Nominal Sets

Andrew Pitts

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Mathematics of syntax

- Seems of little interest to mathematicians and of only slight interest to logicians. (?)
- Vital for computer science — because of *symbolic computation* and *automated reasoning*.
- Has yet to reach an intellectual fixed point for syntax involving scope, binding and freshness of names.

Nominal sets

- Mathematical theory of names: scope, binding, freshness.
- Simple math to do with properties invariant under permuting names.
- Originally introduced by Gabbay & AMP circa 2000, but the math goes back to 1930's set theory & logic (Fraenkel & Mostowski).

Nominal sets

- Mathematical theory of names: scope, binding, freshness.
- Simple math to do with properties invariant under permuting names.
- Originally introduced by Gabbay & AMP circa 2000, but the math goes back to 1930's set theory & logic (Fraenkel & Mostowski).
- Applications: theorem-proving tools for PL semantics; metaprogramming (within functional programming, mainly); verification.

Outline

- **Lecture 1.** Structural recursion and induction in the presence of name-binding operations.
- **Lecture 2.** Introducing the category of nominal sets.

  [Notes, chapters 1–3 +exercises]
- **Lecture 3.** Nominal algebraic data types and α-structural recursion.

  [Notes, chapters 4–5 +exercises]
- **Lecture 4.** Simply typed λ-calculus with local names and name-abstraction.

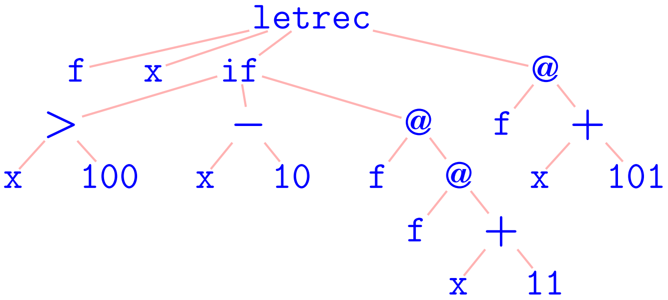
  [www.cl.cam.ac.uk/users/amp12/papers/strrls/strrls.pdf]

Lecture 1

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):



We should aim for compositional semantics of program constructions, rather than of whole programs. (Why?)

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 $\boldsymbol{\lambda}$-terms

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

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

Operations for constructing these ASTs:

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

Structural recursion for Tr

Theorem.

$$\text{Given}\quad \begin{array}{rcl} f_1 & \in & \mathbb{A} \to X \\ f_2 & \in & X \times X \to X \\ f_3 & \in & \mathbb{A} \times X \to X \end{array}$$

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

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

Structural recursion for Tr

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

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

is defined by structural recursion using

- $X = \mathbf{P_f}(\mathbb{A})$ (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

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

is defined by structural recursion using. . .

Structural recursion for *Tr*

Theorem.

$$\begin{array}{lrcl}
\text{Given} & f_1 & \in & \mathbb{A} \to X \\
 & f_2 & \in & X \times X \to X \\
 & f_3 & \in & \mathbb{A} \times X \to X \\
\hline
\text{exists unique} & \hat{f} & \in & \mathit{Tr} \to X \quad \text{satisfying}
\end{array}$$

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

Structural recursion for Tr

Theorem.

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

$f_2 \in X \times$ [illegible]

$f_3 \in$ [illegible] X

exists unique $\hat{f}$ [illegible] X satisfying

$$\text{[illegible]} = f_1\, a$$

$$\text{[illegible]}\, t')) = f_2(\hat{f}\, t, \hat{f}\, t')$$

$$\text{[illegible]}(\mathtt{L}(a, t)) = f_3(a, \hat{f}\, t)$$

Doesn't take binding into account!

Alpha-equivalence

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

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

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

E.g.

$$\begin{aligned}\mathtt{A}(\mathtt{L}(a, \mathtt{A}(\mathtt{V}\,a, \mathtt{V}\,b)), \mathtt{V}\,c) &=_\alpha \mathtt{A}(\mathtt{L}(c, \mathtt{A}(\mathtt{V}\,c, \mathtt{V}\,b)), \mathtt{V}\,c) \\ &\neq_\alpha \mathtt{A}(\mathtt{L}(b, \mathtt{A}(\mathtt{V}\,b, \mathtt{V}\,b)), \mathtt{V}\,c)\end{aligned}$$

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

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:

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

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:

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

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

<u>Not</u> an instance of structural recursion for $\boldsymbol{Tr}$.

Why is f well-defined and total?

Informal structural recursion

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

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

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

where $\left\{ \begin{array}{lcl} app & \in & D \times D \to_{cts} D \\ fun & \in & (D \to_{cts} D) \to_{cts} D \end{array} \right.$

are continuous functions satisfying...

Informal structural recursion

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

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

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

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

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

Yes! — α-structural recursion.

What about other languages with binders?

Is there a recursion principle for $\boldsymbol{\Lambda}$ that legitimises these 'definitions' of $(-)[e_1/a_1] : \boldsymbol{\Lambda} \to \boldsymbol{\Lambda}$ and $[\![-]\!] : \boldsymbol{\Lambda} \to D$ (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 $\mathbf{\Lambda}$ that legitimises these 'definitions' of $(-)[e_1/a_1] : \mathbf{\Lambda} \to \mathbf{\Lambda}$ and $[\![-]\!] : \mathbf{\Lambda} \to D$ (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.

Lecture 2

Outline

- **Lecture 1.** Structural recursion and induction in the presence of name-binding operations.
- **Lecture 2.** Introducing the category of nominal sets.

  [Notes, chapters 1–3 +exercises]
- **Lecture 3.** Nominal algebraic data types and α-structural recursion.

  [Notes, chapters 4–5 +exercises]
- **Lecture 4.** Simply typed λ-calculus with local names and name-abstraction.

  [`www.cl.cam.ac.uk/users/amp12/papers/strrls/strrls.pdf`]

Preliminaries on name-permutations

- $\mathbb{A}$ = fixed countably infinite set of names (a,b,...)

Preliminaries on name-permutations

- $\mathbb{A}$ = fixed countably infinite set of names (a,b,...)
- $\mathbf{Perm}\,\mathbb{A}$ = group of finite permutations of $\mathbb{A}$ (π, π',...)
  - π finite means: $\{a \in \mathbb{A} \mid \pi(a) \neq a\}$ is finite.
  - group: multiplication is composition of functions $\pi' \circ \pi$; identity is identity function ι.

Preliminaries on name-permutations

- $\mathbb{A}$ = fixed countably infinite set of names (a,b,...)
- $\mathbf{Perm}\,\mathbb{A}$ = group of finite permutations of $\mathbb{A}$ (π, π',...)
  - π finite means: $\{a \in \mathbb{A} \mid \pi(a) \neq a\}$ is finite.
  - group: multiplication is composition of functions $\pi' \circ \pi$; identity is identity function ι.
- swapping: $(a\ b) \in \mathbf{Perm}\,\mathbb{A}$ is the function mapping a to b, b to a and fixing all other names.

Fact: every $\pi \in \mathbf{Perm}\,\mathbb{A}$ is equal to
$$(a_1\ b_1) \circ \cdots \circ (a_n\ b_n)$$
for some a_i & b_i (with $\pi\, a_i \neq a_i \neq b_i \neq \pi\, b_i$).

Preliminaries on name-permutations

- $\mathbb{A}$ = fixed countably infinite set of names (a,b,...)
- $\mathbf{Perm}\,\mathbb{A}$ = group of finite permutations of $\mathbb{A}$ (π, π',...)
- action of $\mathbf{Perm}\,\mathbb{A}$ on a set X is a function

$$(-)\cdot(-) : \mathbf{Perm}\,\mathbb{A} \times X \to X$$

satisfying for all $x \in X$
  - $\pi' \cdot (\pi \cdot x) = (\pi' \circ \pi) \cdot x$
  - $\iota \cdot x = x$

Running example

Action of **Perm** $\mathbb{A}$ on set of ASTs for λ-terms

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

$$\begin{array}{rcl} \pi \cdot \mathtt{V}\,a & = & \mathtt{V}(\pi\,a) \\ \pi \cdot \mathtt{A}(t,t') & = & \mathtt{A}(\pi \cdot t, \pi \cdot t') \\ \pi \cdot \mathtt{L}(a,t) & = & \mathtt{L}(\pi\,a, \pi \cdot t) \end{array}$$

This respects α-equivalence and so induces an action on set of λ-terms $\Lambda = \{[t]_\alpha \mid t \in Tr\}$:

$$\pi \cdot [t]_\alpha = [\pi \cdot t]_\alpha$$

Nominal sets

are sets X with with a $\mathbf{Perm}\,\mathbb{A}$-action satisfying

Finite support property: for each $x \in X$, there is a finite subset $\overline{a} \subseteq \mathbb{A}$ that supports x, in the sense that for all $\pi \in \mathbf{Perm}\,\mathbb{A}$

$$((\forall a \in \overline{a})\ \pi\, a = a) \ \Rightarrow\ \pi \cdot x = x$$

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

Nominal sets

are sets X with with a $\mathbf{Perm}\,\mathbb{A}$-action satisfying

Finite support property: for each $x \in X$, there is a finite subset $\overline{a} \subseteq \mathbb{A}$ that supports x, in the sense that for all $\pi \in \mathbf{Perm}\,\mathbb{A}$

$$((\forall a \in \overline{a})\ \pi\, a = a) \ \Rightarrow\ \pi \cdot x = x$$

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

E.g. Tr and Λ are nominal sets—any $\overline{a}$ containing all the variables occurring (free, binding, or bound) in $t \in Tr$ supports t and (hence) $[t]_\alpha$.

Fact: for $e \in \Lambda$, $\mathit{supp}\, e = \mathbf{fv}\, e$. (See Notes, p28.)

Further examples of support

[$\mathbf{Perm}\,\mathbb{A}$ acts of sets of names $S \subseteq \mathbb{A}$ pointwise:
$\pi \cdot S \triangleq \{\pi\, a \mid a \in S\}$.]

What is a support for the following sets of names?

- $S_1 \triangleq \{a\}$
- $S_2 \triangleq \mathbb{A} - \{a\}$
- $S_3 \triangleq \{a_0, a_2, a_4, \ldots\}$, where $\mathbb{A} = \{a_0, a_1, a_2, \ldots\}$

Further examples of support

[**Perm** $\mathbb{A}$ acts of sets of names $S \subseteq \mathbb{A}$ pointwise:
$\pi \cdot S \triangleq \{\pi\, a \mid a \in S\}$.]

What is a support for the following sets of names?

- $S_1 \triangleq \{a\}$
  Answer: $\{a\}$ is smallest support.
- $S_2 \triangleq \mathbb{A} - \{a\}$
- $S_3 \triangleq \{a_0, a_2, a_4, \ldots\}$, where $\mathbb{A} = \{a_0, a_1, a_2, \ldots\}$

Further examples of support

[$\mathbf{Perm}\,\mathbb{A}$ acts of sets of names $S \subseteq \mathbb{A}$ pointwise:
$\pi \cdot S \triangleq \{\pi\, a \mid a \in S\}$.]

What is a support for the following sets of names?

- $S_1 \triangleq \{a\}$
  Answer: $\{a\}$ is smallest support.
- $S_2 \triangleq \mathbb{A} - \{a\}$
  Answer: $\{a\}$ is smallest support.
- $S_3 \triangleq \{a_0, a_2, a_4, \ldots\}$, where $\mathbb{A} = \{a_0, a_1, a_2, \ldots\}$

Further examples of support

[$\mathbf{Perm}\,\mathbb{A}$ acts of sets of names $S \subseteq \mathbb{A}$ pointwise:
$\pi \cdot S \triangleq \{\pi\, a \mid a \in S\}$.]

What is a support for the following sets of names?

- $S_1 \triangleq \{a\}$
  Answer: $\{a\}$ is smallest support.
- $S_2 \triangleq \mathbb{A} - \{a\}$
  Answer: $\{a\}$ is smallest support.
- $S_3 \triangleq \{a_0, a_2, a_4, \ldots\}$, where $\mathbb{A} = \{a_0, a_1, a_2, \ldots\}$
  Answer: $\{a_0, a_2, a_4, \ldots\}$ is a support

Further examples of support

[$\mathbf{Perm}\,\mathbb{A}$ acts of sets of names $S \subseteq \mathbb{A}$ pointwise:
$\pi \cdot S \triangleq \{\pi\, a \mid a \in S\}$.]

What is a support for the following sets of names?

- $S_1 \triangleq \{a\}$
  Answer: $\{a\}$ is smallest support.
- $S_2 \triangleq \mathbb{A} - \{a\}$
  Answer: $\{a\}$ is smallest support.
- $S_3 \triangleq \{a_0, a_2, a_4, \ldots\}$, where $\mathbb{A} = \{a_0, a_1, a_2, \ldots\}$
  Answer: $\{a_0, a_2, a_4, \ldots\}$ is a support, and so is $\{a_1, a_3, a_5, \ldots\}$—but there is no finite support. S_3 does not exist in the 'world of nominal sets'—in that world $\mathbb{A}$ is infinite, but not enumerable.

Category of nominal sets, **Nom**

- objects are nominal sets
- morphisms are functions $f \in X \to Y$ that are equivariant:

$$\pi \cdot (f\, x) = f(\pi \cdot x)$$

for all $\pi \in \mathbf{Perm}\,\mathbb{A}$, $x \in X$.

Category of nominal sets, **Nom**

Fact. **Nom** is equivalent to the Schanuel topos, a well-known Grothendieck topos classifying the geometric theory of an infinite decidable object.

So in particular **Nom** is a model of classical higher-order logic.

Category of nominal sets, **Nom**

Fact. **Nom** is equivalent to the Schanuel topos, a well-known Grothendieck topos classifying the geometric theory of an infinite decidable object.

Finite products: $X_1 \times \cdots \times X_n$ is cartesian product of sets with $\mathbf{Perm}\,\mathbb{A}$-action

$$\pi \cdot (x_1, \ldots, x_n) \triangleq (\pi \cdot x_1, \ldots, \pi \cdot x_n)$$

which satisfies

$$supp(x, \ldots, x_n) = (supp\, x_1) \cup \cdots \cup (supp\, x_n)$$

Category of nominal sets, $\mathbf{Nom}$

Fact. $\mathbf{Nom}$ is equivalent to the Schanuel topos, a well-known Grothendieck topos classifying the geometric theory of an infinite decidable object.

Coproducts are given by disjoint union.

Natural number object: $\mathbb{N} = \{0, 1, 2, \ldots\}$ with trivial $\mathbf{Perm}\,\mathbb{A}$-action: $\pi \cdot n \triangleq n$ (so $\mathit{supp}\, n = \emptyset$).

Category of nominal sets, $\mathbf{Nom}$

Fact. $\mathbf{Nom}$ is equivalent to the Schanuel topos, a well-known Grothendieck topos classifying the geometric theory of an infinite decidable object.

Exponentials: $X \to_{\mathrm{fs}} Y$ is the set of functions $f \in Y^X$ that are finitely supported w.r.t. the $\mathbf{Perm}\,\mathbb{A}$-action

$$\pi \cdot f \triangleq \lambda(x \in X) \to \pi \cdot (f(\pi^{-1} \cdot x))$$

(Can be tricky to see when $f \in Y^X$ is in $X \to_{\mathrm{fs}} Y$.)

Category of nominal sets, $\mathbf{Nom}$

Fact. $\mathbf{Nom}$ is equivalent to the Schanuel topos, a well-known Grothendieck topos classifying the geometric theory of an infinite decidable object.

Subobject classifier: $\Omega = \{\mathbf{true}, \mathbf{false}\}$ with trivial $\mathbf{Perm}\,\mathbb{A}$-action: $\pi \cdot b \triangleq b$ (so $supp\, b = \emptyset$).

($\mathbf{Nom}$ is a Boolean topos: $\Omega = \mathbf{1} + \mathbf{1}$.)

Power objects: $X \to_{\mathbf{fs}} \Omega \cong \mathbf{P}_{\mathbf{fs}}\, X$, the set of subsets $S \subseteq X$ that are finitely supported w.r.t. the $\mathbf{Perm}\,\mathbb{A}$-action

$$\pi \cdot S \triangleq \{\pi \cdot x \mid x \in S\}$$

The nominal set of names

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

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

which satisfies $\mathit{supp}\, a = \{a\}$.

N.B. $\mathbb{A}$ is not $\mathbb{N}$! Although $\mathbb{A} \in \mathbf{Set}$ is a countable, any $f \in \mathbb{N} \to_{\mathbf{fs}} \mathbb{A}$ has to satisfy

$$\{f\, n\} = \mathit{supp}(f\, n) \subseteq \mathit{supp}\, f \cup \mathit{supp}\, n = \mathit{supp}\, f$$

for all $n \in \mathbb{N}$, and so f cannot be surjective.

Nom $\not\models$ choice

Nom models classical higher-order logic, but not Hilbert's ε-operation, $\varepsilon x.\varphi(x)$ satisfying

$$(\forall x : X)\ \varphi(x) \Rightarrow \varphi(\varepsilon x.\varphi(x))$$

Theorem. There is no equivariant function $c : \{S \in \mathrm{P_{fs}}\,\mathbb{A} \mid S \neq \emptyset\} \to \mathbb{A}$ satsifying $c(S) \in S$ for all non-empty $S \in \mathrm{P_{fs}}\,\mathbb{A}$.

Proof. Suppose there were such a c. Putting $a \triangleq c\,\mathbb{A}$ and picking some $b \in \mathbb{A} - \{a\}$, we get a contradiction to $a \neq b$:

$$a = c\,\mathbb{A} = c((a\ b)\cdot\mathbb{A}) = (a\ b)\cdot c\,\mathbb{A} = (a\ b)\cdot a = b$$

Nom $\not\models$ choice

Nom models classical higher-order logic, but not Hilbert's ε-operation, $\varepsilon x.\varphi(x)$ satisfying

$$(\forall x : X)\,\varphi(x) \Rightarrow \varphi(\varepsilon x.\varphi(x))$$

In fact **Nom** does not model even very weak forms of choice, such as Dependent Choice.

Freshness

For each nominal set X, we can define a relation $\# \subseteq \mathbb{A} \times X$ of freshness:

$$a \# x \triangleq a \notin supp\, x$$

Freshness

For each nominal set X, we can define a relation $\# \subseteq \mathbb{A} \times X$ of freshness:

$$a \# x \triangleq a \notin \mathit{supp}\, x$$

- In $\mathbb{N}$, $a \# n$ always.
- In $\mathbb{A}$, $a \# b$ iff $a \neq b$.
- In Λ, $a \# t$ iff $a \notin \mathbf{fv}\, t$.
- In $X \times Y$, $a \# (x, y)$ iff $a \# x$ and $a \# y$.
- In $X \to_{\mathbf{fs}} Y$, $a \# f$ can be subtle!
  (and hence ditto for $\mathbf{P}_{\mathbf{fs}} X$)

Lecture 3

Outline

- **Lecture 1.** Structural recursion and induction in the presence of name-binding operations.
- **Lecture 2.** Introducing the category of nominal sets.

  [Notes, chapters 1–3 +exercises]
- **Lecture 3.** Nominal algebraic data types and α-structural recursion.

  [Notes, chapters 4–5 +exercises]
- **Lecture 4.** Simply typed λ-calculus with local names and name-abstraction.

  [`www.cl.cam.ac.uk/users/amp12/papers/strrls/strrls.pdf`]

Alpha-equivalence

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

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

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

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

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

Name abstraction

Each $X \in \mathbf{Nom}$ 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

$$\begin{aligned}(a, x) \sim (a', x') \;\Leftrightarrow\; & \exists b \# (a, x, a', x') \\ & (b\ a) \cdot x = (b\ a') \cdot x'\end{aligned}$$

The $\mathbf{Perm}\,\mathbb{A}$-action on $[\mathbb{A}]X$ is well-defined by

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

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

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

(See Notes, p40.)

Name abstraction

Each $X \in \mathbf{Nom}$ 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

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

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

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

Name abstraction

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

Name abstraction

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

Theorem. $f \in (\mathbb{A} \times X) \to_{\mathrm{fs}} Y$ factors through the subquotient $\{(a,x) \mid a \# f\} \subseteq \mathbb{A} \times X \twoheadrightarrow [\mathbb{A}]X$ to give a unique element of $\overline{f} \in ([\mathbb{A}]X) \to_{\mathrm{fs}} Y$ satisfying

$$\overline{f}(\langle a \rangle x) = f(a,x) \quad \text{if } a \# f$$

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

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

(Notes, p46.)

Initial algebras

- $[\mathbb{A}](-)$ has excellent exactness properties. It can be combined with $\times$, $+$ and $X \to_{\mathrm{fs}} (-)$ to give functors $\mathbf{T} : \mathbf{Nom} \to \mathbf{Nom}$ that have initial algebras $I : \mathrm{T}\,D \to D$

$$
\begin{tikzcd}
\mathrm{T}\,D \arrow[d, "I"'] & \mathrm{T}\,X \arrow[d, "F", "\text{for all}"'] \\
D & X
\end{tikzcd}
$$

Initial algebras

- $[\mathbb{A}](-)$ has excellent exactness properties. It can be combined with $\times$, $+$ and $X \to_{\mathrm{fs}} (-)$ to give functors $\mathbf{T} : \mathbf{Nom} \to \mathbf{Nom}$ that have initial algebras $I : \mathbf{T}\,D \to D$

$$
\begin{tikzcd}
\mathbf{T}\,D \arrow[r, dashed, "\mathbf{T}\,\hat{F}"] \arrow[d, "I"'] & \mathbf{T}\,X \arrow[d, "F"] \\
D \arrow[r, dashed, "\text{exists unique}", "\hat{F}"'] & X
\end{tikzcd}
$$

Initial algebras

- $[\mathbb{A}](-)$ has excellent exactness properties. It can be combined with $\times$, $+$ and $X \to_{\mathbf{fs}} (-)$ to give functors $\mathbf{T} : \mathbf{Nom} \to \mathbf{Nom}$ that have initial algebras $I : \mathbf{T}\, D \to D$
- For a wide class of such functors (nominal algebraic functors) the initial algebra D coincides with ASTs/α-equivalence.
  E.g. Λ is the initial algebra for

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

Nominal algebraic signatures

- Sorts $\mathtt{S} ::= \mathtt{N}$ name-sort (here just one, for simplicity)

$\mathtt{S} ::=$	$\mathtt{N}$	name-sort (here just one, for simplicity)
$\mid$	$\mathtt{D}$	data-sorts
$\mid$	$\mathtt{1}$	unit
$\mid$	$\mathtt{S}\,\mathtt{,}\,\mathtt{S}$	pairs
$\mid$	$\mathtt{N}\,\mathtt{.}\,\mathtt{S}$	name-binding

- Typed operations $\mathtt{op} : \mathtt{S} \to \mathtt{D}$

Signature $\boldsymbol{\Sigma}$ is specified by the stuff in red.

Nominal algebraic signatures

Example: λ-calculus

name-sort `Var` for variables, data-sort `Term` for terms, and operations

$$
\begin{aligned}
&\mathtt{V} : \mathtt{Var} \longrightarrow \mathtt{Term}\\
&\mathtt{A} : \mathtt{Term},\mathtt{Term} \longrightarrow \mathtt{Term}\\
&\mathtt{L} : \mathtt{Var}.\mathtt{Term} \longrightarrow \mathtt{Term}
\end{aligned}
$$

Nominal algebraic signatures

Example: π-calculus

name-sort `Chan` for channel names, data-sorts `Proc`, `Pre` and `Sum` for processes, prefixed processes and summations, and operations

$$
\begin{aligned}
\mathtt{S} &: \mathtt{Sum} \longrightarrow \mathtt{Proc} \\
\mathtt{Comp} &: \mathtt{Proc}, \mathtt{Proc} \longrightarrow \mathtt{Proc} \\
\mathtt{Nu} &: \mathtt{Chan}.\mathtt{Proc} \longrightarrow \mathtt{Proc} \\
\mathtt{!} &: \mathtt{Proc} \longrightarrow \mathtt{Proc} \\
\mathtt{P} &: \mathtt{Pre} \longrightarrow \mathtt{Sum} \\
\mathtt{0} &: \mathtt{1} \longrightarrow \mathtt{Sum} \\
\mathtt{Plus} &: \mathtt{Sum}, \mathtt{Sum} \longrightarrow \mathtt{Sum} \\
\mathtt{Out} &: \mathtt{Chan}, \mathtt{Chan}, \mathtt{Proc} \longrightarrow \mathtt{Pre} \\
\mathtt{In} &: \mathtt{Chan}, (\mathtt{Chan}.\mathtt{Proc}) \longrightarrow \mathtt{Pre} \\
\mathtt{Tau} &: \mathtt{Proc} \longrightarrow \mathtt{Pre} \\
\mathtt{Match} &: \mathtt{Chan}, \mathtt{Chan}, \mathtt{Pre} \longrightarrow \mathtt{Pre}
\end{aligned}
$$

Nominal algebraic signatures

Closely related notions:

- *binding signatures* of Fiore, Plotkin & Turi (LICS 1999)
- *nominal algebras* of Honsell, Miculan & Scagnetto (ICALP 2001)

N.B. all these notions of signature restrict attention to iterated, but *unary* name-binding—there are other kinds of lexically scoped binder (e.g. see Pottier's Cαml language.)

$\Sigma(\mathtt{S})$ = raw terms over Σ of sort $\mathtt{S}$

$$\frac{a \in \mathbb{A}}{a \in \Sigma(\mathtt{N})} \qquad \frac{t \in \Sigma(\mathtt{S}) \qquad \mathtt{op} : \mathtt{S} \to \mathtt{D}}{\mathtt{op}\, t \in \Sigma(\mathtt{D})} \qquad \frac{}{() \in \Sigma(\mathtt{1})}$$

$$\frac{t_1 \in \Sigma(\mathtt{S}_1) \qquad t_2 \in \Sigma(\mathtt{S}_2)}{t_1 \,,\, t_2 \in \Sigma(\mathtt{S}_1 \,,\, \mathtt{S}_2)} \qquad \frac{a \in \mathbb{A} \qquad t \in \Sigma(\mathtt{S})}{a \,.\, t \in \Sigma(\mathtt{N} \,.\, \mathtt{S})}$$

Each $\Sigma(\mathtt{S})$ is a nominal set once equipped with the obvious **Perm** $\mathbb{A}$-action—any finite set of atoms containing all those occurring in t supports $t \in \Sigma(\mathtt{S})$.

Alpha-equivalence

$$=_\alpha \subseteq \Sigma(\mathsf{S}) \times \Sigma(\mathsf{S})$$

$$\frac{a \in \mathbb{A}}{a =_\alpha a} \qquad \frac{t =_\alpha t'}{\mathsf{op}\, t =_\alpha \mathsf{op}\, t'} \qquad \frac{}{() =_\alpha ()}$$

$$\frac{t_1 =_\alpha t_1' \qquad t_2 =_\alpha t_2'}{t_1 , t_2 =_\alpha t_1' , t_2'}$$

$$\frac{(a_1\ a) \cdot t_1 =_\alpha (a_2\ a) \cdot t_2 \qquad a \# (a_1, t_1, a_2, t_2)}{a_1 . t_1 =_\alpha a_2 . t_2}$$

Alpha-equivalence

$$=_\alpha \subseteq \Sigma(\mathrm{S}) \times \Sigma(\mathrm{S})$$

Fact: $=_\alpha$ is equivariant ($t_1 =_\alpha t_2 \ \Rightarrow \pi \cdot t_1 =_\alpha \pi \cdot t_2$) and each quotient

$$\Sigma_\alpha(\mathrm{S}) \triangleq \{[t]_\alpha \mid t \in \Sigma(\mathrm{S})\}$$

is a nominal set with

$$\begin{array}{rcl} \pi \cdot [t]_\alpha & = & [\pi \cdot t]_\alpha \\ \mathit{supp}\,[t]_\alpha & = & \mathit{fn}\,t \end{array}$$

where

$$\begin{array}{rcl} \mathit{fn}(a\,.\,t) & = & \mathit{fn}\,t - \{a\} \\ \mathit{fn}(t_1\,,t_2) & = & \mathit{fn}\,t_1 \cup \mathit{fn}\,t_2 \end{array}$$

etc.

Theorem. Given a nominal algebraic signature Σ (for simplicity, assume Σ has a single data-sort $\mathtt{D}$ as well as a single name-sort $\mathtt{N}$)
$\Sigma_\alpha(\mathtt{D})$ is an initial algebra for the associated functor $\mathbf{T}_\Sigma : \mathbf{Nom} \longrightarrow \mathbf{Nom}$.

(Notes, p61.)

Theorem. Given a nominal algebraic signature Σ (for simplicity, assume Σ has a single data-sort $\mathtt{D}$ as well as a single name-sort $\mathtt{N}$)
$\Sigma_\alpha(\mathtt{D})$ is an initial algebra for the associated functor $\mathbf{T}_\Sigma : \mathbf{Nom} \to \mathbf{Nom}$.

$$\mathbf{T}_\Sigma(-) = [\![\mathtt{S}_1]\!](-) + \cdots + [\![\mathtt{S}_n]\!](-)$$

where Σ has operations $\mathtt{op}_i : \mathtt{S}_i \to D$ $(i = 1..n)$

and $[\![\mathtt{S}]\!](-) : \mathbf{Nom} \to \mathbf{Nom}$ is defined by:

$$\begin{array}{rcl}
[\![\mathtt{N}]\!](-) & = & \mathbb{A} \\
[\![\mathtt{D}]\!](-) & = & (-) \\
[\![\mathtt{1}]\!](-) & = & \mathbf{1} \\
[\![\mathtt{S}_1\,,\mathtt{S}_2]\!](-) & = & [\![\mathtt{S}_1]\!](-) \times [\![\mathtt{S}_2]\!](-) \\
[\![\mathtt{N}\,.\,\mathtt{S}]\!](-) & = & [\mathbb{A}]([\![\mathtt{S}]\!](-))
\end{array}$$

Theorem. Given a nominal algebraic signature Σ (for simplicity, assume Σ has a single data-sort $\mathtt{D}$ as well as a single name-sort $\mathtt{N}$)
$\Sigma_\alpha(\mathtt{D})$ is an initial algebra for the associated functor $\mathbf{T}_\Sigma : \mathbf{Nom} \longrightarrow \mathbf{Nom}$.

E.g. for the λ-calculus signature with operations

$\mathtt{V} : \mathtt{Var} \longrightarrow \mathtt{Term}$

$\mathtt{A} : \mathtt{Term}, \mathtt{Term} \longrightarrow \mathtt{Term}$

$\mathtt{L} : \mathtt{Var}.\mathtt{Term} \longrightarrow \mathtt{Term}$

we have

$$\mathbf{T}_\Sigma(-) = \mathbb{A} + (- \times -) + [\mathbb{A}](-)$$

Theorem. Given a nominal algebraic signature Σ (for simplicity, assume Σ has a single data-sort $\mathtt{D}$ as well as a single name-sort $\mathtt{N}$)
$\Sigma_\alpha(\mathtt{D})$ is an initial algebra for the associated enriched functor $\mathbf{T}_\Sigma : \mathbf{Nom} \to \mathbf{Nom}$.

$\mathbf{T}_\Sigma$ not only acts on equivariant (=emptily supported) functions, but also on finitely supported functions:

$$\begin{array}{rcl} (X \to_{\mathbf{fs}} Y) & \to & (\mathbf{T}_\Sigma X \to_{\mathbf{fs}} \mathbf{T}_\Sigma Y) \\ F & \mapsto & \mathbf{T}_\Sigma F \end{array}$$

α-Structural recursion

For λ-terms:

Theorem.
Given any $X \in \mathbf{Nom}$ and $\begin{cases} f_1 & \in & \mathbb{A} \to_{\mathbf{fs}} X \\ f_2 & \in & X \times X \to_{\mathbf{fs}} X \\ f_3 & \in & [\mathbb{A}]X \to_{\mathbf{fs}} X \end{cases}$

$\exists! \ \hat{f} \in \Lambda \to_{\mathbf{fs}} X$ s.t. $\begin{cases} \hat{f}\, a = f_1\, a \\ \hat{f}\,(e_1\, e_2) = f_2(\hat{f}\, e_1, \hat{f}\, e_2) \\ \hat{f}(\lambda a.e) = f_3(\langle a \rangle(\hat{f}\, e)) \quad \text{if } a \mathrel{\#} (f_1, f_2, f_3) \end{cases}$

The enriched functor $[\mathbb{A}](-) : \mathbf{Nom} \to \mathbf{Nom}$ sends $f \in X \to_{\mathbf{fs}} Y$ to $[\mathbb{A}]f \in [\mathbb{A}]X \to_{\mathbf{fs}} [\mathbb{A}]Y$ where

$$[\mathbb{A}]f\,(\langle a \rangle x) = \langle a \rangle (f\, x) \quad \text{if } a \mathrel{\#} f$$

α-Structural recursion

For λ-terms:

Theorem.
Given any $X \in \mathbf{Nom}$ and $\begin{cases} f_1 \in \mathbb{A} \to_{\mathrm{fs}} X \\ f_2 \in X \times X \to_{\mathrm{fs}} X \\ f_3 \in \mathbb{A} \times X \to_{\mathrm{fs}} X \end{cases}$ s.t.

$$(\forall a)\ a \# (f_1, f_2, f_3) \Rightarrow (\forall x)\ a \# f_3(a, x) \qquad \text{(FCB)}$$

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

Name abstraction

Recall:

Theorem. $f \in (\mathbb{A} \times X) \to_{\mathrm{fs}} Y$ factors through the subquotient $\{(a,x) \mid a \# f\} \subseteq \mathbb{A} \times X \twoheadrightarrow [\mathbb{A}]X$ to give a unique element of $\overline{f} \in ([\mathbb{A}]X) \to_{\mathrm{fs}} Y$ satisfying

$$\overline{f}(\langle a \rangle x) = f(a,x) \quad \text{if } a \# f$$

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

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

α-Structural recursion

For λ-terms:

Theorem.
Given any $X \in \mathbf{Nom}$ and $\begin{cases} f_1 & \in & \mathbb{A} \to_{\mathrm{fs}} X \\ f_2 & \in & X \times X \to_{\mathrm{fs}} X \\ f_3 & \in & \mathbb{A} \times X \to_{\mathrm{fs}} X \end{cases}$ s.t.

$$(\forall a)\ a \# (f_1, f_2, f_3) \Rightarrow (\forall x)\ a \# f_3(a, x) \qquad \text{(FCB)}$$

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

E.g. capture-avoiding substitution $(-)[e'/a'] : \Lambda \to \Lambda$ is the $\hat{f}$ for

$$\begin{array}{rcl} f_1\, a & \triangleq & \textbf{if } a = a' \textbf{ then } e' \textbf{ else } a \\ f_2(e_1, e_2) & \triangleq & e_1\, e_2 \\ f_3(a, e) & \triangleq & \lambda a.e \end{array}$$

for which (FCB) holds, since $a \# \lambda a.e$

α-Structural recursion

For λ-terms:

Theorem.
Given any $X \in \mathbf{Nom}$ and $\begin{cases} f_1 & \in & \mathbb{A} \to_{\mathrm{fs}} X \\ f_2 & \in & X \times X \to_{\mathrm{fs}} X \\ f_3 & \in & \mathbb{A} \times X \to_{\mathrm{fs}} X \end{cases}$ s.t.

$$(\forall a)\ a \# (f_1, f_2, f_3) \ \Rightarrow\ (\forall x)\ a \# f_3(a, x) \qquad \text{(FCB)}$$

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

E.g. size function $\Lambda \to \mathbb{N}$ is the $\hat{f}$ for

$$\begin{aligned} f_1\, a &\triangleq \mathbf{0} \\ f_2(n_1, n_2) &\triangleq n_1 + n_2 \\ f_3(a, n) &\triangleq n + \mathbf{1} \end{aligned}$$

for which (FCB) holds, since $a \# (n + \mathbf{1})$

α-Structural recursion

For λ-terms:

Theorem.
Given any $X \in \mathbf{Nom}$ and $\begin{cases} f_1 & \in & \mathbb{A} \to_{\mathrm{fs}} X \\ f_2 & \in & X \times X \to_{\mathrm{fs}} X \\ f_3 & \in & \mathbb{A} \times X \to_{\mathrm{fs}} X \end{cases}$ s.t.

$$(\forall a)\ a \mathbin{\#} (f_1, f_2, f_3) \ \Rightarrow\ (\forall x)\ a \mathbin{\#} f_3(a, x) \qquad \text{(FCB)}$$

$\exists!\ \hat{f} \in \Lambda \to_{\mathrm{fs}} X$ s.t. $\begin{cases} \hat{f}\, a = f_1\, a \\ \hat{f}\,(e_1\, e_2) = f_2(\hat{f}\, e_1, \hat{f}\, e_2) \\ \hat{f}(\lambda a.e) = f_3(a, \hat{f}\, e) \quad \text{if } a \mathbin{\#} (f_1, f_2, f_3) \end{cases}$

Non-example: trying to list the bound variables of a λ-term

$$\begin{aligned} f_1\, a &\triangleq \mathbf{nil} \\ f_2(\ell_1, \ell_2) &\triangleq \ell_1 \mathbin{@} \ell_2 \\ f_3(a, \ell) &\triangleq a :: \ell \end{aligned}$$

for which (FCB) does not hold, since $a \in supp(a :: \ell)$.

α-Structural recursion

For λ-terms:

Theorem.
Given any $X \in \mathbf{Nom}$ and $\begin{cases} f_1 & \in & \mathbb{A} \to_{\mathrm{fs}} X \\ f_2 & \in & X \times X \to_{\mathrm{fs}} X \\ f_3 & \in & \mathbb{A} \times X \to_{\mathrm{fs}} X \end{cases}$ s.t.

$$(\forall a)\ a \mathrel{\#} (f_1, f_2, f_3) \ \Rightarrow\ (\forall x)\ a \mathrel{\#} f_3(a, x) \qquad \text{(FCB)}$$

$\exists!\ \hat{f} \in \Lambda \to_{\mathrm{fs}} X$ s.t. $\begin{cases} \hat{f}\, a = f_1\, a \\ \hat{f}\,(e_1\, e_2) = f_2(\hat{f}\, e_1, \hat{f}\, e_2) \\ \hat{f}(\lambda a.e) = f_3(a, \hat{f}\, e) \quad \text{if } a \mathrel{\#} (f_1, f_2, f_3) \end{cases}$

Similar results hold for any nominal algebraic signature—see J ACM 53(2006)459–506.

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

Seems to capture informal usage well, but (FCB) can be tricky...

Counting bound variables

For each $e \in \Lambda$, $\boxed{\mathbf{cbv}\, e \triangleq f\, e\, \rho_0 \in \mathbb{N}}$

where we want $f \in \Lambda \to_{\mathbf{fs}} X$ with
$X = (\mathbb{A} \to_{\mathbf{fs}} \mathbb{N}) \to_{\mathbf{fs}} \mathbb{N}$ to satisfy

$$\begin{array}{rcl} f\, a\, \rho & = & \rho\, a \\ f\,(e_1\, e_2)\,\rho & = & (f\, e_1 \rho) + (f\, e_2\, \rho) \\ f(\lambda a.e)\,\rho & = & f\, e\,(\rho[a \mapsto \mathbf{1}]) \end{array}$$

and where $\rho_0 \in \mathbb{A} \to_{\mathbf{fs}} \mathbb{N}$ is $\lambda(a \in \mathbb{A}) \to \mathbf{0}$.

Counting bound variables

For each $e \in \Lambda$, $\boxed{\mathbf{cbv}\, e \triangleq f\, e\, \rho_0 \in \mathbb{N}}$

where we want $f \in \Lambda \to_{\mathrm{fs}} X$ with
$X = (\mathbb{A} \to_{\mathrm{fs}} \mathbb{N}) \to_{\mathrm{fs}} \mathbb{N}$ to satisfy

$$
\begin{array}{rcl}
f\, a\, \rho & = & \rho\, a \\
f\,(e_1\, e_2)\,\rho & = & (f\, e_1 \rho) + (f\, e_2\, \rho) \\
f(\lambda a.e)\,\rho & = & f\, e\,(\rho[a \mapsto 1])
\end{array}
$$

and where $\rho_0 \in \mathbb{A} \to_{\mathrm{fs}} \mathbb{N}$ is $\lambda(a \in \mathbb{A}) \to 0$.

Looks like we should take
$f_3(a, x) = \lambda(\rho \in \mathbb{A} \to_{\mathrm{fs}} \mathbb{N}) \to x(\rho[a \mapsto 1])$,
but this does not satisfy (FCB). Solution: take X to be a certain nominal subset of $(\mathbb{A} \to_{\mathrm{fs}} \mathbb{N}) \to_{\mathrm{fs}} \mathbb{N}$. (See Notes, p67.)

Lecture 4

Outline

- **Lecture 1.** Structural recursion and induction in the presence of name-binding operations.
- **Lecture 2.** Introducing the category of nominal sets.

  [Notes, chapters 1–3 +exercises]
- **Lecture 3.** Nominal algebraic data types and α-structural recursion.

  [Notes, chapters 4–5 +exercises]
- **Lecture 4.** Simply typed λ-calculus with local names and name-abstraction.

  [www.cl.cam.ac.uk/users/amp12/papers/strrls/strrls.pdf]

α-Structural recursion

For λ-terms:

Theorem.
Given any $X \in \mathbf{Nom}$ and $\begin{cases} f_1 \in \mathbb{A} \to_{\mathrm{fs}} X \\ f_2 \in X \times X \to_{\mathrm{fs}} X \\ f_3 \in \mathbb{A} \times X \to_{\mathrm{fs}} X \end{cases}$ s.t.

$$(\forall a)\ a \mathbin{\#} (f_1, f_2, f_3) \Rightarrow (\forall x)\ a \mathbin{\#} f_3(a, x) \qquad \text{(FCB)}$$

$\exists!\ \hat{f} \in \Lambda \to_{\mathrm{fs}} X$ s.t. $\begin{cases} \hat{f}\, a = f_1\, a \\ \hat{f}\,(e_1\, e_2) = f_2(\hat{f}\, e_1, \hat{f}\, e_2) \\ \hat{f}(\lambda a.e) = f_3(a, \hat{f}\, e) \quad \text{if } a \mathbin{\#} (f_1, f_2, f_3) \end{cases}$

Can we avoid explicit reasoning about finite support, # and (FCB) when computing 'mod α'?

Want definition/computation to be separate from proving.

$$
\begin{aligned}
\hat{f} &= f_1\, a \\
\hat{f}(e_1\, e_2) &= f_2(\hat{f}\, e_1, \hat{f}\, e_2) \\
\hat{f}(\lambda a.\, e) &= f_3(a, \hat{f}\, e) \qquad \text{if } a \,\#\, (f_1, f_2, f_2)
\end{aligned}
$$

$= \lambda a'.\, e'$ $\qquad\qquad$ $= f_3(a', \hat{f}\, e')$

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

$$\begin{array}{rcl} \hat{f}\,a & = & f_1\,a \\ \hat{f}(e_1\,e_2) & = & f_2(\hat{f}\,e_1, \hat{f}\,e_2) \\ \hat{f}(\lambda a.\,e) & = & \nu a.\,f_3(a, \hat{f}\,e) \quad [\,a \mathbin{\#} (f_1, f_2, f_2)\,] \end{array}$$

$= \lambda a'.\,e'$ $\qquad = \nu a'.\,f_3(a', \hat{f}\,e')$ ***OK!***

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

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

$$
\begin{aligned}
\hat f \; &= \; f_1\, a \\
\hat f(e_1\, e_2) \; &= \; f_2(\hat f\, e_1, \hat f\, e_2) \\
\hat f(\lambda a.\, e) \; &= \; \nu a.\, f_3(a, \hat f\, e) \quad [\; a \# (f_1, f_2, f_2)\;]
\end{aligned}
$$

$= \lambda a'.\, e'$ $\qquad$ $= \nu a'.\, f_3(a', \hat f\, e')$ **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

- Stateful: $\nu a.t$ 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.fresh-ocaml.org].

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: $\nu a.t$ 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.fresh-ocaml.org`].

Dynamic allocation

- Stateful: $\nu a.\, t$ means "add a fresh name a' to the current state and return $t[a'/a]$".

Statefulness disrupts familiar mathematical properties of pure datatypes. So we will try to reject it in favour of...

Odersky's $\nu a.(-)$

[M. Odersky, *A Functional Theory of Local Names*, 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):

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

- New: 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 $(-)\backslash(-) \in \mathbf{Nom}(\mathbb{A} \times X, X)$ satisfying

- $a \mathbin{\#} a\backslash x$
- $a \mathbin{\#} x \;\Rightarrow\; a\backslash x = x$
- $a\backslash(b\backslash x) = b\backslash(a\backslash x)$

Equivalently, a morphism $\rho : [\mathbb{A}]X \to X$ making

$$\begin{tikzcd} X \arrow[r, "\kappa"] \arrow[dr, "id_X"'] & {[\mathbb{A}]X} \arrow[d, "\rho"] \\ & X \end{tikzcd} \qquad \begin{tikzcd} {[\mathbb{A}][\mathbb{A}]X} \arrow[rr, "\delta"] \arrow[d, "{[\mathbb{A}]\rho}"'] & & {[\mathbb{A}][\mathbb{A}]X} \arrow[d, "{[\mathbb{A}]\rho}"] \\ {[\mathbb{A}]X} \arrow[dr, "\rho"'] & & {[\mathbb{A}]X} \arrow[dl, "\rho"] \\ & X & \end{tikzcd}$$

commute, where $\kappa\, x = \langle a \rangle x$ for some (or indeed any) $a \mathbin{\#} x$; and where $\delta(\langle a \rangle \langle a' \rangle x) = \langle a' \rangle \langle a \rangle x$.

Given any $X \in \mathbf{Nom}$ and $\begin{cases} f_1 & \in & \mathbb{A} \to_{\mathrm{fs}} X \\ f_2 & \in & X \times X \to_{\mathrm{fs}} X \\ f_3 & \in & \mathbb{A} \times X \to_{\mathrm{fs}} X \end{cases}$ s.t.

$$(\forall a)\ a \mathrel{\#} (f_1, f_2, f_3) \Rightarrow (\forall x)\ a \mathrel{\#} f_3(a, x) \qquad \text{(FCB)}$$

$\exists!\ \hat{f} \in \Lambda \to_{\mathrm{fs}} X$ s.t. $\begin{cases} \hat{f}\, a = f_1\, a \\ \hat{f}\,(e_1\, e_2) = f_2(\hat{f}\, e_1, \hat{f}\, e_2) \\ \hat{f}(\lambda a.e) = f_3(a, \hat{f}\, e) \quad \text{if } a \mathrel{\#} (f_1, f_2, f_3) \end{cases}$

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

Given any $X \in \mathbf{Nom}$ and $\begin{cases} f_1 & \in & \mathbb{A} \to_{\mathrm{fs}} X \\ f_2 & \in & X \times X \to_{\mathrm{fs}} X \\ f_3 & \in & \mathbb{A} \times X \to_{\mathrm{fs}} X \end{cases}$

and a restriction operation $(-)\backslash(-)$ on X,

$\exists!\ \hat{f} \in \Lambda \to_{\mathrm{fs}} X$ s.t. $\begin{cases} \hat{f}\, a = f_1\, a \\ \hat{f}\,(e_1\, e_2) = f_2(\hat{f}\, e_1, \hat{f}\, e_2) \\ \hat{f}(\lambda a.e) = a \backslash f_3(a, \hat{f}\, e) \end{cases}$

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

Not much. . .

Examples of name-restriction

- For $\mathbb{N}$: $$a \backslash n \triangleq n$$

Examples of name-restriction

- For $\mathbb{N}$: $$a \backslash n \triangleq n$$

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

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

Examples of name-restriction

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

Examples of name-restriction

- For $\mathbb{N}$: $$a \backslash n \triangleq n$$

- For $\mathbb{A}' \triangleq \mathbb{A} \uplus \{\mathbf{anon}\}$: $$a \backslash t \triangleq t[\mathbf{anon}/a]$$

- For $\Lambda' \triangleq \{t ::= \mathtt{V}\, a \mid \mathtt{A}(t\,,t) \mid \mathtt{L}(a\,.\,t) \mid \mathbf{anon}\}/{=_\alpha}$: $$a \backslash [t]_\alpha \triangleq [t[\mathbf{anon}/a]]_\alpha$$

- Nominal sets with name-restriction are closed under products, coproducts, name-abstraction and exponentiation by a nominal set.

$\lambda\alpha\nu$-Calculus

[AMP, *Structural Recursion with Locally Scoped Names*, preprint 2011, `www.cl.cam.ac.uk/users/amp12/papers/strrls/strrls.pdf`]

is standard simply-typed λ-calculus with booleans and products, extended with:

- type of names, `Name`

$\lambda\alpha\nu$-Calculus

[AMP, *Structural Recursion with Locally Scoped Names*, preprint 2011, `www.cl.cam.ac.uk/users/amp12/papers/strrls/strrls.pdf`]

is standard simply-typed λ-calculus with booleans and products, extended with:

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

$\lambda\alpha\nu$-Calculus

[AMP, *Structural Recursion with Locally Scoped Names*, preprint 2011, `www.cl.cam.ac.uk/users/amp12/papers/strrls/strrls.pdf`]

is standard simply-typed λ-calculus with booleans and products, extended with:

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

$\lambda\alpha\nu$-Calculus

[AMP, *Structural Recursion with Locally Scoped Names*, preprint 2011,
`www.cl.cam.ac.uk/users/amp12/papers/strrls/strrls.pdf`]

is standard simply-typed λ-calculus with booleans and products, extended with:

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

    with type-directed computation rules, e.g.

    $$(a \wr b)(\lambda x \to t) = \lambda x \to (a \wr b)(t[(a \wr b)x \mathbin{/} x])$$

$\lambda\alpha\nu$-Calculus

[AMP, *Structural Recursion with Locally Scoped Names*, preprint 2011, `www.cl.cam.ac.uk/users/amp12/papers/strrls/strrls.pdf`]

is standard simply-typed λ-calculus with booleans and products, extended with:

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

[AMP, *Structural Recursion with Locally Scoped Names*, preprint 2011,
`www.cl.cam.ac.uk/users/amp12/papers/strrls/strrls.pdf`]

is standard simply-typed λ-calculus with booleans and products, extended with:

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

$\lambda\alpha\nu$-Calculus

[AMP, *Structural Recursion with Locally Scoped Names*, preprint 2011, `www.cl.cam.ac.uk/users/amp12/papers/strrls/strrls.pdf`]

is standard simply-typed λ-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

[AMP, *Structural Recursion with Locally Scoped Names*, preprint 2011, `www.cl.cam.ac.uk/users/amp12/papers/strrls/strrls.pdf`]

is standard simply-typed λ-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

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

$$
\begin{array}{rcl}
[\![\mathtt{Bool}]\!] & = & \{\mathbf{true}, \mathbf{false}\} \\
[\![\mathtt{Name}]\!] & = & \mathbb{A} \uplus \{\mathbf{anon}\} \\
[\![T \times T']\!] & = & [\![T]\!] \times [\![T']\!] \\
[\![T \to T']\!] & = & [\![T]\!] \to_{\mathbf{fs}} [\![T]\!] \\
[\![\mathtt{Name}\,.\,T]\!] & = & [\mathbb{A}][\![T]\!]
\end{array}
\qquad [\![\nu a.\, a]\!]
$$

$\lambda\alpha\nu$-Calculus

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

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

(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

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

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

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

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 `Lam` with

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

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

Theorem. Computation of normal forms in this extension of $\lambda\alpha\nu$-calculus adequately represents α-structurally recursive functions on Λ.

$\lambda\alpha\nu$-Calculus

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

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

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

Theorem. Computation of normal forms in this extension of $\lambda\alpha\nu$-calculus adequately represents α-structurally recursive functions on Λ.

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

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

$\lambda\alpha\nu$-calculus as a FP language

To do: revisit FreshML using Odersky-style local names rather than dynamic allocation

```
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′)
```

'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
```

‘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...

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

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

$$\nu a.\,(e_1\,, e_2) \stackrel{?}{=} (\nu a.\, e_1\,, \nu a.\, e_2)$$

$e_1 : T_1$

$e_2 : T_2[e_1]$

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

$$\nu a.\,(e_1\,,e_2) \stackrel{?}{=} (\nu a.\, e_1\,, \nu a.\, e_2)$$

$e_1 : T_1$

$e_2 : T_2[e_1]$

$\nu a.\,(e_1\,,e_2) : (x : T_1) \times T_2[x]$
if $a \notin fn(T_1, T_2)$

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

$$\nu a.\,(e_1\,,e_2) \stackrel{?}{=} (\nu a.\, e_1\,, \nu a.\, e_2)$$

$e_1 : T_1$

$\nu a.\, e_1 : T_1$

$e_2 : T_2[e_1]$

$\nu a.\,(e_1\,,e_2) : (x : T_1) \times T_2[x]$
if $a \notin fn(T_1, T_2)$

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

$$\nu a.\,(e_1\,, e_2) \stackrel{?}{=} (\nu a.\, e_1\,, \nu a.\, e_2)$$

$e_1 : T_1$

$\nu a.\, e_1 : T_1$

$e_2 : T_2[e_1]$

$\nu a.\, e_2 : T_2[\nu a.\, e_1]$???

$\nu a.\,(e_1\,, e_2) : (x : T_1) \times T_2[x]$
if $a \notin fn(T_1, T_2)$

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 α, 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.)