Compositional Program Verification with Polynomial Functors in Dependent Type Theory

C.B. Aberlé
Carnegie Mellon University
caberle@andrew.cmu.edu

We present a framework for compositional program verification based on polynomial functors in dependent type theory. In this framework, polynomial functors serve as program interfaces, Kleisli morphisms for the free monad monad serve as implementations, and dependent polynomials encode pre/postcondition specifications. We show that implementations and their verifications compose via wiring diagrams, and that Mealy machines provide a compositional coalgebraic operational semantics. We identify the abstract categorical structure underlying this compositionality as a monoidal functor from specifications to interfaces with a compatible monoidal natural transformation of lax monoidal presheaves; this opens the door to generalizations to other categories, monoidal products, etc., including settings for concurrency and relational verification, which we sketch. As a proof-of-concept, the entire framework has been formalized in Agda.

1 Introduction

Large software systems are often opaque; their capabilities need not be. To verify the behavior of complex programs, we ought to be able to decompose them into simpler components, verify those components independently, and compose the results. In this paper, we develop such a *compositional* approach to program verification using *polynomial functors* in dependent type theory. A polynomial functor $P(y) = \sum_{a:A} y^{B(a)}$ naturally represents the interface of a dependent function $(a : A) \to B(a)$, and a morphism to the *free monad* on another polynomial represents a program module implementing one interface by calling another. These constructions generalize to *dependent polynomials* encoding pre/postcondition specifications, allowing programs to be verified in exactly the same manner they are built up.

This perspective connects to work on polynomial functors as a theory of interaction (Niu and Spivak 2024; Libkind and Spivak 2025; Spivak 2022). Our contribution is to extend this picture to dependent type theory, introducing dependent polynomials that encode program specifications, and showing that the resulting verification framework is fully compositional. Specifically, we show that implementations and verifications compose along *wiring diagrams* (§2–3); that *Mealy machines* provide a compatible coalgebraic operational semantics (§4); that *dependent polynomials* and *dependent free monads* yield compositional verification (§5–6); and that the abstract categorical structure underlying this compositionality can be identified as a monoidal functor with compatible lax monoidal presheaves (§7) connected by a monoidal natural transformation. We also sketch an extension to concurrent modules via a parallel monoidal product (Appendix A). The entire development has been formalized in Agda (Norell 2009).

Submitted to:
ACT 2026

2 Polynomial Functors and Program Interfaces

2.1 Polynomial Functors

A *polynomial functor* is an endofunctor on the category of sets (or, in our case, types) of the form

$$P(y) = \sum_{a:A} y^{B(a)}$$

where A is a type and $B : A \to \mathsf{Set}$ is a family of types indexed by A. We call A the type of *positions* and $B(a)$ the type of *directions* at position a (Niu and Spivak 2024). Concretely, we represent a polynomial functor as a pair (A, B):

```
Poly : Set₁
Poly = Σ Set (λ A → A → Set)
```

Given a polynomial $p = (A, B)$ and a functor $F : \mathsf{Set} \to \mathsf{Set}$, the type of *morphisms* (natural transformations) from p to F is:

```
_⇒_ : Poly → (Set → Set) → Set
(A , B) ⇒ F = (x : A) → F (B x)
```

A morphism $p \Rightarrow q$ between two polynomials, where $p = (A, B)$ and $q = (C, D)$, thus consists of a *forward* map $f_0 : A \to C$ on positions and a *backward* map $f^1 : (a : A) \to D(f_0(a)) \to B(a)$ on directions. In the language of the polynomial functors literature, this is a *lens* from p to q (Niu and Spivak 2024, Ch. 3).

Thinking of (A, B) as the interface of a dependent function $g : (a : A) \to B(a)$, and (C, D) as the interface of a function $h : (c : C) \to D(c)$, a lens from (A, B) to (C, D) encodes a program that implements g by calling h: given input $a : A$, it calls h on $f_0(a)$, then applies $f^1(a)$ to the result to produce an output of type $B(a)$. This is the basic pattern of *assume-guarantee reasoning* that we shall see throughout the paper: assuming an implementation of h, we guarantee an implementation of g.

2.2 Free Monads and Sequential Composition

The notion of morphism/lens described above only captures situations where the assumed function is called *exactly once*. To express more general patterns of interaction, where the assumed interface may be called zero, one, or many times—with each subsequent call potentially depending on the results of previous ones—we use the *free monad* on a polynomial functor (Niu and Spivak 2024; Libkind and Spivak 2025):

```
data Free (p : Poly) (C : Set) : Set where
    return : C → Free p C
    bind : (x : p .fst) → (p .snd x → Free p C) → Free p C
```

An element of $\mathsf{Free} p C$ is a computation that either returns a value of type C, or makes a call to $p = (A, B)$ with some input x, receives a response, and continues depending on the result—equivalently, such programs correspond to well-founded trees with leaves in C (Libkind and Spivak 2025; Niu and Spivak 2024) and internal nodes labeled by positions $a : A$, with branches indexed by directions $b : B(a)$.

As the name would imply, the free monad carries a monadic structure, with `bind` sequencing computations. We use Agda's `syntax` mechanism to introduce notations as follows: `call[ b ← a ] e` makes a single call with input `a`, binds the result to `b`, and continues with `e`; `do[ b ← m ] e` sequences a sub-computation `m` similarly (the reader familiar with monadic programming in languages such as Haskell will recognize these patterns). These are used throughout the paper.

This brings us to our central definition for program modules:

Definition 2.1. An *implementation* of interface p depending on interface q is a morphism $p \Rightarrow \mathsf{Free}\, q$, i.e., a function that, given any input to p, produces a computation in the free monad on q that returns an appropriate output.

2.3 Sums of Polynomial Functors

In practice, program modules and their interfaces often involve *multiple* functions. We handle this via the *sum* of polynomial functors. Given an indexing type U and a family $P : U \to \mathsf{Poly}$, their sum $\sum_{u:U} P(u)$ is again a polynomial, which is in fact the coproduct in the category **Poly** of polynomial functors and their morphisms:

```
sum : (U : Set) → (p : U → Poly) → Poly
sum U p .fst = Σ U (λ u → p u .fst)
sum U p .snd (u , x) = p u .snd x
```

A morphism $\mathsf{sum}\, U\, P \Rightarrow \mathsf{Free}\, q$ then amounts to a family of implementations, one for each label $u : U$. As special cases, we write $p \oplus q$ for the binary sum and $\mathsf{zeroPoly}$ for the nullary sum (the empty interface). An implementation $p \Rightarrow \mathsf{Free}\, \mathsf{zeroPoly}$ depends on no external interfaces—i.e. it is a *closed* module—and therefore corresponds precisely to a dependent function $(a : A) \to B(a)$, since $\mathsf{Free}\, \mathsf{zeroPoly}\, C \cong C$.

3 Program Modules and Composition

3.1 Example: Fold, Append, Concat

To illustrate the framework, we present a generic fold operation on lists. The fold module has interface $\mathsf{Fold}\, A\, B\, C = (A \times \mathsf{List}\, B, \lambda\_ \to C)$ and depends on two sub-interfaces: a *base case* $\mathsf{Base}\, A\, C = (A, \lambda\_ \to C)$ and a *recursive step* $\mathsf{Step}\, A\, B\, C = (A \times B \times C, \lambda\_ \to C)$.

The implementation proceeds by structural recursion on the input list: on the empty list, it calls the base case; on a cons cell, it recursively folds the tail, then calls the step with the accumulated result:

```
fold : {A B C : Set}
     → Fold A B C
       ⇒ Free (sum foldlabels
                   (λ{ base → Base A C
                     ; step → Step A B C}))
fold (a , nil) =
    call[ c ← (base , a) ]
    return c
```

```
fold (a , cons b bs) =
    do[ c ← fold (a , bs) ]
    call[ c' ← (step , (a , b , c)) ]
    return c'
```

Specific instances of folds, such as appending two lists or concatenating a list of lists, are then obtained by providing closed implementations of the base and step interfaces and plugging them into the fold module. For append, the base case returns the accumulator unchanged, and the step prepends an element—both are closed (depending on zeroPoly). Concat is obtained similarly, using `append` as a subroutine for its step case (see Appendix D for the full code).

3.2 Composing Implementations

Given $f : p \Rightarrow \mathsf{Free}\, q$ and $g : q \Rightarrow \mathsf{Free}\, r$, we can substitute g for each call to q in the computations produced by f, obtaining a composite $f \circ g : p \Rightarrow \mathsf{Free}\, r$. This is precisely the Kleisli composition for the free monad monad on **Poly**: given $e : \mathsf{Free}\, p\, E$, we recursively substitute g for each `bind` node, using `>>=` to splice the resulting trees. More generally, when a module depends on a *sum* of interfaces, we can compose it with a family of implementations:

```
comp : (U : Set) {p : Poly} {q : U → Poly} {r : Poly}
     → (p ⇒ Free (sum U q))
     → ((u : U) → q u ⇒ Free r)
     → p ⇒ Free r
comp U f g = f ∘ (λ (u , x) → g u x)
```

Using these combinators, we compose fold with the append base and step to obtain a closed module implementing append, and similarly for concat, where the step case composes with append to call it as a subroutine:

```
append : {A : Set}
       → Fold (List A) A (List A) ⇒ Free zeroPoly
append =
    comp foldlabels fold
         (λ{ base → appendBase
           ; step → appendStep})

concat : {A : Set}
       → Fold ⊤ (List A) (List A) ⇒ Free zeroPoly
concat =
    comp foldlabels fold
         (λ{ base → concatBase
           ; step → concatStep ∘ append})
```

3.3 Wiring Diagrams

The pattern of composing modules along dependency structures can be generalized using *wiring diagrams* (Spivak 2013), where boxes represent modules, wires represent dependencies, and the outer boundary represents the composite interface. A wiring diagram is parameterized by a set

of `Boxes` with their `Arity` (dependencies), `Dom`/`Cod` (dependency and output interfaces), and `Inputs` (external dependencies):

```
module WD (Boxes : Set)
          (Arity : Boxes → Set)
          (Dom : (b : Boxes) → Arity b → Poly)
          (Cod : Boxes → Poly)
          (Inputs : Set) (inputs : Inputs → Poly) where

    data Wiring : (output : Poly) → Set₁ where
        wire : (i : Inputs) → Wiring (inputs i)
        box : (b : Boxes) → ((a : Arity b) → Wiring (Dom b a))
            → Wiring (Cod b)
```

The constructor `wire i` represents an external dependency routed in from outside the diagram, while `box b f` places box `b` in the diagram and wires each of its dependencies `a : Arity b` to the sub-diagram `f a`.

The key theorem is that every wiring diagram, together with implementations for all of its boxes, composes to a single implementation of the outer interface:

Theorem 3.1. *Given a wiring diagram with output interface p and external inputs* inputs*, together with an implementation of each box b as a morphism* $\mathsf{Cod}(b) \Rightarrow \mathsf{Free}(\mathsf{sum}\,(\mathsf{Arity}\,b)\,(\mathsf{Dom}\,b))$*, there is an induced implementation* $p \Rightarrow \mathsf{Free}(\mathsf{sum}\,\mathsf{Inputs}\,\mathsf{inputs})$*.*

The proof is by induction on the wiring diagram: for a `wire i`, the external input is forwarded; for a `box b f`, we compose the implementation of box `b` with the recursively composed sub-diagrams:

```
compose : Wiring Boxes Arity Dom Cod Inputs inputs output
        → ((b : Boxes) → Cod b ⇒ Free (sum (Arity b) (Dom b)))
        → output ⇒ Free (sum Inputs inputs)
compose (wire i) f x = bind (i , x) return
compose (box b f) g = comp _ (g b) (λ a → compose (f a) g)
```

As a concrete illustration, Figure 1 shows the wiring diagrams for `append` and `concat`.

4 Coalgebraic Operational Semantics

4.1 Mealy Machines

To actually *run* programs built up in the manner described above, we need an *operational semantics*. For this purpose, we develop a *coalgebraic* approach based on *Mealy machines* (Niu and Spivak 2024, Ch. 4; Myers 2022):

Definition 4.1. A *Mealy machine* with interface $p = (A, B)$ is an element of the following coinductive record type:

```
record Mealy (p : Poly) : Set where
    coinductive
```

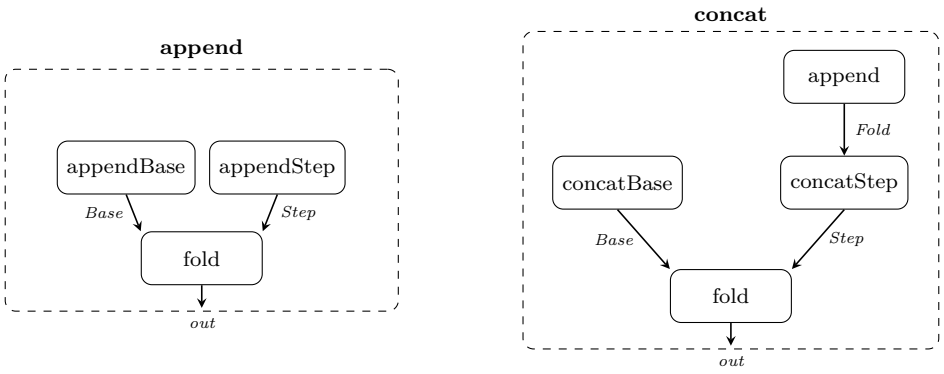

Figure 1: Wiring diagrams for `append` (left) and `concat` (right). In `concat`, the step box's dependency is wired through the `append` module.

```
    field
        app : (x : p .fst) → p .snd x × Mealy p
```

In other words, a Mealy machine with interface (A, B) is a stateful automaton that, given an input $a : A$, produces an output of type $B(a)$ together with an updated machine state. The type $\mathsf{Mealy}\, p$ is equivalently the *final coalgebra* of the functor $X \mapsto \prod_{a:A} B(a) \times X$.

Given a Mealy machine with interface p, we can *run* a computation $e : \mathsf{Free}\, p\, C$ on it by threading the machine state through each call, yielding a result and an updated machine. More generally, given an implementation $f : p \Rightarrow \mathsf{Free}\, q$ and a Mealy machine for q, we obtain a Mealy machine for p:

```
prog→mealy : {p q : Poly}
           → (f : p ⇒ Free q) → Mealy q → Mealy p
prog→mealy f m .app a =
    let (b , m') = run-mealy (f a) m in
    (b , prog→mealy f m')
```

Moreover, Mealy machines are closed under the coproduct of polynomials, so a pair of Mealy machines for p and q can be combined into one for $p \oplus q$, as follows:

```
_mealy⊕_ : {p q : Poly}
         → Mealy p → Mealy q → Mealy (p ⊕ q)
_mealy⊕_ m₁ m₂ .app (inl a) =
    let (b , m') = m₁ .app a in (b , (m' mealy⊕ m₂))
_mealy⊕_ m₁ m₂ .app (inr c) =
    let (d , m') = m₂ .app c in (d , (m₁ mealy⊕ m'))
```

Together, `prog→mealy` and `_mealy⊕_` give a functorial operational semantics: given Mealy machines m_1, m_2 with interfaces p and q, and a program module $f : r \Rightarrow \mathsf{Free}\,(p \oplus q)$ specifying a pattern of interaction between them, we obtain a Mealy machine for the interface r that executes the specified interactions. This approach is closely related to the module structure of the free monad monad over the cofree comonad comonad described in Libkind and Spivak's "Pattern Runs on Matter" (Libkind and Spivak 2025). Details are given in Appendix B.

4.2 Effects and Runners

The fact that Mealy machines represent *stateful* computations, yet can be composed along program modules, means that the functions specified by a polynomial interface need not be *pure*; in particular, they can have *effects* that are implemented by the Mealy machine.

A polynomial morphism $f : p \Rightarrow \mathsf{Free}\, q$ can thus be seen as using *effects* encoded by q, and a Mealy machine for q then corresponds to a *runner* for these effects, as defined by Ahman and Bauer (Ahman and Bauer 2020). As a concrete example, the *state effect* for a type S is encoded by the following interface:

```
State : Set → Poly
State S .fst = StateI S
State S .snd get = S
State S .snd (put s) = ⊤
```

where $\mathsf{StateI}\, S$ has constructors get and $\mathsf{put}\, s$ for $s : S$, corresponding to operations for reading from and writing to the state, respectively. A Mealy machine implementing this interface simply threads the current state:

```
state-mealy : {S : Set} → S → Mealy (State S)
state-mealy s .app get = (s , state-mealy s)
state-mealy s .app (put s') = (_ , state-mealy s')
```

4.3 Example: Fibonacci

Using the state effect, we implement a Fibonacci Mealy machine. The update reads the current pair of Fibonacci numbers from state, advances it, and returns the first component:

```
fib-update : (⊤ , λ _ → Nat) ⇒ Free (State (Nat × Nat))
fib-update _ =
    call[ (x , y) ← get ]
    call[ _ ← put (y , x + y) ]
    return x

fib : Mealy (⊤ , λ _ → Nat)
fib = prog→mealy fib-update (state-mealy (0 , 1))
```

The machine `fib` is obtained by composing the update program with the state runner initialized to $(0,1)$. On each invocation, it outputs $0, 1, 1, 2, 3, 5, \ldots$ in sequence.

5 Dependent Polynomials and Compositional Verification

We now come to the heart of the paper: representing *specifications* of program modules in a way that supports compositional verification.

5.1 Dependent Polynomials as Specifications

Definition 5.1. A *dependent polynomial* over a polynomial $p = (A, B)$ consists of:

- a family of *preconditions* $C : A \to \mathsf{Set}$ on inputs $a : A$, and
- a family of *postconditions* $D : (a : A) \to C(a) \to B(a) \to \mathsf{Set}$ on outputs $b : B(a)$, depending on $a : A$ and evidence $c : C(a)$ that the precondition holds.

```
DepPoly : Poly → Set₁
DepPoly (A , B) = Σ (A → Set) (λ C → (x : A) → C x → B x → Set)
```

If $p = (A, B)$ represents the interface of a function $f : (a : A) \to B(a)$, then a dependent polynomial (C, D) over p specifies a *contract*: for each input a satisfying precondition $C(a)$, the output $f(a)$ should satisfy postcondition $D(a, c, f(a))$, where c is the evidence that the precondition holds. This is the polynomial-functorial analogue of a *Hoare triple* $\{C(a)\}\, f(a)\, \{D(a, -, -)\}$.

Categorically, a dependent polynomial over p is equivalently a polynomial functor on the arrow category $\mathbf{Type}^{\to}$ lying over p via the codomain functor $\mathsf{cod} : \mathbf{Type}^{\to} \to \mathbf{Type}$.

A *dependent morphism* $\Rightarrow \mathsf{Dep}\, p\, r\, G\, f$ over a morphism $f : p \Rightarrow F$—where F is an endofunctor on types and G is a *dependent* endofunctor, i.e. an endofunctor on $\mathbf{Type}^{\to}$ lying over F via the codomain map—is a function that, for each input a and evidence $c : C(a)$ that the precondition holds, produces a G-structure witnessing that the postcondition will be satisfied on the output $f(a)$.

```
⇒Dep : (p : Poly) → DepPoly p → {F : Set → Set}
     → ((X : Set) → (X → Set) → F X → Set)
     → (f : p ⇒ F) → Set
⇒Dep (A , B) (C , D) G f =
    (a : A) (c : C a) → G (B a) (D a c) (f a)
```

5.2 Dependent Free Monads

To verify implementations $f : p \Rightarrow \mathsf{Free}\, q$, we need a notion of *verified computation* over the free monad. This is provided by the *dependent free monad*, which is equivalently the free monad monad on the category of dependent polynomials:

```
data FreeDep (p : Poly) (r : DepPoly p)
             (E : Set) (F : E → Set) : Free p E → Set where
    returnD : {e : E} → F e → FreeDep p r E F (return e)
    bindD : {a : p .fst} (c : r .fst a)
          → {k : (p .snd a) → Free p E}
          → ((b : p .snd a) → r .snd a c b
                            → FreeDep p r E F (k b))
          → FreeDep p r E F (bind a k)
```

An element of $\mathsf{FreeDep}\, p\, r\, E\, F\, e$ proves that computation e respects specification r: assuming each call's response satisfies the postcondition, the final result satisfies F—this is *assume-guarantee* reasoning now lifted to the realm of dependent types.

Definition 5.2. A *verified implementation* of specification (p, r) depending on specification (q, s) consists of an implementation $f : p \Rightarrow \mathsf{Free}\, q$ together with a dependent morphism of type

$$\Rightarrow \mathsf{Dep}\, p\, r\, (\mathsf{FreeDep}\, q\, s)\, f$$

witnessing that f satisfies specification r assuming q satisfies specification s.

The crucial compositionality theorem is then that verified implementations compose in the same manner as implementations:

Theorem 5.3. *Given a wiring diagram with implementations for each box, and specifications for the interfaces of each box and of the outer box as a whole, along with verifications for each box's specification, the composite implementation carries a composite verification for the specifications of the outer interfaces. That is, verifications compose along wiring diagrams in the same way that their underlying implementations do.*

The proof uses the dependent analogue ∘Dep of Kleisli composition for the dependent free monad monad, along with the fact that dependent polynomials are closed under sums:

```
-- sums of dependent polynomials
sumDep : (U : Set) (p : U → Poly)
       → (r : (u : U) → DepPoly (p u)) → DepPoly (sum U p)
sumDep U p r .fst (u , x) = r u .fst x
sumDep U p r .snd (u , x) y = r u .snd x y
```

The multi-ary composition `compDep` then mirrors `comp`.

```
compDep : (U : Set) {p : Poly} {q : U → Poly} {r : Poly}
          {s : DepPoly p} {t : (u : U) → DepPoly (q u)}
          {v : DepPoly r} {f : p ⇒ Free (sum U q)}
        → {g : (u : U) → q u ⇒ Free r}
        → (⇒Dep p s (FreeDep (sum U q) (sumDep U q t)) f)
        → ((u : U) → ⇒Dep (q u) (t u) (FreeDep r v) (g u))
        → ⇒Dep p s (FreeDep r v) (comp U f g)
compDep U ff gg = ∘Dep ff (λ (u , x) c → gg u x c)
```

As an example (Appendix C), we verify that our `append` implementation correctly computes list-append. The key idea is that proofs by *induction*—or more specifically using *loop invariants*—on recursive functions on lists are simply the dependent analogue of folds on lists: we define a generic verification `foldInd` over the `fold` module parameterized by specifications for the base and step. To verify `append`, we instantiate this module with a relational specification, and use `compDep` to compose the base and step verifications with `foldInd`, yielding a complete verification of the append module.

6 Verified Operational Semantics

6.1 Dependent Mealy Machines

Just as Mealy machines provide an operational semantics for program modules, *dependent Mealy machines* provide a notion of *specification/verification* for these operational semantics:

Definition 6.1. A *dependent Mealy machine* for specification (p, r) over a Mealy machine m for p is a coinductive record:

```
record DepMealy (p : Poly) (r : DepPoly p) (m : Mealy p) : Set where
    coinductive
    field
        appD : (x : p .fst) (y : r .fst x)
             → r .snd x y (m .app x .fst)
               × DepMealy p r (m .app x .snd)
```

The key compositionality results carry over to the dependent setting:

Proposition 6.2. *Given a verified implementation $f : p \Rightarrow Freeq$ with respect to specifications r on p and s on q, and a dependent Mealy machine for (q, s) over m, the induced Mealy machine* $\mathsf{prog} \to \mathsf{mealy}\, f\, m$ *carries a dependent Mealy machine for (p, r).*

The proof uses `run-mealyD`, which threads verification evidence alongside the machine state:

```
prog→mealyD : {p q : Poly} {r : DepPoly p} {s : DepPoly q}
            → {f : p ⇒ Free q} {m : Mealy q}
            → (⇒Dep p r (FreeDep q s) f) → DepMealy q s m
            → DepMealy p r (prog→mealy f m)
prog→mealyD ff mm .appD a c =
    let (b , mm') = run-mealyD (ff a c) mm in
    (b , (prog→mealyD ff mm'))
```

Proposition 6.3. *Dependent Mealy machines are closed under binary sums: given a dependent Mealy machine for (p, r) over m_1 and one for (q, s) over m_2, there is a dependent Mealy machine for $(p \oplus q, r \oplus s)$ over $m_1 \oplus m_2$.*

```
_depMealy⊕_ : {p q : Poly} {r : DepPoly p} {s : DepPoly q}
            → {m₁ : Mealy p} {m₂ : Mealy q}
            → DepMealy p r m₁ → DepMealy q s m₂
            → DepMealy (p ⊕ q) (depPoly⊕ r s) (m₁ mealy⊕ m₂)
_depMealy⊕_ m₁ m₂ .appD (inl a) c =
    let (b , m') = m₁ .appD a c in
    (b , (m' depMealy⊕ m₂))
_depMealy⊕_ m₁ m₂ .appD (inr c) d =
    let (e , m') = m₂ .appD c d in
    (e , (m₁ depMealy⊕ m'))
```

6.2 State Invariants

A natural application is verifying *state invariants*. Given $T : S \to \mathsf{Set}$, we specify that if the initial state satisfies T and all written states satisfy T, then all read states satisfy T:

```
-- specification for state invariants
Invariant : (S : Set) (T : S → Set) → DepPoly (State S)
Invariant S T .fst get = ⊤
Invariant S T .fst (put s) = T s
Invariant S T .snd get _ s = T s
Invariant S T .snd (put s) t _ = ⊤
```

```
-- verification of a state invariant
invariant : {S : Set} {T : S → Set}
          → (s : S) → T s
          → DepMealy (State S) (Invariant S T) (state-mealy s)
invariant s t .appD get _ = (t , (invariant s t))
invariant s t .appD (put s') t' = (_ , invariant s' t')
```

The verification is a straightforward coinductive argument.

6.3 Example: Verified Fibonacci

We can verify that the `fib` Mealy machine computes Fibonacci numbers by defining a relational specification `IsFib` and a state invariant asserting that the state (y, z) satisfies $\mathsf{IsFib}(x, y) \times \mathsf{IsFib}(x+1, z)$ for some index x. The verification of `fib-update` shows that each update step preserves this invariant and returns the correct Fibonacci number. Composing with the state invariant verification via `prog→mealyD` yields a full dependent Mealy machine for the specification. The full code is given in Appendix C.

7 Categorical Perspective

Stepping back from the concrete constructions, we can identify the abstract categorical structure that makes the above-described compositional verification work in general.

Let **Int** (the category of *interfaces*) be the category whose objects are polynomial functors and whose morphisms $p \to q$ are implementations $p \Rightarrow \mathsf{Free}\, q$, composed via Kleisli composition. Let **Spec** (the category of *specifications*) be the category whose objects are pairs (p, r) of a polynomial and a dependent polynomial, and whose morphisms $(p, r) \to (q, s)$ are pairs $(f, \hat{f})$ of an implementation and a verification of that implementation.

Both categories carry monoidal structures given by $\oplus$, and the forgetful functor $\pi : \mathbf{Spec} \to \mathbf{Int}$ is (strictly) monoidal; this is precisely why verified implementations compose compatibly with wiring diagrams.

The assignment $p \mapsto \mathsf{Mealy}\, p$ then defines a *presheaf* on **Int**, i.e. a contravariant functor $\mathsf{Mealy} : \mathbf{Int}^{\mathrm{op}} \to \mathbf{Set}$, with the functorial action given by `prog→mealy`. Moreover, this presheaf is *lax monoidal*, since the operations `mealy⊕` and `mealyzero` provide natural maps

$$\mathsf{Mealy}(p) \times \mathsf{Mealy}(q) \to \mathsf{Mealy}(p \oplus q) \qquad \text{and} \qquad 1 \to \mathsf{Mealy}(\mathsf{zeroPoly})$$

satisfying the coherence conditions of a lax monoidal functor.

Similarly, DepMealy defines a lax monoidal presheaf on $\mathbf{Spec}^{\mathrm{op}}$ that maps a dependent polynomial (p, r) to the set of Mealy machines with interface p satisfying the specification r, and the projection map $\mathsf{DepMealy}(p, r) \to \mathsf{Mealy}(p)$ is then a monoidal natural transformation $\mathsf{DepMealy} \Rightarrow \mathsf{Mealy} \circ \pi$.

The most general abstract setting for compositional verification is then as follows:

Definition 7.1. A *compositional verification framework* consists of: (i) a monoidal category $(\mathbf{Int}, \oplus, 0)$ with a lax monoidal presheaf $\mathsf{Sys} : \mathbf{Int}^{\mathrm{op}} \to \mathbf{Set}$ of *systems*; (ii) a monoidal category

$(\mathbf{Spec}, \oplus, 0)$ with a monoidal functor $\pi : \mathbf{Spec} \to \mathbf{Int}$ and a lax monoidal presheaf $\mathsf{Verif} : \mathbf{Spec}^{\mathrm{op}} \to \mathbf{Set}$ of *verified systems*; (iii) a monoidal natural transformation $\mathsf{Verif} \Rightarrow \mathsf{Sys} \circ \pi$.

Our concrete development instantiates this with **Int** as the Kleisli category of the free monad on Poly, **Spec** as the corresponding category of dependent polynomials, $\mathsf{Sys} = \mathsf{Mealy}$, and $\mathsf{Verif} = \mathsf{DepMealy}$. The value of the abstraction is that it identifies precisely what structure is needed for compositional verification, and opens the door to other instantiations. In particular, the sequential modules considered so far can be extended to *concurrent* modules by replacing the coproduct $\oplus$ with a *parallel sum* $p \| q$ that allows calling either or both interfaces simultaneously. This yields a second monoidal structure on **Int**. Details are given in Appendix A.

8 Related Work

The theory of polynomial functors as a framework for interaction has been developed by Spivak and collaborators (Niu and Spivak 2024; Spivak 2022; Spivak and Tan 2017), and Libkind and Spivak (Libkind and Spivak 2025) study the free monad–cofree comonad duality (see Appendix B). The connection of polynomial functors to dependent type theory has been explored through *containers* (Abbott et al. 2005) and W-types (Gambino and Hyland 2004); our dependent polynomials extend this with a specification layer.

9 Conclusion and Future Work

We have presented a framework for compositional program verification based on polynomial functors in dependent type theory, providing a uniform treatment of interfaces, implementations, specifications, and operational semantics, all composing along wiring diagrams, and unified under an account of the abstract categorical structure behind such compositionality.

Several directions for future work suggest themselves. First, the compositional nature of the framework makes it highly amenable to *automation*: the composition operations could be implemented as tactics or elaboration procedures in a proof assistant, reducing the burden on the user to supply only the local verifications for individual modules. Second, while our formalization is in Agda, the general principles should be applicable in other dependently typed languages such as Lean, Idris, or Coq, and we are interested in exploring such implementations. Third, wiring diagrams make *information flow* explicit in a program architecture, and we are interested in applying this to the verification of security properties and information flow policies. In a world where digital systems are increasingly built from opaque components—API calls, machine learning models—techniques for formally guaranteeing their safety through compositional, interface-level reasoning become ever more valuable.

9.1 Acknowledgements

I am grateful to my colleagues at the Topos Institute during my Summers there as a Research Assistant for many stimulating discussions that ultimately led to the development of this framework. In particular, conversations with David Spivak, David Jaz Myers, Evan Patterson, and Kevin Carlson were instrumental in shaping the ideas presented here, and my conversations with Dana Scott about Agda likewise inspired the formalization presented in this paper.

References

Abbott, Michael, Thorsten Altenkirch, and Neil Ghani. 2005. "Containers: Constructing Strictly Positive Types." *Theoretical Computer Science* 342 (1): 3–27. https://doi.org/10.1016/j.tcs.2005.06.002.

Ahman, Danel, and Andrej Bauer. 2020. "Runners in Action." *Programming Languages and Systems (ESOP 2020)*, Lecture notes in computer science, vol. 12075: 29–55. https://doi.org/10.1007/978-3-030-44914-8_2.

Gambino, Nicola, and Martin Hyland. 2004. "Wellfounded Trees and Dependent Polynomial Functors." *Types for Proofs and Programs*, Lecture notes in computer science, vol. 3085: 210–25. https://doi.org/10.1007/978-3-540-24849-1_14.

Libkind, Sophie, and David I. Spivak. 2025. "Pattern Runs on Matter: The Free Monad Monad as a Module over the Cofree Comonad Comonad." *International Conference on Applied Category Theory (ACT 2024)*, Electronic proceedings in theoretical computer science, vol. 429: 1–28. https://doi.org/10.4204/EPTCS.429.1.

Myers, David Jaz. 2022. *Categorical Systems Theory*.

Niu, Nelson, and David I. Spivak. 2024. *Polynomial Functors: A Mathematical Theory of Interaction*. https://arxiv.org/abs/2312.00990.

Norell, Ulf. 2009. "Dependently Typed Programming in Agda." In *Advanced Functional Programming (AFP 2008)*, vol. 5832. Lecture Notes in Computer Science. Springer. https://doi.org/10.1007/978-3-642-04652-0_5.

Spivak, David I. 2013. "The Operad of Wiring Diagrams: Formalizing a Graphical Language for Databases, Recursion, and Plug-and-Play Circuits." *arXiv Preprint arXiv:1305.0297*. https://arxiv.org/abs/1305.0297.

Spivak, David I. 2022. *Polynomial Functors and Shannon Entropy*. https://arxiv.org/abs/2201.12878.

Spivak, David I., and Joshua Tan. 2017. "Nesting of Dynamical Systems and Mode-Dependent Networks." *Journal of Complex Networks* 5 (3): 389–408. https://doi.org/10.1093/comnet/cnw022.

A Concurrency and Relational Verification

So far, the program modules we have considered have been *sequential*, in that they may only call one of their dependencies at a time. However, our operational semantics in terms of Mealy machines does not require this, and we can extend our framework to allow for *concurrent* program modules that may call multiple dependencies at once.

To this end, we make use of the *conjunctive sum* of types, which is defined as $A \vee B = A + B + (A \times B)$, representing the possibility of either an A, a B, or both at the same time. In Agda, we can define this as follows:

```
data _∨_ (A B : Set) : Set where
    left : A → A ∨ B
    right : B → A ∨ B
    both : A → B → A ∨ B
```

Using this, we define the *parallel sum* of polynomials as follows:

```
_||_ : Poly → Poly → Poly
_||_ (A , B) (C , D) .fst = A ∨ C
_||_ (A , B) (C , D) .snd (left a) = B a
_||_ (A , B) (C , D) .snd (right c) = D c
_||_ (A , B) (C , D) .snd (both a c) = B a × D c
```

Thinking of this as an interface to a pair of functions/program modules, what this allows is for us to call either one or both of the functions/modules at once, and to obtain a corresponding result of the appropriate type.

The parallel sum of dependent polynomials is defined similarly:

```
_||Dep_ : {p q : Poly} → DepPoly p → DepPoly q → DepPoly (p || q)
_||Dep_ (A , B) (C , D) .fst (left a) = A a
_||Dep_ (A , B) (C , D) .fst (right c) = C c
_||Dep_ (A , B) (C , D) .fst (both a c) = A a × C c
_||Dep_ (A , B) (C , D) .snd (left a) y = B a y
_||Dep_ (A , B) (C , D) .snd (right c) y = D c y
_||Dep_ (A , B) (C , D) .snd (both a c) (y , z) (x , w) =
    (B a y x) × (D c z w)
```

The parallel sum is in fact a (symmetric) monoidal product on the category of polynomials, with unit `zeroPoly`. We can exhibit this by defining the following operations that allow us to put program modules *in parallel* using the parallel sum:

```
-- injections into the parallel sum
leftProg : {p q : Poly} {E : Set} → Free p E → Free (p || q) E
leftProg (return e) = return e
leftProg (bind a k) = bind (left a) (λ b → leftProg (k b))

rightProg : {p q : Poly} {F : Set} → Free q F → Free (p || q) F
rightProg (return f) = return f
rightProg (bind c h) = bind (right c) (λ d → rightProg (h d))

-- parallel composition of programs
bothProg : {p q : Poly} {E F : Set}
         → Free p E → Free q F → Free (p || q) (E × F)
bothProg (return e) (return f) = return (e , f)
bothProg (bind a k) (return f) =
```

```
    bind (left a) (λ b → bothProg (k b) (return f))
bothProg (return e) (bind c h) =
    bind (right c) (λ d → bothProg (return e) (h d))
bothProg (bind a k) (bind c h) =
    bind (both a c) (λ (b , d) → bothProg (k b) (h d))

-- parallel composition of program modules
_||Prog_ : {p q r s : Poly} → (p ⇒ Free r) → (q ⇒ Free s)
         → ((p || q) ⇒ Free (r || s))
_||Prog_ f g (left a) = leftProg (f a)
_||Prog_ f g (right c) = rightProg (g c)
_||Prog_ f g (both a c) = bothProg (f a) (g c)
```

And similarly for dependent polynomial morphisms/verifications:

```
-- injections into the parallel sum of dependent polynomials
leftDep : {p q : Poly} {r : DepPoly p} {s : DepPoly q}
        → {X : Set} {Y : X → Set}
        → {f : Free p X}
        → FreeDep p r X Y f
        → FreeDep (p || q) (r ||Dep s) X Y (leftProg f)
leftDep (returnD f) = returnD f
leftDep (bindD c h) = bindD c (λ d e → leftDep (h d e))

rightDep : {p q : Poly} {r : DepPoly p} {s : DepPoly q}
         → {X : Set} {Y : X → Set}
         → {g : Free q X}
         → FreeDep q s X Y g
         → FreeDep (p || q) (r ||Dep s) X Y (rightProg g)
rightDep (returnD f) = returnD f
rightDep (bindD c h) = bindD c (λ d e → rightDep (h d e))

-- parallel composition of verifications
bothDep : {p q : Poly} {r : DepPoly p} {s : DepPoly q}
        → {X Y : Set} {Z : X → Set} {W : Y → Set}
        → {f : Free p X} {g : Free q Y}
        → FreeDep p r X Z f → FreeDep q s Y W g
        → FreeDep (p || q) (r ||Dep s) (X × Y)
                  (λ (x , y) → Z x × W y) (bothProg f g)
bothDep (returnD z) (returnD w) = returnD (z , w)
bothDep (bindD c h) (returnD w) =
    bindD c (λ b x → bothDep (h b x) (returnD w))
bothDep (returnD z) (bindD d k) =
    bindD d (λ b x → bothDep (returnD z) (k b x))
bothDep (bindD c h) (bindD d k) =
    bindD (c , d) (λ (a , b) (x , y) → bothDep (h a x) (k b y))
```

```
-- parallel composition of dependent polynomial morphisms
_||DepProg_ : {p q r s : Poly}
            → {p' : DepPoly p} {q' : DepPoly q}
            → {r' : DepPoly r} {s' : DepPoly s}
            → {f : p ⇒ Free r} {g : q ⇒ Free s}
            → ⇒Dep p p' (FreeDep r r') f → ⇒Dep q q' (FreeDep s s') g
            → ⇒Dep (p || q) (p' ||Dep q')
                   (FreeDep (r || s) (r' ||Dep s')) (f ||Prog g)
_||DepProg_ ff gg (left a) b = leftDep (ff a b)
_||DepProg_ ff gg (right c) d = rightDep (gg c d)
_||DepProg_ ff gg (both a c) (b , d) = bothDep (ff a b) (gg c d)
```

Hence, since our generalized account of compositional verification is based on the existence of a monoidal structure on the category of interfaces and the category of specifications, we can apply this same framework for compositional verification of concurrent program modules, by simply using the parallel sum as the monoidal structure instead of the coproduct.

To complete the picture, we also show that Mealy machines and their verifications are closed under the parallel sum, allowing for Mealy machines to genuinely be run in parallel:

```
-- parallel sum of Mealy machines
_mealy||_ : {p q : Poly} → Mealy p → Mealy q → Mealy (p || q)
_mealy||_ m₁ m₂ .app (left a) =
    let (b , m') = m₁ .app a in
    (b , m' mealy|| m₂)
_mealy||_ m₁ m₂ .app (right c) =
    let (d , m') = m₂ .app c in
    (d , m₁ mealy|| m')
_mealy||_ m₁ m₂ .app (both a c) =
    let ((b , m₁') , (d , m₂')) = (m₁ .app a , m₂ .app c) in
    ((b , d) , m₁' mealy|| m₂')

-- parallel sum of dependent Mealy machines
_depMealy||_ : {p q : Poly} {r : DepPoly p} {s : DepPoly q}
             → {m₁ : Mealy p} {m₂ : Mealy q}
             → DepMealy p r m₁ → DepMealy q s m₂
             → DepMealy (p || q) (r ||Dep s) (m₁ mealy|| m₂)
_depMealy||_ m₁ m₂ .appD (left a) c =
    let (b , m') = m₁ .appD a c in
    (b , (m' depMealy|| m₂))
_depMealy||_ m₁ m₂ .appD (right c) d =
    let (b , m') = m₂ .appD c d in
    (b , (m₁ depMealy|| m'))
_depMealy||_ m₁ m₂ .appD (both a c) (b , d) =
    let ((e , m₁') , (f , m₂')) = (m₁ .appD a b , m₂ .appD c d) in
    ((e , f) , (m₁' depMealy|| m₂'))
```

Hence we have another instance of our general framework for compositional verification, this

time for concurrent program modules and their operational semantics via parallel composition of Mealy machines.

A.1 Affine vs. Cocartesian Structure and Race-Freedom

An important distinction between the coproduct $\oplus$ and the parallel sum $\|$ is that $\oplus$ is *cocartesian*—it admits a codiagonal $p \oplus p \to p$—while $\|$ is only *affine symmetric monoidal* (it admits weakening but not contraction, in the jargon of linear logic/sequent calculus). Concretely, this means that in a parallel wiring diagram, a wire cannot be used in two places simultaneously (though it can remain unused, i.e. it can be left dangling). In terms of concurrency, this suffices to prevent *race conditions* with respect to the effects encoded by polynomial interfaces: by design, concurrent programs structured via the parallel sum cannot invoke the same effect in two places simultaneously, and hence there is no ambiguity or nondeterminism as to whether one such invocation will resolve before another. In other words, any effect/interface that is used in such a wiring diagram is only ever *in one place* at a time.

A.2 Relational Specifications

A key feature of the parallel sum is that it enables *relational verification*. Observe that $p \| q \cong p \oplus q \oplus (p \otimes q)$, where $\otimes$ is the *parallel product*:

```
_⊗_ : Poly → Poly → Poly
(A , B) ⊗ (C , D) = (A × C , λ (a , c) → B a × D c)
```

A dependent polynomial over $p \| q$ therefore decomposes into three parts: a specification for p alone, a specification for q alone, and a *relational* specification over $p \otimes q$ whose precondition may relate the inputs to p and q, and whose postcondition may relate their outputs. This enables verification of properties such as *monotonicity* (if the input to one module increases, the output of another increases correspondingly), *noninterference* (the output of one module is independent of the input to another), and other relational properties that cannot be expressed as unary pre/postconditions.

B Connection to "Pattern Runs on Matter"

The construction `prog→mealy` in the paper is closely related to the "pattern runs on matter" framework of Libkind and Spivak (Libkind and Spivak 2025), specifically:

$\mathsf{Free}\, p\, C$ corresponds to $\mathfrak{m}_p$, the free monad on polynomial p (a "pattern" or decision tree). The type $\mathsf{Mealy}\, q$ corresponds to $\mathfrak{c}_{[q,y]}$, the cofree comonad on the internal hom $[q, y]$ in **Poly**, viewed as a monoidal category with the Dirichlet product $\otimes$, where y is the monoidal unit.

Given $f : p \to \mathfrak{m}_q$, the derivation proceeds as:

$$p \otimes \mathfrak{c}_{[q,y]} \xrightarrow{f \otimes \mathrm{id}} \mathfrak{m}_q \otimes \mathfrak{c}_{[q,y]} \xrightarrow{\Xi_{q,[q,y]}} \mathfrak{m}_{q \otimes [q,y]} \xrightarrow{\mathfrak{m}_{app}} \mathfrak{m}_y \to y$$

where Ξ is the module structure map and $app : q \otimes [q, y] \to y$ is the evaluation map. Transposing gives $\mathfrak{c}_{[q,y]} \to [p, y]$, and applying the co-Kleisli lift for $\mathfrak{c}$, we obtain $\mathfrak{c}_{[q,y]} \to \mathfrak{c}_{[p,y]}$, whose forward part is precisely `prog→mealy` f.

C Extended Examples

C.1 Verified Fold and Append

The verification of the `fold` module captures the general pattern of proofs by *induction*. Given a specification D on the accumulator and a postcondition E relating accumulator, input list, and result, the verification `foldInd` proceeds by structural induction, mirroring the recursive structure of `fold`:

```
FoldSpec : (A B C : Set) (D : A → Set)
         → (E : (a : A) → D a → List B → C → Set)
         → DepPoly (Fold A B C)
FoldSpec A B C D E .fst (a , _) = D a
FoldSpec A B C D E .snd (a , bs) d c = E a d bs c

BaseSpec : (A B C : Set) (D : A → Set)
         → (E : (a : A) → D a → List B → C → Set)
         → DepPoly (Base A C)
BaseSpec A B C D E .fst a = D a
BaseSpec A B C D E .snd a d c = E a d nil c

StepSpec : (A B C : Set) (D : A → Set)
         → (E : (a : A) → D a → List B → C → Set)
         → DepPoly (Step A B C)
StepSpec A B C D E .fst (a , _ , c) =
    Σ (D a) (λ d → Σ (List B) (λ bs → E a d bs c))
StepSpec A B C D E .snd (a , b , _) (d , bs , _) c =
    E a d (cons b bs) c

foldInd : {A B C : Set} {D : A → Set}
        → {E : (a : A) → D a → List B → C → Set}
        → ⇒Dep _ (FoldSpec A B C D E)
               (FreeDep _ (sumDep foldlabels _
                          (λ{ base → BaseSpec A B C D E
                            ; step → StepSpec A B C D E})))
               fold
foldInd (_ , nil) d =
    bindD d (λ _ e → returnD e)
foldInd (a , cons b bs) d =
    >>=Dep _ (foldInd (a , bs) d) (λ _ e →
        bindD (d , (bs , e)) (λ _ e' →
            returnD e'))
```

For append, we define a relational specification and verify the base and step:

```
data Append {A : Set} : List A → ⊤ → List A → List A → Set where
    append-nil : {xs : List A} → Append xs _ nil xs
    append-cons : {xs ys zs : List A} {x : A}
```

```
                → Append ys _ xs zs
                → Append ys _ (cons x xs) (cons x zs)

appendBaseSpec : {A : Set}
               → ⇒Dep _ (BaseSpec (List A) A (List A) (λ _ → ⊤) Append)
                     (FreeDep _ depPolyzero) appendBase
appendBaseSpec _ _ = returnD append-nil

appendStepSpec : {A : Set}
               → ⇒Dep _ (StepSpec (List A) A (List A) (λ _ → ⊤) Append)
                     (FreeDep _ depPolyzero) appendStep
appendStepSpec (_ , _ , _) (_ , (_ , e)) = returnD (append-cons e)

appendSpec : {A : Set}
           → ⇒Dep _ (FoldSpec (List A) A (List A) (λ _ → ⊤) Append)
                (FreeDep _ depPolyzero) append
appendSpec =
    compDep _ foldInd
            (λ{ base → appendBaseSpec
              ; step → appendStepSpec})
```

C.2 Verified Fibonacci

The verification that `fib` computes Fibonacci numbers uses a relational specification and a state invariant:

```
data IsFib : Nat → Nat → Set where
    isFibZero : IsFib 0 0
    isFibOne : IsFib 1 1
    isFibRec : {x y z : Nat}
             → IsFib x y → IsFib (suc x) z
             → IsFib (suc (suc x)) (plus y z)

FibDep : DepPoly (⊤ , λ _ → Nat)
FibDep .fst _ = ⊤
FibDep .snd _ _ y = Σ Nat (λ x → IsFib x y)

FibInvariant : (Nat × Nat) → Set
FibInvariant (y , z) =
    Σ Nat (λ x → IsFib x y × IsFib (suc x) z)

fib-updateSpec :
    ⇒Dep _ FibDep
         (FreeDep _ (Invariant _ FibInvariant))
         fib-update
fib-updateSpec _ _ =
```

```
    bindD _ (λ _ (x , fy , fz) →
        bindD (suc x , fz , isFibRec fy fz) (λ _ _ →
            returnD (_ , fy)))

fibSpec : DepMealy _ FibDep fib
fibSpec =
    prog→mealyD fib-updateSpec
                (invariant _ (zero , isFibZero , isFibOne))
```

D Full Agda Code

```
{-# OPTIONS --without-K --guardedness #-}
module cpv where

open import Agda.Builtin.Unit
open import Agda.Builtin.Sigma

-- binary products
_×_ : Set → Set → Set
A × B = Σ A (λ _ → B)

infixr 5 _×_

-- binary coproducts
data _+_ (A : Set) (B : Set) : Set where
    inl : A → A + B
    inr : B → A + B

-- empty type
data ⊥ : Set where

-- polynomial functors
Poly : Set₁
Poly = Σ Set (λ A → A → Set)

-- action of a polynomial on a set
_⊙_ : Poly → (Set → Set)
(A , B) ⊙ y = Σ A (λ x → B x → y)

-- polynomial morphisms
_⇒_ : Poly → (Set → Set) → Set
(A , B) ⇒ F = (x : A) → F (B x)

-- free monad on a polynomial
data Free (p : Poly) (C : Set) : Set where
```

```
    return : C → Free p C
    bind : (x : p .fst) → (p .snd x → Free p C) → Free p C

-- monadic bind/Kleisli lift for Free monads
_>>=_ : {p : Poly} {A B : Set}
      → Free p A → (A → Free p B) → Free p B
(return a) >>= f = f a
(bind x k) >>= f = bind x (λ b → k b >>= f)

syntax bind a (λ b → e) = call[ b ← a ] e

>>=-syntax = _>>=_

syntax >>=-syntax m (λ b → e) = do[ b ← m ] e

-- sums of polynomials
sum : (U : Set) → (p : U → Poly) → Poly
sum U p .fst = Σ U (λ u → p u .fst)
sum U p .snd (u , x) = p u .snd x

-- binary sum
_⊕_ : Poly → Poly → Poly
((A , B) ⊕ (C , D)) .fst = A + C
((A , B) ⊕ (C , D)) .snd (inl a) = B a
((A , B) ⊕ (C , D)) .snd (inr c) = D c

-- nullary sum
zeroPoly : Poly
zeroPoly .fst = ⊥
zeroPoly .snd ()

-- examples: fold, append, concat
module example1 where

    -- simple list type
    data List (A : Set) : Set where
        nil  : List A
        cons : A → List A → List A

    -- labels for fold dependencies
    data foldlabels : Set where
        base : foldlabels
        step : foldlabels

    -- interface for base case of a fold
    Base : (A C : Set) → Poly
```

```
Base A C = (A , λ _ → C)

-- interface for recursive step of a fold
Step : (A B C : Set) → Poly
Step A B C = (A × B × C , λ _ → C)

-- interface for a fold
Fold : (A B C : Set) → Poly
Fold A B C = (A × List B , λ _ → C)

-- fold as a program module
fold : {A B C : Set}
     → Fold A B C
       ⇒ Free (sum foldlabels
                   (λ{ base → Base A C
                     ; step → Step A B C}))
fold (a , nil) =
    call[ c ← (base , a) ]
    return c
fold (a , cons b bs) =
    do[ c ← fold (a , bs) ]
    call[ c' ← (step , (a , b , c)) ]
    return c'

-- program modules for append
appendBase : {A : Set}
           → Base (List A) (List A) ⇒ Free zeroPoly
appendBase xs = return xs

appendStep : {A : Set}
           → Step (List A) A (List A) ⇒ Free zeroPoly
appendStep (_ , x , xs) = return (cons x xs)

-- ...and for concat
concatBase : {A : Set}
           → Base ⊤ (List A)
             ⇒ Free zeroPoly
concatBase _ = return nil

concatStep : {A : Set}
           → Step ⊤ (List A) (List A)
             ⇒ Free (Fold (List A) A (List A))
concatStep (_ , xs , ys) =
    call[ zs ← (ys , xs) ]
    return zs
```

```
-- mapping a polynomial kleisli morphism over a free monad
Free⇒ : {p q : Poly} {E : Set}
      → Free p E → (p ⇒ Free q) → Free q E
Free⇒ (return e) f = return e
Free⇒ (bind a k) f = (f a) >>= (λ b → Free⇒ (k b) f)

-- Kleisli composition for the free-monad-monad
_∘_ : {p q r : Poly}
    → (p ⇒ Free q) → (q ⇒ Free r) → p ⇒ Free r
f ∘ g = λ a → Free⇒ (f a) g

-- U-ary composition
comp : (U : Set) {p : Poly} {q : U → Poly} {r : Poly}
     → (p ⇒ Free (sum U q))
     → ((u : U) → q u ⇒ Free r)
     → p ⇒ Free r
comp U {p = p} {q = q} {r = r} f g = f ∘ (λ (u , x) → g u x)

-- composing the implementations for append and concat
module example2 where

    open example1

    append : {A : Set}
           → Fold (List A) A (List A) ⇒ Free zeroPoly
    append =
        comp foldlabels fold
             (λ{ base → appendBase
               ; step → appendStep})

    concat : {A : Set}
           → Fold ⊤ (List A) (List A) ⇒ Free zeroPoly
    concat =
        comp foldlabels fold
             (λ{ base → concatBase
               ; step → concatStep ∘ append})

-- wiring diagrams
module WD (Boxes : Set)
          (Arity : Boxes → Set)
          (Dom : (b : Boxes) → Arity b → Poly)
          (Cod : Boxes → Poly)
          (Inputs : Set) (inputs : Inputs → Poly) where

    data Wiring : (output : Poly) → Set₁ where
        wire : (i : Inputs) → Wiring (inputs i)
```

```
        box : (b : Boxes) → ((a : Arity b) → Wiring (Dom b a))
            → Wiring (Cod b)

open WD public

compose : {Boxes : Set} {Arity : Boxes → Set}
        → {Dom : (b : Boxes) → Arity b → Poly}
        → {Cod : Boxes → Poly} {Inputs : Set}
        → {inputs : Inputs → Poly} {output : Poly}
        → Wiring Boxes Arity Dom Cod Inputs inputs output
        → ((b : Boxes) → Cod b ⇒ Free (sum (Arity b) (Dom b)))
        → output ⇒ Free (sum Inputs inputs)
compose (wire i) f x = bind (i , x) return
compose (box b f) g = comp _ (g b) (λ a → compose (f a) g)

-- Mealy machines
record Mealy (p : Poly) : Set where
    coinductive
    field
        app : (x : p .fst) → p .snd x × Mealy p

open Mealy public

-- run a mealy machine on a "program" of type Free p C
run-mealy : {p : Poly} {C : Set}
          → Free p C → Mealy p → C × Mealy p
run-mealy (return c) m = (c , m)
run-mealy (bind x f) m =
    let (b , m') = m .app x in
    run-mealy (f b) m'

-- use a polynomial morphism p ⇒ Free q to convert a Mealy
-- machine implementing q to a Mealy machine implementing p
prog→mealy : {p q : Poly}
           → (f : p ⇒ Free q) → Mealy q → Mealy p
prog→mealy {p = p} {q = q} f m .app a =
    let (b , m') = run-mealy (f a) m in
    (b , prog→mealy f m')

-- type of inputs for the state effect
data StateI (S : Set) : Set where
    get : StateI S
    put : S → StateI S

State : Set → Poly
State S .fst = StateI S
```

```
State S .snd get = S
State S .snd (put s) = ⊤

state-mealy : {S : Set} → S → Mealy (State S)
state-mealy s .app get = (s , state-mealy s)
state-mealy s .app (put s') = (_ , state-mealy s')

-- closure of Mealy machines under binary sum
_mealy⊕_ : {p q : Poly}
         → Mealy p → Mealy q → Mealy (p ⊕ q)
_mealy⊕_ m₁ m₂ .app (inl a) =
    let (b , m') = m₁ .app a in (b , (m' mealy⊕ m₂))
_mealy⊕_ m₁ m₂ .app (inr c) =
    let (d , m') = m₂ .app c in (d , (m₁ mealy⊕ m'))

-- closure of Mealy machines under nullary sum
mealyzero : Mealy zeroPoly
mealyzero .app ()

-- example: fibonacci
module example4 where

    open import Agda.Builtin.Nat

    fib-update : (⊤ , λ _ → Nat) ⇒ Free (State (Nat × Nat))
    fib-update _ =
        call[ (x , y) ← get ]
        call[ _ ← put (y , x + y) ]
        return x

    fib : Mealy (⊤ , λ _ → Nat)
    fib = prog→mealy fib-update (state-mealy (0 , 1))

-- dependent polynomials
DepPoly : Poly → Set₁
DepPoly (A , B) = Σ (A → Set) (λ C → (x : A) → C x → B x → Set)

-- action of a dependent polynomial
⊙Dep : (p : Poly) → DepPoly p → (E : Set) → (E → Set) → p ⊙ E → Set
⊙Dep (A , B) (C , D) E F (a , f) =
    Σ (C a) (λ c → (b : B a) → D a c b → F (f b))

-- morphisms of dependent polynomials
⇒Dep : (p : Poly) → DepPoly p → {F : Set → Set}
     → ((X : Set) → (X → Set) → F X → Set)
     → (f : p ⇒ F) → Set
```

```
⇒Dep (A , B) (C , D) G f =
    (a : A) (c : C a) → G (B a) (D a c) (f a)

-- free dependent monads
data FreeDep (p : Poly) (r : DepPoly p)
             (E : Set) (F : E → Set) : Free p E → Set where
    returnD : {e : E} → F e → FreeDep p r E F (return e)
    bindD : {a : p .fst} (c : r .fst a)
          → {k : (p .snd a) → Free p E}
          → ((b : p .snd a) → r .snd a c b
                            → FreeDep p r E F (k b))
          → FreeDep p r E F (bind a k)

-- monadic bind for FreeDep
>>=Dep : {p : Poly} {r : DepPoly p}
       → {E : Set} {F : E → Set} {G : Set} {H : G → Set}
       → (f : Free p E) → FreeDep p r E F f
       → {g : E → Free p G}
       → ((e : E) → F e → FreeDep p r G H (g e))
       → FreeDep p r G H (f >>= g)
>>=Dep (return e) (returnD f) gg = gg e f
>>=Dep (bind a k) (bindD c h) gg =
    bindD c (λ b x → >>=Dep (k b) (h b x) gg)

-- sums of dependent polynomials
sumDep : (U : Set) (p : U → Poly)
       → (r : (u : U) → DepPoly (p u)) → DepPoly (sum U p)
sumDep U p r .fst (u , x) = r u .fst x
sumDep U p r .snd (u , x) y = r u .snd x y

-- binary coproduct of dependent polynomials
depPoly⊕ : {p q : Poly} (r : DepPoly p) (s : DepPoly q)
         → DepPoly (p ⊕ q)
depPoly⊕ (A , B) (C , D) .fst (inl a) = A a
depPoly⊕ (A , B) (C , D) .fst (inr c) = C c
depPoly⊕ (A , B) (C , D) .snd (inl a) y = B a y
depPoly⊕ (A , B) (C , D) .snd (inr c) y = D c y

-- nullary coproduct of dependent polynomials
depPolyzero : DepPoly zeroPoly
depPolyzero .fst ()
depPolyzero .snd ()

-- mapping a dependent polynomial morphism over
-- a free dependent monad
FreeDep⇒ : {p q : Poly} {r : DepPoly p} {s : DepPoly q}
```

```
          → {E : Set} {F : E → Set}
          → (f : Free p E) → FreeDep p r E F f
          → (g : p ⇒ Free q) → (⇒Dep p r (FreeDep q s) g)
          → FreeDep q s E F (Free⇒ f g)
FreeDep⇒ (return e) (returnD f) g gg = returnD f
FreeDep⇒ (bind a k) (bindD c h) g gg =
    >>=Dep (g a) (gg a c) (λ e x → FreeDep⇒ (k e) (h e x) g gg)

-- Kleisli composition for the free dependent monad
○Dep : {p : Poly} {q : Poly} {r : Poly}
     → {s : DepPoly p} {t : DepPoly q} {v : DepPoly r}
     → {f : p ⇒ Free q} {g : q ⇒ Free r}
     → (⇒Dep p s (FreeDep q t) f)
     → (⇒Dep q t (FreeDep r v) g)
     → ⇒Dep p s (FreeDep r v) (f ○ g)
○Dep ff gg a c = FreeDep⇒ _ (ff a c) _ gg

-- multi-ary Kleisli composition of
-- dependent polynomial morphisms
compDep : (U : Set) {p : Poly} {q : U → Poly} {r : Poly}
          {s : DepPoly p} {t : (u : U) → DepPoly (q u)}
          {v : DepPoly r} {f : p ⇒ Free (sum U q)}
        → {g : (u : U) → q u ⇒ Free r}
        → (⇒Dep p s (FreeDep (sum U q) (sumDep U q t)) f)
        → ((u : U) → ⇒Dep (q u) (t u) (FreeDep r v) (g u))
        → ⇒Dep p s (FreeDep r v) (comp U f g)
compDep U ff gg = ○Dep ff (λ (u , x) c → gg u x c)

-- dependent Mealy machines
record DepMealy (p : Poly) (r : DepPoly p) (m : Mealy p) : Set where
    coinductive
    field
        appD : (x : p .fst) (y : r .fst x)
             → r .snd x y (m .app x .fst)
               × DepMealy p r (m .app x .snd)

open DepMealy public

-- specification for state invariants
Invariant : (S : Set) (T : S → Set) → DepPoly (State S)
Invariant S T .fst get = ⊤
Invariant S T .fst (put s) = T s
Invariant S T .snd get _ s = T s
Invariant S T .snd (put s) t _ = ⊤

-- verification of a state invariant
```

```
invariant : {S : Set} {T : S → Set}
          → (s : S) → T s
          → DepMealy (State S) (Invariant S T) (state-mealy s)
invariant s t .appD get _ = (t , (invariant s t))
invariant s t .appD (put s') t' = (_ , invariant s' t')

-- run a verified Mealy machine on a verified program
run-mealyD : {p : Poly} {r : DepPoly p} {E : Set} {F : E → Set}
           → {e : Free p E} {m : Mealy p}
           → FreeDep p r E F e → DepMealy p r m
           → (F (run-mealy e m .fst)
             × DepMealy p r (run-mealy e m .snd))
run-mealyD (returnD f) mm = (f , mm)
run-mealyD (bindD c h) mm =
    let (d , mm') = mm .appD _ c in
    run-mealyD (h _ d) mm'

-- apply a verified program to a verified Mealy
-- machine to obtain a new verified Mealy machine
prog→mealyD : {p q : Poly} {r : DepPoly p} {s : DepPoly q}
            → {f : p ⇒ Free q} {m : Mealy q}
            → (⇒Dep p r (FreeDep q s) f) → DepMealy q s m
            → DepMealy p r (prog→mealy f m)
prog→mealyD ff mm .appD a c =
    let (b , mm') = run-mealyD (ff a c) mm in
    (b , (prog→mealyD ff mm'))

-- closure of DepMealy under binary coproducts
_depMealy⊕_ : {p q : Poly} {r : DepPoly p} {s : DepPoly q}
            → {m₁ : Mealy p} {m₂ : Mealy q}
            → DepMealy p r m₁ → DepMealy q s m₂
            → DepMealy (p ⊕ q) (depPoly⊕ r s) (m₁ mealy⊕ m₂)
_depMealy⊕_ m₁ m₂ .appD (inl a) c =
    let (b , m') = m₁ .appD a c in
    (b , (m' depMealy⊕ m₂))
_depMealy⊕_ m₁ m₂ .appD (inr c) d =
    let (e , m') = m₂ .appD c d in
    (e , (m₁ depMealy⊕ m'))
```