An Introduction to Nominal Sets

Andrew Pitts

UNIVERSITY OF CAMBRIDGE
Computer Science & Technology

EWSCS 2020
Lecture 4
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:

\([A\backslash](\_\_\_\) 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 \(A + (\_ \times \_\_ + [A\backslash](\_\)\) is isomorphic to the usual set of untyped \(\lambda\)-terms.
Recall: $\alpha$-Structural recursion

For $\lambda$-terms:

**Theorem.**

Given any $X \in \text{Nom}$ and

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

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

\[
\exists! \hat{f} \in \Lambda \rightarrow_{fs} X \\
\text{s.t.} \begin{cases} 
  \hat{f} a = f_1 a \\
  \hat{f}(e_1 e_2) = f_2(\hat{f} e_1, \hat{f} e_2) \\
  \hat{f}(\lambda a.e) = f_3(a, \hat{f} e) \quad \text{if} \ a \neq (f_1, f_2, f_3)
\end{cases}
\]

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

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

\[ = \lambda a'. e' \]
\[ = f_3(a', \hat{f} e') \]

Q: how to get rid of this inconvenient proof obligation?
\[
\hat{f} = f_1 a \\
\hat{f}(e_1 e_2) = f_2(\hat{f} e_1, \hat{f} e_2) \\
\hat{f}(\lambda a. e) = va. f_3(a, \hat{f} e) \quad [a \neq (f_1, f_2, f_2)]
\]

\[
= \lambda a'. e' \\
= va'. f_3(a', \hat{f} e') \quad \text{OK!}
\]

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

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

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

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

which one?!
Dynamic allocation

- **Stateful:** $\textit{va. t}$ 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]
(* syntax *)

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

(* semantics *)

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

(* reify : sem -> term *)

```ocaml
let rec reify d =
    match d with L f -> let x = fresh in Lam(<<x>>(reify(f(function () -> N(V x)))))
    | N n -> reify n

and reify n =
    match n with V x -> Var x
    | A(n',d') -> App(reify n', reify d');;
```

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

```ocaml
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 *)

```ocaml
let rec eval t = evals [] t;;
```

(* norm : lam -> lam *)

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

- Stateful: \textit{va.} \textit{t} means “add a fresh name \textit{a’} to the current state and return \textit{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**: \( va.t \) 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:

\[ i : [\forall] (X + Y) \cong [\forall]X + [\forall]Y \]

\[ i(z) = \text{fresh } a \text{ in case } z @ a \text{ of} \]

\[ \text{inl}(x) \rightarrow \langle a \rangle x \]

\[ | \text{inr}(y) \rightarrow \langle a \rangle y \]

[Ex. 7]
Locally fresh names

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

\[ i : ([\forall] (X + Y)) \cong ([\forall]X + [\forall]Y) \]

\[ i(z) = \text{fresh } a \text{ in case } z @ a \text{ of } \]

\[ \text{inl}(x) \rightarrow \langle a \rangle x \]
\[ | \text{inr}(y) \rightarrow \langle a \rangle y \]

Given \( f \in \text{Nom}(X \ast \forall, Y) \) satisfying \( a \# x \Rightarrow a \# f(x, a) \), we get \( \hat{f} \in \text{Nom}(X, Y) \) well-defined by:

\( \hat{f}(x) = f(x, a) \) for some/any \( a \# x \).

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:

\[
i : \forall X (X + Y) \cong \forall X + \forall Y
\]
\[
i(z) = \text{fresh } a \text{ in case } z @ a \text{ of }
\]
\[
\quad \text{inl}(x) \rightarrow \langle a \rangle x
\]
\[
\mid \text{inr}(y) \rightarrow \langle a \rangle y
\]
\[
j : (\forall X \to_{fs} \forall Y) \cong \forall (X \to_{fs} Y)
\]
\[
j(f) = \text{fresh } a \text{ in }
\]
\[
\quad \langle a \rangle (\lambda x. f(\langle a \rangle x) @ a)
\]

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 (+)

AMP, J. Matthiesen and J. Derikx, A Dependent Type Theory with Abstractable Names, ENTCS 312(2015)19-50.
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 \( \text{if } t = a \text{ then } t_1 \text{ else } t_2 \)

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 \( a \noin x \Rightarrow a \noin f(x, a) \)
Definitional freshness

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

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

\[ \Gamma \vdash a \# T \quad \Gamma \vdash t : T \]

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

\[ \Gamma \vdash a \# t : T \]

bunched contexts, generated by

\[ \Gamma \mapsto \Gamma(x : T) \]
\[ \Gamma \mapsto \Gamma\#(a : A) \]

definitional freshness

definitional equality
Definitional freshness

\[
\Gamma \vdash a \not\in T \quad \Gamma \vdash t : T
\]

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

\[
\Gamma \vdash a \not\in t : T
\]

Freshness info in bunched contexts gets used via:

\[
\Gamma(x : T)\Gamma' \text{ ok} \quad a, b \in \Gamma'
\]

\[
\Gamma(x : T)\Gamma' \vdash (\text{swap } a, b \text{ in } x) = x : T
\]
Definitional freshness

\[ \Gamma \vdash a \# T \quad \Gamma \vdash t : T \]

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

\[ \Gamma \vdash a \# t : T \]

definitional freshness for types:

\[ \Gamma \vdash T \quad a \in \Gamma \]

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

\[ \Gamma \vdash a \# T \]
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 $\pi \cdot e = e$ and $Y_{\pi \cdot x} = Y_x$
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

$\forall a. \varphi(a, \bar{x})$

Curry-Howard

$\leftrightarrow$

dependent name-abstraction

$[a \in A] Y_a$
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

\[ \forall a. \varphi(a, \vec{x}) \]

Curry-Howard dependent name-abstraction

\[ [a \in A] \land \neg \exists \vec{\lambda}(a, \vec{x}) \]

\[ \leftrightarrow \]

\[ \exists a \# \vec{x}, \varphi(a, \vec{x}) \]

\[ = \forall a \# \vec{x}, \varphi(a, \vec{x}) \]

‘some/any fresh $a$’
But much remains to do, e.g.

- Explore inductively defined types involving $[a : A\setminus()$ (e.g. propositional freshness).
- Dependently typed pattern-matching with name-abstraction patterns.

Difficulties:

- Is definitional freshness too weak? (cf. experience with FreshML2000)
- Name-swapping with variables of type $A\setminus$
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+⋯]
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+····]

- **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 \( (a \not\# x) = \text{degeneracy} \ (x(i/a) = x) \)
  - 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…


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:

▶ In general, computation modulo symmetry deserves further exploration.

▶ More particularly, in meta-programming/proving, permutation comes before substitution and (hence) name-abstraction before lambda-abstraction.