Towards a General Theory of Names, Binding, and Scope

James Cheney

September 30, 2005
“You can have any color car you like, as long as it is black.”
[Henry Ford]
The gap

- High-level formalisms (higher-order, nominal, theory of contexts, de Bruijn, etc.) typically *bind one name at a time*, and *its scope is a subtree adjacent to the binding occurrence.*
  - Call this form of scoping *unary lexical scoping* (ULS)

- Real logics, programming languages display other forms of scoping that do not fit this mold
  - Non-lexical scoping (scope is not an adjacent subtree)
  - Global scope and unique definitions
  - Anonymity
  - Simultaneous binding (e.g., patterns, letrec)
Is this really a problem?

- True, ULS can be used to simulate all of the above

- But, encodings are not always *adequate*; there may be “junk” terms or “confusion” terms

- Moreover, translation apparently cannot be formalized in the meta-logic, but must be done “on paper”

- But “elaboration” translations from, e.g., letrec + patterns to fix + case are often *not* trivial.

- **Claim**: Gap between formalisms and real languages hinders adoption by non-experts.

- **This paper**: Show how to capture such approaches *adequately* within nominal logic
**Our approach**

- In nominal logic, ULS is not “built-in”, but “definable”.

- Other forms of binding are also definable.

- Program: Investigate four classes of more exotic binding situations and show how to axiomatize them in NL.
  - Pseudo-unary scoping
  - Global/unique scoping
  - Anonymity
  - Simultaneous binding (patterns)
What’s special about nominal logic?

• My feeling: NL’s explicit treatment of names as data makes it more flexible for talking about non-ULS binding.

• This is just a feeling.

• It’s entirely possible that the same ideas/tricks are sensible in other approaches, but I don’t see how.

• Reverse psychology, anyone?
Nominal Logic

- Nominal logic [Pitts 2003] is a extension of FOL that axiomatizes:
  - *names* a, b ∈ A,
  - *swapping* (i.e. invertible renaming) (a b) ∙ x,
  - *freshness* (the “not free in” relation”) a # x,
  - a *name-abstraction* operation ⟨a⟩x providing unary lexical scoping.
- Terms
  $$t ::= a \mid f(\overline{t}) \mid c \mid \langle a \rangle t$$
- Types
  $$\tau ::= \nu \mid \delta \mid \langle \nu \rangle \tau$$
  \(\nu\): name types, \(\delta\): data types
Nominal equational logic

- **Well-formedness**

\[
\begin{align*}
\frac{a : \nu \in \Sigma}{a : \nu} & \quad \frac{c : \tau \in \Sigma}{c : \tau} & \quad \frac{a : \nu \quad t : \tau}{\langle a \rangle t : \langle \nu \rangle \tau} \\
\end{align*}
\]

\[
\begin{align*}
t_i : \tau_i & \quad f : (\tau_1, \ldots, \tau_n) \rightarrow \delta \in \Sigma \\
\end{align*}
\]

\[
\begin{align*}
f(\overline{t}) & : \delta
\end{align*}
\]

- **Swapping** \((\pi : A \rightarrow A \text{ a permutation})\)

\[
\begin{align*}
\pi \cdot a & = \pi(a) \\
\pi \cdot c & = c \\
\pi \cdot f(\overline{t}) & = f(\pi \cdot \overline{t}) \\
\pi \cdot \langle a \rangle t & = \langle \pi \cdot a \rangle \pi \cdot t
\end{align*}
\]
**Nominal equational logic**

- **Freshness**

\[
\begin{align*}
(a \neq b) & \quad \qquad a \neq c \\
\therefore a \neq b & \quad \quad a \neq t_i & (i = 1, \ldots, n) \\
& \quad \quad a \neq f(t_1, \ldots, t_n) \\
& \quad \quad a \neq \langle b \rangle t & \quad \quad a \neq \langle a \rangle t
\end{align*}
\]

- **Equality**

\[
\begin{align*}
\overline{a \approx a} & \quad \quad \overline{c \approx c} \\
& \quad \quad \quad \overline{f(t_1, \ldots, t_n) \approx f(u_1, \ldots, u_n)} \\
& \quad \quad \quad \overline{a \approx b} & \quad \quad \quad \overline{t \approx u} \\
& \quad \quad \quad \quad \overline{\langle a \rangle t \approx \langle b \rangle u} & \quad \quad \quad \overline{a \neq (b, u)} & \quad \quad \quad \overline{t \approx (a \, b) \cdot u} \\
& \quad \quad \quad \quad \quad \overline{\langle a \rangle t \approx \langle b \rangle u}
\end{align*}
\]

- **Note:** abstraction “just another function symbol”; no binding at NL level
Pseudo-unary lexical scoping

- Examples:

  \[
  \text{let } x = e \text{ in } e' \quad \triangleq \quad \text{let}(e, \langle x \rangle e')
  \]

  \[
  p \xrightarrow{x(y)} q \quad \triangleq \quad \text{in}_\text{trans}(p, x, \langle y \rangle q)
  \]

- These can be shoehorned into ULS, by rearranging the abstract syntax trees

  \[
  \begin{align*}
  \text{let}\_\text{exp} & : \quad (\text{exp}, \langle \text{id} \rangle \text{exp}) \rightarrow \text{exp}. \\
  \text{in}\_\text{trans} & : \quad (\text{proc}, \text{id}, \langle \text{id} \rangle \text{proc}) \rightarrow \text{trans}.
  \end{align*}
  \]
Pseudo-unary lexical scoping

- Alternative: Use “natural” syntax

  \[\text{let\_exp} : (\text{id}, \text{exp}, \text{exp}) \rightarrow \text{exp}.\]
  \[\text{trans} : (\text{proc}, \text{act}, \text{proc}) \rightarrow \text{trans}.\]
  \[\text{in} : (\text{id}, \text{id}) \rightarrow \text{act}\]

- Axiomatize equality as follows:

  \[
  \frac{\times \# e_1}{\times \# \text{let\_exp}(x, e_1, e_2)} \quad \frac{y \# q}{\text{let\_exp}(x, e_1, e_2) \approx \text{let\_exp}(y, f_1, f_2)}
  \]
  \[
  \frac{\times \# f_2 \quad e_1 \approx f_1 \quad e_2 \approx (x \cdot y) \cdot f_2}{\times \# q'} \quad p \approx q \quad x \approx x' \quad q \approx (x \cdot y) \cdot q'}
  \]
  \[
  \frac{\text{trans}(p, \text{in}(x, y), q) \approx \text{trans}(p', \text{in}(x', y'), q')}{\text{trans}(p, \text{in}(x, y), q) \approx \text{trans}(p', \text{in}(x', y'), q')}
  \]
Global scoping

- Many languages have “global” scoping:
  - *an identifier may be defined at most once*
  - *identifiers may be defined in one module and referenced anywhere*
- Examples: C program scope, XML IDs, module systems
- Also, in a namespace system, defined identifiers must be unique within namespace.
Global scoping

- Our solution: add type and term constructor for “unique definitions”

\[ t ::= \cdots | a!! \quad \tau ::= \cdots | \nu!! \]

- Refine well-formedness so that at most one name can be uniquely defined in a term.

- Judgment \( S \vdash t : \tau \) means that \( t : \tau \) and uniquely defines the names \( S \subseteq \Lambda \).

\[
\begin{align*}
  &a : \nu \in \Sigma & c : \tau \in \Sigma & S \uplus \{a\} \vdash t : \tau & a : \nu \in \Sigma & a \in S \\
  &S \vdash a : \nu & S \vdash c : \tau & S \vdash \langle a \rangle t : \langle \nu \rangle \tau & S \vdash a!! : \nu \\
  &S = \biguplus_{i=1}^{n} S_i & \bigwedge_{i=1}^{n} S_i \vdash t_i : \tau_i & f : (\tau_1, \ldots, \tau_n) \to \tau \in \Sigma & S \vdash f(t_1, \ldots, t_n) : \tau
\end{align*}
\]
Anonymous identifiers

- Names are often used as “dummies” to describe a data structure.
- e.g., graph vertices, automaton state names, universal variables in ML type schemes or Horn clauses.
- The choice of names is arbitrary; that is, such data structures are invariant up to name permutations.
- e.g., the following are equivalent:

\[ \alpha \rightarrow \beta \rightarrow \beta \equiv_{MLTypeScheme} \beta \rightarrow \gamma \rightarrow \gamma \]

\[ \{1, 2, 3\}, \{(1, 2), (1, 3)\} \equiv_{Graph} \{x, y, z\}, \{(x, y), (x, z)\} \]
Anonymous identifiers

- To handle anonymity within NL, add a type \( \tau \) of “anonymous values of type \( \tau \)”

- Equivalently, \( \tau \) is the type of equivalence classes of \( \tau \) up to renaming.

- Axiomatized as follows:

\[
\begin{align*}
& (a \, b) \cdot t \approx u \\
\text{a} \not\equiv t \quad \text{(a \# t)} \quad \text{t \cong u} \quad \text{t \equiv u}
\end{align*}
\]

- Then type schemes, Horn clauses, graphs, automata etc. can be encoded by using \( \tau \) at the appropriate place.

- Observe that \( t \) always has an equivalent form such that all names are completely fresh (for any finite name context).
Aside

- As a aside, note that the obvious syntactic encoding of sets/transition relations as lists used in graphs and automata is inadequate.
- To recover adequacy, need to equate lists up to commutativity and idempotence.
- But this is no problem in NL: just add axioms.
- More generally, structural congruences (including laws involving binding) translate directly to axioms in NL.
- E.G. $\pi$-calculus

\[
\begin{align*}
\frac{x \not\in P}{\nu x. P \approx P} & \quad \frac{x \not\in Q}{(\nu x. P) \mid Q \approx \nu x.(P\mid Q)} & \quad \nu x.\nu y. P \approx \nu y.\nu x. P
\end{align*}
\]
Simultaneous binding (pattern matching)

- ML-style pattern matching binds “all names in a pattern” simultaneously
- Example:

  \[
  \text{case } e \text{ of } f(x, g(y, z)) \Rightarrow e'[x, y, z] | \cdots
  \]
Simultaneous binding (pattern matching)

- Our solution: define auxiliary predicate(s) \( bnd(x, p) \), meaning “pattern \( p \) binds \( x \)"

\[
\begin{align*}
\text{bnd}(x, x) & \quad \text{bnd}(x, e_i) \\
\text{bnd}(x, f(e_1, \ldots, e_n))
\end{align*}
\]

- Axiomatize pattern equivalence-up-to-renaming in terms of \( bnd \)

\[
\frac{\text{bnd}(x, p)}{x \# (p \Rightarrow e)} \quad \ldots
\]

- Could also axiomatize pattern variable linearity
Putting it all together: letrec

- Let’s show how to handle a realistic “letrec” construct.

\[
\begin{align*}
\text{letrec } f_1 \ p_1^1 & = e_1^1 \\
\vdots & \\
\text{and } f_m \ p_m^1 & = e_m^1 \\
\vdots & \\
\text{and } f_m \ p_m^{n_m} & = e_m^{n_m}
\end{align*}
\]
Basic problem

- Syntax encoding:

  \[ \text{letrec} : \text{list (fname!!; list (list pattern, exp)) \rightarrow decl} \]

- Handle uniqueness of function names using `!!`.
- Handle binding of \( \text{list (list pattern, exp)} \) using `bnd` predicate
- Can’t just treat like iterated “let”, since later names have scope in earlier function bodies.
**Approach #1**

- Specify binding behavior of only the first function

\[
\begin{align*}
\text{letrec}(\langle \text{f; body} \rangle :: \text{l}) & \approx \text{letrec}(\langle \text{f} \rangle :: \text{l}) \\
\text{letrec}(\langle \text{f} \rangle :: \text{l}) & \approx (f \ g) \cdot (b, l')
\end{align*}
\]

- Observation: Does work for “the first” \( f \)

- Treat all function bodies as “the first” in parallel

\[
\begin{align*}
\text{perm}(l, l') & \Rightarrow \text{letrec}(l) \approx \text{letrec}(l')
\end{align*}
\]

where \( \text{perm} \) says that \( l \) is a permutation of \( l' \).
Approach #2

- Approach #1 presumes that order of bodies is immaterial.
- This might be OK for pure formalization purposes.
- But not realistic for e.g. source to source translation
- since programmers *don’t like* unnecessary syntactic changes.
- If we really do care about the order of letrec bodies, can axiomatize using *bnd* instead.
Summary

- Advantages of this approach
  - Seems very flexible
  - Nice equational characterizations
- Disadvantages
  - Ad hoc axiomatic extensions to equational/freshness theory
  - Not clear how portable to other approaches
Related work

- FreshOCaml [Shinwell]: allows arbitrary data structures in abstractions, can specify that only some name type becomes bound, fairly mature

- Cαml [Pottier]: also allows general data structures in abstractions, has keywords “binds”, “inner” and “outer” for describing how names are scoped.

- Sewell, Zdancewic, others (conversations this week): ideas for generalized BNF+binding syntax

- All notations are more compact (and likely more convenient in common cases) but can be translated to NL axioms.

- Exploration of the design space is good!
Big picture

- Lots of examples of axiomatizations of interesting binding behavior
- Observation: $\alpha$ is just one of several structural congruence principles that can be freely combined in NL
- Need more unifying principles for how to handle, e.g. patterns, letrec, general structural congruences
- Conjecture: All “reasonable” structural congruences can be expressed in NL, are decidable in PTIME and unifiable in NPTIME.
- How to get induction/recursion principles for arbitrary (nominal) structural congruences?
- Future work: Nominal equational unification (and NPTIME subclasses), integration into $\alpha$Prolog?
- Future work: Investigate higher-level binding specifications/types