Interactive Computer Theorem Proving

Lecture 3: Data structures and Induction

CS294-9
September 7, 2006
Adam Chlipala
UC Berkeley

The Peano Axioms

$0 \in \mathcal{N}$

$\forall\, n \in \mathcal{N},\ \mathrm{S}(n) \in \mathcal{N}$

$\forall\, n \in \mathcal{N},\ \mathrm{S}(n) \neq 0$

$\forall\, a, b \in \mathcal{N},\ a = b \leftrightarrow \mathrm{S}(a) = \mathrm{S}(b)$

For any property P:

$$P(0) \wedge (\forall\, n \in \mathcal{N},\ \mathrm{P}(n) \rightarrow \mathrm{P}(\mathrm{S}(n))) \rightarrow \forall\, n \in \mathcal{N},\ \mathrm{P}(n)$$

We can define $\mathcal{N}$ (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:

$$P(\text{nil}) \wedge (\forall\, n \in \mathcal{N}, \forall\, ls \in \text{natlist}, P(ls) \rightarrow P(\text{cons}(n, ls)))$$
$$\rightarrow \forall\, ls \in \text{natlist}, P(ls)$$

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…

$$\textbf{Inductive}\ \mathrm{nat} : \textbf{Set} :=$$

$$|\ \mathrm{O} : \mathrm{nat}$$

$$|\ \mathrm{S} : \mathrm{nat} \to \mathrm{nat}.$$

What we get:

- A type nat
- Two **constructors** O and S for building nats
- **Case analysis** (pattern matching) on nats
- The ability to write **recursive functions** over nats

Verifying the Peano Axioms

There exists set $\mathcal{N}$...

Check nat.

nat : **Set**.

0 $\in \mathcal{N}$

Check O.

O : nat.

$\forall$ *n* $\in \mathcal{N}$, S(*n*) $\in \mathcal{N}$

Check S.

S : nat $\rightarrow$ nat.

Pattern Matching

General form for nat:

$\mathbf{match}\ \mathrm{n}\ \mathbf{with}$

$\mid \mathrm{O} => \mathrm{e1}$

$\mid \mathrm{S}\ n' => \mathrm{e2}(n')$

$\mathbf{end}$

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

$\mathbf{fun}\ n => \mathbf{match}\ n\ \mathbf{with}$

$\mid \mathrm{O} => \mathrm{O}$

$\mid \mathrm{S}\ n' => n'$

$\mathbf{end}$

Examples

$\mathbf{match}\ \mathrm{O}\ \mathbf{with}$

$\mid \mathrm{O} => \mathrm{O}$

$\mid \mathrm{S}\ n' => n'$

$\mathbf{end}$

Evaluates to: O

$\mathbf{match}\ \mathrm{S}\ (\mathrm{S}\ \mathrm{O})\ \mathbf{with}$

$\mid \mathrm{O} => \mathrm{O}$

$\mid \mathrm{S}\ n' => n'$

$\mathbf{end}$

Evaluates to: $\mathrm{S}\ \mathrm{O}$

Peano Axiom #3

$\forall\, n \in \mathcal{N}$, S($n$) ≠ 0

Define f as:

```
fun n => match n with
  | O => True
  | S n′ => False
end
```

- **Proof**. Let n be given.
- Assume for a contradiction that $\mathrm{S}\, n = 0$.
- Assert True.
- ***By computation***, we have the equivalent $f\, 0$.
- By the assumption, $f\, (\mathrm{S}\; \mathrm{n})$.
- ***By computation***, False.

Contradiction!

Peano Axiom #4

$\forall$ *a*, *b* $\in \mathcal{N}$, S(*a*) = S(*b*) $\rightarrow$ *a* = *b*

Define p as:

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

- **Proof**. Let a and b be given.
- Assume $\mathrm{S}\,a = \mathrm{S}\,b$.
- By reflexivity, $p\ (\mathrm{S}\,b) = p\ (\mathrm{S}\,b)$.
- By the assumption, $p\ (\mathrm{S}\,a) = p\ (\mathrm{S}\,b)$.
- ***By computation***, $a = b$.

Peano Axiom #5

$$P(0) \wedge (\forall\, n \in \mathcal{N}, \mathrm{P}(n) \rightarrow \mathrm{P}(\mathrm{S}(n))) \rightarrow \forall\, n \in \mathcal{N}, \mathrm{P}(n)$$

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

```
Check nat_ind.
  nat_ind : forall P : nat -> Prop,
    P O
    -> (forall n : nat, P n -> P (S n))
    -> forall n : nat, P n
```

Recursive Functions

Analogue of the standard named function definition syntax i

Two arguments of t

Return type nat

```
Fixpoint add (n m : nat) {struct n} : nat :=
  match n with
    | O => m
    | S n′ => S (add n′ m)
  end.
```

No recursive calls allowed in this **match** branch

e

recursion over

argument n

Only **recursive calls with first argument equal to n'** allowed in this branch

Aside: Why So Fussy About Termination?

Imagine that Coq allowed this definition:

$$\mathbf{Fixpoint}\, \mathrm{f}\, (n : \mathrm{nat})\, \{\mathbf{struct}\, n\} : \mathrm{nat} :=$$
$$\mathrm{S}\, (\mathrm{f}\, n).$$

- We would then have $\mathrm{f}\, n = \mathrm{S}\, (\mathrm{f}\, n)$, for all *n*.
- But we can also prove $m \neq \mathrm{S}\, m$, for all *m*.
- So $\mathrm{f}\, \mathrm{O} = \mathrm{S}\, (\mathrm{f}\, \mathrm{O})$ and $\mathrm{f}\, \mathrm{O} \neq \mathrm{S}\, (\mathrm{f}\, \mathrm{O})$.
- **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
```

Simple Inductive Types in General

Sort specification
(We'll see more possibilities later,
but for now we only consider **Set**.)

Inductive tname : **Set** :=

| $c_1 : t_{1,1}$ -> ... -> $t_{1,k1}$ -> tname

| ...

| $c_n : t_{n,1}$ -> ... -> $t_{n,kn}$ -> tname.

Zero or more named **constructors**

Arguments types are restricted so that they either **don't refer to tname** or **are exactly tname**.

Each constructor is given a **function type** from **zero or more arguments** to **the type being defined**.

Using an Inductive Type

Pattern matching

match e **with**

| $c_1\ x_1 \ldots x_{k1}$ => $e_1(x_1, \ldots, x_{k1})$

| …

| $c_n\ x_1 \ldots x_{kn}$ => $e_n(x_1, \ldots, x_{kn})$

end

Inductive tname : **Set** :=

| $c_1 : t_{1,1}$ -> … -> $t_{1,k1}$ -> tname

| …

| $c_n : t_{n,1}$ -> … -> $t_{n,kn}$ -> 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 (x_1 : T_1) … (x_k : tname) … (x_n : T_n)

{**struct** x_k} : T := e(x_1, …, x_n).

(**fix** f (x : tname) : T := e(x))

Using an Inductive Type II

Induction principles are derived by Coq as a convenience. They are implemented behind the scenes using **recursive functions**. (We'll see how later in the course.)

Inductive tname : **Set** :=

| c_1 : $t_{1,1}$ -> ... -> $t_{1,k1}$ -> tname

| ...

Induction principle

...and the **induction** tactic automatically figures out the right induction principle and how to apply it, so you usually don't have to think about the details of these things....

“For every predicate P over t

If for every constructor c_i

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 $e_{i,1}$ … $e_{i,ki}$)

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

(**fun** x => S x) (S O) $\Rightarrow$ S (S O)

(**fix** f (x : nat) : nat => S x) (S O) $\Rightarrow$ S (S O)

Atomic evaluation step: Simplifying a case analysis

(**match** S x **with** O => O | S n => n **end**) $\Rightarrow$ x

Atomic evaluation step: Expanding a definition

f O $\Rightarrow$ (**fun** x => S (S x)) O

Definition f :=
fun x => S (S x).

Reduction Order

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

$$(\mathbf{fun}\ x \texttt{ => } (\mathbf{fun}\ y \texttt{ => } \mathrm{S}\ y)\ x) \Rightarrow (\mathbf{fun}\ x \texttt{ => } \mathrm{S}\ x)$$

$$(\mathbf{match}\ x\ \mathbf{with}\ \mathrm{O} \texttt{ => } \mathrm{O} \mid \mathrm{S}\ n \texttt{ => } (\mathbf{fun}\ y \texttt{ => } \mathrm{S}\ y)\ n\ \mathbf{end})$$
$$\Rightarrow (\mathbf{match}\ x\ \mathbf{with}\ \mathrm{O} \texttt{ => } \mathrm{O} \mid \mathrm{S}\ n \texttt{ => } \mathrm{S}\ n\ \mathbf{end})$$

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**:

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

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