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

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

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

Problems

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

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

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

- $\_ \bullet \_ :: \alpha$ 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 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

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

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

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

- $\forall \pi_{\alpha_1}$

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

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

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 $(\forall a.\ \mathsf{sort}(\pi a) = \mathsf{sort}(a))$
  - finite domain $(\mathsf{finite}\{a.\ \pi a \neq a\})$

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

A Better Way

datatype atom = Atom string nat

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

- What about **swappings?**

$$(a\ b) \stackrel{\mathsf{def}}{=} \mathsf{if}\ \mathsf{sort}(a) = \mathsf{sort}(b)$$
$$\mathsf{then}\ \lambda c.\mathsf{if}\ a = c\ \mathsf{then}\ b\ \mathsf{else}\ \mathsf{if}\ b = c\ \mathsf{then}\ a\ \mathsf{else}\ c$$
$$\mathsf{else}\ ?$$

A Better Way

datatype atom = Atom string nat

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

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

A Smoother Nominal Theory

From there it is essentially plain sailing:

A Smoother Nominal Theory

From there it is essentially plain sailing:

- $(a\ b) = (b\ a)$

A Smoother Nominal Theory

From there it is essentially plain sailing:

- $(a\ b) = (b\ a)$
- permutations are an instance of Isabelle's group_add (0, $\pi_1 + \pi_2$, $-\pi$)

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, $\pi_1 + \pi_2$, $-\pi$)

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, $\pi_1 + \pi_2$, $-\pi$)

This is slightly odd, since in general:

$$\pi_1 + \pi_2 \neq \pi_2 + \pi_1$$

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 ($\mathbf{0}$, $\pi_1 + \pi_2$, $-\pi$)
- _ • _ :: perm $\Rightarrow \alpha \Rightarrow \alpha$
  - $\mathbf{0} \bullet x = x$
  - $(\pi_1 + \pi_2) \bullet x = \pi_1 \bullet (\pi_2 \bullet x)$

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:

$$M ::= c \mid x \mid \lambda x\!:\!A.M \mid M_1\ M_2$$

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

$$\text{atom}(a)\ \#\ x$$

$$(a \leftrightarrow b) \stackrel{\text{def}}{=} (\text{atom}(a) \quad \text{atom}(b))$$

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

$$\text{atom}(a) \;\#\; x$$
$$(a \leftrightarrow b) \stackrel{\text{def}}{=} (\text{atom}(a)\;\; \text{atom}(b))$$

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

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

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

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

$$\text{sort\_ty}\ (\text{TVar}\ x) \stackrel{\text{def}}{=} \text{Sort}\ \text{"TVar"}\ [\text{Sort}\ x\ []]$$

$$\text{sort\_ty}\ (\tau_1 \rightarrow \tau_2) \stackrel{\text{def}}{=} \text{Sort}\ \text{"Fun"}\ [\text{sort\_ty}\ \tau_1, \text{sort\_ty}\ \tau_2]$$

A Working Solution

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

$$\text{sort\_ty}\ (\text{TVar}\ x) \stackrel{\text{def}}{=} \text{Sort}\ \text{''TVar''}\ [\text{Sort}\ x\ []]$$

$$\text{sort\_ty}\ (\tau_1 \rightarrow \tau_2) \stackrel{\text{def}}{=} \text{Sort}\ \text{''Fun''}\ [\text{sort\_ty}\ \tau_1, \text{sort\_ty}\ \tau_2]$$

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

A Working Solution

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

$$\text{sort\_ty (TVar x)} \overset{\text{def}}{=} \text{Sort "TVar" [Sort x []]}$$
$$\text{sort\_ty} \ (\tau_1 \rightarrow \tau_2) \overset{\text{def}}{=} \text{Sort "Fun" [sort\_ty } \tau_1\text{, sort\_ty } \tau_2\text{]}$$

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

$$\text{Var x } \tau \overset{\text{def}}{=} \lceil \text{Atom (sort\_ty } \tau\text{) x} \rceil$$

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

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?