Quiz?

Assuming that a and b are distinct variables, is it possible to find λ-terms M_1 to M_7 that make the following pairs α-equivalent?

- $\lambda a.\lambda b.(M_1\ b)$ and $\lambda b.\lambda a.(a\ M_1)$
- $\lambda a.\lambda b.(M_2\ b)$ and $\lambda b.\lambda a.(a\ M_3)$
- $\lambda a.\lambda b.(b\ M_4)$ and $\lambda b.\lambda a.(a\ M_5)$
- $\lambda a.\lambda b.(b\ M_6)$ and $\lambda a.\lambda a.(a\ M_7)$

If there is one solution for a pair, can you describe all its solutions?

Nominal Techniques in Isabelle/HOL (II):

Alpha-Equivalence Classes

based on work by Andy Pitts

joint work with Stefan, Markus, Alexander...

Recap (I): α-Equivalence

The following rules define α-equivalence on lambda-term (syntax-trees):

$$\frac{}{a \approx a}\ \approx\text{-atm} \qquad \frac{t_1 \approx s_1 \quad t_2 \approx s_2}{t_1\, t_2 \approx s_1\, s_2}\ \approx\text{-app}$$

$$\frac{t \approx s}{\lambda a.t \approx \lambda a.s}\ \approx\text{-lam}_1 \qquad \frac{t \approx (a\, b)\bullet s \quad a \notin \mathsf{fv}(s)}{\lambda a.t \approx \lambda b.s}\ \approx\text{-lam}_2$$

assuming $a \neq b$

Recap (II): Support and Freshness

The **support** of an object $x : \iota$ is a set of atoms α:

$$\mathsf{supp}_\alpha\ x \stackrel{\mathsf{def}}{=} \{a \mid \mathsf{infinite}\{b \mid (a\ b)\bullet x \neq x\}$$

An atom is **fresh** for an x, if it is not in the support of x:

$$a \# x \stackrel{\mathsf{def}}{=} a \notin \mathsf{supp}_\alpha(x)$$

I often drop the α in supp_α.

Nominal Abstractions

We are now going to specify what abstraction 'abstractly' means: it is an operation $[\_].(\_) : \alpha \Rightarrow \iota \Rightarrow \iota$ which has to satisfy:

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

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

- these two properties imply for finitely supported x

  $$\mathsf{supp}([a].x) = \mathsf{supp}(x) - \{a$$

Nominal Abstractions

Remember the definition of α-equivalence from the beginning:

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

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

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

- these two properties imply for finitely supported x

$$\mathsf{supp}([a].x) = \mathsf{supp}(x) - \{a$$

Nominal Abstractions

We are now going to specify what abstraction 'abstractly' means: it is an operation $[\_].(\_) : \alpha \Rightarrow \iota \Rightarrow \iota$ which has to satisfy:

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

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

- these two properties imply for finitely supported x

  $$\mathsf{supp}([a].x) = \mathsf{supp}(x) - \{a$$

Freshness and Abstractions

Given $pt_{\alpha,\iota}$, $\text{finite}(\text{supp}\ x)$ and $a \neq b$ then

$$a \# x \text{ iff } a \# [b].x$$

Proof. There exists a c with $c \# (a, b, x, [b].x)$.

($\Leftarrow$) From $a \# [b].x$ and $c \# [b].x$

$$[b].x = (a\,c)\bullet([b].x) = [b].(a\,c)\bullet x$$

Hence $x = (a\,c)\bullet x$. Now from $c \# x$:

$$c \# x \Leftrightarrow (a\,c)\bullet c \# (a\,c)\bullet x \Leftrightarrow a \# x$$

Freshness and Abstractions

Given $pt_{\alpha,\iota}$, finite(supp x) and $a \neq b$ then

$$a \# x \text{ iff } a \# [b].x$$

Proof. There exists a c with $c \# (a, b, x, [b].x)$.

($\Rightarrow$) From $c \# [b].x$ we also have

$$(a\,c)\bullet c \# (a\,c)\bullet[b].x$$

and

$$a \# [b].(a\,c)\bullet x$$

Because $a \# x$ and $c \# x$, $(a\,c)\bullet x = x$.

Freshness and Abstractions

We also have

$$a \;\#\; [a].x$$

Again from $c \;\#\; (a, x, [a].x)$ we can infer

$$\begin{aligned} c \;\#\; [a].x \quad &\Leftrightarrow \quad (a\,c)\bullet c \;\#\; (a\,c)\bullet[a].x \\ &\Leftrightarrow \quad a \;\#\; [c].(a\,c)\bullet x. \end{aligned}$$

However:

$$[c].(a\,c)\bullet x = [a].x$$

(since $c \neq a$, $[c].(a\,c)\bullet x = [a].x$
iff $(a\,c)\bullet x = (a\,c)\bullet x \wedge c \;\#\; x$)

Freshness and Abstractions

So we have shown that

$$\frac{a \neq b \quad a \# x}{a \# [b].x} \qquad \frac{}{a \# [a].x}$$

and

$$a \# x \stackrel{\text{def}}{=} a \notin \mathsf{supp}(x)$$

therefore

$$\mathsf{supp}([a].x) = \mathsf{supp}(x) - \{a$$

We als

Again f

$c \#$ … x

Howeve

$[c].(a\,c)\bullet x = [a].x$

(since $c \neq a$, $[c].(a\,c)\bullet x = [a].x$
iff $(a\,c)\bullet x = (a\,c)\bullet x \wedge c \# x$)

Nominal Abstractions

We have specified what abstraction 'abstractly' means by an operation $[\_].(\_) : \alpha \Rightarrow \iota \Rightarrow \iota$ which satisfies:

- $\pi \bullet ([a].x) = [\pi \bullet a].(\pi \bullet x)$
- $[a].x = [b].y$ 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? Are there any structures that are "supported" in Isabelle/HOL?

Possibilities

- α-equivalence classes (sets of syntax trees), e.g.$[\lambda a.(a\ c)]_\alpha = [\lambda b.(b\ c)]_\alpha$
- terms with de-Bruijn indices and named free variables, like $\lambda(1\ c)$.
  (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 $\lambda b.(b\ c)$)

Remember the user will only see the "axioms" from the previous slide.

Possibilities

- α-equivalence classes (sets of syntax trees), e.g.$[\lambda a.(a\ c)]_\alpha = [\lambda b.(b\ c)]_\alpha$
- terms with de-Bruijn indices and named free variables, like $\lambda(1\ c)$

> I could now stop here (this is all known), and probably go for α-equivalence classes (Norrish did this with the help of a package by Hohmeier for HOL4), **but** I do not ;o)

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

Remember the user will only see the "axioms" from the previous slide.

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

$$
\begin{aligned}
[a].t \stackrel{\text{def}}{=} (\lambda b.\, & \text{if } a = b \\
& \text{then Some}(t) \\
& \text{else if } b \# t \text{ then Some}((b\,a)\bullet t) \text{ else None})
\end{aligned}
$$

type: $\alpha \rightarrow \iota$ option

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

$$
\begin{aligned}
[a].t \stackrel{\mathsf{def}}{=} (\lambda b.\ & \mathsf{if}\ a = b \\
& \mathsf{then\ Some}(t) \\
& \mathsf{else\ if}\ b \# t\ \mathsf{then\ Some}((b\,a)\bullet t)\ \mathsf{else\ None})
\end{aligned}
$$

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

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

$$[a].(a,c) \stackrel{\text{def}}{=}$$
$$\begin{aligned}&(\lambda b.\text{if } a = b\\ &\qquad \text{then Some}(a,c)\\ &\qquad \text{else if } b \# (a,c)\\ &\qquad\qquad \text{then Some}((b\,a)\bullet(a,c)) \text{ else None})\end{aligned}$$

Let's check this for $[a].(a,c)$:

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

$$[a].(a, c) \stackrel{\text{def}}{=}$$
$$(\lambda b.\text{if } a = b$$
$$\text{then Some}(a, c)$$
$$\text{else if } b \# (a, c)$$
$$\text{then Some}((b\,a) \bullet (a, c)) \text{ else None})$$

Let's check this for $[a].(a, c)$:

a 'applied to' $[a].(a, c)$ 'gives' Some(a, c)

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

$$
\begin{aligned}
&[a].(a,c) \stackrel{\text{def}}{=} \\
&\qquad (\lambda b.\text{if } a = b \\
&\qquad\qquad \text{then Some}(a,c) \\
&\qquad\qquad \text{else if } b \# (a,c) \\
&\qquad\qquad\qquad \text{then Some}((b\,a)\bullet(a,c)) \text{ else None})
\end{aligned}
$$

Let's check this for $[a].(a,c)$:

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

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

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

$$[a].(a,c) \stackrel{\text{def}}{=}$$
$$(\lambda b.\text{if } a = b$$
$$\text{then Some}(a,c)$$
$$\text{else if } b \# (a,c)$$
$$\text{then Some}((b\,a) \bullet (a,c)) \text{ else None})$$

Let's check this for $[a].(a,c)$:

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

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

c 'applied to' $[a].(a,c)$ 'gives' None

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

$$[a].(a,c) \stackrel{\text{def}}{=}$$
$$(\lambda b.\text{if } a = b$$
$$\text{then Some}(a,c)$$
$$\text{else if } b \# (a,c)$$
$$\text{then Some}((b\,a)\bullet(a,c)) \text{ else None})$$

Let's check this for $[a].(a,c)$:

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

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

c 'applied to' $[a].(a,c)$ 'gives' None

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

$\vdots$

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

$$
\begin{aligned}
[a].(a,c) &\stackrel{\text{def}}{=}\\
&(\lambda b.\text{if } a = b\\
&\qquad \text{then Some}(a,c)\\
&\qquad \text{else if } b \# (a,c)\\
&\qquad\qquad \text{then Some}((b\,a)\bullet(a,c)) \text{ else None})
\end{aligned}
$$

Let's check this for $[a].(a,c)$:

a 'applied to' $[a].(a,c)$ 'gives' Some(a,c) '$\lambda a.(a\,c)$'

b 'applied to' $[a].(a,c)$ 'gives' Some(b,c) '$\lambda b.(b\,c)$'

c 'applied to' $[a].(a,c)$ 'gives' None

d 'applied to' $[a].(a,c)$ 'gives' Some(d,c) '$\lambda d.(d\,c)$'

$\vdots$

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

$$
\begin{aligned}
&[a].(a,c) \stackrel{\text{def}}{=}\\
&\qquad (\lambda b.\text{if } a = b\\
&\qquad\qquad \text{then Some}(a,c)\\
&\qquad\qquad \text{else if } b \# (a,c)\\
&\qquad\qquad\qquad \text{then Some}((b\,a)\bullet(a,c)) \text{ else None})
\end{aligned}
$$

Let's check this for $[a].(a,c)$:

- a 'applied to' $[a].(a,c)$ 'gives' Some(a,c)
- b 'applied to' $[a].(a,c)$ 'gives' Some(b,c)
- c 'applied to' $[a].(a,c)$ 'gives' None
- d 'applied to' $[a].(a,c)$ 'gives' Some(d,c)

$\vdots$

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

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

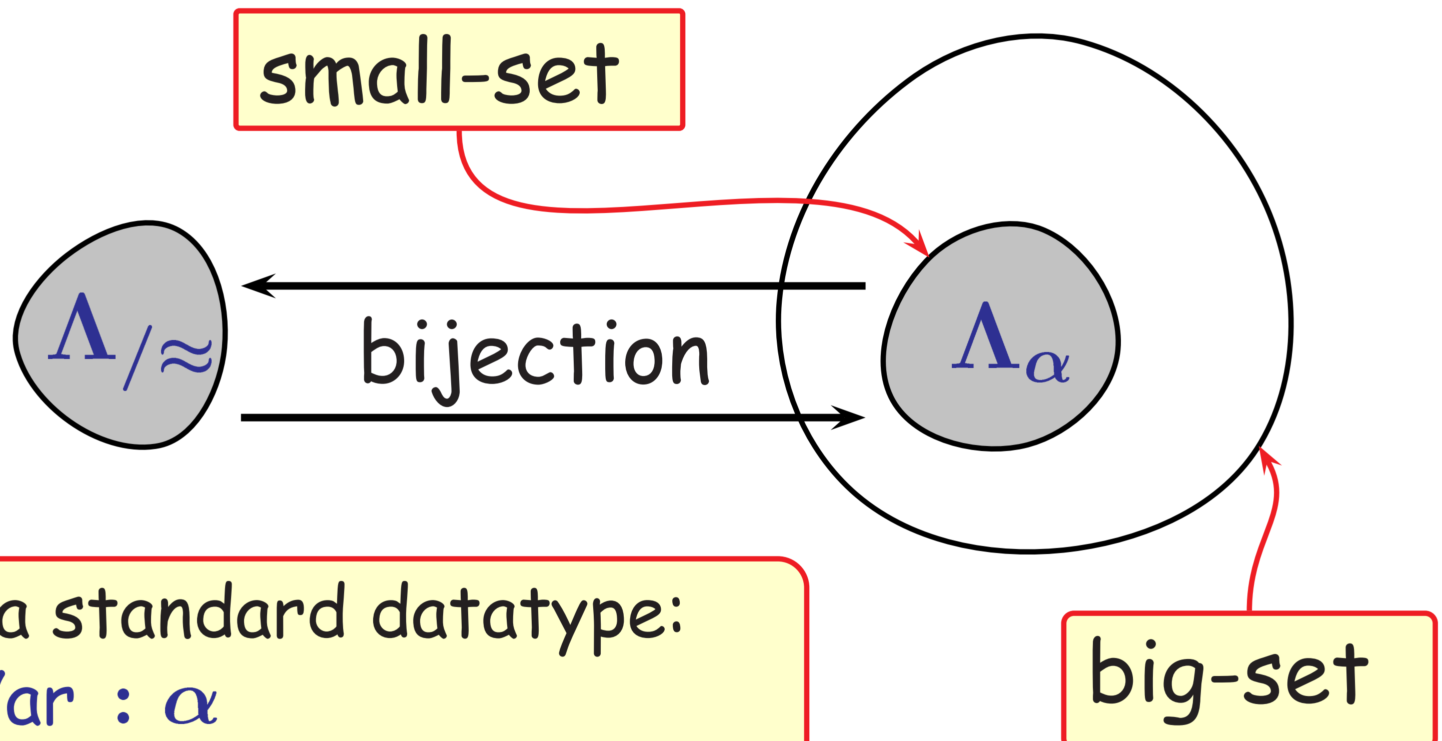
'$\lambda d.(d\,c)$'
$\vdots$

Nominal Datatypes

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

Nominal Datatypes

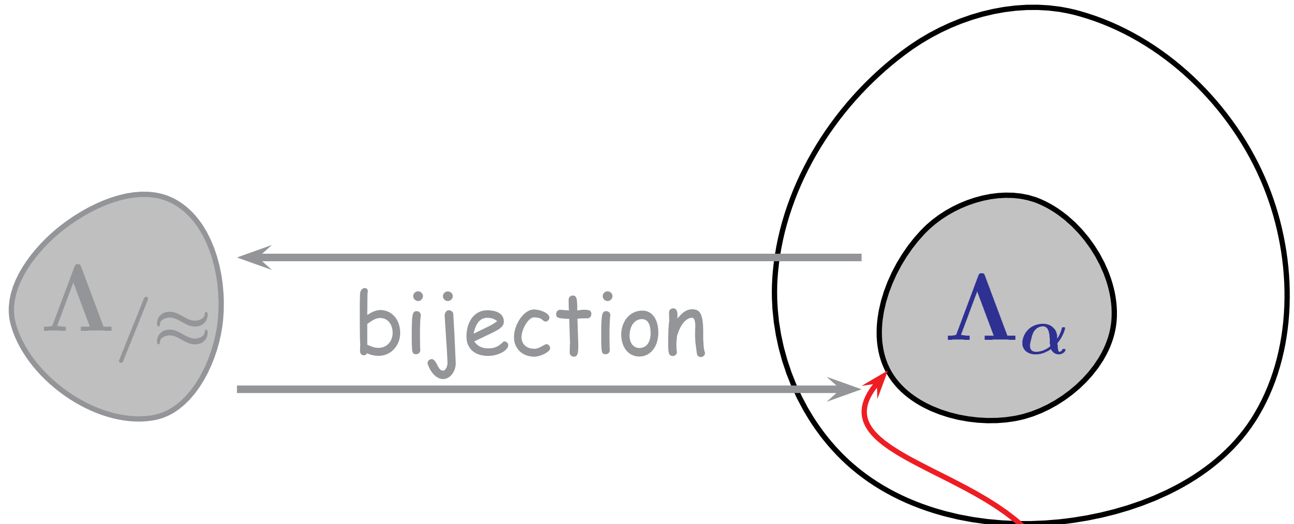
We define **inductively** α-equivalence classes of lambda-terms—but they still have **names**.



big set is a standard datatype:

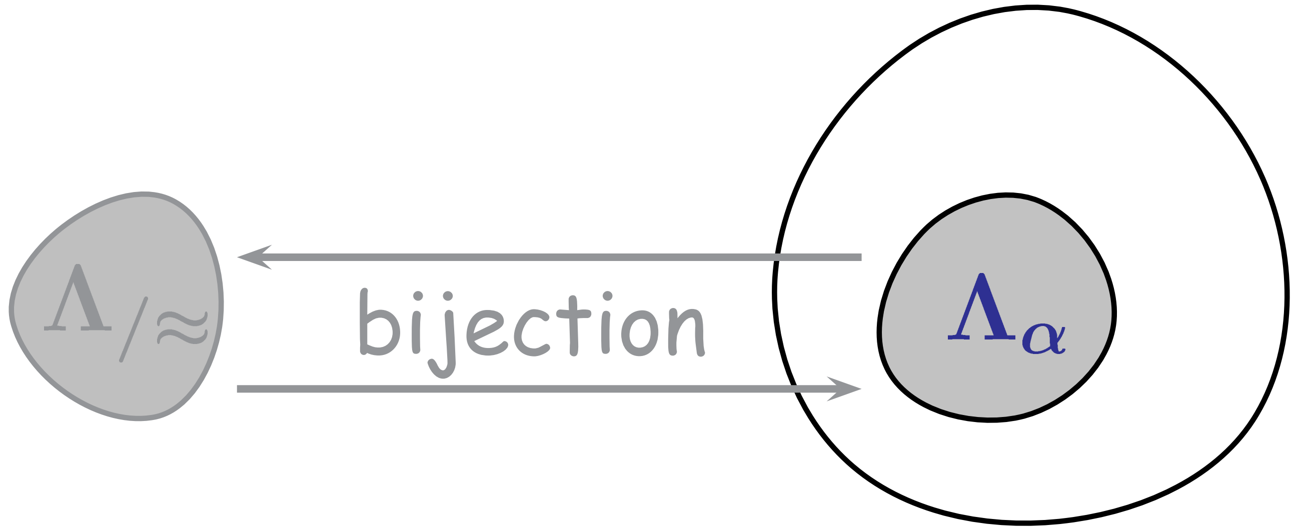
$$
\begin{array}{lll}
trm := & \mathsf{Var} & : \alpha \\
\quad\mid & \mathsf{App} & : trm \times trm \\
\quad\mid & \mathsf{Lam} & : \alpha \to trm\ \mathsf{option}
\end{array}
$$

Definition of Small-Set



$$
\begin{array}{rcl}
t ::= & & \mathsf{Var}(a) \\
& | & \mathsf{App}(t_1, t_2) \\
& | & \mathsf{Lam}\,[a].t
\end{array}
$$

Definition of Small-Set



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

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

Definition of Small-Set

Which also means that we have a familiar induction principle in place for Λ_α (in a moment). And all terms in Λ_α have finite support.

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

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

Definition of Small-Set

Which also means that we have a familiar induction principle in place for Λ_α (in a moment). And all terms in Λ_α have finite support.

$$
\begin{array}{ll}
\mathsf{supp}(\mathsf{Var}(a)) & = \{a\} \\
\mathsf{supp}(\mathsf{App}(t_1, t_2)) & = \mathsf{supp}(t1, t2) \\
\mathsf{supp}(\mathsf{Lam}\,[a].t) & = \mathsf{supp}([a].t) = \mathsf{supp}(t) - \{a\}
\end{array}
$$

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

Bijection

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

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

with the property

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

Struct. Induction on Λ_α

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

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

Structural Induction Principle:

$$\frac{\begin{array}{l} \forall a.\ P\,(\mathsf{Var}(a)) \\ \forall t_1, t_2.\ P\,t_1 \Rightarrow P\,t_2 \Rightarrow P\,(\mathsf{App}(t_1, t_2)) \\ \forall a, t.\ P\,t \Rightarrow P\,(\mathsf{Lam}\,[a].t) \end{array}}{\forall t.\ P\,t}$$

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.

"$\square$"

Outlook

- nominal induction-principles (over nominal datatypes and inductive definitions)
- why the present version of the axiomatic type-classes are fairly unwieldy for this work
- functions over nominal datatypes (what are the conditions that allow a definition by "recursion" over α-equivalence classes)

$$
\begin{array}{rcl}
(\mathsf{Var}\,a)[b:=s] & = & \mathsf{if}\; a=b \;\mathsf{then}\; s \;\mathsf{else}\; (\mathsf{Var}\,a) \\
(\mathsf{App}\,t_1\,t_2)[b:=s] & = & \mathsf{App}\,(t_1[b:=s])\,(t_2[b:=s \\
(\mathsf{Lam}\,[a].t)[b:=s] & = & \mathsf{Lam}\,[a].(t[b:=s]) \\
 & & \quad \mathsf{provided}\; a\,\#\,(b,s)
\end{array}
$$

Outlook

- nominal induction-principles (over nominal datatypes and inductive definitions)
- why the present version of the axiomatic
-

Nominal Datatype Package:

http://isabelle.in.tum.de/nominal/

Mailing List:

https://mailbroy.informatik.tu-muenchen.de/cgi-bin/mailman/listinfo/nominal-isabelle

$(\mathsf{Var}\ a)$

$$
\begin{array}{rcl}
(\mathsf{App}\ t_1\ t_2)[b := s] & = & \mathsf{App}\ (t_1[b := s])\ (t_2[b := s \\
(\mathsf{Lam}\ [a].t)[b := s] & = & \mathsf{Lam}\ [a].(t[b := s]) \\
 & & \text{provided}\ a \# (b, s)
\end{array}
$$

Outlook

- nominal induction-principles (over nominal datatypes and inductive definitions)
- why the present version of the axiomatic type-classes are fairly unwieldy for this work
- funct … what are th … inition by “re … classes)

The End?

$$
\begin{array}{rcl}
(\mathsf{Var}\,a)[b := s] & = & \mathsf{if}\ a = b\ \mathsf{then}\ s\ \mathsf{else}\ (\mathsf{Var}\,a)\\
(\mathsf{App}\,t_1\,t_2)[b := s] & = & \mathsf{App}\,(t_1[b := s])\,(t_2[b := s\\
(\mathsf{Lam}\,[a].t)[b := s] & = & \mathsf{Lam}\,[a].(t[b := s])\\
 & & \quad\mathsf{provided}\ a \mathrel{\#} (b, s)
\end{array}
$$