CS 3110

Lecture 19: Logic

To Truth through Proof

Prof. Clarkson

Fall 2014

Today's music: Theme from *Sherlock*

Review

Current topic:

- How to reason about correctness of code
- Last week: informal arguments

Today:

- Logic
- Necessary step on our way to having machine-checked proofs of correctness

Question #1

What is your background in logic?

A. I've never studied any formal logic AFAIK.

B. I saw a little bit in CS 2800.

C. I've taken a CS logic class.

D. I've taken a math logic class.

E. I've taken a philosophy logic class.

A biased history of logic

- Originated in philosophy
- Mathematicians became interested in late 1800s and early 1900s
  - goal: formalize mathematical reasoning
  - **impossible**: Kurt Gödel
- Computer scientists found many applications in the late 20th century
  - AI: formalize reasoning of robots, agents
  - Theorem proving: verify mathematical theorems, even discover new theorems
  - Verification: prove correctness of programs!

A biased perspective on logic

- A *logic* is a programming language for expressing reasoning about evidence
  - Like how OCaml is a programming language for expressing computation on data (ints, bools, etc.)
  - Data and evidence are analogous
  - Computation and reasoning are analagous
- Like any PL, a logic has
  - **syntax**
  - **dynamic semantics** (evaluation rules) *--*omitted here
  - **static semantics** (type checking)

A logic: IPC

Syntax:

```
f ::= P | f1 /\ f2 | f1 \/ f2
        | f1 => f2 | ~f
```

- **`f`** is a meta-variable for formulae
- **`P`** is a meta-variable for propositions
  - We'll use any capital letter for propositions
  - except: **`true`** and **`false`** are also propositions

A logic: IPC

Syntax:

```
f ::= P | f1 /\ f2 | f1 \/ f2
        | f1 => f2 | ~f
```

- **/** is logical and (aka *conjunction*)
- **\\/** is logical or (aka *disjunction*)
- **=>** is logical implication
- **~** is logical negation
  - actually syntactic sugar: **~f** means **f => false**

A logic: IPC

Syntax:

```
f ::= P | f1 /\ f2 | f1 \/ f2
        | f1 => f2 | ~f
```

- Note on notation:
  - Slides use an ASCII syntax
  - Online notes use nicer math symbols
  - Either is fine, but be consistent
- IPC= Intuitionistic Propositional Calculus

Formal syntax

- Abstracts from ambiguities and details of natural language
- Examples:
  - *Mammals have hair. Monkeys have hair. So monkeys are mammals.*
  - *Mammals have hair. Teddybears have hair. So teddybears are mammals.*
  - `((M => H) /\ (X => H)) => (X => M)`
  - All are flawed reasoning!
    - (Want a way to distinguish flawed reasoning from correct reasoning...)
- A logic is a precise way to express structure of reasoning
- Just like a PL is a precise way to express structure of computation

Parts of syntax

- *Connectives*
  - and /\, or \/, implies **=>**, not **~**
  - like binary operators in a PL
  - create larger formulae (expressions) out of smaller
- *Propositions*
  - the basic "atoms" being reasoned about
  - like built-in data types (int, bool) in a PL
  - the simplest kind of formulae (expressions)

Formalization of an argument

- If there is a snowstorm, then the roads will be closed.
- The roads are open.
- So there can't be a snowstorm.

Formalization of an argument

- If there is a snowstorm, then the roads will be closed. $S \Rightarrow C$
- The roads are open.
- So there can't be a snowstorm.

Formalization of an argument

- If there is a snowstorm, then the roads will be closed. `S=>C`
- The roads are open. `O`
- So there can't be a snowstorm.

Formalization of an argument

- If there is a snowstorm, then the roads will be closed. `S=>C`
- The roads are open. `O`
- So there can't be a snowstorm. `~S`

Formalization of an argument

- If there is a snowstorm, then the roads will be closed. `S=>C`
- The roads are open. `O`
- So there can't be a snowstorm. `~S`
- *Implicit:* A road is either open or closed. `O=>~C /\ C=>~O`

Formalization of an argument

- If there is a snowstorm, then the roads will be closed. `S=>C`
- The roads are open. `O`
- So there can't be a snowstorm. `~S`
- *Implicit:* A road is either open or closed.
  `O=>~C /\ C=>~O`
- Combining them all:

```
((S=>C) /\ O /\ ((O=>~C)
   /\ (C=>~O))) => ~S
```

Question #2

Which subformula does not appear in formalization?

If there is a snowstorm then the roads will be closed. There is no snowstorm. So the roads must be open.

A. S=>C
B. ~S
C. C=>S
D. O
E. O=>~C

Question #2

Which subformula does not appear in formalization?

If there is a snowstorm then the roads will be closed. There is no snowstorm. So the roads must be open.

A. S=>C
B. ~S
C. C=>S
D. O
E. O=>~C

Valid vs. invalid arguments

- How to separate them?
- What constitutes correct reasoning?
- Analogy: how did we distinguish "valid" from "invalid" programs?
  - Static semantics = type system
- So let's build a "type system" for valid arguments
  - Usually called a "proof system" or "deductive system"

Proof system for IPC

- Only one type: **`provable`**
  - e.g., **`(A /\ B) => A : provable`**
  - e.g., **`A => (A /\ B)`** is not provable so can't be given a type
- No reason to keep writing "**`f : provable`**" everywhere
  - the colon and word "provable" are too verbose
- Instead, write **`|- f`**
  - pronounced as "provable **`f`**" or "**`f`** is provable"
  - or "derivable" instead of "provable"

Proof system for IPC

- We'll give *proof rules* for each syntactic form in IPC
- Just like we gave *type-checking rules* for each syntactic form in OCaml
  - `5 : int`
  - `fun x -> e : ta-> tb` if `e:tb` under assumption `x:ta`

Proof system for IPC

- We'll give *introduction* and *elimination* rules for each form
- Just like we gave rules for *building* and *accessing* pieces of data in OCaml
  - `(e1,e2) : a*b if e1:a and e2:b`
  - `fst e : a if e : a*b`

All rules will be based on *evidence* for each form...

Evidence for `/\`

Q: What constitutes evidence for **`f1 /\ f2`**?

A: Evidence for both **`f1`** and **`f2`**, individually

so evidence for **`f1/\f2`** is really a **pair** of the evidence for **`f1`** and the evidence for **`f2`**...

Proof rules for `/\`

- if `|- f1` and `|- f2` then `|- f1 /\ f2`
  - **introduction rule:** shows how to build/introduce a formula out of smaller pieces
  - **intuition:** if you have evidence for `f1` and evidence for `f2`, then you can combine those pieces of evidence to get evidence `for f1 /\ f2`

Proof rules for `/\`

- if `|- f1 /\ f2` then `|- f1`
- if `|- f1 /\ f2` then `|- f2`
  - **elimination rules:** show how to access smaller formulae out of larger, i.e., eliminate parts of formulae
  - **intuition:** if you have evidence for `f1 /\ f2`, then you can break apart that to get evidence for `f1` individually, likewise for `f2`
  - **further intuition:** these rules are really just `fst` and `snd`

Evidence for =>

Q: What constitutes evidence for **`f1=>f2`**?

A: A way to transform evidence for **`f1`** into evidence for **`f2`**.

So evidence for **`f1=>f2`** is really a **function** that transforms evidence for **`f1`** into evidence for **`f2`**...

Proof rules for `=>`

- if `|- f` and `|- f => g` then `|- g`
  - traditionally called *modus ponens*: "way that affirms"
  - elimination rule
  - **intuition:** if you have evidence for `f`, and you have a way of transforming evidence for `f` into evidence for `g`, then you have evidence for `g`
  - **further intuition:** this rule is really just function application

Proof rules for =>

- if under the assumption `|- f` we can conclude `|- g`, then `|- f => g`
  - introduction rule
  - **intuition:** the way you reached that conclusion must be a way of transforming evidence for `f` into evidence for `g`, so you have evidence for `f=>g`
  - further intuition: this rule is really just anonymous function definition
  - *hypothetical reasoning*: "if I assume X, then I can conclude Y."

Notation for assumptions

- `f |- g` means "under the assumption that **f** is provable, it holds that **g** is provable"

- So instead of:

  if under the assumption `|- f` we can conclude `|- g`,

  then `|- f => g`

  we can write:

  if `f |- g` then `|- f => g`

- Generalize to entire set of assumptions: `F |- g` means "under the assumption that all formulas in set **F** are provable, it holds that **g** is provable"
  - Write comma instead of set union: `F, f` means **F** ∪ {**f**}

Revised proof rules

Adding assumptions to all rules so far:

- if `F |- f1` and `F |- f2` then `F |- f1 /\ f2`
- if `F |- f1 /\ f2` then `F |- f1`
- if `F |- f1 /\ f2` then `F |- f2`
- if `F |- f` and `F |- f => g` then `F |- g`
- if `F, f |- g` then `F |- f => g`

Proof rules for assumptions

- `f |- f`
  - Intuition: if you have assumed that you have evidence for `f`, then you can proceed as though you have evidence for `f`
  - This rule is an *axiom*: it has no premises
- if `F |- f` then `F, g |- f`
  - Intuition: if assuming `F` is enough to derive evidence for `f`, then additionally assuming `g` makes no difference
  - This rule is called *weakening*: assuming more weakens the claim

A proof

Let's show `|-  (A => (B => A))`

<table>
<tr><th>Rule name</th><th>Rule</th></tr>
<tr><td>/\ intro</td><td>if <code>F |- f1</code> and <code>F |- f2</code> then <code>F |- f1 /\ f2</code></td></tr>
<tr><td>/\ elim L</td><td>if <code>F |- f1 /\ f2</code> then <code>F |- f1</code></td></tr>
<tr><td>/\ elim R</td><td>if <code>F |- f1 /\ f2</code> then <code>F |- f2</code></td></tr>
<tr><td>=> elim</td><td>if <code>F |- f</code> and <code>F |- f => g</code> then <code>F |- g</code></td></tr>
<tr><td>=> intro</td><td>if <code>F, f |- g</code> then <code>F |- f => g</code></td></tr>
<tr><td>assump</td><td><code>f |- f</code></td></tr>
<tr><td>weak</td><td>if <code>F |- f</code> then <code>F, g |- f</code></td></tr>
</table>

A proof

Let's show `|- (A => (B => A))`

1. `A |- A` by assumption rule
2. `A,B |- A` by (1) and weakening rule
3. `A |- B => A` by (2) and `=>` introduction rule
4. `|- A => (B => A)` by (3) and `=>` introduction rule

Proof structure

- Each step numbered
- Each step derives one new formula from previous step(s) and from named rule
- At each rule application, all the *premises* of a rule must already have been derived. Get to add *conclusion* of rule as new numbered step.
- Final step is the formula we want to prove, with no assumptions

A graphical notation: proof trees

```
─────────────────   assump.
      A |- A
─────────────────   weak.
     A,B |- A
─────────────────   => intro.
   A |- B => A
─────────────────   => intro.
|- (A => (B => A))
```

Proof structure

- Goal formula is at root of tree (bottom)
- Each node in tree is a formula
  - i.e., a numbered step from linear form
- Each edge in tree is labeled by rule name
  - i.e., a justification from linear form
- If rule has no premises, there's an "empty" node at top
  - i.e., an axiom

That proof as an OCaml program

```
let t (a:'a) (b:'b) : 'a = a
```

How to think about this program:

`t` is a function that takes in evidence for `'a`, evidence for `'b`, and returns the evidence for `'a`

What is its type?

```
'a -> ('b -> 'a)
```

What is the formula we proved?

```
A => (B => A)
```

Programs and Proofs

- We were able to write a program whose type is the very formula we were trying to prove

- That program is an *evidence transformer*: it manipulates input evidence to construct output evidence of the right type

- This correspondence between
  - programs and proofs
  - types and formulae

  goes very, very deep.

- Known as the *Curry-Howard isomorphism*

Another proof

Let's show `|- A => (B => (A/\B))`.

Rule name	Rule
/\ intro	if `F |- f1` and `F |- f2` then `F |- f1 /\ f2`
/\ elim L	if `F |- f1 /\ f2` then `F |- f1`
/\ elim R	if `F |- f1 /\ f2` then `F |- f2`
=> elim	if `F |- f` and `F |- f => g` then `F |- g`
=> intro	if `F, f |- g` then `F |- f => g`
assump	`f |- f`
weak	if `F |- f` then `F, g |- f`

Another proof: linear form

Let's show `|- A => (B => (A/\B))`.

1. `A |- A` by assumption rule
2. `A,B |- A` by weakening rule
3. `B |- B` by assumption rule
4. `A,B |- B` by weakening rule
5. `A,B |- A/\B` by (2), (4), and `/\` introduction rule
6. `A |- B => (A/\B)` by (5) and `=>` introduction rule
7. `|- A => (B => (A/\B))` by (6) and `=>` introduction rule

Another proof: tree form

assump

```
A |- A
```

assump

```
B |- B
```

weak

```
A,B |- A
```

weak

```
A,B |- B
```

/\ intro

```
A,B |- A /\ B
```

=> intro

```
A |- B => (A/\B)
```

=> intro

```
|- A => (B => (A/\B))
```

As an OCaml program

```
let pair (a:'a) (b:'b) : ('a*'b)
  = (a,b)
```

How to think about this program:

`pair` is a function that takes in evidence for `'a`, evidence for `'b`, and returns the pair containing both pieces of evidence

What is its type?

```
'a -> ('b -> ('a * 'b))
```

What is the formula we proved?

```
A => (B => (A /\ B))
```

Please hold still for 1 more minute

WRAP-UP FOR TODAY

Upcoming events

- **PS5 checkins this week**
- Clarkson office hour today cancelled; moved to tomorrow
- Thursday: Guest lecture by Yaron Minsky (Cornell PhD) from Jane Street on "OCaml in the Real World"

This is logical.

THIS IS 3110