Interactive Computer Theorem Proving

Lecture 3: Data structures and Induction

CS294-9
September 7, 2006
Adam Chlipala
UC Berkeley
The Peano Axioms

0 ∈ \mathbb{N}

∀ n ∈ \mathbb{N}, S(n) ∈ \mathbb{N}

∀ n ∈ \mathbb{N}, S(n) ≠ 0

∀ a, b ∈ \mathbb{N}, a = b ↔ S(a) = S(b)

For any property P:

P(0) \land (\forall n ∈ \mathbb{N}, P(n) \rightarrow P(S(n)))) \rightarrow \forall n ∈ \mathbb{N}, P(n)

We can define \mathbb{N} (up to isomorphism) as the least set satisfying these properties.
The Set Theory Approach

“Now that we have natural numbers, let's use them to define some data structures....”

\[
\text{natlist}(0) = \{\emptyset\}
\]

\[
\text{natlist}(S(n)) = \{\emptyset\} \cup \mathcal{N} \times \text{natlist}(n)
\]

\[
\text{natlist} = \bigcup_{n \in \mathcal{N}} \text{natlist}(n)
\]

\[
\text{nil} = \emptyset
\]

\[
\text{cons}(n, ls) = \langle n, ls \rangle
\]

**Derived induction principle:** For any property \( P \):

\[
P(\text{nil}) \land (\forall n \in \mathcal{N}, \forall ls \in \text{natlist}, P(ls) \rightarrow P(\text{cons}(n, ls)))
\]

\[
\rightarrow \forall ls \in \text{natlist}, P(ls)
\]
Why This Isn't Such a Great Idea

- These definitions are pretty awkward!
  - Set theorists usually don't write all their proofs formally, so they can get away with it.

- Proofs at this level of detail must be very large.
  - Mathematicians aren't used to optimizing for space!

- What about more complicated data structures?
Type Theory's Great Idea

**Functions** and **data structures** should be the fundamental building blocks of math, not sets!

**Coq**
- Function types
- **Inductive types**
- Constructors
- Case analysis
- **Recursive functions**

**ZF Set Theory**
- Negation
- Conjunction
- Universal quantifier
- Equality
- Natural deduction proof rules
- Empty set
- Set equality
- Set pairing
- Set union
- Natural numbers
- Mathematical induction
...
Back to the Beginning...

**Inductive** \( \text{nat} : \text{Set} := \)

\( | \text{O} : \text{nat} \)
\( | \text{S} : \text{nat} \rightarrow \text{nat}. \)

**What we get:**

- A type \( \text{nat} \)
- Two **constructors** \( \text{O} \) and \( \text{S} \) for building \( \text{nats} \)
- **Case analysis** (pattern matching) on \( \text{nats} \)
- The ability to write **recursive functions** over \( \text{nats} \)
Verifying the Peano Axioms

There exists set \( \mathcal{N} \)...

\[
0 \in \mathcal{N}
\]

\[
\forall n \in \mathcal{N}, \ S(n) \in \mathcal{N}
\]

**Check nat.**
- \( \text{nat} : \text{Set} \).

**Check O.**
- \( O : \text{nat} \).

**Check S.**
- \( S : \text{nat} \rightarrow \text{nat} \).
Pattern Matching

General form for nat:

```plaintext
match n with
  | O => e1
  | S n' => e2(n')
end
```

And with anonymous function notation (like Scheme `lambda` and OCaml `fun`):

```plaintext
fun n => match n with
  | O => O
  | S n' => n'
end
```

Examples

```plaintext
match O with
  | O => O
  | S n' => n'
end
Evaluates to: O
```

```plaintext
match S (S O) with
  | O => O
  | S n' => n'
end
Evaluates to: S O
```
Peano Axiom #3

\[ \forall n \in \mathbb{N}, S(n) \neq 0 \]

\[
\text{fun } n \Rightarrow \text{match } n \text{ with }
\]

Define \( f \) as:

- \( O \Rightarrow \text{True} \)
- \( S\ n' \Rightarrow \text{False} \)

• **Proof.** Let \( n \) be given.

• Assume for a contradiction that \( S\ n = 0 \).

• Assert True.

• By **computation**, we have the equivalent \( f\ 0 \).

• By the assumption, \( f\ (S\ n) \).

• **Contradiction!**
Peano Axiom #4
\[ \forall a, b \in \mathbb{N}, S(a) = S(b) \rightarrow a = b \]

fun \( n \rightarrow \) match \( n \) with

Define \( p \) as:
\[
\begin{align*}
\mid O & \Rightarrow O \\
\mid S \ n' & \Rightarrow n'
\end{align*}
\]

end

• **Proof.** Let \( a \) and \( b \) be given.

• Assume \( S \ a = S \ b \).

• By reflexivity, \( p \ (S \ b) = p \ (S \ b) \).

• By the assumption, \( p \ (S \ a) = p \ (S \ b) \).

• **By computation**, \( a = b \).
Peano Axiom #5

\[ P(0) \land (\forall n \in \mathbb{N}, P(n) \rightarrow P(S(n))) \rightarrow \forall n \in \mathbb{N}, P(n) \]

We could prove this manually using recursive functions, but...

Check nat_ind.

\[
\text{nat}_\text{ind} : \text{forall } P : \text{nat} \rightarrow \text{Prop}, \n\]

\[
P \ 0 \n\]

\[
\rightarrow \ (\text{forall } n : \text{nat}, P \ n \rightarrow P \ (S \ n)) \n\]

\[
\rightarrow \text{forall } n : \text{nat}, P \ n \n\]
Recursive Functions

Analogue of the standard named function definition syntax in most programming languages:

Two arguments of type \( \text{nat} \)

Return type \( \text{nat} \)

Fixpoint \( \text{add} \ (n \ m : \text{nat}) \ \{\text{struct} \ n\} : \text{nat} := \)

\[
\text{match } n \ \text{with} \\
| O \Rightarrow m \\
| S \ n' \Rightarrow S \ (\text{add} \ n' \ m) \\
\text{end.}
\]

No recursive calls allowed in this \text{match} branch

Recursion over argument \( n \)

Only \textbf{recursive calls with first argument equal to} \( n' \) allowed in this branch
Aside: Why So Fussy About Termination?

Imagine that Coq allowed this definition:

```coq
Fixpoint f (n : nat) {struct n} : nat :=
    S (f n).
```

- We would then have \( f(n) = S(f(n)) \), for all \( n \).
- But we can also prove \( m \neq S(m) \), for all \( m \).
- So \( f(0) = S(f(0)) \) and \( f(0) \neq S(f(0)) \).
- **Contradiction!** Our logic is unsound!
More Datatypes: Booleans

**Inductive** bool : Set :=

| false : bool
| true : bool.

**Check** bool_ind.

bool_ind : **forall** P : bool -> Prop,

P false

-> P true

-> **forall** b : bool, P b
More Datatypes: Lists

**Inductive** natlist : Set :=

| nil : natlist
| cons : nat -> natlist -> natlist.

**Check** natlist_ind.

natlist_ind : forall P : natlist -> Prop,

P nil

-> (forall (n : nat) (ls : natlist),

P ls -> P (cons n ls))

-> forall ls : natlist, P ls
More Datatypes: Trees

Inductive nattree : Set :=
  | Leaf : nattree
  | Node : nattree -> nat -> nattree -> nattree.

Check nattree_ind.

nattree_ind : forall P : nattree -> Prop,
  P Leaf
  -> (forall (t1 : nattree) (n : nat)
    (t2 : nattree),
    P t1 -> P t2 -> P (Node t1 n t2))
  -> forall t : nattree, P t

Check nattree_ind.
Simple Inductive Types in General

$$\text{Inductive } \text{tname} : \text{Set} :=$$

- $$c_1 : t_{1,1} \rightarrow \ldots \rightarrow t_{1,k_1} \rightarrow \text{tname}$$
- $$\ldots$$
- $$\ldots$$
- $$c_n : t_{n,1} \rightarrow \ldots \rightarrow t_{n,k_n} \rightarrow \text{tname}.$$
Using an Inductive Type

Pattern matching

match e with
| c₁ x₁ ... xₖ₁ => e₁(x₁, ..., xₖ₁)
| ...
| cₙ x₁ ... xₖₙ => eₙ(x₁, ..., xₖₙ)
end

Inductive tname : Set :=
| c₁ : t₁₁ -> ... -> t₁ₖ₁ -> tname
| ...
| cₙ : tₙ₁ -> ... -> tₙₖₙ -> tname.

Must use a match somewhere to obtain a strict subterm of x to use in a recursive call.

Recursive functions

Fixpoint f (x : tname) : T := e(x).
Fixpoint f (x₁ : T₁) ... (xₖ : tname) ... (xₙ : Tₙ)
{struct xₖ} : T := e(x₁, ..., xₙ).
(fix f (x : tname) : T := e(x))
Using an Inductive Type II

**Inductive tname : Set :=**

\[ | c_1 : t_{1,1} \rightarrow ... \rightarrow t_{1,k_1} \rightarrow tname \\
| ... \]

Induction principle

“For every predicate \( P \) over \( tname \) s,

**If** for every constructor \( c_i \) of \( tname \):

For every set \( e_{i,j} \) of arguments to \( c_i \),

Assuming \( P e_{i,j} \) for every \( e_{i,j} \) of type \( tname \),

We can prove \( P (c_i e_{i,1} \ldots e_{i,k_i}) \)

Then

For every value \( e \) of type \( tname \),

We can prove \( P e. \)”
So what's the deal with this “by computation” stuff, anyway?

Coq considers to be interchangeable any two expressions that evaluate to a common result.

Atomic evaluation step: Applying a function

\[(\text{fun } x \Rightarrow S 
\ append x) \ (S \ O) \Rightarrow S \ (S \ O)\]

\[(\text{fix } f \ (x : \text{nat}) : \text{nat} \Rightarrow S \ x) \ (S \ O) \Rightarrow S \ (S \ O)\]

Atomic evaluation step: Simplifying a case analysis

\[(\text{match } S \ x \ \text{with } O \Rightarrow O \ | \ S \ n \Rightarrow n \ \text{end}) \Rightarrow x\]

Atomic evaluation step: Expanding a definition

\[f \ O \Rightarrow (\text{fun } x \Rightarrow S \ (S \ x)) \ O\]

\[\text{Definition } f := \text{fun } x \Rightarrow S \ (S \ x)\]
Reduction Order

Reductions can happen *anywhere in an expression*, so:

\[
(fun\ x \Rightarrow (fun\ y \Rightarrow S\ y)\ x) \Rightarrow (fun\ x \Rightarrow S\ x)
\]

\[
(match\ x\ with\ O \Rightarrow O \mid S\ n \Rightarrow (fun\ y \Rightarrow S\ y)\ n\ end) \\
\Rightarrow (match\ x\ with\ O \Rightarrow O \mid S\ n \Rightarrow S\ n\ end)
\]

*Important meta-theorem about Coq:* For any expression, *any order of reductions leads to the same result.*
Why Should I Care?

All of these theorems can be proved by reflexivity:

- $1 + 1 = 2$
- $0 + x = x$
- $\text{length} (\text{cons} \ 0 \ (\text{cons} \ 1 \ \text{nil})) = 2$
- $\text{append} \ (\text{cons} \ 0 \ \text{nil}) \ (\text{cons} \ 1 \ \text{nil}) = \text{cons} \ 0 \ (\text{cons} \ 1 \ \text{nil})$
- $\text{append} \ \text{nil} \ ls = ls$
- compiler $myProgram = outputAssemblyCode$

Proving theorems about programs and math in general is much more pleasant when these things come for free.
Conclusion

- Sample HW1 solution is on the web site.
- HW2 is posted
  - Fun with data structures and induction
- Next lecture: Using inductive types to define new logical predicates and the rules that can be used to prove them