An Introduction to Nominal Sets

Andrew Pitts

UNIVERSITY OF CAMBRIDGE
Computer Science & Technology

EWSCS 2020
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


Original motivation for Gabbay & AMP to introduce nominal sets and name abstraction:

$[\mathbb{A}](-)$ can be combined with $\times$ and $+$ to give functors $\text{Nom} \rightarrow \text{Nom}$ that have initial algebras coinciding with sets of abstract syntax trees modulo $\alpha$-equivalence.

E.g. the initial algebra for $\mathbb{A} + (- \times -) + [\mathbb{A}](-)$ is isomorphic to the usual set of untyped $\lambda$-terms.
Recall: \(\alpha\)-Structural recursion

For \(\lambda\)-terms:

\textbf{Theorem.} Given any \(X \in \text{Nom}\) and

\[
\begin{align*}
    f_1 &\in \Lambda \rightarrow_{fs} X \\
    f_2 &\in X \times X \rightarrow_{fs} X \quad \text{s.t.} \\
    f_3 &\in \Lambda \times X \rightarrow_{fs} X
\end{align*}
\]

\(\forall a, \forall x, \; a \neq f_3(a, x) \quad \text{(FCB)}\)

\(\exists! \; \hat{f} \in \Lambda \rightarrow_{fs} X \quad \text{s.t.}
\]

\[
\begin{align*}
    \hat{f} a &= f_1 a \\
    \hat{f} (e_1 e_2) &= f_2(\hat{f} e_1, \hat{f} e_2) \\
    \hat{f}(\lambda a.e) &= f_3(a, \hat{f} e) \quad \text{if} \; a \neq (f_1, f_2, f_3)
\end{align*}
\]

Can we avoid explicit reasoning about finite support, \# and (FCB) when computing ‘mod \(\alpha\)?’

Want definition/computation to be separate from proving.
\[
\hat{f} = f_1 a \\
\hat{f}(e_1 e_2) = f_2(f(e_1), f(e_2)) \\
\hat{f}(\lambda a. e) = f_3(a, \hat{f} e) \quad \text{if} \ a \neq (f_1, f_2, f_2) \\
\]

Q: how to get rid of this inconvenient proof obligation?
\[
\begin{align*}
\hat{f} &= f_1 \ a \\
\hat{f}(e_1 e_2) &= f_2(\hat{f} \ e_1, \hat{f} \ e_2) \\
\hat{f}(\lambda a. \ e) &= \nu a. \ f_3(a, \hat{f} \ e) \quad [\ a \ # \ (f_1, f_2, f_2) \ ]
\end{align*}
\]

Q: how to get rid of this inconvenient proof obligation?

A: use a local scoping construct \( \nu a. (-) \) for names
\[
\begin{align*}
\hat{f} &= f_1 a \\
\hat{f}(e_1 e_2) &= f_2(\hat{f} e_1, \hat{f} e_2) \\
\hat{f}(\lambda a. e) &= \nu a. f_3(a, \hat{f} e) \quad [ a \# (f_1, f_2, f_2) ] \\
\end{align*}
\]

Q: how to get rid of this inconvenient proof obligation?

A: use a local scoping construct \( \nu a. (\_\,) \) for names

which one?!
Dynamic allocation

- Stateful: va. t means “add a fresh name a’ to the current state and return t[a’/a]”.
- Used in Shinwell’s Fresh OCaml = OCaml +
  - name types and name-abstraction type former
  - name-abstraction patterns
    - matching involves dynamic allocation of fresh names

[www.cl.cam.ac.uk/users/amp12/fresh-ocaml]
Sample Fresh OCaml code

(* syntax *)

type t;;
type var = t name;;
type term = Var of var | Lam of <<var>>term | App of term*term;;

(* semantics *)
type sem = L of ((unit -> sem) -> sem) | N of neu
and neu = V of var | A of neu*sem;;

(* reify : sem -> term *)

let rec reify d =
match d with L f -> let x = fresh in Lam(<<x>>(reify(f(function () -> N(V x)))))
| N n -> reifyn n
and reifyn n =
match n with V x -> Var x
| A(n’,d’) -> App(reifyn n’, reify d’);;

(* evals : (var * (unit -> sem))list -> term -> sem *)

let rec evals env t =
match t with Var x -> (match env with [] -> N(V x)
| (x’,v)::env -> if x=x’ then v() else evals env (Var x))
| Lam(<<x>>t) -> L(function v -> evals ((x,v)::env) t)
| App(t1,t2) -> (match evals env t1 with L f -> f(function () -> evals env t2)
| N n -> N(A(n,evals env t2))));;

(* eval : term -> sem *)

let rec eval t = evals [] t;;

(* norm : lam -> lam *)

let norm t = reify(eval t);;
Dynamic allocation

- Stateful: $va.t$ means “add a fresh name $a'$ to the current state and return $t[a'/a]$”.
- Used in Shinwell’s Fresh OCaml = OCaml +
  - name types and name-abstraction type former
  - name-abstraction patterns
    — matching involves dynamic allocation of fresh names


[www.cl.cam.ac.uk/users/amp12/fresh-ocaml]
Dynamic allocation

- **Stateful**: \( \text{va. } t \) means “add a fresh name \( a' \) to the current state and return \( t[a'/a] \).”

Statefulness disrupts familiar mathematical properties of pure datatypes. So let’s try to reject it in favour of...
Aim

A version of Martin-Löf Type Theory enriched with constructs for
locally fresh names and name-abstraction
from the theory of nominal sets.

Motivation:

Machine-assisted construction of
humanly understandable formal proofs
about software (PL semantics).
Aim

More specifically: extend (dependently typed) \(\lambda\)-calculus with

- names \(a\)
- name swapping \(\text{swap } a, b \text{ in } t\)
- name abstraction \(\langle a \rangle t\) and concretion \(t @ a\)
- locally fresh names \(\text{fresh } a \text{ in } t\)
- name equality \(\text{if } t = a \text{ then } t_1 \text{ else } t_2\)
Locally fresh names

For example, here are some isomorphisms, described in an informal pseudocode:

\[ i : \lbrack \text{A}\rbrack (X + Y) \cong \lbrack \text{A}\rbrack X + \lbrack \text{A}\rbrack Y \]
\[ i(z) = \text{fresh } a \text{ in case } z \odot a \text{ of} \]
\[ \text{inl}(x) \rightarrow \langle a \rangle x \]
\[ \mid \text{inr}(y) \rightarrow \langle a \rangle y \]

[Ex. 7]
Locally fresh names

For example, here are some isomorphisms, described in an informal pseudocode:

\[ i : [\text{\_}](X + Y) \cong [\text{\_}]X + [\text{\_}]Y \]
\[ i(z) = \text{fresh } a \text{ in case } z @ a \text{ of} \]
\[ \text{inl}(x) \rightarrow \langle a \rangle x \]
\[ | \text{inr}(y) \rightarrow \langle a \rangle y \]

given \( f \in \text{Nom}(X \ast \underline{\text{\_}}, Y) \)
satisfying \( a \neq x \Rightarrow a \neq f(x, a) \),
we get \( \hat{f} \in \text{Nom}(X, Y) \) well-defined by:
\( f(x) = f(x, a) \) for some/any \( a \neq x \).

Notation: fresh \( a \) in \( f(x, a) \) \( \triangleq \hat{f}(x) \)
Locally fresh names

For example, here are some isomorphisms, described in an informal pseudocode:

\[ i : [\forall](X + Y) \cong [\forall]X + [\forall]Y \]
\[ i(z) = \text{fresh } a \text{ in case } z \oplus a \text{ of} \]
\[ \text{inl}(x) \rightarrow \langle a \rangle x \]
\[ \text{| inr}(y) \rightarrow \langle a \rangle y \]

\[ j : ([\forall]X \to_{fs} [\forall]Y) \cong [\forall](X \to_{fs} Y) \]
\[ j(f) = \text{fresh } a \text{ in} \]
\[ \langle a \rangle (\lambda x. \ f(\langle a \rangle x) \oplus a) \]

Can one turn the pseudocode into terms in a formal ‘nominal’ \( \lambda \)-calculus?
Prior art

▶ Stark-Schöpp [CSL 2004]
bunched contexts (+), extensional & undecidable (–)

▶ Westbrook-Stump-Austin [LFMTP 2009] CNIC
semantics/expressivity?

▶ Cheney [LMCS 2012] DNTT
bunched contexts (+), no local fresh names (–)

▶ Fairweather-Fernández-Szasz-Tasistro [2012]
based on nominal terms (+), explicit substitutions (–), first-order (±)

▶ Crole-Nebel [MFPS 2013]
simple types (–), definitional freshness (+)
Our art

- **Stark-Schöpp [CSL 2004]**
  bunched contexts (+), extensional & undecidable (−)

- **Westbrook-Stump-Austin [LFMTP 2009] CNIC**
  semantics/expressivity?

- **Cheney [LMCS 2012] DNTT**
  bunched contexts (+), no local fresh names (−)

- **Fairweather-Fernández-Szasz-Tasistro [2012]**
  based on nominal terms (+), explicit substitutions (−), first-order (±)

- **Crole-Nebel [MFPS 2013]**
  simple types (−), definitional freshness (+)

More specifically: extend (dependently typed) $\lambda$-calculus with

- names $a$
- name swapping `swap a, b in t`
- name abstraction $\langle a \rangle t$ and concretion $t @ a$
- locally fresh names `fresh a in t`
- name equality `if t = a then t₁ else t₂`

Difficulty: concretion and locally fresh names are partially defined – have to check freshness conditions.

e.g. for `fresh a in f(x, a)` to be well-defined, we need
\[ a \not\approx x \Rightarrow a \not\approx f(x, a) \]
Definitional freshness

In a nominal set of (higher-order) functions, proving \( a \neq f \) can be tricky (undecidable). Common proof pattern:

Given \( a, f, \ldots \), pick a fresh name \( b \) and prove \((a \ b) \cdot f = f\). (For functions, equivalent to proving \( \forall x, \ (a \ b) \cdot f(x) = f((a \ b) \cdot x) \).)
Definitional freshness

In a nominal set of (higher-order) functions, proving $a \neq f$ can be tricky (undecidable). Common proof pattern:

Given $a, f, \ldots$, pick a fresh name $b$ and prove $(a \ b) \cdot f = f$.

Since by choice of $b$ we have $b \neq f$, we also get $a = (a \ b) \cdot b \neq (a \ b) \cdot f = f$, QED.
Definitional freshness

\[
\begin{align*}
\Gamma \vdash a \# T & \quad \Gamma \vdash t : T \\
\Gamma#(b : A) \vdash (\text{swap } a, b \text{ in } t) = t : T \\
\hline
\Gamma \vdash a \# t : T
\end{align*}
\]

bunched contexts, generated by
\[
\begin{align*}
\Gamma & \mapsto \Gamma(x : T) \\
\Gamma & \mapsto \Gamma#(a : A)
\end{align*}
\]
definitional freshness
definitional equality
Definitional freshness

\[ \Gamma \vdash a \# T \quad \Gamma \vdash t : T \]
\[ \Gamma\#(b : A) \vdash (\text{swap } a, b \text{ in } t) = t : T \]
\[ \Gamma \vdash a\# t : T \]

definitional freshness for types:
\[ \Gamma \vdash T \quad a \in \Gamma \]
\[ \Gamma\#(b : A) \vdash (\text{swap } a, b \text{ in } T) = T \]
\[ \Gamma \vdash a \# T \]
Definitional freshness

\[
\begin{align*}
\Gamma \vdash a \# T & \quad \Gamma \vdash t : T \\
\Gamma\#(b : A) \vdash (\text{swap } a, b \text{ in } t) &= t : T \\
\hline
\Gamma \vdash a\# t : T
\end{align*}
\]

Freshness info in bunched contexts gets used via:

\[
\begin{align*}
\Gamma(x : T)\Gamma' \text{ ok} & \quad a, b \in \Gamma' \\
\hline
\Gamma(x : T)\Gamma' \vdash (\text{swap } a, b \text{ in } x) &= x : T
\end{align*}
\]
A type theory
A type theory
Nominal set semantics of dependent type theory

A family over $X \in \text{Nom}$ is specified by:

- $X$-indexed family of sets $(Y_x \mid x \in X)$
- dependently type permutation action

\[ \prod_{\pi \in \text{Perm}_A} \prod_{x \in X} (Y_x \to Y_{\pi \cdot x}) \]

with dependent version of finite support property:

for all $x \in X, e \in Y_x$ there is a finite set $A$ of names supporting $x$ in $X$ and such that any $\pi$ fixing each $a \in A$ satisfies

\[ \pi \cdot e = e \]
\[ \text{\_\_} = \text{\_\_} \]
\[ Y_{\pi \cdot x} = Y_x \]
Nominal set semantics of dependent type theory

A family over $X \in \text{Nom}$ is specified by...

Get a **category with families** (CwF) [Dybjer] modelling extensional MLTT, plus

nominal logic’s freshness quantifier

[[a \in A] Y_a] ←→ [a \in A \setminus] Y_a

Curry-Howard name-abstraction
For more details, see

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

But much remains to do, e.g.

- Explore inductively defined types involving $[a : \mathcal{A}](\_)$ (e.g. propositional freshness).
- Dependentely typed pattern-matching with name-abstraction patterns.

Difficulties:

- Is definitional freshness too weak? (cf. experience with FreshML2000)
- Name-swapping with variables of type $\mathcal{A}$
Other applications of nominal sets

- **Computational logic**
  - Higher-order logic: Urban & Berghofer’s Nominal package for the interactive theorem-prover Isabelle/HOL.
  - Equational logic: unification & rewriting for nominal terms [Fernandez+Gabbay+Levy+Villaret+⋯] Logic programming mod $\alpha$ (e.g. Cheney’s $\alpha$Prolog)
Other applications of nominal sets

- Computational logic
  - Higher-order logic: Urban & Berghofer’s Nominal package for the interactive theorem-prover Isabelle/HOL.
  - Equational logic: unification & rewriting for nominal terms [Fernandez+Gabbay+Levy+Villaret+···]
    Logic programming mod $\alpha$ (e.g. Cheney’s $\alpha$Prolog)

- Automata theory & verification
  - HD-automata [Montanari el al]
  - fresh-register automata [Tzevelekos]
  - orbit-finite computation theory [Bojańczyk et al]
Other applications of nominal sets

▶ **Homotopy Type Theory (HoTT)**

Cubical sets [Bezem-Coquand-Huber] model of Voevodsky’s axiom of univalence makes use of nominal sets equipped with an operation of substitution $x \mapsto x(i/a)$ where $i \in \{0, 1\}$.

- names are *names of directions* (cartesian axes)
  (so e.g., if an object has support $\{a, b, c\}$ it is 3-dimensional)
- freshness ($a \# x$) = degeneracy ($x(i/a) = x$)
- identity types are modelled by name-abstraction: $\langle a \rangle x$ is a proof that $x(0/a)$ is equal to $x(1/a)$.

HoTT and univalence is about (computable) *mathematical foundations* (a topic no longer very popular with mathematicians). That’s where the mathematics of nominal sets came from...
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 Auswahlaxioms, 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.
Impact can take a long time

The mathematics behind nominal sets goes back a long way… …and it’s still too early to tell what will be the impact of the applications of it to CS developed over the last 20 years.

Two take-home messages from these lectures:

▶ Specific: in meta-programming/proving, permutation comes before substitution and (hence) name-abstraction before lambda-abstraction

▶ General: computation modulo symmetry deserves further exploration.