Nominal Datatypes in Isabelle/HOL

Christian Urban

joint work with Stefan Berghofer and the Isabelle-team in Munich

The Nominal Logic Work

- Implemented a version of the nominal logic work as a definitional extension of Isabelle/HOL—that is as "syntactic sugar" (early nominal logic work had problems with the axiom of choice).
- Added an infrastructure for defining nominal datatypes—some of the more advanced reasoning infrastructure is however not yet in place. :o(
- My goal with this work is to make formalisations of informal "pencil-and-paper" proofs easy.

α-Equivalence Classes

Implemented Barendregt's:

<u>**Convention**</u>: Terms that are α-congruent are identified. So now we write $\lambda x.x \equiv \lambda y.y$ etcetera.

- You could define the type of lists containing only 1s, 2s and 3s from the existing type of lists.
- The polymorphic type $\alpha \times \beta$ is defined as a subset of the type $\alpha \Rightarrow \beta \Rightarrow$ bool. The pair (a, b) is represented as the function:

$$\lambda x\, y.\ (x = a) \wedge (y = b)$$

α-Equivalence Classes

Implemented Barendregt's:

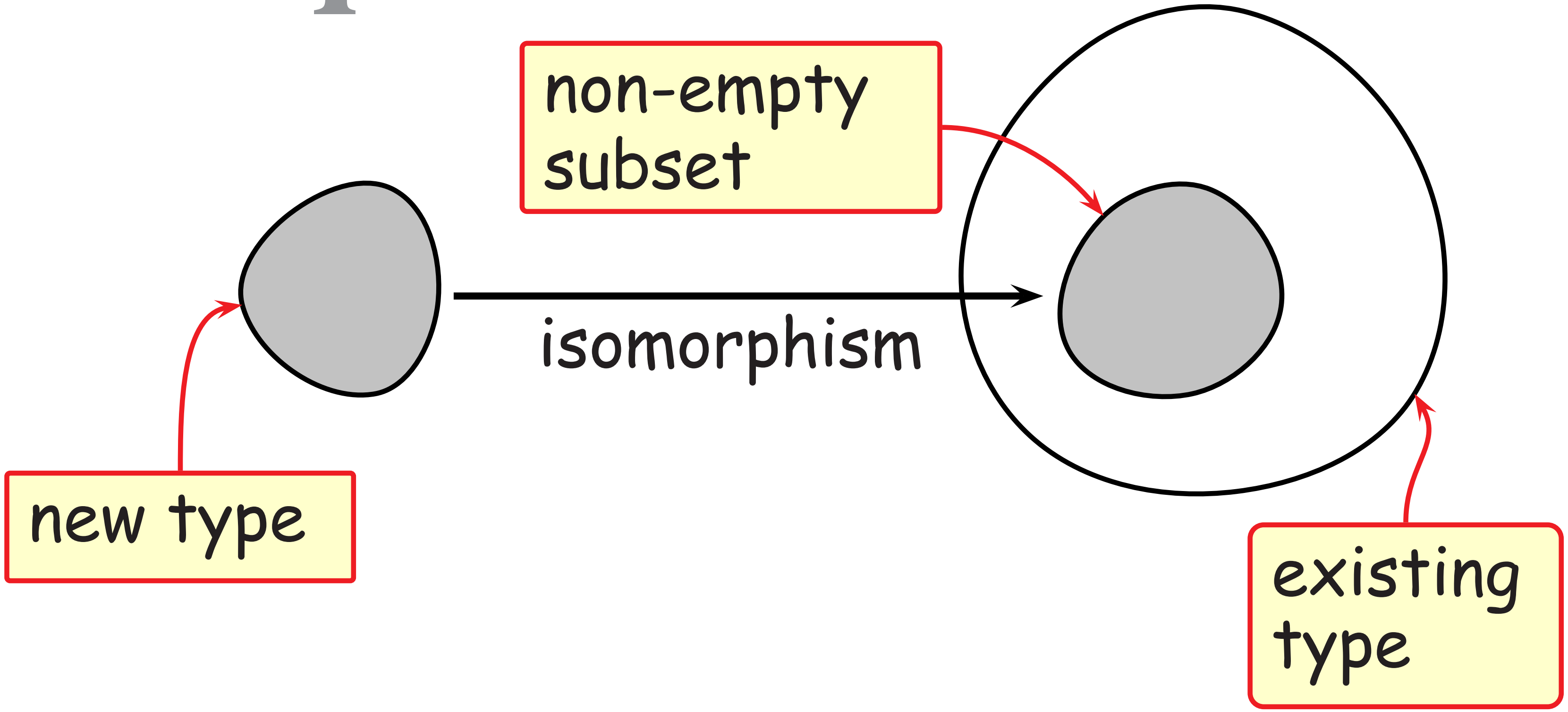
<u>**Convention**</u>: Terms that are α-congruent are identified. So now we write $\lambda x.x = \lambda y.y$ etcetera.

$=$ is equality in HOL

- You could define the type of lists containing only 1s, 2s and 3s from the existing type of lists.
- The polymorphic type $\alpha \times \beta$ is defined as a subset of the type $\alpha \Rightarrow \beta \Rightarrow$ bool. The pair (a, b) is represented as the function:

$$\lambda x\, y.\ (x = a) \wedge (y = b)$$

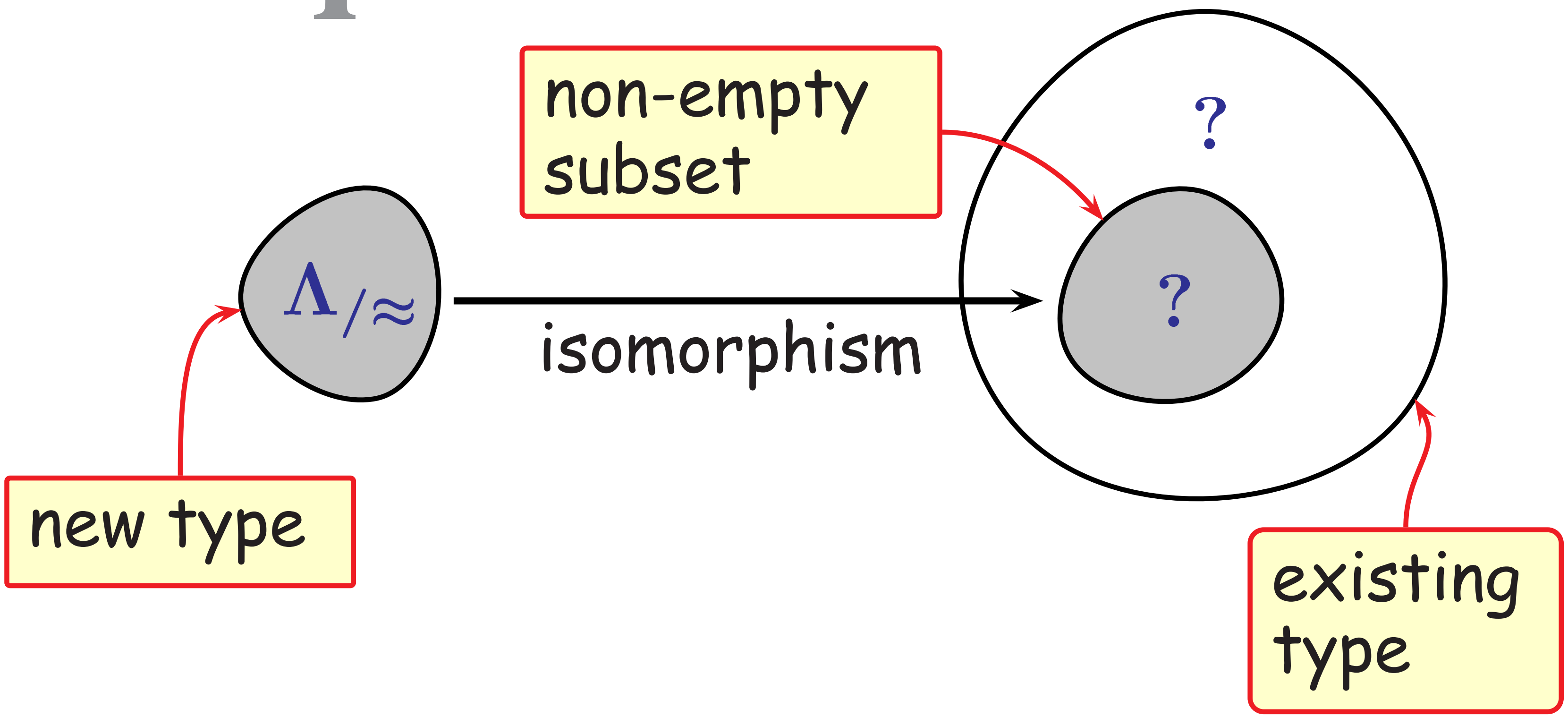
α-Equivalence Classes



- You could define the type of lists containing only 1s, 2s and 3s from the existing type of lists.
- The polymorphic type $\alpha \times \beta$ is defined as a subset of the type $\alpha \Rightarrow \beta \Rightarrow \mathsf{bool}$. The pair (a, b) is represented as the function:

$$\lambda x\, y.\ (x = a) \wedge (y = b)$$

α-Equivalence Classes

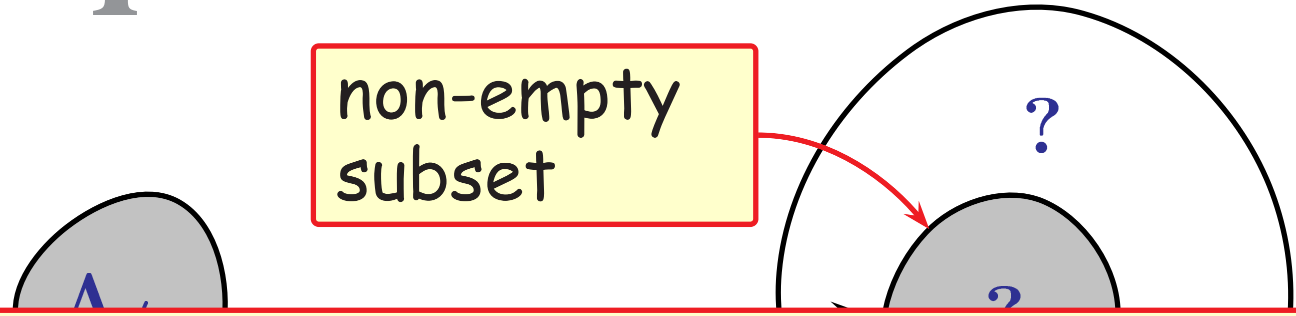


Question:

- What is a suitable existing type and a suitable subset to represent α-equivalent lambda-terms? (And α-equivalence classes in general?)
- Answer: this talk.

α-Equivalence Classes

non-empty subset



ne

There are many ways how to obtain α-equivalence classes. I will present my way (in HOL4 the answer would be different: Norrish uses a package written by Homeier).

In some sense it is a completely uninteresting question, because the user is only interested in an "interface", that is they are α-equivalence classes, but I hope you are interested in what is going on behind the scenes.

Questi

- W
  su
  (A
- Answer: this talk.

α-Equivalence

Take one step back and consider "raw" lambda-terms. The following four rules define α-equivalence:

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

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

assuming $a \neq b$

α-Equivalence

Take one step back and consider "raw" lambda-terms. The following four rules define α-equivalence:

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

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

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

Take one step back and consider "raw"

lam

α-

$\overline{\lambda}$

λ

wh

to a... oops permuted to a.

Permutations acting on lambda-terms (permutations are lists of swappings):

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

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

$$\pi\bullet(t\,s) \stackrel{\text{def}}{=} (\pi\bullet t)(\pi\bullet s)$$

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

α-Equivalence

Take one step back and consider "raw" lambda-terms. The following four rules define α-equivalence:

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

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

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

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

α-Equivalence

Take one step back and consider "raw" lambda-terms. The following four rules define α-

λa

λa

wh

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

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, i.e. $a \# s$.

α-Equivalence

Take one step back and consider "raw" lambda-terms. The following four rules define α-equivalence:

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

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

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

$$\frac{}{a \# b}\ \#\text{-var} \qquad \frac{a \# t \quad a \# s}{a \# t\, s}\ \#\text{-app}$$

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

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

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

is defined rather asymmetrically. 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

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

is de... ...ve:

(Ref...

(Sy...

And coincides with the usual definition:

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

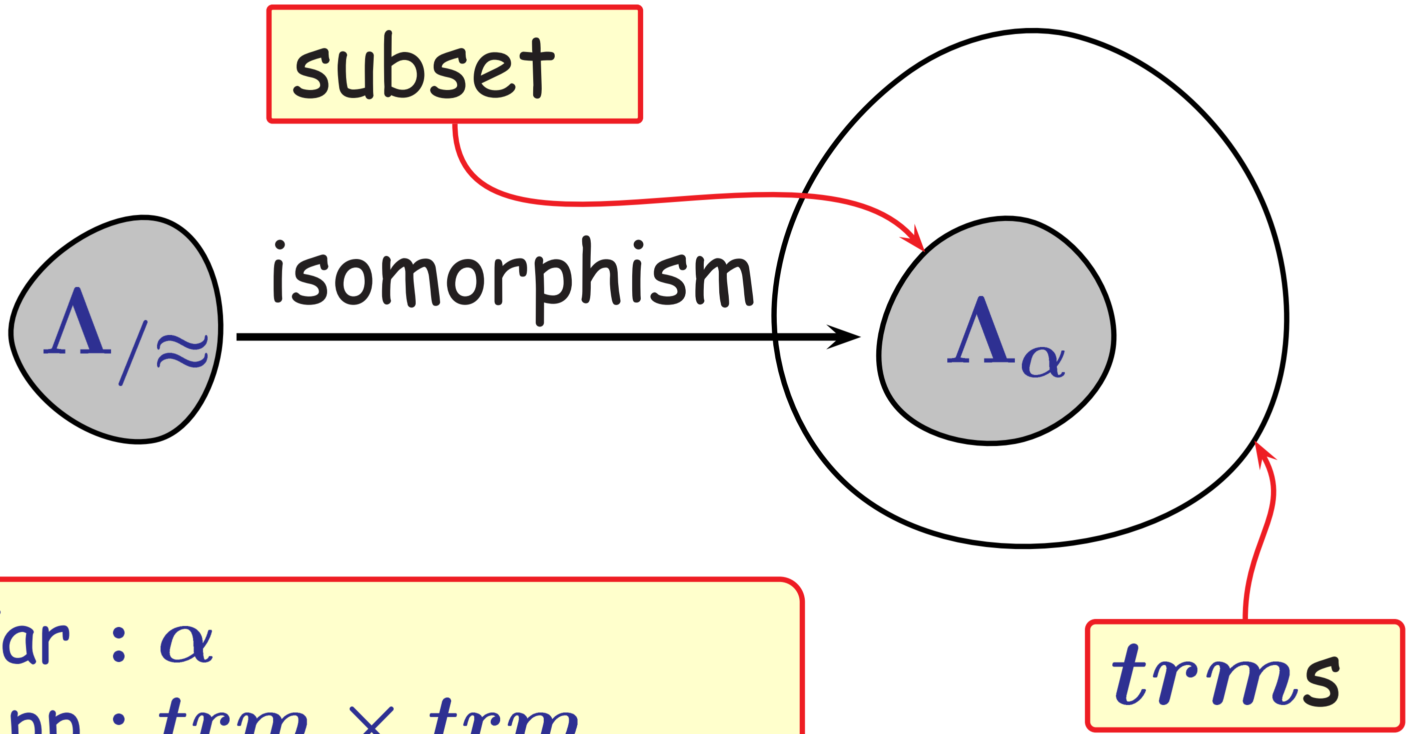
Proof left as an exercise.

(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

Weak HOAS Encoding

I defined **inductively** the α-equivalence classes of lambda-terms using some carefully chosen "partial" functions.



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

Nominal Abstractions

The functions (called nominal abstractions and written $[a].t$) satisfy:

- $\pi \bullet ([a].t) = [\pi \bullet a].(\pi \bullet t)$
- $[a].t \;=\; [b].s$ iff

  $(a = b \wedge t = s) \vee$
  $(a \neq b \wedge t = (a\,b) \bullet s \wedge a \# s)$

- btw., these two properties imply for finitely supported t

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

Nominal Abstractions

Remember the definition of α-equivalence from the beginning:

$$\frac{t \approx s}{\lambda a.t \approx \lambda a.s} \qquad \frac{a \neq b \quad t \approx (a\,b)\bullet s \quad a \# s}{\lambda a.t \approx \lambda b.s}$$

- $\pi\bullet([a].t) = [\pi\bullet a].(\pi\bullet t)$
- $[a].t \;=\; [b].s$ iff

  $(a = b \wedge t = s) \vee$
  $(a \neq b \wedge t = (a\,b)\bullet s \wedge a \# s)$

- btw., these two properties imply for finitely supported t

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

Nominal Abstractions

The functions (called nominal abstractions and written $[a].t$) satisfy:

- $\pi\bullet([a].t) = [\pi\bullet a].(\pi\bullet t)$
- $[a].t \;=\; [b].s$ iff

  $(a = b \wedge t = s) \vee$
  $(a \neq b \wedge t = (a\,b)\bullet s \wedge a \# s)$

- btw., these two properties imply for finitely supported t

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

Nominal Abstractions

Definition of support and freshness:

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

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

$$\text{supp}\ a = \{a\}$$

$$\text{supp}\ (t, s) = \text{supp}(t) \cup \text{supp}(s)$$

finitely supported t

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

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

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

type: $\alpha \rightarrow trm$ option

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

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

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

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

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

$$\begin{array}{l} (\lambda b.\,\mathsf{if}\ a = b \\ \quad \mathsf{then\ Some}(a,c) \\ \quad \mathsf{else\ if}\ b \# (a,c) \\ \qquad \mathsf{then\ Some}((b\,a)\bullet(a,c))\ \mathsf{else\ None}) \end{array}$$

Let’s check this for $[a].(a,c)$:

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

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

Let’s check this for $[a].(a,c)$:

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

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

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

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

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

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

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

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

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

- a 'applied to' $[a].(a, c)$ 'gives' $\text{Some}(a, c)$
- b 'applied to' $[a].(a, c)$ 'gives' $\text{Some}(b, c)$
- c 'applied to' $[a].(a, c)$ 'gives' None

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

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

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

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

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

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

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

$\vdots$

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

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

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

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

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

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

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

$\vdots$

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

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

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

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

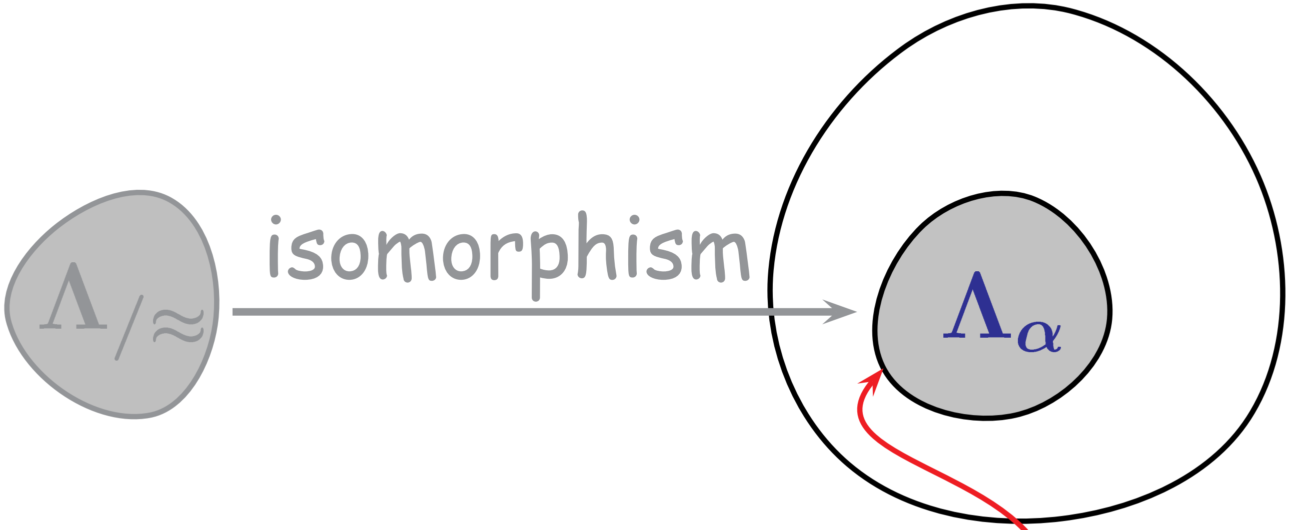
a 'applied to' $[a].(a,c)$ 'gives' Some(a,c)
b 'applied to' $[a].(a,c)$ 'gives' Some(b,c)
c 'applied to' $[a].(a,c)$ 'gives' None
d 'applied to' $[a].(a,c)$ 'gives' Some(d,c)
$\vdots$

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

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

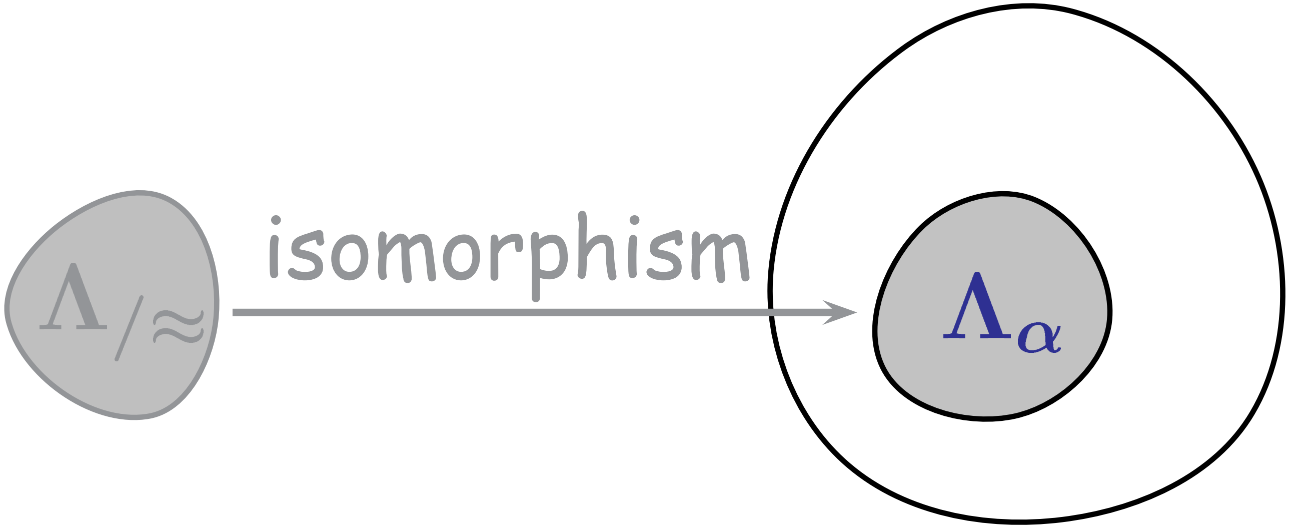
'$\lambda d.(d\,c)$'
$\vdots$

Definition of Small-Set



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

Definition of Small-Set



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

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

Definition of Small-Set

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

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

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

Definition of Small-Set

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

$$
\begin{aligned}
\mathsf{supp}(\mathsf{Var}(a)) &= \{a\} \\
\mathsf{supp}(\mathsf{App}(t,s)) &= \mathsf{supp}(t,s) \\
\mathsf{supp}(\mathsf{Lam}\,[a].t) &= \mathsf{supp}([a].t) = \mathsf{supp}(t) - \{a\}
\end{aligned}
$$

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

Isomorphism

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

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

with the property

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

"Term-Constructors"

We did not define α-equivalence classes in a constructors-based way, but we can “fake“ that: we can make the elements in the new **type** be of the form:

$$\mathsf{Var}(a),\ \mathsf{App}(s,t),\ \mathsf{Lam}\,[a].t$$

and have the following “injectivity“ properties:

$$\frac{a = b}{\mathsf{Var}(a) = \mathsf{Var}(b)} \qquad \frac{t_1 = s_1 \quad t_2 = s_2}{\mathsf{App}(t_1, t_2) = \mathsf{App}(s_1, s_2)}$$

$$\frac{t = s}{\mathsf{Lam}\,[a].t = \mathsf{Lam}\,[a].s} \qquad \frac{a \neq b \quad t = (a\,b)\bullet s \quad a \# s}{\mathsf{Lam}\,[a].t = \mathsf{Lam}\,[b].s}$$

Struct. Induction on Λ_α

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

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

Structural induction principle:

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

Substitution Lemma: If $x \not\equiv y$ and $x \notin FV(L)$, then

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]].$$

Proof: By induction on the structure of M.

- **Case 1:** M is a variable.

  Case 1.1. $M \equiv x$. Then both sides equal $N[y := L]$ since $x \not\equiv y$.

  Case 1.2. $M \equiv y$. Then both sides equal L, for $x \notin FV(L)$ implies $L[x := \ldots] \equiv L$.

  Case 1.3. $M \equiv z \not\equiv x, y$. Then both sides equal z.

- **Case 2:** $M \equiv \lambda z.M_1$. By the variable convention we may assume that $z \not\equiv x, y$ and z is not free in N, L. Then by induction hypothesis

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

- **Case 3:** $M \equiv M_1 M_2$. The statement follows again from the induction hypothesis. "$\square$"

Substitution Lemma: If $x \not\equiv y$ and $x \notin FV(L)$, then

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]].$$

Proof: By induction on the structure of M.

- **Case 1:** M is a variable.

  Case 1.1. $M \equiv x$. Then both sides equal $N[y := L]$ since $x \not\equiv y$.

  Case 1.2. $M \equiv$ ... $V(L)$

  implies

  Case 1.3. $M \equiv$

- **Case 2:** $M \equiv$ ... e may assume that $z \not\equiv x, y$ and z is not free in N, L. Then by induction hypothesis

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

- **Case 3:** $M \equiv M_1 M_2$. The statement follows again from the induction hypothesis. "$\square$"

Can we do better?

Answer: yes.

Can We Do Better?

We consider induction hypotheses of the form $P\ x\ t$ with x **finitely supported**. In the case of the substitution lemma:

$$P = \lambda(x, y, N, L).\ \ \lambda M.$$
$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]]$$
$$x = (x, y, N, L)$$

$$\frac{\begin{array}{l}\forall a\, x.\ P\ x\ (\mathsf{Var}(a)) \\ \forall t\, s\, x.\ (\forall z.\, P\ z\ t) \Longrightarrow (\forall z.\, P\ z\ s) \Longrightarrow P\ x\ (\mathsf{App}(t, s)) \\ \forall a\, t\, x.\ a \# x \Longrightarrow (\forall z.\, P\ z\ t) \Longrightarrow P\ x\ (\mathsf{Lam}\,[a].t)\end{array}}{P\ x\ t}$$

A Formal Proof in Isabelle

lemma forget:
assumes a: "$x \# L$"
shows "$L[x ::= N] = L$"
using a by (nominal_induct L avoiding: x N rule: lam_induct)
(auto simp add: abs_fresh fresh_atm)

lemma fresh_fact:
fixes x :: "name"
assumes a: "$x \# M$" and b: "$x \# N$"
shows "$x \# M[y ::= N]$"
using a b by (nominal_induct M avoiding: x y N rule: lam_induct)
(auto simp add: abs_fresh fresh_atm)

lemma subst_lemma:
assumes a: "$x \neq y$" and b: "$x \# L$"
shows "$M[x ::= N][y ::= L] = M[y ::= L][x ::= N[y ::= L]]$"
using a b by (nominal_induct M avoiding: x y N L rule: lam_induct)
(auto simp add: forget fresh_fact)

A Formal Proof in Isabelle

lemma forget:
assumes a: "$x \# L$"
shows "$L[x ::= N] = L$"
using a by (nominal_induct L avoiding: x N rule: lam_induct)
(auto simp add: abs_fresh fresh_atm)

lemma fresh_fact:
fixes x :: "nam
assumes a: "x
shows "$x \#$
using a b by (
(

lemma subst_
assumes a: "$x \neq y$" and b: "$x \# L$"
shows "$M[x ::= N][y ::= L] = M[y ::= L][x ::= N[y ::= L]]$"
using a b by (nominal_induct M avoiding: x y N L rule: lam_induct)
(auto simp add: forget fresh_fact)

abs_fresh:

$$(x \# \text{Lam}\,[y].M) = (x = y \vee x \# M)$$

fresh_atm:

$$(x \# y) = (x \neq y)$$

provided y is an atom

A Formal Proof in Isabelle

lemma forget:
assumes a: "$x \# L$"
shows "$L[x ::= N] = L$"
using a by (nominal_induct L avoiding: x N rule: lam_induct)
(auto simp add: abs_fresh fresh_atm)

lemma fresh_fact:
fixes x :: "name"
assumes a: "$x \# M$" and b: "$x \# N$"
shows "$x \# M[y ::= N]$"
using a b by (nominal_induct M avoiding: x y N rule: lam_induct)
(auto simp add: abs_fresh fresh_atm)

lemma subst_lemma:
assumes a: "$x \neq y$" and b: "$x \# L$"
shows "$M[x ::= N][y ::= L] = M[y ::= L][x ::= N[y ::= L]]$"
using a b by (nominal_induct M avoiding: x y N L rule: lam_induct)
(auto simp add: forget fresh_fact)

Weakening Lemma

Weakening Lemma: If $\Gamma_1 \vdash t : \sigma$ and Γ_2 is a valid super-context of Γ_1, then $\Gamma_2 \vdash t : \sigma$.

- **valid**: no variable has two assigned types
- **super-context**: Γ_2 has all the assignments of Γ_1

Proof: The literature is not helpful, because either too complicated[1] (if it is done right, e.g. Gallier, McKinna & Pollack, Pitts) or not helpful[2] (labelled as obvious, routine, trivial, e.g. Hindley).

[1] We can do better.
[2] I am sure they would need lots of fixing, if they were spelled out.

Formal Proof

lemma weakening_lemma:
assumes a: "$\Gamma_1 \vdash t : \sigma$" and b: "$\Gamma_1 \ll \Gamma_2$" and c: "valid Γ_2"
shows "$\Gamma_2 \vdash t : \sigma$"
using a b c
apply (nominal_induct Γ_1 t σ avoiding: Γ_2 rule: typing_induct)
apply (auto simp add: sc_def)
done

Reducibility Lemma: Let t be any term (not assumed to be reducible), and suppose all the free variables of t are among $x_1, \ldots, x_n$ of types $U_1, \ldots, U_n$. If $u_1, \ldots, u_n$ are reducible terms of types $U_1, \ldots, U_n$ then $t[x_1 := u_1, \ldots, x_n := u_n]$ is reducible.

Proof: By induction on t. We write $t[\underline{x} := \underline{u}]$ for the substitution.

- **Case 1** t is x_i:

  one has to check the tautology "if u_i is reducible, then u_i is reducible"; details are left to the reader.

- **Case 2** t is $w\,v$:

  by induction hypothesis $w[\underline{x} := \underline{u}]$ and $v[\underline{x} := \underline{u}]$ are reducible, and so (by definition) $w[\underline{x} := \underline{u}](v[\underline{x} := \underline{u}])$; but this term is nothing other than $t[\underline{x} := \underline{u}]$.

- **Case 3** t is $\lambda y.w$ of type $V \to W$:

  by induction hypothesis, $w[\underline{x} := \underline{u}, y := v]$ is reducible for all v of type V. Lemma 6.3.2 says that $t[\underline{x} := \underline{u}] = \lambda y.(w[\underline{x} := \underline{u}])$ is reducible. "$\square$"

Reducibility Lemma: Let t be any term (not assumed to be reducible), and suppose all the free variables of t are among $x_1,\ldots,x_n$ of types $U_1,\ldots,U_n$. If $u_1,\ldots,u_n$ are reducible terms of types $U_1,\ldots,U_n$ then $t[x_1 := u_1,\ldots,x_n := u_n]$ is reducible.

Proof: By induction on t. We write $t[\underline{x} := \underline{u}]$ for the substitution.

- **Case 1** t is x_i:

  one has to check the tautology "if u_i is reducible, then u_i is reducible"; details are left to the reader.

- **Case 2** t is $w\ v$:

  by induction hypothesis $w[\underline{x} := \underline{u}]$ and $v[\underline{x} := \underline{u}]$ are reducible, and so (by definition) $w[\underline{x} := \underline{u}](v[\underline{x} := \underline{u}])$; but this term is nothing other than $t[\underline{x} := \underline{u}]$.

- **Case 3** t is $\lambda y.w$ of type $V \to W$:

  by induction hypothesis, $w[\underline{x} := \underline{u}, y := v]$ is reducible for all v of type V. Lemma 6.3.2 says that $t[\underline{x} := \underline{u}] = \lambda y.(w[\underline{x} := \underline{u}])$ is reducible. "$\square$"

Lemma 6.3.2: If for all reducible u of type U, $v[x := u]$ is reducible, then so is $\lambda x.v$.

lemma all_Rd:
assumes a: "$\Gamma \vdash t : \tau$"
    and b: "$\forall x\, \sigma.\ (x,\sigma) \in \mathsf{set}\, \Gamma \longrightarrow x \in \mathsf{dom}(\theta) \wedge \theta\langle x\rangle \in Rd_\sigma$"
shows "$t[\theta] \in Rd_\tau$"
using a b **proof** (nominal_induct t avoiding: $\Gamma\ \tau\ \theta$ rule: lam_induct)
case (Lam a t)
  have ih: "$\bigwedge \Gamma\, \tau\, \theta.\ \Gamma \vdash t : \tau \Longrightarrow$

Reducibility Lemma: Let t be any term (not assumed to be reducible), and suppose all the free variables of t are among $x_1,\ldots,x_n$ of types $U_1,\ldots,U_n$. If $u_1,\ldots,u_n$ are reducible terms of types $U_1,\ldots,U_n$ then $t[x_1 := u_1, \ldots, x_n := u_n]$ is reducible.

  then obtain τ_1 τ_2 **where** τ_inst: "$\tau = \tau_1 \to \tau_2$"
    and ty: "$(a,\tau_1)\#\Gamma \vdash t : \tau_2$" **by** blast
  from ih **have** "$\forall s \in Rd_{\tau_1}. t[\theta][a ::= s] \in Rd_{\tau_2}$" **using** fr ty θ_cond
    by (force dest: fresh_context simp add: psubts_subst)
  hence "Lam $[a].(t[\theta]) \in Rd_{\tau_1 \to \tau_2}$" **by** (simp only: abs_Rd) –"Lem. 6.3.2"
thus "$(\mathsf{Lam}\ [a].t)[\theta] \in Rd_\tau$" **using** fr τ_inst **by** simp
qed (force dest!: t1_elim t2_elim)+

lemma all_Rd:
assumes a: "$\Gamma \vdash t : \tau$"
and b: "$\forall x\, \sigma.\ (x,\sigma) \in \mathsf{set}\, \Gamma \longrightarrow x \in \mathsf{dom}(\theta) \wedge \theta\langle x\rangle \in Rd_\sigma$"
shows "$t[\theta] \in Rd_\tau$"
using a b proof (nominal_induct t avoiding: $\Gamma\ \tau\ \theta$ rule: lam_induct)
case (Lam a t)
have ih: "$\bigwedge \Gamma\, \tau\, \theta.\ \Gamma \vdash t : \tau \Longrightarrow$
$(\forall x\, \sigma.\ (x,\sigma) \in \mathsf{set}\, \Gamma \longrightarrow x \in \mathsf{dom}(\theta) \wedge \theta\langle x\rangle \in Rd_\sigma) \Longrightarrow t[\theta] \in Rd_\tau$"
and θ_cond: "$\forall x\, \sigma.\ (x,\sigma) \in \mathsf{set}\, \Gamma \longrightarrow x \in \mathsf{dom}(\theta) \wedge \theta\langle x\rangle \in Rd_\sigma$"
and fr: "$a \# \Gamma$" "$a \# \theta$"
and "$\Gamma \vdash \mathsf{Lam}\,[a].t : \tau$" by fact
hence "$\exists \tau_1 \tau_2 . \tau = \tau_1 \rightarrow \tau_2 \wedge (a,\tau_1) :: \Gamma \vdash t : \tau_2$" using t3_elim fr by simp
then obtain τ_1 τ_2 where τ_inst: "$\tau = \tau_1 \rightarrow \tau_2$"
and ty: "$(a,\tau_1) \# \Gamma \vdash t : \tau_2$" by blast
from ih have "$\forall s \in Rd_{\tau_1} . t[\theta][a ::= s] \in Rd_{\tau_2}$" using fr ty θ_cond
by (force dest: fresh_context simp add: psubts_subst)
hence "$\mathsf{Lam}\,[a].(t[\theta]) \in Rd_{\tau_1 \rightarrow \tau_2}$" by (simp only: abs_Rd) –"Lem. 6.3.2"
thus "$(\mathsf{Lam}\,[a].t)[\theta] \in Rd_\tau$" using fr τ_inst by simp
qed (force dest!: t1_elim t2_elim)+