A Pronominal Account of Binding and Computation

Robert Harper

Carnegie Mellon University

TAASN
March 2009

Thanks

Thanks to Daniel R. Licata and Noam Zeilberger, my collaborators on this work.

Thanks to the TAASN organizers for the invitation!

Overview

Goal: datatype mechanism with binding and computation.

- LF-like representations of syntactic objects with binding and scope.
- ML-like computation by structural induction (modulo renaming).
- Dependent families of types indexed by such objects.

Applications:

- Security-typed languages based on proof-carrying API's.
- Mechanized metatheory via total functional programming.

Overview

Main methods: polarization and contextualization.

- Distinguish positive from negative types.
- Manage binding and scope in the types.

Key idea: positive and negative function spaces.

- Negative = computational = admissible.
- Positive = representational = derivable.

Judgements and Evidence

Judgements are forms of assertion.

- e expr, $e : \tau$, *etc.*.
- Defined by a collection of rules.

Evidence for J is a derivation, ∇, composing rules.

- Abstract syntax trees, typing derivations, *etc.*.
- Write $\nabla : J$ to mean that ∇ is a derivation of J.

Derivability

The derivability judgement $J_1 \vdash J_2$ states that J_2 is derivable from the assumption J_1.

- Assumption is a local axiom.
- Evidence is a pattern, $a.\nabla$, consisting of evidence $\nabla : J_2$ involving the parameter $a : J_1$.
- Primitive rules are just assumed evidence for derivabilities.

In general, a rule

$$\frac{J_1 \quad \dots \quad J_n}{J}$$

is derivable iff $J_1, \dots, J_n \vdash J$.

Iterated Derivability

Left-iterated derivability $(J_1 \vdash J_2) \vdash J$ states that J is derivable from rule $J_1 \vdash J_2$.

- *cf.* Schroeder-Heister's definitional reflection
- Gives rise to higher-order rules (*cf.* LF representations).
- Evidence is a pattern with a parameter corresponding to the assumed rule.

Right-iterated derivability $J_1 \vdash (J_2 \vdash J_3)$ means $J_1, J_2 \vdash J_3$, with multiple assumptions.

Iterated Derivability

Higher-order rules:

$$\frac{A\,\mathsf{true} \vdash B\,\mathsf{true}}{A \supset B\,\mathsf{true}}$$

Expressed as a derivability,

$$(A\,\mathsf{true} \vdash B\,\mathsf{true}) \vdash A \supset B\,\mathsf{true}$$

Derivable rules:

$$(A\,\mathsf{true} \vdash B\,\mathsf{true}) \vdash (A \wedge C\,\mathsf{true} \vdash B \wedge C\,\mathsf{true})$$

Admissibility

The admissibility judgement $J_1 \models J_2$ states that evidence for J_1 may be transformed into evidence for J_2.

- Evidence is any (computable) function sending any $\nabla_1 : J_1$ to some $\nabla_2 : J_2$.
- Typically defined by pattern matching against derivations $\nabla_1 : J_1$ to obtain $\nabla_2 : J_2$ in each case.

A rule

$$\frac{J_1 \quad \ldots \quad J_n}{J}$$

is admissible iff $J_1, \ldots, J_n \models J$.

Admissibility

Admissibility, being implication, is structural:

- Reflexivity: $J \models J$.
- Transitivity: if $J_1 \models J_2$ and $J_2 \models J_3$, then $J_1 \models J_3$.
- Weakening: if $J_1 \models J$, then $J_1, J_2 \models J$.
- Contraction: if $J_1, J_1 \models J$, then $J_1 \models J$.
- Exchange: if $J_1, J_2 \models J$, then $J_2, J_1 \models J$.

These properties may be phrased as iterated admissibilities, *e.g.*,

$$(J_1 \models J) \models (J_1, J_2 \models J).$$

Admissibility

Admissibilities $J_1 \models J_2$ are not stable under rule extension!

- If $J_1 \models J_2$, then $J \models (J_1 \models J_2)$, but not $J \vdash (J_1 \models J_2)$.
- Why? Admissibility considers all derivations of antecedent.

Adding new rules disrupts evidence for admissibility.

- $(\mathbf{IL} \vdash \exists x.\phi\,\mathsf{true}) \models (\mathbf{IL} \vdash \phi(t)\,\mathsf{true})$ for some term t.
- But this fails for $\mathbf{CL} = \mathbf{IL} + \mathbf{LEM}$.

Admissibilities circumscribe the evidence for a judgement.

Admissibility

If all primitive rules are pure, then derivability is structural.

- Reflexivity: $J \vdash J$.
- Transitivity: $(J_1 \vdash J_2, J_2 \vdash J_3) \models (J_1 \vdash J_3)$.
- Weakening: $(J_1 \vdash J) \models (J_1, J_2 \vdash J)$.
- Contraction: $(J_1, J_1 \vdash J) \models (J_1 \vdash J)$.
- Exchange: $(J_1, J_2 \vdash J) \models (J_2, J_1 \vdash J)$.

Pure rules are those without side conditions, *i.e.*, without constraints on applicability.

Weakening

Evidence for weakening transforms derivations rule-by-rule.

$$\frac{\Gamma \vdash J_1 \quad \dots \quad \Gamma \vdash J_n}{\Gamma \vdash J}$$

That is, we pattern match on the last rule of $\nabla : \Gamma \vdash J$, and recursively transform premises and apply the same rule.

The validity of this argument depends on purity! The rule continues to apply after transformation of premises.

Weakening

Evidence for weakening transforms derivations rule-by-rule.

$$\frac{\Gamma\,\Gamma' \vdash J_1 \quad \dots \quad \Gamma\,\Gamma' \vdash J_n}{\Gamma\,\Gamma' \vdash J}$$

That is, we pattern match on the last rule of $\nabla : \Gamma \vdash J$, and recursively transform premises and apply the same rule.

The validity of this argument depends on purity! The rule continues to apply after transformation of premises.

Side Conditions

Side conditions on rules may be seen as admissibility premises.

- $\neg J$ is just $J \models \#$.
- Need not be negations, but this is a common case.

Side conditions may disrupt structural properties, *e.g.*,

$$\frac{\Gamma \vdash J_1 \quad \dots \quad \Gamma \vdash J_n \quad \Gamma \vdash \neg J}{\Gamma \vdash J}$$

Side Conditions

Side conditions on rules may be seen as admissibility premises.

- $\neg J$ is just $J \models \#$.
- Need not be negations, but this is a common case.

Side conditions may disrupt structural properties, *e.g.*,

$$\frac{\Gamma\,\Gamma' \vdash J_1 \quad \ldots \quad \Gamma\,\Gamma' \vdash J_n \quad \Gamma\,\Gamma' \not\models \neg J}{\Gamma\,\Gamma' \vdash J}$$

Derivability and Admissibility

Two notions of entailment:

- Derivability: introduced by patterns, eliminated by pattern matching.
- Admissibility: introduced by any computable transformation and eliminated by application.

Intermixing these leads to a general theory of rules that accounts for side conditions, and allows us to express meta-theoretic properties such as admissibility and derivability of rules.

It also generalizes higher-order abstract syntax, and typical syntactic operations such as substitution.

Polarized Types

Two views of the meaning of a logical connective:

- Verificationist: defined by introduction; elimination inverts introduction.
- Pragmatist: defined by elimination; introduction inverts elimination.

Operationally, these determine different connectives:

- Positive, or eager: values are compositions of patterns; elimination by pattern matching.
- Negative, or lazy: experiments are compositions of patterns; introduction by pattern matching.

Polarized Types

Positive type: natural numbers.

- Introduction: z, s(z), s(s(z)), ...
- Elimination:

$$\phi \text{ s.t. } \begin{cases} \mathsf{z} & \mapsto e_0 \\ \mathsf{s(z)} & \mapsto e_1 \\ \mathsf{s(s(z))} & \mapsto e_2 \\ \ldots \end{cases}$$

Crucially, elimination must cover all values!

Polarized Types

Negative type: infinite streams.

- Elimination: hd, tl.
- Introduction:

$$\sigma \text{ s.t. } \begin{cases} \mathsf{hd} & \mapsto e_0 \\ \mathsf{tl}; \mathsf{hd} & \mapsto e_1 \\ \mathsf{tl}; \mathsf{tl}; \mathsf{hd} & \mapsto e_2 \\ \ldots \end{cases}$$

Crucially, introduction must cover all experiments!

Polarized Types

Computational (ML, Coq) functions are negative:

- Introduced by defining response to an argument, not by internal structure.
- Eliminated by application to an argument value.

Computational functions are open-ended:

- Any mapping from domain to range is acceptable.
- Pragmatically, allows us to import functions from other systems.

Polarized Types

Representational (LF) functions are positive:

- Introduced by compositions of constructors, starting with variables.
- Eliminated by pattern matching, not application.

Representational functions are closed-ended:

- Cannot enrich with operations that analyze form of input.
- Essentially a value with (some/any) indeterminate.

Functions and Entailments

Positive (representational) functions witness derivability.

- Parameters are "fresh" axioms/assumptions.
- Body is a derivation schema with distinguished parameters.
- Generalizes higher-order abstract syntax.

Negative (computational) functions witness admissibility.

- Analyzes all possible derivations of antecedent.
- Computes a derivation for each possible argument.
- Captures meta-reasoning and meta-computation.

Types for Binding and Computation

Focusing (Andreoli, Girard, Zeilberger)

- Patterns mediate between focus and inversion.
- Positive: (right) focus = values, (left) inversion = matching.
- Negative: (left) focus = matching, (right) inversion = values.

Contextual Modality (Nanevski and Pientka)

- Object $M : \langle\Psi\rangle A$ has type A with parameters in Ψ.
- Supports pronominal account of derivability.
  - Parameters are pronouns, not nouns.
  - Specializes to binding and scope of identifiers.
- cf. pre-sheaf models of Plotkin, Tiuri, Fiori.

Types for Binding and Computation

Type structure (simplified):

Positive	A^+	$::=$	$\downarrow A^- \mid A_1^+ \otimes A_2^+ \mid A_1^+ \oplus A_2^+ \mid R^+ \Rightarrow A^+ \mid D$
Negative	A^-	$::=$	$\uparrow A^+ \mid A_1^+ \to A_2^-$
Rules	R	$::=$	$D \Leftarrow A^+$

Extensible pronominal data types, D, defined by rules.

- Higher-order rules: $D \Leftarrow (A_1^+ \Rightarrow A_2^+)$.
- Side conditions on rules: $D \Leftarrow \downarrow (\uparrow A_1^+ \to A_2^+)$.

Types for Binding and Computation

Rule contexts: $\Psi = u_1 : R_1, \ldots, u_n : R_n$.

- Each rule is represented by a parameter, u_i.
- Order matters: $\Psi \approx R_1 \times \cdots \times R_n$ (names are surface syntax.)
- Not necessarily structural (because rules need not be pure).

Judgements (simplified):

- Positive values: $\Gamma \vdash v^+ : \langle \Psi \rangle A^+$.
- Positive matches: $\Gamma \vdash k^+ : \langle \Psi_0 \rangle A^+ > \langle \Psi_1 \rangle B^-$.
- Neutral: $\Gamma \vdash e : \langle \Psi \rangle A$.

Pronominal Data Types

D introduction: create an instance of a rule.

$$\frac{u{:}D{\Leftarrow}A^{+} \in \Psi \quad \Gamma \vdash v^{+} : \langle\Psi\rangle A^{+}}{\Gamma \vdash u(v^{+}) : \langle\Psi\rangle D}$$

D elimination: pattern matching on all rules for D.

$$\frac{\begin{array}{c}\Gamma \vdash e : \langle\Psi\rangle D \\ (u{:}D{\Leftarrow}A^{+} \in \Psi) \longrightarrow \Gamma, x : \langle\Psi\rangle A^{+} \vdash e' : \langle\Psi'\rangle C\end{array}}{\Gamma \vdash \mathrm{case}\, e \,\{ \ldots u(x) \mapsto e' \ldots \} : \langle\Psi'\rangle C}$$

Positive Functions

Positive functions extend the rule context:

$$\frac{\Gamma \vdash v^+ : \langle \Psi, u : R \rangle A^+}{\Gamma \vdash \lambda^+ u.v^+ : R \Rightarrow \langle \Psi \rangle A^+}$$

Positive functions are eliminated by matching:

$$\frac{\Gamma \vdash e : \langle \Psi \rangle R \Rightarrow A^+ \quad \Gamma, x : \langle \Psi, u : R \rangle A^+ \vdash e' : \langle \Psi' \rangle C}{\Gamma \vdash \mathrm{case}\, e \,\{\, \lambda^+ u.x[u] \Rightarrow e' \,\} : \langle \Psi' \rangle C}$$

NB: parameters may or may not induce substitution functions!

Contextual Hypotheses

Variables in context are instantiated on use:

$$\frac{\Psi' \vdash \theta : \Psi}{\Gamma, x : \langle\Psi\rangle A \vdash x[\theta] : \langle\Psi'\rangle A}$$

Officially, Ψ is an ordered product: $\Psi = \Psi'$, $\theta = id$.

External syntax supports renamings of parameters (exchange, contraction) witnessed by θ.

Structural Properties

Structurality of Ψ is not assured (side conditions disrupt it).

- May not validate weakening = adding a new rule.
- May not validate substitution = deriving a rule.
- Always supports exchange (swapping of parameters).

Structurality must be programmed wherever needed.

- When rules are pure: generically definable.
- When subordination ensures that parameter is irrelevant.
- Admissibility witnessed by computational (negative) functions.

Subordination

A type A^+ is subordinate to a type B^+ (modulo Ψ) iff a value of type A may be used to construct a value of type B.

> *For example, nat might be subordinate to exp, but not vice versa.*

If A is not subordinate to B, then weakening by A cannot disrupt a side condtion that circumscribes B.

> *For example, a computational function on nat cannot be affected by adding parameters of type exp, but would be disrupted by a parameter of type nat.*

Example

A simple expression language:

$$e \quad ::= \quad \mathsf{num}[k] \mid e_1 \odot_f e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$$

Represented by rule context Ψ_{exp}:

$$\begin{aligned}
\mathsf{zero} &: \mathsf{nat} \\
\mathsf{succ} &: \mathsf{nat} \Leftarrow \mathsf{nat} \\
\mathsf{num} &: \mathsf{nat} \Leftarrow \mathsf{exp} \\
\mathsf{binop} &: \mathsf{exp} \Leftarrow \mathsf{exp} \Leftarrow (\mathsf{nat} \otimes \mathsf{nat} \rightarrow \mathsf{nat}) \Leftarrow \mathsf{exp} \\
\mathsf{let} &: \mathsf{exp} \Leftarrow \mathsf{exp} \Leftarrow (\mathsf{exp} \Rightarrow \mathsf{exp})
\end{aligned}$$

Example

We wish to define an evaluator for expressions:

$$\text{eval} : \langle \Psi_{\text{exp}} \rangle (\text{exp} \to \text{nat})$$

Match on argument x of type $\langle \Psi_{\text{exp}} \rangle \text{exp}$:

$$\begin{aligned} \text{num } n &\longmapsto n \\ \text{binop } e_1 \; f \; e_2 &\longmapsto f \; (\text{eval } e_1) \; (\text{eval } e_2) \\ \text{let } e_1 \; (\lambda u. e_2[u]) &\longmapsto \text{eval}(\text{subst} \; (\lambda u. e_2[u]) \; e_1) \end{aligned}$$

Example

The function subst witnesses admissibility of transitivity.

- Realizing $\Psi_{\mathsf{exp}}, u : \mathsf{exp}$ in Ψ.
- Definable because exp not subordinate to nat.

By contrast we cannot substitute for, say, z in an exp!

- Binary operation f analyzes each value of type nat.
- Cannot expect f to be stable under substitution.

Normalization by Evaluation

Define context Ψ_{nbe} for syntax and semantics.

$$\begin{array}{l} \texttt{app} : \texttt{exp} \Leftarrow (\texttt{exp} \otimes \texttt{exp}) \\ \texttt{lam} : \texttt{exp} \Leftarrow (\texttt{exp} \Rightarrow \texttt{exp}) \end{array}$$

$$\begin{array}{l} \texttt{napp} : \texttt{neu} \Leftarrow (\texttt{neu} \otimes \texttt{sem}) \\ \texttt{neut} : \texttt{sem} \Leftarrow \texttt{neu} \\ \texttt{slam} : \texttt{sem} \Leftarrow (\forall\ (\Psi \in \texttt{neu*}).\ \Psi \Rightarrow \texttt{sem} \rightarrow \texttt{sem}) \end{array}$$

In what follows Ψ consists of parameters of types exp and neu.

Normalization by Evaluation

The function `eval` has type

$$\langle \Psi_{\mathsf{nbe}} \rangle \ \ \forall \ \ \Psi \ \ \Psi \ \Rightarrow \ (\texttt{exp} \ \longrightarrow \ (\texttt{exp} \ \texttt{\#} \ \longrightarrow \ \texttt{sem}) \ \longrightarrow \ \texttt{sem})$$

Spelled out, this means that

Normalization by Evaluation

The function `eval` has type

$\langle\Psi_{\mathsf{nbe}}\rangle \;\; \forall\; \Psi \;\; \Psi \;\; \Rightarrow \; (\mathtt{exp} \;\rightarrow\; (\mathtt{exp}\ \#\;\rightarrow\; \mathtt{sem}) \;\rightarrow\; \mathtt{sem})$

Spelled out, this means that

- in context Ψ_{nbe} ...

Normalization by Evaluation

The function `eval` has type

$$\langle \Psi_{\mathsf{nbe}} \rangle \;\; \forall\; \Psi \;\; \Psi \;\; \Rightarrow \; (\texttt{exp} \rightarrow (\texttt{exp}\ \texttt{\#} \rightarrow \texttt{sem}) \rightarrow \texttt{sem})$$

Spelled out, this means that

- in context Ψ_{nbe} ...
- in any extension by `neu` and `exp` parameters ...

Normalization by Evaluation

The function `eval` has type

$\langle \Psi_{\mathsf{nbe}} \rangle \;\; \forall \;\; \Psi \;\; \Psi \;\; \Rightarrow \; (\mathtt{exp} \; \longrightarrow \; (\mathtt{exp} \; \# \; \longrightarrow \; \mathtt{sem}) \; \longrightarrow \; \mathtt{sem})$

Spelled out, this means that

- in context Ψ_{nbe} ...
- in any extension by `neu` and `exp` parameters ...
- given an expression and ...

Normalization by Evaluation

The function `eval` has type

$\langle \Psi_{\mathsf{nbe}} \rangle \;\; \forall \;\; \Psi \;\;\; \Psi \;\; \Rightarrow \; (\mathtt{exp} \; \rightarrow \; (\mathtt{exp} \; \# \; \rightarrow \; \mathtt{sem}) \; \rightarrow \; \mathtt{sem})$

Spelled out, this means that

- in context Ψ_{nbe} ...
- in any extension by `neu` and `exp` parameters ...
- given an expression and ...
- a mapping of `expr` variables to semantic values ...

Normalization by Evaluation

The function `eval` has type

$$\langle \Psi_{\mathsf{nbe}} \rangle \;\; \forall \;\; \Psi \quad \Psi \;\; \Rightarrow \;\; (\texttt{exp} \;\rightarrow\; (\texttt{exp}\;\#\;\rightarrow\; \texttt{sem}) \;\rightarrow\; \texttt{sem})$$

Spelled out, this means that

- in context Ψ_{nbe} ...
- in any extension by `neu` and `exp` parameters ...
- given an expression and ...
- a mapping of `expr` variables to semantic values ...
- `eval` yields a semantic value.

Normalization by Evaluation

The function `reify` has type

$\langle \Psi_{\mathsf{nbe}} \rangle \, \forall \; \Psi \, \Psi \;\; \Rightarrow \; (\texttt{sem} \; \rightarrow \; (\texttt{exp} \; \texttt{\#} \; \rightarrow \; \texttt{neu} \; \texttt{\#}) \; \rightarrow \; \texttt{exp})$

That is,

Normalization by Evaluation

The function `reify` has type

$$\langle \Psi_{\mathsf{nbe}} \rangle\ \forall\ \Psi\ \Psi\ \Rightarrow (\mathtt{sem} \rightarrow (\mathtt{exp}\ \#\ \rightarrow \mathtt{neu}\ \#) \rightarrow \mathtt{exp})$$

That is,

- in context Ψ_{nbe} ...

Normalization by Evaluation

The function `reify` has type

$\langle \Psi_{\mathsf{nbe}} \rangle \,\forall\ \psi\ \psi\ \Rightarrow \texttt{(sem} \rightarrow \texttt{(exp \#} \rightarrow \texttt{neu \#)} \rightarrow \texttt{exp)}$

That is,

- in context Ψ_{nbe} ...
- in any extension with `neu` and `expr` parameters ...

Normalization by Evaluation

The function `reify` has type

$\langle \Psi_{\mathsf{nbe}} \rangle \, \forall \;\; \Psi \; \Psi \;\; \Rightarrow \; (\texttt{sem} \; \rightarrow \; (\texttt{exp} \; \texttt{\#} \; \rightarrow \; \texttt{neu} \; \texttt{\#}) \; \rightarrow \; \texttt{exp})$

That is,

- in context Ψ_{nbe} ...
- in any extension with `neu` and `expr` parameters ...
- given a semantic value and ...

Normalization by Evaluation

The function `reify` has type

$\langle\Psi_{\mathsf{nbe}}\rangle\,\forall\ \Psi\ \Psi\ \Rightarrow\ (\mathtt{sem}\ \rightarrow\ (\mathtt{exp}\ \#\ \rightarrow\ \mathtt{neu}\ \#)\ \rightarrow\ \mathtt{exp})$

That is,

- in context Ψ_{nbe} ...
- in any extension with `neu` and `expr` parameters ...
- given a semantic value and ...
- a mapping from syntactic to semantic variables ...

Normalization by Evaluation

The function `reify` has type

$$\langle \Psi_{\mathsf{nbe}} \rangle \; \forall \;\; \Psi \; \Psi \;\; \Rightarrow \; (\mathtt{sem} \; \rightarrow \; (\mathtt{exp} \; \# \; \rightarrow \; \mathtt{neu} \; \#) \; \rightarrow \; \mathtt{exp})$$

That is,

- in context Ψ_{nbe} ...
- in any extension with `neu` and `expr` parameters ...
- given a semantic value and ...
- a mapping from syntactic to semantic variables ...
- `reify` yields an expression.

Evaluation

```
eval : ∀ Ψ. Ψ  ⇒ (exp → (exp # → sem) → sem)
eval[Ψ] x            σ =  σ x
eval[Ψ] app(e1,e2)   σ =  appsem (eval[Ψ] e1 σ)
                                 (eval[Ψ] e2 σ)
eval[Ψ] lam(λx.e[x]) σ = slam φ where φ = ...

appsem : ∀ Ψ. Ψ  ⇒ (sem → sem → sem)
appsem[Ψ] slam(φ) s2 = φ [·] s2
appsem[Ψ] neut(n) s2 = neut(napp(n , s2))
```

Evaluation

The semantic function φ is defined as follows:

```
φ : < Ψ > (∀ (Ψ′ ∈ neu*). Ψ′ ⇒ sem → sem)

φ[Ψ′] s' = strengthen x from
    (eval[Ψ, x:exp , Ψ′] (weaken e[x] with Ψ′) σ′)
  where
   σ′ : < Ψ, x:exp , Ψ′ > (exp # → sem)
   σ′ x          = weaken s' with x
   σ′ (y ∈ Ψ) = weaken (σ y) with (x,Ψ′)
```

Evaluation

The definition of φ uses auxiliaries `strengthen` and `weaken`.

- `weaken` is a computational function that weakens with respect to a fresh parameter of type `exp`.
- `strengthen` uses subordination to remove parameter of type `exp` in result of type `sem`.

These are type-generic programs that are generated automatically, when they exist.

Reification

```
reify : ∀ Ψ. Ψ ⇒ (sem → (exp # → neu #) → exp)

reify[Ψ] neut(n) σ = reifyn[Ψ] n σ
reify[Ψ] slam(φ) σ =
 lam (λx.
       strengthen y from
         (reify[Ψ, y:neu , x:exp]
           (weaken (φ [y:neu] neut(y)) with x)
           σ′))
 where
  σ′ : < Ψ, y:neu , x:exp > exp # -> neu #
  σ′ x           = y
  σ′ (x' ∈ Ψ′) = weaken (σ x) with [x , y]
```

Semantic Application

$\texttt{reifyn : } \forall\ \Psi.\ \Psi \Rightarrow (\texttt{neu} \rightarrow (\texttt{exp \#} \rightarrow \texttt{neu \#}) \rightarrow \texttt{exp})$

```
reifyn[Ψ] x         σ = σ x
reifyn[Ψ] napp(n,s) σ = napp (reifyn[Ψ] n σ ,
                              reify [Ψ] s σ)
```

Summary

A pronominal approach to binding and computation:

- Names are pronouns (references), not nouns (objects).
- Avoids reliance on state, or associated logics of purity.
- Captures central concepts of judgements-as-types, including higher-order abstract syntax.
- Admits precise types for admissibilities.

But there is a cost for expressiveness and generality:

- If impurities are admitted, admissibilities are not assured.
- Expressing more precise types takes real work.
- Extension to dependent computation and representation types?

Ongoing and Future Work

Implementation.

- Implemented as a universe within Agda (see my web page).
- Designing an external language with elaboration for named form.

Positive Dependent Types [LH PLPV09]

- Admit $\Pi x : A_1^+ . A_2^-$ (negative) and $\Sigma x : A_1^+ . A_2^+$ (positive).
- Avoids testing equivalence of negative values.
- Relies on simultaneous induction-recursion.

Richer Rule Formalisms

- Pure dependent LF, without side conditions.
- Impure LF: how to intermix dependency and side conditions?

Thank You!

Questions?