Proofs are Programs

Prof. Clarkson
Fall 2016

Today’s music: Two Sides to Every Story by Dyan Cannon and Willie Nelson
Review

Currently in 3110: Advanced topics
• Futures
• Monads

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

ACT I
Three innocent functions

```ocaml
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 formulas

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 formulas

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 formulas

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 \( x \ y \ z = x \ z \ (y \ z)\):

\[( A \Rightarrow (B \Rightarrow C)) \Rightarrow (( A \Rightarrow B) \Rightarrow (A \Rightarrow C))\]

Do you recognize these formulas?
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/formulas

```ocaml
define apply f x = f x : ( A \rightarrow B) \rightarrow A \rightarrow B
define const x = fun _ -> x : A \rightarrow (B \rightarrow A)
define 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 formulas

Logical formulas (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>
</tbody>
</table>
Types and formulas

Logical formulas (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>
</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 \text{fst} \ (a,b) = a : \ (A \land B) \Rightarrow A
let \text{snd} \ (a,b) = b : \ (A \land B) \Rightarrow B
let \text{pair} \ a \ b = (a,b) : \ A \Rightarrow (B \Rightarrow (A \land B))
let \text{tt} = () : \ true
Logical formulas (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 <code>a</code></td>
<td>Atomic proposition A</td>
</tr>
<tr>
<td>Function type <code>-&gt;</code></td>
<td>Implication ⇒</td>
</tr>
<tr>
<td>Product type <code>*</code></td>
<td>Conjunction ∧</td>
</tr>
<tr>
<td>unit</td>
<td>True</td>
</tr>
<tr>
<td>??</td>
<td>Disjunction ∨</td>
</tr>
<tr>
<td>??</td>
<td>False</td>
</tr>
</tbody>
</table>
Disjunction

type (\texttt{'}a,'b\texttt{)} disj = \texttt{Left of 'a | Right of 'b}

\textbf{let} \ \texttt{left (x:'a)} = \texttt{Left x} \\
\hspace{1cm} : \texttt{'a -> ('a, 'b) disj}

\textbf{let} \ \texttt{right (y:'b)} = \texttt{Right y} \\
\hspace{1cm} : \texttt{'b -> ('a, 'b) disj}

\textbf{let} \ \texttt{case (lb:'a -> 'c) (rb:'b -> 'c)} = \texttt{function} \\
\hspace{1cm} | \texttt{Left x} \rightarrow \texttt{lb x} \\
\hspace{2cm} | \texttt{Right y} \rightarrow \texttt{rb y} \\
\hspace{1cm} : \texttt{('a -> 'c) -> ('b -> 'c) -> ('a, 'b) disj -> 'c}

Read \texttt{"('a,'b) disj"} as \texttt{"A \lor B"}
Disjunction

type ('a, 'b) disj = 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 case (lb:'a -> 'c) (rb:'b -> 'c) = function
| Left x -> lb x
| Right y -> rb y
: (A ⇒ C) ⇒ (B ⇒ C) ⇒ (A ∨ B) ⇒ C
False

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

Creating a value of type void isn't possible!

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

```ocaml
let absurd (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 absurd (f:void) : 'b = f.nope : false ⇒ B
Negation

• Syntactic sugar: define $\neg A$ as $A \Rightarrow false$
• As a type, that would be 'a -> void
## Types and formulas

Logical formulas (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>
<tr>
<td>Type with no values</td>
<td>False</td>
</tr>
<tr>
<td>(syntactic sugar)</td>
<td>Negation ¬</td>
</tr>
</tbody>
</table>
Programs = Proofs

ACT II
Innocent typing rule

• Recall typing contexts and judgements [L15]
  – Typing context $T$ is a map from variable names to types
  – Typing judgement $T \vdash e : t$ 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 $T \vdash e_1 : t \to u$
and $T \vdash e_2 : t$
then $T \vdash e_1 \ e_2 : u$
Innocent typing rule

\[
\begin{align*}
\text{if} & \quad T \vdash e_1 : t \rightarrow u \\
\text{and} & \quad T \vdash e_2 : t \\
\text{then} & \quad T \vdash e_1 \ e_2 : u
\end{align*}
\]
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

\[
A \Rightarrow B \\
A \\
\hline
B
\]
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)
    • what you probably saw in CS 2800
  – Gentzen:
    • lots of inference rules (a couple for each operator)
    • few axioms
    • what I need to show you now
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
Judgements

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

- 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 $\Rightarrow$ and $\land$

$$
\Gamma, A \vdash B \\
\hline
\Gamma \vdash A \Rightarrow B
$$

$\Rightarrow$ intro

$$
\Gamma \vdash A \Rightarrow B \\
\Gamma \vdash A
$$

$\Rightarrow$ elim

$$
\Gamma \vdash B
$$

$$
\Gamma \vdash A \\
\Gamma \vdash B
$$

$\land$ intro

$$
\Gamma \vdash A \land B
$$

$\land$ elim 1

$$
\Gamma \vdash A
$$

$$
\Gamma \vdash A \land B
$$

$\land$ elim 2

$$
\Gamma \vdash B
$$
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 \( T \vdash e_1 : t \to u \)
and \( T \vdash e_2 : t \)
then \( T \vdash e_1 \ e_2 : u \)
Innocent typing rule

\[
\text{if } T \vdash e_1 : t \rightarrow u \\
\text{and } T \vdash e_2 : t \\
\text{then } T \vdash e_1 \ e_2 \ : \ u
\]

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


goes to 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 not just a program...
• It's a proof of $u$, in that it provides evidence for $u$

\[ T \vdash e_1 : t \rightarrow u \quad T \vdash e_2 : t \]

\[ \begin{array}{c}
T \vdash e_1 \ e_2 : u
\end{array} \]
More typing rules

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

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

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

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

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

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

\[ \begin{align*}
\Gamma \vdash e_1 : t_1 & \quad \Gamma \vdash e_2 : t_2 \\
\hline
\Gamma \vdash (e_1, e_2) : t_1 \land t_2 & \quad \land \text{intro}
\end{align*} \]
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

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

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

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

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

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

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

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

\[
\Gamma \vdash \text{snd } e : t_2
\]

\[
\land \text{ elim 1}
\]

\[
\land \text{ elim 2}
\]
Even more computing with evidence

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

\[
\begin{align*}
\Gamma \vdash e &: t_1 \times t_2 \\
\hline
\Gamma \vdash \text{snd } e &: t_2 \\
\end{align*}
\]

given evidence \(e\) for both \(t_1\), project out the evidence for one of them
A program that transforms evidence

\[ \begin{align*}
\text{assump} & \quad z : 'a*'b \vdash z : 'a*'b \\
\text{∧ elim 2} & \quad z : 'a*'b \vdash \text{snd } z : 'b \\
\text{assump} & \quad z : 'a*'b \vdash z : 'a*'b \\
\text{∧ elim 1} & \quad z : 'a*'b \vdash \text{fst } z : 'a \\
\text{∧ intro} & \quad z : 'a*'b \vdash (\text{snd } z, \text{fst } z) : 'b*'a \\
\text{⇒ intro} & \quad \vdash \text{fun } z \to (\text{snd } z, \text{fst } z) : 'a*'b \to 'b*'a
\end{align*} \]
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 vice versa
Evaluation = Simplification

ACT III
Many proofs/programs

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

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

Programs:
• \( \text{fun } x \rightarrow \text{fun } y \rightarrow \)
  \( (\text{fun } z \rightarrow (\text{snd } z, \text{fst } z)) \) \( y,x \)
• \( \text{fun } x \rightarrow \text{fun } y \rightarrow (\text{snd } (y,x), \text{fst } (y,x)) \)
• \( \text{fun } x \rightarrow \text{fun } y \rightarrow (x,y) \)

Proofs:
• ...too big for slides
• but perhaps you can imagine that proofs of well-typed-ness of each program
  are progressively simpler
Many proofs/programs

Key part (body) of each 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 gets simpler

Taking an evaluation step corresponds to simplifying
the proof
CONCLUSION
These are all the same ideas

<table>
<thead>
<tr>
<th>Programming</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>Formulas</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

• Happy Thanksgiving Break!

This is fundamental.

THIS IS 3110