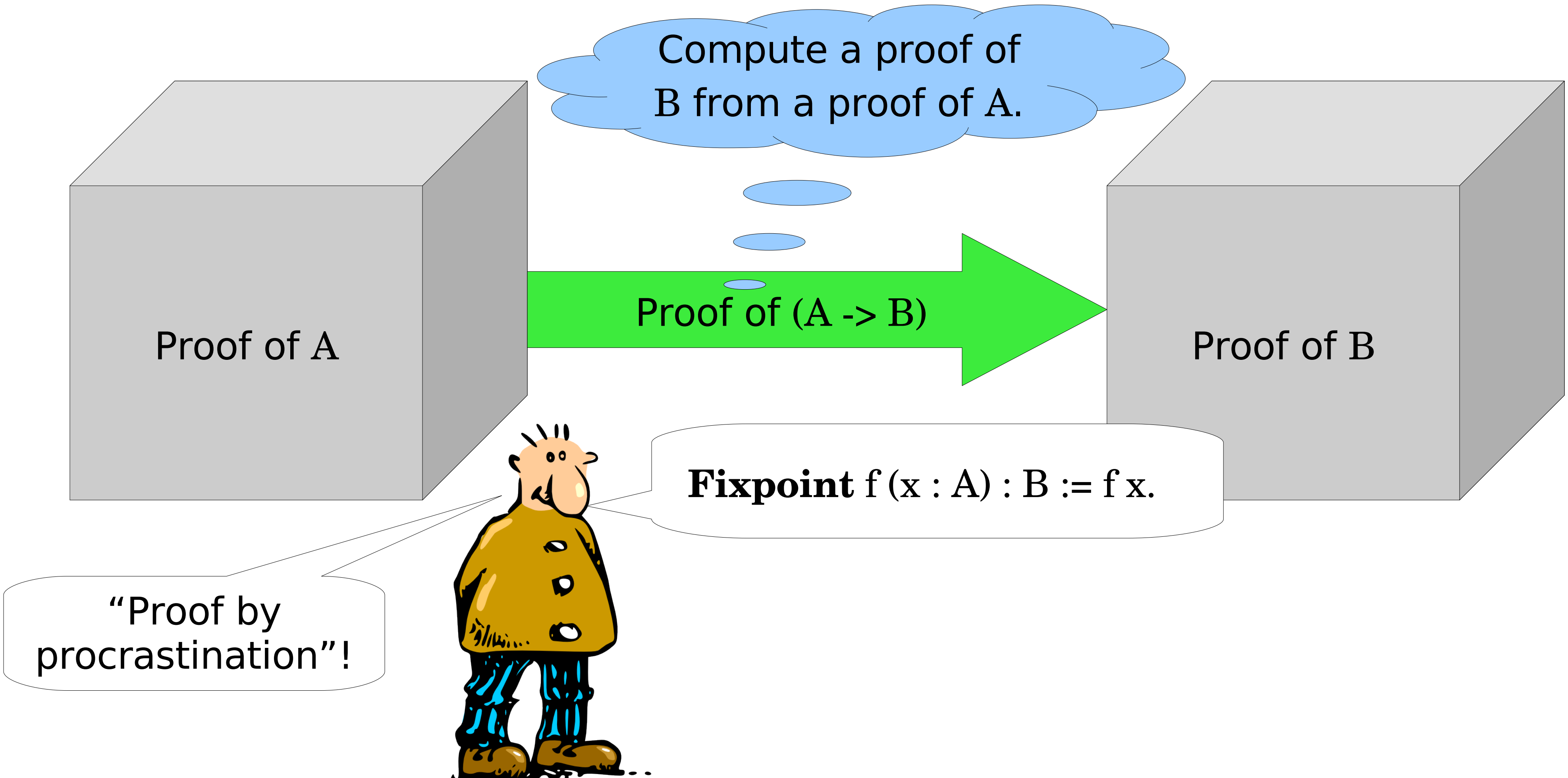
Interactive Computer Theorem Proving

Lecture 9: Beyond Primitive Recursion

CS294-9
October 19, 2006
Adam Chlipala
UC Berkeley

Recap: Termination Matters



If proof functions could run forever, **everything** would be "true"! So guaranteeing termination is critical to soundness....

Primitive Recursion

```
Fixpoint fib (n : nat) : nat :=
 match n with
  | 0 => 1
  | S n' =>
   match n' with
    | O => 1
    | S n'' => fib n'' + fib n'
   end
 end.
```

Every recursive call must have an argument that has a *syntactic* path from the original.

General Recursion

```
let rec nat_to_int = function
    O -> 0
  | S O -> 1
  | S (S n) ->
    let i = nat_to_int (n / 2) in
    if isEven n then
      2 * i
    else
      1 + 2 * i
```

Recursion Principle:
When defining f(n), you may use f(n') for all n' < n.

Alternative Principle:
When defining f(n) with n > 1, you may use f(n / 2).

```
let rec mergeSort = function
    [] -> []
  | [x] -> [x]
  | ls ->
    let (ls1, ls2) = split ls in
    merge (mergeSort ls1) (mergeSort ls2)
```

Recursion Principle:
When defining f(L), you may use f(L') for all L' with length(L') < length(L).

```
let rec looper = function
    true -> ()
  | false -> looper false
```

This really **is** non-terminating, but we want to reason about the terminating cases!

Outline of Techniques

- Relations instead of functions
- Bounded recursion
- Recursion on ad-hoc predicates
- Well-founded recursion
- Constructive domain theory

Using Relations

Inductive plusR : nat -> nat -> nat -> **Set** :=

| plusR_O : **forall** *n*,

plusR O *n* *n*

| plusR_Sn : **forall** *n* *m* *sum*,

plusR *n* *m* *sum*

-> plusR (S *n*) *m* (S *sum*).

Extraction plusR.

```
type plusR =
  | PlusR_O of nat
  | PlusR_Sn of nat * nat * nat * plusR
```

Bounded Recursion

```
Fixpoint nat_to_int (bound : nat) (n : nat) {struct bound} : int :=
  match bound with
   | O => 0
   | S bound' =>
     match n with
      | O -> 0
      | S O -> 1
      | S (S n') ->
        let i := nat_to_int bound' (n / 2) in
        if isEven n then
          2 * i
        else
          1 + 2 * i
     end
  end.
```

Pros

- We can prove that nat_to_int (S n) n satisfies the spec, for any n.

Cons

- ...but nat_to_int gives the wrong answer if we pass it too low a bound!
  - Alternatively, we could have it return an error code, but that isn't much better.
- Threading a nat around is a pain.
- The extraction of this function retains the extra argument, though we'd probably rather it didn't.

The Big Problem: Compositional Reasoning

```
Variable f : nat -> A -> option B.
Variable g : nat -> C -> option D.
Variable h : B -> D -> E.

Definition foo (n : nat) (x : A) (y : C) :=
  match f n x, g n y with
    | Some r1, Some r2 => Some (h r1 r2)
    | _, _ => None
```

Proposal: For any *F* : nat -> *T1* -> option *T2*, say that "*F*(x) = y" if there exists *n* such that *F n* x = Some y.

If we know f(u) = v and g(w) = x,
we want to conclude foo(u)(w) = h(v)(x).

This requires **looking inside the definitions** of f and g!

Recursion on an Ad-Hoc Predicate

```
Fixpoint nat_to_int (n : nat) : int :=
 match n with
  | O -> 0
  | S O -> 1
  | S (S n′) ->
   let i := nat_to_int (n / 2) in
   if isEven n then
    2 * i
   else
    1 + 2 * i
 end.
```

This may not be primitive recursive, but the recursive structure is still very predictable and "obviously" well-founded!

```
Inductive P : nat -> Set :=
 | P_0 : P 0
 | P_1 : P 1
 | P_div2 : forall n, P (n / 2) -> P n.
```

Key Property: There exists a P n for any n!

Recursion on an Ad-Hoc Predicate

```
Fixpoint nat_to_int (n : nat) (p : P n) {struct p} : int :=
 match n with
  | O -> 0
  | S O -> 1
  | S (S n') ->
   match p with
    | P_div2 _ p' =>
     let i := nat_to_int (n / 2) p' in
     if isEven n then
      2 * i
     else
      1 + 2 * i
    | _ => (* show a contradiction *)
   end
end
```

```
Inductive P : nat -> Set :=
  | P_0 : P 0
  | P_1 : P 1
  | P_div2 : forall n, P (n / 2) -> P n.
```

<u>Pros</u>

- nat_to_int always returns a correct answer!

<u>Cons</u>

- To call nat_to_int, we have to come up with a P n value through some ad-hoc mechanism.
- The P n values survive extraction and add even more runtime complexity than the nats from bounded recursion.

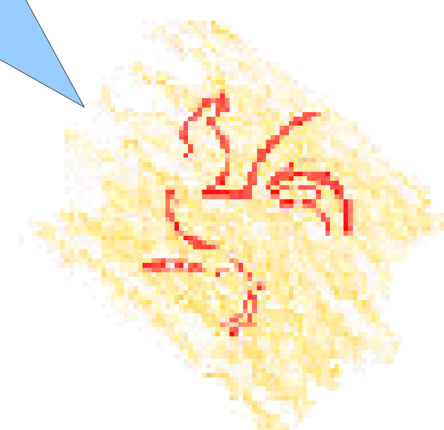
This one turns out to be easy to solve! Just put P in **Prop**.

Recursion on an Ad-Hoc Predicate

```
Fixpoint nat_to_int (n : nat) (p : P n) {struct p} : int :=
 match n with
  | O -> 0
  | S O -> 1
  | S (S n') ->
    match p with
     | P_div2 _ p' =>
      let i := nat_to_int (n / 2) p' in
      if isEven n then
        2 * i
      else
        1 + 2 * i
     | _ => (* show a contradiction *)
    end
end.
```

```
Inductive P : nat -> Prop :=
  | P_0 : P 0
  | P_1 : P 1
  | P_div2 : forall n, P (n / 2) -> P n.
```

You can't eliminate a **Prop** to form a **Set**!



Recursion on an Ad-Hoc Predicate

```
Fixpoint nat_to_int (n : nat) (p : P n) {struct p} : int :=
 match n with
  | O -> 0
  | S O -> 1
  | S (S n') ->
   let i := nat_to_int (n / 2) (match p with
      | P_div2 _ p' => p'
      | _ => (* show a contradiction *)
    end) in
   if isEven n then
    2 * i
   else
    1 + 2 * i
end.
```

```
Inductive P : nat -> Prop :=
  | P_0 : P 0
  | P_1 : P 1
  | P_div2 : forall n, P (n / 2) -> P n.
```

<u>Pros</u>

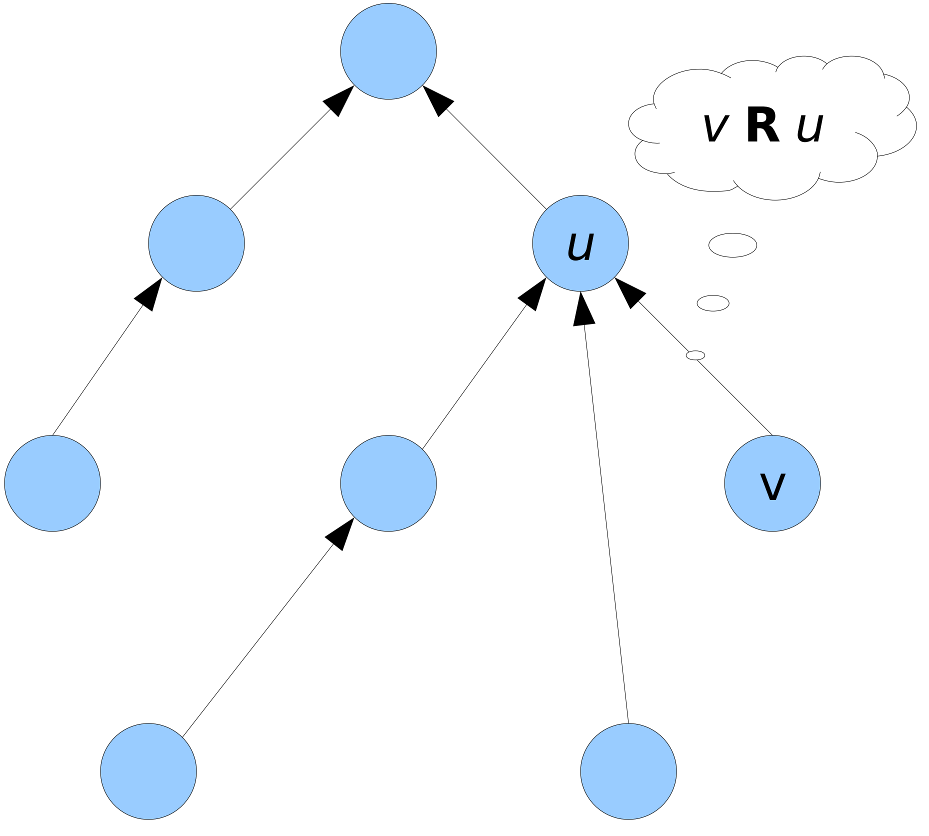
- Finally nat_to_int extracts to exactly the OCaml program we want, since any P n values are erased.

<u>Cons</u>

- Manipulating these *witnesses* is still a bookkeeping hassle.

Well-Founded Recursion

A **well-founded relation** on set **X** is a binary relation **R** on such that there are **no infinite descending chains**.
That is, there exists no sequence *x1* **R** *x2* **R** *x3* **R** *x4* **R** ....



Accessibility graph:
Connect *x* to *y* if *x* **R** *y*.

Say *x* is **accessible** if it has no outgoing edges or all of its successors are accessible.

Alternate definition:
R is well-founded iff every element of **X** is accessible.

Back to Our Exa

Not quite legal Coq syntax in 8.0, but something similar added in 8.1 beta.

```
Fixpoint nat_to_int (n : nat) {well_founded R} : int :=
 match n with
  | O -> 0
  | S O -> 1
  | S (S n') ->
   let i := nat_to_int pf (n / 2) in
   if isEven n then
    2 * i
   else
    1 + 2 * i
 end.
```

$\mathrm{R}\ n\ (n / 2)$

$\mathrm{R}\ x\ y = x > y$
or even
$\mathrm{R}\ x\ y = y = x / 2$

Prove your relation is well-founded by showing that every nat is accessible for it.

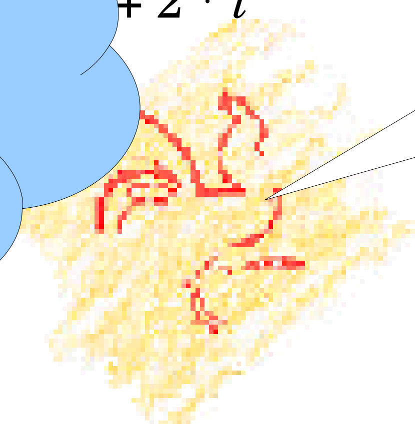
One catch....

We have to show that our function is **extensional**.

```
Definition f (self : nat -> int) (n : nat) : int :=
 match n with
  | O -> 0
  | S O -> 1
  | S (S n') ->
   let i := self pf (n / 2) in
   if isEven n then
     2 * i
   else
     + 2 * i
```

For any self1 and self2 that **return equal values on equal inputs**, f behaves the same.

Universal extensionality can be expressed as an **axiom**, and the result is a *new* sound formal system....



Waaait a minute. Coq doesn't allow you to “look inside of functions,” so every function must be extensional!

That may be true, but the logic isn't strong enough to prove it!

Real General Recursion

```
let rec looper = function
    true -> ()
  | false -> looper false
```

A Turing-complete programming language **must** allow general recursion, which implies **allowing non-termination**.

How can we **“add Turing completeness”** to Coq in a way that:

- Preserves logical soundness?
- Allows us to reason about programs?
- Allows extraction of executable programs?

My answer: A principled version of bounded recursion
...inspired by **domain theory**

Solving The Big Problem

Whenever f n x = Some y, for any $n' > n$, f n' x = Some y.

Variable f : nat -> A -> option B.

Variable g : nat -> C -> option D.

Variable h : B -> D -> E.

What very general condition can we impose on f and g to avoid this problem?

Definition foo (n : nat) (x : A) (y : C) :=

  match f n x, g n y **with**

    | Some r1, Some r2 => Some (h r1 r2)

    | _, _ => None

Proposal: For any F : nat -> $T1$ -> option $T2$, say that "F(x) = y" if there exists n such that F n x = Some y.

This requires **looking inside the definitions** of f and g!

If we know f(u) = v and g(w) = x,
we want to conclude foo(u)(w) = h(v)(x).

Solving the Little Problem

Threading bounds throughout a program is a pain. We want to build up a library of combinators that let us program naturally.

Return e

Theorem:

Return e $\Rightarrow$ e

Implementation:

λn. Some e

x <- e1; e2

Theorem:

If e1 => v1,

And e2[x := v1] => v2,

Then x <- e1; e2 => v2

Implementation:

λn. **match** e1 n **with**

| None => None

| Some v => (e2 v) n

end.

For f : (A -> B) -> (A -> B):

Fix f

Theorem:

If f (**Fix** f) x => v,

Then Fix f x => v

Implementation:

$\lambda n. \lambda x.$ $f^n\ n\ x$

where $f^0 = \lambda x. \lambda n.$ None

and $f^{n+1} = f\ (f^n)$