# Nominal Sets

## Part 2

Andrew Pitts

UNIVERSITY OF CAMBRIDGE

**Computer Laboratory**

# Outline

- The end.
- Motivation: structural recursion for data modulo $\alpha$-equivalence.
- ~~Introduction to~~ nominal sets continued

- Nominal restriction sets.
- A simply-typed $\lambda$-calculus with name-abstraction types.

# Recap

# Nominal sets

group of finite permutations of names

are sets $X$ with with a $\mathfrak{S}(\mathbb{A})$-action satisfying

**Finite support property**: for each $x \in X$, there is a finite subset $\overline{a} \subseteq \mathbb{A}$ that supports $x$:

$$a, a' \notin \overline{a} \Rightarrow (a\ a') \cdot x = x$$

**Fact:** in a nominal set every $x \in X$ possesses a *smallest* finite support, written $supp(x)$.

- $a \# x$ means $a \notin supp(x)$
- a morphism of nominal sets is a function preserving the permutation action

# The nominal set of names

$\mathbb{A}$ is a nominal set once equipped with the action

$$\pi \cdot a = \pi(a)$$

which satisfies $supp(a) = \{a\}$.

# Name abstraction

Each $X \in \mathcal{N}om$ yields a nominal set $\boxed{[\mathbb{A}]X}$ of

name-abstractions $\langle a \rangle x$ are $\sim$-equivalence classes of pairs $(a, x) \in \mathbb{A} \times X$, where

$$(a, x) \sim (a', x') \;\Leftrightarrow\; \exists b \,\#\, (a, x, a', x') \quad (b\ a) \cdot x = (b\ a') \cdot x'$$

The $\mathfrak{S}(\mathbb{A})$-action on $[\mathbb{A}]X$ is well-defined by

$$\pi \cdot \langle a \rangle x = \langle \pi(a) \rangle (\pi \cdot x)$$

and satisfies $supp(\langle a \rangle x) = supp(x) - \{a\}$, so that

$$b \,\#\, \langle a \rangle x \;\Leftrightarrow\; b = a \;\vee\; b \,\#\, x$$

# Name abstraction

Each $X \in \mathcal{N}om$ yields a nominal set $[\mathbb{A}]X$ of

name-abstractions $\langle a \rangle x$ are $\sim$-equivalence classes of pairs $(a, x) \in \mathbb{A} \times X$, where

$$(a, x) \sim (a', x') \Leftrightarrow \exists b \# (a, x, a', x') \quad (b\ a) \cdot x = (b\ a') \cdot x'$$

We get a functor $[\mathbb{A}](-) : \mathcal{N}om \to \mathcal{N}om$ sending $f \in \mathcal{N}om(X, Y)$ to $[\mathbb{A}]f \in \mathcal{N}om([\mathbb{A}]X, [\mathbb{A}]Y)$ where

$$[\mathbb{A}]f\,(\langle a \rangle x) = \langle a \rangle (f\,x)$$

# Name abstraction

$[\mathbb{A}](-) : \mathcal{N}om \to \mathcal{N}om$ is a kind of (affine) function space—it is right adjoint to the functor $\mathbb{A} \otimes (-) : \mathcal{N}om \to \mathcal{N}om$ sending $X$ to $\mathbb{A} \otimes X = \{(a, x) \mid a \# x\}$.

That explains what morphisms *into* $[\mathbb{A}]X$ look like. More important is the following characterization of morphisms *out of* $[\mathbb{A}]X$.

**Theorem.** $f \in Y^{\mathbb{A} \times X}$ factors through the quotient $\mathbb{A} \times X \twoheadrightarrow [\mathbb{A}]X$ to give an element of $Y^{([\mathbb{A}]X)}$

iff $(\forall a \in \mathbb{A})\ a \# f \ \Rightarrow\ (\forall x \in X)\ a \# f(a, x)$

iff $(\exists a \in \mathbb{A})\ a \# f \ \wedge\ (\forall x \in X)\ a \# f(a, x)$.

# Inductive datatypes in $\mathcal{N}om$

$[\mathbb{A}](-)$ has excellent exactness properties. It can be combined with $\times$, $+$ and $(-)^X$ to give functors $\mathcal{N}om \to \mathcal{N}om$ that have initial algebras.

E.g. intial algebra for

$$F(-) \triangleq \mathbb{A} + (- \times -) + [\mathbb{A}](-)$$

is $\Lambda$ with $i : F(\Lambda) \cong \Lambda$ given by $\begin{cases} i(a) & = \mathtt{V}\, a \\ i(t, t') & = \mathtt{A}\, t\, t' \\ i(\langle a \rangle t) & = \mathtt{L}\, a.t \end{cases}$

This leads to...

$\{t ::= \mathtt{V}a \mid \mathtt{A}t\,t \mid \mathtt{L}a\,t\}/\equiv_\alpha$

# $\alpha$-Structural recursion for $\Lambda$

"freshness condition for binders"

$$\frac{\begin{array}{lll} & f_1 \in X^{\mathbb{A}} & \\ & f_2 \in X^{X \times X} & \\ & f_3 \in X^{\mathbb{A} \times X} & \\ (\forall a) \;\; a \# (f_1, f_2, f_3) & \Rightarrow & (\forall x)\; a \# f_3(a, x) \quad \text{(FCB)} \end{array}}{(\exists!\;\; f \in X^{\Lambda})}$$

$$\begin{aligned} f\,(\mathtt{V}\,a) &= f_1\,a \\ f\,(\mathtt{A}\,t\,t') &= f_2(f\,t, f\,t') \\ f(\mathtt{L}\,a.\,t) &= f_3(a, f\,t) \qquad \text{if } a \# (f_1, f_2, f_3) \end{aligned}$$

# $\alpha$-Structural recursion for $\Lambda$

$$\frac{\begin{array}{lll} & f_1 \in X^{\mathbb{A}} & \\ & f_2 \in X^{X \times X} & \\ & f_3 \in X^{\mathbb{A} \times X} & \\ (\forall a) & a \# (f_1, f_2, f_3) \Rightarrow (\forall x)\ a \# f_3(a, x) & \text{(FCB)} \end{array}}{(\exists!\ f \in X^{\Lambda})}$$

$$\begin{array}{rcll} f\,(\mathtt{V}\,a) & = & f_1\,a & \\ f\,(\mathtt{A}\,t\,t') & = & f_2(f\,t, f\,t') & \\ f(\mathtt{L}\,a.\,t) & = & f_3(a, f\,t) & \text{if } a \# (f_1, f_2, f_3) \end{array}$$

Proof (for any nominal signature): see J ACM 53(2006)459-506.

Seems to capture informal usage well.

# $\alpha$-Structural recursion for $\Lambda$

$$\frac{\begin{array}{lll} f_1 & \in & X^{\mathbb{A}} \\ f_2 & \in & X^{X \times X} \\ f_3 & \in & X^{\mathbb{A} \times X} \\ (\forall a) \;\; a \# (f_1, f_2, f_3) & \Rightarrow & (\forall x) \; a \# f_3(a, x) \end{array} \quad \text{(FCB)}}{\begin{array}{lll} (\exists! & f & \in X^{\Lambda}) \end{array}}$$

$$\begin{array}{rcll} f\,(\mathtt{V}\,a) & = & f_1\,a & \\ f\,(\mathtt{A}\,t\,t') & = & f_2(f\,t, f\,t') & \\ f(\mathtt{L}\,a.\,t) & = & f_3(a, f\,t) & \text{if } a \# (f_1, f_2, f_3) \end{array}$$

Implemented in Urban & Berghofer's Nominal package for Isabelle/HOL (classical higher-order logic).

What about Type Theory? (Coq, Agda, etc.)

# $\alpha$-Structural recursion for $\Lambda$

$$\frac{\begin{array}{lll} f_1 & \in & X^{\mathbb{A}} \\ f_2 & \in & X^{X \times X} \\ f_3 & \in & X^{\mathbb{A} \times X} \end{array} \qquad (\forall a)\ a \# (f_1, f_2, f_3) \Rightarrow (\forall x)\ a \# f_3(a, x) \quad \text{(FCB)}}{(\exists!\ f \in X^{\Lambda})}$$

$$\begin{array}{rcll} f\,(\mathtt{V}\,a) & = & f_1\,a & \\ f\,(\mathtt{A}\,t\,t') & = & f_2(f\,t, f\,t') & \\ f(\mathtt{L}\,a.\,t) & = & f_3(a, f\,t) & \text{if } a \# (f_1, f_2, f_3) \end{array}$$

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

Want definition/computation to be separate from proving.

$$
\begin{aligned}
f(\mathtt{V}\,a) &= f_1\,a \\
f(\mathtt{A}\,t\,t') &= f_2(f\,t, f\,t') \\
f(\mathtt{L}\,a.t) &= f_3(a, f\,t) \quad \text{if } a \# (f_1, f_2, f_2)
\end{aligned}
$$

$= \mathtt{L}\,a'.t'$ $\qquad$ $= f_3(a', f\,t')$

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

$$
\begin{aligned}
f(\mathtt{V}\,a) &= f_1\,a \\
f(\mathtt{A}\,t\,t') &= f_2(f\,t, f\,t') \\
f(\mathtt{L}\,a.t) &= \nu a.\, f_3(a, f\,t) \quad [\, a \,\#\, (f_1, f_2, f_2)\,]
\end{aligned}
$$

$= \mathtt{L}\,a'.t'$ $= \nu a'.\, f_3(a', f\,t')$ **OK!**

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

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

$$
\begin{aligned}
f(\mathtt{V}\,a) &= f_1\,a \\
f(\mathtt{A}\,t\,t') &= f_2(f\,t, f\,t') \\
f(\mathtt{L}\,a.t) &= \nu a.\, f_3(a, f\,t) \quad [\, a \,\#\, (f_1, f_2, f_2)\,]
\end{aligned}
$$

$= \mathtt{L}\,a'.t'$

$= \nu a'.\, f_3(a', f\,t')$ **OK!**

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

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

which one?!

# Dynamic allocation

- Familiar—widely used in practice.
- Stateful: $\nu a.t$ means "add a fresh name $a'$ to the current state and return $t[a'/a]$"

# Dynamic allocation

- Familiar—widely used in practice.
- Stateful: $\nu a.t$ means "add a fresh name $a'$ to the current state and return $t[a'/a]$"
- Disrupts familiar mathematical properties of pure datatypes.

So we reject it in favour of...

# Odersky's $\nu a.(-)$ [POPL'94]

- Unfamiliar—apparently not used in practice (so far).
- Pure equational calculus, in which local scopes "intrude" rather than extrude (as per dynamic allocation):

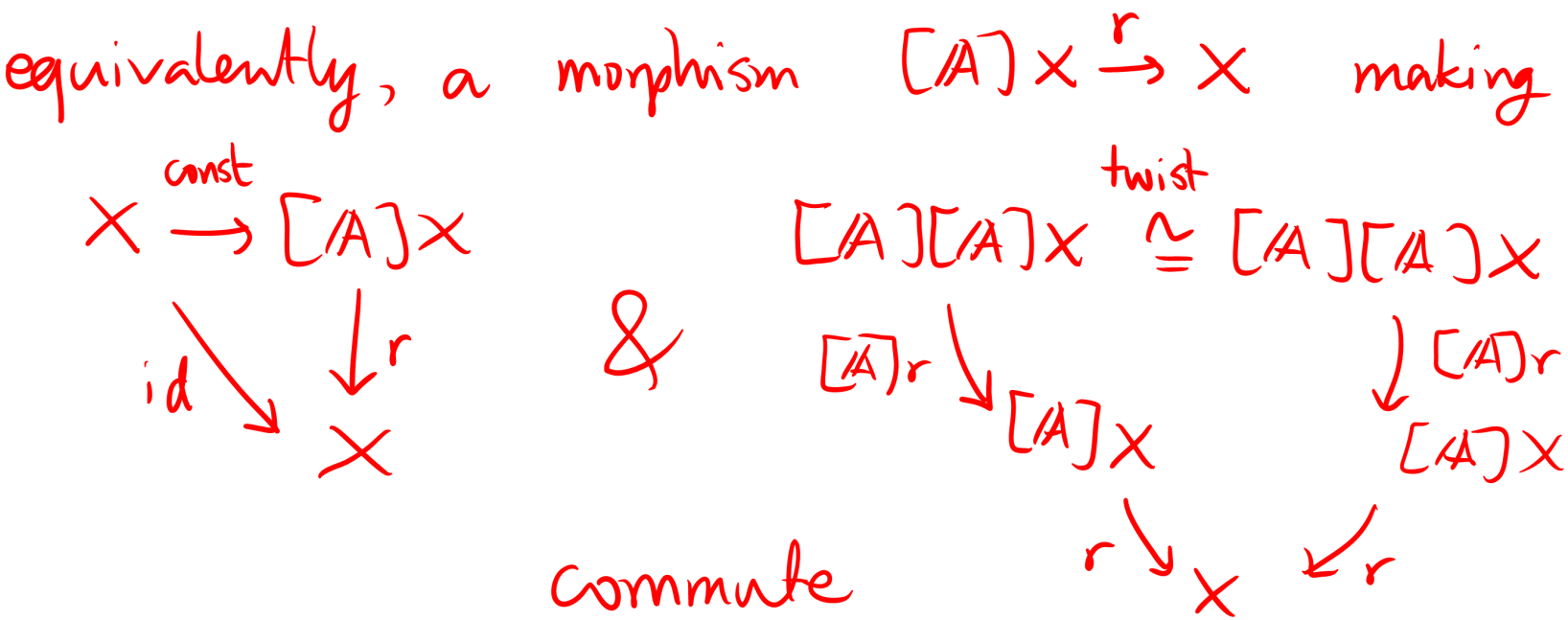
$$\begin{array}{rcll} \nu a.(\lambda x \to t) & \approx & \lambda x \to (\nu a.t) & [a \neq x] \\ \nu a.(t, t') & \approx & (\nu a.t, \nu a.t') & \end{array}$$

- Has a straightforward semantics using nominal sets equipped with a "name-restriction operation"...

# Name-restriction

A name-restriction operation on a nominal set $X$ is a morphism $(-)\setminus(-) \in \mathcal{N}om(\mathbb{A} \times X, X)$ satisfying

- $a \# a\setminus x$
- $a \# x \Rightarrow a\setminus x = x$
- $a\setminus(b\setminus x) = b\setminus(a\setminus x)$



$$\frac{\begin{array}{lll} f_1 & \in & X^{\mathbb{A}} \\ f_2 & \in & X^{X \times X} \\ f_3 & \in & X^{\mathbb{A} \times X} \end{array} \qquad (\forall a)\ \ a \# (f_1, f_2, f_3) \Rightarrow (\forall t, x)\ (a \# f_3(a, x))}{(\exists!\ \ f \in X^{\Lambda})}$$

$$\begin{array}{rcll} f\,(\mathtt{V}\,a) & = & f_1\,a & \\ f\,(\mathtt{A}\,t\,t') & = & f_2(f\,t, f\,t') & \\ f(\mathtt{L}a.t) & = & f_3(a, f\,t) & \text{if } a \# (f_1, f_2, f_3) \end{array}$$

If $X$ has a name restriction operation $(-)\setminus(-)$,
we can trivially satisfy (FCB) by using
$a\setminus(f_3\,a\,t\,x)$ in place of $f_3\,a\,t\,x$.

$$f_1 \in X^{\mathbb{A}}$$
$$f_2 \in X^{X \times X}$$
$$f_3 \in X^{\mathbb{A} \times X}$$

---

$$(\exists!\ f \in X^{\Lambda})$$

$$\begin{aligned} f\,(\mathtt{V}\,a) &= f_1\,a \\ f\,(\mathtt{A}\,t\,t') &= f_2(f\,t, f\,t') \\ f(\mathtt{L}a.\,t) &= a \nsim f_3(a, f\,t) \quad \text{if } a \,\#\, (f_1, f_2, f_3) \end{aligned}$$

Is requiring $X$ to carry a name-restriction operation much of a hindrance for applications?

$$f_1 \in X^{\mathbb{A}}$$
$$f_2 \in X^{X \times X}$$
$$f_3 \in X^{\mathbb{A} \times X}$$

---

$$(\exists!\ f \in X^{\Lambda})$$

$$\begin{aligned} f\,(\mathtt{V}\,a) &= f_1\,a \\ f\,(\mathtt{A}\,t\,t') &= f_2(f\,t, f\,t') \\ f(\mathtt{L}a.\,t) &= a \leftthreetimes f_3(a, f\,t) \quad \text{if } a \,\#\, (f_1, f_2, f_3) \end{aligned}$$

Is requiring $X$ to carry a name-restriction operation much of a hindrance for applications?

No!

# Examples of name-restriction

- For $\mathbb{N}$:

$$a \setminus n \triangleq n$$

# Examples of name-restriction

- For $\mathbb{N}$:

$$a \setminus n \triangleq n$$

- For $\mathbb{A}' \triangleq \mathbb{A} \uplus \{\texttt{anon}\}$:

$$\begin{aligned} a \setminus a &\triangleq \texttt{anon} \\ a \setminus a' &\triangleq a' \quad \text{if } a' \neq a \\ a \setminus \texttt{anon} &\triangleq \texttt{anon} \end{aligned}$$

# Examples of name-restriction

- For $\mathbb{N}$: $$a \setminus n \triangleq n$$
- For $\mathbb{A}' \triangleq \mathbb{A} \uplus \{\mathtt{anon}\}$:
$$a \setminus t \triangleq t[\mathtt{anon}/a]$$
- For $\Lambda' \triangleq \{t ::= \mathtt{V}\, a \mid \mathtt{A}\, t\, t \mid \mathtt{L}\, a.t \mid \mathtt{anon}\}$:
$$a \setminus t \triangleq t[\mathtt{anon}/a]$$

# Examples of name-restriction

- For $\mathbb{N}$: $$a \setminus n \triangleq n$$
- For $\mathbb{A}' \triangleq \mathbb{A} \uplus \{\mathtt{anon}\}$:
$$a \setminus t \triangleq t[\mathtt{anon}/a]$$
- For $\Lambda' \triangleq \{t ::= \mathtt{V}\, a \mid \mathtt{A}\, t\, t \mid \mathtt{L}\, a.t \mid \mathtt{anon}\}$:
$$a \setminus t \triangleq t[\mathtt{anon}/a]$$
- Nominal sets with name-restriction are closed under products and exponentiation by a nominal set.

# $\lambda\alpha\nu$-Calculus

is standard simply-typed $\lambda$-calculus with booleans and products, extended with:

- type of names, `Name`

# $\lambda\alpha\nu$-Calculus

is standard simply-typed $\lambda$-calculus with booleans and products, extended with:

- type of names, $\mathtt{Name}$, with terms for
  - names, $a : \mathtt{Name}$ ($a \in \mathbb{A}$)

# $\lambda\alpha\nu$-Calculus

is standard simply-typed $\lambda$-calculus with booleans and products, extended with:

- type of names, $\mathtt{Name}$, with terms for
  - names, $a : \mathtt{Name}$ ($a \in \mathbb{A}$)
  - equality test, $= : \mathtt{Name} \to \mathtt{Name} \to \mathtt{Bool}$

# $\lambda\alpha\nu$-Calculus

is standard simply-typed $\lambda$-calculus with booleans and products, extended with:

- type of names, $\mathtt{Name}$, with terms for
  - names, $a : \mathtt{Name}$ $(a \in \mathbb{A})$
  - equality test, $= : \mathtt{Name} \to \mathtt{Name} \to \mathtt{Bool}$
  - name-swapping, $\dfrac{t : T}{(a /\!/ a')t : T}$

  with type-directed computation rules, e.g.

$$(a /\!/ b)(\lambda x \to t) = \lambda x \to (a /\!/ b)(t[(a /\!/ b)x \,/\, x])$$

# $\lambda\alpha\nu$-Calculus

is standard simply-typed $\lambda$-calculus with booleans and products, extended with:

- type of names, $\mathtt{Name}$, with terms for
  - names, $a : \mathtt{Name}$ ($a \in \mathbb{A}$)
  - equality test, $= : \mathtt{Name} \to \mathtt{Name} \to \mathtt{Bool}$
  - name-swapping, $\dfrac{t : T}{(a \# a')t : T}$
  - locally scoped names $\dfrac{t : T}{\nu a.\, t : T}$ (binds $a$)

  with Odersky-style computation rules, e.g.

$$\nu a.\, \lambda x \to t = \lambda x \to \nu a.\, t$$

# $\lambda\alpha\nu$-Calculus

is standard simply-typed $\lambda$-calculus with booleans and products, extended with:

- type of names, `Name`
- name-abstraction types, `Name` . $T$

# $\lambda\alpha\nu$-Calculus

is standard simply-typed $\lambda$-calculus with booleans and products, extended with:

- type of names, $\mathtt{Name}$
- name-abstraction types, $\mathtt{Name}\,.\,T$, with terms for
  - name-abstraction, $\dfrac{t : T}{\alpha a.\, t : \mathtt{Name}\,.\,T}$ (binds $a$)

# $\lambda\alpha\nu$-Calculus

is standard simply-typed $\lambda$-calculus with booleans and products, extended with:

- type of names, $\mathtt{Name}$
- name-abstraction types, $\mathtt{Name}\,.\,T$, with terms for
  - name-abstraction, $\dfrac{t : T}{\alpha a.\, t : \mathtt{Name}\,.\,T}$ (binds $a$)
  - unbinding, $\dfrac{t : \mathtt{Name}\,.\,T \qquad t' : T'}{\mathtt{let}\; a\,.\,x = t\;\mathtt{in}\; t' : T'}$ (binds $a$ & $x$ in $t'$)

    with computation rule that uses local scoping

$$\boxed{\mathtt{let}\; a\,.\,x = \alpha a.\, t\;\mathtt{in}\; t' = \nu a.\,(t'[t/x])}$$

# $\lambda\alpha\nu$-Calculus

**Normalization.** Terms possess normal forms with respect to the computation rules that are unique up a simple structural congruence relation generated by:

$$\begin{aligned} \nu a.t &\equiv t \quad \text{if } a \notin fn(t) \\ \nu a.\nu b.t &\equiv \nu b.\nu a.t \end{aligned}$$

(Proof in the paper *Structural Recursion with Locally Scoped Names* uses Coquand's technique of evaluation to weak head normal form (whnf) combined with a "readback" of whnfs to normal forms.)

# $\lambda\alpha\nu$-Calculus

**Denotational semantics.** $\lambda\alpha\nu$-calculus has a straightforward interpretation in $\mathcal{N}om$ that is sound for the computation rules—types denote nominal sets equipped with a name-restriction operation:

$$
\begin{aligned}
[\![\mathtt{Bool}]\!] &= \mathbb{B} \\
[\![\mathtt{Name}]\!] &= \mathbb{A} \uplus \{\mathtt{anon}\} \\
[\![T \times T']\!] &= [\![T]\!] \times [\![T']\!] \\
[\![T \to T']\!] &= [\![T']\!]^{[\![T]\!]} \\
[\![\mathtt{Name}\,.\,T]\!] &= [\mathbb{A}][\![T]\!]
\end{aligned}
$$

# $\lambda\alpha\nu$-Calculus

**Denotational semantics.** $\lambda\alpha\nu$-calculus has a straightforward interpretation in $\mathcal{N}om$ that is sound for the computation rules—types denote nominal sets equipped with a name-restriction operation:

$$
\begin{aligned}
[\![\mathtt{Bool}]\!] &= \mathbb{B} \\
[\![\mathtt{Name}]\!] &= \mathbb{A} \uplus \{\mathtt{anon}\} \\
[\![T \times T']\!] &= [\![T]\!] \times [\![T']\!] \\
[\![T \to T']\!] &= [\![T']\!]^{[\![T]\!]} \\
[\![\mathtt{Name}\,.\,T]\!] &= [\mathbb{A}][\![T]\!]
\end{aligned}
$$

[[ν a. a ]]

finitely supported functions

name-abstractions

# $\lambda\alpha\nu$-Calculus

**Denotational semantics.** $\lambda\alpha\nu$-calculus has a straightforward interpretation in $\mathcal{N}om$ that is sound for the computation rules—types denote nominal sets equipped with a name-restriction operation:

$$\begin{aligned}
[\![\mathtt{Bool}]\!] &= \mathbb{B} \\
[\![\mathtt{Name}]\!] &= \mathbb{A} \uplus \{\mathtt{anon}\} \\
[\![T \times T']\!] &= [\![T]\!] \times [\![T']\!] \\
[\![T \to T']\!] &= [\![T']\!]^{[\![T]\!]} \\
[\![\mathtt{Name}\,.\,T]\!] &= [\mathbb{A}][\![T]\!]
\end{aligned}$$

Is there an analogue for $\lambda\alpha\nu$-calculus of the Friedman completeness theorem for STLC?

# $\lambda\alpha\nu$-Calculus

**Nominal datatypes.** E.g. add type $\mathtt{Lam}$ with

constructors $\begin{cases} \mathtt{V} & : & \mathtt{Name} \to \mathtt{Lam} \\ \mathtt{A} & : & (\mathtt{Lam} \times \mathtt{Lam}) \to \mathtt{Lam} \\ \mathtt{L} & : & (\mathtt{Name}\,.\,\mathtt{Lam}) \to \mathtt{Lam} \end{cases}$

iterator $\dfrac{t_1 : \mathtt{Name} \to T \quad t_2 : (T \times T) \to T \quad t_3 : (\mathtt{Name}\,.\,T) \to T}{\mathtt{lrec}\, t_1\, t_2\, t_3 : \mathtt{Lam} \to T}$

computation rules (writing $f$ for $\mathtt{lrec}\, t_1\, t_2\, t_3$)

$$\begin{cases} f(\mathtt{V}\, t) & = & t_1\, t \\ f(\mathtt{A}(t, t')) & = & t_2(f\, t, f\, t') \\ f(\mathtt{L}\, \alpha a.\, t) & = & t_3\, (\alpha a.\, f\, t) \quad \text{if } a \notin fn(t_1, t_2, t_3) \end{cases}$$

# $\lambda\alpha\nu$-Calculus

**Nominal datatypes.** E.g. add type $\mathtt{Lam}$ with

computation rules (writing $f$ for $\mathtt{lrec}\, t_1\, t_2\, t_3$)

$$\begin{cases} f(\mathtt{V}\, t) &= t_1\, t \\ f(\mathtt{A}(t, t')) &= t_2(f\, t, f\, t') \\ f(\mathtt{L}\, \alpha a.\, t) &= t_3\, (\alpha a.\, f\, t) \quad \text{if } a \notin fn(t_1, t_2, t_3) \end{cases}$$

**Main theorem:** computation of normal forms in this extension of $\lambda\alpha\nu$-calculus adequately represents $\alpha$-structurally recursive functions on $\Lambda$.

# $\lambda\alpha\nu$-Calculus

**Nominal datatypes.** E.g. add type $\mathtt{Lam}$ with

computation rules (writing $f$ for $\mathtt{lrec}\, t_1\, t_2\, t_3$)

$$\begin{cases} f(\mathtt{V}\, t) &= t_1\, t \\ f(\mathtt{A}(t, t')) &= t_2(f\, t, f\, t') \\ f(\mathtt{L}\, \alpha a.\, t) &= t_3\, (\alpha a.\, f\, t) \quad \text{if } a \notin fn(t_1, t_2, t_3) \end{cases}$$

**Main theorem:** computation of normal forms in this extension of $\lambda\alpha\nu$-calculus adequately represents $\alpha$-structurally recursive functions on $\Lambda$.

E.g. capture-avoiding substitution of $t$ for $a$ is represented by $\mathtt{lrec}\, t_1\, t_2\, t_3$ with

$$\begin{aligned} t_1 &\triangleq \mathtt{if}\, x = a\, \mathtt{then}\, t\, \mathtt{else}\, \mathtt{V}\, x \\ t_2 &\triangleq \lambda x \to \mathtt{let}(y, z) = x\, \mathtt{in}\, \mathtt{A}\, y\, z \\ t_3 &\triangleq \lambda x \to \mathtt{let}\, a\, .\, y = x\, \mathtt{in}\, \mathtt{L}\alpha b.\, (a /\!\!/ b)\, y \end{aligned}$$

```
names Var : Set

data Term : Set where                          --(possibly open) λ-terms mod α
  V : Var -> Term                              --variable
  A : (Term × Term)-> Term                     --application term
  L : (Var . Term) -> Term                     --λ-abstraction

_/_ : Term -> Var -> Term -> Term              --capture-avoiding substitution
(t / x)(V x′) = if x = x′ then t else V x′
(t / x)(A(t′ , t″)) = A((t / x )t′ , (t / x )t″)
(t / x)(L(x′ . t′)) = L(x′ . (t / x)t′)
```

# In my dreams: "nominal Agda"

```
names Var : Set

data Term : Set where                          --(possibly open) λ-terms mod α
  V : Var -> Term                              --variable
  A : (Term × Term)-> Term                     --application term
  L : (Var . Term) -> Term                     --λ-abstraction

_/_ : Term -> Var -> Term -> Term              --capture-avoiding substitution
(t / x)(V x′) = if x = x′ then t else V x′
(t / x)(A(t′ , t″)) = A((t / x )t′ , (t / x )t″)
(t / x)(L(x′ . t′)) = L(x′ . (t / x)t′)

data _==_ (t : Term) : Term -> Set where       --intensional equality
  Refl : t == t                                --is term equality mod α

eg : (x x′ : Var) ->
  ((V x) / x′)(L(x . V x′)) == L(x′ . V x)     --(λx.x′)[x/x′] = λx′.x
eg x x′ = {! !}
```

# Dependent types

- Can the $\lambda\alpha\nu$-calculus be extended from simple to dependent types?
  At the moment I do not see how to do this, because...

$$\frac{\Gamma, a : \mathtt{Name} \vdash t : T \qquad a \notin fn(T)}{\Gamma \vdash \nu a.t : T}$$

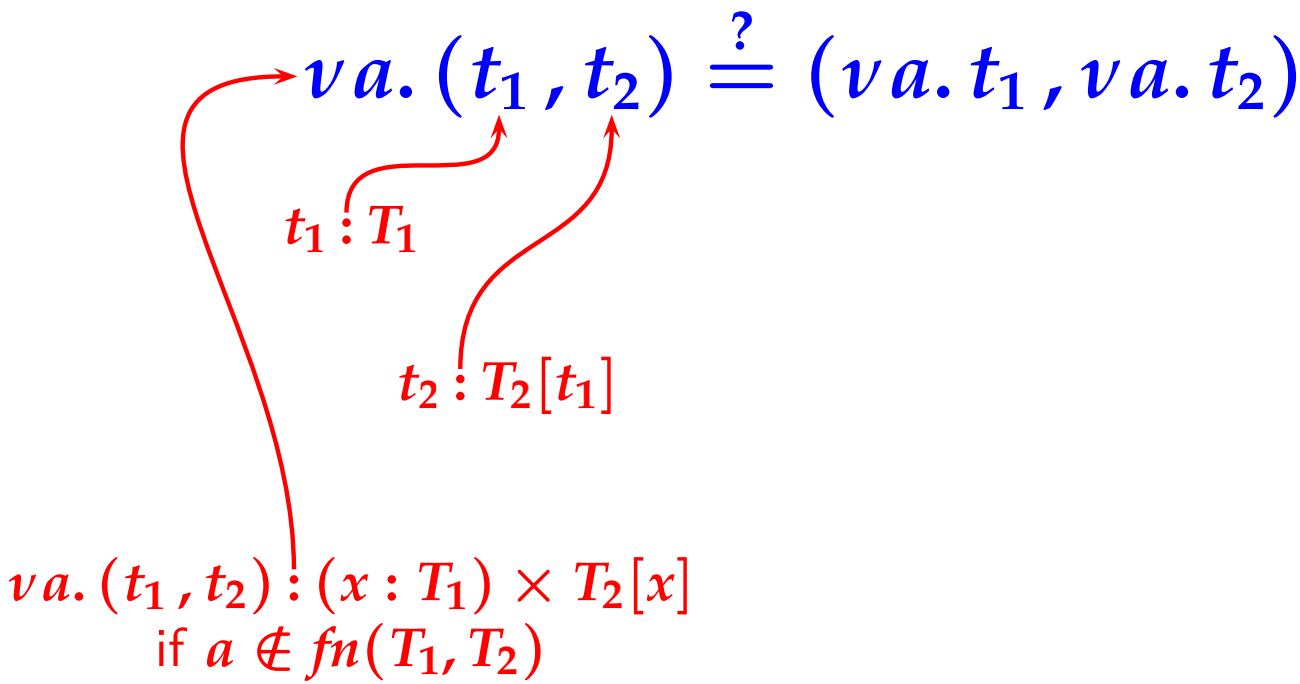
$$\frac{\Gamma, a : \mathtt{Name} \vdash t : T \qquad a \notin fn(T)}{\Gamma \vdash \nu a.\, t : T}$$

$$\nu a.\,(t_1, t_2) \stackrel{?}{=} (\nu a.\, t_1, \nu a.\, t_2)$$

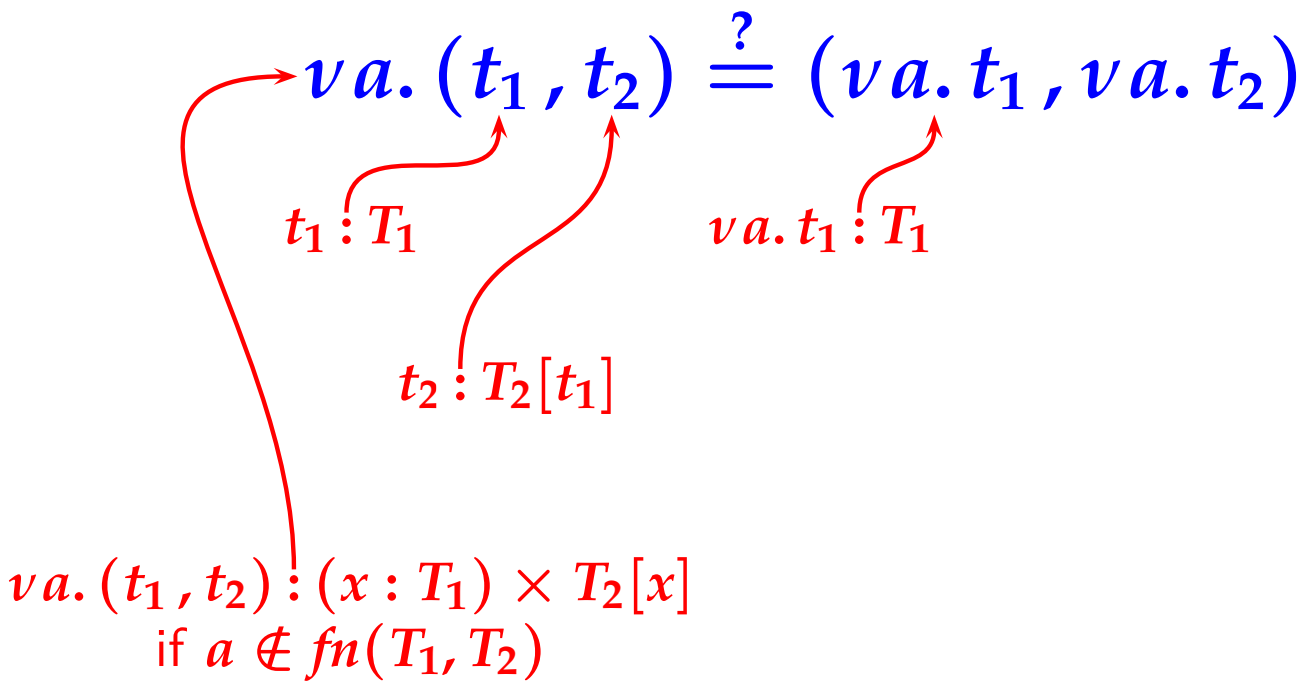
$t_1 : T_1$

$t_2 : T_2[t_1]$

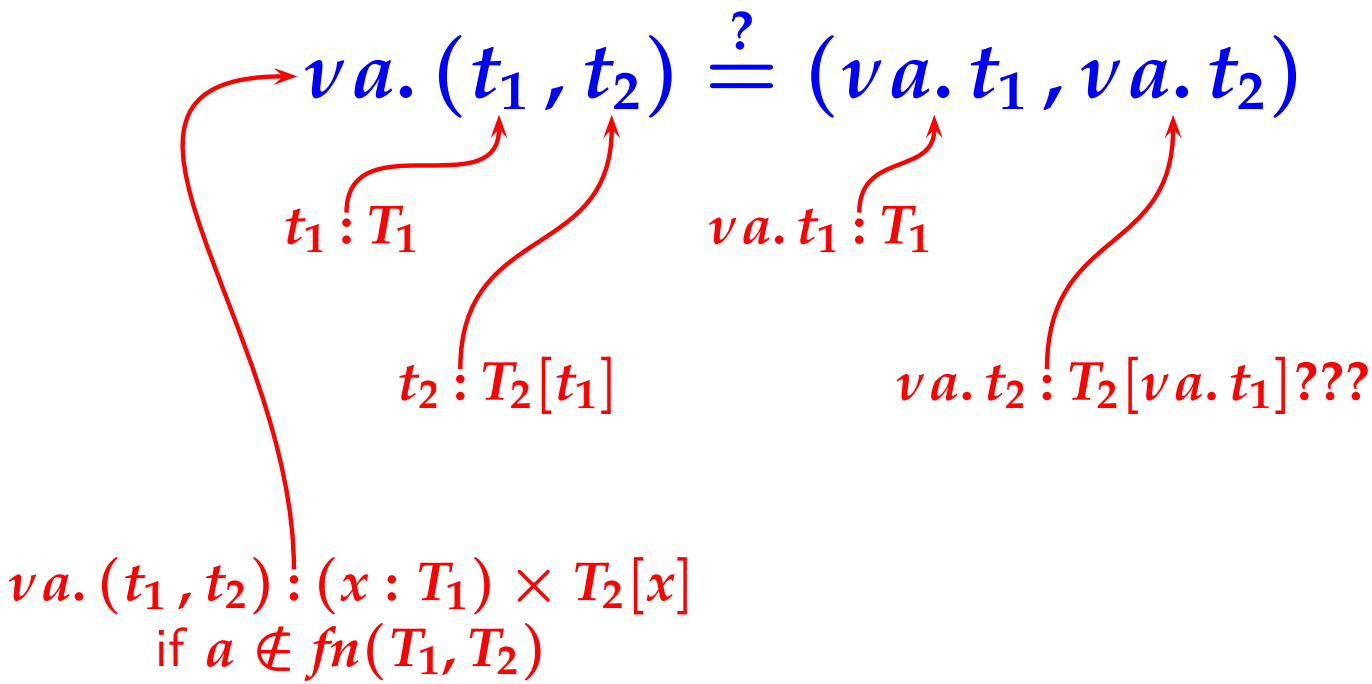
$$\frac{\Gamma, a : \mathtt{Name} \vdash t : T \qquad a \notin fn(T)}{\Gamma \vdash \nu a.\, t : T}$$



$$\frac{\Gamma, a : \mathtt{Name} \vdash t : T \qquad a \notin fn(T)}{\Gamma \vdash \nu a.\, t : T}$$



$$\frac{\Gamma, a : \mathtt{Name} \vdash t : T \qquad a \notin fn(T)}{\Gamma \vdash \nu a.\, t : T}$$



# Dependent types

- Can the $\lambda\alpha\nu$-calculus be extended from simple to dependent types?
  At the moment I do not see how to do this, because...
- In any case, is there a useful/expressive form of indexed structural induction mod $\alpha$, whether or not we try to use Odersky-style locally scoped names?

  (Recent work of Cheney on DNTT is interesting, but probably not sufficiently expressive.)