Nominal Syntax and Semantics

Andrew Pitts
University of Cambridge
Computer Laboratory

Mathematics of syntax

How best to reconcile

> syntactical issues to do with name-binding and α-conversion

with a structural approach to semantics?

Specifically: improved forms of structural recursion and structural induction for syntactical structures.

Mathematics of syntax

How best to reconcile

> syntactical issues to do with name-binding and α-conversion

with a structural approach to semantics?

Specifically: improved forms of structural recursion and structural induction for syntactical structures.

Lectures provide a taster of the nominal sets model of names and name-binding. (Simplified version of [Gabbay-Pitts, 2002].)

Lecture 1

- Introduction: from structural to α-structural recursion.
- Nominal sets—first look.

Lecture 2

- Nominal sets, continued.
- α-Structural recursion—proof sketch.
- Nominal signatures.

Lecture 3

- Extended example: NBE.
- Mechanization [extra].

Lecture materials available at:

www.cl.cam.ac.uk/users/amp12/talks/appsem2005

Lecture 1

Structural recursion and induction

Structural recursion and induction

position

<u>Structural</u> recursion and induction

positionality

Structural recursion and induction

Compositionality

Structural recursion and induction

Compositionality

is crucial in [programming language] semantics

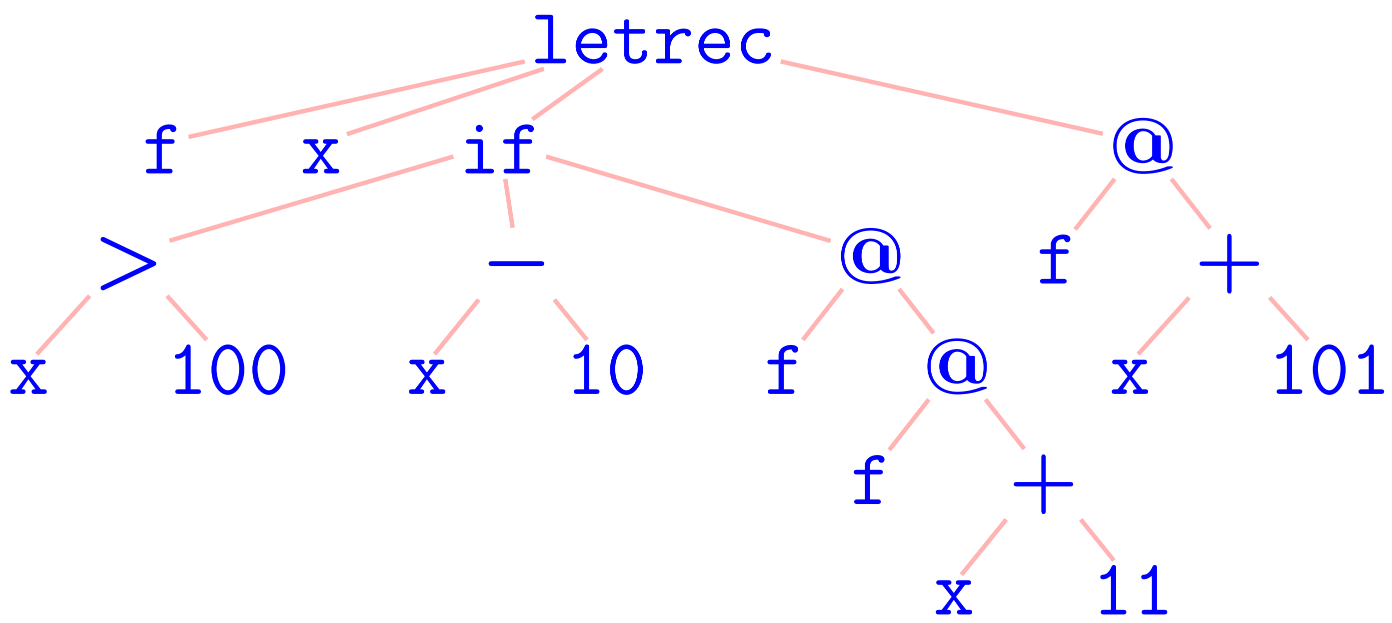
—it's preferable to give meaning to program constructions rather than just to whole programs.

Structural recursion and induction

In particular, as far as semantics is concerned, 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):



Structural recursion and induction

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.

Running example

Concrete syntax:

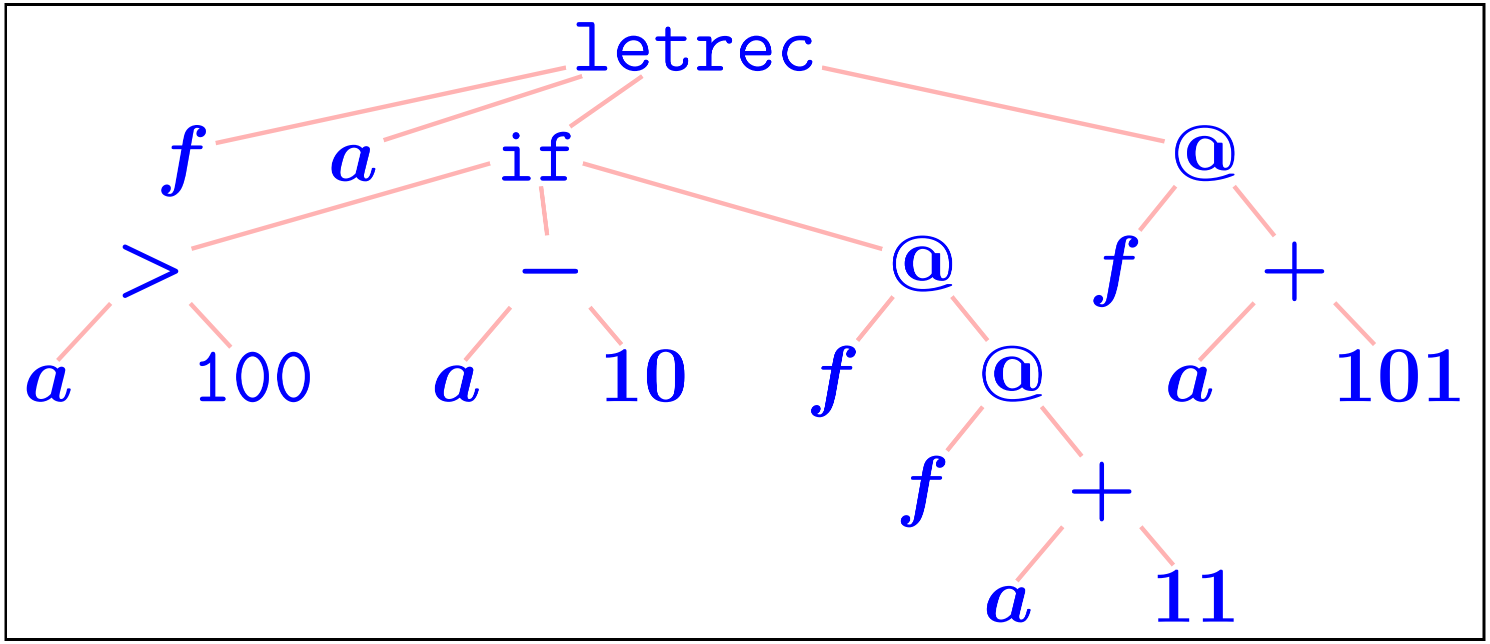
$$t ::= a \mid t\,t \mid \lambda a.t \mid \texttt{letrec}\, a\, a = t \texttt{ in } t$$

ASTs:

$$\Lambda \triangleq \mu S.(\mathbb{V} + (S \times S) + (\mathbb{V} \times S) + (\mathbb{V} \times \mathbb{V} \times S \times S))$$

where $\mathbb{V}$ is some fixed, countably infinite set (of names a of variables).

$$\begin{aligned}\texttt{letrec}\, f\, a = \;& \texttt{if}\, a > 100\, \texttt{then}\, a - 10 \\ & \texttt{else}\, f(f(a+11)) \\ \texttt{in}\, f(a+101)&\end{aligned}$$



Structural recursion for Λ

Given a set S

and functions $\begin{cases} f_{\mathrm{V}} : \mathbb{V} \to S \\ f_{\mathrm{A}} : S \times S \to S \\ f_{\mathrm{L}} : \mathbb{V} \times S \to S \\ f_{\mathrm{F}} : \mathbb{V} \times \mathbb{V} \times S \times S \to S, \end{cases}$

there is a unique function $\hat{f} : \Lambda \to S$ satisfying

$$\begin{aligned} \hat{f}\, a_1 &= f_{\mathrm{V}}\, a_1 \\ \hat{f}(t_1\, t_2) &= f_{\mathrm{A}}(\hat{f}\, t_1, \hat{f}\, t_2) \\ \hat{f}(\lambda a_1 . t_1) &= f_{\mathrm{L}}(a_1, \hat{f}\, t_1) \\ \hat{f}(\mathtt{letrec}\, a_1\, a_2 = t_1\ \mathtt{in}\ t_2) &= f_{\mathrm{F}}(a_1, a_2, \hat{f}\, t_1, \hat{f}\, t_2) \end{aligned}$$

for all $a_1, a_2 \in \mathbb{V}$ and $t_1, t_2 \in \Lambda$.

Structural recursion for Λ

A more complicated version ("primitive recursion" instead of "iteration") is derivable:

$$
\begin{aligned}
\hat{g}\,a_1 &= g_{\mathrm{V}}\,a_1\\
\hat{g}(t_1\,t_2) &= g_{\mathrm{A}}(t_1, t_2, \hat{g}\,t_1, \hat{g}\,t_2)\\
\hat{g}(\lambda a_1.t_1) &= g_{\mathrm{L}}(a_1, t_1, \hat{g}\,t_1)\\
\hat{g}(\mathtt{letrec}\,a_1\,a_2 = t_1\ \mathtt{in}\ t_2) &= g_{\mathrm{F}}(a_1, a_2, t_1, t_2, \hat{g}\,t_1, \hat{g}\,t_2)
\end{aligned}
$$

$$
\begin{aligned}
\hat{f}\,a_1 &= f_{\mathrm{V}}\,a_1\\
\hat{f}(t_1\,t_2) &= f_{\mathrm{A}}(\hat{f}\,t_1, \hat{f}\,t_2)\\
\hat{f}(\lambda a_1.t_1) &= f_{\mathrm{L}}(a_1, \hat{f}\,t_1)\\
\hat{f}(\mathtt{letrec}\,a_1\,a_2 = t_1\ \mathtt{in}\ t_2) &= f_{\mathrm{F}}(a_1, a_2, \hat{f}\,t_1, \hat{f}\,t_2)
\end{aligned}
$$

for all $a_1, a_2 \in \mathbb{V}$ and $t_1, t_2 \in \Lambda$.

Finite set of free variables $\mathit{fv}\,t$ of an AST t:

$$
\begin{aligned}
\mathit{fv}\,a_1 &\triangleq \{a_1\}\\
\mathit{fv}(t_1\,t_2) &\triangleq (\mathit{fv}\,t_1) \cup (\mathit{fv}\,t_2)\\
\mathit{fv}(\lambda a_1.t_1) &\triangleq (\mathit{fv}\,t_1) - \{a_1\}\\
\mathit{fv}(\texttt{letrec}\,a_1\,a_2 = t_1\ \texttt{in}\ t_2) &\triangleq (\mathit{fv}\,t_1) - \{a_1, a_2\}\\
&\qquad \cup (\mathit{fv}\,t_2) - \{a_1\}
\end{aligned}
$$

Finite set of free variables $fv\,t$ of an AST t:

$$\begin{aligned} fv\,a_1 &\triangleq \{a_1\} \\ fv(t_1\,t_2) &\triangleq (fv\,t_1) \cup (fv\,t_2) \\ fv(\lambda a_1.t_1) &\triangleq (fv\,t_1) - \{a_1\} \\ fv(\mathtt{letrec}\,a_1\,a_2 = t_1\ \mathtt{in}\ t_2) &\triangleq (fv\,t_1) - \{a_1, a_2\} \\ &\quad \cup (fv\,t_2) - \{a_1\} \end{aligned}$$

defined by structural recursion using

- $S \triangleq P_{\mathrm{fin}}(\mathbb{V})$ (finite sets of variables),
- $f_{\mathrm{V}}\,a_1 \triangleq \{a_1\}$,
- $f_{\mathrm{A}}(A_1, A_2) \triangleq A_1 \cup A_2$,
- $f_{\mathrm{L}}(a_1, A_1) \triangleq A_1 - \{a_1\}$,
- $f_{\mathrm{F}}(a_1, a_2, A_1, A_2) \triangleq (A_1 - \{a_1, a_2\}) \cup (A_2 - \{a_1\})$.

Finite set of all variables $var\, t$ of an AST t:

$$
\begin{aligned}
var\, a_1 &\triangleq \{a_1\} \\
var(t_1\, t_2) &\triangleq (var\, t_1) \cup (var\, t_2) \\
var(\lambda a_1 . t_1) &\triangleq (var\, t_1) \cup \{a_1\} \\
var(\mathtt{letrec}\, a_1\, a_2 = t_1\ \mathtt{in}\ t_2) &\triangleq \{a_1, a_2\} \cup (var\, t_1) \\
&\qquad \cup (var\, t_2)
\end{aligned}
$$

$t\{b/a\} \triangleq$ replace all occurrences of a with b in an AST t:

- $a_1\{b/a\} \triangleq$ if $a_1 = a$ then b else a_1
- $(t_1\, t_2)\{b/a\} \triangleq (t_1\{b/a\})\,(t_2\{b/a\})$
- $(\lambda a_1 .\, t_1)\{b/a\} \triangleq \lambda a_1\{b/a\} .\, t_1\{b/a\}$
- $(\mathtt{letrec}\; a_1\, a_2 = t_1 \;\mathtt{in}\; t_2 \triangleq$ $\mathtt{letrec}\,(a_1\{b/a\})(a_2\{b/a\}) = t_1\{b/a\} \;\mathtt{in}\; t_2\{b/a\}$

Structural recursion for Λ

Given a set S and functions
$$\begin{cases} f_{\mathrm{V}} : \mathbb{V} \to S \\ f_{\mathrm{A}} : S \times S \to S \\ f_{\mathrm{L}} : \mathbb{V} \times S \to S \\ f_{\mathrm{F}} : \mathbb{V} \times \mathbb{V} \times S \times S \to S, \end{cases}$$
there is a unique function $\hat{f} : \Lambda \to S$ satisfying

$$\begin{aligned} \hat{f}\, a_1 &= f_{\mathrm{V}}\, a_1 \\ \hat{f}(t_1\, t_2) &= f_{\mathrm{A}}(\hat{f}\, t_1, \hat{f}\, t_2) \\ \hat{f}(\lambda a_1 . t_1) &= f_{\mathrm{L}}(a_1, \hat{f}\, t_1) \\ \hat{f}(\mathtt{letrec}\, a_1\, a_2 = t_1\ \mathtt{in}\ t_2) &= f_{\mathrm{F}}(a_1, a_2, \hat{f}\, t_1, \hat{f}\, t_2) \end{aligned}$$

for all $a_1, a_2 \in \mathbb{V}$ and $t_1, t_2 \in \Lambda$.

Structural recursion for Λ

Given a set S and functions
$$\begin{cases} f_{\mathrm{V}} : \mathbb{V} \to S \\ f_{\mathrm{A}} : S \times S \to S \\ f_{\mathrm{L}} : \mathbb{V} \times S \to S \\ f_{\mathrm{F}} : \mathbb{V} \times \mathbb{V} [illegible] \to S, \end{cases}$$

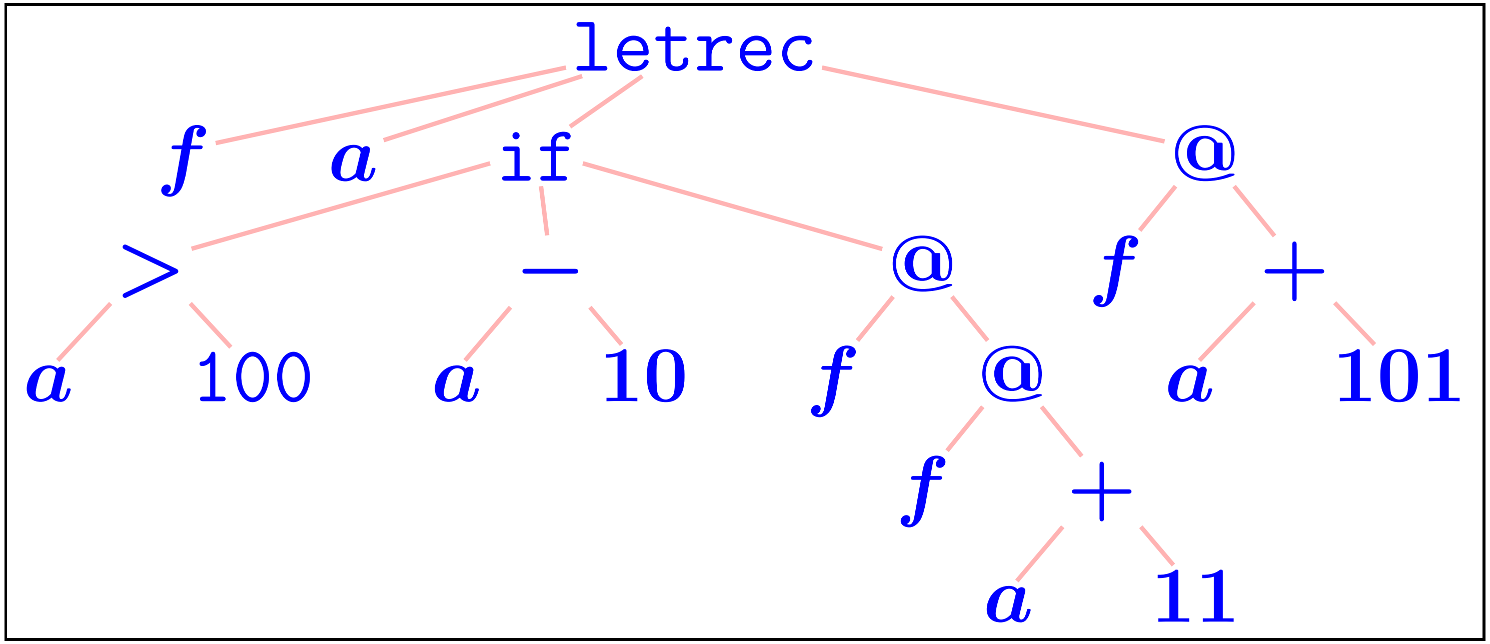
there is a unique functi[illegible] $\to S$ satisfying

[illegible]	$=$	$f_{\mathrm{V}}\, a_1$
[illegible] $t_2)$	$=$	$f_{\mathrm{A}}(\hat{f}\, t_1, \hat{f}\, t_2)$
[illegible] $(\lambda a_1.t_1)$	$=$	$f_{\mathrm{L}}(a_1, \hat{f}\, t_1)$
$\hat{f}($[illegible] $a_2 = t_1 \ \mathtt{in}\ t_2)$	$=$	$f_{\mathrm{F}}(a_1, a_2, \hat{f}\, t_1, \hat{f}\, t_2)$

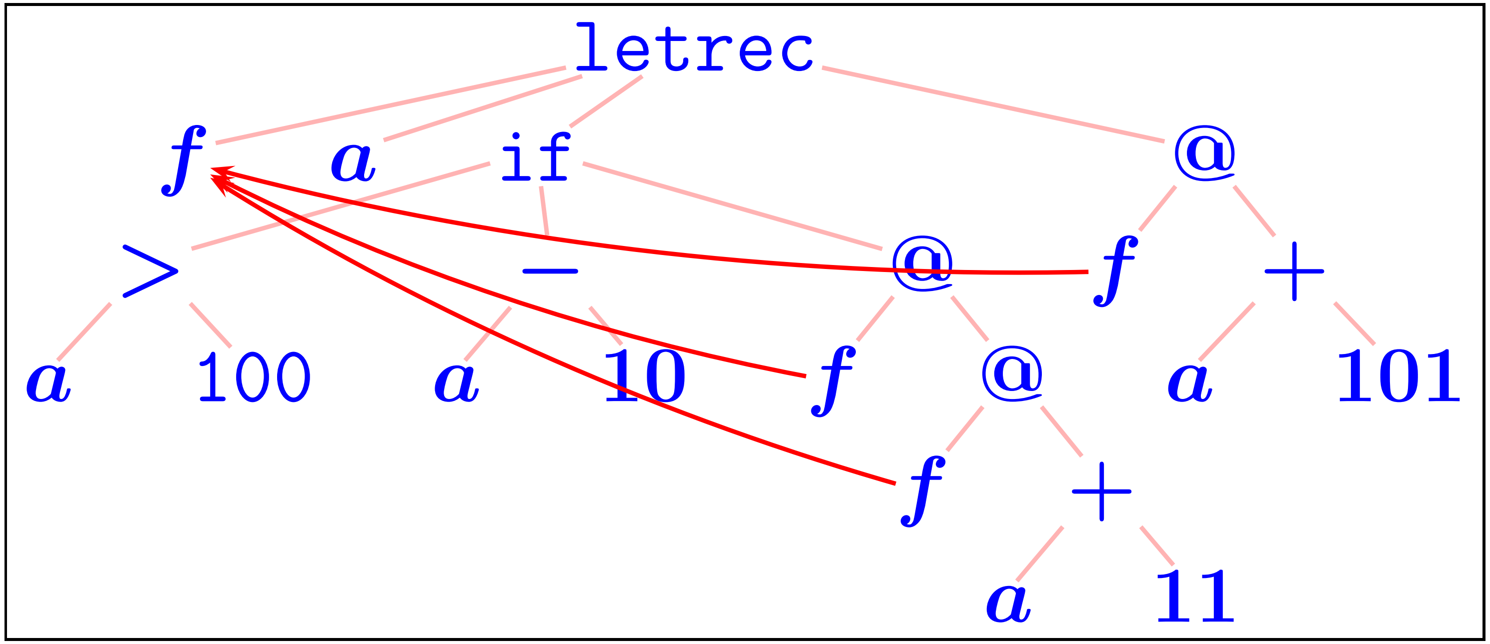
[illegible] $a_1, a_2 \in \mathbb{V}$ and $t_1, t_2 \in \Lambda$.

Doesn't take binding into account!

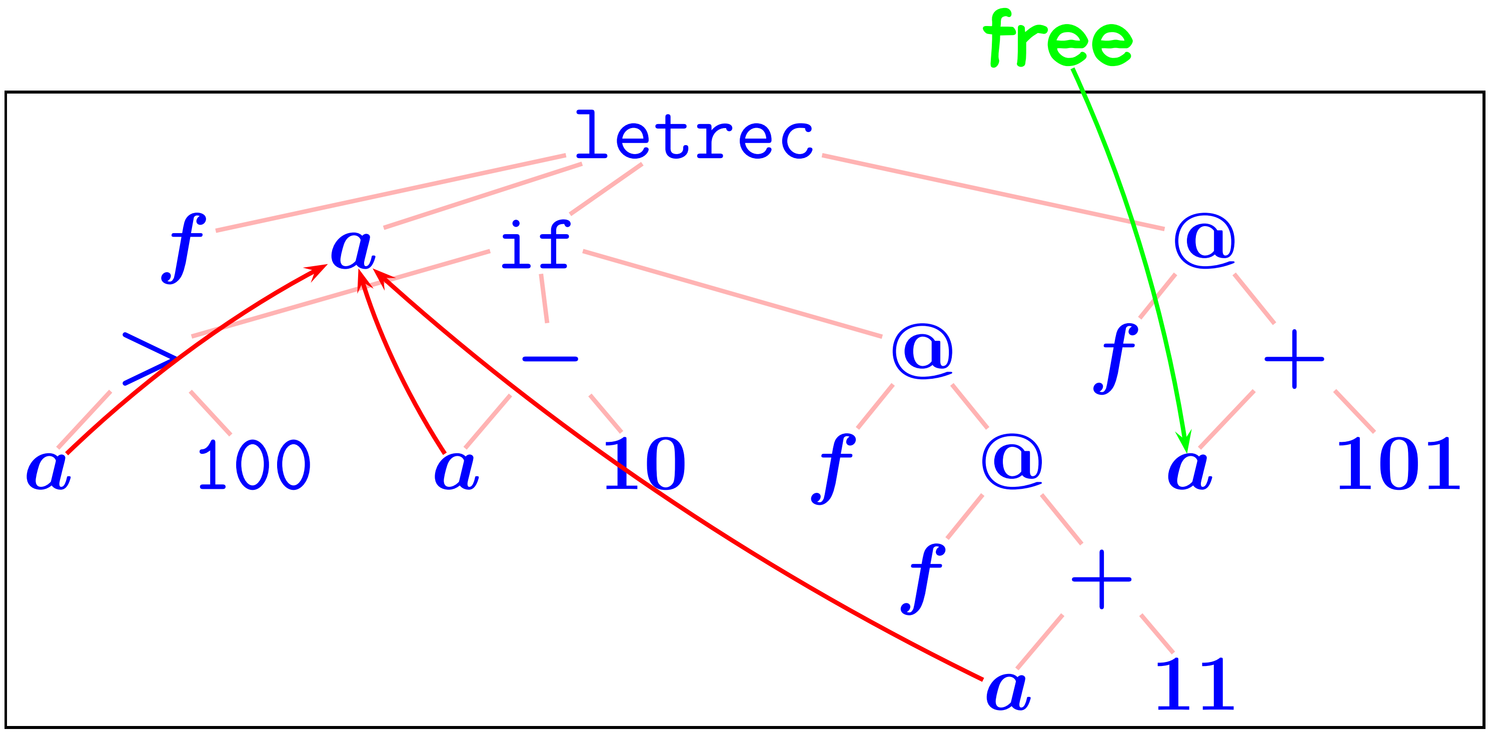
$$\begin{aligned}\texttt{letrec}\, f\, a = \;& \texttt{if}\, a > 100\, \texttt{then}\, a - 10 \\ & \texttt{else}\, f(f(a + 11)) \\ \texttt{in}\, f(a + 101)&\end{aligned}$$



$$\texttt{letrec}\ f\ a = \texttt{if}\ a > 100\ \texttt{then}\ a - 10$$
$$\texttt{else}\ f(f(a + 11))$$
$$\texttt{in}\ f(a + 101)$$



$$\texttt{letrec}\ f\ a = \ \texttt{if}\ a > 100\ \texttt{then}\ a - 10$$
$$\texttt{else}\ f(f(a + 11))$$
$$\texttt{in}\ f(a + 101)$$



α-Equivalence

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

$$\frac{a \in \mathbb{V}}{a =_\alpha a} \qquad \frac{t_1 =_\alpha t'_1 \qquad t_2 =_\alpha t'_2}{t_1\, t_2 =_\alpha t'_1\, t'_2}$$

$$\frac{t_1\{a''_1/a_1\} =_\alpha t'_1\{a''_1/a'_1\} \qquad a''_1 \notin var(a_1, t_1, a'_1, t'_1)}{\lambda a_1.\, t_1 =_\alpha \lambda a'_1.\, t'_1}$$

$$\frac{\begin{array}{c} t_1\{a''_1, a''_2/a_1, a_2\} =_\alpha t'_1\{a''_1, a''_2/a'_1, a'_2\} \\ t_2\{a''_1/a_1\} =_\alpha t'_2\{a''_1/a'_1\} \\ a''_1 \neq a''_2 \qquad a''_1, a''_2 \notin var(a_1, a_2, t_1, t_2, a'_1, a'_2, t'_1, t'_2) \end{array}}{\mathtt{letrec}\, a_1\, a_2 = t_1 \;\mathtt{in}\; t_2 =_\alpha \mathtt{letrec}\, a'_1\, a'_2 = t'_1 \;\mathtt{in}\; t'_2}$$

Exercise: prove that $=_\alpha$ is transitive (and reflexive and symmetric).

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

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

$$\frac{t_1\{a_1''/a_1\} =_\alpha t_1'\{a_1''/a_1'\} \qquad a_1'' \notin var(a_1, t_1, a_1', t_1')}{\lambda a_1.\, t_1 =_\alpha \lambda a_1'.\, t_1'}$$

$$\frac{\begin{array}{c} t_1\{a_1'', a_2''/a_1, a_2\} =_\alpha t_1'\{a_1'', a_2''/a_1', a_2'\} \\ t_2\{a_1''/a_1\} =_\alpha t_2'\{a_1''/a_1'\} \\ a_1'' \neq a_2'' \qquad a_1'', a_2'' \notin var(a_1, a_2, t_1, t_2, a_1', a_2', t_1', t_2') \end{array}}{\texttt{letrec}\, a_1\, a_2 = t_1 \;\texttt{in}\; t_2 =_\alpha \texttt{letrec}\, a_1'\, a_2' = t_1' \;\texttt{in}\; t_2'}$$

Abstract syntax / α

Dealing with issues to do with binders and α-conversion is

- irritating (want to get on with more interesting aspects of semantics!)
- pervasive (very many languages involve binding operations; cf. POPLMark Challenge [TPHOLs '05])
- difficult to formalise/mechanise without losing sight of common informal practice:

Abstract syntax / α

Dealing with issues to do with binders and α-conversion is

- irritating (want to get on with more interesting aspects of semantics!)
- pervasive (very many languages involve binding operations; cf. POPLMark Challenge [TPHOLs '05])
- difficult to formalise/mechanise without losing sight of common informal practice:

"We identify expressions up to α-equivalence"...

Abstract syntax / α

Dealing with issues to do with binders and α-conversion is

- irritating (want to get on with more interesting aspects of semantics!)
- pervasive (very many languages involve binding operations; cf. POPLMark Challenge [TPHOLs '05])
- difficult to formalise/mechanise without losing sight of common informal practice:

> "We identify expressions up to α-equivalence"...
> ...and then forget about it, referring to α-equivalence classes $e = [t]_\alpha$ only via representatives, t.

For example...

E.g. – capture-avoiding substitution

$(a := e)e_1$ = substitute $e \in \Lambda/{=_\alpha}$ for all free occurrences of a in $e_1 \in \Lambda/{=_\alpha}$, avoiding capture of free variables in e by binders in e_1.

E.g. – capture-avoiding substitution

- $(a := e)a_1 \triangleq$ if $a_1 = a$ then e else a_1
- $(a := e)(e_1\, e_2) \triangleq ((a := e)e_1)((a := e)e_2)$
- $(a := e)(\lambda a_1.e_1) \triangleq$

  if $a_1 \notin fv(a, e)$ then $\lambda a_1.(a := e)e_1$

  else don't care!
- $(a := e)(\texttt{letrec}\, a_1\, a_2 = e_1\ \texttt{in}\ e_2) \triangleq$?

E.g. – capture-avoiding substitution

- $(a := e)a_1 \triangleq$ if $a_1 = a$ then e else a_1
- $(a := e)(e_1\, e_2) \triangleq ((a := e)e_1)((a := e)e_2)$
- $(a := e)(\lambda a_1.e_1) \triangleq$

  if $a_1 \notin fv(a, e)$ then $\lambda a_1.(a := e)e_1$

  else don't care!
- $(a := e)(\mathtt{letrec}\, a_1\, a_2 = e_1 \;\mathtt{in}\; e_2) \triangleq$

  if $a_1, a_2 \notin fv(a, e)$ & $a_2 \notin fv(a_1, e_2)$

  then $\mathtt{letrec}\, a_1\, a_2 = (a := e)e_1 \;\mathtt{in}\; (a := e)e_2$

  else don't care!

E.g. – capture-avoiding substitution

- $(a := e)a_1 \triangleq$ if $a_1 = a$ then e else a_1
- $(a := e)(e_1\, e_2) \triangleq ((a := e)e_1)((a := e)e_2)$
- $(a := e)(\lambda a_1.e_1) \triangleq$

  if $a_1 \notin fv(a, e)$ then $\lambda a_1.(a := e)e_1$

  else don't care!
- $(a := e)(\mathtt{letrec}\, a_1\, a_2 = e_1 \;\mathtt{in}\; e_2) \triangleq$

  if $a_1, a_2 \notin fv(a, e)\ \&\ a_2 \notin fv(a_1, e_2)$

  then $\mathtt{letrec}\, a_1\, a_2 = (a := e)e_1 \;\mathtt{in}\; (a := e)e_2$

  else don't care!

Does uniquely specify a well-defined function on α-equivalence classes, $(a := e)(-) : \Lambda/\alpha \to \Lambda/\alpha$, but not via an obvious, structurally recursive definition of a function $\hat{f} : \Lambda \to \Lambda$ respecting α-equivalence.

E.g. – denotational semantics

of Λ/α in some suitable domain D:

- $[\![a_1]\!]\rho \triangleq \rho(a_1)$
- $[\![e_1\, e_2]\!]\rho \triangleq app([\![e_1]\!]\rho, [\![e_2]\!]\rho)$
- $[\![\lambda a_1.e_1]\!]\rho \triangleq fun(\lambda d \in D.\ [\![e_1]\!](\rho[a_1 \mapsto d]))$
- $[\![\mathtt{letrec}\, a_1\, a_2 = e_1\ \mathtt{in}\ e_2]\!]\rho \triangleq \cdots$

where

- ρ ranges over environments mapping variables to elements of D
- D comes equipped with continuous functions $app : D \times D \to D$ and $fun : (D \to D) \to D$.

E.g. – denotational semantics

of Λ/α in some suitable domain D:

- $[\![a_1]\!]\rho \triangleq \rho(a_1)$
- $[\![e_1\, e_2]\!]\rho \triangleq app([\![e_1]\!]\rho, [\![e_2]\!]\rho)$
- $[\![\lambda a_1.e_1]\!]\rho \triangleq fun(\lambda d \in D.\ [\![e_1]\!](\rho[a_1 \mapsto d]))$
- $[\![\mathtt{letrec}\, a_1\, a_2 = e_1\ \mathtt{in}\ e_2]\!]\rho \triangleq \cdots$

Why is this (very standard) definition independent of the choice of bound variable a_1?

E.g. – denotational semantics

of Λ/α in some suitable domain D:

- $[\![a_1]\!]\rho \triangleq \rho(a_1)$
- $[\![e_1\, e_2]\!]\rho \triangleq app([\![e_1]\!]\rho, [\![e_2]\!]\rho)$
- $[\![\lambda a_1.e_1]\!]\rho \triangleq fun(\lambda d \in D.\ [\![e_1]\!](\rho[a_1 \mapsto d]))$
- $[\![\texttt{letrec}\, a_1\, a_2 = e_1\ \texttt{in}\ e_2]\!]\rho \triangleq \cdots$

In this case we can use ordinary structural recursion to first define denotations of ASTs and then prove that they respect α-equivalence.

But is there a quicker way, working directly with ASTs/α?

α-Structural recursion

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

α-Structural recursion

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

Yes! — α-structural recursion
(and induction too—see lecture notes).

α-Structural recursion

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

Yes! — α-structural recursion
(and induction too—see lecture notes).

What about other languages with binders?

α-Structural recursion

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

Yes! — α-structural recursion
(and induction too—see lecture notes).

What about other languages with binders?

Yes! — available for any nominal signature.

α-Structural recursion

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

Yes! — α-structural recursion
(and induction too—see lecture notes).

What about other languages with binders?

Yes! — available for any nominal signature.

Great. What's the catch?

α-Structural recursion

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

Yes! — α-structural recursion
(and induction too—see lecture notes).

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.

Pause

Running example (reminder)

Concrete syntax:

$$t ::= a \mid t\,t \mid \lambda a.t \mid \texttt{letrec}\; a\, a = t \;\texttt{in}\; t$$

ASTs:

$$\Lambda \triangleq \mu S.(\mathbb{V} + (S \times S) + (\mathbb{V} \times S) + (\mathbb{V} \times \mathbb{V} \times S \times S))$$

where $\mathbb{V}$ is some fixed, countably infinite set (of names a of variables).

Structural recursion for Λ

Given a set S

and functions
$$\begin{cases} f_{\mathrm{V}} : \mathbb{V} \to S \\ f_{\mathrm{A}} : S \times S \to S \\ f_{\mathrm{L}} : \mathbb{V} \times S \to S \\ f_{\mathrm{F}} : \mathbb{V} \times \mathbb{V} \times S \times S \to S, \end{cases}$$

there is a unique function $\hat{f} : \Lambda \to S$ satisfying

$$\boxed{\begin{array}{rcl} \hat{f}\, a_1 & = & f_{\mathrm{V}}\, a_1 \\ \hat{f}(t_1\, t_2) & = & f_{\mathrm{A}}(\hat{f}\, t_1, \hat{f}\, t_2) \\ \hat{f}(\lambda a_1 . t_1) & = & f_{\mathrm{L}}(a_1, \hat{f}\, t_1) \\ \hat{f}(\mathtt{letrec}\, a_1\, a_2 = t_1\ \mathtt{in}\ t_2) & = & f_{\mathrm{F}}(a_1, a_2, \hat{f}\, t_1, \hat{f}\, t_2) \end{array}}$$

for all $a_1, a_2 \in \mathbb{V}$ and $t_1, t_2 \in \Lambda$.

α-Structural recursion for Λ/α

Given a nominal set X

and functions $\begin{cases} f_{\mathrm{V}} : \mathbb{V} \to X \\ f_{\mathrm{A}} : X \times X \to X \\ f_{\mathrm{L}} : \mathbb{V} \times X \to X \\ f_{\mathrm{F}} : \mathbb{V} \times \mathbb{V} \times X \times X \to X, \end{cases}$

all supported by a finite subset $A \subseteq \mathbb{V}$,

there is a unique function $\hat{f} : \Lambda/\alpha \to X$ such that. . .

α-Structural recursion for Λ/α

... ∃! function $\hat{f} : \Lambda/\alpha \to X$ such that:

$$
\begin{aligned}
\hat{f}\, a_1 &= f_{\mathrm{V}}\, a_1 \\
\hat{f}(e_1\, e_2) &= f_{\mathrm{A}}(\hat{f}\, e_1, \hat{f}\, e_2) \\
a_1 \notin A \Rightarrow \hat{f}(\lambda a_1.e_1) &= f_{\mathrm{L}}(a_1, \hat{f}\, e_1) \\
a_1, a_2 \notin A \;\&\; a_1 \neq a_2 \;\&\; a_2 \notin fv(e_2) \Rightarrow & \\
\hat{f}(\mathtt{letrec}\, a_1\, a_2 = e_1\ \mathtt{in}\ e_2) &= f_{\mathrm{F}}(a_1, a_2, \hat{f}\, e_1, \hat{f}\, e_2)
\end{aligned}
$$

for all $a_1, a_2 \in \mathbb{V}$ & $e_1, e_2 \in \Lambda/\alpha$,

α-Structural recursion for Λ/α

... ∃! function $\hat{f} : \Lambda/\alpha \to X$ such that:

$$
\begin{array}{rcl}
\hat{f}\, a_1 & = & f_{\mathrm{V}}\, a_1 \\
\hat{f}(e_1\, e_2) & = & f_{\mathrm{A}}(\hat{f}\, e_1, \hat{f}\, e_2) \\
a_1 \notin A \;\Rightarrow\; \hat{f}(\lambda a_1.e_1) & = & f_{\mathrm{L}}(a_1, \hat{f}\, e_1) \\
\multicolumn{3}{l}{a_1, a_2 \notin A \;\&\; a_1 \neq a_2 \;\&\; a_2 \notin fv(e_2) \;\Rightarrow} \\
\hat{f}(\mathtt{letrec}\, a_1\, a_2 = e_1\; \mathtt{in}\; e_2) & = & f_{\mathrm{F}}(a_1, a_2, \hat{f}\, e_1, \hat{f}\, e_2)
\end{array}
$$

provided freshness condition for binders (FCB) holds

for f_L: $(\exists a_1 \notin A)(\forall x \in X)\; a_1 \# f_L(a_1, x)$

for f_F: $(\exists a_1, a_2 \notin A)\; a_1 \neq a_2 \;\&$

$(\forall x_1, x_2 \in X)\; a_2 \# x_1 \;\Rightarrow$

$a_1, a_2 \# f_F(a_1, a_2, x_1, x_2)$

α-Structural recursion for Λ/α

The **freshness** relation $(-) \# (-)$ between names and elements of nominal sets generalises the $(-) \notin fv(-)$ relation between variables and ASTs.

E.g. for the capture-avoiding substitution example, $f_L(a_1, e) \triangleq \lambda a_1.e$ and (FCB) holds trivially because $a_1 \notin fv(\lambda a_1.e)$ (and similarly for f_F).

provided freshness condition for binders (FCB) holds

for f_L: $(\exists a_1 \notin A)(\forall x \in X)\ a_1 \# f_L(a_1, x)$

for f_F: $(\exists a_1, a_2 \notin A)\ a_1 \neq a_2\ \&$

$(\forall x_1, x_2 \in X)\ a_2 \# x_1 \Rightarrow$

$a_1, a_2 \# f_F(a_1, a_2, x_1, x_2)$

To be explained:

- Nominal sets, support
  and the freshness relation, $(-)\ \#\ (-)$.
- How is α-structural recursion proved?
- How to generalise α-structural recursion from the example language $\mathbf{\Lambda}$ to general languages with binders?
- What's involved with applying α-structural recursion in any particular case?
- Example: normalisation by evaluation.
- Machine-assisted support?

Atoms

From now on assume bindable names in ASTs are drawn from a fixed, countably infinite set $\mathbb{A}$ (elements called **atoms**)

Need different flavours of names (variables, references, channels, nonces, ...), so assume

- $\mathbb{A}$ is partitioned into countably infinite number of **sorts**.

  Write $sort(a)$ for the sort of $a \in \mathbb{A}$.
- There are infinitely many atoms of each sort.

Atoms

From now on assume bindable names in ASTs are drawn from a fixed, countably infinite set $\mathbb{A}$ (elements called atoms)

The mathematical model of bindable names we use is very abstract: in the world of nominal sets, the only attributes of an atom are identity and sort.

Probably interesting & pragmatically useful to consider more structured atoms (!), e.g. linearly ordered ones, but we don't do that here.

Permutations

Set $\boxed{Perm}$ of atom-permutations consists of all bijections $\pi : \mathbb{A} \leftrightarrow \mathbb{A}$ such that

- $\{a \in \mathbb{A} \mid \pi(a) \neq a\}$ is finite
- $sort(\pi(a)) = sort(a)$ (all $a \in \mathbb{A}$).

$Perm$ is a group:

- multiplication $\pi\,\pi'$ = function composition $\pi \circ \pi'$:
  $\pi \circ \pi'(a) \triangleq \pi(\pi'(a))$
- identity element ι = identity function on $\mathbb{A}$
- inverse π^{-1} of $\pi \in Perm$ is inverse qua function.

Permutations

Set $\boxed{Perm}$ of atom-permutations consists of all bijections $\pi : \mathbb{A} \leftrightarrow \mathbb{A}$ such that

- $\{a \in \mathbb{A} \mid \pi(a) \neq a\}$ is finite
- $sort(\pi(a)) = sort(a)$ (all $a \in \mathbb{A}$).

Given $a_1, a_2 \in \mathbb{A}$ with $sort(a_1) = sort(a_2)$, transposition $\boxed{(a_1\, a_2)}$ is the $\pi \in Perm$ given by

$$\pi(a) \triangleq \begin{cases} a_2 & \text{if } a = a_1 \\ a_1 & \text{if } a = a_2 \\ a & \text{otherwise} \end{cases}$$

Permutations

Set $\boxed{Perm}$ of atom-permutations consists of all bijections $\pi : \mathbb{A} \leftrightarrow \mathbb{A}$ such that

- $\{a \in \mathbb{A} \mid \pi(a) \neq a\}$ is finite
- $sort(\pi(a)) = sort(a)$ (all $a \in \mathbb{A}$).

Given $a_1, a_2 \in \mathbb{A}$ with $sort(a_1) = sort(a_2)$, transposition $\boxed{(a_1\ a_2)}$ is the $\pi \in Perm$ given by

Exercise: prove that every $\pi \in Perm$ can be expressed as a composition of (finitely many) transpositions.

Actions of permutations

An action of $Perm$ on a set S is a function

$$Perm \times S \to S \quad \text{written} \quad (\pi, s) \mapsto \pi \cdot s$$

satisfying $\boxed{\iota \cdot s = s}$ and $\boxed{\pi \cdot (\pi' \cdot s) = (\pi\pi') \cdot s}$

A $Perm$-set is a set S equipped with an action of $Perm$ on S.

Actions of permutations

An action of $Perm$ on a set S is a function

$$Perm \times S \to S \quad \text{written} \quad (\pi, s) \mapsto \pi \cdot s$$

satisfying $\boxed{\iota \cdot s = s}$ and $\boxed{\pi \cdot (\pi' \cdot s) = (\pi\pi') \cdot s}$

Three simple examples of $Perm$-sets:

- Natural numbers $\mathbb{N}$ with trivial action: $\pi \cdot n = n$.
- $\mathbb{A}$ with action: $\pi \cdot a = \pi(a)$.
- $Perm$ itself with conjugation action: $\pi \cdot \pi' = \pi \circ \pi' \circ \pi^{-1}$.

More examples in a mo.

Finite support

<u>Definition</u>. A finite subset $A \subseteq \mathbb{A}$ supports an element $s \in S$ of a $Perm$-set S if

$$(a\, a') \cdot s = s$$

holds for all $a, a' \in \mathbb{A}$ (of same sort) <u>not in A</u>

A nominal set is a $Perm$-set all of whose elements have a finite support.

Lemma. If $s \in S$ has a finite support, then it has a smallest one, written $\boxed{supp(s)}$.

Lemma. If $s \in S$ has a finite support, then it has a smallest one, written $\boxed{supp(s)}$.

Proof Suffices to show that if the finite sets A_1 and A_2 support s, so does $A_1 \cap A_2$.

Lemma. If $s \in S$ has a finite support, then it has a smallest one, written $\boxed{supp(s)}$.

Proof Suffices to show that if the finite sets A_1 and A_2 support s, so does $A_1 \cap A_2$.
So given $a, a' \in \mathbb{A} - (A_1 \cap A_2)$, have to show $(a\,a') \cdot s = s$. W.l.o.g. $a \neq a'$.

Lemma. If $s \in S$ has a finite support, then it has a smallest one, written $\boxed{supp(s)}$.

Proof Suffices to show that if the finite sets A_1 and A_2 support s, so does $A_1 \cap A_2$.
So given $a, a' \in \mathbb{A} - (A_1 \cap A_2)$, have to show $(a\, a') \cdot s = s$. W.l.o.g. $a \neq a'$.

Pick any a'' (of same sort) in the infinite set $\mathbb{A} - (A_1 \cup A_2 \cup \{a, a'\})$. Then

$$(a\, a') = (a\, a'') \circ (a'\, a'') \circ (a\, a'')$$

is a composition of transpositions each of which fixes s. $\square$

Freshness relation

Given nominal sets X and Y and elements $x \in X$ and $y \in Y$,

write $\boxed{x \mathbin{\#} y}$ to mean $supp(x) \cap supp(y) = \emptyset$.

Freshness relation

Given nominal sets X and Y and elements $x \in X$ and $y \in Y$,

write $\boxed{x \# y}$ to mean $supp(x) \cap supp(y) = \emptyset$.

So if $a \in \mathbb{A}$, then $a \# x$ means $a \notin supp(x)$.

Freshness relation

Given nominal sets X and Y and elements $x \in X$ and $y \in Y$,
write $x \mathrel{\#} y$ to mean $supp(x) \cap supp(y) = \emptyset$.

So if $a \in \mathbb{A}$, then $a \mathrel{\#} x$ means $a \notin supp(x)$.

Hence

Key fact for atoms a and a' of the same sort:

$$(a, a') \mathrel{\#} x \;\Rightarrow\; (a\, a') \cdot x = x$$

Languages/α form nominal sets

For example, there's a $Perm$-action on Λ/α satisfying:

$$
\begin{aligned}
\pi \cdot a &= \pi(a)\\
\pi \cdot (e_1\, e_2) &= (\pi \cdot e_1)(\pi \cdot e_2)\\
\pi \cdot (\lambda a.e) &= \lambda \pi(a).(\pi \cdot e)\\
\pi \cdot (\mathtt{letrec}\, a_1\, a_2 = e_1 \ \mathtt{in}\ e_2) &= \\
&\quad \mathtt{letrec}\, \pi(a_1)\, \pi(a_2) = \pi \cdot e_1 \ \mathtt{in}\ \pi \cdot e_2
\end{aligned}
$$

Languages/α form nominal sets

For example, there's a $Perm$-action on Λ/α satisfying:

$$\pi \cdot a = \pi(a)$$
$$\pi \cdot (e_1\, e_2) = (\pi \cdot e_1)(\pi \cdot e_2)$$
$$\pi \cdot (\lambda a.e) = \lambda \pi(a).(\pi \cdot e)$$
$$\pi \cdot (\texttt{letrec}\, a_1\, a_2 = e_1 \texttt{ in } e_2) = \texttt{letrec}\, \pi(a_1)\, \pi(a_2) = \pi \cdot e_1 \texttt{ in } \pi \cdot e_2$$

N.B. binding and non-binding constructs are treated just the same

Languages/α form nominal sets

For example, there's a Perm-action on Λ/α satisfying:

$$
\begin{aligned}
\pi \cdot a &= \pi(a) \\
\pi \cdot (e_1\, e_2) &= (\pi \cdot e_1)(\pi \cdot e_2) \\
\pi \cdot (\lambda a.e) &= \lambda \pi(a).(\pi \cdot e) \\
\pi \cdot (\mathtt{letrec}\, a_1\, a_2 = e_1 \;\mathtt{in}\; e_2) &= \\
&\qquad \mathtt{letrec}\, \pi(a_1)\, \pi(a_2) = \pi \cdot e_1 \;\mathtt{in}\; \pi \cdot e_2
\end{aligned}
$$

Proof (exercise) First define $\pi \cdot (-) : \Lambda \longrightarrow \Lambda$ by structural recursion, and then prove that $t =_\alpha t' \Rightarrow (\forall \pi \in \mathit{Perm})\ \pi \cdot t =_\alpha \pi \cdot t'$.

Languages/α form nominal sets

For example, there's a $Perm$-action on Λ/α satisfying:

$$\begin{aligned}
\pi \cdot a &= \pi(a) \\
\pi \cdot (e_1\, e_2) &= (\pi \cdot e_1)(\pi \cdot e_2) \\
\pi \cdot (\lambda a.e) &= \lambda \pi(a).(\pi \cdot e) \\
\pi \cdot (\mathtt{letrec}\, a_1\, a_2 = e_1 \;\mathtt{in}\; e_2) &= \\
&\mathtt{letrec}\, \pi(a_1)\, \pi(a_2) = \pi \cdot e_1 \;\mathtt{in}\; \pi \cdot e_2
\end{aligned}$$

For this action, it is not hard to see (exercise) that $e \in \Lambda/\alpha$ is supported by any finite set of variables containing all those occurring free in e and hence

$$a \# e \text{ iff } a \notin fv(e).$$

End of lecture 1