CS720

Logical Foundations of Computer Science

Lecture 3: data structures

Tiago Cogumbreiro

Recap

- We are currently learning the Logical Foundations (volume 1 of the SF book)
- We are learning a **programming language** that allows us formalize programming languages

> What do we mean by formalizing programming languages?

Recap

- We are currently learning the Logical Foundations (volume 1 of the SF book)
- We are learning a **programming language** that allows us formalize programming languages

> What do we mean by formalizing programming languages?

1. A way to describe the abstract syntax (do we know how to do this?)
2. A way to describe how language executes (do we know how to do this?)
3. A way to describe properties of the language (do we know how to do this?)

Homework submission reminder

The star system was confusing, so we no longer use it: **Complete all non-optional exercises.**

- For instance, if an exercise says `Exercise: 3 stars, optional`, then that exercise is *not* be graded.
- For instance, if an exercise says `Exercise: 3 stars`, then that exercise *is* graded.

A quick sure way to check if your homework is acceptable by the autograder is to run `coqc YourHomework.v` it should compile **without errors**

Today we will...

- Review how to define data structures and how to prove

Why are we learning this?

- Today we will be honing the tools you have learned so far.

Homework 2 (`Induction.v`, `Lists.v`) due:

Tuesday, September 18, 11:59 EST

By email: Tiago.Cogumbreiro@umb.edu

List.v

Due Tuesday, September 18, 11:59 EST

How do we define a data structure that holds two nats?

A pair of nats

```
Inductive natprod : Type :=
| pair : nat → nat → natprod.

Notation "( x , y )" := (pair x y).
```

> Explicit vs implicit: be cautious when declaring notations, they make your code harder to understand.

How do we read the contents of a pair?

Accessors of a pair

Accessors of a pair

```
Definition fst (p : natprod) : nat :=
```

Accessors of a pair

```
Definition fst (p : natprod) : nat :=
  match p with
  | pair x y => x
  end.

Definition snd (p : natprod) : nat :=
  match p with
  | (x, y) => y (* using notations in a pattern to be matched *)
   end.
```

How do we prove the correctness of our accessors?

(What do we expect fst/snd to do?)

Proving the correctness of our accessors:

```
Theorem surjective_pairing : forall (p : natprod),
  p = (fst p, snd p).
Proof.
  intros p.
```

```
1 subgoal
p : natprod
______________________________________(1/1)
p = (fst p, snd p)
```

> Does `simpl` work? Does `reflexivity` work? Does `destruct` work? What about `induction`?

How do we define a list of nats?

A list of nats

```
Inductive natlist : Type :=
  | nil : natlist
  | cons : nat → natlist → natlist.

(* You don't need to learn notations, just be aware of its existence:*)

Notation "x :: l" := (cons x l) (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).

Compute cons 1 (cons 2 (cons 3 nil)).
```

outputs:

```
= [1; 2; 3]
: list nat
```

How do we concatenate two lists?

Concatenating two lists

```
Fixpoint app (l1 l2 : natlist) : natlist :=
 match l1 with
 | nil => l2
 | h :: t => h :: (app t l2)
 end.

Notation "x ++ y" := (app x y) (right associativity, at level 60).
```

Proving results on list concatenation

```
Theorem nil_app_l : forall l:natlist,
  [] ++ l = l.
Proof.
  intros l.
```

> Can we prove this with `reflexivity`? Why?

Proving results on list concatenation

```
Theorem nil_app_l : forall l:natlist,
  [] ++ l = l.
Proof.
  intros l.
```

> Can we prove this with `reflexivity`? Why?

```
  reflexivity.
Qed.
```

Nil is a neutral element wrt app

```
Theorem nil_app_l : forall l:natlist,
  l ++ [] = l.
Proof.
  intros l.
```

> Can we prove this with `reflexivity`? Why?

Nil is a neutral element wrt app

```
Theorem nil_app_l : forall l:natlist,
  l ++ [] = l.
Proof.
  intros l.
```

> Can we prove this with `reflexivity`? Why?

```
In environment
l : natlist
Unable to unify "l" with "l ++ [ ]".
```

> How can we prove this result?

We need an induction principle of `natlist`

For some property `P` we want to prove.

- Show that $P([])$ holds.
- Given the induction hypothesis $P(l)$ and some number n, show that $P(n :: l)$ holds.

Conclude that $P(l)$ holds for all l.

> How do we know this principle? Hint: compare `natlist` with `nat`.

Comparing nats with natlists

```
Inductive natlist : Type :=
 | O : natlist                        | A: T
 | S : nat → nat.                     | B: T → T
```

1. $\vdash P(A)$
2. $t : T, P(t) \vdash P(B\ t)$

```
Inductive natlist : Type :=
 | nil : natlist                      | A: T
 | cons : nat → natlist → natlist.    | B: X → T → T
```

1. $\vdash P(A)$
2. $x : X, t : T, P(t) \vdash P(B\ t)$

How do we know the induction principle?

Use search

```
Search natlist.
```

which outputs

```
 nil: natlist
 cons: nat → natlist → natlist
(* trimmed output *)
natlist_ind:
   forall P : natlist → Prop,
   P [] →
   (forall (n : nat) (l : natlist), P l → P (n::l)) → forall n : natlist, P n
```

Nil is neutral on the right (1/2)

```
Theorem nil_app_r : forall l:natlist,
  l ++ [] = l.
Proof.
  intros l.
  induction l.
  - reflexivity.
  -
```

yields

```
1 subgoal
n : nat
l : natlist
IHl : l ++ [ ] = l
______________________________________(1/1)
(n :: l) ++ [ ] = n :: l
```

Nil is neutral on the right (2/2)

```
1 subgoal
n : nat
l : natlist
IHl : l ++ [ ] = l
______________________________________(1/1)
(n :: l) ++ [ ] = n :: l
```

Nil is neutral on the right (2/2)

```
1 subgoal
n : nat
l : natlist
IHl : l ++ [ ] = l
______________________________________(1/1)
(n :: l) ++ [ ] = n :: l
```

```
simpl.           (* app (n::l) [] = n :: (app l []) *)
rewrite -> IHl.  (*   n :: (app l []) = n :: l *)
                 (*        ^^^^^^^^^          ^ *)
reflexivity.     (* conclude *)
```

> Can we apply rewrite directly without simplifying?
> Hint: before and after stepping through a tactic show/hide notations.
> How do we state a theorem that leads to the same proof state (without ltac)?

How do we signal failure in a functional language?

Partial functions

How declare a function that is not defined for empty lists?

```
(* Pairs the head and the list *)
Fixpoint indexof n (l:natlist) :=
  match l with
  | [] ⇒ ???
  | h :: t ⇒
    match beq_nat h n with
    | true ⇒ 0
    | false ⇒ S (indexof t)
    end
  end.
```

Optional results

```
Inductive natoption : Type :=
  | Some : nat → natoption
  | None : natoption.
```

How do we declare indexof with optional types?

```
Fixpoint indexof n (l:natlist) : natoption :=
```

How do we declare indexof with optional types?

```
Fixpoint indexof n (l:natlist) : natoption :=
  match l with
  | [] ⇒ None
  | h :: t ⇒
    match beq_nat h n with
    | true ⇒ Some 0
    | false ⇒ S (indexof n t)
    end
  end.
```

How do we declare indexof with optional types?

```
Fixpoint indexof n (l:natlist) : natoption :=
  match l with
  | [] ⇒ None
  | h :: t ⇒
    match beq_nat h n with
    | true ⇒ Some 0
    | false ⇒ S (indexof n t)
    end
  end.
```

```
    | false ⇒ S (indexof n t)
                ^^^^^^^^^^^

The term "indexof n t" has type "natoption" while it is expected to have type "nat".
```

How do we declare indexof with optional types?

```
Fixpoint indexof (n:nat) (l:natlist) : natoption :=
  match l with
  | [] => None
  | h :: t =>
    match beq_nat h n with
    | true => Some 0            (* element found at the head *)
    | false =>
      match indexof n t with  (* check for error *)
      | Some i => Some (S i)  (* increment successful result *)
      | None => None          (* propagate error *)
      end
    end
  end.
```

Summary

Summary

- implemented containers: pair, list, option
- partial functions via option types
- reviewed case analysis, proof by induction
- used `Search` to browse definitions

Next class: read Poly.v

Ltac vocabulary

- `simpl`
- `reflexivity`
- `intros`
- `rewrite`
- `destruct`
- `induction`
- `assert`

(Nothing new from Lesson 2.)