Nominal Techniques in Coq

Danil Annenkov

University of Copenhagen, DIKU
HIPERFIT Workshop

November 16, 2017

Names

> There are only two hard things in Computer Science: cache invalidation and naming things.
>
> — Phil Karlton

Names

> There are only two hard things in Computer Science: cache invalidation and **naming things**.
>
> — Phil Karlton

The other side of naming things is to be independent of names!

Variable binding

- Variable binding is a ubiquitous concept in the programming language research.
- One wants definitions to be independent of the choice of names for bound variables.
- It is relatively easy to deal with binding in pen-and-paper proofs.
- It is notoriously hard to deal with in proof assistants.

Variable binding: Examples

Haskell	Java

`plusTwo a = let b = 2`
`             in a + b`
	`public int plusTwo (int a) {`
	`    int b = 2;`
	`    return a + b; }`

Variable binding: Examples

Haskell	Java

`plusTwo a = let b = 2`
`              in  a + b`
	`public int plusTwo (int a) {`
	`    int b = 2;`
	`    return a + b; }`

We can pick other names for a and b:

Haskell	Java

`plusTwo c = let d = 2`
`              in c + d`
	`public int plusTwo (int c) {`
	`    int d = 2;`
	`    return c + d; }`

Variable binding: Examples

Haskell	Java

`plusTwo a = let b = 2`
`              in a + b`
	`public int plusTwo (int a) {`
	`    int b = 2;`
	`    return a + b; }`

We can pick other names for a and b:

Haskell	Java

`plusTwo c = let d = 2`
`              in c + d`
	`public int plusTwo (int c) {`
	`    int d = 2;`
	`    return c + d; }`

But not arbitrary names: **variable capture**!

Haskell	Java

`plusTwo b = let b = 2`
`              in b + b`
	`public int plusTwo (int b) {`
	`    int b = 2;`
	`    return b + b; }`

Simply-Typed Lambda Calculus

- From now on we will switch to the Simply-Typed Lambda Calculus (STLC).
- STLC is well-studied and has a simple binding structure.
- The grammar of (raw) lambda terms:
  $e \in \texttt{Lam} ::= v \mid \lambda x.e \mid e_1 e_2$

Variable Convention

Barendregt's Variable Convention

If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

Variable Convention

- Consider the following typing rule for lambda abstraction:

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \qquad x \notin dom(\Gamma)}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

- In order prove weakening

$$\forall \Gamma, \Gamma', e, \tau, \quad \Gamma \vdash e : \tau \wedge \Gamma \subseteq \Gamma' \Rightarrow \Gamma' \vdash e : \tau$$

  in case of lambda abstraction one have to show $x \notin dom(\Gamma')$, knowing only $x \notin dom(\Gamma)$.

- This is possible using the variable convention.

Variable Convention

- Consider the following typing rule for lambda abstraction:

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \qquad x \notin dom(\Gamma)}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}$$

- In order prove weakening

$$\forall \Gamma, \Gamma', e, \tau, \quad \Gamma \vdash e : \tau \wedge \Gamma \subseteq \Gamma' \Rightarrow \Gamma' \vdash e : \tau$$

  in case of lambda abstraction one have to show $x \notin dom(\Gamma')$, knowing only $x \notin dom(\Gamma)$.
- This is possible using the variable convention.

  ✗ Not enough to formalise in a proof assistant!

α-equivalence

- We want to identify lambda-expressions up to renaming of bound variables.

α-equivalence

- We want to identify lambda-expressions up to renaming of bound variables.
- For example, $\lambda x.x =_\alpha \lambda y.y$.

α-equivalence

- We want to identify lambda-expressions up to renaming of bound variables.
- For example, $\lambda x.x =_\alpha \lambda y.y$.
- We could use substitution. Since $y[x/y] = x$, we say that $\lambda x.x =_\alpha \lambda y.y$.

α-equivalence

- We want to identify lambda-expressions up to renaming of bound variables.
- For example, $\lambda x.x =_\alpha \lambda y.y$.
- We could use substitution. Since $y[x/y] = x$, we say that $\lambda x.x =_\alpha \lambda y.y$.
- But substitution must be capture-avoiding, otherwise we would identify $\lambda y.y\ x$ and $\lambda x.x\ x$.

α-conversion with transpositions

- A *transposition* swaps two names:

$$(a\ b)\ c = \begin{cases} a, \text{if } b = c \\ b, \text{if } a = c \\ c, \text{otherwise} \end{cases}$$

- We apply transpositions to **all** occurrences of variables in the lambda-expression: $(y\ x) \cdot (\lambda y.y) = \lambda x.x$

α-conversion with transpositions

Important differences with substitution-based definitions:

- Transpositions cannot lead to variable capture:
  $(y\ x) \cdot (\lambda y.y\ x) = \lambda x.x\ y$
- We can implement the capture-avoiding substitution behavior using restrictions on variables (we write $x \# y$ for $x \neq y$ and say "x is fresh for "y"):

  pick $z \# x$ and $z \# y$, then $(z\ y) \cdot (\lambda y.y\ x) = \lambda z.z\ x$

Nominal Sets

- A transposition is special case of a *finitary* permutation: a bijective function on atoms, which is not the identity on a finite subset of atoms.

Nominal Sets

- A transposition is special case of a *finitary* permutation: a bijective function on atoms, which is not the identity on a finite subset of atoms.
- The theory of nominal sets [Gabbay and Pitts 1999, Pitts 2013] is a

  mathematical theory of names: scope, binding, freshness.

Nominal Sets

- A transposition is special case of a *finitary* permutation: a bijective function on atoms, which is not the identity on a finite subset of atoms.
- The theory of nominal sets [Gabbay and Pitts 1999, Pitts 2013] is a

  mathematical theory of names: scope, binding, freshness.
- Uniform theory based on notions of permutation of variables and finite support.

Nominal Sets

- A transposition is special case of a *finitary* permutation: a bijective function on atoms, which is not the identity on a finite subset of atoms.
- The theory of nominal sets [Gabbay and Pitts 1999, Pitts 2013] is a

  mathematical theory of names: scope, binding, freshness.
- Uniform theory based on notions of permutation of variables and finite support.
- Applies to various binding structures.

Nominal Sets

- A transposition is special case of a *finitary* permutation: a bijective function on atoms, which is not the identity on a finite subset of atoms.
- The theory of nominal sets [Gabbay and Pitts 1999, Pitts 2013] is a

  mathematical theory of names: scope, binding, freshness.
- Uniform theory based on notions of permutation of variables and finite support.
- Applies to various binding structures.
- Allows to bring formalisations in a proof assistant closer to pen-and-paper proofs.

Nominal Sets: Definitions

- Assume a countably infinite set $\{a, b, c, \dots\}$ of *atoms* $\mathbb{A}$:

$$AtomInf : \forall X \in \mathcal{P}_{fin}(\mathbb{A}),\ \exists a,\ a \notin X$$

- An *action* of a permutation on a set X is an operation $- \cdot - : Perm\ \mathbb{A} \times X \to X$ with the following properties:
  - for any $x \in X$, $id \cdot x = x$
  - for any $x \in X$, permutations π_1 and π_2, $\pi_1 \cdot (\pi_2 \cdot x) = (\pi_1 \circ \pi_2) \cdot x$
- *Finite support* in terms of transpositions:

$$\forall a, b \notin supp\ x.\ (a\ b) \cdot x = x$$

- A *nominal set* $\mathbf{X}$ is a set X, equipped with an action $- \cdot -$, s.t. each element in X is finitely supported.

Nominal Set of Lambda Expressions

- Action (a permutation is applied to **all** occurrences of atoms uniformly):

$$\begin{aligned}\pi \cdot v &= \pi v \\ \pi \cdot (\lambda x.e) &= \lambda(\pi x).\pi \cdot e \\ \pi \cdot (e_1 e_2) &= (\pi \cdot e_1)(\pi \cdot e_2)\end{aligned}$$

- Support is a set of **all** atoms:

$$\begin{aligned}\mathit{supp}\ v &= \{v\} \\ \mathit{supp}\ (\lambda x.e) &= \{x\} \cup \mathit{supp}\ e \\ \mathit{supp}\ (e_1 e_2) &= (\mathit{supp}\ e_1) \cup (\mathit{supp}\ e_2)\end{aligned}$$

Nominal Techniques: α-equivalence

We can define α-equivalence just in terms of the freshness relation and transpositions:

$$\frac{}{a =_\alpha a} \qquad \frac{t_1 =_\alpha t_1' \qquad t_2 =_\alpha t_2'}{t_1 t_2 =_\alpha t_1' t_2'}$$

$$\frac{(a_1\ b) \cdot t_1 =_\alpha (a_2\ b) \cdot t_2 \qquad b\#(a_1, a_2, fv(t_1), fv(t_2))}{\lambda a_1.t_1 =_\alpha \lambda a_2.t_2}$$

The freshness relation (a is fresh for x)

$$a\#x = a \notin supp\ x$$

We write $a\#(x_1, \ldots, x_n)$ for $a\#x_1 \wedge \cdots \wedge a\#x_n$.

The support $t \in Lam/ =_\alpha$ is a set of free variables of t.

Nominal Techniques: α-equivalence

$$\frac{}{\lambda x.x =_\alpha \lambda y.y}$$

Nominal Techniques: α-equivalence

We get z from the *AtomInf* axiom with $\{x, y\}$

$$\frac{z \# (x, y, \{x\}, \{y\})}{\lambda x.x =_\alpha \lambda y.y}$$

Nominal Techniques: α-equivalence

We get z from the *AtomInf* axiom with $\{x, y\}$

$$\frac{(x\ z) \cdot x =_\alpha (y\ z) \cdot y \qquad z \# (x, y, \{x\}, \{y\})}{\lambda x.x =_\alpha \lambda y.y}$$

Nominal Techniques: α-equivalence

We get z from the *AtomInf* axiom with $\{x, y\}$

$$\dfrac{\dfrac{}{z =_\alpha z} \qquad z \# (x, y, \{x\}, \{y\})}{\lambda x.x =_\alpha \lambda y.y}$$

Nominal Techniques in Coq: Permutations

Two ways of defining a permutation:

- ```
  Record Perm :=
      { perm : Atom → Atom;
        is_biject_perm : (is_inj perm) ∧ (is_surj perm);
        has_fin_supp_perm : has_fin_supp perm}.
  ```
  ```

Nominal Techniques in Coq: Permutations

Two ways of defining a permutation:

- ```
  Record Perm :=
      { perm : Atom → Atom;
        is_biject_perm : (is_inj perm) ∧ (is_surj perm);
        has_fin_supp_perm : has_fin_supp perm}.
  ```

  We cannot use `is_biject_perm` to construct an inverse permutation!
  ```

Nominal Techniques in Coq: Permutations

Two ways of defining a permutation:

- ```
  Record Perm :=
      { perm : Atom → Atom;
        is_biject_perm : (is_inj perm) ∧ (is_surj perm);
        has_fin_supp_perm : has_fin_supp perm}.
  ```

  We cannot use is_biject_perm to construct an inverse permutation!

- ```
  Record Perm :=
      { perm : Atom → Atom;
        perm_inv : Atom → Atom;
        l_inv : (perm_inv ∘perm) = id ;
        r_inv : (perm ∘ perm_inv) = id ;
        fin_supp : has_fin_supp perm}.
  ```

Nominal Techniques in Coq: Nominal Sets

We use type classes to define nominal sets:

```
Class NomSet :=
  { Carrier : Type;
    action : Perm → Carrier → Carrier;
    supp : Carrier → FinSetA;
    action_id : forall (x : Carrier), action id_perm x = x;
    action_compose : forall (x : Carrier) (p p' : Perm),
        action p (action p' x) = action (p ∘p p') x;
    support_spec : forall (p : Perm)  (x :  Carrier),
        (forall (a : Atom), V.In a (supp x) → p a = a) →
        action p x = x}.
```

Nominal Techniques in Coq: Lambda Expressions

The nominal set of lambda expressions is an instance of NomSet

```
Instance NomExp : NomSet :=
       {| Carrier := Exp;
          action := fun p e ⇒ ac_exp p e;
          supp := fun e ⇒ supp_exp e;
          action_id := fun e ⇒ (* omitted *);
          action_compose := fun e p1 p2 ⇒ (* omitted *);
          support_spec := (* omitted *) |}.
```

ac_exp p e recursively applies p to all atoms in e.
supp_exp e returns a set of all atoms in e.

Nominal Techniques in Coq: α-equivalence

The definition of α-equivalence:

```
Inductive ae_exp : NomExp → NomExp → Prop :=
| ae_var : forall (a : NomAtom),
    (Var a) =α (Var a)
| ae_lam : forall (a b c : NomAtom) (e1 e2 : NomExp),
    c # (a, b, fv_exp e1, fv_exp e2) →
    ((swap a c) @ e1) =α ((swap b c) @ e2) →
    (Lam a e1) =α (Lam b e2)
| ae_app : forall (e1 e2 e1' e2' : NomExp),
    e1 =α e1' →
    e2 =α e2' →
    (App e1 e2) =α (App e1' e2')
where "e1 =α e2" := (ae_exp e1 e2).
```

We use the notation `(swap a b) @ e` for $(a\ b) \cdot e$.

Nominal Techniques: Summary

- Actions are “uniform”: permutations apply to variables in any positions - binding, bound, free.

Nominal Techniques: Summary

- Actions are "uniform": permutations apply to variables in any positions - binding, bound, free.
- Permutations cannot lead to variable capture.

Nominal Techniques: Summary

- Actions are "uniform": permutations apply to variables in any positions - binding, bound, free.
- Permutations cannot lead to variable capture.
- There is a simple characterisation of α-equivalence in terms of transpositions and freshness.

Nominal Techniques: Summary

- Actions are “uniform”: permutations apply to variables in any positions - binding, bound, free.
- Permutations cannot lead to variable capture.
- There is a simple characterisation of α-equivalence in terms of transpositions and freshness.
- α-equivalence can be generalized to various structures involving bound variables.

Nominal Techniques: Summary

- Actions are "uniform": permutations apply to variables in any positions - binding, bound, free.
- Permutations cannot lead to variable capture.
- There is a simple characterisation of α-equivalence in terms of transpositions and freshness.
- α-equivalence can be generalized to various structures involving bound variables.
- Our development is available on GitHub: `https://github.com/annenkov/stlcnorm`

Nominal Techniques in Coq: Future Work

- Ideally, we would like to have a “nominal” induction principle.
- This requires quotienting with α-equivalence.
- Defining quotients in Coq is not easy.
- Implementation of Aydemir et al. axiomatises a nominal induction principle for lambda expressions quotiented with α-equivalence and provides the soundness proof.
- Higher inductive types could be an interesting option.

Nominal Techniques: Related Work

- The most developed library for nominal techniques is the Nominal Isabelle package (Isabelle/HOL proof assistant) [Urban and Tasson 2005].
- The theory of nominal sets in Agda [Choudhury 2015].
- Aydemir, Bohannon, Weirich. Nominal Reasoning Techniques in Coq. 2007.
  `http://www.seas.upenn.edu/~sweirich/papers/nominal-coq/`
- Nominal techniques in Coq are part of the DeepSpec summer school course:
  `https://github.com/DeepSpec/dsss17/tree/master/Stlc`

Thank you!

Thank you for your attention!