# THE SYNTHETIC THEORY OF POLYNOMIAL FUNCTORS

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

## 1. Verity's Rules (Working Document)

Blah blah blah Martin-Löf Type Theory.

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 $x^- = \lambda x \to (x \mapsto x^-)$ (eta-expansion of sinks). In terms of polynomial functors, $x^+$ will usually have type *Base* $P$, and $x^-$ and $x$ will both have type *Fib* $P$ $x^+$ but as negative and positive terms respectively.

MLTT has three things ("kinds"?):

(1) Contexts, with variables of types
(2) Types
(3) Terms

To this we add:

(1) A type of polynomial functors $P, Q$ : *Poly* and a type of morphisms between them $P \Rightarrow Q$
(2) Negative (linear) contexts $\Psi$, with variables of negative types
(3) Negative types $A^-$, which are exactly like MLTT types but may mention negative values via borrowing (note that $-^-$ is no longer an operation, just notation to distinguish negative types)
(4) Programs $\pi$, which set negative terms to values
(5) Program environments $\chi$, which are normalized programs $\chi = \downarrow \pi$ writing variables (as opposed to generalized negative terms) in *cons*-order (not *snoc*!)
(6) Negative terms $a^-$, which can be set to values in programs
(7) Morphism bodies $\rho$, which have a sequence of programs and then conclude in a positive return value and negative fibrational component

Programs have a few operations:

(1) Write, of course. We will write it as $a^+ \mapsto a^-$.
(2) Done, which is necessary for the grammar to be right-associative. We will write it as $\varnothing$.
(3) Unpack, which splits a sigma-wire into two wires (with a type dependency). We will write it as $v^- \prec (x^-, y^-)$.
(4) Pair, which bundles up two wires (with a type dependency) into a sigma-wire. This can be implemented in user-land: $(a^-, b^-) := \lambda\ ab \to (\mathrm{fst}\, ab \mapsto a^-); (\mathrm{snd}\, ab \mapsto b^-)$.

(5) Morphism application can also be implemented in user-land (although it loses sharing but eh we don't have that anyways).

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

Writing a variable only occurs in negative terms in programs and morphism bodies. This introduces additional definitional equalities for borrow $a^-$ into scope. Namely, $a \mapsto x^- : A$ allows for the reduction borrow $x^- \equiv a$.

Deriv:Borrow-reduce

$$\frac{}{\Gamma \mid \Psi \mid \chi_L, (a \mapsto x^- : A^-), \chi_R \vdash \text{borrow } x^- \equiv a : A}$$

TODO: what to do about compound negative terms? disallow them? yes! definitely.

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

Deriv:Borrow-negative

$$\frac{\Gamma \mid \Psi \mid \chi \vdash a^- : A^-}{\Gamma \mid \Psi \mid \chi \vdash \text{borrow } a^- : A}$$

This means that the MLTT fragment does not need to care about linearity. It only needs to pass along the compound context $\Gamma \mid \Psi \mid \chi$. ($\chi$ is here for definitional equalities that may be necessary for some terms to typecheck.) For example, the application rule looks like this:

Deriv:Ap

$$\frac{\Gamma \mid \Psi \mid \chi \vdash f : (a : A) \to B[a] \qquad \Gamma \mid \Psi \mid \chi \vdash v : A}{\Gamma \mid \Psi \mid \chi \vdash f(v) : B[v]}$$

It would be nice to get rid of the morphism-body, since various judgments are polymorphic over it in their return (e.g. sequencing and negsima-elim), but let's gloss over that for now. Sequencing is right-associative: each statement may bind variables and write values that appear in all following statements. In fact that needs to be baked into the syntax, so that the environment can remain a reader context and not devolve into writer or state context.

## 2. Negative contexts and types

Negative contexts have variables of specified negative types. Negative types are exactly regular types with access to borrow $x^-$ for all negative variables $x^-$ in scope thus far, so we omit its derivation rules. ($\chi$ is here for definitional equalities that make be necessary for some terms to typecheck.)

Judg:Neg-ctx

$$\frac{\Gamma\ \mathit{Ctx}}{\Gamma \vdash \Psi\ \mathit{NegCtx}}$$

Judg:Neg-type

$$\frac{\Gamma\ \mathit{Ctx} \qquad \Gamma \vdash \Psi\ \mathit{NegCtx}}{\Gamma \mid \Psi \mid \chi \vdash A^-\ \mathit{Type}}$$

$$\textsc{Deriv:Neg-ctx-empty}\ \frac{\Gamma\ \mathit{Ctx}}{\Gamma \vdash \cdot\ \mathit{NegCtx}} \qquad \textsc{Deriv:Neg-ctx-snoc}\ \frac{\Gamma \vdash \Psi\ \mathit{NegCtx} \qquad \Gamma \mid \Psi \vdash A^-\ \mathit{Type}}{\Gamma \vdash \Psi, x^- : A^-\ \mathit{NegCtx}}$$

## 3. Negative terms and programs and program environments

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

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

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 $\chi$.

$$\textsc{Deriv:Neg-Write}\ \frac{\Gamma \vdash a : A \qquad \Gamma \mid \Psi \mid \chi \vdash a^- : A^- \qquad \Gamma \mid \Psi \mid {\downarrow}\,(a \mapsto a^-); \chi \vdash \pi\ \mathit{Program}}{\Gamma \mid \Psi \mid \chi \vdash (a \mapsto a^-); \pi\ \mathit{Program}}$$

Program environments write variables in LtR order, but are constructed RtL.

Operationally, we think of them as operating on mutable cells corresponding to each variable in scope. These cells are initialized with borrow $self$ and written to in order to trace out the reverse direction of poly morphisms.

$$\textsc{Judg:Neg-env}\ \frac{\Gamma \vdash \Psi\ \mathit{NegCtx}}{\Gamma \mid \Psi \vdash \chi\ \mathit{NegEnv}}$$

Their scoping is a bit weird, since they can depend on any variables in $\Psi$ (and are stable under weakening, meaning adding variables to $\Psi$). Note that, unlike Deriv:Neg-Set, their value *can* depend on further borrows! (And definitional equalities introduced into the environment already.) The only place we use this affordance is in Deriv:Unpack.

$$\textsc{Deriv:Neg-env-empty}\ \frac{\Gamma \vdash \Psi\ \mathit{NegCtx}}{\Gamma \mid \Psi \vdash \varnothing\ \mathit{NegEnv}}$$

$$\textsc{Deriv:Neg-env-cons}\ \frac{\Gamma \mid \Psi \vdash \chi\ \mathit{NegEnv} \qquad x^- \notin \chi \qquad \Gamma \mid \Psi \vdash A^-\ \mathit{Type} \qquad \Gamma \mid \Psi \mid \chi \vdash a : A}{\Gamma \mid \Psi \vdash (a \mapsto x^- : A^-); \chi\ \mathit{NegEnv}}$$

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

$$\textsc{Deriv:Neg-env-filled}\ \frac{Vars(\Psi) = Vars(\chi)}{\Psi \cong \chi\ \mathit{Filled}}$$

In particular, this is used for program extraction of the backwards action on fibrations, by seeing what the input $p^-$ is set to in the environment.

We can let-bind negative terms, but it doesn't really buy us much at this point. (Since we will never borrow against them.)

DERIV:LENDING??

$$\frac{\Gamma \mid \Psi \mid \chi \vdash a^- : A^- \qquad \Gamma \mid \Psi, x^- : A^- \mid \downarrow (\text{borrow } \ x^- \mapsto a^-); \chi \vdash \pi_2 \ \textit{Program}}{\Gamma \mid \Psi \mid \chi \vdash \text{let } \ x^- : A^- := a^-; \pi_2 \ \textit{Program}}$$

## 4. Morphism body

This provides the forward component $p^+$ and adds an obligation $p^-$, then checks that the body of the morphism $\rho$ will fulfill the obligation.

DERIV:MORPH-INTRO

$$\frac{\Gamma, p^+ : \textit{Base } P \mid p^- : \textit{Fib } P \ p^+ \mid \varnothing \vdash \rho :\Rightarrow Q}{\Gamma \vdash (\lambda p^+ p^- \rightsquigarrow \rho) : P \Rightarrow Q}$$

The obligation is fulfilled when the body ends, providing a positive value for the base and a negative sink for the fibration, and every negative variable in $\Psi$ has been set to a positive value in $\chi$.

DERIV:BODY-DONE

$$\frac{\Gamma \vdash q^+ : \textit{Base } Q \qquad \Gamma \mid \Psi \vdash q^- : (\textit{Fib } Q \ q^+)^- \qquad \Psi \cong (q \mapsto q^-); \chi \ \textit{Filled}}{\Gamma \mid \Psi \mid \chi \vdash (q^+ \leftsquigarrow q^-) :\Rightarrow Q}$$

The action on fibrations can be recovered as something like $\lambda q.p$ where $\Gamma, p^+ : \textit{Base } P, q^+ : \textit{Base } Q \mid p^- : \textit{Fib } P \ p^+ \mid (q \mapsto q^-) : \textit{Fib } Q \ q^+, (p \mapsto p^-) : \textit{Fib } P \ p^+$, where $p^-$ gets set to $p$ 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

$$\frac{\begin{array}{c} \Gamma \vdash \phi : P \Rightarrow Q \qquad \Gamma \vdash p^+ : \textit{Base } P \qquad \Gamma \mid \Psi \mid \chi \vdash p^- : (\textit{Fib } P \ p^+)^- \\ \Gamma' := \Gamma, q^+ := \textit{Base } \phi(p^+) : \textit{Base } Q \qquad \Psi' := \Psi, q^- := (\textit{Fib } Q \ q^+)^- \\ \chi' := (\textit{Fib } \phi \ p^+(\text{borrow } \ p^-) \mapsto p^-); \chi \qquad \Gamma' \mid \Psi' \mid \chi' \vdash \pi \ \textit{Program} \end{array}}{\Gamma \mid \Psi \mid \chi \vdash (p^+, p^-) \xrightarrow{\phi} (q^+, q^-); \pi \ \textit{Program}}$$

## 5. Sigmas

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

$$\frac{\begin{array}{c}\textsc{Deriv:Unpack}\\ \Gamma \mid \Psi \mid \chi_1 \vdash ab^- : (a : A) \times B^-[a] \\ \chi_2 := \downarrow ((\text{borrow } x^-, \text{borrow } y^-) \mapsto ab^-); \chi_1 \\ \Gamma \mid \Psi, x^- : A^-, y^- : B^-[\text{borrow } x^-] \mid \chi_2 \vdash \pi \textit{ Program}\end{array}}{\Gamma \mid \Psi \mid \chi_1 \vdash ab^- \prec (x^-, y^-); \pi \textit{ Program}}$$

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

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

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