

Nominal Techniques as an Isabelle/HOL-Theory

Christian Urban

University of Munich (LMU)

On Paper

Lemma: $a \notin FV(t_2)$ implies $a \notin FV(t_1[a := t_2])$

On Paper

Lemma: $a \notin FV(t_2)$ implies $a \notin FV(t_1[a := t_2])$

Proof: By induction on the structure of t_1 .

On Paper

Lemma: $a \notin FV(t_2)$ implies $a \notin FV(t_1[a := t_2])$

Proof: By induction on the structure of t_1 .

case variables:

On Paper

Lemma: $a \notin FV(t_2)$ implies $a \notin FV(t_1[a := t_2])$

Proof: By induction on the structure of t_1 .

case variables:

subcase 1: $a \notin FV(a[a := t_2])$

subcase 2: $a \notin FV(b[a := t_2])$

On Paper

Lemma: $a \notin FV(t_2)$ implies $a \notin FV(t_1[a := t_2])$

Proof: By induction on the structure of t_1 .

case variables:

subcase 1: $a \notin FV(t_2)$ by assumption
subcase 2: $a \notin FV(b[a := t_2])$

On Paper

Lemma: $a \notin FV(t_2)$ implies $a \notin FV(t_1[a := t_2])$

Proof: By induction on the structure of t_1 .

case variables:

subcase 1: $a \notin FV(t_2)$

by assumption

subcase 2: $a \notin FV(b)$

ok

On Paper

Lemma: $a \notin FV(t_2)$ implies $a \notin FV(t_1[a := t_2])$

Proof: By induction on the structure of t_1 .

case variables:

subcase 1: $a \notin FV(t_2)$

by assumption

subcase 2: $a \notin FV(b)$

ok

case applications:

$a \notin FV(s_1 s_2 [a := t_2])$

On Paper

Lemma: $a \notin FV(t_2)$ implies $a \notin FV(t_1[a := t_2])$

Proof: By induction on the structure of t_1 .

case variables:

subcase 1: $a \notin FV(t_2)$

by assumption

subcase 2: $a \notin FV(b)$

ok

case applications:

$a \notin FV(s_1[a := t_2] s_2[a := t_2])$

On Paper

Lemma: $a \notin FV(t_2)$ implies $a \notin FV(t_1[a := t_2])$

Proof: By induction on the structure of t_1 .

case variables:

subcase 1: $a \notin FV(t_2)$

by assumption

subcase 2: $a \notin FV(b)$

ok

case applications:

$a \notin FV(s_1[a := t_2]) \cup FV(s_2[a := t_2])$

On Paper

Lemma: $a \notin FV(t_2)$ implies $a \notin FV(t_1[a := t_2])$

Proof: By induction on the structure of t_1 .

case variables:

subcase 1: $a \notin FV(t_2)$ by assumption
subcase 2: $a \notin FV(b)$ ok

case applications:

$a \notin FV(s_1[a := t_2]) \cup FV(s_2[a := t_2])$ by IH

On Paper

Lemma: $a \notin FV(t_2)$ implies $a \notin FV(t_1[a := t_2])$

Proof: By induction on the structure of t_1 .

case variables:

subcase 1: $a \notin FV(t_2)$ by assumption
subcase 2: $a \notin FV(b)$ ok

case applications:

$a \notin FV(s_1[a := t_2]) \cup FV(s_2[a := t_2])$ by IH

case abstractions (c sufficiently fresh):

$a \notin FV(\lambda c.s [a := t_2])$

On Paper

Lemma: $a \notin FV(t_2)$ implies $a \notin FV(t_1[a := t_2])$

Proof: By induction on the structure of t_1 .

case variables:

subcase 1: $a \notin FV(t_2)$ by assumption
subcase 2: $a \notin FV(b)$ ok

case applications:

$a \notin FV(s_1[a := t_2]) \cup FV(s_2[a := t_2])$ by IH

case abstractions (c sufficiently fresh):

$a \notin FV(\lambda c.(s[a := t_2]))$

On Paper

Lemma: $a \notin FV(t_2)$ implies $a \notin FV(t_1[a := t_2])$

Proof: By induction on the structure of t_1 .

case variables:

subcase 1: $a \notin FV(t_2)$ by assumption
subcase 2: $a \notin FV(b)$ ok

case applications:

$a \notin FV(s_1[a := t_2]) \cup FV(s_2[a := t_2])$ by IH

case abstractions (c sufficiently fresh):

$a \notin FV(s[a := t_2]) - \{c\}$

On Paper

Lemma: $a \notin FV(t_2)$ implies $a \notin FV(t_1[a := t_2])$

Proof: By induction on the structure of t_1 .

case variables:

subcase 1: $a \notin FV(t_2)$ by assumption
subcase 2: $a \notin FV(b)$ ok

case applications:

$a \notin FV(s_1[a := t_2]) \cup FV(s_2[a := t_2])$ by IH

case abstractions (c sufficiently fresh):

$a \notin FV(s[a := t_2]) - \{c\}$ by IH

On Paper

Lemma: $a \notin FV(t_2)$ implies $a \notin FV(t_1[a := t_2])$

Proof: By induction on the structure of t_1 .

case variables:

subcase 1: $a \notin FV(t_2)$ by assumption
subcase 2: $a \notin FV(b)$ ok

case applications:

$a \notin FV(s_1[a := t_2]) \cup FV(s_2[a := t_2])$ by IH

case abstractions (c sufficiently fresh):

$a \notin FV(s[a := t_2]) - \{c\}$ by IH

Done.

On Paper

Lemma: $a \notin FV(t_2)$ implies $a \notin FV(t_1[a := t_2])$

Proof: By **induction** on the structure of t_1 .

case variables:

subcase 1: $a \notin FV(t_2)$ by assumption
subcase 2: $a \notin FV(b)$ ok

case applications:

$a \notin FV(s_1[a := t_2]) \cup FV(s_2[a := t_2])$ by IH

case abstractions (**c sufficiently fresh**):

$a \notin FV(s[a := t_2]) - \{c\}$ by IH

Done.

Existing Formalisation Techniques

- with "bare hands"
- de-Brujin indices*
- HOAS
(you might be interested in that I give an inductive definition that is in **bijection** with the α -equated lambda-terms)

* If you like them in formal reasoning, you may quietly leave now—nothing I can do for you.

And Now in Isabelle

lemma: " $a \# t_2 \implies a \# t_1 [a ::= t_2]$ "

I will write

$$a \# t \text{ for } a \notin FV(t)$$

I will also start from (plus a few minor facts):

$$\text{Am } a [a ::= t] = t$$

$$\text{Am } b [a ::= t] = \text{Am } b \quad \text{if } a \neq b$$

$$\text{Pr } s_1 s_2 [a ::= t] = \text{Pr}(s_1 [a ::= t]) (s_2 [a ::= t])$$

$$[b].s [a ::= t] = [b].(s [a ::= t]) \quad \text{if } b \# (a, t)$$

And Now in Isabelle

```
lemma: "a # t2  $\implies$  a # t1 [a ::= t2]"
apply(rule ind[of " $\lambda t_1 (a, t_2). a \# t_1 [a ::= t_2]$ "
                  "(a, t2)", simplified])
apply(case_tac "c=a")
apply(simp add: subs_simps)
apply(simp add: subs_simps)
apply(simp add: subs_simps)
apply(subgoal_tac: " $\exists c. c \# (a, t_2)$ ")
apply(force intro!: fresh_absI1
        simp add: fresh_prod subs_simps)
apply(rule exists_fresh)
done
```

And Now in Isabelle

```
lemma: "a # t2  $\implies$  a # t1 [a ::= t2]"
```

```
apply(rule ind[of " $\lambda t_1 (a, t_2). a \# t_1 [a ::= t_2]$ "  
                "(a, t2)", simplified])
```

```
apply(case_tac "c=a")
```

```
apply(simp add: subs_simps)
```

```
apply(goal (lemma, 1 subgoal):
```

```
apply(1. a # t2  $\implies$  a # t1 [a ::= t2]
```

```
apply(subgoal_tac: " $\exists c. c \# (a, t_2)$ ")
```

```
apply(force intro!: fresh_absI1  
       simp add: fresh_prod subs_simps)
```

```
apply(rule exists_fresh)
```

```
done
```

And Now in Isabelle

```
lemma: "a # t2  $\implies$  a # t1 [a ::= t2]"  
apply (rule ind [of " $\lambda t_1 (a, t_2). a \# t_1 [a ::= t_2]$ "  
                    "(a, t2)", simplified])
```

```
apply (case_tac "c=a")
```

```
apply (simp add: subs_simps)
```

```
apply goal (lemma, 3 subgoals):
```

```
apply 1.  $\bigwedge c. a \# t_2 \implies a \# \text{Am } c [a ::= t_2]$ 
```

```
apply 2.  $\bigwedge t_1 t_2 a. [| a \# t_2; a \# t_1 [a ::= t_2] \wedge$   
                     $t_2 a [a ::= t_2] |] \implies a \# \text{Pr } t_1 t_2 a [a ::= t_2]$ 
```

```
apply 3.  $a \# t_2 \implies \exists c. c \# (a, t_2) \wedge$   
         $(\forall t. a \# t [a ::= t_2] \longrightarrow a \# [c]. t [a ::= t_2])$ 
```

```
apply (auto simp only: subst)
```

```
done
```

And Now in Isabelle

```
lemma: "a # t2  $\implies$  a # t1 [a ::= t2]"  
apply (rule ind [of " $\lambda t_1 (a, t_2). a \# t_1 [a ::= t_2]$ "  
" (a, t2)", simplified])  
apply (case_tac "c=a")
```

```
apply (simp add: subs_simps)
```

```
apply goal (lemma, 4 subgoals):
```

```
apply 1.  $\bigwedge c. [| a \# t_2; c=a |] \implies a \# Am\ c [a ::= t_2]$ 
```

```
apply 2.  $\bigwedge c. [| a \# t_2; c \neq a |] \implies a \# Am\ c [a ::= t_2]$ 
```

```
apply 3.  $\bigwedge t_1 t_{2a}. [| a \# t_2; a \# t_1 [a ::= t_2] \wedge$   
 $t_{2a} [a ::= t_2] |] \implies a \# Pr\ t_1\ t_{2a} [a ::= t_2]$ 
```

```
apply 4.  $a \# t_2 \implies \exists c. c \# (a, t_2) \wedge$   
 $(\forall t. a \# t [a ::= t_2] \longrightarrow a \# [c]. t [a ::= t_2])$   
done
```

And Now in Isabelle

```
lemma: "a # t2  $\implies$  a # t1 [a ::= t2]"
apply(rule ind[of " $\lambda t_1 (a, t_2). a \# t_1 [a ::= t_2]$ "
                  "(a, t2)", simplified])
apply(case_tac "c=a")
apply(simp add: subs_simps)
apply(simp add: subs_simps)
apply(goal (lemma, 3 subgoals):
1.  $\bigwedge c. [| a \# t_2; c \neq a |] \implies a \# \text{Am } c [a ::= t_2]$ 
2.  $\bigwedge t_1 t_{2a}. [| a \# t_2; a \# t_1 [a ::= t_2] \wedge$ 
    $t_{2a} [a ::= t_2] |] \implies a \# \text{Pr } t_1 t_{2a} [a ::= t_2]$ 
3.  $a \# t_2 \implies \exists c. c \# (a, t_2) \wedge$ 
    $(\forall t. a \# t [a ::= t_2] \longrightarrow a \# [c]. t [a ::= t_2])$ 
done
```

And Now in Isabelle

```
lemma: "a # t2  $\implies$  a # t1 [a ::= t2]"
apply(rule ind[of " $\lambda t_1 (a, t_2). a \# t_1 [a ::= t_2]$ "
                  "(a, t2)", simplified])
apply(case_tac "c=a")
apply(simp add: subs_simps)
apply(simp add: subs_simps)
apply(simp add: subs_simps)
apply(goal (lemma, 2 subgoals):
1.  $\bigwedge t_1 t_2 a. [| a \# t_2; a \# t_1 [a ::= t_2] \wedge$ 
    $t_2 a [a ::= t_2] |] \implies a \# \text{Pr } t_1 t_2 a [a ::= t_2]$ 
2.  $a \# t_2 \implies \exists c. c \# (a, t_2) \wedge$ 
    $(\forall t. a \# t [a ::= t_2] \longrightarrow a \# [c]. t [a ::= t_2])$ 
done
```

And Now in Isabelle

```
lemma: "a # t2  $\implies$  a # t1 [a ::= t2]"
```

```
apply(rule ind[of " $\lambda t_1 (a, t_2). a \# t_1 [a ::= t_2]$ "])
```

```
goal (lemma, 1 subgoal):
```

```
1. a # t2  $\implies$   $\exists c. c \# (a, t_2) \wedge$   
   ( $\forall t. a \# t [a ::= t_2] \longrightarrow a \# [c]. t [a ::= t_2]$ )
```

```
apply(simp add: subs_simps)
```

```
apply(simp add: subs_simps)
```

```
apply(simp add: subs_simps)
```

```
apply(subgoal_tac: " $\exists c. c \# (a, t_2)$ ")
```

```
apply(force intro!: fresh_absI1  
      simp add: fresh_prod subs_simps)
```

```
apply(rule exists_fresh)
```

```
done
```

And Now in Isabelle

```
lemma: "a # t2  $\implies$  a # t1 [a ::= t2]"
```

```
apply(rule ind[of " $\lambda t_1 (a, t_2). a \# t_1 [a ::= t_2]$ "])
```

```
goal (lemma, 2 subgoals):
```

```
apply 1. [| a # t2;  $\exists c. c \# (a, t_2)$  |]  $\implies$   
       $\exists c. c \# (a, t_2) \wedge$ 
```

```
apply      ( $\forall t. a \# t [a ::= t_2] \longrightarrow a \# [c]. t [a ::= t_2]$ )
```

```
apply 2. a # t2  $\implies$   $\exists c. c \# (a, t_2)$ 
```

```
apply(simp add: subs_simps)
```

```
apply(subgoal_tac: " $\exists c. c \# (a, t_2)$ ")
```

```
apply(force intro!: fresh_absI1  
      simp add: fresh_prod subs_simps)
```

```
apply(rule exists_fresh)
```

```
done
```

And Now in Isabelle

```
lemma: "a # t2  $\implies$  a # t1 [a ::= t2]"
```

```
apply(rule ind[of " $\lambda t_1 (a, t_2). a \# t_1 [a ::= t_2]$ "]
```

```
goal (lemma, 1 subgoal):
```

```
1. a # t2  $\implies$   $\exists c. c \# (a, t_2)$ 
```

```
apply(simp add: subs_simps)
```

```
apply(simp add: subs_simps)
```

```
apply(simp add: subs_simps)
```

```
apply(subgoal_tac: " $\exists c. c \# (a, t_2)$ ")
```

```
apply(force intro!: fresh_absI1  
      simp add: fresh_prod subs_simps)
```

```
apply(rule exists_fresh)
```

```
done
```

And Now in Isabelle

```
lemma: "a # t2  $\implies$  a # t1 [a ::= t2]"
```

```
apply(rule ind[of " $\lambda t_1 (a, t_2). a \# t_1 [a ::= t_2]$ "])
```

```
goal (lemma):
```

```
No subgoals!
```

```
apply
```

```
apply(simp add: subs_simps)
```

```
apply(simp add: subs_simps)
```

```
apply(simp add: subs_simps)
```

```
apply(subgoal_tac: " $\exists c. c \# (a, t_2)$ ")
```

```
apply(force intro!: fresh_absI1
```

```
simp add: fresh_prod subs_simps)
```

```
apply(rule exists_fresh)
```

```
done
```

And Now in Isabelle

```
lemma: "a # t2  $\implies$  a # t1 [a ::= t2]"
apply(rule ind[of " $\lambda t_1 (a, t_2). a \# t_1 [a ::= t_2]$ "
                  "(a, t2)", simplified])
apply(case_tac "c=a")
apply(simp add: subs_simps)
apply(simp add: subs_simps)
apply(simp add: subs_simps)
apply(subgoal_tac: " $\exists c. c \# (a, t_2)$ ")
apply(force intro!: fresh_absI1
        simp add: fresh_prod subs_simps)
apply(rule exists_fresh)
done
```

And Now in Isabelle

```
lemma: "a # t2  $\implies$  a # t1 [a ::= t2]"
```

```
apply(rule ind[of " $\lambda$ t1 (a, t2). a # t1 [a ::= t2]"
```

Well, I cheated you by one line (the induction requires us to prove four subgoals), but apart from that, this **is** the proof in Isabelle. And you have to admit, one can hardly get any "closer" to the original "proof".

The **aim** of this talk is to give you an idea what is going on behind this proof (about the details I am **more** than happy to talk off-line).

```
apply(rule exists_fresh)
```

```
done
```

What Is Behind?

- a trick
- and plenty of ideas from Pitts' nominal work

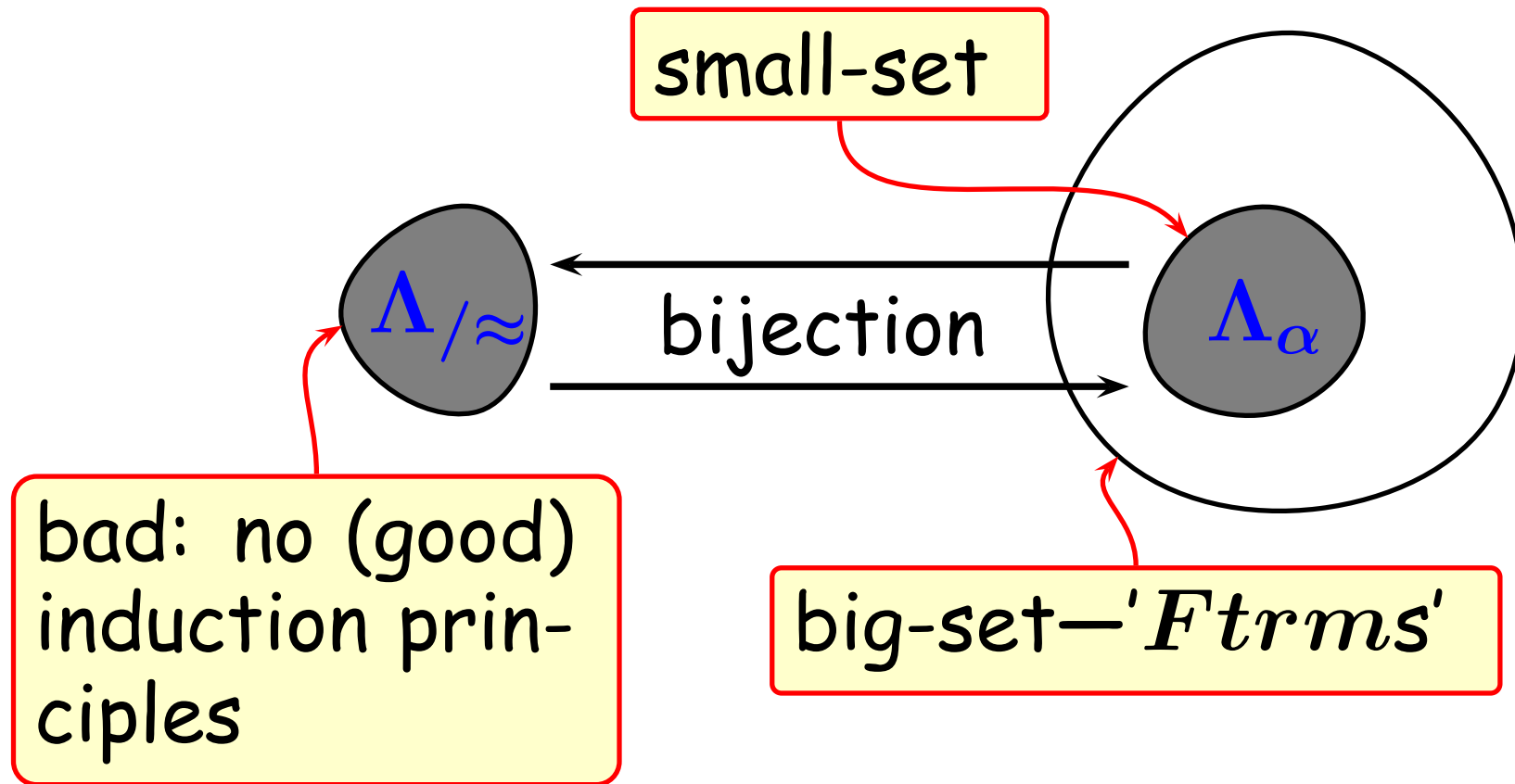
(In its original formulation Nominal Logic is incompatible with the axiom of choice; FM-sets aren't the best starting point for formal verifications. Therefore some adjustments are needed for theorem provers.)

The Trick

I define inductively the α -equivalence classes of lambda-terms.

The Trick

I define inductively the α -equivalence classes of lambda-terms.



The Trick

I define inductively the α -equivalence classes of lambda-terms.

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

$$t ::= \text{Am}(a) \\ \quad | \text{Pr } t \ t \\ \quad | [a].t$$

be
in
ciples

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

Big-Set / *Ftrms*

One step back: naive attempt for big-set

Ftrm ::=

	<i>Am</i> : <i>Atom</i>	'atoms'
	<i>Pr</i> : <i>Ftrm</i> × <i>Ftrm</i>	'pairs'
	<i>Se</i> : <i>Ftrm Set</i>	'α-eq-cl'



Big-Set / *Ftrms*

One step back: naive attempt for big-set

Ftrm ::=

	<i>Am</i> : <i>Atom</i>	'atoms'
	<i>Pr</i> : <i>Ftrm</i> × <i>Ftrm</i>	'pairs'
	<i>Se</i> : <i>Ftrm Set</i>	'α-eq-cl'

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

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



Big-Set / *Ftrms*

Better attempt for big-set

$Ftrm ::= Er$	'error'
$Am : Atom$	'atoms'
$Pr : Ftrm \times Ftrm$	'pairs'
$Se : Atom \rightarrow Ftrm$	' α -eq-cl'

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

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



1st Idea: Permutations

A permutation is a list of pairs of atoms

Permutations are bijective mappings from atoms to atoms:

$$\begin{pmatrix} a \rightarrow c \\ b \rightarrow a \\ c \rightarrow b \end{pmatrix} \bullet a = c$$

represented by swapping $(a\ b)(a\ c) \bullet a = c$

1st Idea: Permutations

A permutation is a list of pairs of atoms

- $[]$ 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

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

Permutations for Big-Set

Given the permutation action for atoms, we can permute all free atoms in *ftrms*:

$$\begin{aligned}\pi \bullet Er &\stackrel{\text{def}}{=} Er \\ \pi \bullet Am(a) &\stackrel{\text{def}}{=} Am(\pi \bullet a) \\ \pi \bullet Pr(t_1, t_2) &\stackrel{\text{def}}{=} Pr(\pi \bullet t_1, \pi \bullet t_2) \\ \pi \bullet Se(fn) &\stackrel{\text{def}}{=} Se(\lambda a. \pi \bullet (fn(\pi^{-1} \bullet a)))\end{aligned}$$



Properties of Permutations

We can prove

- $(a\ a) \bullet t = t$

- $\pi^{-1} \bullet (\pi \bullet t) = t$

- $\pi \bullet t_1 = t_2$ iff $t_1 = \pi^{-1} \bullet t_2$

- etc.

Actually we can do all this abstractly by requiring:

- $[] \bullet x = x$

- $(\pi_1 @ \pi_2) \bullet x = \pi_1 \bullet (\pi_2 \bullet x)$

- $ds(\pi_1, \pi_2) = \emptyset$ implies $\pi_1 \bullet x = \pi_2 \bullet x$

Axclasses Are Great

PTypes

UI

FSTypes

UI

NomTypes

$$\blacksquare [] \bullet x = x$$

$$\blacksquare (\pi_1 @ \pi_2) \bullet x = \pi_1 \bullet (\pi_2 \bullet x)$$

$$\blacksquare ds(\pi_1, \pi_2) = \emptyset \Rightarrow \\ \pi_1 \bullet x = \pi_2 \bullet x$$

Given an appropriate definition of a permutation action, everything is a PType (in particular our big-set and small-set).

2nd Idea: Support and Not in the Support

For every *Ptype* we can define the notion of support, a set of atoms (in a minute).

An old friend can be defined in terms of support:

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

So for our small-set, the support should coincide with $FV(x)$.

SUPPORT!!!

Something to chew on:

$\text{supp} : PType \rightarrow Atom Set$

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

In words: all atoms a where the set

$\{b \mid (a b) \cdot x \neq x\}$

is infinite (each swapping $(a b)$ needs to change something "syntactically" in x).

Support of an Atom

What is the support of the atom c ?

$$\text{supp}(c) \stackrel{\text{def}}{=} \{a \mid \text{infinite } \{b \mid (a b) \cdot c \neq c\}\}$$

Let's check the (infinitely many) atoms one by one:

Support of an Atom

What is the support of the atom c ?

$$\text{supp}(c) \stackrel{\text{def}}{=} \{a \mid \text{infinite } \{b \mid (a b) \cdot c \neq c\}\}$$

Let's check the (infinitely many) atoms one by one:

$$a: \quad (a ?) \cdot c \neq c$$

Support of an Atom

What is the support of the atom c ?

$$\text{supp}(c) \stackrel{\text{def}}{=} \{a \mid \text{infinite } \{b \mid (a b) \cdot c \neq c\}\}$$

Let's check the (infinitely many) atoms one by one:

$$a: \quad (a ?) \cdot c \neq c \quad \text{no}$$

$$b: \quad (b ?) \cdot c \neq c$$

Support of an Atom

What is the support of the atom c ?

$$\text{supp}(c) \stackrel{\text{def}}{=} \{a \mid \text{infinite } \{b \mid (a b) \cdot c \neq c\}\}$$

Let's check the (infinitely many) atoms one by one:

$$a: \quad (a ?) \cdot c \neq c \quad \text{no}$$

$$b: \quad (b ?) \cdot c \neq c \quad \text{no}$$

$$c: \quad (c ?) \cdot c \neq c$$

Support of an Atom

What is the support of the atom c ?

$$\text{supp}(c) \stackrel{\text{def}}{=} \{a \mid \text{infinite } \{b \mid (a b) \cdot c \neq c\}\}$$

Let's check the (infinitely many) atoms one by one:

$a:$	$(a?) \cdot c \neq c$	no
$b:$	$(b?) \cdot c \neq c$	no
$c:$	$(c?) \cdot c \neq c$	yes
$d:$	$(d?) \cdot c \neq c$	

Support of an Atom

What is the support of the atom c ?

$$\text{supp}(c) \stackrel{\text{def}}{=} \{a \mid \text{infinite } \{b \mid (a b) \cdot c \neq c\}\}$$

Let's check the (infinitely many) atoms one by one:

a :	$(a ?) \cdot c \neq c$	no
b :	$(b ?) \cdot c \neq c$	no
c :	$(c ?) \cdot c \neq c$	yes
d :	$(d ?) \cdot c \neq c$	no
	\vdots	no

Support of an Atom

What is the support of the atom c ?

$\text{supp}(c) \text{ def } \{a \mid \text{infinite } [b \mid (a \cdot b) = c / a]\}$

So:

$$\text{supp}(Am(c)) = \{c\}$$

$$\text{supp}(Pr(t_1, t_2)) = \text{supp}(t_1) \cup \text{supp}(t_2)$$

provided $\text{supp}(t)$ is a finite set, then

$$\text{supp}([c].t) = \text{supp}(t) - \{c\}$$

$c: (c!) \cdot c \neq c$ yes

$d: (d?) \cdot c \neq c$ no

\vdots no

Finite Support

PTypes

■ $\text{finite}(\text{supp}(x))$

U

Atoms are finitely supported; so are pairs of FSTypes, lists, finite sets of FSTypes, ... Our small-set is finitely supported (set of free variables), but big-set isn't.

FSTypes

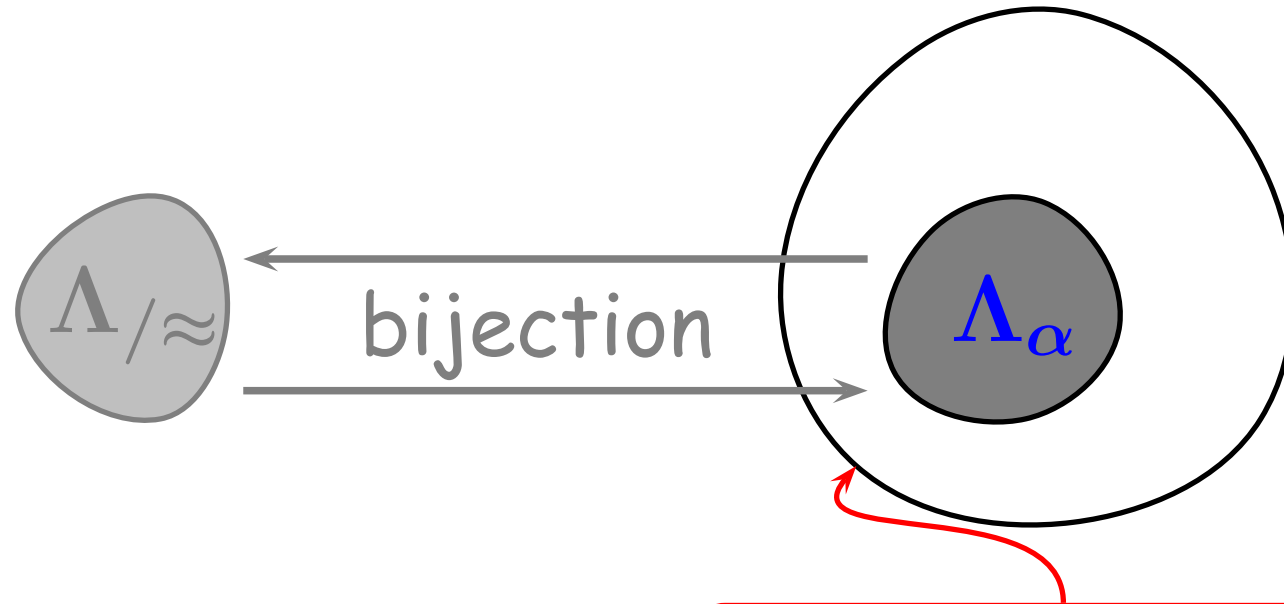
U

NomTypes

Why finite support?

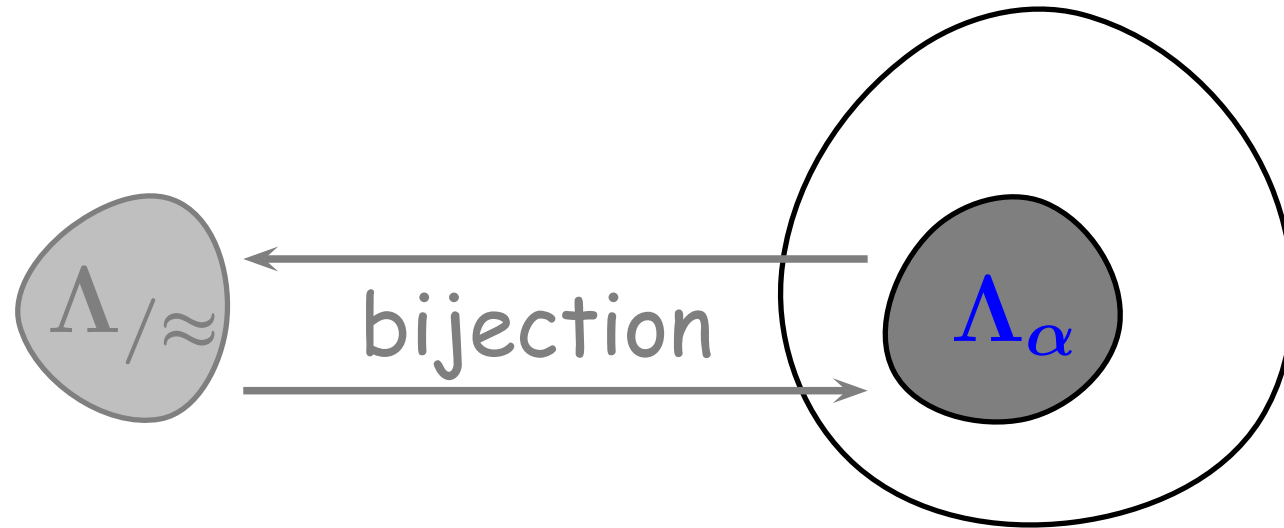
$(\forall x : FSType)(\exists a : Atm) a \neq x$!!

What About Small-Set?



$t ::= E_r$
| $A_m(a)$
| $Pr(t_1, t_2)$
| $Se(fn)$

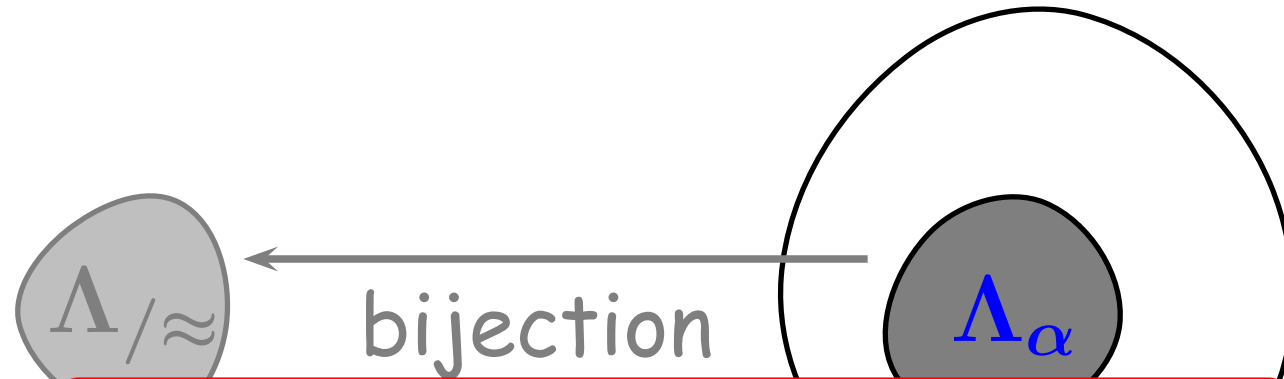
What About Small-Set?



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

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

What About Small-Set?



Breath deeply: the user will never ever see anything about functions!

For Λ_α , we have specific functions, namely

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

Function $[a].t \text{ '}' \equiv \text{' } [\lambda a.t]_{\alpha}$

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

Function $[a].t \text{ '}' \equiv \text{' } [\lambda a.t]_{\alpha}$

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

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

Function $[a].t \text{ '=='} [\lambda a.t]_a$

$[a].Pr(a, c) \stackrel{\text{def}}{=}$

Se $(\lambda b. \text{if } a = b$
then $Pr(a, c)$
else if $b \neq Pr(a, c)$
then $(b a) \bullet Pr(a, c)$ else Er)

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

Function $[a].t \text{ '=='} [\lambda a.t]_a$

$[a].Pr(a, c) \stackrel{\text{def}}{=}$

$Se (\lambda b. \text{if } a = b$
 $\text{then } Pr(a, c)$
 $\text{else if } b \neq Pr(a, c)$
 $\text{then } (b a) \bullet Pr(a, c) \text{ else } Er)$

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

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

Function $[a].t \text{ '=='} [\lambda a.t]_a$

$[a].Pr(a, c) \stackrel{\text{def}}{=}$

Se $(\lambda b. \text{if } a = b$
then $Pr(a, c)$
else if $b \neq Pr(a, c)$
then $(b a) \bullet Pr(a, c)$ else Er)

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

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

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

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

$[a].Pr(a, c) \stackrel{\text{def}}{=}$

$Se (\lambda b. \text{if } a = b$
 then $Pr(a, c)$
 else if $b \neq Pr(a, c)$
 then $(b a) \bullet Pr(a, c)$ else Er)

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

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

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

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

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

$[a].Pr(a, c) \stackrel{\text{def}}{=}$

Se $(\lambda b. \text{if } a = b$
then $Pr(a, c)$
else if $b \neq Pr(a, c)$
then $(b a) \bullet Pr(a, c)$ else Er)

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

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

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

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

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

⋮

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

$[a].Pr(a, c) \stackrel{\text{def}}{=}$

$Se (\lambda b. \text{if } a = b$
 then $Pr(a, c)$
 else if $b \neq Pr(a, c)$
 then $(b a) \bullet Pr(a, c)$ else Er)

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

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

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

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

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

⋮

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

$[a].Pr(a, c) \stackrel{\text{def}}{=}$

Se $(\lambda b. \text{if } a = b$
then $Pr(a, c)$
else if $b \neq Pr(a, c)$
then $(b a) \bullet Pr(a, c)$ else Er)

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

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

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

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

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

\vdots

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

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

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

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

\vdots

Properties of $[a].t$

PTypes

UI

FSTypes

UI

NomTypes

$$\blacksquare \pi \bullet ([a].t) = [\pi \bullet a].(\pi \bullet t)$$

$$\blacksquare t_1 = t_2 \Leftrightarrow [a].t_1 = [a].t_2$$

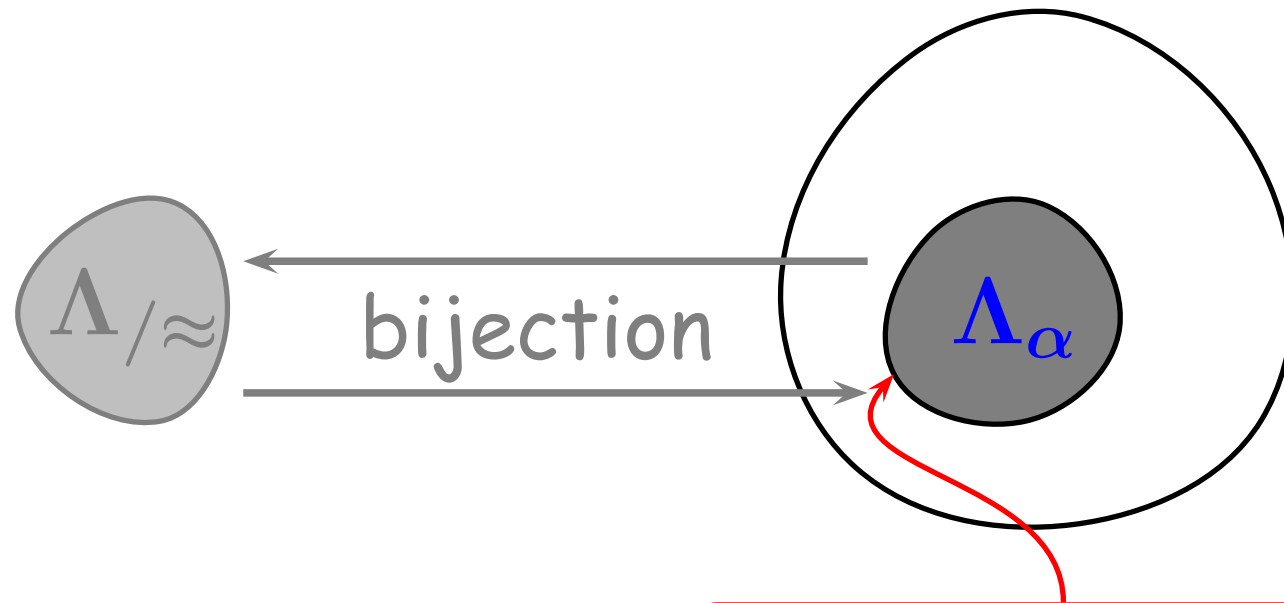
$$\blacksquare a \neq b \Rightarrow$$

$$t_1 = (a \ b) \bullet t_2 \wedge a \# t_2$$

$$\Leftrightarrow [a].t_1 = [b].t_2$$

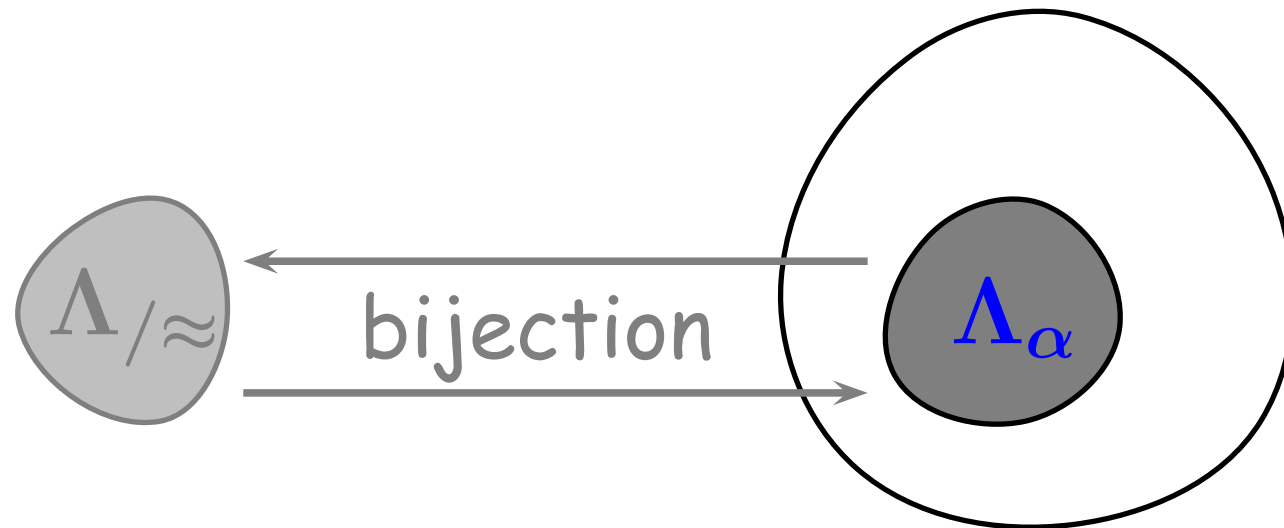
The last two axioms say that $[a].t$ behaves like an abstraction.

Definition of Small-Set



$t ::= Am(a)$
| $Pr(t_1, t_2)$
| $[a].t$

Definition of Small-Set



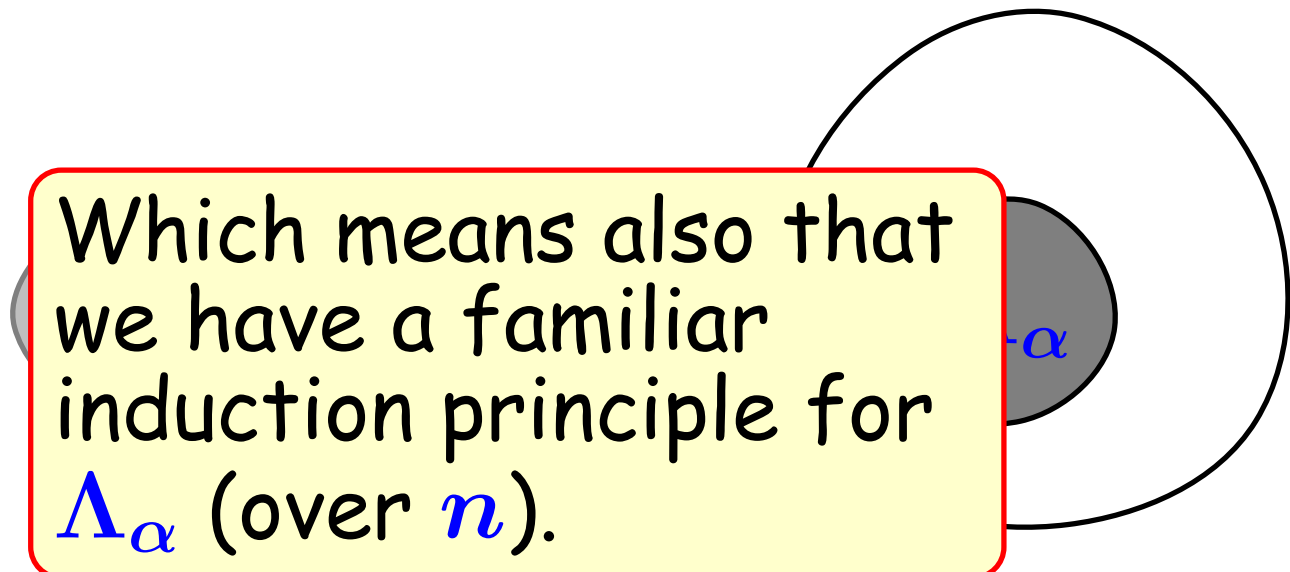
$$F(X) \stackrel{\text{def}}{=} AM \cup PR(X) \cup AS(X)$$

$$\Lambda_\alpha \stackrel{\text{def}}{=} \text{lfp}(F) = \bigcup_n F_n$$

$$\text{where } F_0 \stackrel{\text{def}}{=} F(\emptyset)$$

$$F_{n+1} \stackrel{\text{def}}{=} F(F_n)$$

Definition of Small-Set



Which means also that we have a familiar induction principle for Λ_α (over n).

$$F(X) \stackrel{\text{def}}{=} AM \cup PR(X) \cup AS(X)$$

$$\Lambda_\alpha \stackrel{\text{def}}{=} \text{lfp}(F) = \bigcup_n F_n$$

$$\text{where } F_0 \stackrel{\text{def}}{=} F(\emptyset)$$

$$F_{n+1} \stackrel{\text{def}}{=} F(F_n)$$

Bijection

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

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

with the property

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

Bijection

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

$$\begin{aligned} q(a) & \stackrel{\text{def}}{=} [a] \\ q(t_1 \sigma_2) & \stackrel{\text{def}}{=} [(q(t_1), q(t_2))] \\ q(\lambda a.t) & \stackrel{\text{def}}{=} [\lambda a.q(t)] \end{aligned}$$

Believe me—I am not going to show this here.

with the property

$$t_1 \approx t_2 \iff q(t_1) = q(t_2)$$

Induction on Λ_α

Since we defined Λ_α inductively, we have:

$$(\forall a) P (Am(a)) x$$

$$(\forall t_1, t_2) P t_1 x \wedge P t_2 x \Rightarrow P (Pr(t_1, t_2)) x$$

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

$$(\forall t) P t x$$

Proof: By induction on n (the "stages" of constructing Λ_α).

Remember

Lemma: $a \notin FV(t_2)$ implies $a \notin FV(t_1[a := t_2])$

By induction on the structure of t_1 .

case variables:

subcase 1: $a \notin FV(t_2)$

by assumption

subcase 2: $a \notin FV(b)$

ok

case applications:

$a \notin FV(s_1[a := t_2]) \cup FV(s_2[a := t_2])$

by IH

case abstractions (c sufficiently fresh):

$a \notin FV(s[a := t_2]) - \{c\}$

by IH

Done.

Induction on Λ_α

$$(\forall a) P (Am(a)) x$$

$$(\forall t_1, t_2) P t_1 x \wedge P t_2 x \Rightarrow P (Pr(t_1, t_2)) x$$

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

$$(\forall t) P t x$$

Induction on Λ_α

$eqvt(P)$

$(\forall a) P (Am(a)) x$

$(\forall t_1, t_2) P t_1 x \wedge P t_2 x \Rightarrow P (Pr(t_1, t_2)) x$

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

$(\forall t) P t x$

We can strengthen our induction principle if we know that the induction hypothesis is **equivariant**—invariant under renamings/permutations.

Equivariance

$$\text{eqvt}(P) \stackrel{\text{def}}{=} (\forall t : \Lambda_a) (\forall x : FSType) (\forall \pi) \\ P t x \Rightarrow P(\pi \bullet t)(\pi \bullet x)$$

In my inductions, I have an Λ_α -term as first argument and an $FSType$ as second argument. Therefore, this slightly unusual definition for equivariance (this is a workaround, see complaints).

Some /Any-Property

Assuming $eqvt(P)$ then

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

if and only if

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

Proof: Essentially you chose a fresh c , use the equivariance property of P and

$$a \# x \wedge c \# x \Rightarrow (a c) \bullet x = x$$

What I Have Done

- constructed an inductive set (Λ_α) that is bijective with the α -equated lambda-terms
- Λ_α has very much the feel of (named) lambda-terms equated up to α -equivalence
- if we can prove equivariance for the IH, then we only need to prove the abstraction-case for **one** fresh atom
- formalised Henk's proof of Church-Rosser given at the beginning of the lambda-calculus book (includes a fair deal of inductions on the structure of α -equated lambda-terms)
- formalised Girard's SN-proof from "Proof and Types" (is extremely sensitive w.r.t. substitutions and α -equivalence classes)

What Doesn't Work Yet

I have a depth function for Λ_α ; I know that it contains only $Am\ a$, $Pr\ t_1\ t_2$ and $[a].t$; I can do inductions over Λ_α , but Isabelle just does not want to let me define functions over it. :o(

I would like to define:

`consts`

`subst :: " $\Lambda_\alpha \Rightarrow atm \Rightarrow \Lambda_\alpha \Rightarrow \Lambda_\alpha$ " (" _ [_ ::= _] ")`

`primrec`

`"Am b[a ::= t] = if a=b then t else Am b"`

`"Pr s1 s2[a ::= t] = Pr (s1[a ::= t]) (s2[a ::= t])"`

`"[b].s [a ::= t] = if b#(a,t) then
 [b].(s[a ::= t]) else arbitrary"`

So far as a Relation

consts

subst :: " $(\Lambda_\alpha \times \text{atm} \times \Lambda_\alpha \times \Lambda_\alpha)$ set"

inductive subst

intros

"subst(Am a, a, t, t) \in subst"

" $a \neq b \implies$ subst(Am b, a, t, Am b) \in subst"

" $[|(s_1, a, t, r_1) \in \text{subst}; (s_2, a, t, r_2) \in \text{subst}|]$
 \implies (Pr s₁ s₂, a, t, Pr r₁ r₂) \in subst"

" $[|(s, a, t, r) \in \text{subst}; b\#(a, t)|]$
 \implies ([b].s, a, t, [b].r) \in subst"

And showed that this is a total "function"; then turned it into a function using *THE*.

Automating Equivariance

When doing an induction, one needs to show equivariance of the IH—this can be a complicated formula build up from predicates. Whether a formula is equivariant, depends on the equivariance of its components. So one can easily show that for example

$$\text{eqvt}(\forall z. P t (z, x)) \Leftrightarrow \text{eqvt}(P t (z, x))$$

holds, but because of the encoding with pairs such “rules” are not applicable: the unification in Isabelle does not unsplit pairs (possibly I am doing something stupid here—help is very much appreciated).

Dreaming

- Wouldn't it be nice to adapt the existing datatype package to allow easy definitions for functions over Λ_α ? (typedef/Rep/Abs is just too cumbersome)
- I showed all this for the lambda-calculus, but of course everything can be generalised to any datatype involving binders, I claim.
- Isabelle with its axclasses is very good positioned to provide real support for this kind of reasoning and then leap ahead from the pack. ;o)
- Even if it did not look so (that is purely my fault), it is really very easy stuff.

