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).

- We are interested in studying reasoning techniques over programming languages.
- Besides we want to formalise theses techniques in Constructive Type Theory.
- Specifically, we choose as our object of study the λ-calculus
- and Agda as proof assistant and programming language where we will develop some λ-calculus meta-theory.
- But we do not want to diverge from classical pen-and-paper proofs.
- So we study the most direct formalisation: the original syntax with one sort of variables.
- We take the swapping operation from the nominal approach
- to define α-conversion and substitution.

Reasoning over α-equivalence classes

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).

- Most of *lambda* calculus's meta-theory implicitly work over α-equivalences classes of terms.
- This common practise even has a name: Barendregt´s Variable convention, that says:
- This practice often doesn´t make explicit:
  - A definition of substitution that avoids variable capture is needed
  - for some proofs an induction over the size of the terms must be made
  - And That the properties being proved should be preserved under α-conversion:
  - we call this α-compatibility of the properties.

Reasoning over α-equivalence classes

We want to be able to reproduce the following induction sketch

λ case:

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

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

- We want to reproduce the following induction sketch:
  - instead of proving the classic abstraction case
  - we can give some finite set of variables from where the abstraction variable will not be chosen
  - and then for an α-equivalent renaming of the term with any variable not in the chosen context
  - we prove the induction step.

Variable Swaping

```
(_•_)a_ : Atom → Atom → Atom → Atom
( a • b )a c  with c ≟a a
... | yes _           = b
... | no  _ with c ≟a b
...         | yes _   = a
...         | no  _   = c
```

- From this slide until the end of this presentation we will directly show Agda's code of our formalisation.
- We begin with the basis of our formalization: which is the swapping operation.
- Next we show the swapping operation over variables, which is a ternary funcion that takes two atoms to be swapped in a third one
- and is defined by cases

Terms

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

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

- Also changes bound variables ⇒ no variable capture
  $(y\ x) \bullet (\lambda\ x.x\ y) = \lambda\ y.y\ x$
- Injective
  ```
  lemma•ₐinj : {a b c d : Atom}
             → c ≢ d
             → ( a • b )ₐ c ≢ ( a • b )ₐ d
  ```
- Idempotent
  ```
  lemma(ab)(ab)c≡c : {a b c : Atom}
     → ( a • b )ₐ ( a • b )ₐ c ≡ c
  ```
  $(y\ x) \bullet (\lambda\ y.y\ x) = \lambda\ x.x\ y$

- Lambda terms are directly defined as follows
- The swapping operation is trivially extended to λ-terms.
- And we swap free variables, and also bound variables.
- Swapping is simpler than substitution and with better properties than renaming:
  - because it also changes bound variables, then there is no possible variable capture
  - it is injective
  - it is idempotent: sofor any swapping there always exists the inverse operation !

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)

- We present a syntax directed α-conversion definition.
-
  - in the abstraction rule we have a stronger hypothesis than in the classic definition, that says:
  - $\forall$ variable c not in a given context xs,
  - we have the α-compatibility for the abstraction sub-terms$M$, N with the binder variable swapped with c.
  - it is Inspired in Aydemir's "cofinite quantification"
- This definition fits with Barendregt´s convension, where we choose the bound variable different from some given finite context.
- This definition is easier to use but more difficult to prove
- But This is not a problem because α-compatibility is usually a hypothesis.

Primitive induction

Comes for free with the definition of terms :

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

- This is the classic primitive induction principle where we have to prove the abstraction case for any variable *b*.
- But this induction principle comes for free with the simple syntax definition.

Permutation induction

A permutation π is a sequence of swappings.

We derive this induction principle, from simple structural induction:

```
TermIndPerm : {l : Level}(P : Λ → Set l)
  → (∀ a → P (v a))
  → (∀ M N → P M → P N →  P (M · N))
  → (∀ M b → (∀ π → P (π • M)) → P (ƛ b M))
  → ∀ M → P M
```

- A permutation π is a sequence of swappings.
- We can derive the following induction principle for lambda terms
- Stronger: Note thay the inductive hypothesis of the abstraction case is given for any permutation
- No need of induction on the length of terms!
- Derived in the same way as we can derive the complete induction principle on natural numbers from primitive induction.

α-structural induction

```
αCompatiblePred : {l : Level} → (Λ → Set l) → Set l
αCompatiblePred P = {M N : Λ} → M ∼α N → P M → P N
```

We can derive this induction principle from the former one:

```
TermαPrimInd : {l : Level}(P : Λ → Set l)
  → αCompatiblePred P
  → (∀ a → P (v a))
  → (∀ M N → P M → P N → P (M · N))
  → ∃ (λ vs → (∀ M b → b ∉ vs → P M → P (ƛ b M)))
  → ∀ M → P M
```

Now we can emulate the BVC!

- We define what is to be an α-compatible predicate: that is, the predicate is preserved for α-**equivalent** terms
- And then we obtain the next induction principle directly from the previous one and we get our goal.
- Not that in the abstraction case of we can exclude some finite set of variables

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

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

- We also derive an iteration principle
- In the abstraction case of this principle:
  - we give some list of variables to be excluded
  - and a function that given a selected variable not in the list (to be choosen as the abstraction variable)
  - and the result of the recursive call of the iteration
  - constructs a result

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
```

- Now we define what it means that a function is strongα-compatible:
- That is, for any two α-equivalent terms the result of the function is the same.
- The following lema states that our iteration principle always defines strongα-compatible funcions
- This is a direct consequence of the fact that the recursive call is done over a renamed sub-term
- Note that because of this, the iteration principle has no way of extracting any information from bound variables

α-structural recursion

Using the last iteration principle we define a recursion principle:

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

Inherits α-compatibility property from the iteration principle.

- We generalize a recursion principle from the iterative one.
- It inherits α-compatibility

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
```

- As an example we use the iteration principle to define the free variables function.
- for the variable case we give the singleton list constructor
- for the application we junt concatenate the recursive calls
- for the abstraction case, given the abstraction variable and the recursive call, we just return the recursive call minus the variable
- The function is α-compatible by definition

Soundness of the fv function

```
Pfv* : Atom → Λ → Set
Pfv* a M = a ∈ fv M → a * M

data _*_ : Atom → Λ → Set where
  *v  : {x : Atom}                              → x * v x
  *·l : {x : Atom}{M N : Λ} → x * M             → x * (M · N)
  *·r : {x : Atom}{M N : Λ} → x * N             → x * (M · N)
  *ƛ  : {x y : Atom}{M : Λ} → x * M → y ≢ x     → x * (ƛ y M)
```

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

```
_free_ : Atom → Λ → Set
(_free_) a = ΛIt Set (λ b → a ≡ b) _⊎_ ([ a ] , λ _ → id)
```

Now _free_ is α-compatible by definition, by being defined with our iteration principle.
As fv is strong α-compatible, then Pfv* is α-compatible!

- We want to prove that the function we have just defined is sound.
- First we define the property to be proven: Pfv* states that if a variable belongs to the result of fv applied to a term, then it is fee in that term
- This is thee classical definition of free variable
- We can also use our iteration principle to define this predicate
- For the variable case, it is a function that for a variable b it is returned a Set inhabited by a proof that $a \equiv b$.
- The application case is the disjoint union of the recursive calls result sets.
- Finally the abstraction case, as we can choose the binder variable not equal to a, so the set only inhabited by the proof that a is free in the abstraction sub-term is directly a proof that a is free in the abstraction.
- And this predicate is α-compatible by definition!!!

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>

$a \in \mathrm{fv}\ \lambda x\, M \overset{iteration pr.}{\Rightarrow} a \in (\mathrm{fv}\ (x \bullet b)\, M) - b$, 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 $x\ / x \neq a \overset{swap-*prop.}{\Rightarrow} a \in \mathrm{fv}\, M \overset{ih)}{\Rightarrow} a * M \overset{x \neq a}{\Rightarrow} a * \lambda x\, M$

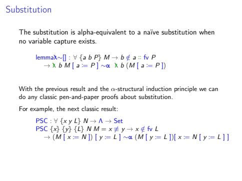
- We can use our induction principle that mimics Barenregt´s convention convention because free predicate is α-compatible.
- In the abstration clase, which is the interesting one, note how our principle makes the proof very short and direct:
- ei belongs to the free variables of lambda ex eM, then by being defined with the iteration principle,
- ei belongs to the free variables of swapping ex with bi in eM
- Now we can assume ex not equal to ei, and then by a swapping-aster property ei belongs to the free variables of eM,
- finally, using again that ex is not not equal to ei, ei is free in lambda ex eM
- The proof does not need to assume $x \neq ei$, it can be proved, but would require several more steps and properties in the proof.

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
```

- Next we define the substitution operation on terms using the iteration principle.
- The variable case is defined with the hvar function, which replaces a variable by a term
- The application case is just the application constructor applied to the recursive subtitution results.
- In the abstraction case we can exclude the variable being susbtituted and the free variables of the term susbtituted to avoid any variable capture.
- then we can proceed as in the application case, returning the abstraction constructor appied to the bound variable and the recursive substitution result.
- This definition is α-compatible because it was defined with the iteration principle.



- The result of substitution is alpha-equivalent to a naive substitution when no variable capture exists.
- With the previous result and the α-structural induction principle we can prove any result about substitution and terms as in classic pen-and-paper proofs.
- For example, the next classic result about composition of substitutions for fresh enough variables

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 ] ]
□
```

- In the abstraction case we can do the following direct equational proof, identical to a classical pen-and-paper proof, for a bi abstraction variable enough fresh.

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich.
Engineering formal metatheory.
In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 3–15, New York, NY, USA, 2008. ACM.