An Introduction to Nominal Sets

Andrew Pitts

UNIVERSITY OF CAMBRIDGE
Computer Science & Technology

EWSCS 2020
An introduction to
nominal techniques
motivated by

Programming language semantics/verification

Constructive type theory for theorem-provers

(leaving out motivations from Automata theory, alas)
‘A pure [atomic] name is nothing but a bit-pattern that is an identifier, and is only useful for comparing for identity with other such bit-patterns — which includes looking up in tables to find other information. The intended contrast is with names which yield information by examination of the names themselves, whether by reading the text of the name or otherwise. …like most good things in computer science, pure names help by putting in an extra stage of indirection; but they are not much good for anything else.’

Local names

- **Local variables** in Algol-like languages:
  
  ```
  new X in ⟨command⟩
  ```

- **Generativity** + local declarations in ML-like languages:
  
  ```
  let x = ref⟨val⟩ in ⟨exp⟩
  ```

- **Channel-name restriction** in π-like process calculi:
  
  ```
  (va)⟨process⟩
  ```

- **Use of fresh names** in meta-programming/reasoning, e.g.
  
  ```
  A-nf(e₁ e₂) ≜ let v₁ = e₁, v₂ = e₂ in v₁ v₂
  where v₁ v₂ are fresh variables
  ```
Local names

- **Local variables** in Algol-like languages:
  \[ \text{new } X \text{ in } \langle \text{command} \rangle \]

- **Generativity** + local declarations in ML-like languages:
  \[ \text{let } x = \text{ref}\langle \text{val} \rangle \text{ in } \langle \text{exp} \rangle \]

- **Channel-name restriction** in \( \pi \)-like process calculi:
  \( (v\alpha)\langle \text{process} \rangle \)

- **Use of fresh names** in meta-programming/reasoning, e.g.

\[
A\text{-nf}(e_1 e_2) \triangleq \text{let } v_1 = e_1, v_2 = e_2 \text{ in } v_1 v_2 \\
\text{where } v_1 v_2 \text{ are fresh variables}
\]

What is the mathematical foundation for these locality constructs? Is it the same in each case?
Nominal sets provide a mathematical theory of atomic names based on some simple math to do with properties invariant under permuting names.
Impact can take a long time

The mathematics behind nominal sets goes back a long way…


Outline

**L1** Structural recursion and induction in the presence of name-binding operations.

**L2** Introducing the category of nominal sets.

**L3** Nominal algebraic data types and $\alpha$-structural recursion.

**L4** Dependently typed $\lambda$-calculus with locally fresh names and name-abstraction.

References:

AMP, *Nominal Sets: Names and Symmetry in Computer Science*, CUP 2013


Lecture 1
Nominal sets provide a mathematical theory of atomic names based on some simple math to do with properties invariant under permuting names.

Application area:
computing with / proving properties of data involving name-binding & scoped local names in functional programming languages and theorem-proving systems.

Theory of nominal sets yields principles of structural recursion and induction for syntax modulo renaming of bound names which is close to informal practice and yet fully formal.
For semantics, concrete syntax

```
letrec \( f \) \( x \) = if \( x \) > 100 \ then \( x \) - 10
          else \( f \) ( \( f \) ( \( x \) + 11 ) ) \ in \( f \) ( \( x \) + 100 )
```

is unimportant compared to abstract syntax (ASTs)

since we should aim for compositional semantics of program constructions, rather than of whole programs.
ASTs enable two fundamental (and inter-linked) tools in programming language semantics:

- Definition of functions on syntax by recursion on its structure.
- Proof of properties of syntax by induction on its structure.
Structural recursion

Recursive definitions of functions whose values at a structure are given functions of their values at immediate substructures.

▶ Gödel System T (1958):

\[
\text{structure} = \text{numbers} \\
\text{structural recursion} = \text{primitive recursion for IN}.
\]

▶ Burstall, Martin-Löf et al (1970s) generalized this to ASTs.
Running example

Set of ASTs for λ-terms

\[ Tr \triangleq \{ t ::= V a \mid A(t, t) \mid L(a, t) \} \]

where \( a \in A \), fixed infinite set of names of variables.

Operations for constructing these ASTs:

\[
\begin{align*}
V : & \ A \rightarrow Tr \\
A : & \ Tr \times Tr \rightarrow Tr \\
L : & \ A \times Tr \rightarrow Tr
\end{align*}
\]
Structural recursion for $Tr$

Theorem.

Given

\[ f_1 \in A \rightarrow X \]
\[ f_2 \in X \times X \rightarrow X \]
\[ f_3 \in A \times X \rightarrow X \]

exists unique $\hat{f} \in Tr \rightarrow X$ satisfying

\[ \hat{f}(V a) = f_1 a \]
\[ \hat{f}(A(t, t')) = f_2(\hat{f}t, \hat{f}t') \]
\[ \hat{f}(L(a, t)) = f_3(a, \hat{f}t) \]
Structural recursion for $Tr$

E.g. the finite set $\text{var} \; t$ of variables occurring in $t \in Tr$:

\[
\text{var}(V \; a) = \{a\} \\
\text{var}(A(t, t')) = (\text{var} \; t) \cup (\text{var} \; t') \\
\text{var}(L(a, t)) = (\text{var} \; t) \cup \{a\}
\]

is defined by structural recursion using

- $X = \mathcal{P}_f(\mathcal{A})$ (finite sets of variables)
- $f_1 \; a = \{a\}$
- $f_2(S, S') = S \cup S'$
- $f_3(a, S) = S \cup \{a\}$. 

Structural recursion for $Tr$

E.g. swapping: $(a \ b) \cdot t = \text{result of transposing all occurrences of } a \text{ and } b \text{ in } t$

For example

$$(a \ b) \cdot L(a, A(V \ b, V \ c)) = L(b, A(V \ a, V \ c))$$
Structural recursion for $Tr$

E.g. swapping: $(a\ b) \cdot t = \text{result of transposing all occurrences of } a \text{ and } b \text{ in } t$

$$(a\ b) \cdot V\ c = \begin{cases} V\ b & \text{if } c = a \\ V\ a & \text{if } c = b \\ V\ c & \text{else} \end{cases}$$

$$(a\ b) \cdot A(t, t') = A((a\ b) \cdot t, (a\ b) \cdot t')$$

$$(a\ b) \cdot L(c, t) = \begin{cases} L(b, (a\ b) \cdot t) & \text{if } c = a \\ L(a, (a\ b) \cdot t) & \text{else if } c = b \\ L(c, (a\ b) \cdot t) & \text{else} \end{cases}$$

is defined by structural recursion using…
Structural recursion for $\text{Tr}$

Theorem.

Given

\[
\begin{align*}
  f_1 & \in \text{A} \rightarrow X \\
  f_2 & \in X \times X \rightarrow X \\
  f_3 & \in \text{A} \times X \rightarrow X
\end{align*}
\]

exists unique $\hat{f} \in Tr \rightarrow X$ satisfying

\[
\begin{align*}
  \hat{f} (V a) &= f_1 a \\
  \hat{f} (A(t, t')) &= f_2 (\hat{f} t, \hat{f} t') \\
  \hat{f} (L(a, t)) &= f_3 (a, \hat{f} t)
\end{align*}
\]
Structural recursion for $Tr$

**Theorem.**

Given

\[
\begin{align*}
&f_1 \in \mathbb{A} \to X \\
&f_2 \in X \times Y \\
&f_3 \in X \times Y
\end{align*}
\]

exists unique $\hat{\mathcal{I}} \in Tr \to \mathbb{B}$ satisfying

\[
\begin{align*}
\hat{\mathcal{I}}(V(a)) &= f_1 a \\
\hat{\mathcal{I}}(A(a, t)) &= f_2(\hat{\mathcal{I}}(a), \hat{\mathcal{I}}(t)) \\
\hat{\mathcal{I}}(L(a, t)) &= f_3(a, \hat{\mathcal{I}}(t))
\end{align*}
\]

Doesn’t take binding into account!
Alpha-equivalence

Smallest binary relation $=_{\alpha}$ on $Tr$ closed under the rules:

- $a \in A \vdash V a =_{\alpha} V a$
- $t_1 =_{\alpha} t_1' \vdash t_2 =_{\alpha} t_2' \vdash A(t_1, t_2) =_{\alpha} A(t_1', t_2')$
- $(a \ b) \cdot t =_{\alpha} (a' \ b) \cdot t'$
- $b \not\in \{a, a'\} \cup \text{var}(t) \cup \text{var}(t')$
- $L(a, t) =_{\alpha} L(a', t')$

E.g. $A(L(a, A(V a, V b)), V c) =_{\alpha} A(L(c, A(V c, V b)), V c)$

Fact: $=_{\alpha}$ is transitive (and reflexive & symmetric). [Ex. 1]
Dealing with issues to do with *binders* and *alpha equivalence* is

- pervasive (very many languages involve binding operations)
- difficult to formalise/mechanise without losing sight of common informal practice:
Dealing with issues to do with binders and alpha equivalence is

- pervasive (very many languages involve binding operations)
- difficult to formalise/mechanise without losing sight of common informal practice:

“We identify expressions up to alpha-equivalence”...
ASTs mod alpha equivalence

Dealing with issues to do with binders and alpha equivalence is

- pervasive (very many languages involve binding operations)
- difficult to formalise/mechanise without losing sight of common informal practice:

“We identify expressions up to alpha-equivalence”...

…and then forget about it, referring to alpha-equivalence classes \([t]_\alpha\) only via representatives \(t\).
Dealing with issues to do with *binders* and *alpha equivalence* is

- **pervasive** (very many languages involve binding operations)
- **difficult** to formalise/mechanise without losing sight of common informal practice:

**E.g. notation for \( \lambda \)-terms:**

\[
\Lambda \triangleq \{ [t]_\alpha \mid t \in Tr \}
\]

- \( a \) means \([V a]_\alpha (=\{V a\})\)
- \( e e' \) means \([A(t, t')]_\alpha\), where \( e = [t]_\alpha \) and \( e' = [t']_\alpha\)
- \( \lambda a. e \) means \([L(a, t)]_\alpha\) where \( e = [t]_\alpha\)
Informal structural recursion

E.g. capture-avoiding substitution:
\[ f = (\neg)[e_1/a_1] : \Lambda \to \Lambda \]

\[
\begin{align*}
  f \ a &= \text{if } a = a_1 \text{ then } e_1 \text{ else } a \\
  f \ (e\ e') &= (f \ e) \ (f \ e') \\
  f \ (\lambda a. \ e) &= \text{if } a \notin \text{var}(a_1, e_1) \text{ then } \lambda a. \ (f \ e) \\
  &\quad \text{else don’t care!}
\end{align*}
\]

Not an instance of structural recursion for \( Tr \).

Why is \( f \) well-defined and total?
Informal structural recursion

E.g. denotation of $\lambda$-term in a suitable domain $D$:

$$[-] : \Lambda \rightarrow ((\forall \rightarrow D) \rightarrow D)$$

$$[a]_\rho = \rho a$$

$$[e e']_\rho = app([e]_\rho, [e']_\rho)$$

$$[\lambda a. e]_\rho = fun(\lambda (d \in D) \rightarrow [e](\rho[a \rightarrow d]))$$

where

$$\begin{cases} app \in D \times D \rightarrow_{cts} D \\ fun \in (D \rightarrow_{cts} D) \rightarrow_{cts} D \end{cases}$$

are continuous functions satisfying...
Informal structural recursion

E.g. denotation of $\lambda$-term in a suitable domain $D$:

$$\llbracket - \rrbracket : \Lambda \to ((\Lambda \to D) \to D)$$

$$\llbracket a \rrbracket \rho = \rho a$$

$$\llbracket e e' \rrbracket \rho = \text{app}(\llbracket e \rrbracket \rho, \llbracket e' \rrbracket \rho)$$

$$\llbracket \lambda a. e \rrbracket \rho = \text{fun}(\lambda (d \in D) \to \llbracket e \rrbracket (\rho[a \to d]))$$

why is this very standard definition independent of the choice of bound variable $a$?

$\rho[a \to d]$ is the element of $\Lambda \to D$ that maps $a$ to $d$ and otherwise acts like $\rho$
Is there a recursion principle for $\Lambda$ that legitimises these ‘definitions’ of $(-)[e_1/a_1]: \Lambda \rightarrow \Lambda$ and $[-]: \Lambda \rightarrow D$ (and many other e.g.s)?
Is there a recursion principle for $\Lambda$ that legitimises these ‘definitions’ of $(-)[e_1/a_1] : \Lambda \to \Lambda$ and $[-] : \Lambda \to D$ (and many other e.g.s)?

Yes! — $\alpha$-structural recursion.
Is there a recursion principle for $\Lambda$ that legitimises these ‘definitions’ of $(\lambda)[e_1/a_1] : \Lambda \rightarrow \Lambda$ and $[\lambda] : \Lambda \rightarrow D$ (and many other e.g.s)?

Yes! — $\alpha$-structural recursion.

What about other languages with binders?
Is there a recursion principle for $\Lambda$ that legitimises these ‘definitions’ of $(-)[e_1/a_1] : \Lambda \rightarrow \Lambda$ and $[-] : \Lambda \rightarrow D$ (and many other e.g.s)?

Yes! — $\alpha$-structural recursion.

What about other languages with binders?

Yes! — available for any nominal signature.
Is there a recursion principle for $\Lambda$ that legitimises these ‘definitions’ of $(-)[e_1/a_1] : \Lambda \to \Lambda$ and $[-] : \Lambda \to D$ (and many other e.g.s)?

Yes! — $\alpha$-structural recursion.

What about other languages with binders?

Yes! — available for any nominal signature.

Great. What’s the catch?
Is there a recursion principle for \( \Lambda \) that legitimises these ‘definitions’ of \((-)[e_1/a_1] : \Lambda \rightarrow \Lambda \) and \([-] : \Lambda \rightarrow D \) (and many other e.g.s)?

Yes! — \( \alpha \)-structural recursion.

What about other languages with binders?

Yes! — available for any nominal signature.

Great. What’s the catch?

Need to learn a bit of possibly unfamiliar math, to do with permutations and support.