Proof Pearl: A New Foundation for Nominal Isabelle

Brian Huffman and **Christian Urban**

Nominal Isabelle

- ...is a definitional extension of Isabelle/HOL (let-polymorphism and type classes)
- ...provides a convenient reasoning infrastructure for terms involving binders (e.g. lambda calculus, variable convention)

Nominal Isabelle

- ...is a definitional extension of Isabelle/HOL (let-polymorphism and type classes)
- ...provides a convenient reasoning infrastructure for terms involving binders (e.g. lambda calculus, variable convention)
- ...mainly used to find errors in my own (published) paper proofs and in those of others **;o)**

Nominal Theory

...by Pitts; at its core are:

- sorted atoms and
- sort-respecting permutations

Nominal Theory

...by Pitts; at its core are:

- sorted atoms and
- sort-respecting permutations

$$\pi \bullet x$$

Nominal Theory

...by Pitts; at its core are:

- sorted atoms and
- sort-respecting permutations

$$\text{inv\_of\_}\pi \bullet (\pi \bullet x) \;=\; x$$

The “Old Way”

- sorted atoms
  $\mapsto$ separate types (“copies” of nat)

- sort-respecting permutations
  $\mapsto$ lists of pairs of atoms (list swappings)

The “Old Way”

- sorted atoms

  $\mapsto$ separate types (“copies” of nat)

- sort-respecting permutations

  $\mapsto$ lists of pairs of atoms (list swappings)

$$[] \bullet c = c \qquad (a\,b) :: \pi \bullet c = \begin{cases} b & \text{if } \pi \bullet c = a \\ a & \text{if } \pi \bullet c = b \\ \pi \bullet c & \text{otherwise} \end{cases}$$

The "Old Way"

- sorted atoms
  $\mapsto$ separate types ("copies" of nat)

- sort-respecting permutations
  $\mapsto$ lists of pairs of atoms (list swappings)

$$[] \bullet c = c \qquad (a\,b) :: \pi \bullet c = \begin{cases} b & \text{if } \pi \bullet c = a \\ a & \text{if } \pi \bullet c = b \\ \pi \bullet c & \text{otherwise} \end{cases}$$

The big benefit: the type system takes care of the sort-respecting requirement.

The “Old Way”

- sorted atoms
  $\mapsto$ separate types (“copies” of nat)
- sort-respecting permutations
  $\mapsto$ lists of pairs of atoms (list swappings)

$$[] \bullet c = c \qquad (a\,b) :: \pi \bullet c = \begin{cases} b & \text{if } \pi \bullet c = a \\ a & \text{if } \pi \bullet c = b \\ \pi \bullet c & \text{otherwise} \end{cases}$$

A small benefit: permutation composition is list append and permutation inversion is list reversal.

Problems

- $\_ \bullet \_ :: \alpha$ perm $\Rightarrow \beta \Rightarrow \beta$

- supp $\_ :: \beta \Rightarrow \alpha$ set

  $$\text{finite}(\text{supp } x)_{\alpha_1 \text{ set}} \ldots \text{finite}(\text{supp } x)_{\alpha_n \text{ set}}$$

- $\forall \pi_{\alpha_1} \ldots \pi_{\alpha_n} \,.\, P$

- type-classes

Problems

- $\_ \bullet \_ :: \alpha\ \text{perm} \Rightarrow \beta \Rightarrow \beta$
- $\text{supp}\ \_ :: \beta \Rightarrow \alpha\ \text{set}$

  $$\text{finite}(\text{supp}\ x)_{\alpha_1\ \text{set}} \ldots \text{finite}(\text{supp}\ x)_{\alpha_n\ \text{set}}$$

- $\forall \pi_{\alpha_1} \ldots \pi_{\alpha_n} . P$
- type-classes
  - $[] \bullet x = x$
  - $(\pi_1 @ \pi_2) \bullet x = \pi_1 \bullet (\pi_2 \bullet x)$
  - if $\pi_1 \sim \pi_2$ then $\pi_1 \bullet x = \pi_2 \bullet x$
  - if π_1, π_2 have diff. type, then $\pi_1 \bullet (\pi_2 \bullet x) = \pi_2 \bullet (\pi_1 \bullet x)$

Problems

- $\_ \bullet \_ :: \alpha \text{ perm} \Rightarrow \beta \Rightarrow \beta$

- $\text{supp}\ \_ :: \beta \Rightarrow \alpha \text{ set}$

  $$\text{finite}(\text{supp } x)_{\alpha_1 \text{ set}} \ldots \text{finite}(\text{supp } x)_{\alpha_n \text{ set}}$$

- $\forall \pi_{\alpha_1} \ldots \pi_{\alpha_n} .\ P$

- type-classes   can only have **one** type parameter
  - $[] \bullet x = x$
  - $(\pi_1 @ \pi_2) \bullet x = \pi_1 \bullet (\pi_2 \bullet x)$
  - if $\pi_1 \sim \pi_2$ then $\pi_1 \bullet x = \pi_2 \bullet x$
  - if π_1, π_2 have diff. type, then $\pi_1 \bullet (\pi_2 \bullet x) = \pi_2 \bullet (\pi_1 \bullet x)$

Problems

- $\_ \bullet \_ :: \alpha\ \text{perm} \Rightarrow \beta \Rightarrow \beta$

- supp _ :: $\beta \Rightarrow \alpha$ set

  $$\text{finite}(\text{supp}\ x)_{\alpha_1\ \text{set}} \ldots \text{finite}(\text{supp}\ x)_{\alpha_n\ \text{set}}$$

- $\forall \pi_{\alpha_1}$

- type-c
  - $[] \bullet$
  - $(\pi_1 @ \pi_2) \bullet x = \pi_1 \bullet (\pi_2 \bullet x)$
  - if $\pi_1 \sim \pi_2$ then $\pi_1 \bullet x = \pi_2 \bullet x$
  - if π_1, π_2 have diff. type, then $\pi_1 \bullet (\pi_2 \bullet x) = \pi_2 \bullet (\pi_1 \bullet x)$

- *lots of ML-code*
- *not pretty*
- *not a* **proof pearl** *:o(*

A Better Way

datatype atom = Atom string nat

A Better Way

datatype atom = Atom string nat

- permutations are (restricted) bijective functions from atom $\Rightarrow$ atom
  - sort-respecting $(\forall a.\ \mathrm{sort}(\pi a) = \mathrm{sort}(a))$
  - finite domain $(\mathrm{finite}\{a.\ \pi a \neq a\})$

$$\_ \bullet \_ :: \mathrm{perm} \Rightarrow \beta \Rightarrow \beta$$

A Better Way

datatype atom = Atom string nat

- permutations are (restricted) bijective functions from atom $\Rightarrow$ atom
  - sort-respecting $(\forall a.\ \mathrm{sort}(\pi a) = \mathrm{sort}(a))$
  - finite domain $(\mathrm{finite}\{a.\ \pi a \neq a\})$

- What about **swappings**?

$$
\begin{aligned}
(a\ b) \overset{\mathrm{def}}{=}\ & \text{if } \mathrm{sort}(a) = \mathrm{sort}(b) \\
& \text{then } \lambda c.\text{if } a = c \text{ then } b \text{ else if } b = c \text{ then } a \text{ else } c \\
& \text{else } ?
\end{aligned}
$$

A Better Way

datatype atom = Atom string nat

- permutations are (restricted) bijective functions from atom $\Rightarrow$ atom
  - sort-respecting $(\forall a.\ \text{sort}(\pi a) = \text{sort}(a))$
  - finite domain $(\text{finite}\{a.\ \pi a \neq a\})$
- What about **swappings**?

$$\begin{aligned}(a\ b) \stackrel{\text{def}}{=} {} & \text{if sort}(a) = \text{sort}(b) \\ & \text{then } \lambda c.\text{if } a = c \text{ then } b \text{ else if } b = c \text{ then } a \text{ else } c \\ & \text{else } \mathbf{id}\end{aligned}$$

A Smoother Nominal Theory

From there it is essentially plain sailing:

A Smoother Nominal Theory

From there it is essentially plain sailing:

- $(\boldsymbol{a}\ \boldsymbol{b}) = (\boldsymbol{b}\ \boldsymbol{a})$

A Smoother Nominal Theory

From there it is essentially plain sailing:

- $(\boldsymbol{a}\ \boldsymbol{b}) = (\boldsymbol{b}\ \boldsymbol{a})$
- permutations are an instance of Isabelle's group_add $(0, \boldsymbol{\pi}_1 + \boldsymbol{\pi}_2, -\boldsymbol{\pi})$

A Smoother Nominal Theory

From there it is essentially plain sailing:

- $(\boldsymbol{a}\ \boldsymbol{b}) = (\boldsymbol{b}\ \boldsymbol{a}) = (\boldsymbol{a}\ \boldsymbol{c}) + (\boldsymbol{b}\ \boldsymbol{c}) + (\boldsymbol{a}\ \boldsymbol{c})$
- permutations are an instance of Isabelle's group_add (0, $\boldsymbol{\pi}_1 + \boldsymbol{\pi}_2$, $-\boldsymbol{\pi}$)

A Smoother Nominal Theory

From there it is essentially plain sailing:

- $(\boldsymbol{a}\ \boldsymbol{b}) = (\boldsymbol{b}\ \boldsymbol{a}) = (\boldsymbol{a}\ \boldsymbol{c}) + (\boldsymbol{b}\ \boldsymbol{c}) + (\boldsymbol{a}\ \boldsymbol{c})$
- permutations are an instance of Isabelle's group_add $(0, \boldsymbol{\pi}_1 + \boldsymbol{\pi}_2, -\boldsymbol{\pi})$

This is slightly odd, since in general:

$$\boldsymbol{\pi}_1 + \boldsymbol{\pi}_2 \neq \boldsymbol{\pi}_2 + \boldsymbol{\pi}_1$$

A Smoother Nominal Theory

From there it is essentially plain sailing:

- $(\boldsymbol{a}\ \boldsymbol{b}) = (\boldsymbol{b}\ \boldsymbol{a}) = (\boldsymbol{a}\ \boldsymbol{c}) + (\boldsymbol{b}\ \boldsymbol{c}) + (\boldsymbol{a}\ \boldsymbol{c})$
- permutations are an instance of Isabelle's group_add $(0, \boldsymbol{\pi}_1 + \boldsymbol{\pi}_2, -\boldsymbol{\pi})$
- $\_ \bullet \_ :: \text{perm} \Rightarrow \boldsymbol{\alpha} \Rightarrow \boldsymbol{\alpha}$
  - $0 \bullet \boldsymbol{x} = \boldsymbol{x}$
  - $(\boldsymbol{\pi}_1 + \boldsymbol{\pi}_2) \bullet \boldsymbol{x} = \boldsymbol{\pi}_1 \bullet (\boldsymbol{\pi}_2 \bullet \boldsymbol{x})$

A Smoother Nominal Theory

From there it is essentially plain sailing:

- $(a\ b) = (b\ a) = (a\ c) + (b\ c) + (a\ c)$
- permutations are an instance of Isabelle's group_add $(0, \pi_1 + \pi_2, -\pi)$
- $\_ \bullet \_ ::$ perm $\Rightarrow \alpha \Rightarrow \alpha$
  - $0 \bullet x = x$
  - $(\pi_1 + \pi_2) \bullet x = \pi_1 \bullet (\pi_2 \bullet x)$

$\mapsto$ only one type class needed, finite(supp x), $\forall \pi . P$

One Snatch

```
datatype atom = Atom string nat
```

- You like to get the advantages of the old way back: you cannot mix atoms of different sort:

  e.g. LF-objects:

$$\boldsymbol{M ::= c \mid x \mid \lambda x : A.M \mid M_1\ M_2}$$

Our Solution

- <u>concrete</u> atoms:

typedef name = ”{a :: atom. sort a = ”name”}”
typedef ident = ”{a :: atom. sort a = ”ident”}”

- they are a “subtype” of the generic atom type
- there is an overloaded function **atom**, which injects concrete atoms into generic ones

$$\text{atom}(\boldsymbol{a}) \ \# \ \boldsymbol{x}$$
$$(\boldsymbol{a} \leftrightarrow \boldsymbol{b}) \overset{\text{def}}{=} (\text{atom}(\boldsymbol{a}) \quad \text{atom}(\boldsymbol{b}))$$

Our Solution

- concrete atoms:

```
typedef name = "{a :: atom. sort a = ''name''}"
typedef ident = "{a :: atom. sort a = ''ident''}"
```

- they are a "subtype" of the generic atom type
- there is an overloaded function **atom**, which injects concrete atoms into generic ones

$$\begin{array}{l} \text{atom}(\boldsymbol{a}) \;\#\; \boldsymbol{x} \\ (\boldsymbol{a} \leftrightarrow \boldsymbol{b}) \overset{\text{def}}{=} (\text{atom}(\boldsymbol{a}) \;\; \text{atom}(\boldsymbol{b})) \end{array}$$

One would like to have $\boldsymbol{a} \;\#\; \boldsymbol{x}$, $(\boldsymbol{a}\,\boldsymbol{b})$, ...

Sorts Reloaded

datatype atom = Atom string nat

Sorts Reloaded

datatype atom = Atom string nat

Problem: HOL-binders or Church-style lambda-terms

$$\lambda x_\alpha . x_\alpha \; x_\beta$$

Sorts Reloaded

datatype atom = Atom string nat

Problem: HOL-binders or Church-style lambda-terms

$$\boldsymbol{\lambda x_\alpha . x_\alpha\ x_\beta}$$

datatype ty = TVar string | ty $\rightarrow$ ty
datatype var = Var name ty

Sorts Reloaded

datatype atom = Atom string nat

Problem: HOL-binders or Church-style lambda-terms

$$\lambda x_\alpha .\, x_\alpha\ x_\beta$$

datatype ty = TVar string | ty $\rightarrow$ ty
datatype var = Var name ty
$(x \leftrightarrow y) \bullet (x_\alpha, x_\beta) = (y_\alpha, y_\beta)$

Non-Working Solution

Instead of

datatype atom = Atom string nat

have

datatype ’a atom = Atom ’a nat

Non-Working Solution

Instead of

datatype atom = Atom string nat

have

datatype 'a atom = Atom 'a nat

But then

$$\_ \bullet \_ :: \alpha\ \text{perm} \Rightarrow \beta \Rightarrow \beta$$

A Working Solution

```
datatype sort = Sort string "sort list"
datatype atom = Atom sort nat
```

A Working Solution

datatype sort = Sort string "sort list"
datatype atom = Atom sort nat

$$\text{sort\_ty}\ (\text{TVar}\ x) \overset{\text{def}}{=} \text{Sort "TVar" [Sort}\ x\ [\,]]$$
$$\text{sort\_ty}\ ({}_1 \Rightarrow {}_2) \overset{\text{def}}{=} \text{Sort "Fun" [sort\_ty}\ {}_1,\ \text{sort\_ty}\ {}_2]$$

A Working Solution

datatype sort = Sort string ”sort list”
datatype atom = Atom sort nat

$$\text{sort\_ty (TVar x)} \overset{\text{def}}{=} \text{Sort ”TVar” [Sort x []]}$$
$$\text{sort\_ty} \ ({}_1 \Rightarrow {}_2) \overset{\text{def}}{=} \text{Sort ”Fun” [sort\_ty} \ {}_1\text{, sort\_ty} \ {}_2\text{]}$$

typedef var = {a :: atom. sort a ∈ range sort_ty}

A Working Solution

datatype sort = Sort string "sort list"
datatype atom = Atom sort nat

sort_ty (TVar x) $\overset{\text{def}}{=}$ Sort "TVar" [Sort x []]
sort_ty ($_1 \Rightarrow {}_2$) $\overset{\text{def}}{=}$ Sort "Fun" [sort_ty $_1$, sort_ty $_2$]

typedef var = {a :: atom. sort a $\in$ range sort_ty}

Var x $\overset{\text{def}}{=}$ $\lceil$ Atom (sort_ty) x $\rceil$

(Var x $\leftrightarrow$ Var y) • Var x = Var y
(Var x $\leftrightarrow$ Var y) • Var x ' = Var x '

Conclusion

- the formalised version of the nominal theory is now much nicer to work with (sorts are occasionally explicit, $\forall\pi.P$)
- permutations: "be as abstract as you can" (group_add is a slight oddity)
- the crucial insight: allow sort-disrespecting swappings

Conclusion

- the formalised version of the nominal theory is now much nicer to work with (sorts are occasionally explicit, $\forall\pi.P$)
- permutations: “be as abstract as you can” (group_add is a slight oddity)
- the crucial insight: allow sort-disrespecting swappings ...just define them as the identity

Conclusion

- the formalised version of the nominal theory is now much nicer to work with (sorts are occasionally explicit, $\forall\pi.P$)
- permutations: "be as abstract as you can" (group_add is a slight oddity)
- the crucial insight: allow sort-disrespecting swappings ...just define them as the identity (a referee called this a "hack")

Conclusion

- the formalised version of the nominal theory is now much nicer to work with (sorts are occasionally explicit, $\forall\pi.P$)
- permutations: “be as abstract as you can” (group_add is a slight oddity)
- the crucial insight: allow sort-disrespecting swappings ...just define them as the identity (a referee called this a “hack”)
- there will be a hands-on tutorial about Nominal Isabelle at POPL’11 in Austin Texas

Thank you very much

Questions?