A Nominal Logical Framework

Logic and Semantics Club
January 20, 2006

James Cheney

University of Edinburgh
A logical framework is a formal system for defining (and reasoning about) other formal systems.

Typically, employs a dependent type theory to represent syntax and judgments of a logic or language

Idea goes back to at least de Bruijn’s AUTOMATH project

(Edinburgh) LF, Calculus of Constructions, and variants have proven very expressive and powerful.

Fertile area for interesting type theories (Linear LF, Ordered LF, Concurrent LF; Inductive Constructions)
Do we really need another LF?

Existing LF family very expressive, but:

1. Names are “second-class”: difficult to encode judgments based on \textit{name-inequality}

2. Inductive (meta-)reasoning apparently must be performed externally; co-induction, logical relations not well understood

3. Encoding generativity (e.g. allocation semantics) requires resorting to counters/linearity

Eventual goal: handle all of these concerns; today focus on (1) and sketch (2), (3)
Another motivation

- Nominal logic: a recently proposed technique for “near first-order” reasoning about names & binding; basis for $\alpha$Prolog

- Related theories: Miller & Tiu’s $FO\lambda^\nabla$, higher-order patterns, Pfenning, Pientka and Nanevski’s contextual modal type theory

- Goal: Identify a common core type theory unifying the basic ideas underlying these systems

- thus providing a constructive foundation for nominal techniques

- and hopefully getting certain type theorists off my case...
Syntax

- Extends LF

\[
K ::= \text{type} \mid \Pi x:A.K \mid \text{name}
\]
\[
A, B ::= a \mid A.M \mid \Pi x:A.B \mid \forall a:A.B
\]
\[
M, N ::= c \mid x[\rho] \mid \lambda x:A.M \mid M \cdot N \mid a \mid \langle a \rangle M \mid M @ a
\]
\[
\rho ::= \text{id} \mid \rho, a/b
\]

Note: *Names* \(a, b, \ldots\) are a new syntactic class similar to (but disjoint from) variables

- \(\rho\): *explicit renamings*

- \(\forall a:A.B, \langle a : A \rangle M\), subject to \(\alpha\)-equivalence

- \(\alpha\)-equivalence & capture-avoiding substitution/renaming operations defined in traditional way for LF
Contexts

- Contexts: as LF, but with extra *name contexts* $\sigma$.
- Also, types in $\Gamma$ are guarded by name-context.

$$
\Sigma ::= \cdot \mid c : A | a : K
\Gamma ::= \cdot \mid x : A[\sigma]
\sigma ::= \cdot \mid a : A
$$

- Well-formedness for contexts: as LF, but types in $\sigma$ must be of kind *name* (and cannot depend on names).

$$
\Gamma \vdash \sigma \text{ nctx} \quad \Gamma \vdash \cdot \triangleright A : \text{name}
\quad \frac{\Gamma \vdash \sigma, a : A \text{ nctx}}{\Gamma \vdash \sigma, a : A \text{ nctx}}
$$

A Nominal Logical Framework – p.6/28
Judgments

Judgments: as LF, but with extra name-context.

<table>
<thead>
<tr>
<th>Judgment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vdash \Sigma \text{sig}$</td>
<td>Signature formation</td>
</tr>
<tr>
<td>$\vdash \Gamma \text{ctx}$</td>
<td>Context formation</td>
</tr>
<tr>
<td>$\Gamma \vdash \sigma \text{nctx}$</td>
<td>Name context formation</td>
</tr>
<tr>
<td>$\Gamma \vdash \sigma \triangleright K : \text{kind}$</td>
<td>Kind formation</td>
</tr>
<tr>
<td>$\Gamma \vdash \sigma \triangleright A : K$</td>
<td>Type family formation</td>
</tr>
<tr>
<td>$\Gamma \vdash \sigma \triangleright M : A$</td>
<td>Object formation</td>
</tr>
<tr>
<td>$\Gamma \vdash \sigma \triangleright \rho : \sigma'$</td>
<td>Renaming formation</td>
</tr>
<tr>
<td>$\Gamma \vdash \sigma \triangleright K = K' : \text{kind}$</td>
<td>Kind equality</td>
</tr>
<tr>
<td>$\Gamma \vdash \sigma \triangleright A = B : K$</td>
<td>Type family equality</td>
</tr>
<tr>
<td>$\Gamma \vdash \sigma \triangleright M = N : A$</td>
<td>Object equality</td>
</tr>
<tr>
<td>$\Gamma \vdash \sigma \triangleright \rho = \rho' : \sigma'$</td>
<td>Renaming equality</td>
</tr>
</tbody>
</table>
The rules for variables and renamings:

\[
\begin{align*}
\Gamma \vdash \sigma' \triangleright \rho : \sigma \\
\Gamma, x : A[\sigma] \vdash \sigma' \triangleright x[\rho] : A[\rho]
\end{align*}
\]

Variables have to be instantiated with appropriate local names.

\[
\Gamma \vdash \sigma \triangleright \text{id} : \cdot
\]

\[
\Gamma \vdash \sigma \triangleright \rho : \sigma' \\
\Gamma \vdash \sigma, b : A \triangleright \rho, a/b : \sigma', a : A
\]

Renamings must be one-to-one.
Typing: highlights

The rules for names, abstractions, and concretions:

\[ \Gamma \vdash \sigma, a : A \Rightarrow a : A \]

\[ \Gamma \vdash \sigma, a : A \Rightarrow M : B \]

\[ \Gamma \vdash \sigma \Rightarrow \langle a: A \rangle M : \forall a : A.B \]

\[ \Gamma \vdash \sigma \Rightarrow M : \forall a : A.B \]

\[ \Gamma \vdash \sigma, a : A \Rightarrow M @ a : B \]

Fresh name \( a \) added to the context in \( \langle a : A \rangle M \) rule.

Names *removed from* the context in \( M @ a \) rule.

Therefore, \( \langle a : A \rangle \lambda x : \forall a : A. \forall b : A. x @ a @ a \) is not typable.

However, \( \lambda x : A \rightarrow A \rightarrow B. \langle a : A \rangle x a a \) is OK.
β-reduction and η-expansion (extensionality) for Π-types/λ-terms as in LF

For Π-types, we have

\[(β) \quad (\langle a:A \rangle M) @ b = M[b/a] \quad (b \notin FV(M))\]

\[(η) \quad \langle a:A \rangle (M @ a) = M : ∀a:A.B \quad (a \notin FV(M))\]

Standard, but note that β-rule can never identify two different names.
Local reductions/expansions

The following *local* soundness and completeness properties for $\mathcal{N}$ are important checks that the system is sensible:

\[
\begin{align*}
\frac{\vdash \sigma, a:A \triangleright M : B}{\Gamma \vdash \sigma \triangleright (\langle a:A \rangle M) : \forall a:A. B} \\
\frac{\vdash \sigma \triangleright (\langle a:A \rangle M) \circ b : B[b/a]}{\Gamma \vdash \sigma, b:A \triangleright (\langle a:A \rangle M) @ b : B[b/a]} \\
\Downarrow_{\beta} \\
\frac{D[b/a]}{\Gamma \vdash \sigma, b:A \triangleright M[b/a] : B[b/a]}
\end{align*}
\]
The following *local* soundness and completeness properties for $\mathcal{N}$ are important checks that the system is sensible:

\[
\begin{align*}
\Gamma \vdash \sigma \triangleright M : \forall \alpha : A.B \\
\Downarrow \eta \\
\Gamma \vdash \sigma \triangleright M : \forall \alpha : A.B
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \sigma, \alpha : A \triangleright M @ \alpha : B \\
\hline \\
\Gamma \vdash \sigma \triangleright \langle \alpha : A \rangle M @ \alpha : \forall \alpha : A.B
\end{align*}
\]
Simple example

As an example, consider lambda term typing encoded in NLF:

\[
\begin{align*}
wf & : \text{ctx} \rightarrow \text{exp} \rightarrow \text{ty} \rightarrow \text{type}. \\
wf_{\text{var}} & : \text{wf} \ G \ (\text{var} \ V) \ T \leftarrow \text{lookup} \ G \ V \ T. \\
wf_{\text{app}} & : \text{wf} \ G \ (\text{app} \ E_1 \ E_2) \ U \\
& \quad \leftarrow \text{wf} \ G \ E_1 \ (\text{arr} \ T \ U) \leftarrow \text{wf} \ G \ E_2 \ T. \\
wf_{\text{lam}} & : \text{wf} \ G \ (\text{lam} \ M) \ (\text{arr} \ T \ U) \\
& \quad \leftarrow \forall a. \text{wf} \ [G, (a, T)] \ (M @ a) \ U.
\end{align*}
\]

- Contexts are just lists.
- Note that we do not use implication for local hypotheses (and it would be incorrect to do so).
Encoding hypotheses

A distinctive feature of LF is the higher-order encoding of hypothetical judgments:

\[ wf_{\text{lam}} : wf \ (\text{lam} \ M) \ (\text{arr} \ T \ U) \]
\[ \quad \leftarrow \ \Pi x. (wf \ x \ T \ \rightarrow \ wf \ (M \ x) \ U). \]

This is nifty because (intuitionistic) object language contexts “disappear” into LF’s (intuitionistic) context.

This \textit{does not work} using \( \forall \) in nominal logic or NLF!

\[ wf_{\text{lam}} : wf \ (\text{lam} \ M) \ (\text{arr} \ T \ U) \]
\[ \quad \leftarrow \ \forall a. (wf \ (\text{var} \ a) \ T \ \rightarrow \ wf \ (M @ a) \ U). \]

Why?
Encoding hypotheses

- Why does this not work?
- Local hypotheses are “lifted” out of their name context.
- So the following is derivable:

\[
\begin{align*}
  x : \text{wf} (\text{var } a) & \quad t[a, b] \vdash a, b \triangleright x[b/a, a/b] : \text{wf} (\text{var } b) \quad t \\
  \vdash a, b \triangleright \lambda x.x[b/a, a/b] : \Pi x : \text{wf} (\text{var } a) \; t.\text{wf} (\text{var } b) \; t \\
  \cdot \vdash \cdot \triangleright \langle b \rangle \lambda x.x[b/a, a/b] : \Pi b.\Pi x : \text{wf} (\text{var } a) \; t.\text{wf} (\text{var } b) \; t \\
  \cdot \vdash \cdot \triangleright \langle a \rangle \langle b \rangle \lambda x.x[b/a, a/b] : \Pi a.\Pi b.\Pi x : \text{wf} (\text{var } a) \; t.\text{wf} (\text{var } b)
\end{align*}
\]

- Bad!
- This problem is similar to the kind caused by *equivariance* in nominal logic.
Another example: Closure conversion

- Important FP compilation phase
- Idea: Make all functions closed
- Translate functions to (closed function, environment) pair
- Environment shape depends on context:
  \[ \Gamma = x_n, \ldots, x_1 \mapsto env = \langle v_n, \langle v_{n-1}, \ldots, v_1 \rangle \cdots \rangle \]
- Need ability to test equality and inequality of names.
Closure conversion, informally

A typical “paper” presentation

\[
C[[\Gamma, x \vdash x]]e = \pi_1(e) \\
C[[\Gamma, x \vdash y]]e = C[[\Gamma \vdash y]]\pi_2(e) \quad (x \neq y) \\
C[[\Gamma \vdash e_1 e_2]]e = \text{let } c = C[[\Gamma \vdash e_1]] \\
\quad \text{in } (\pi_1(c)) \ (C[[\Gamma \vdash e_2]]e, \pi_2(e)) \\
C[[\Gamma \vdash \lambda x.e_0]]e = \langle \lambda y. C[[\Gamma, x \vdash e_0]]y, e \rangle \quad (y \not\in FV(\Gamma, x, e, e_0))
\]

Inequality side conditions: non-obvious how to encode in LF.
Closure conversion in NLF

This is no problem in NLF.

Use $\mathcal{N}$-quantifier; $\# \text{ defined in terms of } \mathcal{N}$.

$cconv$ : $\text{list id} \to \text{exp} \to \text{exp} \to \text{exp} \to \text{type}$.

$cconv_{\text{var2}}$ : $cconv \left[ G, X \right] \left( \text{var} \ Y \right) \text{Env} \ E$

$\leftarrow X \# Y$

$\leftarrow cconv \ G \left( \text{var} \ Y \right) \left( \text{pi}_2 \ \text{Env} \right) \ E.$

$cconv_{\text{lam}}$ : $cconv \ G \left( \text{lam} \ F_1 \right) \text{Env} \ (\text{pair} \left( \text{lam} \ F_2 \right) \text{Env})$

$\leftarrow \mathcal{N}_\xi.\mathcal{N}_\eta.\ cconv \left[ G, \xi \right] \left( F_1 @ \xi \right) \left( \text{var} \ \eta \right) \left( F_2 @ \eta \right).$
This is the best I can do (there may be a better way...)

Idea: Maintain a list $L$ of “bound” variables; ensure that hypotheses $\text{neq } X Y$ are derivable whenever $X, Y$ are distinct elements of list.

$$cconv : \text{list id} \rightarrow \text{list id} \rightarrow \text{tm} \rightarrow \text{tm} \rightarrow \text{tm} \rightarrow \text{type}.$$ 

$$cconv\_\text{var2} : cconv L [G, X] (\text{var } Y) \text{ Env } E \leftarrow \text{neq } X Y$$
$$\leftarrow cconv L G (\text{var } Y) (\text{pi2 } \text{Env}) E.$$
Closure conversion in LF

Tricky part: Use “distinctness” predicate to encode $2|L|$ inequalities compactly.

\[
\begin{align*}
\text{distinct} & : \text{id} \rightarrow \text{list id} \rightarrow \text{type.} \\
\text{neq}_1 & : \text{neq X Y} \leftarrow \text{distinct X L} \leftarrow \text{member Y L} \\
\text{neq}_2 & : \text{neq X Y} \leftarrow \text{distinct Y L} \leftarrow \text{member X L} \\
\text{cconv_lam} & : \text{cconv L G (lam F}_1\text{) Env (pair (lam F}_2\text{) Env)} \\
& \leftarrow \Pi x. \Pi y. \text{distinct x L} \rightarrow \\
& \text{cconv [L,x] [G,x] (F}_1\text{ x) (var y) (F}_2\text{ y)}
\end{align*}
\]
Formal properties

- Weakening, substitution: standard
- Injective Renaming (but not general renaming) of name-contexts:

**Lemma 1.** If $\Gamma \vdash \sigma \triangleright \rho : \sigma'$ and $\Gamma \vdash \sigma \triangleright M : A$ then $\Gamma \vdash \sigma' \triangleright M[\rho] : A[\rho]$.

- Subject Reduction
- Church-Rosser, Strong Normalization: standard, reduction to LF
- Decidability of typechecking: Extensions of proofs by [Goguen 05] or [Harper and Pfenning 05]
- Warning: Currently revising system, still need to check details.
Extensions: induction/coinduction

- Judgments can be defined without recourse to recursion through negative type occurrences
- Hence, supporting “internal” induction/coinduction using standard type theoretic methods ought to be straightforward.

\[
\begin{align*}
\Gamma \vdash \sigma \triangleright M : A[\mu X.A[X]] & \quad \Gamma \vdash \sigma \triangleright M : \mu X.A[X] \\
\Gamma \vdash \sigma \triangleright \text{fold } M : \mu X.A[X] & \quad \Gamma \vdash \sigma \triangleright \text{unfold } M : A[\mu X.A[X]]
\end{align*}
\]

- However, this departs significantly from the “traditional” LF methodology...
- [Momigliano and Tiu 2003] approach can probably be used
Extensions: generativity

- The ability to reason about “fresh for world” name generation is a key aspect of nominal logic.

- Core NLF doesn’t support it: if \( a \not\in FN(B) \), then \( NL \vdash \forall a.B \supset B \) but \( NLF \nvdash M : \forall a.B \rightarrow B \).

- Why? The following derivation attempt is stuck:

\[
\Gamma \vdash \sigma, a : A \triangleright M : B \\
\Gamma \vdash \sigma \triangleright \forall a:A.M : B
\]

- A possible solution: Add a “fresh name choice” proof term \( \nu a:A.M \)

\[
\Gamma \vdash \sigma, a : A \triangleright M : B \\
\Gamma \vdash \sigma \triangleright \nu a:A.M : B
\]

- Equational theory, formal properties seem challenging
What if we vary the allowed renamings?

- Existing name-contexts: allow *weakening*, *exchange*, *injective renaming* but not *contraction* or general substitution.
- If we *limit* renamings so that reordering and weakening are forbidden, we get something like core $FO\lambda^\nabla$.
- If we *relax* renamings so that contraction/arbitrary substitutions are allowed, then we get something like Binding Algebras.
- Both variations might be interesting!
From a type-proof-theoretic point of view, proving that proofs have normal forms and typechecking is decidable is generally enough.

From this follows consistency, adequacy, other key properties.

But it would be nice to work out the semantics of NLF, relate to semantics of NL, $FO\lambda^\forall$, etc.

Probably easier for the $\lambda$-free fragment...
NLF in context

Nominal techniques rely on bijective renamings; NLF’s injective renamings can always be extended to bijective ones (and seem to be more compatible with type theoretic setting).

Rules for $\forall$ type are *self-dual*, as in NL

They also correspond to natural deduction forms of the $\forall$-quantifier rules from $FO\lambda^\forall$:

\[
\begin{align*}
\Sigma : \Gamma, (\sigma, x) \triangleright A & \Rightarrow C & \Sigma : \Gamma & \Rightarrow (\sigma, x) \triangleright A \\
\Sigma : \Gamma, \sigma \triangleright \forall x.A & \Rightarrow C & \Sigma : \Gamma & \Rightarrow \sigma \triangleright \forall x.A
\end{align*}
\]

Well-formedness restrictions on concretion terms correspond to *higher-order pattern restriction* familiar in higher-order unification
Future work

- Integrating full induction/coinduction (following Momigliano and Tiu?)
- Generativity?
- Implementation/translation to basic LF?
- Semantics?!
Conclusions

- Higher-order techniques underlying traditional LF powerful, but have some limitations.
- Using ideas drawn from a number of sources, we’ve seen how “first-class” name-inequality can be supported in a nominal extension of LF.
- Next steps: induction, generativity.
- Still lots to do.