Assuming that $a$ and $b$ are distinct variables, is it possible to find $\lambda$-terms $M_1$..$M_7$ that make the following pairs $\alpha$-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: Quiz

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

\[
\begin{align*}
\lambda a. \lambda b. (b \ M_6) & \quad \text{and} \quad \lambda a. \lambda a. (a \ M_7) \\
\end{align*}
\]

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 on Friday. ;o)

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

every day this week from 11:00 to 12:30 in Room C2

Christian Urban

University of Cambridge
What this course will be about

- syntax with binders (e.g. lambda-calculus)
- how to reason **formally** about binders
- how to use structural induction and structural recursion **conveniently**

- no de-Bruijn indices, no hand-waving using a Barendregt-style naming convention...

- a surprisingly **fresh** look at something quite familiar (unless you have already read the papers by Pitts, of course)
Relevance to Some Other Courses?

Two examples:

- **Morrill**: Type logical grammar (lambda-calculus)
- **Koller et al.**: Computational semantics (accidental bindings, also gives an implementation of the lambda-calculus)
- probably others
Relevance to Some Other Courses?

Two examples:

- **Morrill**: Type logical grammar (lambda-calculus)
- **Koller et al.**: Computational semantics (accidental bindings, also gives an implementation of the lambda-calculus)

...
Relevance to Some Other Courses?

Two examples:

- **Morrill**: Type logical grammar (lambda-calculus)
- **Koller et al.**: Computational semantics (accidental bindings, also gives an implementation of the lambda-calculus)
- probably others
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

\[
\int_0^1 x^2 + x \, dx = x + \frac{1}{3}
\]
Computer-scientist version:

\[ \lambda a. (b\ a)[b := a] \overset{\text{naïvely}}{\longrightarrow} \lambda a. (a\ a) \]

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

\[ \lambda a.((\lambda b. b\ c)(\lambda c. a\ c)) \]
Another Problem

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

Assume we define the set $\Lambda$ of (raw) lambda-terms inductively by the grammar:

\[
\begin{align*}
    t & ::= a & \text{variables} \\
    & | tt & \text{applications} \\
    & | \lambda a.t & \text{abstractions}
\end{align*}
\]
Another Problem
(If you know it, you probably choose to ignore it.)

Assume we define the set $\Lambda$ of (raw) lambda-terms *inductively* by the grammar:

$$
t ::= a \quad \text{variables} \\
     \mid tt \quad \text{applications} \\
     \mid \lambda a.t \quad \text{abstractions}
$$

We can easily define functions over $\Lambda$ by structural recursion; for example

- $\text{depth}(a) \overset{\text{def}}{=} 0$
- $\text{depth}(tt') \overset{\text{def}}{=} 1 + \max(\text{depth}(t), \text{depth}(t'))$
- $\text{depth}(\lambda a.t) \overset{\text{def}}{=} 1 + \text{depth}(t)$
Another Problem
(If you know it, you probably choose to ignore it.)

Assume we define the set $\Lambda$ of (raw) lambda-terms inductively by the grammar:

$$
t ::= a \quad \text{variables}
\mid tt \quad \text{applications}
\mid \lambda a.t \quad \text{abstractions}
$$

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

$$(a) [b := s] \overset{\text{def}}{=} \begin{cases} \text{if } a = b \text{ then } s \text{ else } a & \text{if } \alpha \end{cases}$$

$$(tt') [b := s] \overset{\text{def}}{=} (t[b := s]) (t'[b := s])$$

$$(\lambda a.t) [b := s] \overset{\text{def}}{=} \lambda a. (t[b := s]) \quad \text{plus conditions}$$
Another Problem
(If you know it, you probably choose to ignore it.)

Assume we define the set $\Lambda$ of (raw) lambda-terms inductively by the grammar:

$$t \ ::= \ a \quad \text{variables} \quad \begin{array}{c}\text{j} \quad \text{applications} \quad \begin{array}{c}\text{j} \quad a : t \quad \text{abstractions} \quad \begin{array}{c}\text{j} \quad \text{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?

$$
\begin{align*}
(a) [b := s] & \overset{\text{def}}{=} \text{if } a = b \text{ then } s \text{ else } a \\
(t \ t') [b := s] & \overset{\text{def}}{=} (t[b := s]) (t'[b := s]) \\
(\lambda a. t) [b := s] & \overset{\text{def}}{=} \lambda a. (t[b := s]) \text{ plus conditions}
\end{align*}
$$
Another Problem
(If you know it, you probably choose to ignore it.)

Assume we define the set $\Lambda$ of (raw) lambda-terms inductively by the grammar:

$$t ::= a \quad \text{variables}$$
$$| \quad t \; t \quad \text{applications}$$
$$| \quad \lambda a.t \quad \text{abstractions}$$

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

$$\begin{align*}
(a) \left[ b := s \right] & \overset{\text{def}}{=} \text{if } a = b \text{ then } s \text{ else } a \\
(t \; t') \left[ b := s \right] & \overset{\text{def}}{=} (t[b := s])(t'[b := s]) \\
(\lambda a.t) \left[ b := s \right] & \overset{\text{def}}{=} \lambda a.(t[b := s]) \quad \text{plus conditions}
\end{align*}$$
Another Problem
(If you know it, you probably choose to ignore it.)

Assume we define the set of (raw) lambda-terms inductively by the grammar:

\[ t ::= a \]

variables

\[ \lambda a : t \]

abstractions

\[ t t' \]

applications

However, if we form the quotient-set =

\[ a = b \]

then what is the structural recursion principle?

(\[ a \] \[ b := s \]) \[ def = \]

if \( a = b \) then \( s \) else \( a \)

\[ (t t') \[ b := s \] \[ def = \] \]

\( t[b := s] \) \( (t'[b := s]) \)

\[ (\lambda a . t) \[ b := s \] \[ def = \] \]

\( \lambda a . (t[b := s]) \) plus conditions

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

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

Nancy, 16. August 2004 – p.6 (6/6)
Of course, of course — all these problems would go away, if we had used de-Bruijn indices to encode bindings. Like

\[
\lambda a. \lambda b. (a \ b \ c) \quad \leftrightarrow \quad \lambda \lambda (1 \ 0 \ 2) \\
\lambda a. \lambda b. (a \ (\lambda c. c \ a) \ b) \quad \leftrightarrow \quad \lambda \lambda (1 \ (\lambda (0 \ 2)) \ 0)
\]

But it just is a fact of life that de-Bruijn indices are hard to read and some important definitions are too far ‘away’ from their named counter-parts (see reader, page 3, for a definition of substitution with de-Bruijn indices). So we should attempt to do better.
Of course, of course — all these problems would go away, if we had used de-Bruijn indices to encode bindings. Like
\[
(a:b: ((a b c)))
\]
But it just is a fact of life that de-Bruijn indices are hard to read and some important definitions are too far ‘away’ from their named
counter-parts (see reader, page 3, for a definition of substitution with de-Bruijn indices). So we should attempt to do better.

Aside: We insist on names. In case you were wondering what ‘nominal’ stands for... Well, that we insist on names.
Of course, of course — all these problems would go away, if we had used de-Bruijn indices to encode bindings. Like

$$\lambda a. \lambda b. (a\ b\ c) \quad \mapsto \quad \lambda\lambda(1\ 0\ 2)$$
$$\lambda a. \lambda b. (a\ (\lambda c.\ c\ a)\ b) \quad \mapsto \quad \lambda\lambda(1\ (\lambda(0\ 2))\ 0)$$

But it just is a fact of life that de-Bruijn indices are hard to read and some important definitions are too far ‘away’ from their named counter-parts (see reader, page 3, for a definition of substitution with de-Bruijn indices). So we should attempt to do better.
Of course, of course — all these problems would go away, if we had used de-Bruijn indices to encode bindings. Like $a:b:(a \, b \, c)$.

But it just is a fact of life that de-Bruijn indices are hard to read and some important definitions are too far 'away' from their named counterparts (see reader, page 3, for a definition of substitution with de-Bruijn indices). So we should attempt to do better.

There is a great deal of other work (e.g. HOAS) which alleviate some of these problems (no time to be more specific about them in this course*).

However, none of them has made life cosy and none of them has reached universal acceptance for formal reasoning with binders.

*HOAS would, for example, deserve its own course.
Plan for the Course

Tentative:

- **Today**: further motivation and some ‘exercises’ to become familiar with some of the main nominal concepts (e.g. definition of $\alpha$-equivalence)

- **Tuesday**: Nominal Logic—a showcase for the nominal techniques

- **Wednesday + Thursday**: Justification for the nominal techniques (a bit mathematical)

- **Friday**: a nice application of the nominal techniques—unification of terms with binders
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
...but sometimes eyes just cannot be closed :o(

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

\[
\frac{a : \tau \in \Gamma}{\Gamma \vdash a : \tau} \quad \frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 \ t_2 : \tau_2} \quad \frac{\Gamma, a : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda a. \ t : \tau_1 \rightarrow \tau_2} \quad a \not\in \text{dom}(\Gamma)
\]
Weakening Property

...but sometimes eyes just cannot be closed :o(

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

\[
\frac{a : \tau \in \Gamma}{\Gamma \vdash a : \tau}
\]

\[
\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 t_2 : \tau_2}
\]

\[
\frac{\Gamma, a : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda a.t : \tau_1 \to \tau_2}
\quad a \not\in \text{dom}(\Gamma)
\]
Weakening Property

... but sometimes eyes just cannot be closed :o(

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

\[
\frac{\alpha : \tau \in \Gamma}{\Gamma \vdash \alpha : \tau}
\]

\[
\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 \, t_2 : \tau_2}
\]

\[
\frac{\Gamma, \alpha : \tau_1 \vdash t : \tau_2 \quad \alpha \not\in \text{dom} (\Gamma)}{\Gamma \vdash \lambda a . t : \tau_1 \to \tau_2}
\]

If \( \Gamma \vdash t : \tau \), then also \( \Gamma, \alpha : \tau' \vdash t : \tau \).
Weakening Property

...but sometimes eyes just cannot be closed :o(

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

\[
\frac{a : \tau \in \Gamma}{\Gamma \vdash a : \tau}
\]

\[
\frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 \, t_2 : \tau_2}
\]

\[
\frac{\Gamma, a : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda a.t : \tau_1 \rightarrow \tau_2 \quad a \not\in \text{dom}(\Gamma)}
\]

\[
(\forall \Gamma)(\forall t)(\forall \tau) \quad \Gamma \vdash t : \tau \Rightarrow \\
(\forall \tau')(\forall a \not\in \text{dom}(\Gamma)) \quad \Gamma, a : \tau' \vdash t : \tau
\]

Raw Lambda-Terms? No!

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

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

is derivable, but

\[
\frac{\Gamma, a : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda a. t : \tau_1 \to \tau_2}
\]

is not, because

\[
\frac{\Gamma \vdash \lambda a. t : \tau_1 \to \tau_2}{a \not\in \text{dom}(\Gamma)}
\]
This property does not hold for raw lambda-terms: since

\[ a : \tau \vdash a : \tau \]

\[ \emptyset \vdash \lambda a. a : \tau \rightarrow \tau \]

is derivable, but

\[ a : \tau' \vdash \lambda a. a : \tau \rightarrow \tau \]

is not, because

\[ \Gamma, a : \tau_1 \vdash t : \tau_2 \]

\[ \Gamma \vdash \lambda a.t : \tau_1 \rightarrow \tau_2 \quad a \not\in \text{dom}(\Gamma) \]
Let’s Make This Explicit

Nobody usually bothers, but let’s explicitly write \([t]_\alpha\) for the set of (raw) lambda-terms \(\alpha\)-equivalent with \(t\):

\[
[t]_\alpha \overset{\text{def}}{=} \{ t' \mid t' =_\alpha t \} .
\]

Typing-rules for \(\alpha\)-equated lambda-terms:

\[
\begin{align*}
\frac{a : \tau \in \Gamma}{\Gamma \vdash [a]_\alpha : \tau} \\
\frac{\Gamma \vdash [t_1]_\alpha : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash [t_2]_\alpha : \tau_1}{\Gamma \vdash [t_1 \ t_2]_\alpha : \tau_2} \\
\frac{\Gamma, a : \tau_1 \vdash [t]_\alpha : \tau_2 \quad a \not\in \text{dom}(\Gamma)}{\Gamma \vdash [\lambda a. t]_\alpha : \tau_1 \rightarrow \tau_2}
\end{align*}
\]
Let’s Make This Explicit

Nobody usually bothers, but let’s explicitly write \([t]_\alpha\) for the set of (raw) lambda-terms \(-\)equivalent with \(t\):

\[
[t] = \{t_0\mid t_0 = t\}.
\]

Typing-rules for \(\alpha\)-equated lambda-terms:

\[
\frac{a : \tau \in \Gamma}{\Gamma \vdash [a]_\alpha : \tau}
\]

\[
\Gamma \vdash [t_1]_\alpha : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash [t_2]_\alpha : \tau_1
\]

\[
\Gamma \vdash [t_1 \ t_2]_\alpha : \tau_2
\]

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

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

\[
a \not\in \text{dom}(\Gamma)
\]

Remember, we write \([t_1]_\alpha\), but we mean a set of terms \(\{t_1, \ldots\}\) — namely the \(\alpha\)-equivalence class of \(t_1\).
Attemping the Proof

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

$$(\forall \tau')(\forall a' \not\in \text{dom}(\Gamma)) \quad \Gamma, a' : \tau' \vdash [t]_\alpha : \tau$$

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

$$\frac{\Gamma, a : \tau_1 \vdash [t]_\alpha : \tau_2}{\Gamma \vdash [\lambda a.t]_\alpha : \tau_1 \rightarrow \tau_2} \quad a \not\in \text{dom}(\Gamma)$$
We proceed by rule induction and try to show that the predicate

\[(\forall \tau')(\forall a' \not\in \text{dom}(\Gamma)) \varphi(\Gamma, a': \tau'; [\lambda a.t]_\alpha; \tau_2)\]

is closed under the axiom and the two inference rules. Interestingly:

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

We have to prove:

\[\varphi(\Gamma, a' : \tau'; [\lambda a.t]_\alpha; \tau_2)\]

for all \(\tau'\) and \(a' \not\in \text{dom}(\Gamma)\).

\[\frac{\Gamma, a : \tau_1 \vdash [t]_\alpha : \tau_2}{\Gamma \vdash [\lambda a.t]_\alpha : \tau_1 \rightarrow \tau_2} a \not\in \text{dom}(\Gamma)\]
We proceed by rule induction and try to show that the predicate \( \varphi(\Gamma, a : \tau_1; [t]_\alpha; \tau_2) \) given by

\[
\forall \tau' \forall a' \not\in \text{dom}(\Gamma) \quad (\forall \tau')(\forall a' \not\in \text{dom}(\Gamma))
\]

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

We know (for the premise):

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

We have to prove:

\( \varphi(\Gamma, a' : \tau'; [\lambda a.t]_\alpha; \tau_2) \)

for all \( \tau' \) and \( a' \not\in \text{dom}(\Gamma) \).

\[
\frac{\Gamma, a : \tau_1 \vdash [t]_\alpha : \tau_2}{\Gamma \vdash [\lambda a.t]_\alpha : \tau_1 \rightarrow \tau_2} \quad a \not\in \text{dom}(\Gamma)
\]

But this fails for \( a' = a \)!
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 \( \alpha \)-equivalent classes and their typing.

And, closing your eyes is a non-starter.
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 Wednesday
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 and bindable is an atom (independent from the language at hand).

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
Recall the problem: substitution does not respect $\alpha$-equivalence, e.g.

$$\lambda a. b \quad \lambda c. b$$
Recall the problem: substitution does not respect $\alpha$-equivalence, e.g.

\[
\begin{align*}
[b := a] \lambda a.b &= \lambda a.a \\
[b := a] \lambda c.b &= \lambda c.a
\end{align*}
\]
Recall the problem: substitution does not respect $\alpha$-equivalence, e.g.

$$[b := a] \lambda a. b = \lambda a. a$$
$$[b := a] \lambda c. b = \lambda c. a$$

Traditional Solution: replace $[b := a] t$ by a more complicated, 'capture-avoiding' form of substitution.
Recall the problem: substitution does not respect $\alpha$-equivalence, e.g.

$$
(b\ a) \cdot \lambda a.b \quad (b\ a) \cdot \lambda c.b
$$

$$
= \lambda b.a \quad = \lambda c.a
$$

**Nice Alternative:** use a less complicated operation for renaming

$$(b\ a) \cdot t \overset{\text{def}}{=} \text{swap all occurrences of } b \text{ and } a \text{ in } t$$
Swappings

Recall the problem: substitution does not respect $\alpha$-equivalence, e.g.

\[(b \ a) \cdot \lambda a. b \quad (b \ a) \cdot \lambda c. b\]

\[= \lambda b. a \quad = \lambda c. a\]

Nice Alternative: use a less complicated operation for renaming

\[(b \ a) \cdot t \overset{\text{def}}{=} \text{swap all occurrences of } b \text{ and } a \text{ in } t\]

be they bound, binding or bindable
Recall the problem: substitution does not respect $\alpha$-equivalence, e.g.

$$
(b\ a) \cdot \lambda a. b \quad (b\ a) \cdot \lambda c. b
$$

$$
= \lambda b. a \quad = \lambda c. a
$$

**Nice Alternative:** use a less complicated operation for renaming

$$(b\ a) \cdot t \overset{\text{def}}{=} \text{swap all occurrences of } b \text{ and } a \text{ in } t$$

Unlike for $[b := a](-)$, for $(b\ a) \cdot (-)$ we do have if $t =_{\alpha} t'$ then $(b\ a) \cdot t =_{\alpha} (b\ a) \cdot t'$.
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 \(\pi\) for them). Permutations are bijective mappings from atoms to atoms. For example

\[
\pi = \begin{pmatrix}
  a & \leftrightarrow & b \\
  b & \leftrightarrow & a \\
  c & \leftrightarrow & c
\end{pmatrix} = (c b)(a b)(a c)\
\]
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 \(\pi\) for them). Permutations are bijective mappings from atoms to atoms. For example

\[
\pi = \begin{pmatrix}
  a & \leftrightarrow & b \\
  b & \leftrightarrow & a \\
  c & \leftrightarrow & c
\end{pmatrix}
\]

\((c \ b)(a \ b)(a \ c) \cdot a = b\)
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 \(\pi\) for them). Permutations are bijective mappings from atoms to atoms. For example

\[\pi = \begin{pmatrix} a & \mapsto & b \\ b & \mapsto & a \\ c & \mapsto & c \end{pmatrix} \quad (c b)(a b)(a c) \cdot b = a\]
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 $\pi$ for them). Permutations are bijective mappings from atoms to atoms. For example

\[
\pi = \begin{pmatrix} 
  a & \mapsto & b \\
  b & \mapsto & a \\
  c & \mapsto & c \\
\end{pmatrix} \quad (c \; b) \; (a \; b) \; (a \; c) \; \cdot \; c = c
\]
Permutations

We shall extend ‘swappings’ to ‘(finite) lists of swappings’
\[ (a_1 b_1) (a_2 b_2) \cdots (a_n b_n) \];
also called permutations (we shall often write \( \pi \) for them). Permutations are bijective mappings from atoms to atoms. For example

\[ \pi = \begin{pmatrix}
    a & \leftrightarrow & b \\
    b & \leftrightarrow & a \\
    c & \leftrightarrow & c
\end{pmatrix} \]

\( \pi = (c b) (a b) (a c) \cdot c = c \)

Our list-representation is not unique, because
\( (c b) (a b) (a c) \) and \( (a b) \)
are the ‘same’ permutation.
A permutation acts on an atom as follows:

\[
[] \cdot a \ \overset{\text{def}}{=} \ a
\]

\[
((a_1 \ a_2) \ :: \ \pi) \cdot a \ \overset{\text{def}}{=} \begin{cases} 
a_1 & \text{if } \pi \cdot a = a_2 \\
a_2 & \text{if } \pi \cdot a = a_1 \\
\pi \cdot a & \text{otherwise}
\end{cases}
\]

[] stands for the empty list (the identity permutation), and

\[(a_1 \ a_2) :: \pi\] stands for the permutation \(\pi\) followed by the swapping \((a_1 \ a_2)\).
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 $\pi^{-1}$, and

the disagreement set of two permutations $\pi$ and $\pi'$ is the set of atoms

$$ds(\pi, \pi') \overset{\text{def}}{=} \{ a \mid \pi \cdot a \neq \pi' \cdot a \}$$
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 $\pi^{-1}$, and

the permutation

$$
\pi = \begin{pmatrix}
a & \mapsto b \\
b & \mapsto c \\
c & \mapsto a
\end{pmatrix}
$$

$$
\pi^{-1} = \begin{pmatrix}
b & \mapsto a \\
c & \mapsto b \\
a & \mapsto c
\end{pmatrix}
$$

is

$$
= (a \ c)(a \ b)
$$

$$
= (a \ b)(a \ c)
$$
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 $\pi^{-1}$, and

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

$$ds(\pi, \pi') \overset{\text{def}}{=} \{ a \mid \pi \cdot a \neq \pi' \cdot a \}$$
the composition of two permutations is given by list-concatenation, written as $\pi \odot \pi'$, the inverse of a permutation is given by list reversal, written as $\pi^{-1}$, and the disagreement set of two permutations $\pi$ and $\pi'$ is the set of atoms $a$ such that $\pi \cdot a \neq \pi' \cdot a$.

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

\[
\begin{pmatrix}
a & \leftrightarrow & b \\
b & \leftrightarrow & c \\
c & \leftrightarrow & a
\end{pmatrix}
\begin{pmatrix}
a & \leftrightarrow & b \\
b & \leftrightarrow & a \\
c & \leftrightarrow & c
\end{pmatrix}
= \{b, c\}
Properties of Permutations

Here \( a, b \) and \( c \) are arbitrary atoms:

- \((b \, b) \cdot a = a, \, (b \, c) \cdot a = (c \, b) \cdot a\)
- \(\pi^{-1} \cdot (\pi \cdot a) = a\)
- \(\pi \cdot a = b \iff a = \pi^{-1} \cdot b\)
- \(\pi_1 \circ \pi_2 \cdot a = \pi_1 \cdot (\pi_2 \cdot a)\)
- \(\pi \cdot ((b \, c) \cdot a) = (\pi \cdot b \, \pi \cdot c) \cdot (\pi \cdot a)\)

The first, second and last fact can be generalised to:

- if \( ds(\pi, \pi') = \emptyset \) then \( \pi \cdot a = \pi' \cdot a\)
Here \( a, b \) and \( c \) are arbitrary atoms:

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

1. \([\ ] \cdot x = x\)
2. \(\pi_1 \circ \pi_2 \cdot x = \pi_1 \cdot (\pi_2 \cdot x)\)
3. If \( ds(\pi, \pi') = \emptyset \) then \( \pi \cdot x = \pi' \cdot x \)

where \( x \) stands also for other ‘things’, not just atoms. Don’t worry this will become clearer later on.

4. If \( ds(\pi, \pi') = \emptyset \) then \( \pi \cdot a = \pi' \cdot a \)
Properties of Permutations

Here $a$, $b$ and $c$ are arbitrary atoms:

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

The first, second and last fact can be generalised to

- if $ds(\pi, \pi') = \emptyset$ then $\pi \cdot a = \pi' \cdot a$
Permutations on $\lambda$-Terms

$$\pi \ast (\alpha) \quad \text{given by the action on atoms}$$

$$\pi \ast (t_1 t_2) \overset{\text{def}}{=} (\pi \ast t_1)(\pi \ast t_2)$$

$$\pi \ast (\lambda \alpha. t) \overset{\text{def}}{=} \lambda(\pi \ast \alpha).(\pi \ast t_2)$$

We have:

1. $\pi^{-1} \ast (\pi \ast t) = t$
2. $t_1 = t_2$ if and only if $\pi \ast t_1 = \pi \ast t_2$
3. $\pi \ast t_1 = t_2$ if and only if $t_1 = \pi^{-1} \ast t_2$

(The attentive listener might like to prove these properties. You never know what you are being told.)
Permutations on $\lambda$-Terms

$\pi \cdot (a)$ given by the action on atoms

$\pi \cdot (t_1 t_2) \overset{\text{def}}{=} (\pi \cdot t_1)(\pi \cdot t_2)$

$\pi \cdot (\lambda a.t) \overset{\text{def}}{=} \lambda(\pi \cdot a).(\pi \cdot t_2)$

We have:

1. $\pi^{-1} \cdot (\pi \cdot t_1) = \pi \cdot t_2$

2. $t_1 = t_2$ if and only if $\pi \cdot t_1 = \pi \cdot t_2$

3. $\pi \cdot t_1 = t_2$ if and only if $t_1 = \pi^{-1} \cdot t_2$

(The attentive listener might like to prove these properties. You never know what you are being told.)
Permutations on $\lambda$-Terms

\[ \pi \cdot (a) \quad \text{given by the action on atoms} \]
\[ \pi \cdot (t_1 \cdot t_2) \quad \overset{\text{def}}{=} \quad (\pi \cdot t_1)(\pi \cdot t_2) \]
\[ \pi \cdot (\lambda a \cdot t) \quad \overset{\text{def}}{=} \quad \lambda (\pi \cdot a). (\pi \cdot t_2) \]

We have:

\[ \pi^{-1} \cdot (\pi \cdot t) = t \]
\[ t_1 = t_2 \quad \text{if and only if} \quad \pi \cdot t_1 = \pi \cdot t_2 \]
\[ \pi \cdot t_1 = t_2 \quad \text{if and only if} \quad t_1 = \pi^{-1} \cdot t_2 \]

(The attentive listener might like to prove these properties. You never know what you are being told.)
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 $\alpha$-equivalence (shown next)
- and don’t get me started ;o)

(The attentive listener might like to prove these properties. You never know what you are being told.)
\[ \alpha\text{-Equivalence} \]

Consider the following four rules:

- **\( a \approx a \approx_{\text{-atm}} \)**

- **\( t_1 \approx s_1 \quad t_2 \approx s_2 \approx_{\text{-app}} \)**

- **\( t_1 t_2 \approx s_1 s_2 \)**

- **\( t \approx (a\,b) \cdot s \quad a \# s \approx_{\text{-lam}} \)**

\[ \lambda a.t \approx \lambda a.s \approx_{\text{-lam}_1} \]

\[ \lambda a.t \approx \lambda b.s \approx_{\text{-lam}_2} \]

**assuming** \( a \neq b \)**
\(\alpha\)-Equivalence

Consider the following four rules:

\[
\begin{align*}
    a & \approx a \quad \approx\text{-atm} \\
    t & \approx s \quad \lambda a. t \approx \lambda a. s \quad \approx\text{-lam}_1 \\
    t_1 \approx s_1 & \quad t_2 \approx s_2 \quad t_1 t_2 \approx s_1 s_2 \approx\text{-app} \]
\]

\[
\begin{align*}
    t & \approx (ab) \cdot s \quad a \# s \quad \lambda a. t \approx \lambda b. s \quad \approx\text{-lam}_2 \\
    \text{assuming } a \neq b
\end{align*}
\]

\[\lambda a. t \approx \lambda b. s\] iff \(t\) is \(\alpha\)-equivalent with \(s\) in which all occurrences of \(b\) have been renamed to \(a\)...
oopsort{permuted}
oopsort{permuted} permuted to \(a\).
\(\alpha\)-Equivalence

But this alone leads to an ‘unsound’ rule! Consider*

\[ \lambda a.b \quad \text{and} \quad \lambda b.a \]

which are not \(\alpha\)-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 \(a \not\# s\).

*there is a typo in the reader where this example is given
\(\alpha\)-Equivalence

Consider the following four rules:

\[ a \approx a \approx_{\text{atm}} \]

\[ t \approx s \approx_{\text{lam}_1} \lambda a.t \approx \lambda a.s \]

\[ t_1 \approx s_1 \quad t_2 \approx s_2 \approx_{\text{app}} \]

\[ t_1 t_2 \approx s_1 s_2 \]

\[ t \approx (a b) \cdot s \quad a \neq s \approx_{\text{lam}_2} \]

\[ \lambda a.t \approx \lambda b.s \]

\(\lambda a.t \approx \lambda b.s \) iff \( t \) is \(\alpha\)-equivalent with \( s \) in which all occurrences of \( b \) have been renamed to \( a \)...oops permuted to \( a \).
Freshness

Be careful, we have defined two relations over raw lambda-terms. 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

\[
\begin{align*}
  t \approx (a \ b) \cdot s & \quad a \neq s \\
  \lambda a. t & \approx \lambda b. s \\
  \hline
  \text{Nancy, 16. August 2004 – p.24 (1/2)}
\end{align*}
\]

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 \( t_1 \approx t_3 \)
\( \approx \) is an Equivalence

You might be an agnostic and notice that because \( \approx \) and \( \# \) have very good properties:

- \( t \approx t' \) then \( \pi \cdot t \approx \pi \cdot t' \)
- \( a \# t \) then \( \pi \cdot a \# \pi \cdot t \)
- \( t \approx \pi \cdot t' \) then \( (\pi^{-1}) \cdot t \approx t' \)
- \( a \# \pi \cdot t \) then \( (\pi^{-1}) \cdot a \# t \)
- \( a \# t \) and \( t \approx t' \) then \( a \# t' \)

(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 \( t_1 \approx t_3 \)
Comparison with $\equiv_\alpha$

Traditionally $\equiv_\alpha$ 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—wait until Friday).
- permutation respects both relations, whilst renaming-substitution does not.
...with our proof for the weakening property. Let's first extend the permutation operation to:

- **sets of lambda-terms**
  \[ \pi \cdot \{t_1, \ldots, t_n\} \overset{\text{def}}{=} \{\pi \cdot t_1, \ldots, \pi \cdot t_n\} \]

- **pairs**
  \[ \pi \cdot (x, y) \overset{\text{def}}{=} (\pi \cdot x, \pi \cdot y) \]

- **types**
  \[ \tau ::= X \mid \tau \rightarrow \tau \]
  \[ \pi \cdot \tau \overset{\text{def}}{=} \tau \]
...with our proof for the weakening property. Let's first extend the permutation operation to:

\[ \pi \cdot \{t_1, \ldots, t_n\} \overset{\text{def}}{=} \{\pi \cdot t_1, \ldots, \pi \cdot t_n\} \]

you are probably by now not surprised that we have:

\[ t \in X \text{ if and only if } (\pi \cdot t) \in (\pi \cdot X) \]

\[ \pi \cdot [t]_\alpha = [\pi \cdot t]_\alpha \]
...with our proof for the weakening property. Let's first extend the permutation operation to:

- **Sets of lambda-terms**
  \[
  \pi \cdot \{t_1, \ldots, t_n\} \overset{\text{def}}{=} \{\pi \cdot t_1, \ldots, \pi \cdot t_n\}
  \]

- **Pairs**
  \[
  \pi \cdot (x, y) \overset{\text{def}}{=} (\pi \cdot x, \pi \cdot y)
  \]

- **Types**
  \[
  \tau ::= X \mid \tau \rightarrow \tau
  \]
  \[
  \pi \cdot \tau \overset{\text{def}}{=} \tau
  \]
...with our proof for the weakening property. Let's first extend the permutation operation to:

\[
\begin{align*}
\{ t_1, \ldots, t_n \} & \quad \text{sets of lambda-terms} \\
( x, y ) & \quad \text{pairs} \\
X & \quad \text{types}
\end{align*}
\]

So given a typing-context \( \Gamma \), \( \pi \cdot \Gamma \) will always be a typing-context, while \( \Gamma[a := b] \) is only in some specific circumstances.

\[
\pi \cdot \tau \equiv \tau
\]
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:

\[
t_1 \approx t_2 \quad \text{if and only if} \quad \pi \cdot t_1 \approx \pi \cdot t_2
\]

\[
a \ not \notn{t} \quad \text{if and only if} \quad \pi \cdot a \ not \notn{\pi \cdot t}
\]

It seems, equivariance is an important concept when reasoning about properties involving binders.
... Also \( \vdash \) and \( \varphi \)

- The typing relation is equivariant:

\[
\Gamma \vdash t : \tau \iff \pi \cdot \Gamma \vdash \pi \cdot t : \pi \cdot \tau
\]

\[
\frac{a : \tau \in \Gamma \quad \pi \cdot \Gamma \vdash \pi \cdot [a]_\alpha : \tau}{\Gamma \vdash [a]_\alpha : \tau}
\iff
\frac{\pi \cdot (a : \tau) \in \pi \cdot \Gamma \quad \pi \cdot \Gamma \vdash [\pi \cdot a]_\alpha : \pi \cdot \tau}{\pi \cdot \Gamma \vdash [\pi \cdot [t]_\alpha : \pi \cdot \tau
\]

- Our induction-hypothesis is equivariant, i.e.

\[
\varphi(\Gamma;[t]_\alpha; \tau) \iff \varphi(\pi \cdot \Gamma; \pi \cdot [t]_\alpha; \pi \cdot \tau)
\]

\[
(\forall \tau')(\forall a' \not\in \dom(\Gamma)) \quad \Gamma, a' : \tau' \vdash [t]_\alpha : \tau
\iff
(\forall \tau')(\forall a' \not\in \dom(\pi \cdot \Gamma)) \quad \pi \cdot \Gamma, a' : \tau' \vdash \pi \cdot [t]_\alpha : \pi \cdot \tau
\]
the typing relation is equivariant.

Be careful! The $\forall$-quantifiers are not allowed to quantify anything in $\pi$—if they do, we have to rename the quantified meta-variables. How this is done conveniently will be explained on Tuesday and Wednesday.

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

$$t : \pi \cdot \tau$$

$$\in \pi \cdot \Gamma$$

$$\left[ t \right]_\alpha : \pi \cdot \tau$$

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

$$\iff$$

$$\forall \tau' (\forall a' \notin \text{dom}(\pi \cdot \Gamma)) \; \pi \cdot \Gamma, a' : \tau' \vdash \pi \cdot [t]_\alpha : \pi \cdot \tau$$
Case $a' = a$: from the premise we know

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

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

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

By equivariance we know

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

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

Nancy, 16. August 2004 - p.29 (2/6)
Now the Proof

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

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

By equivariance we know

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

for any fresh atom $b$, i.e. one not occurring in $\Gamma$, $t$, 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. $a \not\in \text{dom}(\Gamma)$

By equivariance we know

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

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

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

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

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

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

By equivariance we know

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

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

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

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

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

4. $\Gamma, a' : \tau' \vdash [\lambda b.(a b) \cdot t]_\alpha : \tau_1 \rightarrow \tau_2$
Now the Proof

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

1. $\varphi(\Gamma, t, \{a, a'\})$

By equivariance we know

1'. $\varphi(\Gamma, b : \tau_1, a : \tau' \vdash (a b) \cdot t, \{a, a'\})$

for any \( \{a, a'\} \). 

By definition

3. $\varphi(\Gamma, b : \tau_1, a : \tau' \vdash (a b) \cdot t, \{a, a'\})$

But now

$$\lambda b. (a b) \cdot t \simeq \lambda a. t$$

so we have

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

and finally we know that

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

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

Nancy, 16. August 2004 - p.29 (6/6)
A Bird’s Eye View

Old World

meta-language
binders, quantifiers

object-language

HOAS
FOAS

Nominal World

meta-language
binders, quantifiers

object-language

FOAS

Nancy, 16. August 2004 - p.30 (1/5)
A Bird’s Eye View

Old World

meta-language
binders, quantifiers

object-language

\{HOAS, FOAS\}

Nominal World

meta-language
binders, quantifiers

object-language

\{FOAS\}

lambda-calculus, pi-calculus, proof-theory, ...
A Bird’s Eye View

Old World

meta-language
binders, quantifiers

object-language

HOAS

FOAS

Nominal World

meta-language
binders, quantifiers

object-language

FOAS
A Bird’s Eye View

Old World
meta-language
binders, quantifiers
object-language

Nominal World
meta-language
binders, quantifiers
object-language

HOAS
FOAS
NAS
A Bird’s Eye View

Old World
- meta-language
  - binders, quantifiers
- object-language
  - HOAS
  - FOAS

Nominal World
- meta-language
  - binders, quantifiers
- object-language
  - NAS

Tomorrow
Two Points to Sleep Over

- if you need to rename binders:
  
  permutations behave much better than renaming-substitutions

- if you are trying to prove something about syntax with binders:
  
  equivariance seems to be the key