CS 565

Programming Languages (graduate)
Spring 2024

Week 5
Curry-Howard Correspondence, Induction Principles

Observation

Two ways of thinking about $\rightarrow$:

- As a type constructor:
  f: $A \rightarrow B$ denotes the type of a function that transforms elements of A into elements of B
- As a logical implication:
  $A \rightarrow B$ establishes the validity of proposition B given the validity of proposition A

How are these notions related?

Observation

They are exactly the same!

Logical implication models the type of functions that transforms evidence (aka proofs):

$A \to B$ represents the type of all functions that given evidence for the validity of A, returns a proof (aka evidence) for the validity of B

Curry-Howard Isomorphism

Propositions $\sim$ Types

Proofs $\sim$ Values

- a proof is a program and its type is the proposition it proves
- the return type of a function is a theorem whose validity is established by the types of its arguments

Propositions


```
Inductive ev : nat → Prop :=
|   ev_0 : ev 0
|   ev_SS (n : nat) (H : ev n) : ev (S (S n))
```

Read ":" to mean "proof of"

The type of `ev_SS` is:

```
∀n.  ev n → ev (S (S n))
```

What is an element that inhabits `ev 4`?

It is the proof object (proof tree):

```
ev_SS 2 (ev_SS 0 ev_0)
```

This object is built via the following proof script:

```
apply ev_SS.
apply ev_SS.
apply ev_0.
```

Alternatively


```
Theorem ev_plus4: ∀n, ev n → ev (4 + n).
Proof.
  intros n H.
  simpl.
  apply ev_SS. apply ev_SS. apply H.
Qed.
```

Here is an object that has this type:

```
Definition ev_plus4' : ∀n, ev n → ev (4 + n) :=
  fun (n : nat) => fun (H : ev n) =>
    ev_SS (S (S n)) (ev_SS n H).
```

Also:

```
Definition ev_plus4'' (n : nat) (H : ev n) : ev (4 + n) :=
    ev_SS (S (S n)) (ev_SS n H).
```

Observation

- Quantification allows us to refer to the value of an argument in the type of another:

```
∀n, ev n → ev (4 + n)
```

- Implication is essentially a degenerate form of quantification:

```
∀ (x: nat), nat
∀ (_: nat), nat
nat → nat
```

`∀ (_ : P), Q` is the same as `P → Q`

Equality


```
Inductive eq {X:Type} : X -> X -> Prop :=
  | eq_refl : ∀ x, eq x x.
```

Given a set X, define a family of propositions that characterize what it means for two elements x and y to be equal.

The only evidence for equality is when two elements are "semantically" identical.
- semantic equivalence means convertibility of terms according to a set of meaning-preserving computation rules.

Logical Connectives


```
Inductive and (P Q : Prop) : Prop :=
  | conj : P -> Q -> and P Q.
```

This is a form of product type, defined over propositions (cf. prod in Poly.v)

```
Inductive or (P Q : Prop) : Prop :=
  | or_introl : P -> or P Q
  | or_intror : Q -> or P Q.
```

This is a form of sum type, defined over propositions

Induction Principles


```
Inductive nat :
| 0
| S (n : nat).
```

```
Check nat_ind :
  forall P : nat -> Prop,
    P 0 ->
    (forall n : nat, P n -> P (S n)) ->
    forall n : nat, P n.
```

```
Inductive time :
| day
| night.
```

```
Check time_ind :
  forall P : time -> Prop,
    P day ->
    P night ->
    forall t : time, P t.
```

More generally, for a type with n constructors, an induction principle of the following shape is generated:

```
t_ind : forall P : t -> Prop,
          ... case for c1 ... ->
          ... case for c2 ... -> ...
          ... case for cn ... ->
          forall n : t, P n
```

Polymorphism


```
Inductive list (X:Type) : Type :=
| nil : list X
| cons : X -> list X -> list X.
```

```
list_ind :
  forall (X : Type) (P : list X -> Prop),
     P [] ->
     (forall (x : X) (l : list X), P l -> P (x :: l)) ->
     forall l : list X, P l
```

`list_ind` is a polymorphic function parameterized over type X

Induction Principles for Propositions


```
Inductive ev : nat -> Prop :=
  | ev_0 : ev 0
  | ev_SS : forall n : nat, ev n -> ev (S (S n)))
```

```
Check ev_ind :
  forall P : nat -> Prop,
    P 0 ->
    (forall n : nat, ev n -> P n -> P (S (S n))) ->
    forall n : nat, ev n -> P n.
```