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 op. 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/programming language where we will develop 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 names.
- And we take the swapping operation from the nominal approach. Based in this operation we define α-conversion and substitution.

Reasoning over α-equivalence classes

Barenregt's variable convention (BVC)

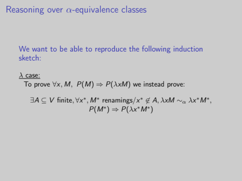
Each λ-term represents its α class, so it could be assumed to have bound and context variables all different.

A complete formilisation often implies:

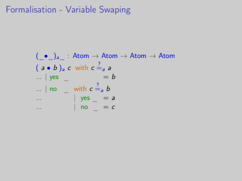
- Define a no capture substitution operation.
- Complete induction over the *size* of terms is needed to fill the gap between terms and α-equivalence classes.
- Prove that all properties being proved are preserved under α-conversion (α-compatibe predicates).

- 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.
- This practice often doesn´t make explicit:
  - A definition of substitution that avoids variable capture
  - That induction must be made over the size of the terms
  - That the properties being proved are preserved under α-conversion:
  - we call this α-compatibility of the predicates.

└─Reasoning over α-equivalence classes



- We want to reproduce the following induction sketch:
  - instead of proving the classic abstraction case
  - we 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 classic induction step.

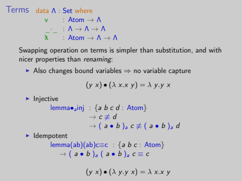


- From this slide until the end of the presentation we will directly show Agda's code.

- 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

- it is defined by cases



- Lambda terms are directly defined as follows
- The swapping operation is trivially extended to λ-terms.
- Note that 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 variable capture
  - it is injective
  - it is idempotent: for any swapping always there 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, [1])

- We present a syntax directed α-conversion definition.
- Inspired in "cofinite quantification":
  - in the abstraction rule we have a stronger hypothesis than in the classic definition
  - $\forall$ 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.
- 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 This is not a problem because α-compatibility is usually a hypothesis.

Primitive induction

Comes for free with terms definition:

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

- In the following slides we will iterate over several induction principles, until we get the desired one, that is, one which capture Barenregt convention.
- This is thee 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

```
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
```

Derived from the former induction principle, i.e. from simple structural induction!!

- In the same way as we can derive the complete induction principle on natural numbers from primitive induction.
- We can derive the following induction principle for lambda terms
- No need of induction on the length of terms!
- Note thay the inductive hypothesis of the abstraction case is given for any permutation (sequences of swapings)

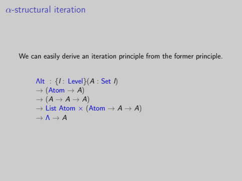
α-structural induction

Now we can emulate BVC.

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

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
```

- We define what is to be an α-compatible predicate
- 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



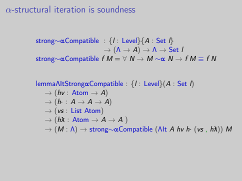
α-structural iteration

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

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

- 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 given list
  - and the recursive result of the iteration over the abstraction sub-term renamed with the fresh variable, construct a result



- Now we define what it means that a function is strongα-compatible:
- That is, for any two α-convertible 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:

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

Inherits α-compatibility property from the iteration principle.

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

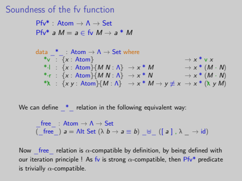
Applications - Free variables

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



- We want to prove that the function we have just defined is sound.
- for this we define an inductive predicate * that states when a veriable is free in a term
- The 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
- We can return Set in this example because of we use levels in our induction/recursion principles.
- 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.

Soundness of the fv function

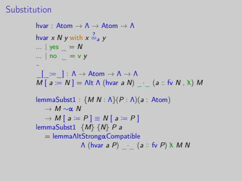
Abstraction case sketch of the α-structural proof.

$hi)$ $\mathrm{Pfv}^*(M) \equiv\ a \in \mathrm{fv}\ M \Rightarrow a * M$
$ti)$ $\mathrm{Pfv}^*(ƛ\, x\, M) \equiv\ a \in \mathrm{fv}\ ƛ\, x\, M \Rightarrow a * ƛ\, x\, M$

<u>Proof:</u>

$a \in \mathrm{fv}\ ƛ\, x\, M \overset{iterationpr.}{\Rightarrow} a \in (\mathrm{fv}\ (x \bullet b)\, M) - b$, with b fresh in $ƛ\, x\, M$
By (-) operation prop. $\Rightarrow b \neq a\ \wedge a \in \mathrm{fv}\ (x \bullet b)\, M$
We can restrict $x\ / x \neq a \overset{swap-*prop.}{\Rightarrow} a \in \mathrm{fv}\, M \overset{hi)}{\Rightarrow} a * M \overset{x \neq a}{\Rightarrow} a * ƛ\, x\, M$

- We can use our induction principle that mimics Barenregt 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 belong to the free variables of lambda ex eM, then by being defined with the iteration principle,
- ei belong to the free variables of swapping ex with bi in M
- now we can assume ex not equal to ei, then by a swapping-aster property ei belong to the free variables of eM,
- finally, using again that ex not equal to ei, a is free in lambda ex eM
- The proof does not need to assume $x \neq a$, it can be proved, but it requires several more steps and properties in the proof.



- Next we define the substitution operation in terms using the iteration principle.
- The variable case is defined with the hvar function, which replaces a variable by a term in some variable.
- 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 by being defined with the iteration

Substitution

Next we show the interesting 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 ] ]
□
```

- We only show the proof to show that is a equational one, identical to a classical pen-and-paper proof.

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.