Nominal Datatypes in Isabelle/HOL

Stefan Berghofer
with
Christian Urban, Julien Narboux and Markus Wenzel
Motivation
Substitution lemma: If $x \neq y$ and $x \notin FV(L)$, then
$$M[x \mapsto N][y \mapsto L] = M[y \mapsto L][x \mapsto N[y \mapsto L]].$$

Proof: By induction on the structure of $M$.

Case 1: $M$ is a variable.

Case 1.1. $M = x$. Then both sides equal $N[y \mapsto L]$, since $x \neq y$.

Case 1.2. $M = y$. Then both sides equal $L$, for $x \notin FV(L)$ implies $L[x \mapsto \ldots] = L$.

Case 1.3. $M = z \neq x, y$. Then both sides equal $z$.

Case 2: $M = \lambda z. M_1$. By the variable convention we may assume that $z \neq x, y$ and $z$ is not free in $N, L$. Then by induction hypothesis

$$\begin{align*}
(\lambda z. M_1)[x \mapsto N][y \mapsto L] &= \lambda z. (M_1[x \mapsto N][y \mapsto L]) \\
&= \lambda z. (M_1[y \mapsto L][x \mapsto N[y \mapsto L]]) \\
&= (\lambda z. M_1)[y \mapsto L][x \mapsto N[y \mapsto L]]
\end{align*}$$

Case 3: $M = M_1 M_2$. The statement follows again from the induction hypothesis.
What the experts say...

“We thank T. Thacher Robinson for showing us on August 19, 1962 by a counterexample the existence of an error in our handling of bound variables.”


“When doing the formalization, I discovered that the core part of the proof . . . is fairly straightforward and only requires a good understanding of the paper version. However, in completing the proof I observed that in certain places I had to invest much more work than expected, e.g. proving lemmas about substitution and weakening.”

Thorsten Altenkirch in Proceedings of TLCA, 1993

“Proving theorems about substitutions (and related operations such as $\alpha$-conversion) required far more time than any other variety of theorem.”

Myra VanInwegen in her PhD-thesis, 1996

⇒ Better tool support necessary
Our tool: Isabelle

- Developed (since 1986) by Larry Paulson (Cambridge) and Tobias Nipkow
- Interactive theorem prover
- Logical Framework
  Description of various object logics using a meta logic (Isabelle/Pure)
- Most well-developed object logic: Isabelle/HOL
- Design philosophy
  - Inferences may only be performed by a small kernel (“LCF approach”)
  - Definitional theory extension
    New concepts (such as inductive datatypes and predicates) must be defined using already existing concepts.

“The method of ‘postulating’ what we want has many advantages; they are the same as the advantages of theft over honest toil. Let us leave them to others and proceed with our honest toil.”

Bertrand Russell, Introduction to Mathematical Philosophy
Hierarchy of definitional packages

Nominal Datatype

Datatype

Typedef

Inductive

Rec. Function

Definition

Coinductive

Isabelle/HOL

Isabelle/Pure
Existing approaches for reasoning with bound variables

- “Name-carrying” syntax
  - readable
  - $\alpha$-equivalence must be formalized explicitly
  - substitution function requires variable renaming

- De Bruijn indices
  - $\alpha$-equivalence coincides with syntactic equality
  - simple induction / recursion principles
  - substitution function requires index calculations
  - unreadable

- “Locally nameless” approach
  - $\alpha$-equivalence coincides with syntactic equality
  - substitution function does not require index calculations
  - well-formedness must be formalized explicitly

- Higher order abstract syntax (in Isabelle or Coq, for example)
  - abstraction and substitution “for free”
  - exotic terms
Our approach
A more abstract approach

Problem: How can we hide details of the representation from the user?

Possible solution:
Introduce a new type, whose elements correspond to...
- ... the $\alpha$-equivalence classes of “name-carrying”-terms, or
- ... the well-formed “locally nameless” terms.

\[ \lambda x. x \]
\[ \lambda y. y \]
\[ \lambda x \, y. \, x \]
\[ \lambda y \, x. \, y \]
\[ \lambda x \, y. \, x \]
\[ ... \]

Question: How does abstract “interface” for this type look like? \( \Rightarrow \) Nominal logic!
Nominal logic
[A. M. Pitts et al, TACS 2001, I&C 2003]

• Specific types for names (with infinitely many elements)

• Permutations: $\pi \cdot t$ (bijective)
  where $\pi = [(a_1, b_1), \ldots, (a_n, b_n)]$, $a_i$ and $b_i$ are names

• Support ($\approx$ set of free variables): $\text{supp} \ t$
  – Note: Nominal logic is incompatible with the axiom of choice (which is part of Isabelle/HOL), because “everything” is assumed to have finite support
  – Not all HOL terms and types have finite support
  – We use extra preconditions to characterize terms and types that have finite support
  – Some preconditions can be “hidden” using axiomatic type classes
  – Nominal datatypes always have finite support

• Freshness: $a \# t \equiv a \notin \text{supp} \ t$
Nominal datatypes

Datatypes with abstractions

\[
\text{nominal-datatype } \vec{\alpha} \; \text{ty} = \cdots \mid C_i \langle \vec{a}_i \rangle_{\tau_i}^1 \cdots \langle \vec{a}_i \rangle_{\tau_i}^{m_i} \mid \cdots
\]

where \( \vec{a}_i \) are lists of atom types and \( \tau_i \) are types, possibly containing \( \vec{\alpha} \; \text{ty} \)

Example: untyped \( \lambda \)-calculus

\[
\text{atom-decl name}
\]

\[
\text{nominal-datatype} \; \text{term} = \text{Var name} \mid \text{App term term} \mid \text{Abs } \langle \text{name} \rangle \text{term}
\]

Limitations

<table>
<thead>
<tr>
<th>Lam_2 (\langle \text{name} \rangle \langle \text{name} \rangle \text{term})</th>
<th>OK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lams (\langle \text{name} ; \text{list} \rangle \text{term})</td>
<td>Not yet supported</td>
</tr>
<tr>
<td>Let (\langle \text{pattern} \rangle \text{term}) \text{ term}</td>
<td>Not yet supported</td>
</tr>
</tbody>
</table>
Nominal datatypes – $\alpha$-equivalence

Standard datatypes

$\text{Abs } a \ t = \text{Abs } b \ u \iff (a = b \land t = u)$

Nominal datatypes

$\text{Abs } a \ t = \text{Abs } b \ u \iff (a = b \land t = u) \lor (a \neq b \land t = [(a, b)] \bullet u \land a \# u)$

Example

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

because

- $a \neq b$
- $\text{Var } a = \text{Var } ((a, b) \bullet b) = [(a, b)] \bullet (\text{Var } b)$
- $a \# \text{Var } b$
Nominal datatypes – induction

Weak induction rule
\[ \forall n. P (\text{Var } n) \]
\[ \forall t \ u. P \ t \implies P \ u \implies P (\text{App } t \ u) \]
\[ \forall n \ t. P \ t \implies P (\text{Abs } n \ t) \]
\[ \forall t. P \ t \]

Strong induction rule
\[ \forall n \ c. P \ c \ (\text{Var } n) \]
\[ \forall t \ u \ c. (\forall d. P \ d \ t) \implies (\forall d. P \ d \ u) \implies P \ c \ (\text{App } t \ u) \]
\[ \forall n \ t \ c. n \# c \implies (\forall d. P \ d \ t) \implies P \ c \ (\text{Abs } n \ t) \]
\[ \forall t \ c. P \ c \ t \]
Using the strong induction rule

**Substitution lemma:** If \( x \neq y \) and \( x \notin FV(L) \), then

\[
M[x \mapsto N][y \mapsto L] = M[y \mapsto L][x \mapsto N[y \mapsto L]].
\]

... 

**Case 2:** \( M = \lambda z. M_1 \).

By the variable convention we may assume that \( z \neq x, y \) and \( z \) is not free in \( N, L \).

\[
\forall z \ c. \ P c \ (\text{Var } z)
\]

\[
\forall M_1 \ M_2 \ c. \\
(\forall d. \ P d M_1) \implies (\forall d. \ P d M_2) \implies \\
P c (\text{App } M_1 M_2)
\]

\[
\forall z \ M_1 \ c. \ z \# c \implies (\forall d. \ P d M_1) \implies \\
P c (\text{Abs } z M_1)
\]

\[
\forall M \ c. \ P c M
\]

\[
P := \lambda(x, y, N, L). \ \lambda M. \ I M \ x \ y \ N \ L \\
I M \ x \ y \ N \ L \equiv x \neq y \implies x \notin FV(L) \implies M[x \mapsto N][y \mapsto L] = M[y \mapsto L][x \mapsto N[y \mapsto L]]
\]
Using the strong induction rule

Substitution lemma: If \( x \neq y \) and \( x \notin \text{FV}(L) \), then
\[
M[x \mapsto N][y \mapsto L] = M[y \mapsto L][x \mapsto N[y \mapsto L]].
\]

... 

Case 2: \( M = \lambda z. M_1 \).
By the variable convention we may assume that \( z \neq x, y \) and \( z \) is not free in \( N, L \).

\[
\forall z \ x \ y \ N \ L. \ T (\text{Var} \ z) \ x \ y \ N \ L
\]

\[
\forall M_1 \ M_2 \ x \ y \ N \ L.
(\forall x' \ y' \ N' \ L'. \ T M_1 \ x' \ y' \ N' \ L') \implies (\forall x' \ y' \ N' \ L'. \ T M_2 \ x' \ y' \ N' \ L') \implies T (\text{App} \ M_1 \ M_2) \ x \ y \ N \ L
\]

\[
\forall z \ M_1 \ x \ y \ N \ L. \ z \#(x, y, N, L) \implies (\forall x' \ y' \ N' \ L'. \ T M_1 \ x' \ y' \ N' \ L') \implies T (\text{Abs} \ z \ M_1) \ x \ y \ N \ L
\]

\[
\forall M \ x \ y \ N \ L. \ T M \ x \ y \ N \ L
\]

\[
P := \lambda (x, y, N, L). \ \lambda M. \ T M \ x \ y \ N \ L
\]

\[
T M \ x \ y \ N \ L \equiv x \neq y \implies x \notin \text{FV}(L) \implies M[x \mapsto N][y \mapsto L] = M[y \mapsto L][x \mapsto N[y \mapsto L]]
\]
Recursive functions on $\alpha$-equivalence classes

Is the following function well-defined?

\[
\begin{align*}
  \text{bvars} \ (\text{Var} \ x) &= \{\} \\
  \text{bvars} \ (\text{App} \ t_1 \ t_2) &= \text{bvars} \ t_1 \cup \text{bvars} \ t_2 \\
  \text{bvars} \ (\text{Abs} \ x \ t) &= \{x\} \cup \text{bvars} \ t
\end{align*}
\]

Unproblematic for standard datatypes...

... but not for nominal datatypes:

\[
\text{Abs} \ a \ (\text{Var} \ a) = \text{Abs} \ b \ (\text{Var} \ b)
\]

but

\[
\text{bvars} \ (\text{Abs} \ a \ (\text{Var} \ a)) = \{a\} \neq \{b\} = \text{bvars} \ (\text{Abs} \ b \ (\text{Var} \ b))
\]

\[\implies\text{Result of function may not depend on choice of bound variable names!}\]
Recursion combinator

Standard datatypes

\[
\text{term\_rec } f_1 \ f_2 \ f_3 \ (\text{Var } n) = f_1 \ n \\
\text{term\_rec } f_1 \ f_2 \ f_3 \ (\text{App } t \ u) = f_2 \ t \ u \ (\text{term\_rec } f_1 \ f_2 \ f_3 \ t) \ (\text{term\_rec } f_1 \ f_2 \ f_3 \ u) \\
\text{term\_rec } f_1 \ f_2 \ f_3 \ (\text{Abs } n \ t) = f_3 \ n \ t \ (\text{term\_rec } f_1 \ f_2 \ f_3 \ t)
\]

Nominal datatypes

\[
\text{n\#}(f_1, f_2, f_3) \land (\forall n \ t \ r. \ n\#f_3 \implies n\#f_3 \ n \ t \ r) \implies \\
\text{term\_rec } f_1 \ f_2 \ f_3 \ (\text{Abs } n \ t) = f_3 \ n \ t \ (\text{term\_rec } f_1 \ f_2 \ f_3 \ t)
\]
Substitution function

\[(\text{Var } x)[y \mapsto u] = (\text{if } x = y \text{ then } u \text{ else } (\text{Var } x))\]

\[(\text{App } t_1 \ t_2)[y \mapsto u] = \text{App } (t_1[y \mapsto u]) \ (t_2[y \mapsto u])\]

\[x\#(y, u) \implies (\text{Abs } x \ t)[y \mapsto u] = \text{Abs } x \ (t[y \mapsto u])\]

Instantiation of the recursion combinator

\[f_1 = \lambda x. \text{if } x = y \text{ then } u \text{ else } (\text{Var } x)\]
\[f_2 = \lambda t_1 \ t_2 \ r_1 \ r_2. \text{App } r_1 \ r_2\]
\[f_3 = \lambda x \ t \ r. \text{Abs } x \ r\]

Freshness conditions for binders

\[\forall x \ t \ r. x\#f_3 \implies x\#\text{Abs } x \ r\]

Note:

\[x\#\text{Abs } x \ r\]
\[= x \notin \text{supp } (\text{Abs } x \ r)\]
\[= x \notin \text{supp } r - \{x\}\]
Bound variables

\[ \text{bvars (Var } x) = \{\} \]
\[ \text{bvars (App } t_1 t_2) = \text{bvars } t_1 \cup \text{bvars } t_2 \]
\[ \text{bvars (Abs } x t) = \{x\} \cup \text{bvars } t \]

Instantiation of the recursion combinator

\[ f_1 = \lambda x. \{\} \]
\[ f_2 = \lambda t_1 t_2 r_1 r_2. r_1 \cup r_2 \]
\[ f_3 = \lambda x t r. \{x\} \cup r \]

Freshness conditions for binders

\[ \forall x t r. x \# f_3 \implies x \# \{x\} \cup r \]

Note: a similar problem arises when attempting to define the “immediate subterms” function.
Isabelle proof of substitution lemma

**Lemma** substitution-lemma:

- **Assumes** fresh: $x \neq y \land x \not\in L$
- **Shows** $M[x \mapsto N][y \mapsto L] = M[y \mapsto L][x \mapsto N[y \mapsto L]]$ using fresh

**Proof** (nominal-induct $M$ avoiding: $x$ $y$ $N$ $L$ rule: lam.induct)

**Case** $(\text{Abs } z \ M_1)$
- have $(\text{Abs } z \ M_1)[x \mapsto N][y \mapsto L] = \text{Abs } z (M_1[x \mapsto N][y \mapsto L])$
  - using $\langle z \not\in x \rangle \langle z \not\in y \rangle \langle z \not\in N \rangle \langle z \not\in L \rangle$ by simp
- also from $\text{Abs}$ have $\ldots = \text{Abs } z (M_1[y \mapsto L][x \mapsto N[y \mapsto L]])$
  - using $\langle x \not\in y \rangle \langle x \not\in L \rangle$ by simp
- also have $\ldots = (\text{Abs } z (M_1[y \mapsto L]))[x \mapsto N[y \mapsto L]]$
  - using $\langle z \not\in x \rangle \langle z \not\in N \rangle \langle z \not\in L \rangle$ by (simp add: fresh-fact)
- also have $\ldots = (\text{Abs } z M_1)[y \mapsto L][x \mapsto N[y \mapsto L]]$
  - using $\langle z \not\in y \rangle \langle z \not\in L \rangle$ by simp
- finally show $(\text{Abs } z M_1)[x \mapsto N][y \mapsto L] = (\text{Abs } z M_1)[y \mapsto L][x \mapsto N[y \mapsto L]].$

next

\ldots

qed
Conclusion
Conclusion

• **Bad news:** proofs about calculi with variable binding are inherently complicated
• **Good news:** some of the complexity can be hidden inside nominal datatype package
• Needs quite a bit of infrastructure
  nominal package is currently the biggest package in Isabelle
• Implementation is contained in Isabelle Development Snapshot
• Many applications: see Christian’s talk (and mailing list)

**Further work**

• Strengthened inversion principles
• More general binding constructs
  ```latex
  \text{let } p = t \text{ in } u
  ```
• Generation of code from specifications involving nominal datatypes
• Support for more general recursion schemes
Literature


See the web site: isabelle.in.tum.de/nominal

Thanks for your attention!