Alpha-Structural Induction and Recursion for the Lambda Calculus in Constructive Type Theory

Ernesto Copello, Álvaro Tasistro, Nora Szasz, Ana Bove, Maribel Fernández

10th Workshop on Logical and Semantic Frameworks, with Applications.

Motivation

Studying and formalising reasoning techniques over programming languages.

- Reasoning like in pen-and-paper proofs.
- Using constructive type theory as proof assistant.

More specifically: λ-calculus & Agda.

Motivation

Studying and formalising reasoning techniques over programming languages.

- Reasoning like in pen-and-paper proofs.
- Using constructive type theory as proof assistant.

More specifically: λ-calculus & Agda.

We study a formilisation of λ-calculus with the following characteristics:

- In its original syntax with only one sort of names, like pen-and-paper.
- Substitution and α-conversion is based upon name swapping (*Nominal* approach).

Reasoning over α-equivalence classes

Barenregt's variable convention (BVC)

Each λ-term represents its α class, so we can assume that we have bound and free variables all different.

Reasoning over α-equivalence classes

Barenregt's variable convention (BVC)

Each λ-term represents its α class, so we can assume that we have bound and free variables all different.

A complete formilisation often implies:

- Define a no capture substitution operation.
- Complete induction over the *size* of terms is needed to fill the gap between terms and α-equivalence classes.
- Prove that all properties being proved are preserved under α-conversion (α-compatibe predicates).

Reasoning over α-equivalence classes

We want to be able to reproduce the following induction sketch

<u>λ case:</u>

To prove $\forall x, M,\ P(M) \Rightarrow P(\lambda x M)$ we instead prove:

$$\exists A \subseteq V, A \text{ finite}, \forall x^*, M^* \text{ renamings}/x^* \notin A, \lambda x M \sim_\alpha \lambda x^* M^*, \\ P(M^*) \Rightarrow P(\lambda x^* M^*)$$

Formalisation

All the slides from now on are fragments of our compiled Agda code.

Variable Swaping

$$
\begin{array}{llll}
(\_ \bullet \_)_a \_ & : \mathsf{Atom} \to \mathsf{Atom} \to \mathsf{Atom} \to \mathsf{Atom} & & \\
(\ a \bullet b\ )_a\ c & \mathsf{with}\ c \stackrel{?}{=}_a a & & \\
\ldots \mid \mathsf{yes}\ \_ & & = b & \\
\ldots \mid \mathsf{no}\ \_ & \mathsf{with}\ c \stackrel{?}{=}_a b & & \\
\ldots & \quad \mid \mathsf{yes}\ \_ & = a & \\
\ldots & \quad \mid \mathsf{no}\ \_ & = c &
\end{array}
$$

Terms

```
data Λ : Set where
  v    : Atom → Λ
  _·_  : Λ → Λ → Λ
  ƛ    : Atom → Λ → Λ
```

Terms

$$
\begin{array}{lll}
\textsf{data}\ \Lambda : \textsf{Set where} & & \\
\quad \textsf{v} & : & \textsf{Atom} \rightarrow \Lambda \\
\quad \_\cdot\_ & : & \Lambda \rightarrow \Lambda \rightarrow \Lambda \\
\quad \lambda\!\!\!- & : & \textsf{Atom} \rightarrow \Lambda \rightarrow \Lambda
\end{array}
$$

Swapping operation on terms is simpler than substitution, and with nicer properties than *renaming*:

- Also changes bound variables $\Rightarrow$ no variable capture
  $(y\ x) \bullet (\lambda\ x.x\ y) = \lambda\ y.y\ x$
- Injective

$$
\begin{array}{ll}
\textsf{lemma}\bullet_a\textsf{inj} & : \{a\ b\ c\ d : \textsf{Atom}\} \\
 & \rightarrow c \not\equiv d \\
 & \rightarrow (\ a \bullet b\ )_a\ c \not\equiv (\ a \bullet b\ )_a\ d
\end{array}
$$

- Idempotent

$$
\begin{array}{l}
\textsf{lemma(ab)(ab)c}{\equiv}\textsf{c}\ : \{a\ b\ c : \textsf{Atom}\} \\
\quad \rightarrow (\ a \bullet b\ )_a\ (\ a \bullet b\ )_a\ c \equiv c
\end{array}
$$

$(y\ x) \bullet (\lambda\ y.y\ x) = \lambda\ x.x\ y$

α-conversion, not using substitution!

```
data _∼α_ : Λ → Λ → Set where
  ∼αv  : {a : Atom} → v a ∼α v a
  ∼α·  : {M M' N N' : Λ} → M ∼α M' → N ∼α N'
         → M · N ∼α M' · N'
  ∼αƛ  : {M N : Λ}{a b : Atom}(xs : List Atom)
         → ((c : Atom) → c ∉ xs → ( a • c ) M ∼α ( b • c ) N)
         → ƛ a M ∼α ƛ b N
```

Novel definition.

- syntax directed

α-conversion, not using substitution!

```
data _∼α_ : Λ → Λ → Set where
  ∼αv : {a : Atom} → v a ∼α v a
  ∼α· : {M M' N N' : Λ} → M ∼α M' → N ∼α N'
        → M · N ∼α M' · N'
  ∼αƛ : {M N : Λ}{a b : Atom}(xs : List Atom)
        → ((c : Atom) → c ∉ xs → ( a • c ) M ∼α ( b • c ) N)
        → ƛ a M ∼α ƛ b N
```

Novel definition.

- syntax directed
- equivalent to classical one

α-conversion, not using substitution!

```
data _∼α_ : Λ → Λ → Set where
   ∼αv  : {a : Atom} → v a ∼α v a
   ∼α·  : {M M' N N' : Λ} → M ∼α M' → N ∼α N'
          → M · N ∼α M' · N'
   ∼αƛ  : {M N : Λ}{a b : Atom}(xs : List Atom)
          → ((c : Atom) → c ∉ xs → ( a • c ) M ∼α ( b • c ) N)
          → ƛ a M ∼α ƛ b N
```

Novel definition.

- syntax directed
- equivalent to classical one
- inspired in "cofinite quantification"
  (Brian Aydemir et al, "Engineering formal metatheory", 2008)

Induction Principles for λ-terms

In the following slides we will iterate over several induction principles, until we get one which captures Barenregt´s variable convention.

Each one can be derived from the former one.

Primitive induction

Comes for free with the definition of terms :

$$
\begin{aligned}
&\mathsf{TermPrimInd} : \{l : \mathsf{Level}\}(P : \Lambda \to \mathsf{Set}\ l)\\
&\quad \to (\forall\ a \to P\ (\mathsf{v}\ a))\\
&\quad \to (\forall\ M\ N \to P\ M \to P\ N \to P\ (M \cdot N))\\
&\quad \to (\forall\ M\ b \to P\ M \to P\ (\lambda\ b\ M))\\
&\quad \to \forall\ M \to P\ M
\end{aligned}
$$

Permutation induction

A permutation π is a sequence of swappings.

We derive this induction principle, from simple structural induction:

$$
\begin{array}{l}
\mathsf{TermIndPerm} : \{l : \mathsf{Level}\}(P : \Lambda \to \mathsf{Set}\ l) \\
\quad \to (\forall\ a \to P\ (\mathsf{v}\ a)) \\
\quad \to (\forall\ M\ N \to P\ M \to P\ N \to\ P\ (M \cdot N)) \\
\quad \to (\forall\ M\ b \to (\forall\ \pi \to P\ (\pi \bullet M)) \to P\ (\lambda\!\!\!\!\lambda\ b\ M)) \\
\quad \to \forall\ M \to P\ M
\end{array}
$$

α-structural induction

$$\alpha\text{CompatiblePred} : \{l : \text{Level}\} \to (\Lambda \to \text{Set } l) \to \text{Set } l$$
$$\alpha\text{CompatiblePred } P = \{M\ N : \Lambda\} \to M \sim\alpha\ N \to P\ M \to P\ N$$

We can derive this induction principle from the former one:

$$\begin{aligned}
&\text{Term}\alpha\text{PrimInd} : \{l : \text{Level}\}(P : \Lambda \to \text{Set } l)\\
&\quad \to \alpha\text{CompatiblePred } P\\
&\quad \to (\forall\ a \to P\ (\text{v}\ a))\\
&\quad \to (\forall\ M\ N \to P\ M \to P\ N \to P\ (M \cdot N))\\
&\quad \to \exists\ (\lambda\ vs \to (\forall\ M\ b \to b \notin vs \to P\ M \to P\ (\bar{\lambda}\ b\ M)))\\
&\quad \to \forall\ M \to P\ M
\end{aligned}$$

Now we can emulate the BVC!

α-structural iteration

We can easily derive an iteration principle from the former principle.

$$
\begin{aligned}
&\Lambda\mathrm{It}\ :\ \{l : \mathrm{Level}\}(A : \mathrm{Set}\ l)\\
&\rightarrow (\mathrm{Atom} \rightarrow A)\\
&\rightarrow (A \rightarrow A \rightarrow A)\\
&\rightarrow \mathrm{List}\ \mathrm{Atom} \times (\mathrm{Atom} \rightarrow A \rightarrow A)\\
&\rightarrow \Lambda \rightarrow A
\end{aligned}
$$

α-structural iteration

```
strong∼αCompatible  : {l : Level}{A : Set l}
                      → (Λ → A) → Λ → Set l
strong∼αCompatible f M = ∀ N → M ∼α N → f M ≡ f N


lemmaΛItStrongαCompatible : {l : Level}(A : Set l)
   → (hv : Atom → A)
   → (h· : A → A → A)
   → (vs : List Atom)
   → (hƛ : Atom → A → A )
   → (M : Λ) → strong∼αCompatible (ΛIt A hv h· (vs , hƛ)) M
```

α-structural recursion

Using the last iteration principle we define a recursion principle:

$$
\begin{aligned}
\Lambda\mathsf{Rec}\ &: \{l : \mathsf{Level}\}(A : \mathsf{Set}\ l)\\
&\rightarrow (\mathsf{Atom} \rightarrow A)\\
&\rightarrow (A \rightarrow A \rightarrow \Lambda \rightarrow \Lambda \rightarrow A)\\
&\rightarrow \mathsf{List}\ \mathsf{Atom} \times (\mathsf{Atom} \rightarrow A \rightarrow \Lambda \rightarrow A)\\
&\rightarrow \Lambda \rightarrow A
\end{aligned}
$$

Inherits α-compatibility property from the iteration principle.

Applications

In the following slides we will show some classic results of the λ-calculus meta-theory that can be formalized with our principles.

Free variables

We use the iteration principle to define the free variables function.

```
fv : Λ → List Atom
fv = ΛIt (List Atom) [_] _++_ ([] , λ v r → r - v)
```

```
lemma∼αfv : {M N : Λ} → M ∼α N → fv M ≡ fv N
lemma∼αfv {M} {N}
  = lemmaΛItStrongαCompatible
      (List Atom) [_] _++_ [] (λ v r → r - v) M N
```

Soundness of the fv function

$$\text{Pfv*} : \text{Atom} \rightarrow \Lambda \rightarrow \text{Set}$$
$$\text{Pfv*}\ a\ M = a \in \text{fv}\ M \rightarrow a * M$$

$$
\begin{array}{llll}
\multicolumn{4}{l}{\text{data}\ \_*\_ : \text{Atom} \rightarrow \Lambda \rightarrow \text{Set where}} \\
\quad *\text{v} & : \{x : \text{Atom}\} & & \rightarrow x * \text{v}\ x \\
\quad *{\cdot}\text{l} & : \{x : \text{Atom}\}\{M\ N : \Lambda\} & \rightarrow x * M & \rightarrow x * (M \cdot N) \\
\quad *{\cdot}\text{r} & : \{x : \text{Atom}\}\{M\ N : \Lambda\} & \rightarrow x * N & \rightarrow x * (M \cdot N) \\
\quad *\lambda & : \{x\ y : \text{Atom}\}\{M : \Lambda\} & \rightarrow x * M \rightarrow y \not\equiv x & \rightarrow x * (\lambda\ y\ M)
\end{array}
$$

Soundness of the fv function

$$\mathsf{Pfv*} : \mathsf{Atom} \to \Lambda \to \mathsf{Set}$$
$$\mathsf{Pfv*}\ a\ M = a \in \mathsf{fv}\ M \to a * M$$

$$
\begin{array}{llll}
\mathsf{data}\ \_*\_ : \mathsf{Atom} \to \Lambda \to \mathsf{Set}\ \mathsf{where} & & & \\
\quad *\mathsf{v} & : \{x : \mathsf{Atom}\} & & \to x * \mathsf{v}\ x \\
\quad *{\cdot}\mathsf{l} & : \{x : \mathsf{Atom}\}\{M\ N : \Lambda\} & \to x * M & \to x * (M \cdot N) \\
\quad *{\cdot}\mathsf{r} & : \{x : \mathsf{Atom}\}\{M\ N : \Lambda\} & \to x * N & \to x * (M \cdot N) \\
\quad *\bar{\lambda} & : \{x\ y : \mathsf{Atom}\}\{M : \Lambda\} & \to x * M \to y \not\equiv x & \to x * (\bar{\lambda}\ y\ M)
\end{array}
$$

We can define $\_*\_$ relation in the following equivalent way:

$$\_\mathsf{free}\_ : \mathsf{Atom} \to \Lambda \to \mathsf{Set}$$
$$(\_\mathsf{free}\_)\ a = \Lambda\mathsf{It}\ \mathsf{Set}\ (\lambda\ b \to a \equiv b)\ \_\uplus\_\ ([\ a\ ]\ ,\ \lambda\ \_ \to \mathsf{id})$$

Now $\_\mathsf{free}\_$ is α-compatible by definition, by being defined with our iteration principle.
As fv is strong α-compatible, then Pfv* is α-compatible!

Soundness of the fv function

Abstraction case sketch of the α-structural proof.

$ih)$ $\mathrm{Pfv}*(M) \equiv \ a \in \mathrm{fv}\ \ M \Rightarrow a * M$
$it)$ $\mathrm{Pfv}*(\lambda\, x\, M) \equiv \ a \in \mathrm{fv}\ \ \lambda\, x\, M \Rightarrow a * \lambda\, x\, M$

<u>Proof:</u>

$a \in \mathrm{fv}\ \ \lambda\, x\, M \overset{iterationpr.}{\Rightarrow} a \in (\mathrm{fv}\ \ (x \bullet b)\, M) - b$, with b fresh in $\lambda\, x\, M$

By (-) operation prop. $\Rightarrow b \neq a \ \wedge a \in \mathrm{fv}\ \ (x \bullet b)\, M$

We can restrict $x \ / x \neq a \overset{swap-*prop.}{\Rightarrow} a \in \mathrm{fv}\, M \overset{ih)}{\Rightarrow} a * M \overset{x \neq a}{\Rightarrow} a * \lambda\, x\, M$

Soundness of the fv function

Abstraction case sketch of the α-structural proof.

$ih)$ Pfv*$(M) \equiv \ a \in \text{fv} \ \ M \Rightarrow a * M$
$it)$ Pfv*$(\lambda\, x\, M) \equiv \ a \in \text{fv} \ \ \lambda\, x\, M \Rightarrow a * \lambda\, x\, M$

<u>Proof:</u>

$a \in \text{fv} \ \ \lambda\, x\, M \overset{iteration pr.}{\Rightarrow} a \in (\text{fv} \ \ (x \bullet b)\, M) - b,$ with b fresh in $\lambda\, x\, M$

By (-) operation prop. $\Rightarrow b \neq a \ \wedge a \in \text{fv} \ \ (x \bullet b)\, M$

We can restrict $x \ / x \neq a \overset{swap-*prop.}{\Rightarrow} a \in \text{fv}\, M \overset{ih)}{\Rightarrow} a * M \overset{x \neq a}{\Rightarrow} a * \lambda\, x\, M$

Substitution

```
hvar : Atom → Λ → Atom → Λ
hvar x N y with x ≟ₐ y
... | yes _ = N
... | no  _ = v y
-
_[_:=_] : Λ → Atom → Λ → Λ
M [ a := N ] = Λit Λ (hvar a N) _·_ (a :: fv N , ƛ) M
```

Substitution

```
hvar : Atom → Λ → Atom → Λ
hvar x N y with x ≟ₐ y
... | yes _ = N
... | no  _ = v y
-
_[_:=_] : Λ → Atom → Λ → Λ
M [ a := N ] = ΛIt Λ (hvar a N) _·_ (a ∷ fv N , ƛ) M

lemmaSubst1 : {M N : Λ}(P : Λ)(a : Atom)
  → M ∼α N
  → M [ a := P ] ≡ N [ a := P ]
lemmaSubst1 {M} {N} P a
  = lemmaΛItStrongαCompatible
        Λ (hvar a P) _·_ (a ∷ fv P) ƛ M N
```

Substitution

The substitution is alpha-equivalent to a naïve substitution when no variable capture exists.

$$\begin{aligned}&\mathsf{lemmaƛ{\sim}[]} : \forall\ \{a\ b\ P\}\ M \to b \notin a :: \mathsf{fv}\ P\\ &\quad \to ƛ\ b\ M\ [\ a := P\ ]\ \sim\alpha\ \ ƛ\ b\ (M\ [\ a := P\ ])\end{aligned}$$

Substitution

The substitution is alpha-equivalent to a naïve substitution when no variable capture exists.

$$\text{lemma}ƛ{\sim}[\,] : \forall \{a\ b\ P\}\ M \to b \notin a :: \text{fv}\ P$$
$$\to ƛ\ b\ M\ [\ a := P\ ] \sim\alpha\ ƛ\ b\ (M\ [\ a := P\ ])$$

With the previous result and the α-structural induction principle we can do any classic pen-and-paper proofs about substitution.

Substitution

The substitution is alpha-equivalent to a naïve substitution when no variable capture exists.

$$\text{lemmaƛ}\!\sim\![\,] : \forall \{a\ b\ P\}\ M \to b \notin a :: \mathsf{fv}\ P$$
$$\to ƛ\ b\ M\,[\ a := P\ ] \sim\alpha\ ƛ\ b\ (M\,[\ a := P\ ])$$

With the previous result and the α-structural induction principle we can do any classic pen-and-paper proofs about substitution.

For example, the next classic result:

$$\mathsf{PSC} : \forall\ \{x\ y\ L\}\ N \to \Lambda \to \mathsf{Set}$$
$$\mathsf{PSC}\ \{x\}\ \{y\}\ \{L\}\ N\ M = x \not\equiv y \to x \notin \mathsf{fv}\ L$$
$$\to (M\,[\ x := N\ ])\ [\ y := L\ ] \sim\alpha\ (M\,[\ y := L\ ])[\ x := N\,[\ y := L\ ]\ ]$$

Substitution

Abstraction case of the α-structural induction, choosing $b \notin [y] \ +\!\!+ \ fv \ N \ +\!\!+ \ fv \ N[y := L]$

```
begin
   (ƛ b M [ x := N ])   [ y := L ]
- Inner substitution is α equivalent
- to a naive one because b ∉ x :: fv N
≈⟨ lemmaSubst1 L y (lemmaƛ∼[] M b∉x::fvN)  ⟩
   (ƛ b (M [ x := N ])) [ y := L ]
- Outer substitution is α equivalent
- to a naive one because b ∉ y :: fv L
∼⟨ lemmaƛ∼[] (M [ x := N ]) b∉y::fvL ⟩
   ƛ b ((M [ x := N ])  [ y := L ])
- We can now apply our inductive hypothesis
∼⟨ lemma∼αƛ (IndHip x≢y x∉fvL) ⟩
   ƛ b ((M [ y := L ])  [ x := N [ y := L ] ])
- Outer substitution is α equivalent
- to a naive one because b ∉ x :: fv N [y := L]
∼⟨ σ (lemmaƛ∼[] (M [ y := L ]) b∉x::fvN[y:=L]) ⟩
   (ƛ b (M [ y := L ])) [ x := N [ y := L ] ]
- Inner substitution is α equivalent
- to a naive one because b ∉ y :: fv L
≈⟨ sym (lemmaSubst1 (N [ y := L ])  x (lemmaƛ∼[] M b∉y::fvL))  ⟩
   (ƛ b M [ y := L ])   [ x := N [ y := L ] ]
□
```

Summary

We present an induction principle that mimics Barenregt´s convention for α-compatible predicates, which allows us to choose the bound name in the abstraction case so that it does not belong to a given list of names.

Summary

We present an induction principle that mimics Barenregt´s convention for α-compatible predicates, which allows us to choose the bound name in the abstraction case so that it does not belong to a given list of names.

We derive a recursion principle which defines strong α-compatible functions.

Summary

We present an induction principle that mimics Barenregt´s convention for α-compatible predicates, which allows us to choose the bound name in the abstraction case so that it does not belong to a given list of names.

We derive a recursion principle which defines strong α-compatible functions.

All results are derived from the first primitive recursion, and no induction on the length of terms or accessible predicates were needed.

Summary

We present an induction principle that mimics Barenregt´s convention for α-compatible predicates, which allows us to choose the bound name in the abstraction case so that it does not belong to a given list of names.

We derive a recursion principle which defines strong α-compatible functions.

All results are derived from the first primitive recursion, and no induction on the length of terms or accessible predicates were needed.

With this basic framework we are able to reproduce classic pen-and-paper proofs in a formal proof assistant.

Thank you!

Questions ?