined as a thin wrapper around a **`data`** or **`codata`** syntax tree, just like `InductiveView`. This structure is then passed around to the different steps of the procedure, to represent the specification.

6.4 Adding declarations to the environment

A key part of `elabData` is to generate declarations and add them to the environment. This can be done in multiple ways, in roughly increasing order of both robustness and complexity:

1. Generate source code as a `String`
2. Generate a `Syntax` or `InductiveView` object

3. Generate a fully elaborated `Declaration` object

The first approach, while easy, is too brittle. There exists a function, `runFrontend`, that can be used to parse and elaborate the code in a string, but it is designed to be called with the content of a `.lean` file of user-written source code, not with code generated during command elaboration.

The last approach requires full elaboration of all terms involved, which, as discussed in the previous section, was deemed too complicated for our prototype.

Therefore, the second approach was chosen, relying on *syntax quotations* to simplify the generation of syntax trees. Since `InductiveView` is just a thin wrapper around syntax trees, we consider it as being part of the syntactic approach.

For example, if we want to define `Foo` as the string `"Bar"`, this could be done with

```
let stx : Syntax ← `(def Foo := "Bar")
elabCommand stx
```

The variable `stx` stores the parsed syntax tree, and `elabCommand` is a builtin function `Syntax → CommandElabM Unit` that takes a `command` syntax tree and calls the elaborator that was registered for that particular kind of command.

By default, a syntax quotation will try to parse whatever code we write as either a command or a term. There are situations, however, where we want to build up a command (or term) incrementally, requiring us to generate syntax that, by itself, is not a complete command (or term). Imagine, for example, that, given some array `[`A, `B, `C]` of names, we want to generate the following inductive type:

```
inductive Baz
  | A : Baz
  | B : Baz
  | C : Baz
```

The code to do so might look as follows:

```
open Lean.Parser.Command in
def inductiveFromNames (names : Array Name) : CommandElabM Unit := do
  let Baz := mkIdent `Baz
  let ctors ← names.mapM fun name =>
  let ctorIdent := mkIdent name
    `(ctor|
      | $ctorIdent:ident : $Baz:ident
    )

  elabCommand (←`(
    inductive $id:ident
      $[$ctors:ctor]*
  ))
```

Of special interest is the `(ctor| ...) syntax quotation, which specifies that the syntax should be parsed with the ctor parser.

The shape of the syntax tree produced by a quotation is determined statically, not at runtime, so the parser cannot examine the values of the ctorIdent and ident anti-quotations, it treats them as opaque blobs of syntax. We have to help it a bit by specifying the syntax category explicitly, as :ident. Similarly, in the final **inductive** syntax quotation, we specify that ctors is an array of syntax trees of the ctor category.

> **Remark:** With the overhaul we mentioned at the start of this section, syntax categories are now tracked in the type system, making such hints obsolete in newer versions of Lean.

The $[...]* part is called a *splice*, and it allows us to use an array of syntax objects whenever a parser expects a whitespace-separated list of things (as is the case with the many parser combinator used in the **inductive** parser).

Take note that we have to create the identifier Baz with the mkIdent function if we want it to be accessible. By default, Lean's macros are *hygienic* (see [20]), meaning that an identifier Baz written directly into a syntax quotation is not accessible outside the macro, making sure they don't accidentally clash or shadow other identifiers at the call-site. The mkIdent function allows us to opt out of hygiene so that the identifier is just Baz as intended.

6.5 Implementing the elaborator

Now that we have a general idea of the meta-programming system of Lean, and a high-level implementation strategy, let us briefly examine how elabData, the elaborator for **data** and **codata** is implemented. By design, the steps of the procedure we described in section 5.5 closely correspond to the different functions that make up the elabData implementation. We will now describe roughly the same overview, but with a focus on which functions implement which step.

First, makeNonRecursive takes a DataView and returns a new, non-recursive DataView (step 1). Then, mkShape takes this non-recursive view, defines Shape as an inductive type (step 2) and calls mkQpf to derive the corresponding polynomial functor and establish its isomorphism with Shape (step 3).

Continuing, elabQpfComposition is the entry point into the composition pipeline (step 4), and is used to obtain a syntax tree that represents the Base functor. Nothing is added to the environment at this step.

Depending on which command was called, mkType will take either the fixpoint or cofixpoint of Base, and add the result to the environment (step 5). It will do so first in the uncurried form, as F.Uncurried, and then define F as @TypeFun.curried n F.Uncurried, where F is whatever identifier the user supplied and n is the arity. Finally, mkConstructors will derive the expected constructor functions (step 6)

Chapter 7

Limitation and future work

As mentioned before, the implementation is not yet complete. This chapter will explain some limitations in what is implemented so far, and discuss directions for future implementation efforts.

7.1 Universe polymorphism

Recall from section 5.1 that the procedure starts with synthesizing a head type, roughly like the following:

```
inductive HeadT
  | CtorA : HeadT
  | CtorB : HeadT
```

The names and number of constructors will differ, but crucially Lean will generate a *non* universe polymorphic type. That is, `HeadT` lives in `Type 0`, and only `Type 0`. As a result, any QPF defined with **`data`**/**`codata`** will *not* be universe polymorphic. Recall the running example of lists,

```
data List α
  | nil  : List α
  | cons : α → List α → List α
```

The **`inductive`** analogue of this definition *is* polymorphic, but because of this limitation, `List` defined in terms of **`data`** is *not*.

The solution is fairly simple, we can explicitly state that we want `HeadT` to be polymorphic—noting that the child family of types is already universe polymorphic because of the use of `PFin2` (see sections 4.4 and 5.1).

```
inductive HeadT : Type u
```

```
  | CtorA : HeadT
  | CtorB : HeadT
```

However, there is a bug in the version of Lean we're using (`nightly-2022-04-28`) which causes it to reject the above. Newer versions of Lean 4 don't exhibit this bug, so it is easy to remove this limitation after updating our code to be compatible.

7.2 Characteristic theorems

The formalizations of Avigad *et al.* provide (co)recursion principles, such as `Fix.drec` and `Cofix.corec`. However, because of how these are defined, their signature is quite obscure and not directly suitable for end-users.

```
#check @MvQpf.Fix.drec _ List.Internal _
-- {α : TypeVec 0} →
--   {β : MvQpf.Fix List.Internal α → Type} →
--     ((x : List.Internal (α :::₀ Sigma β)) →
--        β (MvQpf.Fix.mk ((TypeVec.id ::: Sigma.fst) <$$> x))
--     ) →
--       (x : MvQpf.Fix List.Internal α) → β x
```

In contrast, the **inductive** version of `List` defines its recursion principle as

```
def List.rec {α : Type _} {motive : QpfList α → Sort _} :
motive QpfList.nil
  → ((head : α) → (tail : QpfList α) → motive tail
                                       → motive (QpfList.cons head tail))
  → (t : QpfList α)
  → motive t
```

The latter recursion principle is a lot easier to read by comparison. To make our datatypes behave as similarly to Leans native **inductive** types, we should generate the same recursion principles as Lean would. In this particular case, the second recursion principle is fairly easily defined in terms of `MvQpf.Fix.drec`.

```
def List.rec {α : Type _} {motive : QpfList α → Sort _} : /- ... -/ :=
  fun nil cons => MvQpf.Fix.drec (fun x =>
    match x with
    | .nil            => nil
    | .cons head tail => cons head tail.fst tail.snd
  )
```

However, once we involve nested induction, the recursion principle generated by Lean becomes more complex.

```
inductive Tree (α : Type) where
| node : α → List (Tree α) → Tree α

#check @Tree.rec
-- {α : Type} → {motive_1 : Tree α → Sort u_1} →
--   {motive_2 : List (Tree α) → Sort u_1} →
--   ((a : α) → (a_1 : List (Tree α))
--       → motive_2 a_1 → motive_1 (Tree.node a a_1)) →
--   motive_2 [] →
--   ((head : Tree α) → (tail : List (Tree α))
--       → motive_1 head → motive_2 tail → motive_2 (head :: tail)) →
--   (t : Tree α) → motive_1 t
```

More work is required to determine how we can synthesize such specialized recursion principles from a datatype specification with nested induction, and it should be investigated how mixed induction/coinduction interacts with such recursors.

Similarly, we should generate no confusion theorems; proving that different combinations of constructors produce different elements.

7.3 User-friendly (co)recursion and (co)induction

If we just want to do structural recursion on a single variable, it's relatively easy to do so directly in terms of the fundamental recursion principle. For example, determining the length of a list.

```
def List.length : List α → Nat :=
  MvQpf.Fix.rec fun as => match as with
    | .nil               => 0
    | .cons a (as : Nat) => as + 1
```

It is important that Lean can statically determine that recursive functions terminate for all arguments.[1] For this example, termination is straightforward, but Lean also has a sophisticated *equation compiler* that makes it easy to define functions that recurse on multiple variables, with support for termination hints, to help when the system fails to automatically prove termination.

It would be interesting to see if the existing equation compiler mechanisms can be re-used to also provide a high-level way of defining recursive functions for QPFs, so that users can write recursive functions on QPF-based datatypes in the same as they would for a regular **inductive** type.

The dual of termination of recursive functions is productivity of corecursive functions. *Guarded corecursion* is the trivially productive analogue of structured recursion, and supporting an easy syntax for defining guarded corecursive functions would be a great first step. Further steps might focus on a more general strategy for proving productivity of non-guarded corecursion, such as [21] did for Isabelle.

[1] Lean *can* admit non-terminating functions when they're marked partial, but partial functions are not allowed to be used in proofs

Chapter 8

Conclusion

In this thesis, I presented the procedure behind, and implementation of, a (co)datatype package for Lean 4. The package enables users to ergonomically define coinductive types to Lean 4 in a compositional way, with support for mixed induction, coinduction, and quotients.

The package is based on the category theoretical notion of a quotient of polynomial functor. An existing formalization of QPFs and their constructions was ported from Lean 3 to Lean 4. The updated formalizations will eventually be made available as part of the mathlib4 project, the port of Leans community-maintained, comprehensive mathematics library.

The prototype implementation makes extensive use of Leans meta-programming system. Specifically, the `data` and `codata` commands accept the same syntax as Leans standard `inductive` type specifications. We also discuss the details of elaborating both commands in Lean, and weighed the trade-offs involved with synthesizing new declarations.

That being said, defining a type is only part of the equation. More work is needed to simplify the definition of (co)recursive functions. We have briefly outlined some strategies that future work might pursue.

Bibliography

[1] J. Avigad, M. Carneiro, and S. Hudon, "Data Types as Quotients of Polynomial Functors," in *10th International Conference on Interactive Theorem Proving (ITP 2019)* (J. Harrison, J. O'Leary, and A. Tolmach, eds.), vol. 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 6:1–6:19, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. https://doi.org/10.4230/LIPIcs.ITP.2019.6.

[2] J. Biendarra, J. Blanchette, M. Desharnais, L. Panny, A. Popescu, and D. Traytel, "Defining (Co)datatypes and Primitively (Co)recursive Functions in Isabelle/HOL." https://isabelle.in.tum.de/dist/doc/datatypes.pdf. Online Documentation.

[3] D. Traytel, "A category theory based (co)datatype package for Isabelle/HOL," Master's thesis, TU München, München. https://www21.in.tum.de/~traytel/mscthesis.pdf.

[4] B. Fürer, A. Lochbihler, J. Schneider, and D. Traytel, "Quotients of Bounded Natural Functors," *Logical Methods in Computer Science*, vol. 18, Feb. 2022. https://lmcs.episciences.org/9022.

[5] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, "The Lean Theorem Prover (System Description)," in *Automated Deduction - CADE-25* (A. P. Felty and A. Middeldorp, eds.), vol. 9195, (Cham), pp. 378–388, Springer International Publishing, 2015. https://doi.org/10.1007/978-3-319-21401-6_26.

[6] T. Coquand, *Metamathematical Investigations of a Calculus of Constructions*. Report, INRIA, 1989. https://hal.inria.fr/inria-00075471.

[7] E. Giménez, "An application of co-inductive types in Coq: Verification of the alternating bit protocol," in *Types for Proofs and Programs* (G. Goos, J. Hartmanis, J. Leeuwen, S. Berardi, and M. Coppo, eds.), vol. 1158, (Berlin, Heidelberg), pp. 135–152, Springer Berlin Heidelberg, 1996. https://doi.org/10.1007/3-540-61780-9_67.

[8] E. Giménez, "A Tutorial on Recursive Types in Coq," tech. rep., INRIA, May 1998. https://hal.inria.fr/inria-00069950/document.

[9] H. Basold, *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic*. PhD thesis, Radboud Universiteit, Nijmegen, 2018. https://repository.ubn.ru.nl/handle/2066/190323.

[10] H. Basold and H. Geuvers, "Type Theory based on Dependent Inductive and Coinductive Types," in *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, (New York NY USA), pp. 327–336, ACM, July 2016. https://dl.acm.org/doi/10.1145/2933575.2934514.

[11] S. Awodey, *Category Theory*. No. 52 in Oxford Logic Guides, Oxford; New York: Oxford University Press, second ed., 2010.

[12] B. Milewski, *Category Theory for Programmers*. Milton Keynes: Lightning Source UK, 2019.

[13] D. T. Christiansen, "Functional Programming in Lean." https://leanprover.github.io/functional_programming_in_lean/. Online book.

[14] J. Avigad, L. de Moura, S. Kong, and S. Ullrich, "Theorem Proving in Lean 4." https://leanprover.github.io/theorem_proving_in_lean4/. Online Documentation.

[15] J. Girard, *Interprétation Fonctionelle et Élimination Des Coupures de l'arithmétique d'ordre Supérieur*. PhD thesis, Université Paris, Paris, 1972.

[16] The mathlib Community, "The lean mathematical library," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, (New Orleans LA USA), pp. 367–381, ACM, Jan. 2020. https://dl.acm.org/doi/10.1145/3372885.3373824.

[17] The mathlib Community, "Mathlib4." GitHub repository. https://github.com/leanprover-community/mathlib4.

[18] The mathlib Community, "Mathport." GitHub repository. https://github.com/leanprover-community/mathport.

[19] A. Paulino, D. Testa, E. Ayers, H. Böving, J. Limperg, S. Gadgil, and S. Bhat, "Metaprogramming in Lean 4." Online Book. https://github.com/arthurpaulino/lean4-metaprogramming-book.

[20] S. Ullrich and L. de Moura, "Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages," *Logical Methods in Computer Science*, vol. 18, Apr. 2022. https://doi.org/10.46298/lmcs-18(2:1)2022.

[21] J. C. Blanchette, A. Bouzy, A. Lochbihler, A. Popescu, and D. Traytel, "Friends with Benefits: Implementing Corecursion in Foundational Proof Assistants," in *Programming Languages and Systems* (H. Yang, ed.), vol. 10201, pp. 111–140, Berlin, Heidelberg: Springer Berlin Heidelberg, 2017. https://doi.org/10.1007/978-3-662-54434-1_5.