CS 3110

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

```
let apply f x = f x

let const x = fun _ -> x

let subst x y z = x z (y z)
```

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

```
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 $\neg$.

source: http://www.cs.cornell.edu/courses/cs2800/2016fa/lectures/2800logic.pdf

Modus Ponens

$$\begin{array}{l} A \Rightarrow B \\ A \\ \hline B \end{array}$$

source: http://www.cs.cornell.edu/courses/cs2800/2016fa/lectures/2800logic.pdf

Three innocent functions/propositions

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

MP as axiom

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

A1

```
let subst x y z = x z (y z)
  : ( A ⇒ (B ⇒  C))
    ⇒(( A ⇒  B) ⇒ (A ⇒  C))
```

A2

Types and propositions

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

Type	Proposition
Type variable `'a`	Atomic proposition A
Function type `->`	Implication ⇒

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 = ()
   : 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

Type	Proposition
Type variable `'a`	Atomic proposition A
Function type `->`	Implication ⇒
Product type `*`	Conjunction ∧
`unit`	True

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 V B

Disjunction

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

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

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

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

Types and propositions

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

Type	Formula
Type variable `'a`	Atomic proposition A
Function type `->`	Implication ⇒
Product type `*`	Conjunction ∧
`unit`	True
Tagged union	Disjunction ∨

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 ⊢ e : t` says that `e` has type `t` in context T
- Typing rule for function application:
  - if T ⊢ `e1 : t -> u`
  - and T ⊢ `e2 : t`
  - then T ⊢ `e1 e2 : u`

Innocent typing rule

if T ⊢ `e1 : t -> u`

and T ⊢ `e2 : t`

then T ⊢ `e1 e2 : u`

Innocent typing rule

if $T \vdash$ `e1 : t -> u`

and $T \vdash$ `e2 : t`

then $T \vdash$ `e1 e2 : u`

Innocent typing rule

if `T ⊢ e1 : t -> u`

and `T ⊢ e2 : t`

then `T ⊢ e1 e2 : u`

Innocent typing rule

if $\quad T \vdash e1 \; : \; t \Rightarrow u$

and $\quad T \vdash e2 \; : \; t$

then $\quad T \vdash e1 \;\; e2 \; : \; u$

Do you recognize this rule?

Modus Ponens

$$\frac{\begin{array}{l} A \Rightarrow B \\ A \end{array}}{B}$$

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

$$\frac{P_1 \qquad P_2 \qquad \ldots P_n}{Q}$$

- 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 Γ 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 ∧

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow \text{intro}$$

$$\frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B} \Rightarrow \text{elim}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge \text{intro}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge \text{elim 1}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge \text{elim 2}$$

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 $\quad T \vdash$ `e1 : t -> u`

and $\,T \vdash$ `e2 : t`

then $T \vdash$ `e1 e2 : u`

$$\frac{T \vdash \texttt{e1 : t -> u} \qquad T \vdash \texttt{e2 : t}}{T \vdash \texttt{e1 e2 : u}}$$

Innocent typing rule

if $T \vdash$ `e1 : t -> u`

and $T \vdash$ `e2 : t`

then $T \vdash$ `e1 e2 : u`

$$\frac{T \vdash \texttt{e1 : t -> u} \qquad T \vdash \texttt{e2 : t}}{T \vdash \texttt{e1 e2 : u}}$$

Innocent typing rule

if $\quad T \vdash$ `e1 : t -> u`

and $\quad T \vdash$ `e2 : t`

then $\quad T \vdash$ `e1 e2 : u`

$$\frac{\Gamma \vdash \texttt{e1 : t} \Rightarrow \texttt{u} \qquad \Gamma \vdash \texttt{e2 : t}}{\Gamma \vdash \texttt{e1 e2 : u}} \Rightarrow \text{elim}$$

Modus ponens is function application

Computing with evidence

- Modus ponens (aka ⇒ elim) is a way of computing with evidence
  - Given evidence `e2` that `t` holds
  - And given a way `e1` of transforming evidence for `t` into evidence for `u`
  - MP produces evidence for `u` by applying `e1` to `e2`
- So `e1 e2` is a program… and a proof!

$$\frac{\mathtt{T} \vdash \mathtt{e1} : \mathtt{t} \to \mathtt{u} \qquad \mathtt{T} \vdash \mathtt{e2} : \mathtt{t}}{\mathtt{T} \vdash \mathtt{e1}\ \mathtt{e2} : \mathtt{u}}$$

More typing rules

$$\frac{\Gamma, \texttt{x:t} \vdash \texttt{e:u}}{\Gamma \vdash \texttt{fun x -> e : t -> u}}$$

$$\frac{\Gamma \vdash \texttt{e1:t1} \qquad \Gamma \vdash \texttt{e2:t2}}{\Gamma \vdash \texttt{(e1,e2) : t1*t2}}$$

More typing rules

$$\frac{\Gamma, \texttt{x:t} \vdash \texttt{e:u}}{\Gamma \vdash \texttt{fun x -> e : t} \Rightarrow \texttt{u}} \Rightarrow \text{intro}$$

$$\frac{\Gamma \vdash \texttt{e1:t1} \qquad \Gamma \vdash \texttt{e2:t2}}{\Gamma \vdash \texttt{(e1,e2) : t1}\wedge\texttt{t2}} \wedge \text{intro}$$

More computing with evidence

$$\frac{\Gamma, \texttt{x:t} \vdash \texttt{e:u}}{\Gamma \vdash \texttt{fun x -> e : t -> u}}$$

given evidence `e` for `u` predicated on evidence `x` for `t`, produce an evidence transformer

$$\frac{\Gamma \vdash \texttt{e1:t1} \qquad \Gamma \vdash \texttt{e2:t2}}{\Gamma \vdash \texttt{(e1,e2) : t1*t2}}$$

given evidence `ei` for `ti`, produce combined evidence for both

Even more typing rules

$$\frac{\Gamma \vdash \texttt{e : t1*t2}}{\Gamma \vdash \texttt{fst e : t1}}$$

$$\frac{\Gamma \vdash \texttt{e : t1*t2}}{\Gamma \vdash \texttt{snd e : t2}}$$

Even more typing rules

$$\frac{\Gamma \vdash \texttt{e : t1}\wedge\texttt{t2}}{\Gamma \vdash \texttt{fst e : t1}}\ \wedge\text{ elim 1}$$

$$\frac{\Gamma \vdash \texttt{e : t1}\wedge\texttt{t2}}{\Gamma \vdash \texttt{snd e : t2}}\ \wedge\text{ elim 2}$$

Even more computing with evidence

$$\frac{\Gamma \vdash \texttt{e} : \texttt{t1*t2}}{\Gamma \vdash \texttt{fst e} : \texttt{t1}}$$

$$\frac{\Gamma \vdash \texttt{e} : \texttt{t1*t2}}{\Gamma \vdash \texttt{snd e} : \texttt{t2}}$$

given evidence `e` for both `ti`, 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 -> ('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:

- `(fun z -> (snd z, fst z)) (y,x)`
- `(snd (y,x), 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

Programming	Logic
Types	Propositions
Programs	Proofs
Evaluation	Simplification

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`

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

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

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`

Types and propositions

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

Type	Proposition
Type variable `'a`	Atomic proposition A
Function type `->`	Implication ⇒
Product type `*`	Conjunction ∧
`unit`	True
Tagged union	Disjunction ∨
Type with no values	False
(syntactic sugar)	Negation ¬