A lightweight approach for variable binding in Coq

Piotr Polesiuk

University of Wrocław

Approaches for variable binding

Concrete or nominal

Approaches for variable binding

Concrete or nominal

✗ $t_1 \equiv_\alpha t_2 \iff t_1 = t_2$

Approaches for variable binding

Concrete or nominal

✗ $t_1 \equiv_\alpha t_2 \not\Longleftrightarrow t_1 = t_2$

Locally nameless

✓ $t_1 \equiv_\alpha t_2 \Longleftrightarrow t_1 = t_2$

✓ simple definition of substitution

✓ simple proofs?

Approaches for variable binding

Concrete or nominal

✗ $t_1 \equiv_\alpha t_2 \not\Longleftrightarrow t_1 = t_2$

Locally nameless

✓ $t_1 \equiv_\alpha t_2 \Longleftrightarrow t_1 = t_2$

✓ simple definition of substitution

✗ simple proofs? I'm not sure

✗ complex metatheory

Approaches for variable binding

Concrete or nominal

✗ $t_1 \equiv_\alpha t_2 \not\Longleftrightarrow t_1 = t_2$

Locally nameless

✓ $t_1 \equiv_\alpha t_2 \Longleftrightarrow t_1 = t_2$

✓ simple definition of substitution

✗ simple proofs? I'm not sure

✗ complex metatheory

DeBruijn indices

✓ $t_1 \equiv_\alpha t_2 \Longleftrightarrow t_1 = t_2$

✓ simple metatheory

Approaches for variable binding

Concrete or nominal

✗ $t_1 \equiv_\alpha t_2 \not\Longleftrightarrow t_1 = t_2$

Locally nameless

✓ $t_1 \equiv_\alpha t_2 \Longleftrightarrow t_1 = t_2$

✓ simple definition of substitution

✗ simple proofs? I'm not sure

✗ complex metatheory

DeBruijn indices

✓ $t_1 \equiv_\alpha t_2 \Longleftrightarrow t_1 = t_2$

✓ simple metatheory

✗ complex definition of substitution

✗ mess with indices

Monadic representation

- ✓ $t_1 \equiv_\alpha t_2 \iff t_1 = t_2$
- ✓ simple metatheory (only 5 lemmas)
- ✓ simple definition of substitution
- ✓ simple proofs
- ✓ types as a guide
- ✓ no dependent types!

Parametrise your term by a set of variables!

```
Inductive term (A : Set) : Set :=
| tm_var : A → term A
| tm_app : term A → term A → term A
.
```

Parametrise your term by a set of variables!

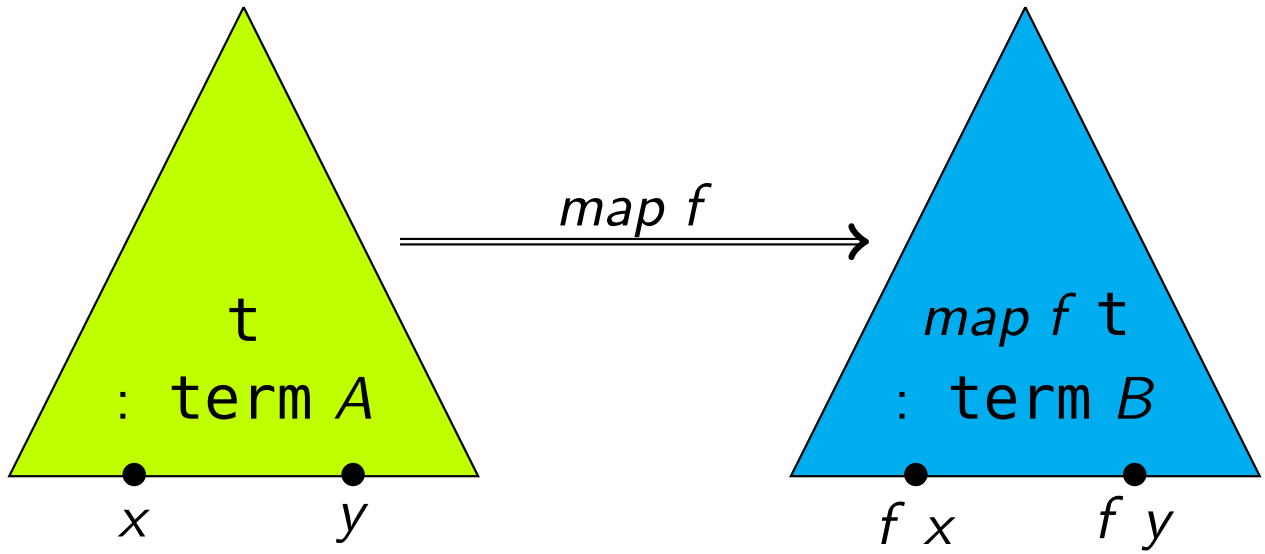
```
Inductive term (A : Set) : Set :=
| tm_var : A → term A
| tm_app : term A → term A → term A
.
```

It is a

MONAD

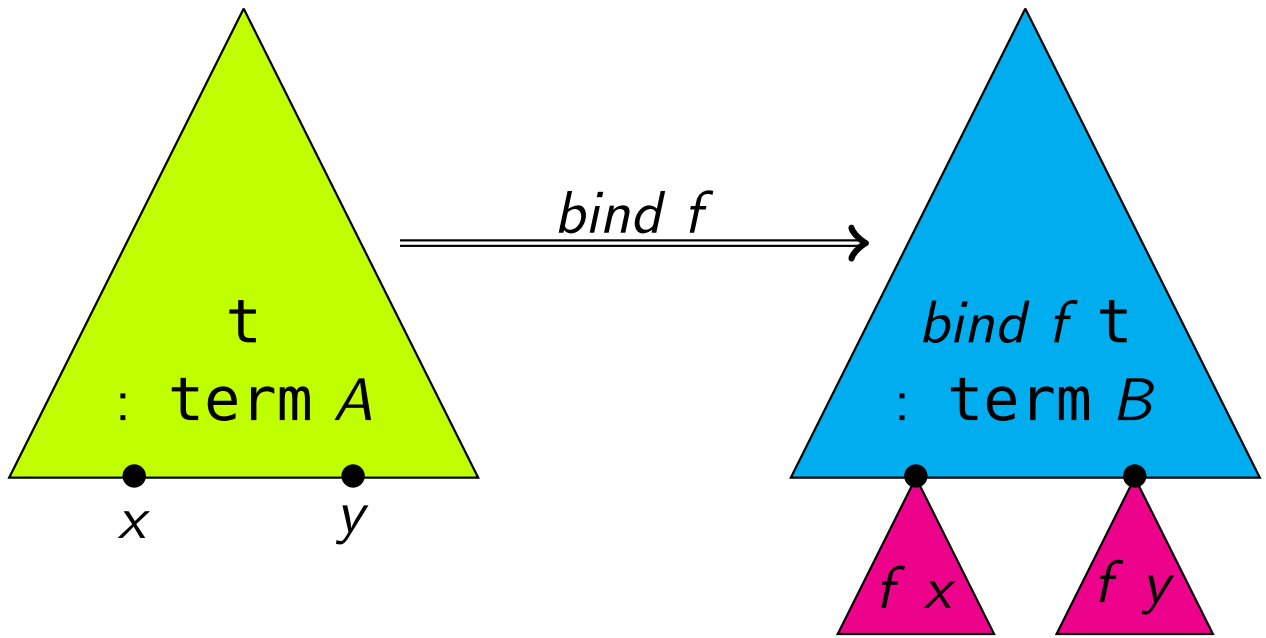
map renames variables

$$map : (A \to B) \to \texttt{term}\ A \to \texttt{term}\ B$$



bind is a simultaneous substitution

$$\mathit{bind} : (A \to \texttt{term}\ B) \to \texttt{term}\ A \to \texttt{term}\ B$$



Binders extends set of variables

```
Inductive inc (V : Set) : Set :=
| VZ : inc V
| VS : V → inc V
.

Inductive term (V : Set) : Set :=
...
| tm_lam : term (inc V) → term V
...
.
```

Metatheory

1. $map\ id = id$
2. $map\ f \circ map\ g = map\ (f \circ g)$
3. $bind\ return = id$
4. $f_1 \circ f_2 = map\ g_1 \circ g_2 \quad \Rightarrow$
   $bind\ f_1 \circ map\ f_2 = map\ g_1 \circ bind\ g_2$
5. $bind\ f \circ bind\ g = bind(bind\ f \circ g)$

BONUS

Type-soundness of CBN STLC in 1 minute

```
Require Import Utf8.

Inductive inc (V : Set) : Set :=
| VZ : inc V
| VS : V → inc V
.

Arguments VZ {V}.
Arguments VS {V}.

Definition inc_map {A B : Set}
    (f : A → B) (x : inc A) : inc B :=
  match x with
  | VZ   => VZ
  | VS y => VS (f y)
  end.
```

```
Inductive term (V : Set) : Set :=
| tm_var : V → term V
| tm_lam : term (inc V) → term V
| tm_app : term V → term V → term V
.

Arguments tm_var [V] _.
Arguments tm_lam [V] _.
Arguments tm_app [V] _ _.

Notation term0 := (term Empty_set).
```

```
Fixpoint map {A B : Set}
    (f : A → B) (t : term A) : term B :=
  match t with
  | tm_var x     => tm_var (f x)
  | tm_lam t     => tm_lam (map (inc_map f) t)
  | tm_app t₁ t₂ => tm_app (map f t₁) (map f t₂)
  end.

Definition lift {A B : Set} (f : A → term B)
    (x : inc A) : term (inc B) :=
  match x with
  | VZ   => tm_var VZ
  | VS y => map VS (f y)
  end.
```

```
Fixpoint bind {A B : Set}
    (f : A → term B) (t : term A) : term B :=
  match t with
  | tm_var x     => f x
  | tm_lam t     => tm_lam (bind (lift f) t)
  | tm_app t₁ t₂ => tm_app (bind f t₁) (bind f t₂)
  end.

Definition subst_func {V : Set}
    (s : term V) (x : inc V) : term V :=
  match x with
  | VZ   => s
  | VS y => tm_var y
  end.

Notation subst t s := (bind (subst_func s) t).
```

```
Inductive red {V : Set} :
    term V → term V → Prop :=
| red_beta : ∀ t s,
    red (tm_app (tm_lam t) s) (subst t s)
| red_app : ∀ t t' s,
    red t t' →
    red (tm_app t s) (tm_app t' s)
.
```

```
Inductive tp : Set :=
| tp_base  : tp
| tp_arrow : tp → tp → tp
.

Definition env (V : Set) : Set := V → tp.
```

```
Definition env_empty (x : Empty_set) : tp :=
  match x with end.

Definition env_ext {V : Set}
    (Γ : env V) (τ : tp) (x : inc V) : tp :=
  match x with
  | VZ   => τ
  | VS y => Γ y
  end.

Notation "ø" := (env_empty).
Notation "Γ ',+' τ" := (env_ext Γ τ)
  (at level 45, left associativity).
```

```
Reserved Notation "'T[' Γ '⊢' t '⇒' τ ']'".

Inductive typing {V : Set} (Γ : env V) :
    term V → tp → Prop :=
| T_var : ∀ x,
    T[ Γ ⊢ tm_var x ⇒ Γ x ]
| T_lam : ∀ t σ τ,
    T[ Γ ,+ σ ⊢ t ⇒ τ ] →
    T[ Γ ⊢ tm_lam t ⇒ tp_arrow σ τ ]
| T_app : ∀ t s σ τ,
    T[ Γ ⊢ t ⇒ tp_arrow σ τ ] →
    T[ Γ ⊢ s ⇒ σ ] →
    T[ Γ ⊢ tm_app t s ⇒ τ ]
where "T[ Γ ⊢ t ⇒ τ ]" := (@typing _ Γ t τ).
```

Time for proofs

```
Lemma progress_aux {V : Set} (Γ : env V) t τ :
  T[ Γ ⊢ t ⇒ τ ] → V = Empty_set →
  (∃ t', t = tm_lam t') ∨ (∃ t', red t t').
Proof.
induction 1; intro; subst.
+ destruct x.
+ eauto.
+ right; destruct IHtyping1 as [[]|[]]; trivial.
  - subst; eexists; constructor 1.
  - eexists; constructor 2; eassumption.
Qed.

Theorem progress (t : term0) τ : T[ ø ⊢ t ⇒ τ ] →
  (∃ t', t = tm_lam t') ∨ (∃ t', red t t').
Proof.
intros; eapply progress_aux; eauto.
Qed.
```

```
Lemma typing_map {A B : Set}
  (Γ : env A) (Δ : env B)
  (f : A → B) t τ :
  (∀ x, Δ (f x) = Γ x) →
  T[ Γ ⊢ t ⇒ τ ] →
  T[ Δ ⊢ map f t ⇒ τ ].
Proof.
intros Hf Htp.
generalize B Δ f Hf; clear B Δ f Hf.
induction Htp; simpl; intros B Δ f Hf.
+ rewrite <- Hf; constructor.
+ constructor; apply IHHtp.
  intros [ | x ]; simpl; auto.
+ econstructor; eauto.
Qed.
```

```
Lemma typing_bind {A B : Set}
  (Γ : env A) (Δ : env B)
  (f : A → term B) t τ :
  (∀ x, T[ Δ ⊢ f x ⇒ Γ x ]) →
  T[ Γ ⊢ t ⇒ τ ] →
  T[ Δ ⊢ bind f t ⇒ τ ].
Proof.
intros Hf Htp.
generalize B Δ f Hf; clear B Δ f Hf.
induction Htp; simpl; intros B Δ f Hf.
+ auto.
+ constructor; apply IHHtp.
  intros [ | x ]; simpl; [ constructor | ].
  eapply typing_map; [ | apply Hf ]; reflexivity.
+ econstructor; eauto.
Qed.
```

```
Theorem subject_reduction (t t' : term0) τ :
  red t t' → T[ ø ⊢ t ⇒ τ ] → T[ ø ⊢ t' ⇒ τ ].
Proof.
intro Hred; generalize τ; clear τ.
induction Hred; intros τ Htp; inversion Htp.
+ inversion H1.
  eapply typing_bind; [ | eassumption ].
  intros [ | [] ]; simpl; assumption.
+ econstructor; eauto.
Qed.
```

Questions?

Ask me some questions about variable binding

ppolesiuk@cs.uni.wroc.pl