

Thinning-Stable Metaprogramming

András Kovács

University of Cambridge

13rd Oct 2026, IFIP WG 2.11 meeting, Niagara-on-the-Lake

Stability

In metaprogramming, it's nice if we don't have to explicitly handle object-language scopes.

$f : \text{Exp } A \rightarrow \text{Exp } A$

We'd like **f** to work on expressions containing arbitrary free variables.

At least, we'd like a closed such **f** to have the following semantics

$\llbracket f \rrbracket : \forall \Gamma. \text{Exp } \Gamma \ A \rightarrow \text{Exp } \Gamma \ A$

Just having $\llbracket \mathbf{f} \rrbracket : \forall \Gamma. \text{Exp } \Gamma \text{ A} \rightarrow \text{Exp } \Gamma \text{ A}$ is not enough.

First, there's no *hygiene*. $\mathbf{f}$ could inspect Γ in arbitrary ways, and e.g. return different expressions in empty and non-empty scopes.

Second, There's not even an obvious way to have *binders*. With STLC as an object language, we'd like at least

$$\mathbf{lam} : (\mathbf{Exp} \ A \ \rightarrow \ \mathbf{Exp} \ B) \ \rightarrow \ \mathbf{Exp} \ (A \Rightarrow B)$$

However, the standard indexed-set semantics demands

$$\llbracket \mathbf{lam} \rrbracket : \forall \Gamma. (\mathbf{Exp} \ \Gamma \ A \ \rightarrow \ \mathbf{Exp} \ \Gamma \ B) \ \rightarrow \ \mathbf{Exp} \ \Gamma \ (A \Rightarrow B)$$

where we can't conjure a fresh variable to apply the inner function to.

Explicit contexts

One approach: have explicit contexts in user-facing language.

$$\mathbf{f} : \forall \Gamma. \mathbf{Exp} \ \Gamma \ \mathbf{A} \rightarrow \mathbf{Exp} \ \Gamma \ \mathbf{A}$$

For example: Moebius, Cocon, layered modal TT, typed FOAS.

- We can do *arbitrary* intensional analysis.
- A lot of bureaucracy; ergonomics is a challenge.

Stability under substitution

$\llbracket f \rrbracket : \forall \Gamma. \text{Exp } \Gamma \text{ A} \rightarrow \text{Exp } \Gamma \text{ A}$

such that

$\forall (\Delta : \text{Scope}) (\sigma : \text{Sub } \Delta \Gamma) (a : \text{Exp } \Gamma \text{ A}). \llbracket f \rrbracket (a[\sigma]) = (\llbracket f \rrbracket a)[\sigma]$

A standard semantics for HOAS, logical frameworks & two-level calculi.

Barely any intensional analysis: no expression can be distinguished from a variable.

Easy faithful representation of object binders:

$\text{lam} : (\text{Exp } A \rightarrow \text{Exp } B) \rightarrow \text{Exp } (A \Rightarrow B)$

$\llbracket \text{lam} \rrbracket : \forall \Gamma. (\forall \Delta. \text{Sub } \Delta \Gamma \rightarrow \text{Exp } \Delta A \rightarrow \text{Exp } \Delta B) \rightarrow \text{Exp } \Gamma (A \Rightarrow B)$

Stability under variable renaming

A renaming from Γ to Δ maps every Δ -variable to a Γ -variable.

To faithfully represent binders, we need an extra *type of variables*, and e.g. have

lam : (Var A -> Exp B) -> Exp (A $\Rightarrow$ B)

Metaprograms can distinguish variables and non-variables, but e.g. can't decide equality of expressions...

... because two distinct variables can become equal by renaming.

Used in metatheory of TTs, but rarely in metaprogramming (?)

Stability under injective variable renaming

Equality of expressions can be decided.

But binders are much more fiddly: it's a *sub-structural* function type.

Example: *nominal sets*.

Example: the nullary parametric type theory (NPTT) of Muijder, Nuyts and Devriese¹.

¹Muijder, Nuyts, Devriese: *Nominal Type Theory by Nullary Internal Parametricity*

Stability under thinning

A thinning from Γ to Δ deletes zero or more binders from Γ , resulting in Δ .

Contravariant action: if $\sigma : \mathbf{Thinning} \ \Gamma \ \Delta$ and $e : \mathbf{Exp} \ \Delta \ A$, then $e[\sigma] : \mathbf{Exp} \ \Gamma \ A$.

It's the *weakest* stability requirement that makes sense for hygienic metaprogramming.

ThinLF

A mostly hypothetical language. Agda/Idris syntax with extra nominal features. Based on NPTT by Muylder, Nuyts & Devriese.

```
data Ty : Type where
```

```
  ι    : Ty
```

```
  _⇒_ : Ty → Ty → Ty
```

```
data Exp : Ty → NomType where
```

```
  var : Var (Exp A) → Exp A
```

```
  lam : (@ (Exp A) → Exp B) → Exp (A ⇒ B)
```

```
  app : Exp (A ⇒ B) → Exp A → Exp B
```

ThinLF

We can declare an inductive type to be nominal, and use **Var** and the nominal function type in the definition.

$$\frac{\Gamma \text{ context} \quad \Gamma \vdash A : \text{NomType}}{\Gamma, x @ A \text{ context}} \qquad \frac{\Gamma \vdash A : \text{NomType} \quad \Gamma, x @ A \vdash B : \text{Type}}{\Gamma \vdash (x @ A) \rightarrow B : \text{Type}}$$

$$\frac{\Gamma, x @ A \vdash t : B}{\Gamma \vdash \lambda x. t : (x @ A) \rightarrow B} \qquad \frac{\Gamma \vdash t : (x @ A) \rightarrow B}{\Gamma, y @ A, \Delta \vdash t y : B[y/x]}$$

$$\frac{\Gamma \vdash A : \text{NomType}}{\Gamma \vdash \text{Var } A : \text{Type}} \qquad \frac{}{\Gamma, x @ A, \Delta \vdash \#x : \text{Var } A}$$

ThinLF

Ordinary HOAS/LF programming is still available:

```
lam' : (Exp A → Exp B) → Exp (A ⇒ B)  
lam' f = lam λ x. f (var #x)
```

No flipping:

```
flip : (@ A → @ B → C) → (@ B → @ A → C)  
flip = λ f b a. f a b    -- ill-typed
```

No duplication:

```
dup : (@ A → @ A → B) → @ A → B  
dup = λ f a. f a a    -- ill-typed
```

Freshness

Fresh A x: the type of **A**-values that don't depend on **x**.

```
Fresh : {A : NomType} → Type → @ A → Type  
fresh : ((x @ A) → Fresh B x) → B  
<_,_> : B → (x @ A) → Fresh B x  
-- fresh and <_,_> are definitional inverses
```

We can match under a binder to extract a **Bool**:

```
extractBool : (@ A → Bool) → Bool  
extractBool f = fresh λ x. case f x of  
  True → <True, x>  
  False → <False, x>
```

Abstracting over free variables

Another primitive operation:

rebind : $((@ A \rightarrow B) \rightarrow @ A \rightarrow C) \rightarrow (@ A \rightarrow B \rightarrow C)$

(I only show the non-dependently typed version here).

We can use it to match on **Either** under a binder:

split : $(@ A \rightarrow \text{Either } B \ C) \rightarrow \text{Either } (@ A \rightarrow B) \ (@ A \rightarrow C)$

split *f* = **fresh** $\lambda x.$ **case** *f* *x* **of**

Left *b* $\rightarrow$ **rebind** $(\lambda b \ x. \langle \text{Left } b, x \rangle)$ *x* *b*

Right *c* $\rightarrow$ **rebind** $(\lambda c \ x. \langle \text{Right } c, x \rangle)$ *x* *c*

Some useful definitions

forget : {A : NomType}{B : Type} → (x @ A) → Fresh B x → B
forget = rebind (λ fr _. fresh fr)

map : (B → C) → (x @ A) → Fresh B x → Fresh C x
map f = rebind (λ b x. <f (fresh b), x>)

bind : (x @ A) → Fresh B x → (B → Fresh C x) → Fresh C x
bind x b k = k (forget x b)

Fresh' : {A : NomType}{B : Type} → B → @ A → Type
Fresh' b x = (b' : Fresh B x) × (forget b' = b)

Structure abstraction principles (SAP)

Most SAP-s from Muylder et al. work fine:

$$\begin{aligned} (@ A \rightarrow T) &\approx T \\ (@ A \rightarrow \perp) &\approx \perp \\ (@ A \rightarrow (B + C)) &\approx ((@ A \rightarrow B) + (@ A \rightarrow C)) \\ (@ A \rightarrow (B \times C)) &\approx ((@ A \rightarrow B) \times (@ A \rightarrow C)) \\ (@ A \rightarrow B \rightarrow C) &\approx ((@ A \rightarrow B) \rightarrow (@ A \rightarrow C)) \\ (@ A \rightarrow \text{List } B) &\approx \text{List } (@ A \rightarrow B) \\ ((x @ A) \rightarrow f \ x = g \ x) &\approx (f = g) \\ ((x @ A) \rightarrow \text{Fresh } B \ x) &\approx B \end{aligned}$$

This one requires name swapping, so we need to make it primitive (or generalize **fresh**):

$$((x @ A) \rightarrow \text{Fresh } (B \ x) \ y) \approx \text{Fresh } ((x @ A) \rightarrow B \ x) \ y$$

Example: counting variables

SAP-Nat : ($@ A \rightarrow \text{Nat}$) $\rightarrow \text{Nat}$

SAP-Nat f = fresh λx .

let go : Nat $\rightarrow$ Fresh Nat x

go Zero = <Zero, x>

go (Suc n) = map Suc x (go n)

go (f x)

countVars : Exp A $\rightarrow$ Nat

countVars (var _) = 1

countVars (app t u) = countVars t + countVars u

countVars (lam f) = SAP-Nat (λx . countVars (f x))

Example: recovering scoped syntax

Scope : Type

Scope = **List** ((**A** : **Ty**) × **Var** (**Exp** **A**))

inScope : (**Γ** : **Scope**) → {**A** : **Ty**} → **Exp** **A** → Type

inScope **Γ** (**var** {**A**} **x**) = (**A**, **x**) ∈ **Γ**

inScope **Γ** (**app** **t** **u**) = **inScope** **Γ** **t** × **inScope** **Γ** **u**

inScope **Γ** (**lam** **f**) = (**x** @ **Exp** **A**) → **inScope** (**Cons** (**A**, #**x**) **Γ**) (**f** **x**)

Note: **inScope** is decidable if we have decidable equality for **Var**.

Example: well-typed interpreter on scoped syntax

$\llbracket _ \rrbracket : \text{Ty} \rightarrow \text{Type}$

$\llbracket \mathbf{!} \rrbracket = \text{Nat}$

$\llbracket A \Rightarrow B \rrbracket = \llbracket A \rrbracket \rightarrow \llbracket B \rrbracket$

$\text{SAP-}\llbracket _ \rrbracket\text{-Ty} : (@ A \rightarrow \llbracket B \rrbracket) \rightarrow \llbracket B \rrbracket$

$\text{SAP-}\llbracket _ \rrbracket\text{-Ty} = \dots$

$\llbracket _ \rrbracket : \text{Scope} \rightarrow \text{Type}$

$\llbracket \text{Nil} \rrbracket = \tau$

$\llbracket \text{Cons } (A, _) \Gamma \rrbracket = \llbracket A \rrbracket \times \llbracket \Gamma \rrbracket$

$\llbracket _ \rrbracket : \{\Gamma A\} \rightarrow (t : \text{Exp } A) \rightarrow \text{inScope } \Gamma \ t \rightarrow \llbracket \Gamma \rrbracket \rightarrow \llbracket A \rrbracket$

$\llbracket \text{var } x \rrbracket \text{ xsc} \quad \gamma = \text{lookup } x \text{ xsc } \gamma$

$\llbracket \text{app } t \ u \rrbracket (\text{tsc}, \text{usc}) \gamma = (\llbracket t \rrbracket \text{ tsc } \gamma) (\llbracket u \rrbracket \text{ usc } \gamma)$

$\llbracket \text{lam } f \rrbracket \text{ fsc} \quad \gamma = \text{SAP-}\llbracket _ \rrbracket\text{-Ty } (\lambda x \alpha. \llbracket f \ x \rrbracket (\text{fsc } x) (\alpha, \gamma))$

Extra assumptions on variables

The semantics supports

- Decidable ordering on **Var**.
- Decidable comparison of a raw variable and a **Var**.

If we have these, we can define

- Decidable ordering (and equality) for **Exp**.
- General variable renaming as a recursive operation on **Exp**.
- Substitution as a recursive operation on **Exp**.
- Strengthening on **Exp** w.r.t. a raw variable, returning **Decide (Fresh (Exp A) x)**.

I'm not sure about the exact specification of variable operations.

ThinLF compared to Nullary Parametric TT

ThinLF has a lot of overlap with NPTT. What are the differences?

- ThinLF supports *indexed nominal datatypes*, where the indexing types are ordinary ThinLF types. NPTT only supports untyped nominal types in a straightforward way, and trying to have first-class indexing types seems rather complicated.
- NPTT supports swapping raw variables in instantiations, ThinLF doesn't.
- ThinLF supports decidable ordering of variables, NPTT is only compatible with decidable equality.

Conclusions

- We can do a lot in ThinLF, but I don't necessarily know how to do it efficiently and concisely.
- I'm hoping that we can build nice machinery on top of the primitives.
- Features like *pattern matching under binders* could be elaborated to ThinLF.
- I'm not quite sure about the set of primitives, or even the basic setup. Can we use “observational” binder types instead of manual SAP plugging?