Functions

jean-jacques.levy@inria.fr
August 5, 2013

Plan

• functions and λ-notation
• higher-order functions
• data types
• notation in Coq
• enumerated sets
• pattern-matching on constructors

Functional calculus (1/6)

\( (\lambda x. x + 1)3 \rightarrow 3 + 1 \rightarrow 4 \)
\( (\lambda x. 2 * x + 2)4 \rightarrow 2 * 4 + 2 \rightarrow 8 + 2 \rightarrow 10 \)
\( (\lambda f.f3)(\lambda x. x + 2) \rightarrow (\lambda x. x + 2)3 \rightarrow 3 + 2 \rightarrow 5 \)
\( (\lambda x.\lambda y.x + y)3 \rightarrow (\lambda y.3 + y)2 \rightarrow (\lambda y.3 + y)2 \rightarrow 3 + 2 \rightarrow 5 \)
\( (\lambda f.\lambda x.f(f x))(\lambda x.x + 2) \rightarrow \ldots \)
Functional calculus (2/6)

\[(\lambda f. \lambda x. f(fx))(\lambda x. x + 2) \rightarrow \ldots\]

Functional calculus (3/6)

\[(\lambda f. \lambda x. f(fx))(\lambda x. x + 2)^3 \rightarrow \ldots\]

Functional calculus (5/6)

Fact(3)

\[\text{Fact} = Y(\lambda f. \lambda x. \text{if} \ x \ \text{then} \ 1 \ \text{else} \ x \ * \ f(x - 1))\]

Thus following term:

\[(\lambda \text{Fact}. \text{Fact}(3))\]

\[(Y(\lambda f. \lambda x. \text{if} \ x \ \text{then} \ 1 \ \text{else} \ x \ * \ f(x - 1)))\]

also written

\[(\lambda \text{Fact}. \text{Fact}(3))\]

\[( (\lambda Y. Y(\lambda f. \lambda x. \text{if} \ x \ \text{then} \ 1 \ \text{else} \ x \ * \ f(x - 1)))\]

\[(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))) \]
Pure lambda-calculus

- **lambda-terms**

  \[ M, N, P \ ::= \ x, y, z, \ldots \quad \text{(variables)} \]
  \[ | \quad \lambda x. M \quad \text{(M as function of x)} \]
  \[ | \quad M (N) \quad \text{(M applied to N)} \]

- **Computations “reductions”**

  \[(\lambda x. M)(N) \rightarrow M[x := N]\]

Examples of reductions (1/2)

- **Examples**

  \[(\lambda x. x) N \rightarrow N\]
  \[(\lambda f. N)(\lambda x. x) \rightarrow (\lambda x. x) N \rightarrow N\]
  \[(\lambda x. x) N(\lambda y. y) \rightarrow (\lambda y. y) N \rightarrow N \quad \text{(name of bound variable is meaningless)}\]
  \[(\lambda x. x)(\lambda x. x) N \rightarrow (\lambda x. x) N \rightarrow N \rightarrow NN\]
  \[(\lambda x. x)(\lambda x. x) \rightarrow \lambda x. x\]

Let \( I = \lambda x. x \), we have \( I(x) = x \) for all \( x \).
Therefore \( I(I) = I \). [Church 41]

Examples of reductions (2/2)

- **Examples**

  \[(\lambda x. x)(\lambda x. x) N \rightarrow (\lambda x. x) N \rightarrow (\lambda x. x) N \rightarrow NN\]
  \[(\lambda x. x)(\lambda x. x) \rightarrow (\lambda x. x)(\lambda x. x) \rightarrow \cdots\]

- **Possible to loop inside applications of functions ...**

  \[ Y_f = (\lambda x. f(x)(x)) \rightarrow f((\lambda x. f(x)(x))(\lambda x. f(x)(x))) = f(Y_f)\]
  \[ f(Y_f) \rightarrow f(f(Y_f)) \rightarrow \cdots \rightarrow f^n(Y_f) \rightarrow \cdots\]

- **Every computable function can be computed by a \( \lambda \)-term**

  \[ \rightarrow \text{Church’s thesis. [Church 41]} \]
**Fathers of computability**

Alonzo Church  
Stephen Kleene

**The Giants of computability**

Hilbert → Gödel → Church → Turing  
Kleene → Post → Curry  
von Neumann

**Typed lambda-calculus (1/5)**

- In Coq, all $\lambda$-terms are *typed*
- In Coq, following $\lambda$-terms are typable
  
  \[
  (\lambda x. x + 1)3 \rightarrow 3 + 1 \rightarrow 4 \\
  (\lambda x. 2 * x + 2)4 \rightarrow 2 * 4 + 2 \rightarrow 8 + 2 \rightarrow 10 \\
  (\lambda f.f3)((\lambda x. x + 2) \rightarrow (\lambda x. x + 2)3 \rightarrow 3 + 2 \rightarrow 5 \\
  (\lambda x. \lambda y.x + y)3 2 = \\
  ((\lambda x. \lambda y.x + y)3 2 \rightarrow (\lambda y.3 + y)2 \rightarrow (\lambda y.3 + y)2 \rightarrow 3 + 2 \rightarrow 5 \\
  (\lambda f.\lambda x.f(f x))(\lambda x.x + 2) \rightarrow ...
  \]

  these terms are allowed

**Typed lambda-calculus (2/5)**

- In Coq, all $\lambda$-terms have only finite reductions  
  *(strong normalization property)*
- In Coq, all $\lambda$-terms have a (unique) normal form.
- In Coq, the following $\lambda$-terms are not typable

  \[
  (\lambda x.x)(\lambda x.x) \\
  (\lambda \text{Fact.Fact}(3)) \\
  ((\lambda Y.Y(\lambda f.\text{if} z \text{ then } 1 \text{ else } x * f(x - 1))) \\
  (\lambda x.\text{if}(x)(\lambda x.f(x)))
  \]

  these terms are not allowed
Typed lambda-calculus (3/5)

- The Coq laws for typing terms are quite complex
  [Coquand-Huet 1985]

- In first approximation, they are the following (1st-order) rules:
  
  Basic types: \( N \) (nat), \( B \) (bool), \( Z \) (int), \ldots

  If \( x \) has type \( \alpha \), then \( (\lambda x. M) \) has type \( \alpha \to \beta \)

  If \( M \) has type \( \alpha \to \beta \), then \( M(N) \) has type \( \beta \)

Example

\[ 1 : \text{nat} \]
\[ x : \text{nat} \implies x + 1 : \text{nat} \]
\[ (\lambda x. x + 1) : \text{nat} \to \text{nat} \]
\[ 3 : \text{nat} \]
\[ (\lambda x. x + 1)3 : \text{nat} \]

Typed lambda-calculus (4/5)

Example

\[ x : \text{nat} \vdash x : \text{nat} \]
\[ x : \text{nat} \vdash x : \text{nat} \]
\[ x : \text{nat} \vdash x + 1 : \text{nat} \]
\[ \vdash (\lambda x. x + 1) : \text{nat} \to \text{nat} \]
\[ \vdash (\lambda x. x + 1) : \text{nat} \to \text{nat} \]
\[ \vdash 3 : \text{nat} \]
\[ \vdash (\lambda x. x + 1)3 : \text{nat} \]
**lambda-terms (1/3)**

**three equivalent definitions:**
Definition plusOne (x: nat) : nat := x + 1.
Check plusOne.

Definition plusOne := fun (x: nat) => x + 1.
Check plusOne.

Definition plusOne := fun x => x + 1.
Check plusOne.

Compute (fun x:nat => x + 1) 3.

**higher-order definitions:**
Definition plusTwo (x: nat) : nat := x + 2.

Definition twice := fun f => fun (x:nat) => f (f x).

Compute twice plusTwo 3.

**lambda-terms (2/3)**

• Coq tries to guess the type, but could fail.
(type inference)

• but always possible to give explicit types.

• Types can be higher-order
(see later with polymorphic functions)

• Types can also depend on values
(see later the constructor cases)

**lambda-terms (3/3)**

• Coq treats with an extension of the $\lambda$-calculus with inductive data types. It’s a programming language.

• the typed $\lambda$-calculus is also used as a trick to make a correspondence between proofs and $\lambda$-terms and propositions and types for constructive logics (see other lectures).
(Curry-Howard correspondence)