Proofs are Programs

Prof. Clarkson
Fall 2017

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

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

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
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))

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

\[ A \Rightarrow B \]
\[ A \]
\[ \hline \]
\[ B \]
Three innocent functions/propositions

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

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

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

MP as axiom

A1

A2
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 →</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 \land B) \Rightarrow A

let snd (a,b) = b
    : (A \land B) \Rightarrow B

let pair a b = (a,b)
    : A \Rightarrow (B \Rightarrow (A \land B))

let tt = ()
    : true
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 →</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

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 ∨ B
Disjunction

\[ \text{type ('a,'b) or' = Left of 'a | Right of 'b} \]

\[
\text{let left (x:'a) = Left x : A \Rightarrow (A \lor B)}
\]

\[
\text{let right (y:'b) = Right y : B \Rightarrow (A \lor B)}
\]

\[
\text{let match' (f1:'a -> 'c) (f2:'b -> 'c) = function}
\]
\[
| \text{Left v1 -> f1 v1} \\
| \text{Right v2 -> f2 v2}
\]
\[
: (A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow (A \lor B) \Rightarrow C
\]
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 -&gt;</td>
<td>Implication ⇒</td>
</tr>
<tr>
<td>Product type *</td>
<td>Conjunction ∧</td>
</tr>
<tr>
<td>unit</td>
<td>True</td>
</tr>
<tr>
<td>Tagged union</td>
<td>Disjunction ∨</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 [lec17]
  – 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:
  – 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 $T \vdash e_1\ e_2 : u$
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?

Modus Ponens

\[
\begin{align*}
A & \Rightarrow B \\
A & \\
\hline
B
\end{align*}
\]
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* $P_1$, $P_2$, ..., $P_n$
- Infer *conclusion* $Q$
- Express allowed means of *inference* or *deductive reasoning*
- *Axiom* is an inference rule with zero premises
Judgments

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

- From *assumptions* \( A_1, A_2, \ldots, A_n \)
  - 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 \( \Rightarrow \) and \( \land \)

\[
\begin{align*}
\Gamma, A & \vdash B \\
\hline
\Gamma & \vdash A \Rightarrow B
\end{align*}
\]

\( \Rightarrow \) intro

\[
\begin{align*}
\Gamma & \vdash A \Rightarrow B \\
\Gamma & \vdash A \\
\hline
\Gamma & \vdash B
\end{align*}
\]

\( \Rightarrow \) elim

\[
\begin{align*}
\Gamma & \vdash A \land B \\
\Gamma & \vdash A \\
\hline
\Gamma & \vdash A
\end{align*}
\]

\( \land \) elim 1

\[
\begin{align*}
\Gamma & \vdash A \land B \\
\Gamma & \vdash A \\
\hline
\Gamma & \vdash B
\end{align*}
\]

\( \land \) 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 \[ T \vdash e_1 \; e_2 : u \]
Innocent typing rule

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

and \( T \vdash e_2 : t \)

then \( T \vdash e_1 \ e_2 : u \)

\[
\Gamma \vdash e_1 : t \Rightarrow u \quad \Gamma \vdash e_2 : t
\]

\[\begin{array}{c}
\Gamma \vdash e_1 \ e_2 : u \\
\end{array}\]

\[\Rightarrow \text{elim}\]

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

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

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

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

\[\Rightarrow \text{intro}\]

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

\[\wedge \text{ intro}\]
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

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

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

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

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

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

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

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

\[
\begin{align*}
\Gamma &\vdash e : t_1 \land t_2 \\
\hline \\
\Gamma &\vdash \text{fst } e : t_1 \\
\end{align*}
\ \land \text{ elim 1}
\]

\[
\begin{align*}
\Gamma &\vdash e : t_1 \land t_2 \\
\hline \\
\Gamma &\vdash \text{snd } e : t_2 \\
\end{align*}
\ \land \text{ elim 2}
\]
Even more computing with evidence

\[
\Gamma \vdash e : t_1 \times t_2 \\
\therefore \quad \Gamma \vdash \text{fst } e : t_1
\]

\[
\Gamma \vdash e : t_1 \times t_2 \\
\therefore \quad \Gamma \vdash \text{snd } e : t_2
\]

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:
- $A \implies (B \implies (A \land B))$
- '(a -> (b -> (a * b)))

Proofs/programs:
- `fun x -> fun y ->
  (fun z -> (snd z, fst z)) (y,x)
- `fun x -> fun y -> (snd (y,x), fst (y,x))
- `fun x -> fun y -> (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.
CONCLUSION
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

• [Wed] A4 due
• [following Wed] MS1 due
• [after that] A5 out

This is fundamental.

THIS IS 3110
**False**

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

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

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

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

Both \( \texttt{ff1} \) and \( \texttt{ff2} \) type check, but neither successfully completes evaluation: not possible to create a value of type \texttt{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
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
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 →</td>
<td>Implication ⇒</td>
</tr>
<tr>
<td>Product type *</td>
<td>Conjunction ∧</td>
</tr>
<tr>
<td>unit</td>
<td>True</td>
</tr>
<tr>
<td>Tagged union</td>
<td>Disjunction ∨</td>
</tr>
<tr>
<td>Type with no values</td>
<td>False</td>
</tr>
<tr>
<td>(syntactic sugar)</td>
<td>Negation ¬</td>
</tr>
</tbody>
</table>