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* $\mathsf{a}, \mathsf{b} \in \mathbb{A}$,
- *swapping* (i.e. invertible renaming) $(\mathsf{a}\ \mathsf{b}) \cdot x$,
- *freshness* (the "not free in" relation") $\mathsf{a} \# x$,
- a *name-abstraction* operation $\langle \mathsf{a} \rangle x$ providing unary lexical scoping.
- Terms

$$t ::= \mathsf{a} \mid f(\bar{t}) \mid c \mid \langle \mathsf{a} \rangle t$$

- Types

$$\tau ::= \nu \mid \delta \mid \langle \nu \rangle \tau$$

ν: name types, δ: data types

Nominal equational logic

- Well-formedness

$$\frac{\mathsf{a} : \nu \in \Sigma}{\mathsf{a} : \nu} \quad \frac{c : \tau \in \Sigma}{c : \tau} \quad \frac{\mathsf{a} : \nu \quad t : \tau}{\langle \mathsf{a} \rangle t : \langle \nu \rangle \tau}$$

$$\frac{t_i : \tau_i \quad f : (\tau_1, \ldots, \tau_n) \longrightarrow \delta \in \Sigma}{f(\bar{t}) : \delta}$$

- Swapping ($\pi : \mathbb{A} \to \mathbb{A}$ a permutation)

$$\begin{aligned} \pi \cdot \mathsf{a} &= \pi(\mathsf{a}) \\ \pi \cdot c &= c \\ \pi \cdot f(\bar{t}) &= f(\pi \cdot \bar{t}) \\ \pi \cdot \langle \mathsf{a} \rangle t &= \langle \pi \cdot \mathsf{a} \rangle \pi \cdot t \end{aligned}$$

Nominal equational logic

- Freshness

$$\frac{(\mathsf{a} \neq \mathsf{b})}{\mathsf{a} \# \mathsf{b}} \quad \frac{}{\mathsf{a} \# c} \quad \frac{\mathsf{a} \# t_i \quad (i = 1, \dots, n)}{\mathsf{a} \# f(t_1, \dots, t_n)}$$

$$\frac{\mathsf{a} \# \mathsf{b} \quad \mathsf{a} \# t}{\mathsf{a} \# \langle \mathsf{b} \rangle t} \quad \frac{}{\mathsf{a} \# \langle \mathsf{a} \rangle t}$$

- Equality

$$\frac{}{\mathsf{a} \approx \mathsf{a}} \quad \frac{}{c \approx c} \quad \frac{t_i \approx u_i \quad (i = 1, \dots, n)}{f(t_1, \dots, t_n) \approx f(u_1, \dots, u_n)}$$

$$\frac{\mathsf{a} \approx \mathsf{b} \quad t \approx u}{\langle \mathsf{a} \rangle t \approx \langle \mathsf{b} \rangle u} \quad \frac{\mathsf{a} \# (\mathsf{b}, u) \quad t \approx (\mathsf{a}\ \mathsf{b}) \cdot u}{\langle \mathsf{a} \rangle t \approx \langle \mathsf{b} \rangle u}$$

- Note: abstraction “just another function symbol”; no binding at NL level

Pseudo-unary lexical scoping

- Examples:

$$\begin{aligned} \mathsf{let}\ x = e\ \mathsf{in}\ e' &\stackrel{\triangle}{=} let(e, \langle x \rangle e') \\ p \xrightarrow{x(y)} q &\stackrel{\triangle}{=} in\_trans(p, x, \langle y \rangle q) \end{aligned}$$

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

$$\begin{aligned} let\_exp &: (exp, \langle id \rangle exp) \to exp. \\ in\_trans &: (proc, id, \langle id \rangle proc) \to trans. \end{aligned}$$

Pseudo-unary lexical scoping

- Alternative: Use "natural" syntax

$$
\begin{aligned}
let\_exp &: (id, exp, exp) \to exp. \\
trans &: (proc, act, proc) \to trans. \\
in &: (id, id) \to act
\end{aligned}
$$

- Axiomatize equality as follows:

$$
\frac{\mathsf{x} \# e_1}{\mathsf{x} \# let\_exp(\mathsf{x}, e_1, e_2)} \qquad \frac{\mathsf{y} \# q}{\mathsf{y} \# trans(p, in(\mathsf{x}, \mathsf{y}), q)}
$$

$$
\frac{\mathsf{x} \# f_2 \quad e_1 \approx f_1 \quad e_2 \approx (\mathsf{x}\ \mathsf{y}) \cdot f_2}{let\_exp(\mathsf{x}, e_1, e_2) \approx let\_exp(\mathsf{y}, f_1, f_2)}
$$

$$
\frac{\mathsf{y} \# q' \quad p \approx q \quad \mathsf{x} \approx \mathsf{x}' \quad q \approx (\mathsf{x}\ \mathsf{y}) \cdot q'}{trans(p, in(\mathsf{x}, \mathsf{y}), q) \approx trans(p', in(\mathsf{x}', \mathsf{y}'), q')}
$$

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"

$$t ::= \cdots \mid \mathsf{a}!! \qquad \tau ::= \cdots \mid \nu!!$$

- 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 $S \subseteq \mathbb{A}$.

$$\frac{\mathsf{a} : \nu \in \Sigma}{S \vdash \mathsf{a} : \nu} \quad \frac{c : \tau \in \Sigma}{S \vdash c : \tau} \quad \frac{S \uplus \{\mathsf{a}\} \vdash t : \tau}{S \vdash \langle \mathsf{a} \rangle t : \langle \nu \rangle \tau} \quad \frac{\mathsf{a} : \nu \in \Sigma \quad \mathsf{a} \in S}{S \vdash \mathsf{a}!! : \nu}$$

$$\frac{S = \biguplus_1^n S_i \quad \bigwedge_{i=1}^n S_i \vdash t_i : \tau_i \quad f : (\tau_1, \ldots, \tau_n) \to \tau \in \Sigma}{S \vdash f(t_1, \ldots, t_n) : \tau}$$

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:

$$\alpha \to \beta \to \beta \equiv_{MLTypeScheme} \beta \to \gamma \to \gamma$$

$$(\{1,2,3\},\{(1,2),(1,3)\}) \equiv_{Graph} (\{x,y,z\},\{(x,y),(x,z)\})$$

Anonymous identifiers

- To handle anonymity within NL, add a type $\tau ⁇$ of "anonymous values of type τ"
- Equivalently, $\tau ⁇$ is the type of equivalence classes of τ up to renaming.
- axiomatized as follows:

$$\frac{}{\mathsf{a} \# t⁇} \qquad \frac{((a\ b) \cdot t)⁇ \approx u⁇}{t⁇ \approx u⁇}$$

- Then type schemes, Horn clauses, graphs, automata etc. can be encoded by using ⁇ 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. π-calculus

$$\frac{x \# P}{\nu x.P \approx P} \qquad \frac{x \# Q}{(\nu x.P) \mid Q \approx \nu x.(P|Q)} \qquad \frac{}{\nu x.\nu y.P \approx \nu y.\nu x.P}$$

Simultaneous binding (pattern matching)

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

$$\mathsf{case}\ e\ \mathsf{of}\ f(x, g(y, z)) \Rightarrow e'[x, y, z] \mid \cdots$$

Simultaneous binding (pattern matching)

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

$$\frac{}{bnd(x, x)} \quad \frac{bnd(x, e_i)}{bnd(x, f(e_1, \ldots, e_n))}$$

- Axiomatize pattern equivalence-up-to-renaming in terms of bnd

$$\frac{bnd(x, p)}{x \# (p \Rightarrow e)} \quad \ldots$$

- Could also axiomatize pattern variable linearity

Putting it all together: letrec

- Let's show how to handle a realistic "letrec" construct.

$$
\begin{aligned}
letrec\ f_1\ \overline{p_1^1} &= e_1^1 \\
&\vdots \\
f_1\ \overline{p_1^{n_1}} &= e_1^{n_1} \\
&\vdots \\
and\ f_m\ \overline{p_m^1} &= e_m^1 \\
&\vdots \\
f_m\ \overline{p_m^{n_m}} &= e_m^{n_m}
\end{aligned}
$$

Basic problem

- Syntax encoding:

$$letrec \quad : \quad list\,(fname!\!!, list\,(list\; pattern, exp)) \to decl$$

- Handle uniqueness of function names using $!\!!$.
- Handle binding of $list\,(list\; pattern, exp)$ 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

$$\frac{}{f \,\#\, letrec((f, body) :: l)} \quad \frac{f \,\#\, b', l' \quad (b, l) \approx (f\ g) \cdot (b, l')}{letrec((f, b) :: l) \approx letrec((g, b') :: l')}$$

- Observation: Does work for “the first” f
- Treat all function bodies as “the first” in parallel

$$\frac{perm(l, l')}{letrec(l) \approx letrec(l')}$$

where $perm$ 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: α 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 αProlog?
- Future work: Investigate higher-level binding specifications/types