CS720

Logical Foundations of Computer Science

Lecture 12: Formalizing an imperative language

Tiago Cogumbreiro

Automation tactics

- `X || Y`
- `repeat X`
- `constructor`
- `X ; Y`
- `all:X` and `n:X`
- `inversion H`
- user tactics
- `try X`
- `specialize`

Or

```
Goal 3 ≤ 6.
Proof.
  apply le_S.
  apply le_S.
  apply le_S.
  apply le_n.
Qed.
```

```
Goal 3 ≤ 6.
Proof.
  (* Try le_n, and then le_S *)
  apply le_n || apply le_S.
  apply le_n || apply le_S.
  apply le_n || apply le_S.
  apply le_n || apply le_S.
Qed.
```

Repeat

`repeat X` uses tactics `X` as many times until `X` fails, 0 or more times.

```
Goal 3 ≤ 6.
Proof.
  (* Try le_n, and then le_S *)
  apply le_n || apply le_S.
  apply le_n || apply le_S.
  apply le_n || apply le_S.
  apply le_n || apply le_S.
Qed.
```

```
Goal 3 ≤ 6.
Proof.
  (* Try one constructor or try the other *
  repeat (apply le_n || apply le_S).
Qed.
```

Constructor

Applies the first constructor available (according to the order defined).

```
Goal 3 ≤ 6.
Proof.
  apply le_S.
  apply le_S.
  apply le_S.
  apply le_n.
Qed.
```

```
Goal 3 ≤ 6.
Proof.
  constructor.
  constructor.
  constructor.
  constructor.
Qed.
```

```
Goal 3 ≤ 6.
Proof.
  repeat constructor.
Qed.
```

Semi-colon ;

Semi-colon to perform a tactic in all branches

```
Lemma aeval_iff_aevalR : forall st a n,
  aevalR st a n <-> aeval st a = n.
Proof.
  split; intros. {
    induction a.
    -
```

```
1 goal
st : state
n0, n : nat
H : aevalR st (ANum n0) n
______________________________________(1/1)
aeval st (ANum n0) = n
```

Semi-colon `;`

Semi-colon to perform a tactic in all branches

```
Lemma aeval_iff_aevalR : forall st a n,
  aevalR st a n <-> aeval st a = n.
Proof.
  split; intros. {
    induction a; simpl.
    -
```

```
1 goal
st : state
n0, n : nat
H : aevalR st (ANum n0) n
______________________________________(1/1)
n0 = n
```

All `all:`

The `all: X` runs `X` in all proof states.

```
Lemma aeval_iff_aevalR : forall st a n,
  aevalR st a n <-> aeval st a = n.
Proof.
  split; intros. {
    induction a.
    all: simpl.
    -
```

```
1 goal
st : state
n0, n : nat
H : aevalR st (ANum n0) n
______________________________________(1/1)
n0 = n
```

All versus semi-colon

- You cannot step through `;` you can step trough `all:`
- `all:` is more verbose, `foo. all: bar.` versus `foo; bar`.
- `X:Y` is more general; for instance, `2: { ... }` allows you to prove the next goal first.
- In some cases you must use `;` and cannot use `all:` (eg, user-defined tactics, discussed next)

Inversion

Inversion gives you the "contents" of an assumption, you can dispose of it after (try doing `destroy H`).

```
Lemma aeval_iff_aevalR : forall st a n,
  aevalR st a n <-> aeval st a = n.
Proof.
  split; intros. {
    induction a.
    all: simpl.
    - inversion H; subst; clear H.
```

User-defined tactics

In ltac you cannot use multiple periods, so you must use a single

```
Ltac invc X := inversion X; subst; clear X.

Lemma aeval_iff_aevalR : forall st a n,
  aevalR st a n <-> aeval st a = n.
Proof.
  split; intros. {
    induction a.
    all: simpl.
    - invc H.
```

Try

With `try X` perform `X` and succeed. Great with `;` and `all:`.

```
Lemma aeval_iff_aevalR : forall st a n,
  aevalR st a n <-> aeval st a = n.
Proof.
  split; intros. {
    induction a.
    all: simpl.
    all: try invc H.
    all: try reflexivity.
```

Identifying a bad induction principle

```
a1, a2 : aexp
n1, n2 : nat
IHa2 : aevalR st a2 (n1 + n2) ->
       aeval st a2 = n1 + n2
IHa1 : aevalR st a1 (n1 + n2) ->
       aeval st a1 = n1 + n2
H2 : aevalR st a1 n1
H4 : aevalR st a2 n2
______________________________________(1/1)
aeval st a1 + aeval st a2 = n1 + n2
```

Identifying a bad induction principle

```
Lemma aeval_iff_aevalR : forall st a n,
  aevalR st a n <-> aeval st a = n.
Proof.
  split; intros. {
    generalize dependent st.
    generalize dependent n.
    induction a; intros; simpl.
    all: try invc H.
    all: try reflexivity.
    -
```

```
IHa1 : forall (n : nat) (st : state),
       aevalR st a1 n -> aeval st a1 = n
IHa2 : forall (n : nat) (st : state),
       aevalR st a2 n -> aeval st a2 = n
H2 : aevalR st a1 n1
H4 : aevalR st a2 n2
______________________________________(1/1)
aeval st a1 + aeval st a2 = n1 + n2
```

Specialize assumptions

```
Lemma aeval_iff_aevalR : forall st a n,
  aevalR st a n <-> aeval st a = n.
Proof.
  split; intros. {
    generalize dependent st.
    generalize dependent n.
    induction a; intros; simpl.
    all: try invc H.
    all: try reflexivity.
    - specialize (IHa1 _ _ H2).
```

```
IHa1 : forall (n : nat) (st : state),
          aevalR st a1 n -> aeval st a1 = n
IHa2 : forall (n : nat) (st : state),
         aevalR st a2 n -> aeval st a2 = n
H2 : aevalR st a1 n1
H4 : aevalR st a2 n2
______________________________________(1/1)
aeval st a1 + aeval st a2 = n1 + n2
```

After:

```
IHa1 : aeval st a1 = n1
```

Proof by induction on the hypothesis

```
Lemma aeval_iff_aevalR : forall st a n,
  aevalR st a n <-> aeval st a = n.
Proof.
  split; intros. {
    induction H.
```

Extra slides

Recap functions as relations (1/2)

> What is the signature of the proposition that represents `plus`?

```
plus: nat → nat → nat
```

Recap functions as relations (1/2)

What is the signature of the proposition that represents `plus`?

```
plus: nat → nat → nat

Plus: nat → nat → nat → Prop
```

Recap functions as relations (2/2)

> How do we represent `plus` as a proposition?

```
Fixpoint plus (n m : nat) : nat :=
  match n with
  | 0 ⇒ m
  | S p ⇒ S (plus p m)
  end.
```

Recap functions as relations (2/2)

How do we represent `plus` as a proposition?

```
Fixpoint plus (n m : nat) : nat :=
  match n with
  | 0 ⇒ m
  | S p ⇒ S (plus p m)
  end.
```

```
Induction Plus: nat → nat → nat → Prop :
| plus_0: forall n, Plus 0 n n
| plus_n: forall n m o,
  Plus n m o →
  Plus (S n) m (S o).
```

$$\frac{}{0 + n = n} \qquad \frac{n + m = o}{\mathrm{S}(n) + m = \mathrm{S}(o)}$$

Recall `optimize_0plus`

```
Fixpoint optimize_0plus (a:aexp) : aexp :=
  match a with
  | ANum n ⇒ ANum n
  | APlus (ANum 0) e2 ⇒ optimize_0plus e2
  | APlus e1 e2 ⇒ APlus (optimize_0plus e1) (optimize_0plus e2)
  | AMinus e1 e2 ⇒ AMinus (optimize_0plus e1) (optimize_0plus e2)
  | AMult e1 e2 ⇒ AMult (optimize_0plus e1) (optimize_0plus e2)
  end.
```

optimize_0plus as a relation

```
Inductive Opt_0plus: aexp → aexp → Prop :=
(* Optmize *)
| opt_0plus_do: forall a, Opt_0plus (APlus (ANum 0) a) a
(* No optimization *)
| opt_0plus_skip: forall a1 a2, a1 <> ANum 0 → Opt_0plus (a1 + a2) (a1 + a2)
(* Recurse *)
| opt_0plus_plus:
  forall a1 a2 a1' a2',
  Opt_0plus a1 a1' →
  Opt_0plus a2 a2' →
  Opt_0plus (APlus a1 a2) (APlus a1 a2')
| opt_0plus_minus: forall a1 a2 a1' a2',
  Opt_0plus a1 a1' → Opt_0plus a2 a2' → Opt_0plus (AMinus a1 a2) (AMinus a1' a2')
| opt_0plus_mult: forall a1 a2 a1' a2',
  Opt_0plus a1 a1' → Opt_0plus a2 a2' → Opt_0plus (AMult a1 a2) (AMult a1' a2').
```

How can we generalize the optimization step?

Generalizing optimizations

```
Inductive Opt (O : aexp → aexp → Prop) : aexp → aexp → Prop :=
(* No optimization *)
| opt_skip : forall a, (forall a', ~ O a a') → Opt O a a
(* Optimize code *)
| opt_do : forall a a', O a a' → Opt O a a'
(* Recurse *)
| opt_plus : forall a1 a2 a1' a2' : aexp,
             Opt O a1 a1' →
             Opt O a2 a2' → Opt O (a1 + a2) (a1' + a2')
| opt_minus : forall a1 a2 a1' a2' : aexp,
             Opt O a1 a1' →
             Opt O a2 a2' → Opt O (a1 - a2) (a1' - a2')
| opt_mult : forall a1 a2 a1' a2' : aexp,
             Opt O a1 a1' →
             Opt O a2 a2' → Opt O (a1 * a2) (a1' * a2').
```

Generalizing Soundness

```
Definition IsSound (O:aexp → aexp → Prop) :=
  forall a a',
  O a a' →
  forall st,
  aeval st a = aeval st a'.

Theorem opt_sound:
  forall O : aexp → aexp → Prop,
  IsSound O →
  IsSound (Opt O).
(* Show that [optimize_0plus] is sound *)
Inductive MyOpt: aexp → aexp → Prop :=
| my_opt_def: forall (a:aexp), MyOpt (0 + a) a.

Theorem my_opt_sound: IsSound (Opt MyOpt).
```

How to write a functional version of `Opt`?

A functional version of Opt

```
Fixpoint opt (f : aexp → option aexp) (a:aexp) : aexp :=
match f a with
| Some a ⇒ a (* Optimize step *)
| None ⇒
  match a with
  | APlus a1 a2 ⇒ opt f a1 + opt f a2    (* Recurse *)
  | AMinus a1 a2 ⇒ opt f a1 - opt f a2
  | AMult a1 a2 ⇒ opt f a1 * opt f a2
  | _ ⇒ a                                (* Skip *)
  end
end.
```

Notice how `option` encodes the fact that the proposition may/may-not hold.

Proving opt_func soundness

```
Definition IsFuncSound f :=
  forall a a',
    f a = Some a' ->
    forall st,
    aeval st a = aeval st a'.

Theorem opt_func_sound:
  forall f : aexp -> option aexp,
  IsFuncSound f ->
  forall (a : aexp) (st : state),
  aeval st a = aeval st (opt f a).
```

On functions as relations

UMASS BOSTON

> Notice how it was simpler to prove the same result using the inductive definition. Why?

On functions as relations

> Notice how it was simpler to prove the same result using the inductive definition. Why?

- Functions-as-relations include an inductive principle ***(Proof by induction on the derivation tree.)***
- Functions-as-relations are more expressive ***(eg, representing non-terminating behaviors.)***
- Functions can use Coq's evaluation power ***(recall proof by reflection, lecture 10)***
- Functions can be translated automatically into OCaml/Haskell ***(next lecture)***