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?

Quiz

Assuming that a and b are distinct variables, is it possible to find λ-terms M_1 to M_7 that make the foll

- λ
- λ
- λ
- $\lambda a.\lambda b.(b\ M_6)$ and $\lambda a.\lambda a.(a\ M_7)$

Don't be fooled by the question's innocent look: some lambda-calculus experts had problems with it. Also, the really interesting question is the one below.

Quiz will be solved in lecture 2. ;o)

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

Nominal Logic and Reasoning about α-Equivalent Syntax

Christian Urban

every Tuesday from 14:15 to 16:00 in Room E27

(except 26. April, 17. May and 5. July)

Very Rough Plan

- syntax, syntax, syntax (motivation: this lecture).

Very Rough Plan

- syntax, syntax, syntax (motivation: this lecture).
- lambda-terms as (first-order) abstract syntax trees; nominal unification – remember the quiz.

Very Rough Plan

- syntax, syntax, syntax (motivation: this lecture).
- lambda-terms as (first-order) abstract syntax trees; nominal unification – remember the quiz.
- mathematics behind the nominal approach; permutation groups, nominal sets etc.

Very Rough Plan

- syntax, syntax, syntax (motivation: this lecture).
- lambda-terms as (first-order) abstract syntax trees; nominal unification – remember the quiz.
- mathematics behind the nominal approach; permutation groups, nominal sets etc.
- actual reasoning with the nominal ideas (CR, maybe SN,. . .); Isabelle, if wanted.

Very Rough Plan

- syntax, syntax, syntax (motivation: this lecture).
- lambda-terms as (first-order) abstract syntax trees; nominal unification - remember the quiz.
- mathematics behind the nominal approach; permutation groups, nominal sets etc.
- actual reasoning with the nominal ideas (CR, maybe SN,. . .); Isabelle, if wanted.
- Slides will be at:

  www.mathematik.uni-muenchen.de/~urban/teaching.html

What is the Problem

(Surely you know this, but just to make sure.)

- Mathematical version:

$$\int_0^1 x^2 + y\,dx = y + \frac{1}{3}$$

What is the Problem

(Surely you know this, but just to make sure.)

- Mathematical version:

$$\int_0^1 x^2 + y\,dx = y + \frac{1}{3}$$

naïvely applying $[y := x]$ gives the incorrect equation

$$\int_0^1 x^2 + x\,dx = x + \frac{1}{3}$$

What is the Problem

(Surely you know this, but just to make sure.)

- Computer-science version:

$$\lambda a.(b\,a)[b := a] \xrightarrow{\text{naïvely}} \lambda a.(a\,a)$$

Naïve substitution does not respect α-equivalence. What needs to be renamed is determined by subtle side-constraints. This makes formal reasoning hard.

Another Problem

(If you know it, you probably choose to ignore it.)

Assume we define the set Λ of lambda-terms/ syntax-trees **inductively** by the grammar:

t	$::=$	a	variables
	$\mid$	$t\,t$	applications
	$\mid$	$\lambda a.t$	abstractions

Another Problem

(If you know it, you probably choose to ignore it.)

Assume we define the set Λ of lambda-terms/ syntax-trees **inductively** by the grammar:

$$
\begin{array}{rcll}
t & ::= & a & \text{variables} \\
  & \mid & t\,t & \text{applications} \\
  & \mid & \lambda a.t & \text{abstractions}
\end{array}
$$

We can easily define functions over Λ by structural recursion; for example

$$
\begin{array}{lcl}
\mathsf{depth}\,(a) & \stackrel{\mathsf{def}}{=} & 0 \\
\mathsf{depth}\,(t\,t') & \stackrel{\mathsf{def}}{=} & 1 + \mathsf{max}(\mathsf{depth}(t), \mathsf{depth}(t')) \\
\mathsf{depth}\,(\lambda a.t) & \stackrel{\mathsf{def}}{=} & 1 + \mathsf{depth}(t)
\end{array}
$$

Another Problem

(If you know it, you probably choose to ignore it.)

Assume we define the set Λ of lambda-terms/ syntax-trees **inductively** by the grammar:

$$\begin{array}{rcll} t & ::= & a & \text{variables} \\ & \mid & t\,t & \text{applications} \\ & \mid & \lambda a.t & \text{abstractions} \end{array}$$

However, if we form the quotient-set $\Lambda_{/=_\alpha}$ then what is the structural recursion principle?

$$\begin{array}{rcl} (a)\,[b := s] & \stackrel{\text{def}}{=} & \text{if } a = b \text{ then } s \text{ else } a \\ (t\,t')\,[b := s] & \stackrel{\text{def}}{=} & (t[b := s])\,(t'[b := s]) \\ (\lambda a.t)\,[b := s] & \stackrel{\text{def}}{=} & \lambda a.(t[b := s]) \quad \text{plus conditions} \end{array}$$

Another Problem

(If you know it, you probably choose to ignore it.)

Assume we define the set Λ of lambda-terms/
syntax-tre ar:

Equating a set by a relation does **not** produce automatically an inductive set.

However, if we form the quotient-set $\Lambda_{/=_\alpha}$ then what is the structural recursion principle?

$$
\begin{aligned}
(a)\,[b := s] &\stackrel{\text{def}}{=} \text{if } a = b \text{ then } s \text{ else } a\\
(t\,t')\,[b := s] &\stackrel{\text{def}}{=} (t[b := s])\,(t'[b := s])\\
(\lambda a.t)\,[b := s] &\stackrel{\text{def}}{=} \lambda a.(t[b := s]) \quad \text{plus conditions}
\end{aligned}
$$

Another Problem

(If you know it, you probably choose to ignore it.)

Assume we define the set Λ of lambda-terms/ syntax-trees **inductively** by the grammar:

$$\begin{array}{rcll} t & ::= & a & \text{variables} \\ & | & t\,t & \text{applications} \\ & | & \lambda a.t & \text{abstractions} \end{array}$$

However, if we form the quotient-set $\Lambda_{/=_\alpha}$ then what is the structural recursion principle?

$$\begin{array}{rcl} (a)\,[b := s] & \stackrel{\text{def}}{=} & \text{if } a = b \text{ then } s \text{ else } a \\ (t\,t')\,[b := s] & \stackrel{\text{def}}{=} & (t[b := s])\,(t'[b := s]) \\ (\lambda a.t)\,[b := s] & \stackrel{\text{def}}{=} & \lambda a.(t[b := s]) \quad \text{plus conditions} \end{array}$$

Another Problem

A

sy

Of course, this can be turned into a proper definition — by recursion on the depth of α-equated lambda-terms.

But for this we need to lift the depth function from raw to α-equated lambda-terms, because clearly depth can also not be directly defined by structural recursion.

H ... en

w

$$
\begin{array}{rcl}
(a)\,[b := s] & \stackrel{\text{def}}{=} & \text{if } a = b \text{ then } s \text{ else } a \\
(t\,t')\,[b := s] & \stackrel{\text{def}}{=} & (t[b := s])\,(t'[b := s]) \\
(\lambda a.t)\,[b := s] & \stackrel{\text{def}}{=} & \lambda a.(t[b := s]) \quad \text{plus conditions}
\end{array}
$$

Barendregt-style Naming Convention

Roughly:

If lambda-terms $M_1, \ldots, M_n$ occur in a certain context, their bound variables are chosen to be different from the free variables.

or (my version)

Close your eyes and hope everything goes well.*

*not to be tried whilst driving

In Barendregt's CR-Proof

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.

In Barendregt's CR-Proof

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.

- M is a variable.
  - $M \equiv x$. Then both sides equal $N[y := L]$ since $x \not\equiv y$.
  - $M \equiv y$. Then both sides equal L, for $x \notin FV(L)$ implies $L[x := \ldots] \equiv L$.
  - $M \equiv z \not\equiv x, y$. Then both sides equal z.

In Barendregt's CR-Proof

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.

- $M \equiv M_1\, M_2$ The statement follows from the induction hypothesis (this case is usually easy).

In Barendregt's CR-Proof

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.

- $M \equiv \lambda z.M_1$. 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:

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

There are numerous techniques to formalise this proof.

We shall formalise this proof as follows. . .

There are numerous techniques to formalise this proof.

We shall formalise this proof as follows...

Again do not get fooled, there is something non-trivial going on 'behind the scenes'.

Weakening Property

…because sometimes eyes just cannot be closed

Example: weakening property for the simply-typed lambda-calculus

$$\frac{a : \tau \in \Gamma}{\Gamma \vdash a : \tau} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \longrightarrow \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\, t_2 : \tau_2}$$

$$\frac{\Gamma, a : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda a.t : \tau_1 \longrightarrow \tau_2}\; a \notin \mathsf{dom}(\Gamma)$$

Weakening Property

. . . because so closed

Example: weak

simply-typed l

Assume for the moment that Γ is a set of variable $\times$ type-pairs with some well-formedness constraints.

$$\frac{a : \tau \in \Gamma}{\Gamma \vdash a : \tau} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\, t_2 : \tau_2}$$

$$\frac{\Gamma, a : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda a.t : \tau_1 \rightarrow \tau_2}\; a \notin \mathrm{dom}(\Gamma)$$

Weakening Property

…because sometimes eyes just cannot be closed

Example: weakening property for the simply-typed lambda-calculus

$$\frac{a : \tau \in \Gamma}{\Gamma \vdash a : \tau} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\, t_2 : \tau_2}$$

$$\frac{\Gamma, a : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda a.t : \tau_1 \rightarrow \tau_2}\; a \notin \mathsf{dom}(\Gamma)$$

If $\Gamma \vdash t : \tau$, then also $\Gamma, a : \tau' \vdash t : \tau$.

Weakening Property

...because sometimes eyes just cannot be closed

Example: weakening property for the simply-typed lambda-calculus

$$\frac{a : \tau \in \Gamma}{\Gamma \vdash a : \tau} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \longrightarrow \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\, t_2 : \tau_2}$$

$$\frac{\Gamma, a : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda a.t : \tau_1 \longrightarrow \tau_2}\; a \notin \mathrm{dom}(\Gamma)$$

$$(\forall \Gamma)(\forall t)(\forall \tau)\ \Gamma \vdash t : \tau \Rightarrow$$
$$(\forall \tau')(\forall a \notin \mathrm{dom}(\Gamma))\ \Gamma, a : \tau' \vdash t : \tau$$

Raw Lambda-Terms? No!

This property does **not** hold for raw lambda-terms: since

$$\frac{a : \tau \vdash a : \tau}{\varnothing \vdash \lambda a.a : \tau \to \tau}$$

is derivable, but

$$a : \tau' \vdash \lambda a.a : \tau \to \tau$$

is not, because

$$\frac{\Gamma, a : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda a.t : \tau_1 \to \tau_2} \; a \notin \mathrm{dom}(\Gamma)$$

Raw Lambda-Terms? No!

This property does **not** hold for raw lambda-terms: since

$$\frac{a : \tau \vdash a : \tau}{\emptyset \vdash \lambda a.a : \tau \to \tau}$$

is derivab

We really mean weakening for α-equated lambda-terms.

$$a : \tau' \vdash \lambda a.a : \tau \to \tau$$

is not, because

$$\frac{\Gamma, a : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda a.t : \tau_1 \to \tau_2}\ a \notin \mathsf{dom}(\Gamma)$$

Let's Make This Explicit

Nobody usually bothers, but let's explicitly write $[t]_\alpha$ for the set of lambda-terms/syntax-trees α-equivalent with t:

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

Typing-rules for α-equated lambda-terms:

$$\frac{a : \tau \in \Gamma}{\Gamma \vdash [a]_\alpha : \tau} \qquad \frac{\Gamma \vdash [t_1]_\alpha : \tau_1 \longrightarrow \tau_2 \quad \Gamma \vdash [t_2]_\alpha : \tau_1}{\Gamma \vdash [t_1\, t_2]_\alpha : \tau_2}$$

$$\frac{\Gamma, a : \tau_1 \vdash [t]_\alpha : \tau_2}{\Gamma \vdash [\lambda a.t]_\alpha : \tau_1 \longrightarrow \tau_2}\; a \notin \mathrm{dom}(\Gamma)$$

Let's Make This Explicit

Nobody usually bothers, but let's explicitly write $[t]_\alpha$ … ees α-eq…

Remember, we write $[t_1]_\alpha$, **but** we mean a set of terms $\{t_1, \ldots\}$ — namely the α-equivalence class of t_1.

Typing-rules for α-equated lambda-terms:

$$\frac{a : \tau \in \Gamma}{\Gamma \vdash [a]_\alpha : \tau} \qquad \frac{\Gamma \vdash [t_1]_\alpha : \tau_1 \longrightarrow \tau_2 \quad \Gamma \vdash [t_2]_\alpha : \tau_1}{\Gamma \vdash [t_1\, t_2]_\alpha : \tau_2}$$

$$\frac{\Gamma, a : \tau_1 \vdash [t]_\alpha : \tau_2}{\Gamma \vdash [\lambda a.t]_\alpha : \tau_1 \longrightarrow \tau_2} \; a \notin \mathrm{dom}(\Gamma)$$

Attempting the Proof

We proceed by rule induction and try to show that the predicate $\varphi(\Gamma; [t]_\alpha; \tau)$ given by

$$(\forall \tau')(\forall a' \notin \mathrm{dom}(\Gamma))\ \Gamma, a' : \tau' \vdash [t]_\alpha : \tau$$

is closed under the axiom and the two inference rules. Interesting case:

$$\frac{\Gamma, a : \tau_1 \vdash [t]_\alpha : \tau_2}{\Gamma \vdash [\lambda a.t]_\alpha : \tau_1 \to \tau_2}\ a \notin \mathrm{dom}(\Gamma)$$

Attempting the Proof

We proceed by r
that the predica

$$(\forall \tau')(\forall a' \notin$$

is closed under t
rules. Interestin

We know (for the premise):

1. $\varphi(\Gamma, a : \tau_1; [t]_\alpha; \tau_2)$
2. $a \notin \mathrm{dom}(\Gamma)$

We have to prove:

$$\Gamma, a' : \tau' \vdash [\lambda a.t]_\alpha : \tau_1 \longrightarrow \tau_2$$

for **all** τ' and **all** $a' \notin \mathrm{dom}(\Gamma)$.

$$\frac{\Gamma, a : \tau_1 \vdash [t]_\alpha : \tau_2}{\Gamma \vdash [\lambda a.t]_\alpha : \tau_1 \longrightarrow \tau_2}\; a \notin \mathrm{dom}(\Gamma)$$

Attempting the Proof

We proceed by r
that the predica

$$(\forall \tau')(\forall a' \notin$$

is closed under t
rules. Interestin

We know (for the premise):

1. $\varphi(\Gamma, a : \tau_1; [t]_\alpha; \tau_2)$
2. $a \notin \mathrm{dom}(\Gamma)$

We have to prove:

$$\Gamma, a' : \tau' \vdash [\lambda a.t]_\alpha : \tau_1 \rightarrow \tau_2$$

for **all** τ' and **all** $a' \notin \mathrm{dom}(\Gamma)$.

$$\frac{\Gamma, a : \tau_1 \vdash [t]_\alpha : \tau_2}{\Gamma \vdash [\lambda a.t]_\alpha : \tau_1 \longrightarrow \tau_2} \quad a \notin \mathrm{dom}(\Gamma)$$

But this fails for $a' = a$!

Because

The proof does not work:

case $a' \neq a$:

We have

$$\Gamma, a' : \tau', a : \tau_1 \vdash [t]_\alpha : \tau_2$$

$$\Downarrow$$

$$\Gamma, a' : \tau' \vdash [\lambda a.t]_\alpha : \tau_1 \rightarrow \tau_2$$

as desired.

case $a' = a$:

We only have

$$\Gamma, a : \tau_1 \vdash [t]_\alpha : \tau_2$$

$$\Downarrow$$

$$\Gamma \vdash [\lambda a.t]_\alpha : \tau_1 \rightarrow \tau_2$$

which is not what we want to prove.

Moral of this Example

- Does this mean the weakening property does **not** hold for the simply-typed lambda-calculus?

  Clearly, **NO**!

  Just our simple-minded reasoning did not work. We have to take into account some facts about α-equivalent classes and their typings.

- You cannot just close your eyes. There is something subtle going on to make Barendregt's variable convention to work.

Now We Start in Earnest

Some bookkeeping first.

We introduce **atoms**. Everything that is **bound**, **binding** and **bindable** is an atom (independent from the language at hand).

a countable infinite set — this will be important on later on.

Now We Start in Earnest

Some bookkeeping first.

We introduce **atoms**. Everything that is **bound**, **binding** and **bindable** is an atom (independent from the language at hand).

Now We Start in Earnest

Some bookkeeping first.

We introduce **atoms**. Everything that is **bound**, **binding** and **bindable** is an atom (independent from the language at hand).

example lambda-calculus

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

a and b are atoms—bound and binding

Now We Start in Earnest

Some bookkeeping first.

We introduce **atoms**. Everything that is **bound**, **binding** and **bindable** is an atom (independent from the language at hand).

example lambda-calculus

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

c is an atom—bindable

Now We Start in Earnest

Some bookkeeping first.

We introduce **atoms**. Everything that is **bound**, **binding** and **bindable** is an atom (independent from the language at hand).

example lambda-calculus

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

now c is bound

Now We Start in Earnest

Some bookkeeping first.

We introduce **atoms**. Everything that is **bound**, **binding** and **bindable** is an atom (independent from the language at hand).

example integrals

$$\int_0^1 x^2 + y \, dx$$

x is an atom—bound and binding

Now We Start in Earnest

Some bookkeeping first.

We introduce **atoms**. Everything that is **bound**, **binding** and **bindable** is an atom (independent from the language at hand).

example integrals

$$\int_{-\infty}^{\infty}\left(\int_{0}^{1} x^2 + y\,dx\right)dy$$

y is an atom—bindable

Now We Start in Earnest

Some bookkeeping first.

We introduce **atoms**. Everything that is **bound**, **binding** and **bindable** is an atom (independent from the language at hand).

example integrals

$$\int_0^1 x^2 + y\,dx$$

0, 1 and 2 are constants

Now We Start in Earnest

Some bookkeeping first.

We introduce **atoms**. Everything that is **bound**, **binding** and **bindable** is an atom (independent from the language at hand).

example integrals

$$\int_{-\infty}^{\infty} \left(\int_{0}^{1} x^{2} + y \, dx \right) d2$$

binding 2 does not make sense

Now We Start in Earnest

Some bookkeeping first.

We introduce **atoms**. Everything that is **bound**,
binding ... ent
from t...

exampl...

Why atoms? Because an operation we introduce shortly will act on atoms **only** and leaves everything else alone.

$$\int_{-\infty}^{\infty}\left(\int_0^1 x^2 + y\,dx\right)d2$$

binding 2 does not make sense

Swappings

Recall the problem: substitution does not respect α-equivalence, e.g.

$$\lambda a.b \qquad\qquad \lambda c.b$$

Swappings

Recall the problem: substitution does not respect α-equivalence, e.g.

$$\begin{aligned}[b := a]\boldsymbol{\lambda a.b} &= \boldsymbol{\lambda a.a}\end{aligned} \qquad \begin{aligned}[b := a]\boldsymbol{\lambda c.b} &= \boldsymbol{\lambda c.a}\end{aligned}$$

Swappings

Recall the problem: substitution does not respect α-equivalence, e.g.

$$\begin{aligned}[b:=a]\lambda a.b &= \lambda a.a\end{aligned} \qquad \begin{aligned}[b:=a]\lambda c.b &= \lambda c.a\end{aligned}$$

Traditional Solution: replace $[b:=a]t$ by a more complicated, 'capture-avoiding' form of substitution.

Swappings

Recall the problem: substitution does not respect α-equivalence, e.g.

$$(b\,a)\bullet \lambda a.b = \lambda b.a \qquad\qquad (b\,a)\bullet \lambda c.b = \lambda c.a$$

Nice Alternative: use a less complicated operation for renaming

$(b\,a)\bullet t \stackrel{\text{def}}{=}$ swap **all** occurrences of b and a in t

Swappings

Recall the problem: substitution does not respect α-equivalence, e.g.

$$(b\,a)\bullet \lambda a.b = \lambda b.a \qquad\qquad (b\,a)\bullet \lambda c.b = \lambda c.a$$

<u>Nice Alternative:</u> use a less complicated operation for renaming

$(b\,a)\bullet t \stackrel{\text{def}}{=}$ swap **all** occurrences of b and a in t

be they bound, binding or bindable

Swappings

Recall the problem: substitution does not respect α-equivalence, e.g.

$$(b\,a)\bullet \lambda a.b = \lambda b.a \qquad\qquad (b\,a)\bullet \lambda c.b = \lambda c.a$$

<u>Nice Alternative:</u> use a less complicated operation for renaming

$(b\,a)\bullet t \stackrel{\text{def}}{=}$ swap **all** occurrences of b and a in t

Unlike for $[b:=a](-)$, for $(b\,a)\bullet(-)$ we do have if $t =_\alpha t'$ then $(b\,a)\bullet t =_\alpha (b\,a)\bullet t'$.

Permutations

We shall extend 'swappings' to '(finite) lists of swappings'

$$(a_1\, b_1) \ldots (a_n\, b_n),$$

also called **permutations** (we shall often write π for them). Permutations are **bijective** mappings from atoms to atoms. For example

$$\pi = \begin{pmatrix} a \mapsto b \\ b \mapsto a \\ c \mapsto c \end{pmatrix} = (c\, b)(a\, b)(a\, c)$$

Permutations

We shall extend 'swappings' to '(finite) lists of swappings'

$$(a_1\, b_1) \ldots (a_n\, b_n),$$

also called **permutations** (we shall often write π for them). Permutations are **bijective** mappings from atoms to atoms. For example

$$\pi = \begin{pmatrix} a \mapsto b \\ b \mapsto a \\ c \mapsto c \end{pmatrix} \qquad (c\, b)(a\, b)(a\, c) \bullet a = b$$

Permutations

We shall extend 'swappings' to '(finite) lists of swappings'

$$(a_1\, b_1) \ldots (a_n\, b_n),$$

also called **permutations** (we shall often write π for them). Permutations are **bijective** mappings from atoms to atoms. For example

$$\pi = \begin{pmatrix} a \mapsto b \\ b \mapsto a \\ c \mapsto c \end{pmatrix} \qquad (c\, b)(a\, b)(a\, c) \bullet b = a$$

Permutations

We shall extend 'swappings' to '(finite) lists of swappings'

$$(a_1\, b_1) \ldots (a_n\, b_n),$$

also called **permutations** (we shall often write π for them). Permutations are **bijective** mappings from atoms to atoms. For example

$$\pi = \begin{pmatrix} a \mapsto b \\ b \mapsto a \\ c \mapsto c \end{pmatrix} \qquad (c\, b)(a\, b)(a\, c) \bullet c = c$$

Permutations

We shall ... lists of swapping ...

also call ... write π for the ... appings from ato...

Our list-representation is **not** unique, because

$$(c\,b)(a\,b)(a\,c) \text{ and } (a\,b)$$

are the 'same' permutation.

$$\pi = \begin{pmatrix} a \mapsto b \\ b \mapsto a \\ c \mapsto c \end{pmatrix} \qquad (c\,b)(a\,b)(a\,c) \bullet c = c$$

Permutations on Atoms

A permutation **acts** on an atom as follows:

$$[]\bullet a \stackrel{\text{def}}{=} a$$

$$((a_1\,a_2) :: \pi)\bullet a \stackrel{\text{def}}{=} \begin{cases} a_1 & \text{if } \pi\bullet a = a_2 \\ a_2 & \text{if } \pi\bullet a = a_1 \\ \pi\bullet a & \text{otherwise} \end{cases}$$

- $[]$ stands for the empty list (the identity permutation), and
- $(a_1\,a_2) :: \pi$ stands for the permutation π followed by the swapping $(a_1\,a_2)$

Permutations on Atoms (ct.)

- the **composition** of two permutations is given by list-concatenation, written as $\pi'@\pi$,
- the **inverse** of a permutation is given by list reversal, written as π^{-1}, and
- the **disagreement set** of two permutations π and π' is the set of atoms

$$ds(\pi, \pi') \stackrel{\text{def}}{=} \{a \mid \pi \bullet a \neq \pi' \bullet a\}$$

- $\pi \sim \pi' \stackrel{\text{def}}{=} ds(\pi, \pi') = \varnothing$.

Permutations on Atoms (ct.)

- the **composition** of two permutations is given by list-concatenation, written as $\pi'@\pi$,
- the **inverse** of a permutation is given by list reversal, written as π^{-1}, and
- the **disagreement set** of two permutations π and π' is the set of atoms

$$\pi = \begin{pmatrix} a \mapsto b \\ b \mapsto c \\ c \mapsto a \end{pmatrix} = (a\,c)(a\,b) \qquad \pi^{-1} = \begin{pmatrix} b \mapsto a \\ c \mapsto b \\ a \mapsto c \end{pmatrix} = (a\,b)(a\,c)$$

- π

Permutations on Atoms (ct.)

- the **composition** of two permutations is given by list-concatenation, written as $\pi'@\pi$,
- the **inverse** of a permutation is given by list reversal, written as π^{-1}, and
- the **disagreement set** of two permutations π and π' is the set of atoms

$$ds(\pi,\pi') \stackrel{\text{def}}{=} \{a \mid \pi\bullet a \neq \pi'\bullet a\}$$

- $\pi \sim \pi' \stackrel{\text{def}}{=} ds(\pi,\pi') = \varnothing$.

Permutations on Atoms (ct.)

- the **composition** of two permutations is given by list-concatenation, written as $\pi'@\pi$,
- the … t
  reve…

for (finite) permutations this set is always finite (namely a subset of the atoms occurring π and π')

- the **disagreement set** of two permutations π and π' is the set of atoms

$$ds(\pi,\pi') \stackrel{\text{def}}{=} \{a \mid \pi \bullet a \neq \pi' \bullet a\}$$

- $\pi \sim \pi' \stackrel{\text{def}}{=} ds(\pi,\pi') = \varnothing$.

Permutations on Atoms (ct.)

- the ... en by li...
- the ... t reve...
- the ... π and ...

Example $ds((a\,c)(a\,b), (a\,b))$?

$$\begin{pmatrix} a \mapsto b \\ b \mapsto c \\ c \mapsto a \end{pmatrix} \quad \begin{pmatrix} a \mapsto b \\ b \mapsto a \\ c \mapsto c \end{pmatrix}$$

$$= \{b, c\}$$

$$ds(\pi, \pi') \stackrel{\text{def}}{=} \{a \mid \pi \bullet a \neq \pi' \bullet a\}$$

- $\pi \sim \pi' \stackrel{\text{def}}{=} ds(\pi, \pi') = \varnothing$.

Permutations on Atoms (ct.)

- the **composition** of two permutations is given by list-concatenation, written as $\pi'@\pi$,
- the **inverse** of a permutation is given by list reversal, written as π^{-1}, and
- the **disagreement set** of two permutations π and π' is the set of atoms

$$ds(\pi, \pi') \stackrel{\text{def}}{=} \{a \mid \pi \bullet a \neq \pi' \bullet a\}$$

- $\pi \sim \pi' \stackrel{\text{def}}{=} ds(\pi, \pi') = \varnothing$.

Properties of Permutations

Here a, b and c are arbitrary atoms:

- $(b\,b)\bullet a = a$, $(b\,c)\bullet a = (c\,b)\bullet a$
- $\pi^{-1}\bullet(\pi\bullet a) = a$
- $\pi\bullet a = b$ if and only if $a = \pi^{-1}\bullet b$
- $\pi_1 @ \pi_2\bullet a = \pi_1\bullet(\pi_2\bullet a)$
- $\pi\bullet((b\,c)\bullet a) = (\pi\bullet b\;\;\pi\bullet c)\bullet(\pi\bullet a)$

the first, second and last fact can be generalised to

- if $\pi \sim \pi'$ then $\pi\bullet a = \pi'\bullet a$

Properties of Permutations

Her

the

to

Preview: in the future, permutations will be completely characterised by the properties:

- $[] \bullet x = x$
- $\pi_1 @ \pi_2 \bullet x = \pi_1 \bullet (\pi_2 \bullet x)$
- if $\pi \sim \pi'$ then $\pi \bullet x = \pi' \bullet x$

where x stands also for other 'things', not just atoms. Don't worry this will become clearer later on.

- if $\pi \sim \pi'$ then $\pi \bullet a = \pi' \bullet a$

Properties of Permutations

Here a, b and c are arbitrary atoms:

- $(b\,b)\bullet a = a$, $(b\,c)\bullet a = (c\,b)\bullet a$
- $\pi^{-1}\bullet(\pi\bullet a) = a$
- $\pi\bullet a = b$ if and only if $a = \pi^{-1}\bullet b$
- $\pi_1@\pi_2\bullet a = \pi_1\bullet(\pi_2\bullet a)$
- $\pi\bullet((b\,c)\bullet a) = (\pi\bullet b\ \ \pi\bullet c)\bullet(\pi\bullet a)$

the first, second and last fact can be generalised to

- if $\pi \sim \pi'$ then $\pi\bullet a = \pi'\bullet a$

Permutations on λ-Terms

$$\pi \bullet (a) \qquad \text{given by the action on atoms}$$

$$\pi \bullet (t_1\, t_2) \stackrel{\text{def}}{=} (\pi \bullet t_1)(\pi \bullet t_2)$$

$$\pi \bullet (\lambda a.t) \stackrel{\text{def}}{=} \lambda(\pi \bullet a).(\pi \bullet t)$$

We have:

- $\pi^{-1} \bullet (\pi \bullet t) = t$
- $t_1 = t_2$ if and only if $\pi \bullet t_1 = \pi \bullet t_2$
- $\pi \bullet t_1 = t_2$ if and only if $t_1 = \pi^{-1} \bullet t_2$

(The attentive listener might like to prove these properties. You never know what you are being told.)

Permutations on λ-Terms

$$\pi\bullet(a) \qquad \text{given by the action on atoms}$$

$$\pi\bullet(t_1\,t_2) \stackrel{\text{def}}{=} (\pi\bullet t_1)(\pi\bullet t_2)$$

$$\pi\bullet(\lambda a.t) \stackrel{\text{def}}{=} \lambda(\pi\bullet a).(\pi\bullet t)$$

'we treat lambdas as if there were no binders'

We have:

- $\pi^{-1}\bullet(\pi\bullet$
- $t_1 = t_2$ if and only if $\pi\bullet t_1 = \pi\bullet t_2$
- $\pi\bullet t_1 = t_2$ if and only if $t_1 = \pi^{-1}\bullet t_2$

(The attentive listener might like to prove these properties. You never know what you are being told.)

Permutations on λ-Terms

$$\begin{array}{rcl} \pi \bullet (a) & & \text{given by the action on atoms} \\ \pi \bullet (t_1\, t_2) & \stackrel{\text{def}}{=} & (\pi \bullet t_1)(\pi \bullet t_2) \\ \pi \bullet (\lambda a.t) & \stackrel{\text{def}}{=} & \lambda(\pi \bullet a).(\pi \bullet t) \end{array}$$

We have:

- $\pi^{-1} \bullet (\pi \bullet t) = t$
- $t_1 = t_2$ if and only if $\pi \bullet t_1 = \pi \bullet t_2$
- $\pi \bullet t_1 = t_2$ if and only if $t_1 = \pi^{-1} \bullet t_2$

(The attentive listener might like to prove these properties. You never know what you are being told.)

Permutations on λ-Terms

What is it about permutations? Well...

- they have much nicer properties than renaming-substitutions (stemming from the fact that they are bijections on atoms),
- they give rise to a very simple definition of α-equivalence (shown next)
- and more later on

- $\pi \bullet t_1 = t_2$ if and only if $t_1 = \pi^{-1} \bullet t_2$

(The attentive listener might like to prove these properties. You never know what you are being told.)

α-Equivalence

Consider the following four rules:

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

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

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

$$\frac{t \approx (a\, b)\bullet s \quad a \# s}{\lambda a.t \approx \lambda b.s}\approx\text{-lam}_2$$

assuming $a \neq b$

α-Equivalence

Consider the following four rules:

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

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

assuming $a \neq b$

$\lambda a.t \approx \lambda b.s$ iff t is α-equivalent with s in which all occurrences of b have been renamed to a. . . oops **permuted** to a.

α-Equivalence

But this alone leads to an 'unsound' rule! Consider

$$\lambda a.b \quad \text{and} \quad \lambda b.a$$

which are **not** α-equivalent. However, if we apply the permutation $(a\,b)$ to a we get

$$b \approx b$$

which leads to non-sense.

We need to ensure that there are **no** 'free' occurrences of a in s. This is achieved by freshness, written $a \;\#\; s$.

a. . . oops **permuted** to a.

α-Equivalence

Consider the following four rules:

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

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

assuming $a \neq b$

$\lambda a.t \approx \lambda b.s$ iff t is α-equivalent with s in which all occurrences of b have been renamed to a...oops **permuted** to a.

Freshness

$$\frac{}{a \# b}\text{ \#-atm} \qquad \frac{a \# t_1 \quad a \# t_2}{a \# t_1\, t_2}\text{ \#-app}$$

$$\frac{}{a \# \lambda a.t}\text{ \#-lam}_1 \qquad \frac{a \# t}{a \# \lambda b.t}\text{ \#-lam}_2$$

$\boxed{\text{assuming } a \neq b}$

Be careful, we have defined two relations over lambda-terms/syntax-trees. We have **not** defined what 'bound' or 'free' means. That is a feature, not a bug.™

$\approx$ is an Equivalence

You might be an agnostic and notice that

$$\frac{t \approx (a\,b)\bullet s \quad a \# s}{\lambda a.t \approx \lambda b.s}\ \approx\text{-lam}_2$$

is defined rather unsymmetrically. Still we have:

Theorem: $\approx$ is an equivalence relation.

(Reflexivity) $t \approx t$

(Symmetry) if $t_1 \approx t_2$ then $t_2 \approx t_1$

(Transitivity) if $t_1 \approx t_2$ and $t_2 \approx t_3$ then $t_1 \approx t_3$

$\Rightarrow$ is rather tricky to prove

$\approx$ is an Equivalence

You might be an agnostic and notice that

the crucial lemmas:

- $a \# t$ and $t \approx t'$ then $a \# t'$
- $t \approx t'$ then $\pi \bullet t \approx \pi \bullet t'$
- $a \# t$ then $\pi \bullet a \# \pi \bullet t$
- $(\forall a \in ds(\pi, \pi').\ a \# t)$ then $\pi \bullet t \approx \pi' \bullet t$

is d ... ave:

(Reflexivity)	$t \approx t$
(Symmetry)	if $t_1 \approx t_2$ then $t_2 \approx t_1$
(Transitivity)	if $t_1 \approx t_2$ and $t_2 \approx t_3$ then $t_1 \approx t_3$ $\Rightarrow$ is rather tricky to prove

Proof for Transitivity

Proof: Over the height of t_1.

Illustration of the proof for $\lambda a_1.s_1 \approx \lambda a_2.s_2$ and $\lambda a_2.s_2 \approx \lambda a_3.s_3$ implies $\lambda a_1.s_1 \approx \lambda a_3.s_3$ for a_1, a_2, a_3 being distinct.

From the assumptions:

(i)	$s_1 \approx (a_1\, a_2) \bullet s_2$	(iii)	$a_1 \# s_2$
(ii)	$s_2 \approx (a_2\, a_3) \bullet s_3$	(iv)	$a_2 \# s_3$

We need to prove:

$s_1 \approx (a_1\, a_3) \bullet s_3$ and $a_1 \# s_3$

Details

From $s_2 \approx (a_2\,a_3)\bullet s_3$

(a) $(a_1\,a_2)\bullet s_2 \approx (a_1\,a_2)(a_2\,a_3)\bullet s_3$

using Lemma: If $t_1 \approx t_2$ then $\pi\bullet t_1 \approx \pi\bullet t_2$

Details

From $s_2 \approx (a_2\, a_3) \bullet s_3$

(a) $(a_1\, a_2) \bullet s_2 \approx (a_1\, a_2)(a_2\, a_3) \bullet s_3$

Then by IH (all terms have smaller depth)

(b) $s_1 \approx (a_1\, a_2)(a_2\, a_3) \bullet s_3$

using assumption $s_1 \approx (a_1\, a_2) \bullet s_2$

Details

From $s_2 \approx (a_2\, a_3) \bullet s_3$

(a) $(a_1\, a_2) \bullet s_2 \approx (a_1\, a_2)(a_2\, a_3) \bullet s_3$

Then by IH (all terms have smaller depth)

(b) $s_1 \approx (a_1\, a_2)(a_2\, a_3) \bullet s_3$

We know that

(c) $ds((a_1\, a_2)(a_2\, a_3), (a_1\, a_3)) = \{a_1, a_2\}$

(d) $a_2 \# s_3$

using the assumption

Details

From $s_2 \approx (a_2\, a_3) \bullet s_3$

(a) $(a_1\, a_2) \bullet s_2 \approx (a_1\, a_2)(a_2\, a_3) \bullet s_3$

Then by IH (all terms have smaller depth)

(b) $s_1 \approx (a_1\, a_2)(a_2\, a_3) \bullet s_3$

We know that

(c) $ds((a_1\, a_2)(a_2\, a_3), (a_1\, a_3)) = \{a_1, a_2\}$

(d) $a_2 \# s_3$

From $a_1 \# s_2$ and $s_2 \approx (a_2\, a_3) \bullet s_3$

(e) $a_1 \# (a_2\, a_3) \bullet s_3$

using Lemma: If $a \# t_1$ and $t_1 \approx t_2$ then $a \# t_2$

Details

From $s_2 \approx (a_2\, a_3) \bullet s_3$

(a) $(a_1\, a_2) \bullet s_2 \approx (a_1\, a_2)(a_2\, a_3) \bullet s_3$

Then by IH (all terms have smaller depth)

(b) $s_1 \approx (a_1\, a_2)(a_2\, a_3) \bullet s_3$

We know that

(c) $ds((a_1\, a_2)(a_2\, a_3), (a_1\, a_3)) = \{a_1, a_2\}$

(d) $a_2 \ \# \ s_3$

From $a_1 \ \# \ s_2$ and $s_2 \approx (a_2\, a_3) \bullet s_3$

(e) $a_1 \ \# \ s_3$

using Lemma: If $a \ \# \ \pi \bullet t_1$ then $\pi^{-1} \bullet a \ \# \ t_1$

Details

From $s_2 \approx (a_2\, a_3) \bullet s_3$

(a) $(a_1\, a_2) \bullet s_2 \approx (a_1\, a_2)(a_2\, a_3) \bullet s_3$

Then by IH (all terms have smaller depth)

(b) $s_1 \approx (a_1\, a_2)(a_2\, a_3) \bullet s_3$

We know that

(c) $ds((a_1\, a_2)(a_2\, a_3), (a_1\, a_3)) = \{a_1, a_2\}$

(d) $a_2 \# s_3$

From $a_1 \# s_2$ and $s_2 \approx (a_2\, a_3) \bullet s_3$

(e) $a_1 \# s_3$

So

(f) $(a_1\, a_2)(a_2\, a_3) \bullet s_3 \approx (a_1\, a_3) \bullet s_3$

using Lem: If $\forall a \in ds(\pi, \pi').\, a \# t$ then $\pi \bullet t \approx \pi' \bullet t$

Details

From $s_2 \approx (a_2\, a_3) \bullet s_3$

(a) $(a_1\, a_2) \bullet s_2 \approx (a_1\, a_2)(a_2\, a_3) \bullet s_3$

Then by IH (all terms have smaller depth)

(b) $s_1 \approx (a_1\, a_2)(a_2\, a_3) \bullet s_3$

We know that

(c) $ds((a_1\, a_2)(a_2\, a_3), (a_1\, a_3)) = \{a_1, a_2\}$

(d) $a_2 \# s_3$

From $a_1 \# s_2$ and $s_2 \approx (a_2\, a_3) \bullet s_3$

(e) $a_1 \# s_3$

So

(f) $(a_1\, a_2)(a_2\, a_3) \bullet s_3 \approx (a_1\, a_3) \bullet s_3$

Finally by IH

$s_1 \approx (a_1\, a_3) \bullet s_3$ and $\lambda a_1.s_1 \approx \lambda a_3.s_3$

Comparison with $=_\alpha$

Traditionally $=_\alpha$ is defined as

> least congruence which identifies $a.t$ with $b.[a := b]t$ provided b is not free in t

where $[a := b]t$ replaces all free occurrences of a by b in t.

- with $\approx$ and $\#$ we never need to choose a 'fresh' atom (good for implementations and for nominal unification—2nd lecture)
- permutation respects both relations, whilst renaming-substitution does not

Does This Help?

...with our proof for the weakening property.
Let's first extend the permutation operation to:

- sets of lambda-terms

$$\pi \bullet \{t_1, \ldots, t_n\} \stackrel{\text{def}}{=} \{\pi \bullet t_1, \ldots, \pi \bullet t_n\}$$

- pairs

$$\pi \bullet (x, y) \stackrel{\text{def}}{=} (\pi \bullet x, \pi \bullet y)$$

- types $\tau ::= X \mid \tau \rightarrow \tau$

$$\pi \bullet \tau \stackrel{\text{def}}{=} \tau$$

Does This Help?

...with our proof for the weakening property.
Let's first extend the permutation operation to:

- sets of lambda-terms

$$\pi \bullet \{t_1, \ldots, t_n\} \stackrel{\text{def}}{=} \{\pi \bullet t_1, \ldots, \pi \bullet t_n\}$$

- pairs

you are probably by now not surprised that we have:

- $t \in X$ if and only if $(\pi \bullet t) \in (\pi \bullet X)$
- $\pi \bullet [t]_\alpha = [\pi \bullet t]_\alpha$

Does This Help?

...with our proof for the weakening property.
Let's first extend the permutation operation to:

- sets of lambda-terms

$$\pi \bullet \{t_1, \ldots, t_n\} \stackrel{\text{def}}{=} \{\pi \bullet t_1, \ldots, \pi \bullet t_n\}$$

- pairs

$$\pi \bullet (x, y) \stackrel{\text{def}}{=} (\pi \bullet x, \pi \bullet y)$$

- types $\tau ::= X \mid \tau \rightarrow \tau$

$$\pi \bullet \tau \stackrel{\text{def}}{=} \tau$$

Does This Help?

...with our proof for the weakening property.
Let's first extend the permutation operation to:

- se
  $\}$
- pa
- types

$$\pi \bullet \tau \stackrel{\text{def}}{=} \tau$$

So given a typing-context Γ

$$\pi \bullet \Gamma$$

will always be a typing-context, while

$$\Gamma[a := b]$$

is only in some specific circumstances.

Equivariance of $\approx$ and $\#$

A relation (or predicate) is called **equivariant** provided it is preserved under permutations, that is its validity is invariant under permutations. For example:

$$t_1 \approx t_2 \quad \text{if and only if} \quad \pi \bullet t_1 \approx \pi \bullet t_2$$

$$a \# t \quad \text{if and only if} \quad \pi \bullet a \# \pi \bullet t$$

It seems, equivariance is an important concept when reasoning about properties involving binders.

...Also $\vdash$ and φ

- the typing relation is equivariant

$$\Gamma \vdash [t]_\alpha : \tau \quad \Leftrightarrow \quad \pi\bullet\Gamma \vdash \pi\bullet[t]_\alpha : \pi\bullet\tau$$

$$\frac{a : \tau \in \Gamma}{\Gamma \vdash [a]_\alpha : \tau} \quad \Leftrightarrow \quad \frac{\pi\bullet(a : \tau) \in \pi\bullet\Gamma}{\pi\bullet\Gamma \vdash [\pi\bullet a]_\alpha : \pi\bullet\tau}$$

- our induction-hypothesis is equivariant,
  i.e. $\varphi(\Gamma;[t]_\alpha;\tau) \Leftrightarrow \varphi(\pi\bullet\Gamma;\pi\bullet[t]_\alpha;\pi\bullet\tau)$

$$(\forall\tau')(\forall a' \notin \mathsf{dom}(\Gamma))\ \Gamma, a' : \tau' \vdash [t]_\alpha : \tau$$
$$\Leftrightarrow$$
$$(\forall\tau')(\forall a' \notin \mathsf{dom}(\pi\bullet\Gamma))\ \pi\bullet\Gamma, a' : \tau' \vdash \pi\bullet[t]_\alpha : \pi\bullet\tau$$

. . . Also $\vdash$ and φ

- the typing relation is equivariant

Be careful! The $\forall$-quantifiers are not allowed to quantify anything in π—if they do, we have to rename the quantified **meta**-variables.

$]_\alpha : \pi \bullet \tau$

$\in \pi \bullet \Gamma$

$]_\alpha : \pi \bullet \tau$

- our induction-hypothesis is equivariant, i.e. $\varphi(\Gamma;[t]_\alpha;\tau) \Leftrightarrow \varphi(\pi \bullet \Gamma;\pi \bullet [t]_\alpha;\pi \bullet \tau)$

$$(\forall \tau')(\forall a' \notin \mathsf{dom}(\Gamma))\ \Gamma, a' : \tau' \vdash [t]_\alpha : \tau$$
$$\Leftrightarrow$$
$$(\forall \tau')(\forall a' \notin \mathsf{dom}(\pi \bullet \Gamma))\ \pi \bullet \Gamma, a' : \tau' \vdash \pi \bullet [t]_\alpha : \pi \bullet \tau$$

Now the Proof

Case $a' = a$: from the premise we know

1. $\varphi(\Gamma, a : \tau_1; [t]_\alpha; \tau_2)$ 2. $a \not\in \mathrm{dom}(\Gamma)$

Now the Proof

Case $a' = a$: from the premise we know

1. $\varphi(\Gamma, a : \tau_1; [t]_\alpha; \tau_2)$ 2. $a \notin \mathrm{dom}(\Gamma)$

By equivariance we know

1'. $\varphi(\Gamma, b : \tau_1; [(a\, b) \bullet t]_\alpha; \tau_2)$ 2'. $b \notin \mathrm{dom}(\Gamma)$

for any **fresh** atom b, i.e. one not occurring in Γ, t, or $\{a, a'\}$.

Now the Proof

Case $a' = a$: from the premise we know

1. $\varphi(\Gamma, a : \tau_1; [t]_\alpha; \tau_2)$ 2. $a \not\in \mathrm{dom}(\Gamma)$

By equivariance we know

1'. $\varphi(\Gamma, b : \tau_1; [(a\,b)\bullet t]_\alpha; \tau_2)$ 2'. $b \not\in \mathrm{dom}(\Gamma)$

for any **fresh** atom b, i.e. one not occurring in Γ, t, or $\{a, a'\}$.

This looks very much like we are closing our eyes again. But not quite! It very much depends on how easy it is to work with 'fresh'. Also, we do not need to explicitly give a b—its existence will be enough.

Now the Proof

Case $a' = a$: from the premise we know

1. $\varphi(\Gamma, a : \tau_1; [t]_\alpha; \tau_2)$ 2. $a \notin \mathrm{dom}(\Gamma)$

By equivariance we know

1'. $\varphi(\Gamma, b : \tau_1; [(a\,b)\bullet t]_\alpha; \tau_2)$ 2'. $b \notin \mathrm{dom}(\Gamma)$

for any **fresh** atom b, i.e. one not occurring in Γ, t, or $\{a, a'\}$.

By definition of φ we have $\forall a' \notin \mathrm{dom}(\Gamma, b : \tau_1)$

3. $\Gamma, b : \tau_1, a' : \tau' \vdash [(a\,b)\bullet t]_\alpha : \tau_2$

Now the Proof

Case $a' = a$: from the premise we know

1. $\varphi(\Gamma, a : \tau_1; [t]_\alpha; \tau_2)$ 2. $a \not\in \mathsf{dom}(\Gamma)$

By equivariance we know

1'. $\varphi(\Gamma, b : \tau_1; [(a\,b)\bullet t]_\alpha; \tau_2)$ 2'. $b \not\in \mathsf{dom}(\Gamma)$

for any **fresh** atom b, i.e. one not occurring in Γ, t, or $\{a, a'\}$.

By definition of φ we have $\forall a' \not\in \mathsf{dom}(\Gamma, b : \tau_1)$

3. $\Gamma, b : \tau_1, a' : \tau' \vdash [(a\,b)\bullet t]_\alpha : \tau_2$

By choice of b we can now apply the typing-rule and get

4. $\Gamma, a' : \tau' \vdash [\lambda b.(a\,b)\bullet t]_\alpha : \tau_1 \to \tau_2$

Now the Proof

Case $a' = a$: from the premise we know

1. $\varphi(\Gamma,$

But now

$$\lambda b.(a\,b)\bullet t \approx \lambda a.t$$

so we have

$$[\lambda b.(a\,b)\bullet t]_\alpha = [\lambda a.t]_\alpha$$

and **finally** we know that

$$\Gamma, a' : \tau' \vdash [\lambda a.t]_\alpha : \tau_1 \to \tau_2$$

holds in the case $a' = a$. Done. :o)

By equiv

1'. $\varphi(\Gamma$ $)$

for any f t, or $\{a, a'\}$.

By defini $)$

3. $\Gamma, b : \tau_1, a' : \tau' \vdash [(a\,b)\bullet t]_\alpha : \tau_2$

By choice of b we can now apply the typing-rule and get

4. $\Gamma, a' : \tau' \vdash [\lambda b.(a\,b)\bullet t]_\alpha : \tau_1 \to \tau_2$

Outlook

- next week we shall see that permutations are also helpful with unification – i.e. we shall solve the quiz
- after that we shall start with a full framework for reasoning with binders and α-equated syntax
- homework ;o) have a look at the transitivity proof