Interactive Computer Theorem Proving

Lecture 9: Beyond Primitive Recursion
Recap: Termination Matters

Compute a proof of $B$ from a proof of $A$.

Proof of $(A \rightarrow B)$

Proof of $B$

Proof of $A$

"Proof by procrastination"!

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


**Primitive Recursion**

**Fixpoint** fib \((n : \text{nat}) : \text{nat} :=\) 

\[
\text{match } n \text{ with } \\
| 0 => 1 \\
| S \ n' => \text{match } n' \text{ with } \\
| O => 1 \\
| S \ n'' => \text{fib } n'' + \text{fib } n' \\
\text{end} \\
\text{end.}
\]

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

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

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

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

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).

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

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

\textbf{Fixpoint} \texttt{nat\_to\_int} (\texttt{bound} : \texttt{nat}) (\texttt{n} : \texttt{nat}) \{\texttt{struct} \texttt{bound}\} : \texttt{int} :=

\begin{verbatim}
match \texttt{bound} with
  | \texttt{O} => 0
  | \texttt{S \texttt{bound'}} =>
    \begin{verbatim}
    match \texttt{n} with
    | \texttt{O} => 0
    | \texttt{S \texttt{O}} => 1
    | \texttt{S \texttt{(S \texttt{n')}}} =>
      \begin{verbatim}
      let \texttt{i} := \texttt{nat\_to\_int} \texttt{bound'} (\texttt{n} / 2) in
      \begin{verbatim}
      if \texttt{isEven} \texttt{n} then
        2 \* \texttt{i}
      \texttt{else}
        1 + 2 \* \texttt{i}
      \end{verbatim}
      \end{verbatim}
    end
end.
\end{verbatim}
\end{verbatim}

\textbf{Pros}

- We can prove that \texttt{nat\_to\_int} (\texttt{S \texttt{n}}) \texttt{n} satisfies the spec, for any \texttt{n}.

\textbf{Cons}

- ...but \texttt{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 \texttt{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** \( f : \text{nat} \to A \to \text{option} B. \)

**Variable** \( g : \text{nat} \to C \to \text{option} D. \)

**Variable** \( h : B \to D \to E. \)

**Definition** \( \text{foo} \ (n : \text{nat}) \ (x : A) \ (y : C) \ := \)

\[
\text{match } f \ n \ x, \ g \ n \ y \ \text{with} \\
\quad | \text{Some } r1, \text{Some } r2 \Rightarrow \text{Some } (h \ r1 \ r2) \\
\quad | _, \ _ \Rightarrow \text{None}
\]

**Proposition:** For any \( F : \text{nat} \to T1 \to \text{option} T2, \) say that “\( F(x) = y \)” if there exists \( n \) such that \( F \ n \ x = \text{Some } y. \)

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

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.

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

Pros
• nat_to_int always returns a correct answer!

Cons
• 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 \( n : \text{nat} \) \((p : P n)\) \{struct p\} : int :=

```ocaml
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** \( P : \text{nat} \rightarrow \text{Prop} \) :=

```ocaml
| P_0 : P 0  
| P_1 : P 1  
| P_div2 : forall n, P (n / 2) -> P n.
```

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

**Cons**
- Manipulating these *witnesses* is still a bookkeeping hassle.
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 $x_1 \ R \ x_2 \ R \ x_3 \ R \ x_4 \ R \ ...$.

**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 Example

**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 (n / 2) in
  if isEven n then
    2 * i
  else
    1 + 2 * i
end.

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

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

R `x y = x > y` or even
R `x y = y = x / 2`
One catch....

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

**Definition**  
\[
f (\text{self} : \text{nat} \rightarrow \text{int}) (n : \text{nat}) : \text{int} :=
\]
\[
\begin{cases}
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
\text{let } i := \text{self pf } (n / 2) \text{ in} \\
\quad \text{if } \text{isEven } n \text{ then} \\
\quad \quad 2 * i \\
\quad \quad \text{else} \\
\quad \quad 1 + 2 * i
\end{cases}
\]

For any \text{self1} and \text{self2} that **return equal values on equal inputs**, \(f\) behaves the same.

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!

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

```plaintext
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

**Variable** \( f : \text{nat} \rightarrow A \rightarrow \text{option B} \).
**Variable** \( g : \text{nat} \rightarrow C \rightarrow \text{option D} \).
**Variable** \( h : B \rightarrow D \rightarrow E \).

**Definition** \( \text{foo} (n : \text{nat}) (x : A) (y : C) := \)

\[
\text{match } f \ n \ x, \ g \ n \ y \text{ with } \\
| \text{Some } r1, \text{ Some } r2 => \text{Some } (h \ r1 \ r2) \\
| \_ , \_ => \text{None}
\]

**Proposal:** For any \( F : \text{nat} \rightarrow T1 \rightarrow \text{option T2} \), say that “\( F(x) = y \)” if there exists \( n \) such that \( F \ n \ x = \text{Some } y \).

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

Whenever \( f \ n \ x = \text{Some } y \), for any \( n' > n \), \( f \ n' \ x = \text{Some } y \).

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

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

x <- e1; e2

Theorem:

Return e ⇒ e

Theorem:

If e1 ⇒ v1,

And e2[x := v1] ⇒ v2,

Then x <- e1; e2 ⇒ v2

Implementation:

Some e

Implementation:

\[ \lambda n. \text{match } e1 n \text{ with } \]

| 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 \( f^0 = \lambda x. \lambda n. \text{None} \)

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