THE SYNTHETIC THEORY OF POLYNOMIAL FUNCTORS

DAVID SPIVAK, OWEN LYNCH, REED MULLANIX, SOLOMON BOTHWELL,
AND VERITY SCHEEL

The theory of polynomial functors is a powerful mathematical tool that sees use across a broad range of mathematics, ranging from the very applied (modelling dynamical systems) to the very abstract (constructing models of type theory). However, mechanizing the theory of polynomial functors makes it unwieldy to use; you either forgo all the power of abstraction and work at the level of sets, or are forced to work in a setting entirely without variables, which is mentally taxing, and does not scale well to large expressions. There currently exist some graphical tools such as wiring diagrams that do help the situation, but these only cover a fragment of the theory, and are difficult to integrate into proof assistants.

In light of this, we argue that we ought to be working in the *synthetic* theory of polynomial functors, where polynomials and their morphisms are defined as part of the theory directly. This has several benefits. From a purely ergonomic standpoint, having the theory be aware of the existence of maps of polynomials means that working with them is much more pleasant; gone are the days of massive chains of composites! Furthermore, we are able to exploit the rich structure of polynomials during evaluation.

To back up these claims, we present `PolyTT`, a type-theoretic encoding of the synthetic theory of polynomial functors. It combines Martin Löf Type Theory with a fragment of linear type theory, used for constructing morphisms of polynomials.

1. The Type Theory

We omit the standard rules for Martin Löf Type Theory with Tarski Universes.

Remark 1. We will write variables as $x^{(\pm)}, y^{(\pm)}, z^{(\pm)}$ when we are referencing explicit bindings, as opposed to general expressions. We will use x^+ for normal variables in the forwards direction of morphisms, x^- for sinks (negative variables, which are necessarily in the reverse direction of morphisms), and plain x for binding the positive value being threaded through the reverse direction. That is, our convention will be that $x^- = (\lambda^-\ x \to (x^- \leftarrow x); \mathsf{done})$ (eta-expansion of sinks). In terms of polynomial functors, x^+ will usually have type $\mathsf{base}\ P$, and x^- and x will both have type $\mathsf{fib}\ P\ x^+$ but as negative and positive terms respectively.

MLTT has three kinds of things:

(1) Contexts Γ *ctx*, with variables of specified types, and possibly definitions for those variables (e.g. we allow a concrete variable $(one := 1 : \mathbb{N})$ to be in context, not just an abstract $(n : \mathbb{N})$).
(2) Types A *type*.
(3) Terms $a : A$.

These form the positive fragment of `PolyTT`. The negative fragment is used for writing morphisms of polynomials in convenient syntax, where linear contexts are used to keep track of obligations that correspond to wires requiring input in wiring diagrams.

Thus we add two type formers to the positive fragment, and more concepts for the negative fragment:

(1) A type of polynomial functors $P, Q : \mathsf{Poly}$.
(2) A type of morphisms between them $\phi, \psi : P \Rightarrow Q$. The following concepts are used to write terms of these morphisms.
(3) Negative (linear) contexts Ω *NegCtx*, which introduce sinks (negative variables of negative types) and record their values when they are written to later. This forms compound contexts $\Gamma \mid \Omega$, where the vertical bar is mere syntax.
(4) Negative types A^- *type* for sinks, which are exactly like MLTT types but may mention negative values via borrowing. These are conceptually thought of as actions *out of* A (like $A \to R$ for some fixed and arbitrary R).
(5) Programs π *Program*, which write values to negative terms (sinks), which gets recorded in the context Ω.
(6) Negative terms $a^- : A^-$ (i.e. sinks), which can be set to values in programs. That is, they are abstracted programs which take in a positive value and perform actions of writing to other sinks based on that value.
(7) Morphism bodies $\rho :\Rightarrow Q$, which have a sequence of programs and then conclude in a positive base value and negative fiber sink (this is a technical wart of our presentation: for a poly-morphism of type $P \Rightarrow Q$, after introducing the bits from P we still need to hold onto the right return type Q for the end of the body, so we encode this in a judgment which I cheekily call $- :\Rightarrow Q$ since it is the "type" of being a program "into Q")

Programs have a few operations:

(1) Writing a positive value to a negative term of the same type. We will write it as $a^- \leftarrow a^+$.
(2) Done, the program which does nothing. (This is necessary for the grammar to be right-associative.) We will write it as done.
(3) Unpack, which splits a sigma-sink into two sinks (with a type dependency). This is a necessary primitive. We will write it as let $(x^-, y^-) := v^-$.
(4) Pair, which bundles up two sinks into a sink of the corresponding sigma type. This can be implemented in user-land with $ab^- := \lambda^- \; ab \to (a^- \leftarrow \mathrm{fst}\, ab); (b^- \leftarrow \mathrm{snd}\, ab); \mathsf{done}$, in place of a builtin combinator like (a^-, b^-), or $(a^- \leftarrow a, b^-)$ for a dependent pair.
(5) Morphism application let $(q^+ \leftsquigarrow q^-) := \phi(p^+ \leftsquigarrow p^-)$ can also be implemented in user-land. (Note: if we make boxes (section 7) first-class, this will just be interpreting $P \Rightarrow Q$ as a function $\mathsf{Box}P \to \mathsf{Box}Q$ and applying it to (p^+, p^-).)

All of these operations are allowed in both programs and morphism bodies. We will only write out the judgments for the latter: the former can be derived by replacing all instances of $\rho :\Rightarrow Q$ with π *Program*, except for DERIV:BODY-DONE which corresponds simply to this derivation:

$$\textsc{Deriv:Done}\ \frac{}{\Gamma \mid \Omega \vdash \mathsf{done}\ \mathit{Program}}$$

Note that this means there is no linearity checking at the end of programs: we only care about linearity at the very end of a morphism, and programs are only small pieces of them. Well, we need to check that any additional sinks introduced during the program are written to, in particular since they cannot be referenced outside of the program at all, but we do not expect that every sink already in the context is written to by the end of a program.

2. Overview on linearity

Linearity is checked by writing a negative variable's written value in context Ω, never allowing it to be written twice, and checking that all variables introduced in Ω have been written by the end of a morphism-body.

Writing a variable only occurs in programs and morphism bodies. This introduces additional definitional equalities for $\mathsf{borrow}\ a^-$ into scope. Namely, $x^- \leftarrow a$ allows for the reduction $\mathsf{borrow}\ x^- \equiv a$, like so:

$$\textsc{Deriv:Borrow-reduce}\ \frac{}{\Gamma \mid \Omega_L, (x^- : A^-), \Omega_M, (x^- \leftarrow a), \Omega_R \vdash \mathsf{borrow}\ x^- \equiv a : A}$$

We can more succinctly write this like so:

$$\textsc{Deriv:Borrow-reduce}\ \frac{(x^- \leftarrow : A^-) \in \Omega}{\Gamma \mid \Omega \vdash \mathsf{borrow}\ x^- \equiv a : A}$$

Negative terms in negative types are not consumed. In particular, mentioning a negative variable in $\mathsf{borrow}\ a^-$ does not consume it, so this means that the only difference between types and negative types are that the latter may mention $\mathsf{borrow}\ a^-$.

$$\textsc{Deriv:Borrow-negative}\ \frac{\Gamma \mid \Omega \vdash a^- : A^-}{\Gamma \mid \Omega \vdash \mathsf{borrow}\ a^- : A}$$

This means that the MLTT fragment does not need to care about linearity. It only needs to pass along the compound context $\Gamma \mid \Omega$. For example, the application rule looks like this:

$$\textsc{Deriv:Ap}\ \frac{\Gamma \mid \Omega \vdash f : (a : A) \to B[a] \qquad \Gamma \mid \Omega \vdash v : A}{\Gamma \mid \Omega \vdash f(v) : B[v]}$$

3. Negative contexts and types

Negative contexts are a bit weird in my presentation here: they introduce variables of specified negative types, but they also write them to values. They are inductively generated with three constructors: empty, introducing a variable, and writing a previously-introduced variable.

Negative types are exactly regular types with access to Ω (namely giving borrow x^- for all negative variables x^- in scope thus far and any known definitional equalities from them), so we omit its derivation rules as they are inherited from MLTT, with the addition of the two mentioned earlier regarding borrow.

$$\text{Judg:Neg-ctx}\ \frac{\Gamma\ ctx}{\Gamma \vdash \Omega\ NegCtx} \qquad \text{Judg:Neg-type}\ \frac{\Gamma\ ctx \qquad \Gamma \vdash \Omega\ NegCtx}{\Gamma \mid \Omega \vdash A^-\ type}$$

$$\text{Deriv:Neg-ctx-empty}\ \frac{\Gamma\ ctx}{\Gamma \vdash \cdot\ NegCtx}$$

We can always introduce a negative variable of a valid negative type (assuming uniqueness of variables blah blah):

$$\text{Deriv:Neg-ctx-intro}\ \frac{\Gamma \vdash \Omega\ NegCtx \qquad \Gamma \mid \Omega \vdash A^-\ type}{\Gamma \vdash \Omega, (x^- : A^-)\ NegCtx}$$

But to write to a variable, we need to know that it is defined in context (with the corresponding type) *and* was not yet written to (the asterisk here is a wildcard):

$$\text{Deriv:Neg-ctx-write}\ \frac{\Gamma \vdash \Omega\ NegCtx \qquad \Gamma \mid \Omega \vdash a : A \qquad (x^- : A^-) \in \Omega \qquad (x^- \leftarrow *) \notin \Omega}{\Gamma \vdash \Omega, (x^- \leftarrow a)\ NegCtx}$$

There is a weird side condition here: we need to know that the graph of borrow-dependencies is acyclic (a DAG). (Perhaps we could encode this invariant more deeply into the context, representing it as a tree or something, but then we need to unblock dependencies when things compute away ...)

Anyways. What happens right now is that we enforce that side-condition by

(1) not allowing users to mention borrow themselves (see Deriv:Neg-Write); and
(2) being careful with the dependencies we introduce in the few rules that do include borrow (Deriv:Unpack, Deriv:Morph-Ap, Deriv:Lending).

We could state this as a metatheorem about our theory: that all derivations from the rules here result in sinks never borrowing their own value. This is clearly necessary for Poly to be a model, thus we also want it to hold for the initial (syntactic) model.

The linearity check is ensured by never writing to the same variable twice and by checking that each variable in Ω is written exactly once by the end of the hom-body.

In particular, this is used for program extraction of the action on fibers, by seeing what the input sink p^- is set to in the environment when the output sink q^- is written with an abstract test value q. That is, the context will ultimately contain a description of p^- in terms of q, and this is the action on fibers.

4. Negative terms and programs and program environments

Programs are exactly the bodies of eta-expanded negative terms:

$$\textsc{Deriv:Neg-Eta}\quad \frac{\Gamma, a : A \mid \Omega \vdash \pi \ \mathit{Program}}{\Gamma \mid \Omega \vdash (\lambda^- \ a \to \pi) : A^-}$$

As an example, there is a negative term drop that accepts anything and writes nothing: $\mathsf{drop} := (\lambda^- \ a \to \mathsf{done}) : A^-$.

We can eta-expand sinks, so a sink $a^- : A^-$ becomes $(\lambda^- \ a \to (a^- \leftarrow a); \mathsf{done}) : A^-$, where we abstract out the value being eventually written to the sink and then have the program that writes it to the original sink.

We can even apply a non-dependent function $f : A \to B$ to map sinks contravariantly: from $b^- : B^-$ we get $(b^- \circ f) := (\lambda^- \ a \to (b^- \leftarrow f(a)); \mathsf{done}) : A^-$.

Programs can write positive values a to negative terms a^-. Two things:

(1) Note that a cannot depend on any negative terms (in particular, cannot borrow). That means that a^- needs its type A^- to reduce to a term that does not mention borrowed values, either by having no dependence on other sinks in the first place or by already being written to a value from previous writes in Ω.
(2) Speaking of Ω, a^- might not be a variable, so we need to be able to write out the side-effects of writing to it back into Ω'. We do this with an operation $\downarrow (a^- \leftarrow a)$ that we will write out later in section 4.1.

$$\textsc{Deriv:Neg-Write}\quad \frac{\Gamma \vdash a : A \qquad \Gamma \mid \Omega \vdash a^- : A^- \qquad \Gamma \mid \Omega, \downarrow (a^- \leftarrow a) \vdash \rho :\Rightarrow Q}{\Gamma \mid \Omega \vdash ((a^- \leftarrow a); \rho) :\Rightarrow Q}$$

We can let-bind negative terms as a form of lending: passing forward the obligation to write to the sinks mentioned in a^- as an obligation to write to x^-. This basic mechanism is also used in Deriv:Morph-Ap.

$$\textsc{Deriv:Lending}\quad \frac{\Gamma \mid \Omega \vdash a^- : A^- \qquad \Gamma \mid \Omega, x^- : A^-, \downarrow (a^- \leftarrow \mathsf{borrow}\ x^-) \vdash \rho :\Rightarrow Q}{\Gamma \mid \Omega \vdash (\mathsf{let}\ \ x^- : A^- := a^-; \rho) :\Rightarrow Q}$$

4.1. **Running programs into context.**

To write programs into context, we need to run them. (Note: this is where I differ from Reed's rules in that I want to be able to write to arbitrary negative terms, instead of simplifying the rules to only write to negative variables.)

It is doing nothing more than encapsulating how the existing rules for each program constructor manipulate the context as they proceed to the next typing derivation.

The reason for having this encapsulation is to keep the meta-theory first-order, pen-and-paper style. If we had continuations/HOAS in the meta-theory, we could use those to avoid writing this out. (In fact, this is what I did in the implementation.) But at least it is not hard.

(Note that this isn't an inductive constructor for negative contexts, it is a function in the metatheory that produces negative contexts.)

$$\frac{\Gamma \vdash \Omega \ \mathit{NegCtx} \qquad \Gamma \mid \Omega \vdash \pi \ \mathit{Program}}{\Gamma \vdash \Omega, \downarrow \pi \ \mathit{NegCtx}} \ \textsc{Judg:Exec}$$

Uhh I am describing its behavior in English for now: A program is a series of writes that we want to write into context (and it ends with done).

(1) Writing to a variable is easy $x^- \leftarrow a$: we add the appropriate constructor into context, so we produce $\Omega, (x^- \leftarrow a)$.
(2) Writing to an eta-expanded negative term is also easy $(\lambda^- \ x \to \pi[x]) \leftarrow a$: we now know the value of x, so we substitute it in, and write that program into context, $\Omega, \downarrow \pi[a]$. (This is recursive.)
(3) If we have a let-binding (lending) of the form let $x^- : A^- := a^-$ we need to represent that by adding this, just as the typing rule Deriv:Lending does: $\Omega, x^- : A^-, \downarrow (a^- \leftarrow \mathsf{borrow}\ x^-)$.
(4) We similarly deal with Deriv:Morph-Ap and Deriv:Unpack which are also recursive.

Again, the semantics of this model the obvious thing they are already doing in their respective typing rules, since we had to write their behavior into the judgments already.

5. Morphism body

This provides the forward component p^+ and adds a sink p^-, then checks that the body of the morphism ρ will fulfill the obligation by writing to the sink.

$$\frac{\Gamma, p^+ : \mathsf{base}\ P \mid p^- : (\mathsf{fib}\ P\ p^+)^- \vdash \rho :\Rightarrow Q}{\Gamma \vdash (\lambda(p^+ \leftsquigarrow p^-) \rightsquigarrow \rho) : P \Rightarrow Q} \ \textsc{Deriv:Morph-Intro}$$

The obligation is fulfilled when the body ends, providing a positive value for the base and a negative sink for the fiber, and every negative variable has been set to a value in Ω after writing to q^- (yes we invent a new variable q to write to q^-).

$$\frac{\Gamma \vdash q^+ : \mathsf{base}\ Q \qquad \Gamma \mid \Omega \vdash q^- : (\mathsf{fib}\ Q\ q^+)^- \qquad \Omega, \downarrow (q^- \leftarrow q) \ \mathit{Filled}}{\Gamma \mid \Omega \vdash (q^+ \leftsquigarrow q^-) :\Rightarrow Q} \ \textsc{Deriv:Body-Done}$$

The action on fibers can be recovered as something like $\lambda q \to p[q]$ where $\Gamma, p^+ : \mathsf{base}\ P, q^+ : \mathsf{base}\ Q \mid \Omega$, where $\Omega := (p^- : \mathsf{fib}\ P\ p^+, \downarrow (q^- \leftarrow q)) = (p^- : \mathsf{fib}\ P\ p^+, ..., (p^- \leftarrow p[q]), ...)$ means p^- gets set to $p[q]$ in the process of setting q^- to q.

Apply a morphism. This takes a base, fib pair and turns it into a new base, fib pair by applying the forward and backwards components of the morphism. It can be implemented via positive and negative let in the obvious way, so it is a conservative extension, but then again, so is everything we are doing here.

DERIV:MORPH-AP

$$\frac{\begin{array}{c}\Gamma \vdash \phi : P \Rightarrow Q \qquad \Gamma \vdash p^+ : \mathsf{base}\ P \qquad \Gamma \mid \Omega \vdash p^- : (\mathsf{fib}\ P\ p^+)^- \\ \Gamma' := \Gamma, q^+ := \mathsf{base}\ \phi(p^+) : \mathsf{base}\ Q \\ \Omega' := \Omega, q^- : (\mathsf{fib}\ Q\ q^+)^-, \downarrow (\mathsf{fib}\ \phi\ p^+ (p^- \leftarrow \mathsf{borrow}\ q^-)) \\ \Gamma' \mid \Omega' \vdash \rho :\Rightarrow Q\end{array}}{\Gamma \mid \Omega \vdash (\text{let } (q^+ \leftsquigarrow q^-) := \phi(p^+ \leftsquigarrow p^-); \rho) :\Rightarrow Q}$$

6. SIGMAS

Consumes ab^-, splitting it into two new sinks (x^-, y^-) whose values get paired and written to ab^-. Note that the type of the second sink depends on the value written to the first sink. Since $\mathsf{borrow}\ x^-$ cannot be mentioned in user-code, due to the typing restriction in DERIV:NEG-WRITE, if $B[\mathsf{borrow}\ x^-]^-$ does not reduce by virtue of being a constant family, one must write to x^- before y^-.

DERIV:UNPACK

$$\frac{\begin{array}{c}\Gamma \mid \Omega_1 \vdash ab^- : ((a : A) \times B[a])^- \\ \Omega_2 := \Omega_1, x^- : A^-, y^- : B[\mathsf{borrow}\ x^-]^-, \downarrow (ab^- \leftarrow (\mathsf{borrow}\ x^-, \mathsf{borrow}\ y^-)) \\ \Gamma \mid \Omega_2 \vdash \rho :\Rightarrow Q\end{array}}{\Gamma \mid \Omega_1 \vdash (\text{let } (x^-, y^-) := ab^-; \rho) :\Rightarrow Q}$$

The inverse can be implemented simply in user-land, as

$$\text{let } \ ab^- := \lambda^-\ ab \to (a^- \leftarrow \mathrm{fst}\, ab); (b^- \leftarrow \mathrm{snd}\, ab); \mathsf{done}.$$

7. BOXES

Probably the better UX is to provide (un)tensoring for polynomials, that handles forwards and backwards directions at once. Macros, anyone???

I guess this corresponds to boxes in the sense of wiring diagrams, and to values in the polynomial functor over a generic type R: $(p^+, p^-) : \Sigma(p^+ : \mathsf{base}\ P), \mathsf{fib}\ P\ p^+ \to R$, since sinks are conceptually maps into the generic R.

(What are programs, then?)

The complication is that boxes are positive mixed with negative, and it's not clear where they should live and how much additional rules they need. In particular, I want to have to avoid special let-binding constructs for every kind of syntax we want to add.

Like, I don't think this works since it doesn't keep track of linearity properly maybe? but maybe it really is okay to pass around sinks freely and only check linearity when they are written to:

DERIV:BOX

$$\frac{\Gamma \vdash P : \mathsf{Poly}}{\Gamma \mid \Omega \vdash \mathsf{Box} P : \mathsf{U}}$$

$$\textsc{Deriv:Box-Intro}\ \frac{\Gamma \vdash p^+ : \mathsf{base}\ P \qquad \Gamma \mid \Omega \vdash p^- : \mathsf{fib}\ P\ p^+}{\Gamma \mid \Omega \vdash (p^+ \leftsquigarrow p^-) : \mathsf{Box} P}$$

8. Future work

Could extend with higher-order abstraction, e.g. the ability to lambda-bind *sinks* and not just positive values. (Note that we just support let-binding sinks right now.)

Could allow users to mention borrow themselves and implement actual cycle-checking to ensure it is still safe.

Could implement control flow for select primitive recursors, such as on enumerations (i.e. case/switch statements) and maybe natural numbers. You need to check that the same sinks are written to in each branch, and then abstract out their values over the recursor. (This could interact badly with higher-order abstraction.)