Proofs are Programs

Timmy Zhu
Spring 2019

Today’s music: *Proof* by Paul Simon
Review

Previously in 3110:
• Functional programming in Coq
• Logic in Coq

Today: A fundamental idea that goes by many names...
• Propositions as types
• Proofs as programs
• Curry–Howard(–Lambek) isomorphism (aka correspondence)
• Brouwer–Heyting–Kolmogorov interpretation
Types = Propositions

**ACT I**
Three innocent functions

```plaintext
let apply f x = f x

let const x = fun _ -> x

let subst x y z = x z (y z)
```
Three innocent functions

```ocaml
let apply f x = f x
  : ('a -> 'b) -> 'a -> 'b

let const x = fun _ -> x
  : 'a -> 'b -> 'a

let subst x y z = x z (y z)
  : ('a -> 'b -> 'c)
-> ('a -> 'b) -> 'a -> 'c
```

Three innocent functions

let apply f x = f x
  : ('a -> 'b) -> 'a -> 'b

let const x = fun _ -> x
  : 'a -> 'b -> 'a

let subst x y z = x z (y z)
  : ('a -> 'b -> 'c)
  -> ('a -> 'b) -> 'a -> 'c
Three innocent functions propositions

let apply f x = f x
  : ('a ⇒ 'b) ⇒ 'a ⇒ 'b

let const x = fun _ → x
  : 'a ⇒ 'b ⇒ 'a

let subst x y z = x z (y z)
  : ('a ⇒ 'b ⇒ 'c)
  ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'c
Three innocent functions propositions

```ocaml
let apply f x = f x
  : ( A ⇒ B) ⇒ A ⇒ B

let const x = fun _ → x
  : A ⇒ B ⇒ A

let subst x y z = x z (y z)
  : ( A ⇒ B ⇒ C)
  ⇒ ( A ⇒ B) ⇒ A ⇒ C
```
Three innocent functions propositions

let apply f x = f x
  : (A \Rightarrow B) \Rightarrow A \Rightarrow B

let const x = fun _ -> x
  : A \Rightarrow (B \Rightarrow A)

let subst x y z = x z (y z)
  : (A \Rightarrow (B \Rightarrow C))
  \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))

Do you recognize these propositions?
A Sound and Complete Axiomatization for Propositional Logic

Consider the following axiom schemes:

A1. $A \Rightarrow (B \Rightarrow A)$
A2. $(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$
A3. $((A \Rightarrow B) \Rightarrow ((A \Rightarrow \neg B) \Rightarrow \neg A))$

These are axioms schemes; each one encodes an infinite set of axioms:

- $P \Rightarrow (Q \Rightarrow P), (P \Rightarrow R) \Rightarrow (Q \Rightarrow (P \Rightarrow R))$ are instances of A1.

**Theorem:** A1, A2, A3 + modus ponens give a sound and complete axiomatization for formulas in propositional logic involving only $\Rightarrow$ and $\neg$. 

Modus Ponens

\[ A \Rightarrow B \]
\[ A \]

\[ B \]
Three innocent functions/propositions

let apply f x = f x
: (A ⇒ B) ⇒ A ⇒ B

let const x = fun _ -> x
: A ⇒ (B ⇒ A)

let subst x y z = x z (y z)
: (A ⇒ (B ⇒ C))
⇒((A ⇒ B) ⇒ (A ⇒ C))
Types and propositions

Logical propositions can be read as program types, and vice versa

<table>
<thead>
<tr>
<th>Type</th>
<th>Proposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type variable ' a</td>
<td>Atomic proposition A</td>
</tr>
<tr>
<td>Function type –&gt;</td>
<td>Implication ⇒</td>
</tr>
</tbody>
</table>
Conjunction and truth

```ocaml
let fst (a,b) = a
  : 'a * 'b -> 'a
let snd (a,b) = b
  : 'a * 'b -> 'b
let pair a b = (a,b)
  : 'a -> 'b -> 'a * 'b
let tt = ()
  : unit
```
Conjunction and truth

let fst (a,b) = a
: (A ∧ B) ⇒ A

let snd (a,b) = b
: (A ∧ B) ⇒ B

let pair a b = (a,b)
: A ⇒ (B ⇒ (A ∧ B))

let tt = ()
: true
### Types and propositions

Logical propositions can be read as program types, and vice versa

<table>
<thead>
<tr>
<th>Type</th>
<th>Proposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type variable 'a</td>
<td>Atomic proposition A</td>
</tr>
<tr>
<td>Function type –&gt;</td>
<td>Implication ⇒</td>
</tr>
<tr>
<td>Product type *</td>
<td>Conjunction ∧</td>
</tr>
<tr>
<td>unit</td>
<td>True</td>
</tr>
</tbody>
</table>
Disjunction

```haskell
type ('a, 'b) or' = Left of 'a | Right of 'b

let left (x:'a) = Left x
  : 'a -> ('a, 'b) or'

let right (y:'b) = Right y
  : 'b -> ('a, 'b) or'

let match' (f1:'a -> 'c) (f2:'b -> 'c) = function
  | Left v1 -> f1 v1
  | Right v2 -> f2 v2
  : ('a -> 'c) -> ('b -> 'c) -> ('a, 'b) or' -> 'c
```

Read ('a,'b) or' as A V B
Disjunction

type ('a,'b) or' = Left of 'a | Right of 'b

let left (x:'a) = Left x
   : A ⇒ (A V B)

let right (y:'b) = Right y
   : B ⇒ (A V B)

let match' (f1:'a -> 'c) (f2:'b -> 'c) = function
   | Left v1 -> f1 v1
   | Right v2 -> f2 v2
   : (A ⇒ C) ⇒ (B ⇒ C) ⇒ (A V B) ⇒ C
Types and propositions

Logical propositions can be read as program types, and vice versa

<table>
<thead>
<tr>
<th>Type</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type variable 'a</td>
<td>Atomic proposition A</td>
</tr>
<tr>
<td>Function type (\rightarrow)</td>
<td>Implication (\Rightarrow)</td>
</tr>
<tr>
<td>Product type *</td>
<td>Conjunction (&amp;)</td>
</tr>
<tr>
<td>unit</td>
<td>True</td>
</tr>
<tr>
<td>Tagged union</td>
<td>Disjunction (\lor)</td>
</tr>
</tbody>
</table>

False and negation also possible; see slides at end
Program types and logical propositions are fundamentally the same idea.
Programs = Proofs

ACT II
Innocent typing rule

• Recall typing contexts and judgements [lec18]
  – Typing context $T$ is a map from variable names to types
  – Typing judgment $T \vdash e : t$ says that $e$ has type $t$ in context $T$

• Typing rule for function application:
  – $T \vdash e_1 : t \rightarrow u$
  – and $T \vdash e_2 : t$
  – then $T \vdash e_1 \ e_2 : u$
Innocent typing rule

if \( T \vdash e_1 : t \rightarrow u \) and \( T \vdash e_2 : t \) then \( T \vdash e_1 \ e_2 : u \)
Innocent typing rule

if $T \vdash e_1 : t \rightarrow u$
and $T \vdash e_2 : t$
then $T \vdash e_1 \; e_2 : u$
Innocent typing rule

if \( T \vdash e_1 : t \rightarrow u \)
and \( T \vdash e_2 : t \)
then \( T \vdash e_1 \ e_2 : u \)
Innocent typing rule

if \( T \vdash e_1 : t \Rightarrow u \)
and \( T \vdash e_2 : t \)
then \( T \vdash e_1 \ e_2 : u \)

Do you recognize this rule?

\[ \text{Modus Ponens} \]

\[
A \Rightarrow B \\
A \\
\hline \\
B
\]
INTERMISSION
Logical proof systems

• Ways of formalizing what is provable
• Which may differ from what is true or decidable
• Two styles:
  – Hilbert:
    • lots of axioms
    • few inference rules (maybe just modus ponens)
  – Gentzen:
    • lots of inference rules (a couple for each operator)
    • few axioms
Inference rules

From premises $P_1, P_2, \ldots, P_n$

Infer conclusion $Q$

Express allowed means of *inference* or *deductive reasoning*

*Axiom* is an inference rule with zero premises
Judgments

\[ A_1, A_2, ..., A_n \vdash B \]

• From *assumptions* \( A_1, A_2, ..., A_n \)
  – traditional to write \( \Gamma \) for set of assumptions
• Judge that \( B \) is *derivable* or *provable*
• Express allowed means of *hypothetical reasoning*
• \( \Gamma, A \vdash A \) is an axiom
**Inference rules for ⇒ and ∧**

\[
\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \quad \Rightarrow \text{intro}
\]

\[
\frac{\Gamma \vdash A \Rightarrow B}{\Gamma, A \vdash B} \quad \Rightarrow \text{elim}
\]

\[
\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \land B} \quad \land \text{intro}
\]

\[
\frac{\Gamma \vdash A \land B}{\Gamma \vdash A} \quad \land \text{elim 1}
\]

\[
\frac{\Gamma \vdash A \land B}{\Gamma \vdash B} \quad \land \text{elim 2}
\]
Introduction and elimination

• Introduction rules say how to *define* an operator
• Elimination rules say how to *use* an operator
• Gentzen's insight: every operator should come with intro and elim rules
BACK TO THE SHOW
**Innocent typing rule**

if \( T \vdash e_1 : t \rightarrow u \)

and \( T \vdash e_2 : t \)

then \( T \vdash e_1 \ e_2 : u \)
Innocent typing rule

if $T \vdash e_1 : t \rightarrow u$
and $T \vdash e_2 : t$
then $T \vdash e_1 \; e_2 : u$

\[ T \vdash e_1 : t \rightarrow u \quad T \vdash e_2 : t \]
\[ \frac{}{T \vdash e_1 \; e_2 : u} \]
Innocent typing rule

If \( T \vdash e_1 : t \rightarrow u \) and \( T \vdash e_2 : t \), then \( T \vdash e_1 \ e_2 : u \)

\[
\Gamma \vdash e_1 : t \Rightarrow u \quad \Gamma \vdash e_2 : t \\
\frac{}{\Gamma \vdash e_1 \ e_2 : u} \Rightarrow \text{elim}
\]

Modus ponens is function application
Computing with evidence

• Modus ponens (aka $\Rightarrow$ elim) is a way of computing with evidence
  – Given evidence $e_2$ that $t$ holds
  – And given a way $e_1$ of transforming evidence for $t$ into evidence for $u$
  – MP produces evidence for $u$ by applying $e_1$ to $e_2$
• So $e_1 \ e_2$ is a program... and a proof!

\[
\begin{align*}
T \vdash e_1 : t & \rightarrow u \quad T \vdash e_2 : t \\
\hline
T \vdash e_1 \ e_2 : u
\end{align*}
\]
More typing rules

\[\Gamma, x : t \vdash e : u\]
\[\Gamma \vdash \text{fun } x \rightarrow e : t \rightarrow u\]

\[\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2\]
\[\Gamma \vdash (e_1, e_2) : t_1 \times t_2\]
More typing rules

\[ \Gamma, x : t \vdash e : u \]

\[ \Gamma \vdash \text{fun} \ x \to e : t \Rightarrow u \]

\[ \Rightarrow \text{intro} \]

\[ \Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2 \]

\[ \Gamma \vdash (e_1, e_2) : t_1 \land t_2 \]

\[ \land \text{intro} \]
More computing with evidence

\[ \Gamma, x : t \vdash e : u \]

\[ \Gamma \vdash \text{fun } x \rightarrow e : t \rightarrow u \]

given evidence \( e \) for \( u \) predicated on evidence \( x \) for \( t \), produce an evidence transformer

\[ \Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2 \]

\[ \Gamma \vdash (e_1, e_2) : t_1 \times t_2 \]

given evidence \( e_i \) for \( t_i \), produce combined evidence for both
Even more typing rules

\[
\Gamma \vdash e : t_1 \times t_2
\]

\[
\Gamma \vdash \text{fst } e : t_1
\]

\[
\Gamma \vdash \text{snd } e : t_2
\]
Even more typing rules

\[ \Gamma \vdash e : t_1 \land t_2 \]

\[ \frac{}{\Gamma \vdash \text{fst } e : t_1} \quad \land \text{elim } 1 \]

\[ \Gamma \vdash e : t_1 \land t_2 \]

\[ \frac{}{\Gamma \vdash \text{snd } e : t_2} \quad \land \text{elim } 2 \]
Even more computing with evidence

\[ \Gamma \vdash e : t1*t2 \]

\[ \Gamma \vdash \text{fst } e : t1 \]

\[ \Gamma \vdash e : t1*t2 \]

\[ \Gamma \vdash \text{snd } e : t2 \]

given evidence \( e \) for both \( t1 \), project out the evidence for one of them
Programs and proofs

• A well-typed program demonstrates that there is at least one value for that type
  – i.e. the that type is inhabited
  – a program is a proof that the type is inhabited

• A proof demonstrates that there is at least one way of deriving a formula
  – i.e. that the formula is provable by manipulating assumptions and doing inference
  – a proof is a program that manipulates evidence

• Proofs are programs, and programs are proofs
Coq proofs are programs

Theorem apply :
   forall A B : Prop, (A -> B) -> A -> B.
Proof.
   intros A B f x. apply f. assumption.
Qed.

Print apply.
apply =
fun (A B : Prop) (f : A -> B) (x : A)
  => f x
  : forall A B : Prop,
     (A -> B) -> A -> B
Programs

and

Proofs

are fundamentally the same idea
Evaluation = Simplification

ACT III
Many proofs/programs

A given proposition/type could have many proofs/programs.

Proposition/type:
• $A \Rightarrow (B \Rightarrow (A \land B))$
• `'a` $\rightarrow$ (`'b` $\rightarrow$ (`'a * `'b))

Proofs/programs:
• `fun x` $\rightarrow$ `fun y` $\rightarrow$
  (`fun z` $\rightarrow$ (`snd z, `fst z)) (y,x)
• `fun x` $\rightarrow$ `fun y` $\rightarrow$ (`snd (y,x), `fst (y,x))
• `fun x` $\rightarrow$ `fun y` $\rightarrow$ (x,y)
Many proofs/programs

Body of each proof/program:
- \( \text{fun } z \rightarrow (\text{snd } z, \text{fst } z) \) (y,x)
- \( \text{snd } (y,x), \text{fst } (y,x) \)
- \( x,y \)

Each is the result of small-stepping the previous
...and in each case, the proof/program gets simpler

Taking an evaluation step corresponds to simplifying the proof
Program evaluation and proof simplification are fundamentally the same idea.
These are all the same ideas

<table>
<thead>
<tr>
<th>Programming</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>Propositions</td>
</tr>
<tr>
<td>Programs</td>
<td>Proofs</td>
</tr>
<tr>
<td>Evaluation</td>
<td>Simplification</td>
</tr>
</tbody>
</table>

Computation is reasoning

Functional programming is fundamental
Upcoming events

• N/A

This is fundamental.

THIS IS 3110
**False**

Read "void" as "false".
Read 'a . 'a as (∀x . x), which is false.

```
type void = {nope : 'a . 'a}

let ff1 = {nope = let rec f x = f x in f ()} : void

let ff2 = {nope = failwith ""} : void
```

Both ff1 and ff2 type check, but neither successfully completes evaluation: not possible to create a value of type void
type void = {nope : 'a . 'a}

let ff1 = {nope = let rec f x = f x in f ()} : void

let ff2 = {nope = failwith ""} : void

let explode (f:void) : 'b = f.nope : void -> 'b

Read "void" as "false". Read 'a . 'a as (\x . x), which is false.
False

type void = {nope : 'a . 'a}

let ff1 = {nope = let rec f x = f x in f ()} : void

let ff2 = {nope = failwith ""} : void

let explode (f:void) : 'b = f.nope : false ⇒ B
Negation

• Syntactic sugar: define $\neg A$ as $A \Rightarrow \text{false}$
• As a type, that would be 'a -> void
Logical propositions can be read as program types, and vice versa

<table>
<thead>
<tr>
<th>Type</th>
<th>Proposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type variable 'a</td>
<td>Atomic proposition A</td>
</tr>
<tr>
<td>Function type (\rightarrow)</td>
<td>Implication (\Rightarrow)</td>
</tr>
<tr>
<td>Product type (*)</td>
<td>Conjunction (\land)</td>
</tr>
<tr>
<td>unit</td>
<td>True</td>
</tr>
<tr>
<td>Tagged union</td>
<td>Disjunction (\lor)</td>
</tr>
<tr>
<td>Type with no values</td>
<td>False</td>
</tr>
<tr>
<td>(syntactic sugar)</td>
<td>Negation (\neg)</td>
</tr>
</tbody>
</table>