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

\[ \pi \cdot \mathcal{X} \]
Nominal Theory

... by Pitts; at its core are:

- sorted atoms and
- sort-respecting permutations

\[ \text{inv}_\text{of}_\pi \cdot (\pi \cdot x) = x \]
The “Old Way”

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

- sort-respecting permutations
  - \(\rightarrow\) lists of pairs of atoms (list swappings)
The “Old Way”

- sorted atoms
  \[\rightarrow \text{separate types} \ ("copies" \ of \ \text{nat})\]

- sort-respecting permutations
  \[\rightarrow \text{lists of pairs of atoms (list swappings)}\]

\[
\begin{align*}
\emptyset \cdot c &= c \\
(a \ b) :: \pi \cdot c &= \begin{cases} 
b & \text{if } \pi \cdot c = a \\
a & \text{if } \pi \cdot c = b \\
\pi \cdot c & \text{otherwise}
\end{cases}
\end{align*}
\]
The “Old Way”

- sorted atoms
  \[ \mapsto \text{separate types} \quad (\text{“copies” of } \text{nat}) \]

- sort-respecting permutations
  \[ \mapsto \text{lists of pairs of atoms (list swappings)} \]

\[
\begin{align*}
[] \cdot c &= c \\
(a \ b) :: \pi \cdot c &= \begin{cases} 
b & \text{if } \pi \cdot c = a \\
a & \text{if } \pi \cdot c = b \\
\pi \cdot c & \text{otherwise}
\end{cases}
\end{align*}
\]

The big benefit: the type system takes care of the sort-respecting requirement.
The “Old Way”

- sorted atoms
  \[\rightarrow\text{separate types ("copies" of nat)}\]

- sort-respecting permutations
  \[\rightarrow\text{lists of pairs of atoms (list swappings)}\]

\[
\begin{align*}
\cdot &\cdot c = c \\
(a \ b) :: \pi \cdot c = \\
&\begin{cases}
  b & \text{if } \pi \cdot c = a \\
  a & \text{if } \pi \cdot c = b \\
  \pi \cdot c & \text{otherwise}
\end{cases}
\end{align*}
\]

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

- _ • _ :: \( \alpha \) perm \( \Rightarrow \beta \Rightarrow \beta \)

- supp _ :: \( \beta \Rightarrow \alpha \) set

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

- \( \forall \pi_{\alpha_1} \ldots \pi_{\alpha_n} \cdot P \)

- type-classes
Problems

_ · _ :: α perm ⇒ β ⇒ β

supp _ :: β ⇒ α set

finite(supp x) α_1 set ... finite(supp x) α_n set

∀ π_ α_1 ... π_ α_n . P

type-classes

[] · x = x

(π_1 @ π_2) · x = π_1 · (π_2 · x)

if π_1 ∼ π_2 then π_1 · x = π_2 · x

if π_1, π_2 have different type, then π_1 · (π_2 · x) = π_2 · (π_1 · x)
Problems

- \_ \cdot \_ :: \alpha \text{ perm} \Rightarrow \beta \Rightarrow \beta

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

\quad \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

- \text{type-classes \ can only have one type parameter}
  - [] \cdot x = x
  - (\pi_1 \circ \pi_2) \cdot x = \pi_1 \cdot (\pi_2 \cdot x)
  - if \pi_1 \sim \pi_2 then \pi_1 \cdot x = \pi_2 \cdot x
  - if \pi_1, \pi_2 \text{ have diff. type, then } \pi_1 \cdot (\pi_2 \cdot x) = \pi_2 \cdot (\pi_1 \cdot x)
Problems

- _ · _ :: \( \alpha \) perm \( \Rightarrow \beta \) \( \Rightarrow \beta \)

- supp _ :: \( \beta \) \( \Rightarrow \alpha \) set

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

- \( \forall \pi_{\alpha_1} \) type-class

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

- [] · x = x
- \((\pi_1 \@ \pi_2) \cdot x = \pi_1 \cdot (\pi_2 \cdot x)\)
- if \( \pi_1 \sim \pi_2 \) then \( \pi_1 \cdot x = \pi_2 \cdot x \)
- if \( \pi_1, \pi_2 \) have diff. type, then \( \pi_1 \cdot (\pi_2 \cdot x) = \pi_2 \cdot (\pi_1 \cdot x) \)
datatype atom = Atom string nat
A Better Way

datatype atom = Atom string nat

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

\_ \_ \_ \_ \_ \_ :: \text{perm} ⇒ β ⇒ β
A Better Way

datatype atom = Atom string nat

permutations are (restricted) bijective functions from atom ⇒ atom

- sort-respecting \((\forall a. \text{sort}(\pi a) = \text{sort}(a))\)
- finite domain \((\text{finite}\{a. \pi a \neq a\})\)

What about swappings?

\[(a \ b) \overset{\text{def}}{=} \text{if sort}(a) = \text{sort}(b)\]
\[\quad \text{then } \lambda c. \text{if } a = c \text{ then } b \text{ else if } b = c \text{ then } a \text{ else } c\]
\[\quad \text{else } ?\]
A Better Way

datatype atom = Atom string nat

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

- What about swappings?

\[(a \ b) \overset{\text{def}}{=} \text{if sort}(a) = \text{sort}(b) \]
\[
\quad \text{then } \lambda c. \text{if } a = c \text{ then } b \text{ else if } b = c \text{ then } a \text{ else } c
\]
\[
\quad \text{else } \text{id}
\]
A Smoother Nominal Theory

From there it is essentially plain sailing:
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 (0, π₁ + π₂, −π)`

- \[\_ \cdot \_ :: \text{perm} \Rightarrow \alpha \Rightarrow \alpha\]
  - \[0 \cdot x = x\]
  - \[(π₁ + π₂) \cdot x = π₁ \cdot (π₂ \cdot x)\]
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 \((0, \pi_1 + \pi_2, -\pi)\)

- `_ · _ :: perm ⇒ α ⇒ α`

- \(0 \cdot x = x\)
- \((\pi_1 + \pi_2) \cdot x = \pi_1 \cdot (\pi_2 \cdot x)\)

→ only one type class needed, finite(supp \(x\)), \(∀\pi.\ P\)
\textbf{One Snatch}

\texttt{datatype} \ atom = Atom \ string \ nat

\begin{itemize}
  \item You like to get the advantages of the old way back: you \textbf{cannot mix} atoms of different sort:
  \end{itemize}

\begin{itemize}
  \item e.g. LF-objects:
  \end{itemize}

\[
M ::= c \mid x \mid \lambda x : A. M \mid M_1 \ M_2
\]
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

  ```
  atom(a) \# x
  (a \leftrightarrow b)^{\text{def}} = (atom(a) \land 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

  ```
  atom(a) ≠ x
  (a ↔ b) ≜ (atom(a) atom(b))  
  One would like to have a ≠ x, (a b), ...  
  ```
Sorted Re-loaded

datatype atom = Atom string nat
Sorts Reloaded

datatype atom = Atom string nat

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

\[ \lambda x_\alpha \cdot x_\alpha \, x_\beta \]
Sorts Reloaded

**datatype** atom = Atom string nat

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

$$\lambda x_\alpha \cdot x_\alpha \ x_\beta$$

**datatype** ty = TVar string | ty → ty

**datatype** var = Var name ty
**Sorts Reloaded**

**datatype** atom = Atom string nat

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

\[ \lambda x^\alpha \cdot x^\alpha \, x^\beta \]

**datatype** ty = TVar string | ty → ty

**datatype** var = Var name ty

\[(x \leftrightarrow y) \cdot (x^\alpha, x^\beta) = (y^\alpha, y^\beta)\]
Non-Working Solution

Instead of

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

have

```plaintext
datatype 'a atom = Atom 'a nat
```
Non-Working Solution

Instead of

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

have

```plaintext
datatype 'a atom = Atom 'a nat
```

But then

```plaintext
_ · _ :: α perm ⇒ β ⇒ β
```


A Working Solution

```plaintext
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

sort_ty (TVar x) def = Sort "TVar" [Sort x []]
sort_ty (τ₁ → τ₂) def = Sort "Fun" [sort_ty τ₁, sort_ty τ₂]
A Working Solution

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

\[
\begin{align*}
\text{sort}_\text{ty} (\text{TVar } x) & \overset{\text{def}}{=} \text{Sort } "\text{TVar}" [\text{Sort } x \ [\ ]] \\
\text{sort}_\text{ty} (\tau_1 \rightarrow \tau_2) & \overset{\text{def}}{=} \text{Sort } "\text{Fun}" [\text{sort}_\text{ty} \tau_1, \text{sort}_\text{ty} \tau_2]
\end{align*}
\]

typedef var = \{a :: atom. sort a \in \text{range sort}_\text{ty}\}
A Working Solution

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

sort_ty (TVar x) ndef= Sort "TVar" [Sort x []]
sort_ty (τ₁ → τ₂) ndef= Sort "Fun" [sort_ty τ₁, sort_ty τ₂]

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

Var x τ ndef= [ Atom (sort_ty τ) x ]

(Var x τ ↔ Var y τ) • Var x τ = Var y τ
(Var x τ ↔ Var y τ) • Var x τ' = Var x τ'
```

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
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
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?