Introduction to Functional Programming (in Coq)

January 8, 2015
Outline

What is Functional Programming?
  What is a function?
  First-order vs. higher-order functions
  Equational reasoning

How to functional program
  Inductive data types
  Defining functions
  Name bindings
A function is **pure** if:

- it always returns the same output for the same inputs
- it doesn’t do anything else — no “side effects”

**In this course: whenever we say “function” we mean a pure function!**
What are and aren’t functions?

Always functions:
- mathematical functions $f(x) = x^2 + 2x + 3$
- lambda abstractions $\lambda x. \lambda y. x y$
- Haskell functions $\text{sum} = \text{foldr} (+) 0$

Usually not functions:
- C, Python, Clojure, … “functions” (procedures)
- Java, C#, Ruby, … methods

Coq only allows you to write (pure) functions!

In fact, Coq only allows you to write obviously terminating functions.
Why procedures/methods aren’t functions

- output depends on environment
- may perform arbitrary side effects
What is Functional Programming?

What is a function?

First-order vs. higher-order functions

Equational reasoning

How to functional program

Inductive data types

Defining functions

Name bindings
First-order functions

Examples

- `pred : nat -> nat`
- `even : nat -> bool`
- `negb : bool -> bool`
Higher-order functions

Examples

- apply : (A -> B) -> A -> B
- flip : (A -> B -> C) -> B -> A -> C
- compose : (B->C) -> (A->B) -> A -> C
Currying / partial application

In Coq, functions that take multiple arguments are implicitly higher order.

\[ \text{plus} : \text{nat} \to \text{nat} \to \text{nat} \]

**Definition** increment : nat -> nat :=

\[ \text{plus one.} \]

Curried
\[ \text{plus} : \text{nat} \to \text{nat} \to \text{nat} \]

plus 2 3

Uncurried
\[ \text{plus} : \text{nat} \times \text{nat} \to \text{nat} \]

plus (2,3)
Outline

What is Functional Programming?
- What is a function?
- First-order vs. higher-order functions
- Equational reasoning

How to functional program
- Inductive data types
- Defining functions
- Name bindings
Referential transparency

An expression can be replaced by its value without changing the overall program behavior.

even (plus zero (succ zero)) = even (succ zero)

**Corollary**: an expression can be replaced by any expression with the same value without changing program behavior.

Supports equational reasoning.
Equational reasoning

Can derive equations from definitions, then reason by substitution

Inductive Nat : Type :=
| zero : Nat
| succ : Nat -> Nat.

Definition one := succ zero.
Definition two := succ one.

Fixpoint plus n m :=
match n with
| zero => m
| succ n' => succ (plus n' m)
end.

Equations

one = succ zero
two = succ one
plus zero m = m
plus (succ n) m = succ (plus n m)

plus one one
plus (succ zero) one
succ (plus zero one)
succ one
two
Outline

What is Functional Programming?
  What is a function?
  First-order vs. higher-order functions
  Equational reasoning

How to functional program
  Inductive data types
  Defining functions
  Name bindings
Functional programming workflow

- Define functions
- Identify/define types
- Refactor

How to functional program
Outline

What is Functional Programming?
  What is a function?
  First-order vs. higher-order functions
  Equational reasoning

How to functional program
  Inductive data types
  Defining functions
  Name bindings
Inductive data types

Data type definition
- Introduces new type of value
- Enumerates ways to construct values of this type

Some pre-defined data types

**Inductive bool : Type :=**
- true : bool
- false : bool.

**Inductive nat : Type :=**
- 0 : nat
- S : nat -> nat.

Definitions consists of …
- A type name
- A list of constructors, each of which has a type

Definition is inductive
- The constructors may recursively refer to the type being defined
- The constructors are the only way to construct values of this type (crucial for inductive proofs)
Exercises

1. Define an inductive data type \texttt{natlist} that represents a list of \texttt{nats}

2. Define terms representing the following lists:
   - \([\text{[]}\]
   - \([0]\]
   - \([1,0,2]\]

The \texttt{nat} data type, for reference:

\begin{verbatim}
Inductive nat : Type :=
  | O : nat
  | S : nat -> nat.
\end{verbatim}
Outline

What is Functional Programming?
   What is a function?
   First-order vs. higher-order functions
   Equational reasoning

How to functional program
   Inductive data types
   Defining functions
   Name bindings
Tools for defining functions

1. Pattern matching
   - Case analysis
   - Decomposition and name binding

```
Definition subTwo (n:nat) : nat :=
  match n with
  | O => O
  | S O => O
  | S (S n') => n'
  end.
```

2. Structural recursion
   - Recursively call this function on a smaller argument

```
Fixpoint sum (l:natlist) : nat :=
  match l with
  | nil => O
  | cons n l' => plus n (sum l')
  end.
```
Exercises

1. Define a recursive function that appends an element to the end of a natlist

2. Define a recursive function that concatenates two natlists

Type definition and example function, for reference:

```
Inductive natlist : Type :=
 | nil  : natlist
 | cons : nat -> natlist -> natlist

Fixpoint sum (l:natlist) : nat :=
 match l with
 | nil    => 0
 | cons n l' => plus n (sum l')
end.
```
More on pattern matching

Case analysis
- Cases are tried **in order**
- Cases must be **complete**
  (a case for every possible value)
- _ matches anything (“don’t care”)
- Can analyze several values at once

```ocaml
Fixpoint leq (l r : nat) : bool :=
  match l, r with
  | O, _ => true
  | _, O => false
  | S l', S r' => leq l' r'
end.
```

Decomposition and name binding
Name on LHS of `=>`:
- If a constructor, match against it
- Otherwise, bind name to corresponding part of value in RHS

```ocaml
Definition isPos (n:nat) : bool :=
  match n with
  | zero => false
  | S _ => true
end.
```

This code is **wrong** – why?
More on structural recursion

All recursive functions in Coq must terminate!
- Needed for decidability of the type system
- Structural recursion is a **heuristic** for ensuring this

**Structural recursion**
- At least one argument must be **decreasing**
- All recursive calls are on a strict **sub-term** of that argument
- Eventually the function will trigger a base case
Outline

What is Functional Programming?
- What is a function?
- First-order vs. higher-order functions
- Equational reasoning

How to functional program
- Inductive data types
- Defining functions
- Name bindings
Name bindings (variables)

**Terminology (not Coq-specific)**

- **declaration**: the (one) place where a name is defined/introduced
- **reference**: the (possibly many) places where a name is used
- **scope**: the region of code in which a name can be referenced

Because Coq has **referential transparency**, a name …

- can **always** be replaced by its value
- will **always** refer to the same value throughout a single evaluation of its scope (no “variable assignment”)

How to functional program
Name binding constructs

**Top-level definitions**

**Definition** \( f \ x \ := \ body. \)
- scope of \( x \): \( body \)
- scope of \( f \): rest of program

**Fixpoint** \( f \ x \ := \ body. \)
- scope of \( x \): \( body \)
- scope of \( f \): \( body \) + rest of program

**Match expressions**

\[
\text{match } \text{expr} \text{ with } \\
| \text{cons} \ x \ \text{xs} \Rightarrow \text{cons-scope} \\
| \text{nil} \Rightarrow \text{nil-scope} \\
\text{end.}
\]
- scope of \( x \) and \( \text{xs}: \text{cons-scope} \)

**Local definitions**

**let** \( f \ x \ := \ \text{bound} \ \text{in} \ \text{scope} \\
- scope of \( x \): \( \text{bound} \)
- scope of \( f \): \( \text{scope} \)
Name shadowing

What happens when you declare a name that is already in scope?

- **most local** declaration takes precedence
- original variable remains **unchanged**

What is the result of this expression?

```plaintext
let x := 3 in x + (let x := 10 in x + x) - x
```

Answer: **20**  (last `x` is still bound to `3`)

---

How to functional program