General Bindings and Alpha-Equivalence in Nominal Isabelle

Or, Nominal Isabelle 2

Christian Urban

joint work with **Cezary Kaliszyk**

Binding in Old Nominal

- the old Nominal Isabelle provided a reasoning infrastructure for single binders

$$\text{Lam }[a].(\text{Var } a)$$

for example

a # Lam [a]. t

Lam [a]. (Var a) = Lam [b]. (Var b)

Barendregt-style reasoning about bound variables

Binding in Old Nominal

- the old Nominal Isabelle provided a reasoning infrastructure for single binders

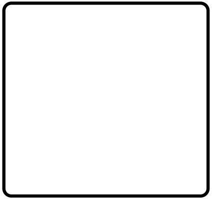
  Lam [a].(Var a)

- but representing

  $$\forall\{a_1, \ldots, a_n\}.\, T$$

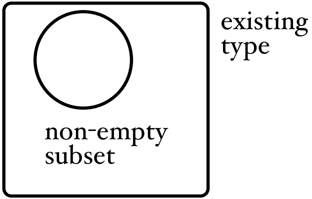
  with single binders and reasoning about it is a **major** pain; take my word for it!

New Types in HOL

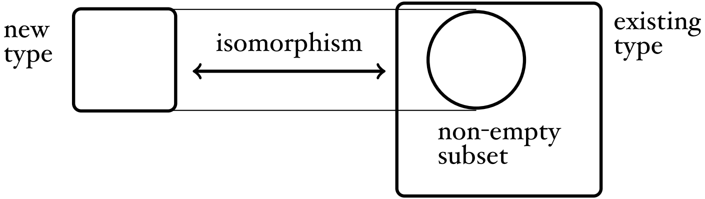


existing
type

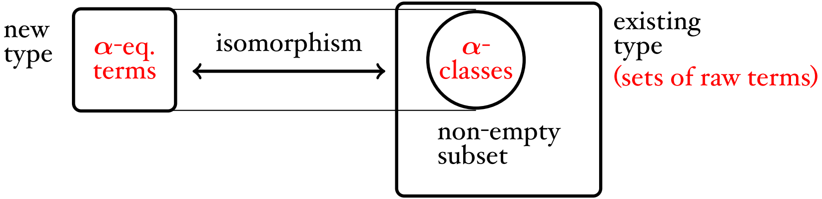
New Types in HOL



New Types in HOL



New Types in HOL

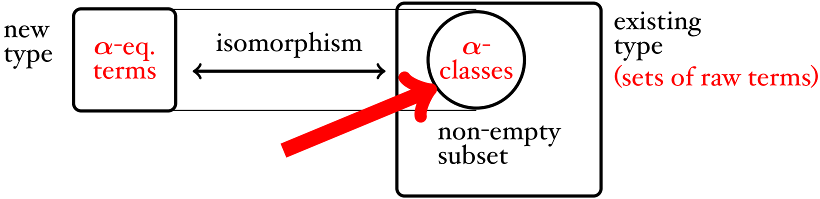


New Types in HOL

new type

$\boxed{\begin{array}{c}\alpha\text{-eq.}\\ \text{terms}\end{array}}$

New Types in HOL



define α-equivalence

Binding Sets of Names

- binding sets of names has some interesting properties:

$$\forall\{x, y\}.\, x \rightarrow y \quad \approx_\alpha \quad \forall\{y, x\}.\, y \rightarrow x$$

* x, y, z are assumed to be distinct

Binding Sets of Names

- binding sets of names has some interesting properties:

$$\forall\{x,y\}.\, x \rightarrow y \quad \approx_\alpha \quad \forall\{y,x\}.\, y \rightarrow x$$

$$\forall\{x,y\}.\, x \rightarrow y \quad \not\approx_\alpha \quad \forall\{z\}.\, z \rightarrow z$$

* x, y, z are assumed to be distinct

Binding Sets of Names

- binding sets of names has some interesting properties:

$$\forall\{x,y\}.\, x \rightarrow y \quad \approx_\alpha \quad \forall\{y,x\}.\, y \rightarrow x$$

$$\forall\{x,y\}.\, x \rightarrow y \quad \not\approx_\alpha \quad \forall\{z\}.\, z \rightarrow z$$

$$\forall\{x\}.\, x \rightarrow y \quad \approx_\alpha \quad \forall\{x,z\}.\, x \rightarrow y$$

provided z is fresh for the type

* x, y, z are assumed to be distinct

Binding Sets of Names

- bind... prop...

For type-schemes the order of bound names does not matter, and α-equivalence is preserved under vacuous binders.

$$\forall\{x, y\}.\, x \to y \quad \not\approx_\alpha \quad \forall\{z\}.\, z \to z$$

$$\forall\{x\}.\, x \to y \quad \approx_\alpha \quad \forall\{x, z\}.\, x \to y$$

provided z is fresh for the type

* x, y, z are assumed to be distinct

Other Binding Modes

- alpha-equivalence being preserved under vacuous binders is <u>not</u> always wanted:

$$\text{let } x = 3 \text{ and } y = 2 \text{ in } x - y \text{ end}$$

Other Binding Modes

- alpha-equivalence being preserved under vacuous binders is <u>not</u> always wanted:

$$\text{let } x = 3 \text{ and } y = 2 \text{ in } x - y \text{ end}$$
$$\approx_\alpha \text{let } y = 2 \text{ and } x = 3 \text{ in } x - y \text{ end}$$

Other Binding Modes

- alpha-equivalence being preserved under vacuous binders is <u>not</u> always wanted:

$$\text{let } x = 3 \text{ and } y = 2 \text{ in } x - y \text{ end}$$

$$\not\approx_\alpha \text{let } y = 2 \text{ and } x = 3 \text{ and } z = \text{loop in } x - y \text{ end}$$

Even Another Binding Mode

- sometimes one wants to abstract more than one name, but the order does matter

$$\begin{aligned}&\text{let } (x, y) = (3, 2) \text{ in } x - y \text{ end}\\ &\quad\not\approx_\alpha \text{let } (y, x) = (3, 2) \text{ in } x - y \text{ end}\end{aligned}$$

Three Binding Modes

- the order does not matter and alpha-equivelence is preserved under vacuous binders (restriction)
- the order does not matter, but the cardinality of the binders must be the same (abstraction)
- the order does matter (iterated single binders)

Three Binding Modes

- the order does not matter and alpha-equivelence is preserved under vacuous binders (restriction)
- the order does not matter, but the cardinality of the binders must be the same (abstraction)
- the order does matter (iterated single binders)

bind (set+) **bind (set)** **bind**

Specification of Binding

```
nominal_datatype trm =
    Var name
  | App trm trm
  | Lam name trm
  | Let assns trm
and assns =
    ANil
  | ACons name trm assns
```

Specification of Binding

```
nominal_datatype trm =
    Var name
  | App trm trm
  | Lam x::name t::trm      bind x in t
  | Let as::assns t::trm    bind bn(as) in t
and assns =
    ANil
  | ACons name trm assns
```

Specification of Binding

nominal_datatype trm =
    Var name
  | App trm trm
  | Lam x::name t::trm   **bind** x **in** t
  | Let as::assns t::trm   **bind** bn(as) **in** t
and assns =
    ANil
  | ACons name trm assns
binder bn **where**
    bn(ANil) = []
  | bn(ACons a t as) = [a] @ bn(as)

Alpha-Equivalence

- lets first look at pairs

$$(as, x)$$

as is a set of names...the binders
x is the body (might be a tuple)
$\approx_{\text{set}}$ is where the cardinality of the binders has to be the same

Alpha-Equivalence

- lets first look at pairs

$$(as, x) \approx_{\text{set}} (bs, y)$$

Alpha-Equivalence

- lets first look at pairs

$$(as, x) \approx_{\text{set}} (bs, y)$$

Alpha-Equivalence

- lets first look at pairs

$$(as, x) \approx_{\text{set}} (bs, y)$$

$$\stackrel{\text{def}}{=} \quad \text{fv}(x) - as = \text{fv}(y) - bs$$

Alpha-Equivalence

- lets first look at pairs

$$(as, x) \approx_{\text{set}} (bs, y)$$

$$\begin{aligned} \stackrel{\text{def}}{=} \quad & \exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ & \wedge\ \text{fv}(x) - as \ \#^{*}\ \pi \\ & \wedge\ (\pi \bullet x) = y \end{aligned}$$

Alpha-Equivalence

- lets first look at pairs

$$(as, x) \approx_{\text{set}} (bs, y)$$

$$\begin{aligned} \stackrel{\text{def}}{=} \quad & \exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ & \wedge\ \text{fv}(x) - as \ \#^*\ \pi \\ & \wedge\ (\pi \bullet x) = y \\ & \wedge\ \pi \bullet as = bs \end{aligned}$$

Alpha-Equivalence

- lets first look at pairs

$$(as, x) \approx_{\text{list}} (bs, y)$$

$$\begin{aligned} &\stackrel{\text{def}}{=} \quad \exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ &\qquad \wedge\ \text{fv}(x) - as \#^* \pi \\ &\qquad \wedge\ (\pi \bullet x) = y \\ &\qquad \wedge\ \pi \bullet as = bs \end{aligned}$$

* as and bs are lists of names

Alpha-Equivalence

- lets first look at pairs

$$(as, x) \approx_{\text{set+}} (bs, y)$$

$$
\begin{aligned}
\stackrel{\text{def}}{=} \quad & \exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\
& \wedge\ \text{fv}(x) - as \ \#^{*}\ \pi \\
& \wedge\ (\pi \bullet x) = y \\
& \cancel{\wedge\ \pi \bullet as = bs}
\end{aligned}
$$

Examples

- lets look at type-schemes:

$$(\boldsymbol{as}, \boldsymbol{x}) \approx_{\text{set}} (\boldsymbol{bs}, \boldsymbol{y})$$

Examples

- lets look at type-schemes:

$$(\boldsymbol{as}, \boldsymbol{x}) \approx_{\mathrm{set}} (\boldsymbol{bs}, \boldsymbol{y})$$

$$\begin{aligned} \mathrm{fv}(\boldsymbol{x}) &= \{\boldsymbol{x}\} \\ \mathrm{fv}(\boldsymbol{T}_1 \rightarrow \boldsymbol{T}_2) &= \mathrm{fv}(\boldsymbol{T}_1) \cup \mathrm{fv}(\boldsymbol{T}_2) \end{aligned}$$

Examples

- lets look at type-schemes:

$$(as, x) \approx_{\text{set}} (bs, y)$$

$$\begin{aligned} \text{fv}(x) &= \{x\} \\ \text{fv}(T_1 \to T_2) &= \text{fv}(T_1) \cup \text{fv}(T_2) \end{aligned}$$

set+:

$$\begin{aligned} &\exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ &\quad \wedge\ \text{fv}(x) - as \#^* \pi \\ &\quad \wedge\ \pi \cdot x = y \end{aligned}$$

set:

$$\begin{aligned} &\exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ &\quad \wedge\ \text{fv}(x) - as \#^* \pi \\ &\quad \wedge\ \pi \cdot x = y \\ &\quad \wedge\ \pi \cdot as = bs \end{aligned}$$

list:

$$\begin{aligned} &\exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ &\quad \wedge\ \text{fv}(x) - as \#^* \pi \\ &\quad \wedge\ \pi \cdot x = y \\ &\quad \wedge\ \pi \cdot as = bs \end{aligned}$$

Examples

$$(\{x, y\}, x \to y) \approx_? (\{x, y\}, y \to x)$$

- $\approx_{\text{set+}}$, $\approx_{\text{set}}$

set+:
$$\exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ \wedge\ \text{fv}(x) - as \#^* \pi \\ \wedge\ \pi \cdot x = y$$

set:
$$\exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ \wedge\ \text{fv}(x) - as \#^* \pi \\ \wedge\ \pi \cdot x = y \\ \wedge\ \pi \cdot as = bs$$

list:
$$\exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ \wedge\ \text{fv}(x) - as \#^* \pi \\ \wedge\ \pi \cdot x = y \\ \wedge\ \pi \cdot as = bs$$

Examples

$$([x, y], x \to y) \approx_? ([x, y], y \to x)$$

- $\approx_{\text{set+}}$, $\approx_{\text{set}}$, $\not\approx_{\text{list}}$

set+:
$$\exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ \wedge\ \text{fv}(x) - as \#^* \pi \\ \wedge\ \pi \cdot x = y$$

set:
$$\exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ \wedge\ \text{fv}(x) - as \#^* \pi \\ \wedge\ \pi \cdot x = y \\ \wedge\ \pi \cdot as = bs$$

list:
$$\exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ \wedge\ \text{fv}(x) - as \#^* \pi \\ \wedge\ \pi \cdot x = y \\ \wedge\ \pi \cdot as = bs$$

Examples

$$(\{x\}, x) \approx_? (\{x, y\}, x)$$

- $\approx_{\text{set+}}$, $\not\approx_{\text{set}}$, $\not\approx_{\text{list}}$

set+:
$$\exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ \wedge\ \text{fv}(x) - as \ \#^* \ \pi \\ \wedge\ \pi \cdot x = y$$

set:
$$\exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ \wedge\ \text{fv}(x) - as \ \#^* \ \pi \\ \wedge\ \pi \cdot x = y \\ \wedge\ \pi \cdot as = bs$$

list:
$$\exists \pi.\ \text{fv}(x) - as = \text{fv}(y) - bs \\ \wedge\ \text{fv}(x) - as \ \#^* \ \pi \\ \wedge\ \pi \cdot x = y \\ \wedge\ \pi \cdot as = bs$$

Examples

- $\approx_{\text{se}}$

- α-equivalences coincide when a single name is abstracted
- in that case they are equivalent to "old-fashioned" definitions of α

set+:
$$\begin{aligned}&\exists\pi.\ \text{fv}(x) - as = \text{fv}(y) - bs\\ &\quad\wedge\ \text{fv}(x) - as \#^* \pi\\ &\quad\wedge\ \pi \cdot x = y\end{aligned}$$

set:
$$\begin{aligned}&\exists\pi.\ \text{fv}(x) - as = \text{fv}(y) - bs\\ &\quad\wedge\ \text{fv}(x) - as \#^* \pi\\ &\quad\wedge\ \pi \cdot x = y\\ &\quad\wedge\ \pi \cdot as = bs\end{aligned}$$

list:
$$\begin{aligned}&\exists\pi.\ \text{fv}(x) - as = \text{fv}(y) - bs\\ &\quad\wedge\ \text{fv}(x) - as \#^* \pi\\ &\quad\wedge\ \pi \cdot x = y\\ &\quad\wedge\ \pi \cdot as = bs\end{aligned}$$

Our Specifications

```
nominal_datatype trm =
    Var name
  | App trm trm
  | Lam x::name t::trm     bind x in t
  | Let as::assns t::trm   bind bn(as) in t
and assns =
    ANil
  | ACons name trm assns
binder bn where
    bn(ANil) = []
  | bn(ACons a t as) = [a] @ bn(as)
```

Binding Functions

$$\text{Foo}\ (\boldsymbol{\lambda y.\lambda x.t})\ \ \boldsymbol{s}$$

$$\{\boldsymbol{y}, \boldsymbol{x}\}$$

Binder Clauses

- We need for a bound variable to have a 'clear scope', and bound variables should not be free and bound at the same time.

shallow binders

Lam x::name t::trm **bind** x **in** t
All xs::name set T::ty **bind** xs **in** T
Foo x::name t_1::trm t_2::trm **bind** x **in** t_1, **bind** x **in** t_2
Bar x::name t_1::trm t_2::trm **bind** x **in** t_1 t_2

Binder Clauses

- We need for a bound variable to have a 'clear scope', and bound variables should not be free and bound at the same time.

deep binders

Let as::assns t::trm **bind** bn(as) **in** t

Foo as::assns t_1::trm t_2::trm

bind bn(as) **in** t_1, **bind** bn(as) **in** t_2

×Bar as::assns t_1::trm t_2::trm

bind bn_1(as) **in** t_1, **bind** bn_2(as) **in** t_2

Binder Clauses

- We need for a bound variable to have a 'clear scope', and bound variables should not be free and bound at the same time.

deep recursive binders

Let_rec as::assns t::trm **bind** bn(as) **in** t as

×Foo_rec as::assns t_1::trm t_2::trm
bind bn(as) **in** t_1 as, **bind** bn(as) **in** t_2

Our Work

- defined fv and α

new
type

$\boxed{\begin{array}{c}\alpha\text{-eq.}\\ \text{terms}\end{array}}$

Our Work

new
type

$\boxed{\begin{array}{c}\alpha\text{-eq.}\\ \text{terms}\end{array}}$

- defined fv and α
- derived a reasoning infrastructure (#, distinctness, injectivity, cases,...)

Our Work

new type $\boxed{\alpha\text{-eq. terms}}$

- defined fv and α
- derived a reasoning infrastructure (#, distinctness, injectivity, cases,...)
- a (weak) induction principle

Our Work

new type $\boxed{\alpha\text{-eq. terms}}$

- defined fv and α
- derived a reasoning infrastructure (#, distinctness, injectivity, cases,...)
- a (weak) induction principle
- derive a **stronger** induction principle (Barendregt variable convention built in)

$$\text{Foo}\ (\lambda x.\lambda y.t)\ (\lambda u.\lambda v.s)$$

Conclusion

- the user does not see anything of the raw level

$$\text{Lam a (Var a)} = \text{Lam b (Var b)}$$

Conclusion

- the user does not see anything of the raw level
- it took quite some time to get here, but it seems worthwhile (Barendregt's variable convention is unsound in general, found bugs in two paper proofs)

Conclusion

- the user does not see anything of the raw level
- it took quite some time to get here, but it seems worthwhile (Barendregt's variable convention is unsound in general, found bugs in two paper proofs)
- http://isabelle.in.tum.de/nominal/

Questions?

Thanks!

Examples

$$(\{a,b\}, a \to b) \approx_\alpha (\{a,b\}, a \to b)$$
$$(\{a,b\}, a \to b) \approx_\alpha (\{a,b\}, b \to a)$$

$$\begin{aligned} &(\{a,b\}, (a \to b, a \to b)) \\ &\quad \not\approx_\alpha (\{a,b\}, (a \to b, b \to a)) \end{aligned}$$

Examples

$$(\{a,b\}, a \rightarrow b) \approx_\alpha (\{a,b\}, a \rightarrow b)$$
$$(\{a,b\}, a \rightarrow b) \approx_\alpha (\{a,b\}, b \rightarrow a)$$

$$(\{a,b\}, (a \rightarrow b, a \rightarrow b))$$
$$\not\approx_\alpha (\{a,b\}, (a \rightarrow b, b \rightarrow a))$$

1.) **bind (set)** as **in** τ_1, **bind (set)** as **in** τ_2

2.) **bind (set)** as **in** τ_1 τ_2