Recursion and Intro to Coq

Armando Solar Lezama
Computer Science and Artificial Intelligence Laboratory
M.I.T.

With content from Arvind and Adam Chlipala. Used with permission.

September 21, 2015

Recursion and Fixed Point Equations

Recursive functions can be thought of as solutions of fixed point equations:

fact = λn. Cond (Zero? n) 1 (Mul n (fact (Sub n 1)))

Suppose

H = λf.λn.Cond (Zero? n) 1 (Mul n (f (Sub n 1)))

then

fact = H fact

fact is a *fixed point* of function H!

Fixed Point Equations

$$f : D \to D$$

A fixed point equation has the form

$$f(x) = x$$

Its solutions are called the *fixed points* of f because if x_p is a solution then

$$x_p = f(x_p) = f(f(x_p)) = f(f(f(x_p))) = \ldots$$

We want to consider fixed-point equations whose solutions are functions, i.e., sets that contain their function spaces

domain theory, Scottary, ...

An example

Consider

f n = *if* n=0 *then* 1
else (*if* n=1 *then* f 3 *else* f (n-2))

H = λf.λn.Cond(n=0 , 1, Cond(n=1, f 3, f (n-2))

Is there an f_p such that $f_p = H\ f_p$?

f1 n	= 1	if n is even
	= ⊥	otherwise

f2 n	= 1	if n is even
	= 5	otherwise

f1 contains no arbitrary information and is said to be the least fixed point (lfp)

Under the assumption of *monotonicity* and *continuity* least fixed points are unique and computable

Computing a Fixed Point

- Recursion requires repeated application of a function
- Self application allows us to recreate the original term
  - Consider: $\Omega = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$
  - Notice β-reduction of Ω leaves Ω : $\Omega \rightarrow \Omega$
- Now to get $F\ (F\ (F\ (F\ \ldots)))$ we insert F in Ω:

$$\Omega_F = (\lambda x.F\ (x\ x))\ (\lambda x.F\ (x\ x))$$

which β-reduces to:

$$\Omega_F \rightarrow F(\lambda x.\ F(x\ x))(\lambda x.\ F(x\ x))$$
$$\rightarrow F\ \Omega_F \rightarrow F(F\ \Omega_F) \rightarrow F(F(F\ \Omega_F)) \rightarrow \ldots$$

- Now λ –abstract F to get a Fix-Point Combinator:

$$Y \equiv \lambda f.(\lambda x.\ (f\ (x\ x)))\ (\lambda x.(f\ (x\ x)))$$

Y : A Fixed Point Operator

$$Y \equiv \lambda f.(\lambda x.\ (f\ (x\ x)))\ (\lambda x.(f\ (x\ x)))$$

Notice

$$\begin{aligned} Y\ F \quad & \rightarrow (\lambda x.F\ (x\ x))\ (\lambda x.F\ (x\ x)) \\ & \rightarrow F\ (\lambda x.F\ (x\ x))\ (\lambda x.F\ (x\ x)) \\ & \rightarrow F\ (Y\ F) \end{aligned}$$

$F\ (Y\ F) = Y\ F$ (Y F) is a fixed point of F

Y computes the least fixed point of any function !

There are many different fixed point operators.

Mutual Recursion

```
odd  n = if n==0 then False else even (n-1)
even n = if n==0 then True  else odd  (n-1)
```

odd = H_1 even
even = H_2 odd
where
$H_1 = \lambda f.\lambda n.\mathrm{Cond}(n=0, \mathrm{False}, f(n-1))$
$H_2 = \lambda f.\lambda n.\mathrm{Cond}(n=0, \mathrm{True}, f(n-1))$

substituting "H_2 odd" for even
odd = H_1 (H_2 odd)
= H odd *where* H = $\lambda f.\ H_1\ (H_2\ f)$
$\Rightarrow$ odd = Y H

Can we express odd using Y ?

Self-application and Paradoxes

Self application, i.e., (x x) is dangerous.

Suppose:

$u \equiv \lambda y.\ \textit{if}\ (y\ y) = a\ \textit{then}\ b\ \textit{else}\ a$

What is (u u) ?

$(u\ u) \rightarrow \textit{if}\ (u\ u) = a\ \textit{then}\ b\ \textit{else}\ a$

Contradiction!!!

Any semantics of λ-calculus has to make sure that functions such as u have the meaning $\perp$, i.e. "totally undefined" or "no information".

Self application also violates *every* type discipline.

Intro to Coq

Warning

I am not a Coq Expert

So if I can do it, you can do it too!

Formal Reasoning About Programs

- New course Prof. Adam Chlipala will teach next semester
- An introduction to a spectrum of techniques for rigorous mathematical reasoning about correctness of software, emphasizing commonalities across approaches.
- Taught around a formalization of all the different correctness approaches with the Coq proof assistant
- Will go into depth into different program logics, different approaches to formalize concurrency, behavioral refinement of interacting modules, etc.

Some useful references

- The reference manual isn't bad:
  - http://coq.inria.fr/distrib/current/refman/
- Prof. Chlipala's book Certified Programming with Dependent Types
  - A draft is available online (http://adam.chlipala.net/cpdt/)
  - most of what it covers goes beyond the scope of 6.820.
- Another popular book: Bertot & Casteran, Interactive Theorem Proving and Program Development (Coq'Art)
  - https://www.labri.fr/perso/casteran/CoqArt/
- A popular online book that uses Coq to introduce ideas in semantics: Software Foundations by Pierce et al.
  - http://www.cis.upenn.edu/~bcpierce/sf/

Key ideas

- Introduce Definitions and theorems
- Prove them by applying simple deductive steps called *tactics*

Example: Defining Natural numbers

```
Inductive nat := O | S (n : nat).
Fixpoint plus (n m : nat) : nat :=
  match n with
    | O => m
    | S n' => S (plus n' m)
  end.
```

Just a familiar ADT and Recursive Function Definition

Proving theorems with tactics

- Basic syntax to introduce lemmas and theorems
  - Lemma O_plus : forall n,
    plus O n = n.
    Proof.
    (* Sequence of tactics *)
    Qed.

- Lemma and Theorem are interchangeable (You can also say Remark, Corollary, Fact or Proposition)

Tactics

- They instruct Coq on the steps to take to prove a theorem
- reflexivity
  - prove an equality goal that follows by normalizing terms.
- induction x
  - prove goal by induction on quantified variable [x]
  - Structural Induction: X is any recursively defined structure
  - All variables appearing _before_ [x] will remain _fixed_ throughout the induction!

More tactics

- simpl
  - apply standard heuristics for computational simplification in conclusion.
  - Often it will involve doing some β reduction
- rewrite H
  - use (potentially quantified) equality [H] to rewrite in the conclusion.
- intros
  - move quantified variables and/or hypotheses "above the double line.
- apply thm
  - apply a named theorem, reducing the goal into one new subgoal for each of the theorem's hypotheses, if any.

And a few more

- assumption
  - Prove a conclusion that matches a known hypothesis.
- destruct E
  - Do case analysis on the constructor used to build term [E].