An Introduction to Nominal Sets

Andrew Pitts

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)

Names in programming languages

‘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.’

RM Needham, *Names* (ACM, 1989) p 90

Local names

- **Local variables** in Algol-like languages:
  $\texttt{new}\ X\ \texttt{in}\ \langle command \rangle$
- **Generativity** + local declarations in ML-like languages:
  $\texttt{let}\ x = \texttt{ref}\langle val \rangle\ \texttt{in}\ \langle exp \rangle$
- Channel-**name restriction** in π-like process calculi:
  $(va)\langle process \rangle$
- Use of **fresh names** in meta-programming/reasoning, e.g.

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

Local names

- **Local variables** in Algol-like languages:
  $\texttt{new}\ X\ \texttt{in}\ \langle command \rangle$
- **Generativity** + local declarations in ML-like languages:
  $\texttt{let}\ x = \texttt{ref}\langle val \rangle\ \texttt{in}\ \langle exp \rangle$
- Channel-**name restriction** in π-like process calculi:
  $(va)\langle process \rangle$
- Use of **fresh names** in meta-programming/reasoning, e.g.

$$\text{A-nf}(e_1\ e_2) \triangleq \texttt{let}\ v_1 = e_1,\ v_2 = e_2\ \texttt{in}\ v_1\ v_2$$
where $v_1\ v_2$ *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…

Abraham Fraenkel, *Der Begriff "definit" und die Unabhängigkeit des Auswahlsaxioms*, Sitzungsberichte der Preussischen Akademie der Wissenschaften, Physikalisch-mathematische Klasse (1922), 253–257.

Andrzej Mostowski, *Uber die Unabhängigkeit des Wohlordnungssatzes vom Ordnungsprinzip*, Fundamenta Mathematicae 32 (1939), 201–252.

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 α-structural recursion.

L4 Dependently typed λ-calculus with locally fresh names and name-abstraction.

References:

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

AMP, *Alpha-Structural Recursion and Induction*, JACM 53(2006)459-506.

AMP, J. Matthiesen and J. Derikx,
A Dependent Type Theory with Abstractable Names, ENTCS 312(2015)19-50.

Lecture 1

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

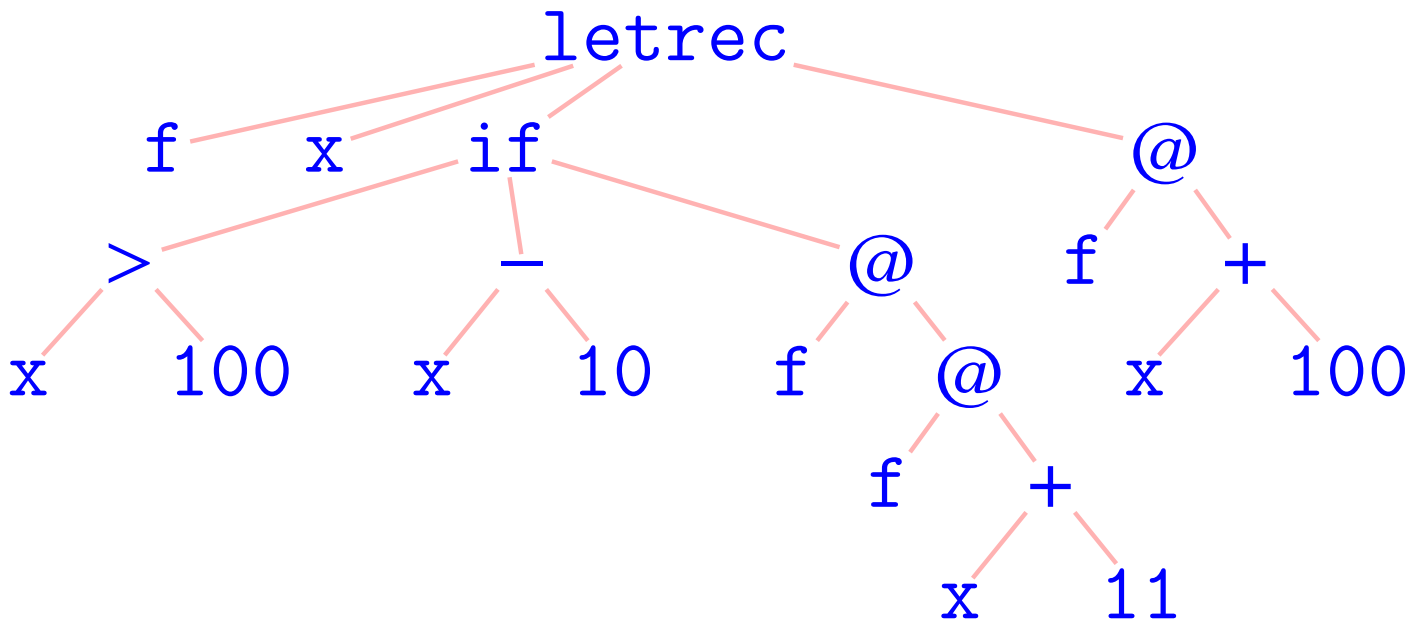
Application area:
computing with / proving properties of
data involving name-binding & scoped local names
in functional programming languages and
theorem-proving systems.

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):

$$\begin{aligned} \text{structure} &= \text{numbers} \\ \text{structural recursion} &= \text{primitive recursion for } \mathbb{N}. \end{aligned}$$

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

Running example

Set of ASTs for λ-terms

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

where $a \in \mathbb{A}$, fixed infinite set of names of variables.

Operations for constructing these ASTs:

$$\begin{aligned} \mathtt{V} &: \mathbb{A} \to Tr \\ \mathtt{A} &: Tr \times Tr \to Tr \\ \mathtt{L} &: \mathbb{A} \times Tr \to Tr \end{aligned}$$

Structural recursion for Tr

Theorem.

Given

$$\begin{array}{lcl} f_1 & \in & \mathbb{A} \to X \\ f_2 & \in & X \times X \to X \\ f_3 & \in & \mathbb{A} \times X \to X \end{array}$$

---

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

$$\begin{array}{rcl} \hat{f}\,(\mathtt{V}\,a) & = & f_1\,a \\ \hat{f}\,(\mathtt{A}(t,t')) & = & f_2(\hat{f}\,t, \hat{f}\,t') \\ \hat{f}(\mathtt{L}(a,t)) & = & f_3(a, \hat{f}\,t) \end{array}$$

Structural recursion for Tr

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

$$\begin{aligned} \mathrm{var}(\mathtt{V}\, a) &= \{a\} \\ \mathrm{var}(\mathtt{A}(t, t')) &= (\mathrm{var}\, t) \cup (\mathrm{var}\, t') \\ \mathrm{var}(\mathtt{L}(a, t)) &= (\mathrm{var}\, t) \cup \{a\} \end{aligned}$$

is defined by structural recursion using

- $X = \mathrm{P_f}(\mathbb{A})$ (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: $(a\ b) \cdot t$ = result of transposing all occurrences of a and b in t

For example

$$(a\ b) \cdot \mathtt{L}(a, \mathtt{A}(\mathtt{V}\, b, \mathtt{V}\, c)) = \mathtt{L}(b, \mathtt{A}(\mathtt{V}\, a, \mathtt{V}\, c))$$

Structural recursion for Tr

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

$$
\begin{array}{rcl}
(a\ b) \cdot \mathtt{V}\, c & = & \text{if } c = a \text{ then } \mathtt{V}\, b \text{ else} \\
 & & \text{if } c = b \text{ then } \mathtt{V}\, a \text{ else } \mathtt{V}\, c \\
(a\ b) \cdot \mathtt{A}(t, t') & = & \mathtt{A}((a\ b) \cdot t, (a\ b) \cdot t') \\
(a\ b) \cdot \mathtt{L}(c, t) & = & \text{if } c = a \text{ then } \mathtt{L}(b, (a\ b) \cdot t) \\
 & & \text{else if } c = b \text{ then } \mathtt{L}(a, (a\ b) \cdot t) \\
 & & \text{else } \mathtt{L}(c, (a\ b) \cdot t)
\end{array}
$$

is defined by structural recursion using…

Structural recursion for Tr

Theorem.

Given $f_1 \in \mathbb{A} \to X$

$f_2 \in X \times X \to X$

$f_3 \in \mathbb{A} \times X \to X$

---

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

$$\begin{aligned} \hat{f}\,(\mathtt{V}\,a) &= f_1\,a \\ \hat{f}\,(\mathtt{A}(t,t')) &= f_2(\hat{f}\,t, \hat{f}\,t') \\ \hat{f}(\mathtt{L}(a,t)) &= f_3(a, \hat{f}\,t) \end{aligned}$$

Structural recursion for Tr

Theorem.

Given $f_1 \in \mathbb{A} \to X$

$f_2 \in X \times$

$f_3 \in$

exists unique X satisfying

$$\begin{aligned} \hat{f}(\quad) &= f_1\, a \\ \hat{f}(\mathrm{B}(t, t')) &= f_2(\hat{f}\, t, \hat{f}\, t') \\ \hat{f}(\mathrm{L}(a, t)) &= f_3(a, \hat{f}\, t) \end{aligned}$$

Doesn't take binding into account!

Alpha-equivalence

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

$$\frac{a \in \mathbb{A}}{\mathtt{V}\, a =_\alpha \mathtt{V}\, a} \qquad \frac{t_1 =_\alpha t_1' \qquad t_2 =_\alpha t_2'}{\mathtt{A}(t_1, t_2) =_\alpha \mathtt{A}(t_1', t_2')}$$

$$\frac{(a\ b) \cdot t =_\alpha (a'\ b) \cdot t' \qquad b \notin \{a, a'\} \cup \mathrm{var}(t) \cup \mathrm{var}(t')}{\mathtt{L}(a, t) =_\alpha \mathtt{L}(a', t')}$$

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

$\neq_\alpha \mathtt{A}(\mathtt{L}(b, \mathtt{A}(\mathtt{V}\, b, \mathtt{V}\, b)), \mathtt{V}\, c)$

Fact: $=_\alpha$ is transitive (and reflexive & symmetric). [Ex. 1]

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:

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"…

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.

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:

E.g. notation for λ-terms:

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

a	means	$[\mathtt{V}\, a]_\alpha$ $(= \{\mathtt{V}\, a\})$
$e\, e'$	means	$[\mathtt{A}(t, t')]_\alpha$, where $e = [t]_\alpha$ and $e' = [t']_\alpha$
$\lambda a.e$	means	$[\mathtt{L}(a, t)]_\alpha$ where $e = [t]_\alpha$

Informal structural recursion

E.g. capture-avoiding substitution:

$$f = (-)[e_1/a_1] : \Lambda \to \Lambda$$

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

Not an instance of structural recursion for Tr.

Why is f well-defined and total?

Informal structural recursion

E.g. denotation of λ-term in a suitable domain D:

$$[\![-]\!] : \Lambda \to ((\mathbb{A} \to D) \to D)$$

$$\begin{aligned} [\![a]\!]\rho &= \rho\, a \\ [\![e\, e']\!]\rho &= app([\![e]\!]\rho\,, [\![e']\!]\rho) \\ [\![\lambda a.\, e]\!]\rho &= fun(\lambda(d \in D) \to [\![e]\!](\rho[a \to d])) \end{aligned}$$

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

are continuous functions satisfying...

Informal structural recursion

E.g. denotation of λ-term in a suitable domain D:

$$[\![-]\!] : \Lambda \to ((\mathbb{A} \to D) \to D)$$

$$\begin{aligned} [\![a]\!]\rho &= \rho\, a \\ [\![e\, e']\!]\rho &= app([\![e]\!]\rho\,, [\![e']\!]\rho) \\ [\![\lambda a.\, e]\!]\rho &= fun(\lambda(d \in D) \to [\![e]\!](\rho[a \to d])) \end{aligned}$$

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

$\rho[a \to d]$ is the element of $\mathbb{A} \to D$ that maps a to d and otherwise acts like ρ

Is there a recursion principle for Λ that legitimises these 'definitions' of $(-)[e_1/a_1] : \Lambda \to \Lambda$ and $[\![-]\!] : \Lambda \to D$ (and many other e.g.s)?

Is there a recursion principle for Λ that legitimises these 'definitions' of $(-)[e_1/a_1] : \Lambda \to \Lambda$ and $[\![-]\!] : \Lambda \to D$ (and many other e.g.s)?

Yes! — α-structural recursion.

Is there a recursion principle for Λ that legitimises these 'definitions' of $(-)[e_1/a_1] : \Lambda \to \Lambda$ and $[\![-]\!] : \Lambda \to D$ (and many other e.g.s)?

Yes! — α-structural recursion.

What about other languages with binders?

Is there a recursion principle for Λ that legitimises these 'definitions' of $(-)[e_1/a_1] : \Lambda \to \Lambda$ and $[\![-]\!] : \Lambda \to D$ (and many other e.g.s)?

Yes! — α-structural recursion.

What about other languages with binders?

Yes! — available for any nominal signature.

Is there a recursion principle for Λ that legitimises these 'definitions' of $(-)[e_1/a_1] : \Lambda \to \Lambda$ and $[\![-]\!] : \Lambda \to D$ (and many other e.g.s)?

Yes! — α-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 Λ that legitimises these 'definitions' of $(-)[e_1/a_1] : \Lambda \to \Lambda$ and $[\![-]\!] : \Lambda \to D$ (and many other e.g.s)?

Yes! — α-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.