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 $\mathsf{base}\ P$, and x^- and x will both have type $\mathsf{fib}\ P\ x^+$ but as negative and positive terms respectively.

MLTT has three things ("kinds"?):

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

To this we add:

(1) A type of polynomial functors $P, Q : \mathit{Poly}$, and a type of morphisms between them $\phi, \psi : P \Rightarrow Q$
(2) Negative (linear) contexts Ω, which introduces sinks (negative variables of negative types) and later will fill in their values (this forms compound contexts $\Gamma|\Omega$, where the vertical bar is mere syntax)
(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 – the fundamental distinction is whether they are typechecked in just Γ or $\Gamma|\Omega$)
(4) Programs π, which set negative terms (sinks) to values, which then are reflected in the context Ω
(5) Negative terms a^- (sinks), which can be set to values in programs (that is, they are abstracted programs which take in a positive value)
(6) Morphism bodies $\rho :\Rightarrow Q$, which have a sequence of programs and then conclude in a positive return value and negative fibrational component (this is a technical wart of our presentation, we need to hold onto the right return type Q for the end of the body of the poly-morphism of type $P \Rightarrow Q$, after introducing the bits from P)

Programs have a few operations:

(1) Writing a positive value to a negative term of the same type. We will write it as $a^+ \mapsto a^-$.

(2) Done, the program which does nothing, which 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). We will write it as $v^- \prec (x^-, y^-)$.
(4) Pair, which bundles up two sinks (with a type dependency) into a sigma-sink. 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 $(p^+, p^-) \xrightarrow{\phi} (q^+, q^-)$ can also be implemented in user-land (although it loses sharing but eh we don't have that anyways).

2. Overview on linearity

We will preview some rules here, but they will be fully defined and presented in the following sections.

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

Deriv:Borrow-reduce

$$\frac{}{\Gamma \mid \Omega_L, (x^- : A^-), \Omega_M, (a \mapsto x^-), \Omega_R \vdash \mathsf{borrow}\ x^- \equiv a : A}$$

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 $\mathsf{borrow}\ a^-$ does not consume it, so this means that the only difference between types and negative types are that the latter may mention $\mathsf{borrow}\ a^-$.

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:

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 variable.

Negative types are exactly regular types with access to Ω ($\mathsf{borrow}\ x^-$ 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.

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

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

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

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

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):

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

Ugly? I don't know, I'm just hoping it works out.

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 backwards action on fibrations, by seeing what the input p^- is set to in the environment when the output q^- is written with a value.

4. Negative terms and programs and program environments

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

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

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

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^- may 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 $\downarrow (a \mapsto a^-)$ that we will write out later.

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

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

$$\textsc{Deriv:Lending??}\quad \frac{\Gamma \mid \Omega \vdash a^- : A^- \qquad \Gamma \mid \Omega, x^- : A^-, \downarrow (\mathsf{borrow}\ x^- \mapsto a^-) \vdash \pi_2 \ \mathit{Program}}{\Gamma \mid \Omega \vdash \text{let}\ \ x^- : A^- := a^-; \pi_2 \ \mathit{Program}}$$

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.)

On the one hand, it is a bit weird to write out all the rules for this kind of evaluation interleaved with our typing judgments. On the other hand, we know something like it is necessary: our implementation needs to run the negative fragment as it is typechecking.

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

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

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 $a \mapsto x^-$: we add the appropriate constructor into context, so we produce $\Omega, (a \mapsto x^-)$.
(2) Writing to an eta-expanded negative term is also easy $a \mapsto (\lambda x \to \pi[x])$: we substitute the value in we now know the program should use, and write that program into context, $\Omega, \downarrow \pi[a]$. (This is recursive.)
(3) I guess there are also let bindings but it should be obvious what to do there? All other sugar will be implemented on top of let bindings.

5. Morphism body

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

DERIV:MORPH-INTRO
$$\frac{\Gamma, p^+ : \mathsf{base}\ P \mid p^- : \mathsf{fib}\ P\ p^+ \vdash \rho :\Rightarrow Q}{\Gamma \vdash (\lambda p^+ p^- \rightsquigarrow \rho) : P \Rightarrow Q}$$

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 has been set to a value in Ω after writing to q^- (yes we invent a new variable q to write to q^-).

DERIV:BODY-DONE
$$\frac{\Gamma \vdash q^+ : \mathsf{base}\ Q \qquad \Gamma \mid \Omega \vdash q^- : (\mathsf{fib}\ Q\ q^+)^- \qquad \Omega, \downarrow (q \mapsto q^-)\ \textit{Filled}}{\Gamma \mid \Omega \vdash (q^+ \leftsquigarrow q^-) :\Rightarrow Q}$$

The action on fibrations can be recovered as something like $\lambda q.p$ where $\Gamma, p^+ : \mathsf{base}\ P, q^+ : \mathsf{base}\ Q \mid \Omega$, where $\Omega := (p^- : \mathsf{fib}\ P\ p^+, \downarrow (q \mapsto q^-)) = (p^- : \mathsf{fib}\ P\ p^+, ..., (p \mapsto p^-), ...)$ means 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.

TODO: p^- needs to be a variable here, since we are borrowing against it thus it needs a cell allocated for it??

DERIV:MORPH-AP
$$\frac{\begin{array}{c}\Gamma \vdash \phi : P \Rightarrow Q \qquad \Gamma \vdash p^+ : \mathsf{base}\ P \qquad \Gamma \mid \Omega \vdash p^- : (\mathsf{fib}\ P\ p^+)^- \\ \Gamma' := \Gamma, q^+ := \mathsf{base}\ \phi(p^+) : \mathsf{base}\ Q \\ \Omega' := \Omega, q^- : (\mathsf{fib}\ Q\ q^+)^-, (\mathsf{fib}\ \phi\ p^+(\mathsf{borrow}\ p^-) \mapsto q^-) \\ \Gamma' \mid \Omega' \vdash \pi\ \textit{Program}\end{array}}{\Gamma \mid \Omega \vdash (p^+, p^-) \xrightarrow{\phi} (q^+, q^-); \pi\ \textit{Program}}$$

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
$$\frac{\begin{array}{c}\Gamma \mid \Omega_1 \vdash ab^- : (a : A) \times B^-[a] \\ \Omega_2 := \Omega_1, x^- : A^-, y^- : B^-[\mathsf{borrow}\ x^-], \downarrow ((\mathsf{borrow}\ x^-, \mathsf{borrow}\ y^-) \mapsto ab^-) \\ \Gamma \mid \Omega_2 \vdash \pi\ \textit{Program}\end{array}}{\Gamma \mid \Omega_1 \vdash ab^- \prec (x^-, y^-); \pi\ \textit{Program}}$$

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

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

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