Interactive Theorem Proving

Lecture 2: The Curry-Howard Correspondence

Elias Castegren and David Broman

2 May 2024

UPPSALA
UNIVERSITET

Inductive Definitions

- Consider the following (incomplete) grammar for propositional logic

$$\phi ::= T \mid F \mid \phi \wedge \phi \mid \phi \vee \phi$$

- The grammar gives an *inductive* definition of the syntax of formulae
  - Any formula ϕ must have one of the four forms above
    - Allows *case analysis*
  - The inductive structure is implicitly *finite*
    - Allows *proof by induction*

Well-Founded Recursion

- When we write this…

$$\phi ::= T \mid F \mid \phi \wedge \phi \mid \phi \vee \phi$$

- …what we're *actually* defining is this:

$$\phi_0 = \{T, F\} \qquad (\phi_n \text{ is the set of formulae with a maximum of } n \text{ connectives})$$

$$\phi_1 = \{A \wedge B \mid A, B \in \phi_0\} \cup \{A \vee B \mid A, B \in \phi_0\} \cup \phi_0$$

$$\ldots$$

$$\phi_\omega = \{A \wedge B \mid A, B \in \phi_{\omega-1}\} \cup \{A \vee B \mid A, B \in \phi_{\omega-1}\} \cup \phi_{\omega-1}$$

Induction principle

We can prove $P(\phi)$ for $\phi \in \phi_n$ for any n by proving:

1. $P(A)$ for $A \in \phi_0$
2. $P(A \wedge B)$, assuming $P(A)$ and $P(B)$, for $A, B \in \phi_k$
3. $P(A \vee B)$, assuming $P(A)$ and $P(B)$, for $A, B \in \phi_k$

1. proves P for all of ϕ_0. 2. and 3. then gives $\phi_1, \phi_2, \ldots, \phi_n$

Algebraic Datatypes

- Consider the definition of an algebraic data type

```
Inductive nat_list :=
| Nil
| Cons (n: nat) (l : nat_list).
```

```
type nat_list =
| Nil
| Cons of nat * nat_list
```

```
data NatList =
  Nil
| Cons Nat NatList
```

- This gives an *inductive* definition of the shape of lists
  - Any value of type `nat_list` must have one of the two forms above
    - Allows *pattern matching*
  - Any value of type `nat_list` is finite (not looking at you, Haskell…)
    - Allows *recursive functions*

Inductive Definitions (again)

- We can further define *inductive relations* over formulae, for example

$$\frac{A\ \mathbf{true} \quad B\ \mathbf{true}}{A \wedge B\ \mathbf{true}} \qquad \frac{A\ \mathbf{true}}{A \vee B\ \mathbf{true}} \qquad \frac{B\ \mathbf{true}}{A \vee B\ \mathbf{true}} \qquad \frac{}{T\ \mathbf{true}}$$

- The relation above gives an inductive definition of its derivation trees
  - Any proof of $\phi\ \mathbf{true}$ must have one of the four forms above
    - Allows *case analysis* (inversion)
  - The inductive structure of the derivation tree is implicitly *finite*
    - Allows *proof by induction*

Well-Founded Recursion (again)

- When we write this…

$$\frac{A\ \mathbf{true} \quad B\ \mathbf{true}}{A \wedge B\ \mathbf{true}} \qquad \frac{A\ \mathbf{tru}}{A \vee B\ \mathbf{t}}$$

- …what we're *actually* defin

$$\tau_0 = \left\{ \frac{}{T\ \mathbf{true}} \right\}$$

(τ_n is the set of derivation trees for formulas in ϕ_n)

Induction principle

We can prove $\phi\ \mathbf{true} \implies P(\phi)$, for $\phi\ \mathbf{true} \in \tau_n$ for any n by proving

1. $P(T)$ ($\phi\ \mathbf{true}$ came from τ_0)
2. $P(A \wedge B)$ assuming $A\ \mathbf{true}, B\ \mathbf{true}, P(A)$ and $P(B)$
3. $P(A \vee B)$ assuming $A\ \mathbf{true}$ and $P(A)$
4. $P(A \vee B)$ assuming $B\ \mathbf{true}$ and $P(B)$

In cases 2-4 we are assuming $A\ \mathbf{true}, B\ \mathbf{true} \in \tau_k$, for any k

$$\tau_1 = \left\{ \frac{A\ \mathbf{true} \quad B\ \mathbf{true}}{A \wedge B\ \mathbf{true}} \mid A\ \mathbf{true}, B\ \mathbf{true} \in \tau_0 \right\} \bigcup \left\{ \frac{A\ \mathbf{true}}{A \vee B\ \mathbf{true}} \mid A\ \mathbf{true} \in \tau_0 \right\} \bigcup \left\{ \frac{B\ \mathbf{true}}{A \vee B\ \mathbf{true}} \mid B\ \mathbf{true} \in \tau_0 \right\} \bigcup \tau_0$$

$\ldots$

$$\tau_\omega = \left\{ \frac{A\ \mathbf{true} \quad B\ \mathbf{true}}{A \wedge B\ \mathbf{true}} \mid A\ \mathbf{true}, B\ \mathbf{true} \in \tau_{\omega-1} \right\} \bigcup \left\{ \frac{A\ \mathbf{true}}{A \vee B\ \mathbf{true}} \mid A\ \mathbf{true} \in \tau_{\omega-1} \right\} \bigcup \left\{ \frac{B\ \mathbf{true}}{A \vee B\ \mathbf{true}} \mid B\ \mathbf{true} \in \tau_{\omega-1} \right\} \bigcup \tau_{\omega-1}$$

Syntactic Proofs

- Both formulae and properties of these define *syntactic* structures

$$\phi ::= T \mid F \mid \phi \wedge \phi \mid \phi \vee \phi$$

$$\tau ::= \frac{A\ \mathbf{true} \quad B\ \mathbf{true}}{A \wedge B\ \mathbf{true}} \mid \frac{A\ \mathbf{true}}{A \vee B\ \mathbf{true}} \mid \frac{B\ \mathbf{true}}{A \vee B\ \mathbf{true}} \mid \frac{}{T\ \mathbf{true}}$$

- Proving $\phi\ \mathbf{true}$ amounts to *producing a derivation tree* τ with $\phi\ \mathbf{true}$ as its root
  - Proof that $T \wedge (F \vee T)\ \mathbf{true}$:

$$\frac{\dfrac{}{T\ \mathbf{true}} \quad \dfrac{\dfrac{}{T\ \mathbf{true}}}{F \vee T\ \mathbf{true}}}{T \wedge (F \vee T)\ \mathbf{true}}$$

Syntactic Proofs (cont.)

$$\tau ::= \frac{A\ \mathbf{true} \quad B\ \mathbf{true}}{A \wedge B\ \mathbf{true}} \mid \frac{A\ \mathbf{true}}{A \vee B\ \mathbf{true}} \mid \frac{B\ \mathbf{true}}{A \vee B\ \mathbf{true}} \mid \frac{}{T\ \mathbf{true}}$$

- Induction principles are *procedures* for building derivation trees
- Prove that $\phi\ \mathbf{megatrue} \implies \phi\ \mathbf{true}$

$$\frac{A\ \mathbf{megatrue} \quad B\ \mathbf{megatrue}}{A \wedge B\ \mathbf{megatrue}} \quad \frac{A\ \mathbf{megatrue} \quad B\ \mathbf{megatrue}}{A \vee B\ \mathbf{megatrue}} \quad \frac{}{T\ \mathbf{megatrue}}$$

- By induction over the derivation tree of $\phi\ \mathbf{megatrue}$
  1. Base case: $T\ \mathbf{true}$ trivially.
  2. Case $A \wedge B\ \mathbf{megatrue}$: By the induction hypotheses we get $A\ \mathbf{true}$ and $B\ \mathbf{true}$. These together give $A \wedge B\ \mathbf{true}$.
  3. Similar to 2. but either rule for $A \vee B\ \mathbf{true}$ can be used.

Syntactic Proofs (cont.)

$$\tau ::= \frac{A\ \textbf{true} \quad B\ \textbf{true}}{A \wedge B\ \textbf{true}} \mid \frac{A\ \textbf{true}}{A \vee B\ \textbf{true}} \mid \frac{B\ \textbf{true}}{A \vee B\ \textbf{true}} \mid \frac{}{T\ \textbf{true}}$$

- Induction principles are *procedures* for building derivation trees
- Prove that $\phi\ \textbf{megatrue} \implies \phi\ \textbf{true}$

$$\sigma ::= \frac{A\ \textbf{megatrue} \quad B\ \textbf{megatrue}}{A \wedge B\ \textbf{megatrue}} \quad \frac{A\ \textbf{megatrue} \quad B\ \textbf{megatrue}}{A \vee B\ \textbf{megatrue}} \quad \frac{}{T\ \textbf{megatrue}}$$

- $buildProof(\sigma : \phi\ \textbf{megatrue}) : \phi\ \textbf{true} =$

  `match` σ `with`

  $\mid\ T\ \textbf{megatrue} \Rightarrow \dfrac{}{T\ \textbf{true}}$

  $\mid\ \dfrac{A\ \textbf{megatrue} \quad B\ \textbf{megatrue}}{A \wedge B\ \textbf{megatrue}} \Rightarrow \dfrac{buildProof(A\ \textbf{megatrue}) \quad buildProof(B\ \textbf{megatrue})}{A \wedge B\ \textbf{true}}$

  $\mid\ \dfrac{A\ \textbf{megatrue} \quad B\ \textbf{megatrue}}{A \vee B\ \textbf{megatrue}} \Rightarrow \dfrac{buildProof(A\ \textbf{megatrue})}{A \vee B\ \textbf{true}}$

Curry-Howard Correspondence

Haskell Curry

- Curry-Howard Isomorphism
- Curry-Howard Equivalence

- Proofs as programs
- Propositions as types

~~OCaml Howard~~

William Howard

A few Correspondences

In Logic	In Programming
Propositions	Types
Proofs	Programs (terms)
Implication $A \implies B$	Function types `A -> B`
Conjunction $A \wedge B$	Product types `A * B`
Disjunction $A \vee B$	Sum types `A | B`
Case analysis	Pattern matching
True	The unit type
False	The empty type

Propositions as Types

- "Propositions as types" connects to writing syntactic proofs

Lemma $1 : \forall n\,.\,n < n + 1$

Given any n, Lemma 1 can produce a proof of $n < n + 1$

"Lemma 1 is a function from numbers n to proofs of $n < n + 1$"

Lemma $2 : \forall m\ n\,.\,m < n \implies m < n + 1$

If I have a proof of $m < n$, Lemma 2 can give me a proof of $m < n + 1$

"Lemma 2 is a function from (a proof of) $m < n$ to (a proof of) $m < n + 1$"

- Prove $x < x + 2$

Using Lemma 1 with x, we get $x < x + 1$ (**Fact 1**).

Using Lemma 2 with **Fact 1**, we get $x < (x + 1) + 1$.

By properties of $+$, we get $(x + 1) + 1 = x + 2$. □

```
let Fact1 = Lemma1(x) in
let Fact2 = Lemma2(x, x+1, Fact1) in
simplify(Fact2)
```

Propositions as Types

- “Propositions as types” helps when writing syntactic proofs

```
function create_log_entry(s: String): LogEntry =
  ...
end

function certify_log_entry(e: LogEntry): Certificate[LogEntry] =
  ...
end

function verify_operation(op: Operation): Certificate[LogEntry] =
  var s_op = to_string(op)
  var entry = create_log_entry(s_op)
  return certify_log_entry(entry)
end
```

“Hm, if I had a log entry I could build a certificate…”

- Pr

By properties of $+$, we get $(x + 1) + 1 = x + 2$. □

True and False

- The proposition $True$ corresponds to the unit type
  - There is a single constructor $()$ carrying no information
  - We can *always* produce a proof of $True$
- The proposition $False$ corresponds to the empty type
  - There are no constructors
  - We can *never* produce a proof of $False$

```
exFalsoQuodlibet(p : False) : Anything =
  match p with
  (* End of proof *)
```

Curry-Howard goes Deep

- There are many more instances of the Curry-Howard correspondence
  - System F — Polymorphic Lambda Calculus
  - Modal logic — Monads
  - Linear Logic — Session Types
  - …

- See (e.g.)“Propositions as types“ by Philip Wadler
  - Talk: https://www.youtube.com/watch?v=aeRVdYN6fE8

Questions so far?

UPPSALA
UNIVERSITET

Proof Objects in Coq

- The tactics we write in Coq actually produce *proof objects*
- Proof objects reify the Curry-Howard correspondence
  - Inductive definitions are inductive data types
  - Proofs explicitly build proof objects
  - Case analysis (`destruct`) turns into pattern matching
  - The proof object of an implication $A \implies B$ is a function from A to B
    - The function builds the proof object B given a proof object A
  - Induction shows up as (guarded) recursive functions

Questions?

UPPSALA
UNIVERSITET

Tactics and Program Synthesis

- Tactics build proof objects…
- Proof objects are programs…
- Can we build programs using tactics?

```
Fixpoint count_false (f: phi): nat.
  destruct f.
  - apply 0.
  - apply 1.
  - apply (count_false f1 + count_false f2).
  - apply (count_false f1 + count_false f2).
Defined.
```

```
(fix count_false (f : phi) : nat := ?Goal)
```

Tactics and Program Synthesis

- Tactics build proof objects…
- Proof objects are programs…
- Can we build programs using tactics?

```
Fixpoint count_false (f: phi): nat.
  destruct f.
  - apply 0.
  - apply 1.
  - apply (count_false f1 + count_false f2).
  - apply (count_false f1 + count_false f2).
Defined.
```

```
(fix count_false (f : phi) : nat :=
   match f with
   | T => ?Goal
   | F => ?Goal0
   | And phi1 phi2 => (fun f1 f2 : phi => ?Goal1) phi1 phi2
   | Or phi1 phi2 => (fun f1 f2 : phi => ?Goal2) phi1 phi2
   end)
```

Tactics and Program Synthesis

- Tactics build proof objects…
- Proof objects are programs…
- Can we build programs using tactics?

```
Fixpoint count_false (f: phi): nat.
  destruct f.
  - apply 0.
  - apply 1.
  - apply (count_false f1 + count_false f2).
  - apply (count_false f1 + count_false f2).
Defined.
```

```
(fix count_false (f : phi) : nat :=
   match f with
   | T => 0
   | F => ?Goal0
   | And phi1 phi2 => (fun f1 f2 : phi => ?Goal1) phi1 phi2
   | Or phi1 phi2 => (fun f1 f2 : phi => ?Goal2) phi1 phi2
   end)
```

Tactics and Program Synthesis

- Tactics build proof objects…
- Proof objects are programs…
- Can we build programs using tactics?

```
Fixpoint count_false (f: phi): nat.
  destruct f.
  - apply 0.
  - apply 1.
  - apply (count_false f1 + count_false f2).
  - apply (count_false f1 + count_false f2).
Defined.
```

```
(fix count_false (f : phi) : nat :=
   match f with
   | T => 0
   | F => 1
   | And phi1 phi2 => (fun f1 f2 : phi => ?Goal1) phi1 phi2
   | Or phi1 phi2 => (fun f1 f2 : phi => ?Goal2) phi1 phi2
   end)
```

Tactics and Program Synthesis

- Tactics build proof objects…
- Proof objects are programs…
- Can we build programs using tactics?

```
Fixpoint count_false (f: phi): nat.
  destruct f.
  - apply 0.
  - apply 1.
  - apply (count_false f1 + count_false f2).
  - apply (count_false f1 + count_false f2).
Defined.
```

```
(fix count_false (f : phi) : nat :=
   match f with
   | T => 0
   | F => 1
   | And phi1 phi2 => count_false f1 + count_false f2
   | Or phi1 phi2 => (fun f1 f2 : phi => ?Goal2) phi1 phi2
   end)
```

(Cheating somewhat for clarity)

Tactics and Program Synthesis

- Tactics build proof objects…
- Proof objects are programs…
- Can we build programs using tactics?

```
Fixpoint count_false (f: phi): nat.
  destruct f.
  - apply 0.
  - apply 1.
  - apply (count_false f1 + count_false f2).
  - apply (count_false f1 + count_false f2).
Defined.
```

```
(fix count_false (f : phi) : nat :=
   match f with
   | T => 0
   | F => 1
   | And phi1 phi2 => count_false f1 + count_false f2
   | Or phi1 phi2 => count_false f1 + count_false f2
   end)
```

(Cheating somewhat for clarity)

Tactics and Program Synthesis

- Tactics build proof objects…
- Proof objects are programs…
- Can we build programs using tactics?

Ending with “Defined” allows computation

```
Fixpoint count_false (f: phi): nat.
  destruct f.
  - apply 0.
  - apply 1.
  - apply (count_false f1 + count_false f2).
  - apply (count_false f1 + count_false f2).
Defined.
```

```
(fix count_false (f : phi) : nat :=
   match f with
   | T => 0
   | F => 1
   | And phi1 phi2 => count_false f1 + count_false f2
   | Or phi1 phi2 => count_false f1 + count_false f2
   end)
```

(Cheating somewhat for clarity)

Induction Principles

- Let's go back to the induction principle for formulae from before:

Induction principle

We can prove $P(\phi)$ for any ϕ by proving

1. $P(T)$
2. $P(F)$
3. $P(A \wedge B)$, assuming $P(A)$ and $P(B)$
4. $P(A \vee B)$, assuming $P(A)$ and $P(B)$

If we can provide these proofs…

…we will get a proof of: $\forall \phi \,.\, P(\phi)$

$$Ind_\phi : \forall P \,.\, P(T) \to P(F) \to (\forall A, B \,.\, P(A) \to P(B) \to P(A \wedge B)) \to (\forall A, B \,.\, P(A) \to P(B) \to P(A \vee B)) \to \forall \phi \,.\, P(\phi)$$

Induction Principles

- We can write this induction principle by hand

$$Ind_\phi : \forall P\,.\,P(T) \to P(F) \to (\forall A, B\,.\,P(A) \to P(B) \to P(A \wedge B)) \to (\forall A, B\,.\,P(A) \to P(B) \to P(A \vee B)) \to \forall \phi\,.\,P(\phi)$$

- …but Coq generates it for us automatically!

```
Fixpoint phi_ind (P : phi -> Prop)
  (case_t : P T) (case_f : P F)
  (case_and : forall A B, P A -> P B -> P (And A B))
  (case_or : forall A B, P A -> P B -> P (Or A B))
  (f : phi)
  : P f :=
  match f with
  | T => case_t
  | F => case_f
  | And A B => case_and A B
                (phi_ind P case_t case_f case_and case_or A)
                (phi_ind P case_t case_f case_and case_or B)
  | Or A B => case_or A B
                (phi_ind P case_t case_f case_and case_or A)
                (phi_ind P case_t case_f case_and case_or B)
  end.
```

Custom Induction Principles

- Sometimes the generated induction principle is not very useful

```
Inductive rose_tree (A: Type) :=
| Node (elem: A) (children: list (rose_tree A)).
```

Generated induction principle has type

$$\forall (A : Type)\ (P : rose\_tree\ A \to Prop).\ (\forall e, ts.\ P(Node\ A\ e\ ts)) \to \forall t, P(t)$$

If you can prove that P holds for a node with any element and children, then P holds for any tree t

- A more useful principle would be:

$$\forall (A : Type)\ (P : rose\_tree\ A \to Prop).\ (\forall e, ts.\ \forall t' \in ts.\ P(t') \to P(Node\ A\ e\ ts)) \to \forall t, P(t)$$

We can write this by hand (or even better, generate it using tactics!)

Custom Induction Principles

```
Fixpoint rose_tree_ind' (A: Type) (P: rose_tree A -> Prop)
  (case_node: forall e ts, Forall P ts -> P (Node A e ts))
  (t: rose_tree A) : P t :=
  match t with
  | Node _ e ts =>
      case_node e ts
        (list_ind (Forall P)
           (Forall_nil P)
           (fun x xs (IH: Forall P xs) => Forall_cons x (rose_tree_ind'' A P case_node x) IH) ts)
  end.
```

Ugh…

```
Fixpoint rose_tree_ind'' (A: Type) (P: rose_tree A -> Prop)
  (case_node: forall e ts, Forall P ts -> P (Node A e ts))
  (t: rose_tree A) : P t.
Proof with auto.
  destruct t. apply case_node.
  induction children...
  apply Forall_cons...
  apply rose_tree_ind''...
Defined.
```

This is nicer to write
(but obviously doesn't make sense without following the proof interactively)

Fixpoints instead of Theorems

- It is possible to write proofs directly as fixpoints:

```
Fixpoint rebuild_id' (A: Type) (t: rose_tree A):
    rebuild A t = t.
Proof with auto.
  destruct t. simpl.
  f_equal. induction children...
  simpl. rewrite rebuild_id'.
  rewrite IHchildren.
  reflexivity.
Qed.
```

Uses the theorem in its own proof!

- The final **Qed** checks that the resulting proof object is terminating
  - Proof search will sometimes be overly optimistic…

Intuitionistic Logic

- The Curry-Howard equivalent of lambda calculus is *intuitionistic logic*
  - Also known as *constructive logic*
- Rather than assign each proposition $True$ or $False$, require *evidence*

- Excludes certain axioms present in classical logic
  - Law of excluded middle $\forall P \,.\, P \vee \neg P$
  - Double negation elimination $\neg \neg P \implies P$
  - Peirce's law $(\neg A \implies A) \implies A$
  - (Each of these imply the others)

Proof by Contradiction

- If we prove P by showing that $\neg P$ leads to a contradiction we have not produced *evidence* of P
- We do not have a procedure for building the proof object for P, only for building the proof object for $\neg\neg P$

- Specific instances of classical axioms *can* be proved:

```
Theorem is_true_lem:
  forall f,
    is_true f \/ ¬is_true f.
Proof.
  ...
Qed.
```

```
Theorem is_true_dne:
  forall f,
    ¬¬ is_true f -> is_true f.
Proof.
  ...
Qed.
```

Software Foundations also has you prove that it is OK to assume the law of excluded middle: $\forall P\,.\,\neg\neg(P \vee \neg P)$

Conclusions

- The Curry-Howard correspondence connects logic and programming
- Coq embodies this connection explicitly
  - Propositions *are* types
  - Proofs *are* programs

- Most of the time, we program with Gallina and prove with tactics
  - Sometimes it is useful to do it the other way around!
    - In particular, we sometimes need custom induction principles