Quiz?

Assuming that \( a \) and \( b \) are distinct variables, is it possible to find \( \lambda \)-terms \( M_1 \) to \( M_7 \) that make the following pairs \( \alpha \)-equivalent?

1. \( \lambda a.\lambda b.(M_1 \ b) \) and \( \lambda b.\lambda a.(a \ M_1) \)
2. \( \lambda a.\lambda b.(M_2 \ b) \) and \( \lambda b.\lambda a.(a \ M_3) \)
3. \( \lambda a.\lambda b.(b \ M_4) \) and \( \lambda b.\lambda a.(a \ M_5) \)
4. \( \lambda a.\lambda b.(b \ M_6) \) 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): $\alpha$-Equivalence

The following rules define $\alpha$-equivalence on lambda-term (syntax-trees):

- $a \approx a \approx_{\text{atm}}$

- $t \approx s$ \quad $\lambda a.t \approx \lambda a.s$ \quad $\approx_{\text{lam}_1}$

- $t_1 \approx s_1$ \quad $t_2 \approx s_2$ \quad $t_1 t_2 \approx s_1 s_2 \approx_{\text{app}}$

- $t \approx (a b) \cdot s$ \quad $a \not\in \text{fv}(s)$ \quad $\lambda a.t \approx \lambda b.s \approx_{\text{lam}_2}$ assuming $a \neq b$
Recap (II): Support and Freshness

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

$$\text{supp}_\alpha x \overset{\text{def}}{=} \{ a \mid \text{infinite}\{ b \mid (a\ b) \cdot x \neq x \}\}$$

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

$$\alpha \# x \overset{\text{def}}{=} \alpha \not\in \text{supp}_\alpha(x)$$

I often drop the $\alpha$ in $\text{supp}_\alpha$. 
Nominal Abstractions

We are now going to specify what abstraction ‘abstractly’ means: it is an operation $\llbracket \_ \rrbracket.(\_): \alpha \Rightarrow \iota \Rightarrow \iota$ which has to satisfy:

\begin{align*}
\pi \cdot (\llbracket a \rrbracket.x) &= \llbracket \pi \cdot a \rrbracket.(\pi \cdot x) \\
\llbracket a \rrbracket.x &= \llbracket b \rrbracket.y \text{ iff } \\
(a = b \land x = y) \lor \\
(a \neq b \land x = (a \cdot b) \cdot y \land a \neq y)
\end{align*}

these two properties imply for finitely supported $x$

$\text{supp}(\llbracket a \rrbracket.x) = \text{supp}(x) - \{a\}$
Nominal Abstractions

Remember the definition of $\alpha$-equivalence from the beginning:

\[ t_1 \approx t_2 \]
\[ \therefore \lambda a.t_1 \approx \lambda a.t_2 \]
\[ a \neq b \quad t_1 \approx (a \ b) \cdot t_2 \quad a \not\in \text{fv}(t_2) \]
\[ \therefore \lambda a.t_1 \approx \lambda b.t_2 \]

\[ \pi \cdot ([a].x) = [\pi \cdot a] \cdot (\pi \cdot x) \]

\[ [a].x = [b].y \text{ iff } \]
\[ (a = b \land x = y) \lor \]
\[ (a \neq b \land x = (a \ b) \cdot y \land a \not\# y) \]

these two properties imply for finitely supported $x$

\[ \text{supp}([a].x) = \text{supp}(x) - \{a\} \]
Nominal Abstractions

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

- \(\pi \cdot ([a].x) = [\pi \cdot a] \cdot (\pi \cdot x)\)
- \([a].x = [b].y\) iff
  \(\begin{align*}
  (a = b \land x = y) \lor \\
  (a \neq b \land x = (a \ b) \cdot y \land a \not\equiv y)
  \end{align*}\)

these two properties imply for finitely supported \(x\)

\(\text{supp}([a].x) = \text{supp}(x) - \{a\}\)
Freshness and Abstractions

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

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

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

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

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

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

$$c \# x \iff (a \ c) \cdot c \# (a \ c) \cdot x \iff a \# x$$
Given $pt_{\alpha,\nu}$, finite(supp $x$) and $a \neq b$ then

$$a \not\equiv x \iff a \not\equiv [b].x$$

Proof. There exists a $c$ with $c \not\equiv (a, b, x, [b].x)$.

$(\Rightarrow)$ From $c \not\equiv [b].x$ we also have

$$(a \cdot c) \cdot c \not\equiv (a \cdot c) \cdot [b].x$$

and

$$a \not\equiv [b].(a \cdot c) \cdot x$$

Because $a \not\equiv x$ and $c \not\equiv x$, $(a \cdot c) \cdot x = x$. 

Freshness and Abstractions

We also have

\[ a \not\# [a].x \]

Again from \( c \not\# (a, x, [a].x) \) we can infer

\[ c \not\# [a].x \iff (a c)\cdot c \not\# (a c)\cdot [a].x \]
\[ \iff a \not\# [c].(a c)\cdot x. \]

However:

\[ [c].(a c)\cdot x = [a].x \]

(since \( c \not\equiv a \), \( [c].(a c)\cdot x = [a].x \)
\[ \text{iff } (a c)\cdot x = (a c)\cdot x \land c \not\# x \)
So we have shown that

\[
\begin{align*}
    a \neq b \quad & a \not\in x \\
    a \not\in [b].x \quad & a \not\in [a].x
\end{align*}
\]

Again from

\[
\begin{align*}
    c \not\in x \quad & \text{def} \\
    a \not\in x = a \not\in \text{supp}(x)
\end{align*}
\]

therefore

\[
\begin{align*}
    \text{supp}([a].x) = \text{supp}(x) - \{a\} \\
    [c].(a \circ c) \cdot x = [a].x
\end{align*}
\]

(since \(c \neq a\),

\[
\begin{align*}
    [c].(a \circ c) \cdot x = [a].x \\
    \text{iff} \quad (a \circ c) \cdot x = (a \circ c) \cdot x \land c \not\in x
\end{align*}
\]

Munich, 15. February 2006 - p.7 (2/2)
Nominal Abstractions

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

\[\pi \cdot ([a].x) = [\pi \cdot a].(\pi \cdot x)\]

\[[a].x = [b].y \text{ iff }\]
\[(a = b \land x = y) \lor (a \neq b \land x = (a b) \cdot y \land a \not\# y)\]

Are there any structures that satisfy these properties? Are there any structures that are “supported” in Isabelle/HOL?
Possibilities

- $\alpha$-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 $\text{abs}$ which "abstracts" a variable: $\text{abs}(x, t) \mapsto \lambda(\ldots)$)

- 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

- $\alpha$-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 $\lambda (1, a)$

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

Remember the user will only see the “axioms” from the previous slide.
Function \([a].t \equiv [\lambda a.t]_\alpha\)

\([a].t \overset{def}{=} (\lambda b. \text{if } a = b \\
                             \text{then } \text{Some}(t) \\
                             \text{else if } b \neq t \text{ then } \text{Some}((b \ a) \cdot t) \text{ else } \text{None})\)

type: \(\alpha \rightarrow \iota\) option
Function \([a].t \equiv [\lambda a.t]_\alpha\)

\[ [a].t \overset{\text{def}}{=} (\lambda b. \text{if } a = b \text{ then } \text{Some}(t) \text{ else if } b \neq t \text{ then } \text{Some}((b a) \cdot t) \text{ else None}) \]

This is supposed to stand for the \(\alpha\)-equivalence class of \(\lambda a.t\).
Function \([a].t \equiv [\lambda a.t]_\alpha\)

\([a].(a, c) \overset{\text{def}}{=} (\lambda b. \text{if } a = b \\
\quad \text{then } \text{Some}(a, c) \\
\quad \text{else if } b \neq (a, c) \\
\quad \quad \text{then } \text{Some}((b a) \cdot (a, c)) \text{ else None})\)

Let’s check this for \([a].(a, c)\):
Function \([a].t \equiv [\lambda a.t]_\alpha\)

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

Let’s check this for \([a].(a, c)\):

- \(a\) ‘applied to’ \([a].(a, c)\) ‘gives’ Some\((a, c)\)
Function \([a].t \equiv [\lambda a.t]_\alpha\)

\([a].(a, c) \overset{\text{def}}{=}\)
\[
(\lambda b. \text{if } a = b \text{ then } \text{Some}(a, c) \text{ else if } b \neq (a, c) \text{ then } \text{Some}((b \cdot a) \cdot (a, c)) \text{ else None})
\]

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' \(\text{Some}(b, c)\)
Function \([a].t \equiv [\lambda a.t]_\alpha\)

\[\begin{align*}
[a].(a, c) \overset{\text{def}}{=} \\
& (\lambda b. \text{if } a = b \\
& \quad \text{then Some}(a, c) \\
& \quad \text{else if } b \neq (a, c) \\
& \quad \text{then Some}((b a) \cdot (a, c)) \text{ else None})
\end{align*}\]

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’ \(\text{None}\)
Function \([a].t \equiv [\lambda a.t]_\alpha\)

\([a].(a, c) \overset{\text{def}}{=} (\lambda b. \text{if } a = b \text{ then } \text{Some}(a, c) \text{ else if } b \neq (a, c) \text{ then } \text{Some}((b \cdot a) \cdot (a, c)) \text{ else } \text{None})\)

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’ \(\text{None}\)
- \(d\) ‘applied to’ \([a].(a, c)\) ‘gives’ \(\text{Some}(d, c)\)
  
- : 
  
  : 

Function \([a].t \equiv [\lambda a.t]_\alpha\)

\[
[a].(a, c) \overset{\text{def}}{=} \quad (
\lambda b. \text{if } a = b \\
\quad \text{then } \text{Some}(a, c)
\quad \text{else if } b \neq (a, c) \\
\quad \text{then } \text{Some}((b a) \cdot (a, c)) \text{ else } \text{None}
\)

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 \([a].t \equiv [\lambda a.t]_{\alpha}\)

\[\begin{align*}
[a].(a, c) & \overset{\text{def}}{=} \\
& (\lambda b. \text{if } a = b \\
& \quad \text{then } \text{Some}(a, c) \\
& \quad \text{else if } b \not\equiv (a, c) \\
& \quad \text{then } \text{Some}((ba) \cdot (a, c)) \text{ else None}
\end{align*}\]

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' \(\text{None}\)
- \(d\) 'applied to' \([a].(a, c)\) 'gives' \(\text{Some}(d, c)\)
- \(\ldots\)
Nominal Datatypes

We define inductively $\alpha$-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:

\[
\text{trm ::= Var : } \alpha \\
\text{    | App : trm } \times \text{trm} \\
\text{    | Lam : } \alpha \rightarrow \text{trm option}
\]
Definition of Small-Set

\[ \Lambda / \approx \overset{\text{bijection}}{\longrightarrow} \Lambda_\alpha \]

\[ t ::= \text{Var}(a) \mid \text{App}(t_1, t_2) \mid \text{Lam } [a].t \]
Definition of Small-Set

\[ \Lambda/\cong \quad \text{bijection} \quad \Lambda_\alpha \]

\[
\begin{align*}
\text{Var}(a) & \in \Lambda_\alpha \\
\text{App}(t_1, t_2) & \in \Lambda_\alpha \\
\text{Lam}[a].t & \in \Lambda_\alpha
\end{align*}
\]
Which also means that we have a familiar induction principle in place for $\Lambda_\alpha$ (in a moment). And all terms in $\Lambda_\alpha$ have finite support.

\[
\begin{align*}
\text{Var}(\alpha) & \in \Lambda_\alpha \\
\text{App}(t_1, t_2) & \in \Lambda_\alpha \\
\text{Lam}[\alpha].t & \in \Lambda_\alpha
\end{align*}
\]
Definition of Small-Set

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

\[
\begin{align*}
\text{supp}(\text{Var}(a)) &= \{a\} \\
\text{supp}(\text{App}(t_1, t_2)) &= \text{supp}(t_1, t_2) \\
\text{supp}(\text{Lam} [a].t) &= \text{supp}([a].t) = \text{supp}(t) - \{a\}
\end{align*}
\]

$\therefore t \in \Lambda_\alpha \quad \Rightarrow \quad \text{Lam} [a].t \in \Lambda_\alpha$
In order to show that $\Lambda/\approx$ and $\Lambda_\alpha$ are bijective we define a function $q$ from $\Lambda$ to $\Lambda_\alpha$:

\[
\begin{align*}
q(a) & \overset{\text{def}}{=} \text{Var}(a) \\
q(t_1 t_2) & \overset{\text{def}}{=} \text{App}(q(t_1), q(t_2)) \\
q(\lambda a.t) & \overset{\text{def}}{=} \text{Lam} [a].q(t)
\end{align*}
\]

with the property

\[t_1 \approx t_2 \iff q(t_1) = q(t_2)\]
Struct. Induction on $\Lambda_{\alpha}$

\[
\begin{align*}
\text{Var}(a) & \in \Lambda_{\alpha} & t_1 & \in \Lambda_{\alpha} & t_2 & \in \Lambda_{\alpha} \\
\text{App}(t_1, t_2) & \in \Lambda_{\alpha} & t & \in \Lambda_{\alpha} \\
\text{Lam} [a].t & \in \Lambda_{\alpha}
\end{align*}
\]

Structural Induction Principle:

\[
\begin{align*}
\forall a. & \quad P (\text{Var}(a)) \\
\forall t_1, t_2. & \quad P t_1 \Rightarrow P t_2 \Rightarrow P (\text{App}(t_1, t_2)) \\
\forall a, t. & \quad P t \Rightarrow P (\text{Lam} [a].t) \\
\forall t. & \quad P t
\end{align*}
\]
Substitution Lemma: If $x \not\equiv y$ and $x \not\in \text{FV}(L)$, 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 $x \not\in \text{FV}(L)$ 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
  \[
  (\lambda z. M_1)[x := N][y := L] \\
  \equiv \lambda z.(M_1[x := N][y := L]) \\
  \equiv \lambda z.(M_1[y := L][x := N[y := L]]) \\
  \equiv (\lambda z. M_1)[y := L][x := N[y := L]].
  \]

- **Case 3**: $M \equiv M_1 M_2$. The statement follows again from the induction hypothesis. “□”
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 $\alpha$-equivalence classes)

$$(\text{Var } a)[b := s] = \text{if } a = b \text{ then } s \text{ else } (\text{Var } a)$$

$$(\text{App } t_1 t_2)[b := s] = \text{App } (t_1[b := s]) (t_2[b := s])$$

$$(\text{Lam } [a].t)[b := s] = \text{Lam } [a].(t[b := s])$$

provided $a \not\equiv (b, s)$
nominal induction-principles (over nominal datatypes and inductive definitions)

why the present version of the axiomatic type-classes is fairly unwieldy for this

functions over nominal datatypes (what are the conditions that allow a definition
by "recursion" over /AB-equivalence classes)

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

Nominal Datatype Package:
http://isabelle.in.tum.de/nominal/

Mailing List:
https://mailbroy.informatik.tu-muenchen.de/cgi-bin/mailman/listinfo/nominal-isabelle
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 $\mathbb{A}$-equivalence classes)

(Var $a)[b := s] = \text{if } a = b \text{ then } s \text{ else } (\text{Var } a)$
(App $t_1 t_2)[b := s] = \text{App } (t_1[b := s]) (t_2[b := s])$
(Lam $[a].t)[b := s] = \text{Lam } [a].(t[b := s])$

provided $a \not\# (b, s)$

Munich, 15. February 2006 - p.16 (3/3)