Recap from Last Time

- we discussed a general theory of nominal sets using certain properties
- Permutations on atoms need to satisfy:
  - $[]\bullet a = a$
  - $((a\ b) :: \pi)\bullet c = swap\ (a\ b)\ (\pi\bullet c)$
  - $swap\ (a\ b)\ c =$
    $\texttt{if}\ a = c\ \texttt{then}\ b\ \texttt{else}$
    $\texttt{if}\ b = c\ \texttt{then}\ a\ \texttt{else}\ c$
  - every set of all atoms of type α is infinite

General Permutations

Permutations have to satisfy:

- $[] \bullet x = x$
- $(\pi_1 @ \pi_2) \bullet x = \pi_1 \bullet (\pi_2 \bullet x)$
- $\pi_1 \sim \pi_2$ implies $\pi_1 \bullet x = \pi_2 \bullet x$

Two important definitions:

- $$supp_\alpha\ x \stackrel{\text{def}}{=} \{a : \alpha \mid \text{infinite}\{b \mid (a\ b) \bullet x \neq x\}\} \qquad x : \iota$$
- $$a \# x \stackrel{\text{def}}{=} a \notin supp_\alpha(x) \qquad a : \alpha, x : \iota$$

Abstractions

- Nominal abstractions need to satisfy $abst_{\alpha,\iota}$.
  - $\pi \bullet ([a].x) = [\pi \bullet a].(\pi \bullet x)$
  - $[a].x = [b].y$ iff

    $(a = b \wedge x = y) \vee$
    $(a \neq b \wedge x = (a\,b) \bullet y \wedge a \# y)$
- Are there any structures that satisfy these properties?

Yes...

- α-equivalence classes (sets of syntax trees)

$$[\lambda a.(a\ c)]_\alpha = [\lambda b.(b\ c)]_\alpha$$

- terms with de-Bruijn indices and named free variables

$$\lambda(1\ c)$$

  (you need a function abs which "abstracts" a variable: $abs(x, t) \mapsto \lambda(\ldots)$)

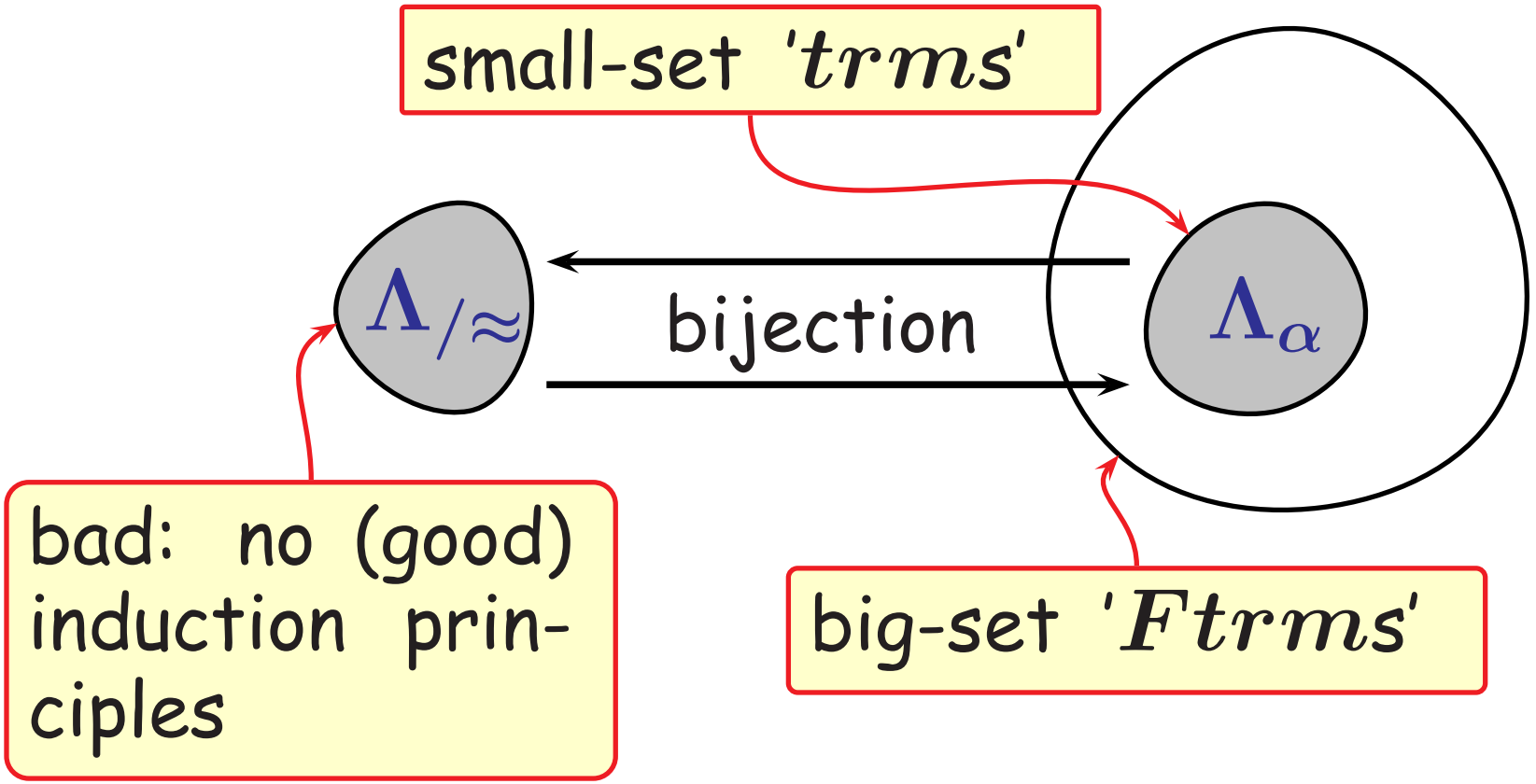
- a weak HOAS encoding (lambdas as functions — the function for $\lambda a.(a\ c)$ will be the same as the one for $\lambda b.(b\ c)$)

Weak HOAS Encoding

I define **inductively** the α-equivalence classes of lambda-terms—but they still have **names**.

Weak HOAS Encoding

I define **inductively** the α-equivalence classes of lambda-terms—but they still have **names**.



Weak HOAS Encoding

I define **inductively** the α-equivalence classes of lambda-terms—but they still have **names**.

The small set Λ_α will consist of $trms$ having the form

$$\begin{array}{rcl} t & ::= & \mathsf{Var}(a)\ ^* \\ & | & \mathsf{App}\ t\ t \\ & | & [a].t \end{array}$$

b
in
ciples

$^*atoms$ come from an infinite set, e.g. the natural numbers

Big-Set / $Ftrms$

One step back: naïve attempt for big-set

$$
\begin{array}{lll}
Ftrm ::= & & \\
\quad \mathsf{Var} & : Atom & \text{'atoms'} \\
\quad | \quad \mathsf{App} & : Ftrm \times Ftrm & \text{'pairs'} \\
\quad | \quad \mathsf{Se} & : Ftrm\ Set & \text{'}\alpha\text{-eq-cl'}
\end{array}
$$

Big-Set / *Ftrms*

One step back: naïve attempt for big-set

$$
\begin{array}{lll}
Ftrm ::= & & \\
\quad \text{Var} & : Atom & \text{'atoms'} \\
\quad | \quad \text{App} & : Ftrm \times Ftrm & \text{'pairs'} \\
\quad | \quad \text{Se} & : Ftrm\ Set & \text{'}\alpha\text{-eq-cl'}
\end{array}
$$

we try to encode the α-equivalence class as the set of lambda-terms

$$[t]_\alpha \stackrel{\text{def}}{=} \{t' \mid t \approx t'\}$$

Big-Set / $Ftrms$

Better attempt for big-set

$$
\begin{array}{lll}
Ftrm ::= & \text{er} & \text{'error'} \\
| & \text{Var} : Atom & \text{'atoms'} \\
| & \text{App} : Ftrm \times Ftrm & \text{'pairs'} \\
| & \text{Se} : Atom \longrightarrow Ftrm & \text{'}\alpha\text{-eq-cl'}
\end{array}
$$

same idea, but encoding with (partial) functions, along the lines:

"if $t' \in [\lambda a.t]_\alpha$ then 'yes' else er"

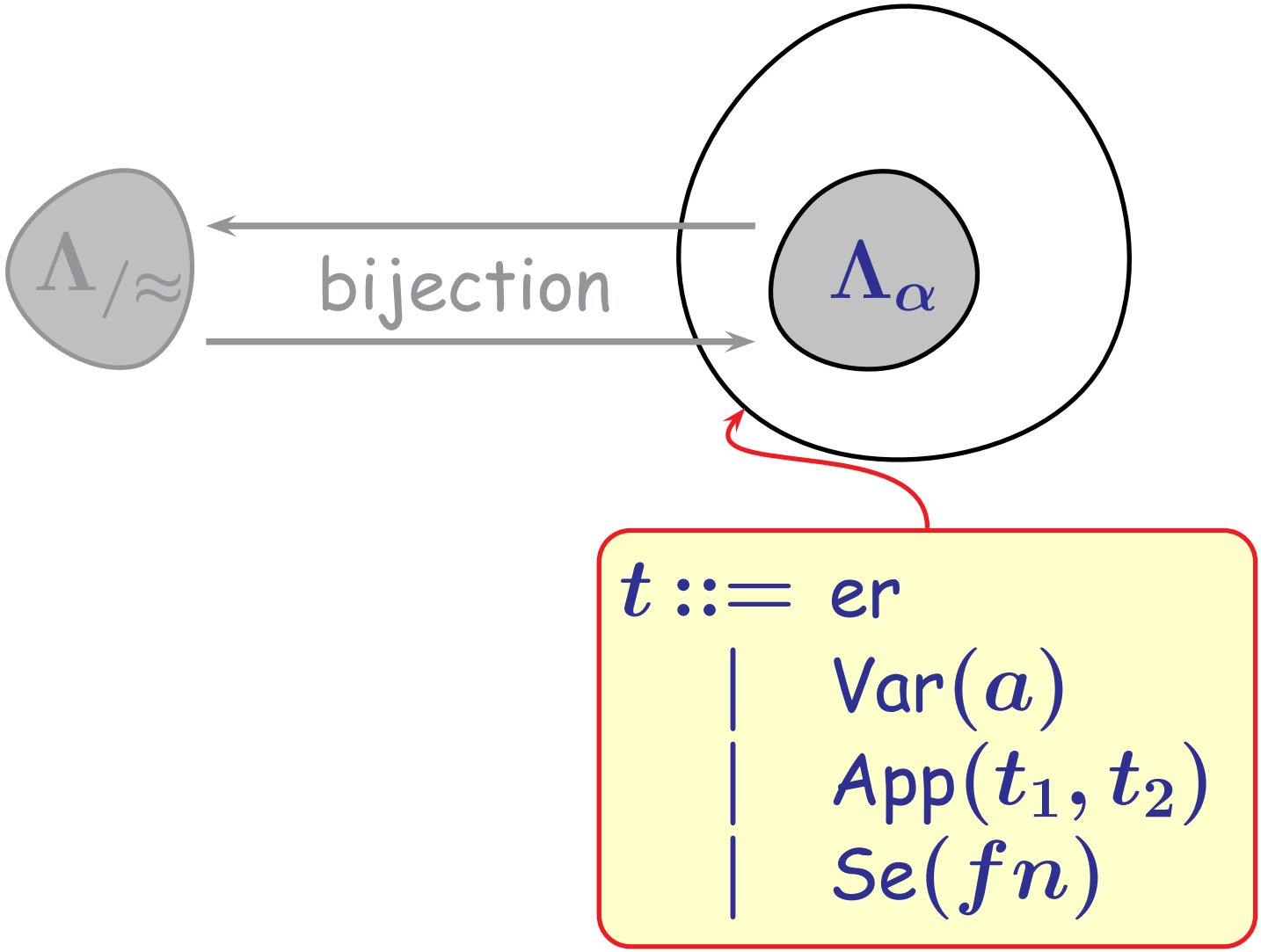
Permutations for Big-Set

Given the permutation action for atoms, we can permute all "**free**" atoms in $Ftrms$:

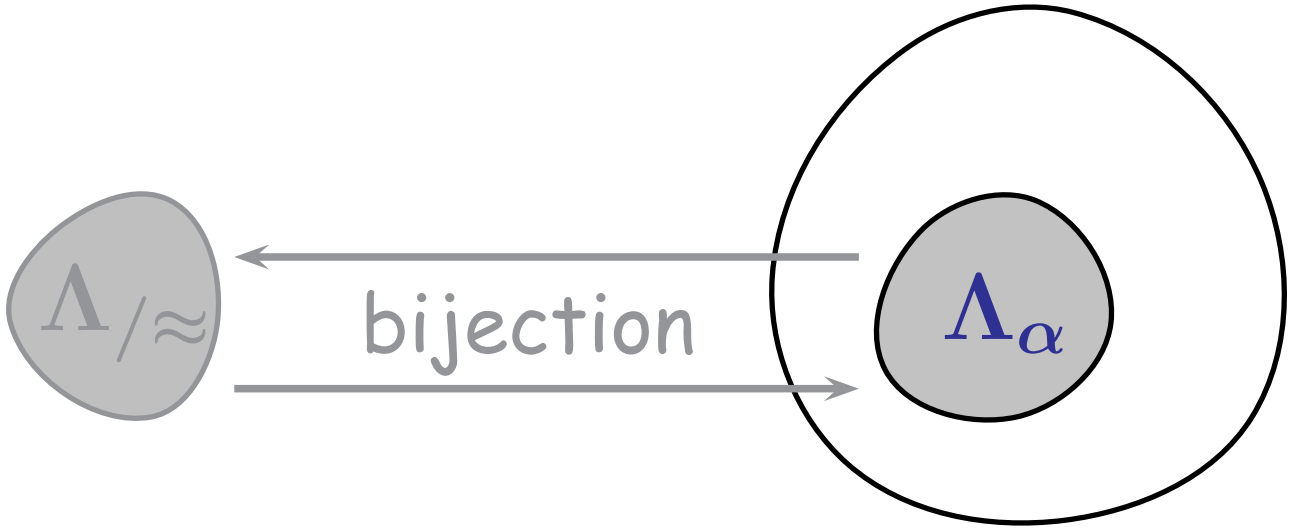
$$
\begin{array}{lcl}
\pi \bullet \mathsf{er} & \stackrel{\text{def}}{=} & \mathsf{er} \\
\pi \bullet \mathsf{Var}(a) & \stackrel{\text{def}}{=} & \mathsf{Var}(\pi \bullet a) \\
\pi \bullet \mathsf{App}(t_1, t_2) & \stackrel{\text{def}}{=} & \mathsf{App}(\pi \bullet t_1, \pi \bullet t_2) \\
\pi \bullet \mathsf{Se}(fn) & \stackrel{\text{def}}{=} & \mathsf{Se}(\lambda a.\pi \bullet (fn(\pi^{-1} \bullet a)))
\end{array}
$$

The $pt_{\alpha,ftrm}$-properties are satisfied.

What About Small-Set?



What About Small-Set?



For Λ_α, we are only interested in some very specific functions, namely

$$[a].t \stackrel{\text{def}}{=} \text{Se}\ (\lambda b.\ \text{if}\ a = b\ \text{then}\ t\ \text{else if}\ b \# t\ \text{then}\ (b\,a) \bullet t\ \text{else er})$$

Function $[a].t$ '=' $[\lambda a.t]_\alpha$

$$[a].t \stackrel{\text{def}}{=} \text{Se}\ (\lambda b.\ \text{if}\ a = b \ \text{then}\ t \ \text{else if}\ b \# t\ \text{then}\ (b\,a) \bullet t\ \text{else er})$$

Function $[a].t$ '=' $[\lambda a.t]_\alpha$

$$[a].t \stackrel{\text{def}}{=} \text{Se}\ (\lambda b.\ \text{if}\ a = b \ \text{then}\ t \ \text{else if}\ b \# t \ \text{then}\ (b\,a) \bullet t \ \text{else er})$$

This is supposed to stand for the α-equivalence class of $\lambda a.t$.

Function $[a].t$ '=' $[\lambda a.t]_\alpha$

$$[a].\mathsf{App}(a,c) \stackrel{\text{def}}{=} \mathsf{Se}\ (\lambda b.\ \text{if } a = b \text{ then } \mathsf{App}(a,c) \text{ else if } b \# \mathsf{App}(a,c) \text{ then } (b\,a) \bullet \mathsf{App}(a,c) \text{ else er})$$

Let's check this for $[a].\mathsf{App}(a,c)$:

Function $[a].t$ '=' $[\lambda a.t]_\alpha$

$$[a].\mathsf{App}(a,c) \stackrel{\text{def}}{=}$$
$$\mathsf{Se}\ (\lambda b.\ \text{if}\ a = b$$
$$\text{then}\ \mathsf{App}(a,c)$$
$$\text{else if}\ b \# \mathsf{App}(a,c)$$
$$\text{then}\ (b\,a) \bullet \mathsf{App}(a,c)\ \text{else er})$$

Let's check this for $[a].\mathsf{App}(a,c)$:

$[a].\mathsf{App}(a,c)$ 'applied to' a 'gives' $\mathsf{App}(a,c)$

Function $[a].t$ '=' $[\lambda a.t]_\alpha$

$$[a].\mathsf{App}(a,c) \stackrel{\text{def}}{=}$$
$$\text{Se } (\lambda b.\ \text{if } a = b \text{ then } \mathsf{App}(a,c) \text{ else if } b \# \mathsf{App}(a,c) \text{ then } (b\,a) \bullet \mathsf{App}(a,c) \text{ else er})$$

Let's check this for $[a].\mathsf{App}(a,c)$:

$[a].\mathsf{App}(a,c)$ 'applied to' a 'gives' $\mathsf{App}(a,c)$

$[a].\mathsf{App}(a,c)$ 'applied to' b 'gives' $\mathsf{App}(b,c)$

Function $[a].t$ '=' $[\lambda a.t]_\alpha$

$$[a].\mathsf{App}(a,c) \stackrel{\text{def}}{=}$$
$$\mathsf{Se}\ (\lambda b.\ \text{if}\ a = b$$
$$\text{then}\ \mathsf{App}(a,c)$$
$$\text{else if}\ b \# \mathsf{App}(a,c)$$
$$\text{then}\ (b\,a) \bullet \mathsf{App}(a,c)\ \text{else er})$$

Let's check this for $[a].\mathsf{App}(a,c)$:

$[a].\mathsf{App}(a,c)$ 'applied to' a 'gives' $\mathsf{App}(a,c)$

$[a].\mathsf{App}(a,c)$ 'applied to' b 'gives' $\mathsf{App}(b,c)$

$[a].\mathsf{App}(a,c)$ 'applied to' c 'gives' er

Function $[a].t$ '=' $[\lambda a.t]_\alpha$

$$[a].\mathsf{App}(a,c) \stackrel{\text{def}}{=}$$
$$\text{Se } (\lambda b.\ \text{if } a = b \text{ then } \mathsf{App}(a,c) \text{ else if } b \# \mathsf{App}(a,c) \text{ then } (b\,a) \bullet \mathsf{App}(a,c) \text{ else er})$$

Let's check this for $[a].\mathsf{App}(a,c)$:

$[a].\mathsf{App}(a,c)$ 'applied to' a 'gives' $\mathsf{App}(a,c)$

$[a].\mathsf{App}(a,c)$ 'applied to' b 'gives' $\mathsf{App}(b,c)$

$[a].\mathsf{App}(a,c)$ 'applied to' c 'gives' er

$[a].\mathsf{App}(a,c)$ 'applied to' d 'gives' $\mathsf{App}(d,c)$

$\vdots$

Function $[a].t$ '=' $[\lambda a.t]_\alpha$

$$[a].\mathsf{App}(a,c) \stackrel{\text{def}}{=}$$
$$\text{Se } (\lambda b.\ \text{if } a = b \text{ then } \mathsf{App}(a,c) \text{ else if } b \# \mathsf{App}(a,c) \text{ then } (b\,a) \bullet \mathsf{App}(a,c) \text{ else er})$$

Let's check this for $[a].\mathsf{App}(a,c)$:

$[a].\mathsf{App}(a,c)$ 'applied to' a 'gives' $\mathsf{App}(a,c)$ '$\lambda a.(a\,c)$'

$[a].\mathsf{App}(a,c)$ 'applied to' b 'gives' $\mathsf{App}(b,c)$ '$\lambda b.(b\,c)$'

$[a].\mathsf{App}(a,c)$ 'applied to' c 'gives' er

$[a].\mathsf{App}(a,c)$ 'applied to' d 'gives' $\mathsf{App}(d,c)$ '$\lambda d.(d\,c)$'

$\vdots$

Function $[a].t$ '=' $[\lambda a.t]_\alpha$

$$[a].\mathsf{App}(a,c) \stackrel{\text{def}}{=}$$
$$\text{Se } (\lambda b.\ \text{if } a = b \text{ then } \mathsf{App}(a,c) \text{ else if } b \# \mathsf{App}(a,c) \text{ then } (b\,a) \bullet \mathsf{App}(a,c) \text{ else er})$$

Let's check this for $[a].\mathsf{App}(a,c)$:

$[a].\mathsf{App}(a,c)$ 'applied to' a 'gives' $\mathsf{App}(a,c)$

$[a].\mathsf{App}(a,c)$ 'applied to' b 'gives' $\mathsf{App}(b,c)$

$[a].\mathsf{App}(a,c)$ 'applied to' c 'gives' er

$[a].\mathsf{App}(a,c)$ 'applied to' d 'gives' $\mathsf{App}(d,c)$

$\vdots$

$[\lambda a.(a\,c)]_\alpha$:

$'\lambda a.(a\,c)'$

$'\lambda b.(b\,c)'$

$'\lambda d.(d\,c)'$

$\vdots$

Properties of $[a].t$

Nominal Abstractions need to satisfy $abst_{\alpha,\iota}$:

- $\pi \bullet ([a].t) = [\pi \bullet a].(\pi \bullet t)$
- $[a].x = [b].y$ iff

  $(a = b \wedge x = y) \vee$
  $(a \neq b \wedge x = (a\,b) \bullet y \wedge a \# y)$

The last axiom says that $[a].t$ behaves like an abstraction.

Properties of $[a].t$

Nominal Abstractions need to satisfy $abst_{\alpha,\iota}$:

- $\pi \bullet ([a].t) = [\pi \bullet a].(\pi \bullet t)$
- $[a].x = [b].y$ iff

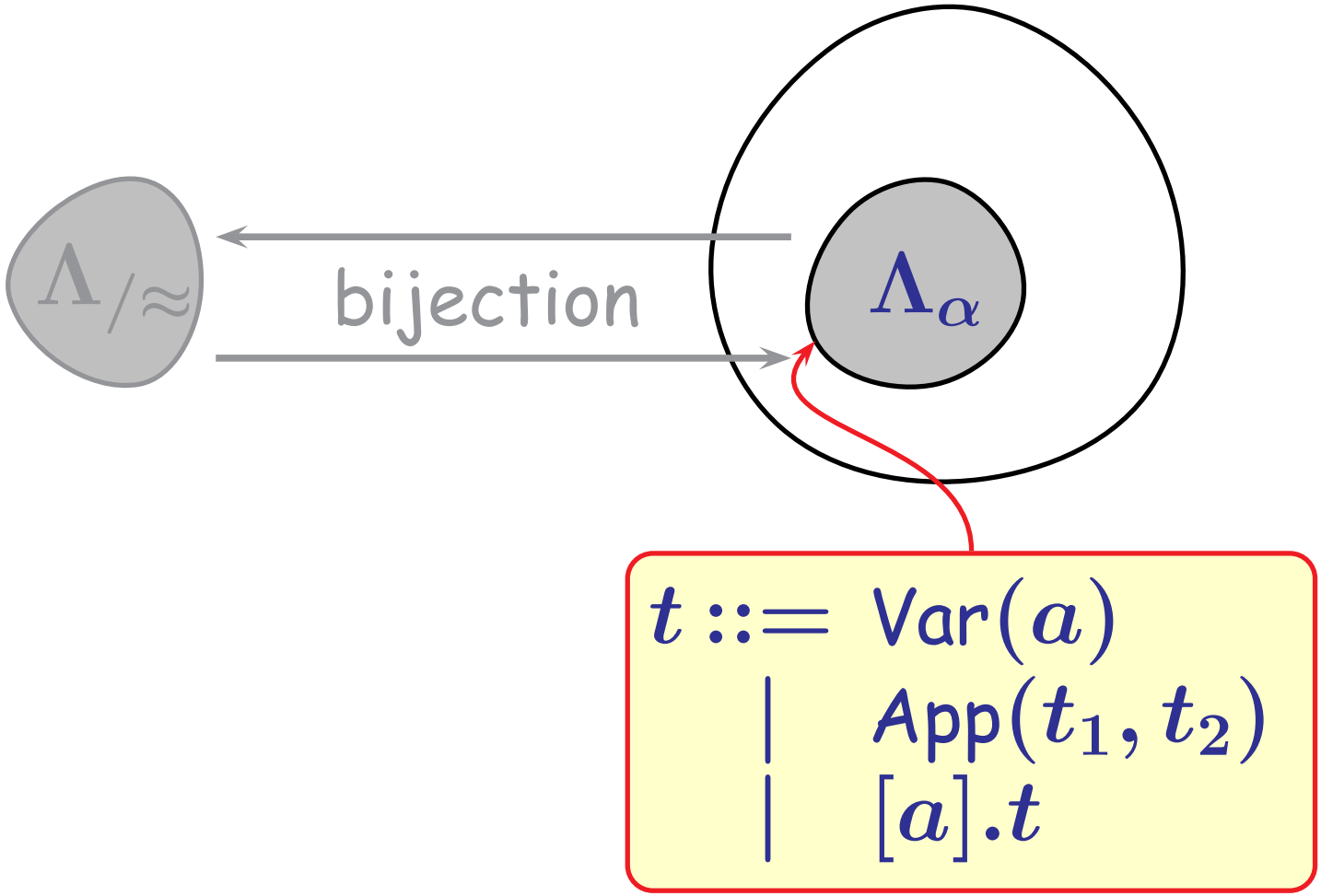
$$(a = b \wedge x = y) \vee$$
$$(a \neq b \wedge x = (a\,b) \bullet y \wedge a \# y)$$

The last axiom says that $[a].t$ behaves like an abstraction.

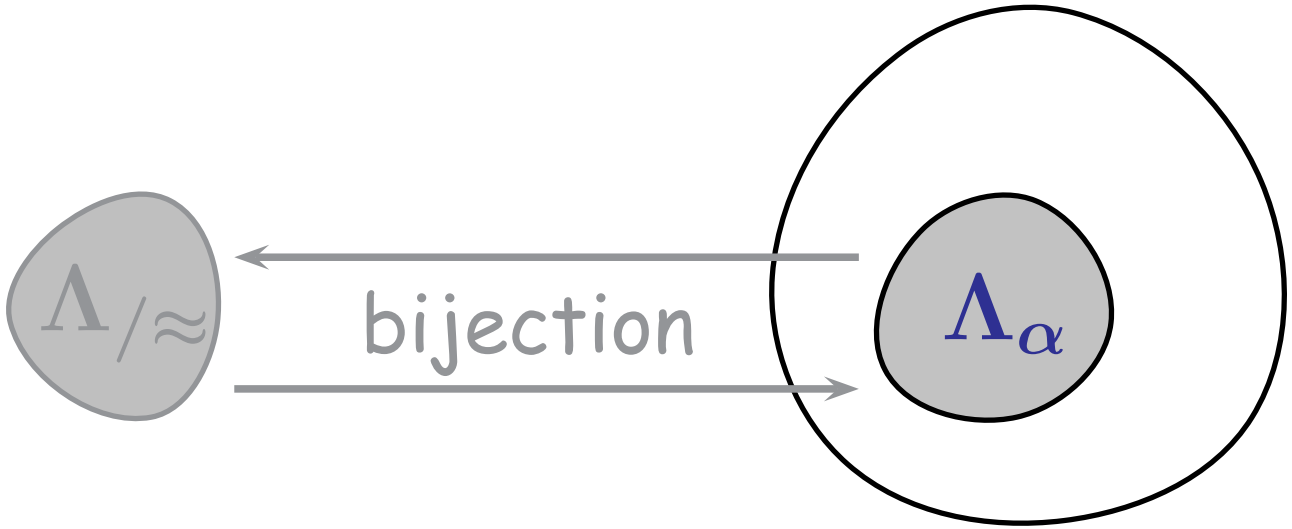
Remember the definition of α-equivalence from the detour:

$$\frac{t_1 \approx t_2}{\lambda a.t_1 \approx \lambda a.t_2} \qquad \frac{t_1 \approx (a\,b) \bullet t_2 \quad a \notin FV(t_2)}{\lambda a.t_1 \approx \lambda b.t_2}$$

Definition of Small-Set



Definition of Small-Set

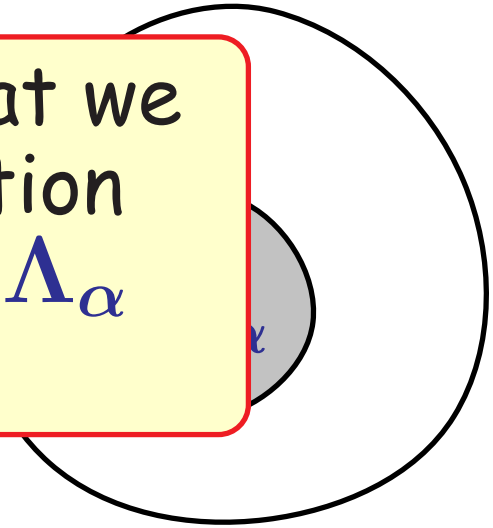


$$\frac{}{\mathsf{Var}(a) \in \Lambda_\alpha} \qquad \frac{t_1 \in \Lambda_\alpha \quad t_2 \in \Lambda_\alpha}{\mathsf{App}(t_1, t_2) \in \Lambda_\alpha}$$

$$\frac{t \in \Lambda_\alpha}{[a].t \in \Lambda_\alpha}$$

Definition of Small-Set

Which also means that we have a familiar induction principle in place for Λ_α (in a moment).



$$\frac{}{\mathsf{Var}(a) \in \Lambda_\alpha} \qquad \frac{t_1 \in \Lambda_\alpha \quad t_2 \in \Lambda_\alpha}{\mathsf{App}(t_1, t_2) \in \Lambda_\alpha}$$

$$\frac{t \in \Lambda_\alpha}{[a].t \in \Lambda_\alpha}$$

Bijection

In order to show that $\Lambda_{/\approx}$ and Λ_α are bijective we define a function q from Λ to Λ_α:

$$\begin{array}{lcl} q(a) & \stackrel{\text{def}}{=} & \mathsf{Var}(a) \\ q(t_1\, t_2) & \stackrel{\text{def}}{=} & \mathsf{App}(q(t_1), q(t_2)) \\ q(\lambda a.t) & \stackrel{\text{def}}{=} & [a].q(t) \end{array}$$

with the property

$$t_1 \approx t_2 \;\Leftrightarrow\; q(t_1) = q(t_2)$$

Struct. Induction on Λ_α

$$\frac{}{\mathsf{Var}(a) \in \Lambda_\alpha} \qquad \frac{t_1 \in \Lambda_\alpha \quad t_2 \in \Lambda_\alpha}{\mathsf{App}(t_1, t_2) \in \Lambda_\alpha}$$

$$\frac{t \in \Lambda_\alpha}{[a].t \in \Lambda_\alpha}$$

Structural Induction Principle:

$$\frac{(\forall a)\ P\,(\mathsf{Var}(a)) \qquad (\forall t_1, t_2)\ P\,t_1 \wedge P\,t_2 \Rightarrow P\,(\mathsf{App}(t_1, t_2)) \qquad (\forall a, t)\ P\,t \Rightarrow P\,([a].t)}{(\forall t)\ P\,t}$$

Substitution Lemma: If $x \not\equiv y$ and $x \notin FV(L)$, then

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]].$$

Proof: By induction on the structure of M.

- **Case 1:** M is a variable.

  Case 1.1. $M \equiv x$. Then both sides equal $N[y := L]$ since $x \not\equiv y$.

  Case 1.2. $M \equiv y$. Then both sides equal L, for $x \notin FV(L)$ implies $L[x := \ldots] \equiv L$.

  Case 1.3. $M \equiv z \not\equiv x, y$. Then both sides equal z.

- **Case 2:** $M \equiv \lambda z.M_1$. By the variable convention we may assume that $z \not\equiv x, y$ and z is not free in N, L. Then by induction hypothesis

$$\begin{aligned}
&(\lambda z.M_1)[x := N][y := L] \equiv \lambda z.(M_1[x := N][y := L]) \\
&\equiv \lambda z.(M_1[y := L][x := N[y := L]]) \\
&\equiv (\lambda z.M_1)[y := L][x := N[y := L]].
\end{aligned}$$

- **Case 3:** $M \equiv M_1 M_2$. The statement follows again from the induction hypothesis. "□"

Can We Do Better?

We consider induction hypotheses of the form $P\ t\ x$ with x finitely supported. In the case of the substitution lemma:

$$P = \lambda M.\ \lambda(x, y, N, L).\ M[x := N][y := L] \equiv M[y := L][x := N[y := L]]$$

$$x = (x, y, N, L)$$

$$\frac{(\forall a)\ P\ (\mathsf{Var}(a))\ x \qquad (\forall t_1, t_2)\ P\ t_1\ x \wedge P\ t_2\ x \Rightarrow P\ (\mathsf{App}(t_1, t_2))\ x \qquad (\forall a)\ a \# x \Rightarrow ((\forall t)\ P\ t\ x \Rightarrow P\ ([a].t)\ x)}{(\forall t)\ P\ t\ x}$$

Proof

By an induction over Λ_α. We need to strengthen the induction hypothesis to

$$(\forall t, \pi)\ P\,(\pi \bullet t)\,x$$

In the lambda-case we need to show:

$$P\,(\pi \bullet [a].t)\,x$$

We have:

$$(\forall t, \pi)\ P\,(\pi \bullet t)\,x$$

$$(\forall a)\ a \# x \Rightarrow ((\forall t)\ P\,t\,x \Rightarrow P\,([a].t)\,x)$$

```
lemma subs_lemma: assumes a1: "x ≠ y" and a2: "x#L"
shows "M[x := N][y := L] = M[y := L][x := N[y := L]]"
proof (nominal_induct M)
  case (Var z) (* case 1: variables *)
  have "z = x ∨ (z ≠ x ∧ z = y) ∨ (z ≠ x ∧ z ≠ y)" by force
  thus "(Var z)[x:=N][y:=L] = (Var z)[y:=L][x:=N[y:=L]]"
    using a1 a2 forget by force
next
  case ([z].M1) (* case 2: lambdas *)
  assume ih: "M1[x:=N][y:=L] = M1[y:=L][x:=N[y:=L]]"
  assume f1: "z#(L, N, x, y)"
  from f1 fresh_fact1 have f2: "z#N[y := L]" by simp
  show "([z].M1)[x:=N][y:=L]=([z].M1)[y:=L][x:=N[y:=L]]"
(is "?LS =?RS")
  proof -
    have "?LS = [z].(M1[x := N][y := L])" using f1 by simp
    also have ". . . = [z].(M1[y:=L][x:=N[y:=L]])" using ih by simp
    also have ". . .=  ?RS" using f1 f2 by simp
    finally show "?LS = ?RS" by simp
  qed
next
  case (App M1 M2) (* case 3: applications *)
  thus "?case" by simp
qed
```