Nominal Isabelle 2

Or, How to Reason Conveniently with General Bindings

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!

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 alpha-equivalence is preserved under vacuous binders.

$\forall$

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

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

provided z is fresh for the type

* x, y, z are assumed to be distinct

Other Binding Modes

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

let $x = 3$ and $y = 2$ in $x - y$ 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 not 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

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

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 assn trm
and assn =
    ANil
  | ACons name trm assn
```

Specification of Binding

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

Specification of Binding

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

Inspiration from Ott

- this way of specifying binding is inspired by **Ott**

Inspiration from Ott

- this way of specifying binding is inspired by **Ott**, **but** we made some adjustments:
  - Ott allows specifications like

$$t ::= t\ t \mid \lambda x.t$$

Inspiration from Ott

- this way of specifying binding is inspired by **Ott**, **but** we made some adjustments:
  - whether something is bound can depend in Ott on other bound things

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

Inspiration from Ott

- this way of specifying binding is inspired by **Ott**, **but** we made some adjustments:
  - whether something is bound can depend in Ott on other bound things

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

$$\{y, x\}$$

this might make sense for "raw" terms, but not at all for α-equated terms

Inspiration from Ott

- this way of specifying binding is inspired by **Ott**, **but** we made some adjustments:
  - we allow multiple "binders" and "bodies"

    bind a b c ...**in** x y z ...
    bind (set) a b c ...**in** x y z ...
    bind (set+) a b c ...**in** x y z ...

    the reason is that with our definition of α-equivalence

    bind (set+) as **in** x y $\not\Leftrightarrow$
    bind (set+) as **in** x, **bind (set+)** as **in** y

    same with **bind (set)**

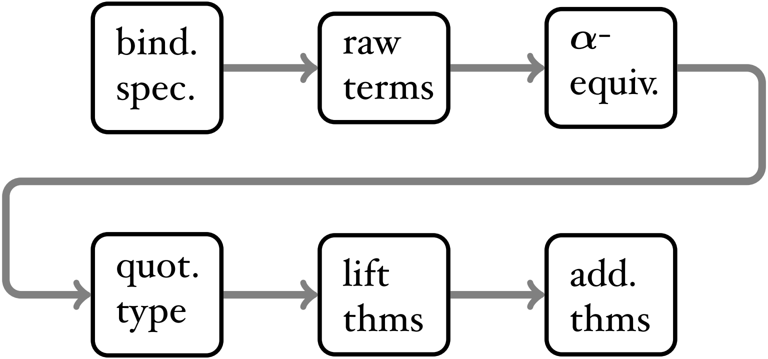
Alpha-Equivalence

- in the old Nominal Isabelle, we represented single binders as partial functions:

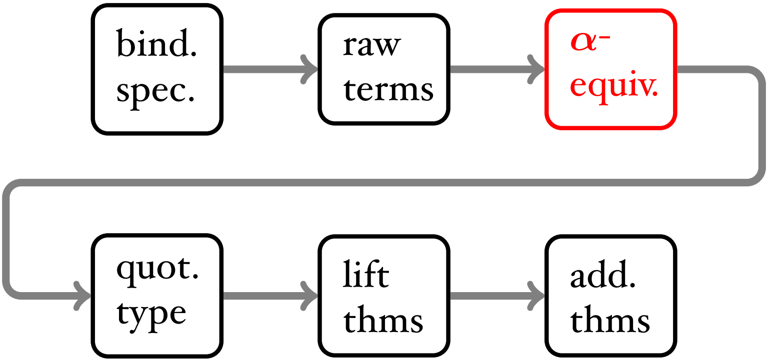
$$\begin{array}{l}\text{Lam}\,[a].\,t \;\;\text{“}\stackrel{\text{def}}{=}\text{”}\\ \quad\lambda b.\ \text{if } a = b \text{ then } t \text{ else}\\ \qquad\qquad \text{if } b \;\#\; t \text{ then } (a\ b)\bullet t \text{ else error}\end{array}$$

* alpha-equality coincides with equality on functions

New Design



New Design



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

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

Alpha-Equivalence

- lets first look at pairs

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

Alpha-Equivalence

- lets first look at pairs

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

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

Alpha-Equivalence

- lets first look at pairs

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

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

Alpha-Equivalence

- lets first look at pairs

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

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

Alpha-Equivalence

- lets first look at pairs

$$(as, x) \approx_{\color{red}{\text{list}}}^{R,\text{fv}} (bs, y)$$

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

* as and bs are lists of names

Alpha-Equivalence

- lets first look at pairs

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

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

Examples

- lets look at "type-schemes":

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

Examples

- lets look at "type-schemes":

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

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

Examples

- lets look at "type-schemes":

$$(as, x) \approx_{\mathrm{set}}^{=,\mathrm{fv}} (bs, y)$$

$$\mathrm{fv}(x) = \{x\}$$
$$\mathrm{fv}(T_1 \to T_2) = \mathrm{fv}(T_1) \cup \mathrm{fv}(T_2)$$

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

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

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

Examples

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

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

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

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

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

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

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

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

General Abstractions

- we take $(\boldsymbol{as}, \boldsymbol{x}) \approx_{*}^{=,\text{supp}} (\boldsymbol{bs}, \boldsymbol{y})$

* set, set+, list

- they are equivalence relations
- we can therefore use the quotient package to introduce the types β abs_*

$$[\boldsymbol{as}].\,\boldsymbol{x}$$

General Abstractions

- we take $(\boldsymbol{as}, \boldsymbol{x}) \approx_{*}^{=,\mathrm{supp}} (\boldsymbol{bs}, \boldsymbol{y})$

  * set, set+, list

- they are equivalence relations

- we can therefore use the quotient package to introduce the types β abs$_*$

$$\mathrm{supp}([\boldsymbol{as}].\boldsymbol{x}) = \mathrm{supp}(\boldsymbol{x}) - \boldsymbol{as}$$

General Abstractions

- we take $(as, x) \approx_{*}^{=,\text{supp}} (bs, y)$

* set, set+, list

- they are equivalence relations
- we can therefore use the quotient package to introduce the types $\beta\ \text{abs}_*$

$$
\begin{aligned}
&[as].x = [bs].y \quad \text{iff} \\
&\qquad \exists \pi.\ \text{supp}(x) - as = \text{supp}(y) - bs \\
&\qquad\quad \wedge\ \text{supp}(x) - as \mathrel{\#^*} \pi \\
&\qquad\quad \wedge\ \pi \bullet x = y \\
&\qquad (\wedge\ \pi \bullet as = bs)\ ^*
\end{aligned}
$$

A Problem

$$\text{let } \boldsymbol{x}_1 = \boldsymbol{t}_1 \ldots \boldsymbol{x}_n = \boldsymbol{t}_n \text{ in } \boldsymbol{s}$$

- we cannot represent this as

$$\text{let } [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n].\boldsymbol{s} \;\; [\boldsymbol{t}_1, \ldots, \boldsymbol{t}_n]$$

  because

$$\text{let } [\boldsymbol{x}].\boldsymbol{s} \;\; [\boldsymbol{t}_1, \boldsymbol{t}_2]$$

Our Specifications

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

“Raw” Definitions

datatype trm =
    Var name
  | App trm trm
  | Lam name trm
  | Let assn trm
and assn =
    ANil
  | ACons name trm assn

function bn **where**
    bn(ANil) = []
  | bn(ACons a t as) = [a] @ bn(as)

“Raw” Definitions

datatype trm =
    Var name
  | App trm trm
  | Lam name trm
  | Let assn trm
and assn =
    ANil
  | ACons name trm assn

+ automatically generate fv’s

function bn **where**
    bn(ANil) = []
  | bn(ACons a t as) = [a] @ bn(as)

“Raw” Alpha-Equivalence

Lam x::name t::trm **bind** x **in** t

$$\frac{([x], t) \approx_{\text{list}}^{\approx_\alpha, \text{fv}} ([x'], t')}{\text{Lam}\ x\ t \approx_\alpha \text{Lam}\ x'\ t'}\ \text{Lam-}\approx_\alpha$$

“Raw” Alpha-Equivalence

Lam x::name y::name t::trm s::trm **bind** x y **in** t s

$$\frac{([x, y], (t, s)) \approx_{\text{list}}^{R, fv} ([x', y'], (t', s'))}{\text{Lam } x\ y\ t\ s \approx_\alpha \text{Lam } x'\ y'\ t'\ s'} \text{ Lam-}\approx_\alpha$$

where $R = \approx_\alpha \times \approx_\alpha$ and $fv = \text{fv} \cup \text{fv}$

“Raw” Alpha-Equivalence

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

$$\frac{(\mathrm{bn}(\boldsymbol{as}), \boldsymbol{t}) \approx_{\mathrm{list}}^{\approx_\alpha, \mathrm{fv}} (\mathrm{bn}(\boldsymbol{as}'), \boldsymbol{t}')}{\mathrm{Let}\ \boldsymbol{as}\ \boldsymbol{t} \approx_\alpha \mathrm{Let}\ \boldsymbol{as}'\ \boldsymbol{t}'}\ \mathrm{Let}\text{-}\approx_\alpha$$

bn-function $\Rightarrow$ deep binders

"Raw" Alpha-Equivalence

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

$$\frac{(\mathrm{bn}(\boldsymbol{as}), \boldsymbol{t}) \approx_{\mathrm{list}}^{\approx_\alpha, \mathrm{fv}} (\mathrm{bn}(\boldsymbol{as}'), \boldsymbol{t}') \quad \boldsymbol{as} \approx_{\boldsymbol{\alpha}}^{\mathrm{bn}} \boldsymbol{as}'}{\mathrm{Let}\ \boldsymbol{as}\ \boldsymbol{t} \approx_\alpha \mathrm{Let}\ \boldsymbol{as}'\ \boldsymbol{t}'}\ \text{Let-}\approx_\alpha$$

bn-function $\Rightarrow$ deep binders

α for Binding Functions

...
binder bn **where**
  bn(ANil) = []
| bn(ACons a t as) = [a] @ bn(as)

$$\frac{}{\text{ANil} \approx_{\alpha}^{\text{bn}} \text{ANil}}$$

$$\frac{t \approx_{\alpha} t' \quad as \approx_{\alpha}^{bn} as'}{\text{ACons } a\ t\ as \approx_{\alpha}^{\text{bn}} \text{ACons } a'\ t'\ as'}$$

“Raw” Alpha-Equivalence

LetRec as::assn t::trm **bind** bn(as) **in** t as

$$\frac{(\mathrm{bn}(\boldsymbol{as}), (\boldsymbol{t}, \boldsymbol{as})) \approx^{R,\mathrm{fv}}_{\mathrm{list}} (\mathrm{bn}(\boldsymbol{as'}), (\boldsymbol{t'}, \boldsymbol{as'}))}{\mathrm{LetRec}\ \boldsymbol{as}\ \boldsymbol{t} \approx_\alpha \mathrm{LetRec}\ \boldsymbol{as'}\ \boldsymbol{t'}}\ \mathrm{LetRec}\text{-}{\approx_\alpha}$$

deep recursive binders

Restrictions

Our restrictions on binding specifications:

- a body can only occur once in a list of binding clauses
- you can only have one binding function for a deep binder
- binding functions can return: the empty set, singletons, unions (similarly for lists)

Automatic Proofs

- we can show that α's are equivalence relations
- as a result we can use our quotient package to introduce the type(s) of α-equated terms

$$\frac{([x], t) \approx_{\text{list}}^{=,\text{supp}} ([x'], t')}{\text{Lam } x\ t = \text{Lam } x'\ t'}$$

- the properties for support are implied by the properties of $[\_].\_$
- we can derive strong induction principles

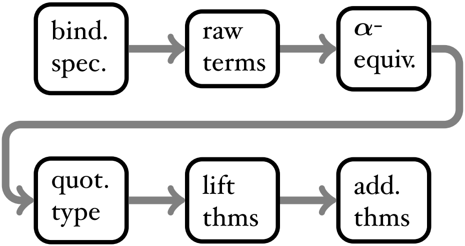
Automatic Proofs

- we can show that α's are equivalence relations
- as a result we can use our quotient package to introduce the type(s) of α-equated terms

$$\frac{[x].t = [x'].t'}{\text{Lam } x\ t = \text{Lam } x'\ t'}$$

- the properties for support are implied by the properties of $[\_].\_$
- we can derive strong induction principles

Runtime is Acceptable



- Core Haskell: 11 types, 49 term-constructors, 7 binding functions

$\sim$ 2 mins

Interesting Phenomenon

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

Should a "naked" assn be quotient?

we cannot quotient assn: ACons a ...$\not\approx_\alpha$ ACons b ...

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
- we have not yet done function definitions (will come soon and we hope to make improvements over the old way there too)

Conclusion

- the user does not see anything of the raw level
- we have not yet done function definitions (will come soon and we hope to make improvements over the old way there too)
- 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, quotient package, POPL 2011 tutorial)

Future Work

- Function definitions

Questions?

Thanks!

Examples

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

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

Examples

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

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

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

2.) **bind (set)** as **in** τ_1 τ_2