Computing Foundations

Release 1.2.2

Kevin Sullivan

Feb 09, 2023

CONTENTS

CHAPTER

ONE

CS6501 SPRING 2023

1.1 To Think Abstractly

To advance science and engineering, we need languages in which we can describe and reason about complex worlds. By a *world* we mean a collection of objects of interest and their structures, properties, behaviors, relations to each other.

1.1.1 Abstract Mathematics

Mathematics and formal logics provide such languages. They give us the intellectual tools we need to think abstractly and yet with great precision about rich and complex systems: to represent, reason about, and ultimately design complex worlds that would otherwise remain beyond our grasps.

As an example, the abstract mathematics of linear and affine spaces give us languages for describing, reasoning about, and designing systems that work in the *classical* physical world we experience every day. The abstract mathematics of tensor fields on topological manifolds are essential for describing, reasoning about, and designing interventions in the *quantum* world of particle physics.

By the term, *abstract*, we mean that descriptions in such languages represent relevant phenomena precisely, concisely, and without any unnecessary complexity or inessential detail.

As an example, a physicist might represent two accelerations applied to a drone in a three-dimensional geometric space in abstract, coordinate-free terms, by writing this: *let* a_1 *and* a_2 *be accelerations of the drone.* This formulation is abstract insofar as no coordinates are given for these vectors. The assignment of coordinates to *physical* quantities is usually arbitrary and unnecessary to express. A physicist might, for example, represent *the sum of these accelerations* simply as $a_1 + a_2$. This expression has an absolutely precise physical meaning even though it's abstract.

A programmer, by contrast, would typically jump to a choice of some coordinate system and would then represent the two physical quantities in the concrete (*parametric*) terms of tuples of floating point numbers; with the summation of the physical accelerations represented by element-wise floating point addition of the corresponding coordinate tuples.

1.1.2 Costs of Concreteness

This ubiquitous approach to programming physical computations is problematical in multiple dimensions. First, as mentioned, it substitues concrete representations for abstract, adding inessesntial complexity to models and computations. Second, it generally strips away crucial mathematical properties of the abstract representations of objects of interest, making it impossible to check programs for consistency with such mathematics.

For example, in the *tf* and *tf2* affine space libraries of the Robot Operating System (ROS) platform for the programming of terrestrial robots, points and vectors are represented in the concrete terms of coordinate tuples relative to arbitrary coordinate frames. But it gets worse: points and vectors in this framework are aliases for the same concrete type: a type of floating-point tuples.

This means, among other things, that one can add points to points in *tf* without receiving any type errors from the programming language system, even though addition of points to points makes no physical sense and is inconsistent with the abstract mathematics of the domain. In an affine space, there is no operation for adding points to point.

The nearly exclusive use of concrete representations in most everyday programming complicates software design and reasoning by requiring the manipulation of often complex, inessential details. And because so much of the structure of the mathematics of the domain is *forgotten* in the programming code, it also becomes impossible for the programming system to check for what we might call *full physical type consistency*.

Programming code thus generally ends up deeply disconnected from the abstract mathematics of the physics of the domain that it's meant to represent, manipulate, implement, and free to carry out inconsistent operations. As one example, programmers often struggle mightily using different frames of reference in a consistent manner, e.g., by adding vectors represented by numerical tuples but with coordinates expressed in different frames of reference.

A special case, by the way, is operating incorrectly on values because they are expressed in different units, such a meters and feet. We can understand 1 meter and 1 foot as each being a basis vector for the *same* one-dimensional physical space. Clearly adding numerical values of 2 (feet) and 3 (meters) to produce 5 of *something,* produces a meaningless result.

1.1.3 A Path Forward

So why haven't we already deeply connected concrete to code the abstract mathematics of the domain in which it's meant to operate? Perhaps the most fundamental reason is that math has up until recently been a quasi-formal, paper-and-pencil exercise, making it, hard even impossible, to connect code to such mathematics.

Now, with rapidly developing work by a still small set of ressearchers in mathematics, the complete formalization and mechanization of advanced abstract mathematics is becoming a reality. As and example Massot and his colleagues in 2022 managed to formalize not only a statement, but a proof, of the local h-principle for first-order, open, ample partial differential relations, with the possibility of eversion of the 3-sphere as a corollary.

Their work showed that their approach to formalizing mathematics is no longer useful mostly just for abstract algebra, but that it also now promises advances in abstract geometry (e.g., on manifolds), which is at the very heart of not only terrestrial robotics but also modern physics and perhaps even areas such areas as deep learning.

1.1.4 The Vision

The insight driving this class is that this kind of work from the mathematics community is now making it possible to develop *computable abstract mathematics* for purposes of software engineering. Promising application domains for such work clearly include divere cyber-physical systems and might be relevant to deep learning as well, with its basic assumption that real-world data have geometric properties, such as lying on high-dimensional manifolds.

What we seek are ways to enable use of the abstract mathematical languages of such domains, coupled with concrete representations to support computation, with *full physical type checking* of the physical mathematics; *foundational proofs* of correctness of the mathematics; and explicit link to concrete (often coordinate-based) representations that are necessary in practical implementations.

1.1.5 This Class

The purpose of this class is to introduce computer science students to the ideas necessary to pursue both research and development activities based on these ideas. We will use the preferred tool of the community of mathematicians pushing the formalization of mathematics, namely the Lean Proof Assistant, developed by Leo DeMoura at Microsoft research, and the ever growing *mathlib* library of formalized mathematics.

1.2 Welcome. Overview.

This is a special topics course in software engineering. The idea that we will explore is that we can now import ongoing advances in the formalization of abstract mathematics (in type theory and to a significant extent around the Lean prover and its mathematics libraries) as new foundations for engineering software programs for systems that inhabit domains that have such abstract mathematical underpinnings. Such domains include physics, and thus also a broad range of cyber-physical systems.

Along the way, you'll learn formal logic and proof construction, foundations of programming languages, functional programming, and more. By the end you will know how to use some cutting-edge tools, type theory and constructive logic proof assistants, to formalize the abstract mathematics of important application domains.

1.2.1 What does *abstract* mean here?

Here, the adjective, abstract, means *being coordinate-free*. For the opposite of *abstract*, we'll use *parametric*. The idea is that a mathematical object, such as a vector, can be understood simply as such, with no reference to coordinates; or that same abstract vector can be represented concretely/parametrically as a struture of parameter values expressed relative to some given frame of reference.

1.2.2 Why *abstract* mathematics?

A premise of this class is that domain experts (e.g., in the physics of terrestrial robotics, or of elemenary particles) speak, model, analyze, and understand the operation of systems in the abstract mathematical language of the domain, and very often not in terms of ultimately arbitrarily selected frames of reference .

This is clear in physics where complex mathematical structures such as tensor fields on topological manifolds are essential for precise definitions of phenomena in particle physics.

Domain-specific abstract mathematics formalized in type theory is what we see as an important language of the domain, to serve as a basis for programming with static checking of abstractions, and with parametric representations carried along as necessary.

1.2.3 What is the point?

A remarkable feature of constructive logics as hosted by many proof assistants is that they can compute. Computation is now wholly integral to practical logical reasoning if only because some proofs require systematic case analysis over very large numbers of cases, given as the outputs of other computations.

This is is relevant because it suggests that fully formalizing the fully developed mathematical language of the given domain, we will be well on our way to having reference specifications and with computable implementations.

Going even further, the most recent version of Lean is intended as an efficient general-purpose programming language as well as a proof assistant, compiling to C, and with workable language interoperability interfaces.

In the future we expect to be able to statically type check and foundationally verify "code" written in the abstract mathematics of the domain and with very little custom coding needed also to have corresponding verified implementations, even if only used as test oracles for production code.

This is the of abstract specifictions from which concrete implementations are derived. I will not say refined because in practice most derivations are not refinements but rather toyish models of semantically rich and complex computations in the source domain.

1.2.4 This class

The first major part of this class will teach you the fundamentals of programming and reasoning in Lean 3. We will mainly use Lean 3, nothwithstaning that Lean 4 is garnering real attention and effort. Student might wish to explore Lean 4 as an optional class activity.

The second part of the class will focus on how to formalized abstract mathematical structures in Lean, and how we might such capabilities to these advances in formalizing mathematics to help meet the need for statically type checked specifications in the abstract language of the domain, with corresponding parametric representations carried along for computational possibly and oher purposes.

CHAPTER

TWO

FORMAL LANGUAGES

2.1 Balanced Parentheses

As a warmup, and to put some basic concepts into play, we'll begin by specifying the syntax and semantics of a simple formal language: the language of strings of balanced parentheses. Before we do that, we'll better explain wht it all menas. So let's get started.

2.1.1 Formal languages

The syntax of a *formal language* defines a (possibly infinite) set of strings. Strings are sequences of symbols from some *alphabet* of symbols. The formal language of basic algebra, for example, includes strings such as x, y, and $x + y$, but not $x\ y$. Propositional logic includes X, Y, and $X \wedge Y$ but not $X\ Y$.

As another example, which shortly we will specify formally, consider the language of all strings of balanced parentheses. The language includes the empty string, *(), (()), ((()))*, etc. It does not include any unbalanced strings, such as *(*, *)*, *((*, or *(()*.

Each string in our language will be of some finite nesting depth (and thus length) but the number of strings in the language is infinite. There is one such string for each possible nesting depth of such a string. That is, for any natural number, n, there is a such a string with nesting depth n.

We clearly can't specify the set of strings by exhaustively enumerating them expliciitly. There are too many for that. Rather, we need a concise, precise, finite, and manageable way to specify the set of all such strings. We will do that by defining a small set of *basic rules for building* strings of this kind (we'll call them *constructors*), sufficient for constructing all and only the strings in the language.

We can specify the balanced parentheses language with just two rules. First, the empty string, $\emptyset$, is in our language. Second, if b is *any* string in the language, then so is *(b)*. That is all we need to construct a string of any nesting depth.

The empty string (nesting depth 0) is just empty, while we can construct a string of any positive depth by applying the first rule once, giving us the *base string*, $\emptyset$, any by applying the second rule iteratively, first to $\emptyset$, as many times as needed to construct a string of balanced parentheses as big as desired.

A key characteristic of this definition is that it's properly *inductive*. That is, it provides ways to build larger values of a given type (balanced parenthesis strings) from smaller values *of the same type*. The complete set of strings that these rules *generate* (by any finite number of applications thereof) is exactly the set of strings, the *formal language* that we set out to specify.

2.1.2 Paper & Pencil Syntax

What we've basically done in this case is to specify the set of strings in our language with a *grammar* or *syntax definition*. Such grammars are often expressed, especially in the programming worls, using so-called *Backus-Naur Form (BNF)*.

Backus first used BNF notation to define the syntax of the Algol 60 programming language. BNF is basically a notation for specifying what the linguist, Noam Chomsky, called *context-free grammars*.

Here's a grammer in BNF for our language of balanced parentheses. We can say that the BNF grammar defines the syntax, or permitted forms, of strings in our language. Be sure you see how this grammar allow larger expressions to be build from smaller ones of the same kind (here *expression*).

> expression ::= | ∅ | (expression)

This definition says that an expression (string) in our language is either the empty string or it's an expression within a pair of parentheses. That's it. That's all it takes.

2.1.3 Formal Syntax

Now we give an equivalent but *completely formal* definition of this language in Lean. The key idea is that we will define a new *data type* the values of which are all and only terms representing strings in our language.

We'll start by defining separate data types (each with just one value) to represent left and right parentheses, respectively. The names of the types are *lparen* and *rparen*. Each has a single value that we will call *mk*. We can use qualified names to distinguish these values: *lparen.mk* and *rparen.mk*.

```
inductive lparen
| mk

inductive rparen
| mk


/-
Here are some examples where we use these values.
In the first case we use *def* in Lean to bind a
value (representing a left parenthesis) to the
identifier, *a_lef_paren.* In the second case we
used *example* in Lean as a way simply to exhibit
a value of a particular type (here representing a
right parenthesis).
-/

def a_left_paren : lparen := lparen.mk
example          : rparen := rparen.mk
```

Now we're set to specify the set of all and only balanced parenthesis strings. We give an inductive definition with the two rules (*constructors*). First, the empty string (which for now we call mk_empty to stand for ∅), is in the set of balanced strings. Second, if *b* is any balanced string, then the term *mk_nonempty l b r* is also (that is also represents) a balanced string, namely *(b)*.

```
inductive bal
| mk_empty
| mk_nonempty (l: lparen) (b : bal) (r : rparen)
```

The only thing that a constructor does in such a definition is to package up its arguments (if any) into a new term with the constructor name as a first element (a label, if you will). The type system of Lean will now recognize any term that can be built using the available constructors as being of type bal.

Here we illustrate the use of these constructors to build the first few balanced strings in our language. We Open the *bal* namespace so that we don't have to write *bal.* before each constructor name. These constructor names do not conflict with any existing definitions in the current (global) namespace. We don't open the lparen and rparen namespaces because then we'd have two (ambiguous) definitions of the identifier, mk, and we'd have to write *lparen.mk* or *rparen.mk* in any case to disambiguate them.

```
open bal

def b0 : bal :=        -- ∅
  mk_empty

def b1 : bal :=        -- (∅)
mk_nonempty            -- constructor
  lparen.mk            -- argument left parenthesis
  b0                   -- note: we could write mk_empty
  rparen.mk            -- argument right parenthesis

def b2 :=              -- ((∅))
mk_nonempty
  lparen.mk
  b1
  rparen.mk

def b3 :=
mk_nonempty
  lparen.mk
  (
    mk_nonempty
      lparen.mk
      (
        mk_nonempty
          lparen.mk
          mk_empty
          rparen.mk
      )
      rparen.mk
  )
  rparen.mk
```

You can confirm that the type of b1 is bal using the *check* command in Lean. The output of this command is visible if you hover your cursor over the blue underline (in VSCode), and in your Lean infoview. You can open and close the infoview window in VSCode by CMD/CTRL-SHIFT-ENTER.

```
#check b1
```

You can now use the *reduce* command in Lean to see that *b1* is bound to the term, *mk_nonempty lparen.mk mk_empty rparen.mk*. If you do the same for *b2* you will see its unfolded value, and the same goes for b3. Be sure to relate the results you get here back to the definitions of *b1, b2,* and *b3* above.

```
#reduce b1
#reduce b2
#reduce b3
```

From here we can build larger and larger strings in *bal*.

2.1.4 Inductive Datatype Definitions

There are three crucial properties of constructors of inductive data types in Lean that you should now understand. First, they are *disjoint*. Different constructors *never* produce the same value. Second, they are *injective*. A constructor applied to different argument values will always produce different terms. Finally, they are complete. The langauge they define contains *all* of the strings constructible by any finite number of applications of the defined constructors *and no other terms*. For example, our *bal* language doesn't contain any *error* or any other terms not constructible by the given constructors.

2.1.5 Semantics

The semantics of a formal language defines an association between some or all of the terms of a language and what each such term means, in some *semantic domain*. For example, we can associate each string in *bal* with the natural number that describes its nesting depth.

In this case, there is total function from terms of type *bal* to $\mathbb{N}$, so we can specify the semantics as a function in Lean. (All functions in Lean represent total functions in mathematics.)

Here is such a function defined using one of several notations available in Lean. We define the function, *sem* as taking a value of type *bal* as an argument and returning a value of type nat ($\mathbb{N}$, natural number, i.e., non-negative integer) as a result.

The function is defined by case analysis on the argument. If it is the empty string, mk_empty, the function returns 0. Otherwise (the only remaining possibility) is that the value to which *sem* is applied is of the form (mk_nonempty l b r) where *l* and *r* are values representing left and right parenthesis, and where *b* is some smaller string/value of type *bal*. In this case, the nesting depth of the argument is one more than the nesting depth of *b*, which we compute by applying *bal* recursively to *b*.

```
def sem : bal → ℕ
| mk_empty := 0
| (mk_nonempty l b r) := 1 + sem b

-- We can now run some tests to see that it works
#reduce sem b0
#reduce sem b1
#reduce sem b2
```

So there you have it. We've defined both a formal language and a semantics of this language using the logic of the Lean proof assistant. We defined an inductive data type the *terms* (values) of which represent all and only the strings in *bal*. We defined a total function that maps any term of this type to its corresponding length expressed as a natural number, which we take to be the *semantic meaning* of that string.

We now have all the machinery we need to formally define the syntax and semantics of more interesting and useful languages. We will now turn to the language of propositional logic as our next major example.

2.2 Simplified Propositional Logic

Our next step toward formalizing abstract mathematics for software engineering, we will specify the syntax and semantics of a simple but important mathatical language, namely *propositional logic*.

Propositional logic is isomorphic to (essentially the same thing as) Boolean algebra. You already know about Boolean algebra from writing conditions in if and loop commands in everyday programming languages such as Java.

Our first task will be to see how to formalize the syntax and semantics of this language in Lean.

2.2.1 Syntax

The set of expressions (strings) comprising the formal language of propositional logic is defined inductively. That is, some smallest expressions are first defined, with larger expressions then defined as being constructed, at least in part, from smaller ones *of the same type*.

The syntax of propositional logic comprises

- variables, e.g., x, y, y, theSkyIsBlue, etc
- a language of propositional expressions (propositions) * constant expressions, *true* and *false* * variable expressions, such as X, Y, Z, TheSkyIsBlue, each such expression having an associated variable * operator expressions, such as ¬X, X ∧ Y, and X ∨ Y

You can think of these operators as taking expressions as their arguments and returning longer expressions as their results.

To begin, we define a datatype the values of which will represent our the variables. We'll name the type *prop_var,* for propositional variable. For now we'll assume we're restricted to using at most three variables. We'll call them *x, y,* and *z* and will just list them as being the distinct constant values of the *prop_var* type.

```
inductive prop_var : Type
| x
| y
| z

open prop_var
```

Next we'll define the set of expressions in our language, which we'll call *prop_expr*, the language of expressions in propositional logic.

```
inductive prop_expr
| var_expr (v : prop_var)
| and_expr (e1 e2 : prop_expr)
| or_expr (e1 e2 : prop_expr)

open prop_expr
```

We can now form both variable and operator expressions! Let's start with some variable expressions.

```
def X : prop_expr := var_expr x
def Y : prop_expr := var_expr y
def Z : prop_expr := var_expr z
```

We can also define operator expressions, which build larger expressions out of smaller ones.

```
def XandY : prop_expr := and_expr X Y
def XandY_and_Z : prop_expr := and_expr XandY Z
```

2.2.2 Semantics

The semantics of propositional logic assigns a Boolean truth value to each expression in the language, but to do this, an additional piece of data is required: one that defines the Boolean meaning (truth value) of each *variable* referenced by any variable expression.

What for example is the meaning of the variable expression, *X*? It's impossible to say unless you know the meaning of the variable, *x*. If the meaning of *x* is true, then we define the meaning of *X* to be true, and likewise for the value, false.

We will use the word *interpretation* to refer to any assignment of Boolean truth values to all variables that can be referenced by any given variable expression. For example, we might define *x*, *y*, and *z* all to have true as their meanings. We can formalize this mapping from variables to truth values as a total function from terms of type *prop_var* to terms of type *bool* in Lean.

```
def all_true : prop_var → bool
| _ := tt   -- for any argument return true (tt in Lean)
```

Similarly here's an interpretation under which all variables are assigned the value, false.

```
def all_false : prop_var → bool
| _ := ff   -- for any argument return true (tt in Lean)
```

Now here's an interpretation under which x is assigned true, and the remaining variables (y and z) are assigned false.

```
def mixed : prop_var → bool
| prop_var.x := tt
| _ := ff

#reduce mixed z

def another_interpretation : prop_var → bool
| x := tt
| y := ff
| z := tt
```

Given one of these interpretations as additional data, we can now assign truth value semantic meanings to expressions such as XandY (and_expr X Y). We do this recursively. First we evaluate X to get its truth value (by applying a given interpretation function to the variable, x, that expression X "contains".

Recall that X is defined to be the term, var_expr x. We just need to *destructure* this term to get the *x* part of it. Remember that constructors simply package up their arguments into terms in which those arguments appear in order. Once we get at the variable, *x*, we just apply an interpretation function to it to get its corresponding Boolean value, and we take that as the meaning of the variable expression, *X*.

Ok, so what about the meaning of *(and_expr X Y)*? First we need to know the meanings of *X* and *Y*. Suppose they are true and false, respectively. Then we define the meaning of *(and_expr X Y)* as the Boolean *conjunction* of these truth values. In this case, that'd be *tt && ff,* which is *ff*.

Here then is a semantic evaluation function that implements these two notions: one in the case where the expression to be given a meaning is a variable expression, and one where it's an *and* expression.

```
def prop_eval : prop_expr → (prop_var → bool) → bool
| (var_expr v) i := i v
| (and_expr e1 e2) i := band (prop_eval e1 i) (prop_eval e2 i)
| (or_expr e1 e2) i := bor (prop_eval e1 i) (prop_eval e2 i)
```

Now we can find the meaning of *any* expression in our initial subset of the language of propositional logic. To be more precise, we'd say that we've specified an *abstract syntax* for our language. In our next unit, we'll see how to use Lean's

syntax extension capabilities to define a corresponding *concrete* syntax, one that'll let us write expressions in our language as if we were using paper and pencil methods and standard syntax for propositional logic.

```
#check all_true


#reduce prop_eval X all_true
#reduce prop_eval Y all_true
#reduce prop_eval Z all_true

#reduce prop_eval X all_false
#reduce prop_eval Y all_false
#reduce prop_eval Z all_false

#reduce prop_eval XandY all_true
#reduce prop_eval XandY all_false
#reduce prop_eval XandY mixed

#reduce prop_eval (and_expr (and_expr X Y) (or_expr X Z)) mixed
```

So we now have is a specification of the syntax and semantics of a subset of propositional logic. As an in-class exercise, let's add some new logical operators: for not, or, implies, bi-implication, and exclusive or.

2.3 Propositional Logic

In this chapter, we'll present a first version of a syntax and semantic specification for a full language of propositional logic. As in the last chapter, we start by defining the syntax, then we present the semantics.

2.3.1 Syntax

```
/-
Propositional logic has an infinite supply of variables.
We will represent each variable, then, as a term, var.mk
with a natural-number-valued argument. This type defines
an infinite set of terms of type *prop_var*, each *indexed*
by a natural number.
-/
inductive prop_var : Type
| mk (n : ℕ)

-- Abstract syntax
inductive prop_expr : Type
| pTrue : prop_expr
| pFalse : prop_expr
| pVar (v: prop_var)
| pNot (e : prop_expr)
| pAnd (e1 e2 : prop_expr)
| pOr (e1 e2 : prop_expr)
| pImp (e1 e2 : prop_expr)
| pIff (e1 e2 : prop_expr)
| pXor (e1 e2 : prop_expr)

open prop_expr
```

We can now *overload* some predefined operators in Lean having appropriate associativity and precedence properties to obtain a nice *concrete syntax* for our language. See also (https://github.com/leanprover/lean/blob/master/library/init/core.lean)

```
notation (name := var_mk) `[` v `]` :=  pVar v
notation (name := pAnd) e1 ∧ e2 :=  pAnd e1 e2
notation (name := pOr) e1 ∨ e2 :=  pOr e1 e2
notation (name := pNot) ¬e := pNot e
notation (name := pImp) e1 => e2 := pImp e1 e2
notation (name := pIff) e1 ↔ e2 := pIff e1 e2
notation (name := pXor) e1 ⊕ e2 := pXor e1 e2
```

Here, after giving nice names, X, Y, and Z, to the first three variables, we givesome examples of propositional logic expressions written using our new *concrete* syntax.

```
def X := [prop_var.mk 0]
def Y := [prop_var.mk 1]
def Z := [prop_var.mk 2]

def e1 := X ∧ Y
def e2 := X ∨ Y
def e3 := ¬ Z
def e4 := e1 => e2  -- avoid overloading →
def e5 := e1 ↔ e1
def e6 := X ⊕ ¬X
```

2.3.2 Semantics

```
-- Helper functions
def bimp : bool → bool → bool
| tt tt := tt
| tt ff := ff
| ff tt := tt
| ff ff := tt

def biff : bool → bool → bool
| tt tt := tt
| tt ff := ff
| ff tt := ff
| ff ff := tt

-- Operational semantics
def pEval : prop_expr → (prop_var → bool) → bool
| pTrue _ := tt
| pFalse _ := ff
| ([v]) i := i v
| (¬ e) i := bnot (pEval e i)
| (e1 ∧ e2) i := (pEval e1 i) && (pEval e2 i)
| (e1 ∨ e2) i := (pEval e1 i) || (pEval e2 i)
| (e1 => e2) i := bimp (pEval e1 i) (pEval e2 i)
| (e1 ↔ e2) i := biff (pEval e1 i) (pEval e2 i)
| (e1 ⊕ e2) i := xor (pEval e1 i) (pEval e2 i)
```

I'll fill in explanatory text later, but for now wanted to get you the *code*.

2.4 A Better Specification

In this chapter we present an improved specification of the syntax and semantics of propositional logic. As usual, we first present the syntax specification then the semantics.

2.4.1 Syntax

```
-- variables, still indexed by natural numbers
inductive prop_var : Type
| mk (n : ℕ)

-- examples
def v₀ := prop_var.mk 0
def v₁ := prop_var.mk 1
def v₂ := prop_var.mk 2
```

We will now refactor our definition of prop_expr to factor out mostly repeated code and to make explicit (1) a class of *literal* expressions, and (2) binary operators as first class citizens and a class of corresponding binary operator expressions. Be sure to compare and contrast our definitions here with the ones in the last chapter.

We'll start by defining a *binary operator* type whose values are abstract syntax terms for binary operators/connectives in propositional logic.

```
-- Syntactic terms for binary operators
inductive binop
| opAnd
| opOr
| opImp
| opIff
| opXor

open binop

-- A much improved language syntax spec
inductive prop_expr : Type
| pLit (b : bool)                          -- literal expressions
| pVar (v: prop_var)                       -- variable expressions
| pNot (e : prop_expr)                     -- unary operator expression
| pBinOp (op : binop) (e1 e2 : prop_expr)  -- binary operator expressions

open prop_expr


-- An example of an "and" expression
def an_and_expr :=
  pBinOp
    opAnd                    -- binary operator
    (pVar (prop_var.mk 0))   -- variable expression
    (pVar (prop_var.mk 1))   -- variable expression
```

We have to update the previous notations definitions, which now need to *desugar* to use the new expression constructors. We also define some shorthands for the two literal expressions in our language.

```
def True := pLit tt
def False := pLit ff
```

(continues on next page)

(continued from previous page)

```
notation (name := pVar) `[` v `]` :=  pVar v
notation (name := pAnd) e1 ∧ e2 :=  pBinOp opAnd e1 e2
notation (name := pOr) e1 ∨ e2 :=  pBinOp opOr e1 e2
notation (name := pNot) ¬e := pNot e
notation (name := pImp) e1 => e2 := pBinOp opImp e1 e2
notation (name := pIff) e1 ↔ e2 := pBinOp opIff e1 e2
notation (name := pXor) e1 ⊕ e2 := pBinOp opXor e1 e2

-- Precedence highest to lowest NOT, NAND, NOR, AND, OR, ->, ==
-- `↓`:37 x:37
reserve notation `↓`:37 x:37
notation (name := pNor) e1 `↓` e2 := pBinOp opAnd e1 e2

#print notation ¬
#print notation ∧
#print notation ↑
#print notation ↓
```

Here are examples of expressions using our concrete syntax

```
-- variable expressions from variables
def X := [v₀]
def Y := [v₁]
def Z := [v₂]

-- operator expressions
def e1 := X ∧ Y
def e2 := X ∨ Y
def e3 := ¬Z
def e4 := e1 => e2
def e5 := e1 ↔ e1
def e6 := X ⊕ ¬X
```

2.4.2 Semantics

A benefit of having made binary operators explicit as a set of syntactic terms is that we can simultaneously simplify and generalize our semantics.

```
-- Helper functions
def bimp : bool → bool → bool
| tt tt := tt
| tt ff := ff
| ff tt := tt
| ff ff := tt

def biff : bool → bool → bool
| tt tt := tt
| tt ff := ff
| ff tt := ff
| ff ff := tt
```

We now define an *interpretation* for operator symbols! Each binop (a syntactic object) has as its meaning some corresponding binary Boolean operator.

```
def op_sem : binop → (bool → bool → bool)
| opAnd := band
| opOr  := bor
| opImp := bimp
| opIff := biff
| opXor := bxor

-- A quick demo
#reduce ((op_sem opAnd) tt ff)
#reduce (op_sem opOr tt ff) -- recall left associativity
```

Now here's a much improved semantic specification. In place of rules for pTrue and pFalse we just have one rule for pLit (literal expressions). Second, in place of one rule for each binary operator we have one rule for *any* binary operator.

```
def pEval : prop_expr → (prop_var → bool) → bool
| (pLit b)          i := b
| ([v])             i := i v                    -- our [] notation on the left
| (¬e)              i := bnot (pEval e i)       -- our ¬ notation; Lean's bnot
| (pBinOp op e1 e2) i := (pEval e1 i) && (pEval e2 i) -- BUG!
```

2.4.3 Exploration

You've heard about Lean and seen in it action, but there's no substitute for getting into it yourself. The goal of this exploration is for you to "connect all the dots" in what we've developed so far, and for you to start to develop "muscle memory" for some basic Lean programming.

- Identify and fix the bug in the last rule of pEval
- Replace pNot with pUnOp ("unary operator"), as with pBinOp
- Add end-to-end support for logical *nand* (↑) and *nor* (↓) expression-building operators
- Define some examples of propositional logic expressions using concrete syntax
- Define several interpretations and evaluate each of your expressions under each one

To avoid future git conflicts, make a copy of src/04_prop_logic_syn_sem.lean, and make changes to that file rather than to the original. Bring your completed work to our next class. Be prepared to share and/or turn in your work at the beginning of next class.

2.5 Formal Validation

So far we've defined (1) an abstract syntax for propositional logic, (2) a "big step" operational semantics for our syntax, and (3) a concrete syntax for it, in the form of prefix (¬) and infix (∧, ∨, =>, etc) operators.

But how do we know that our *specification* is correct? Checking a specification for correctness is called *validating* it, or just validation. *Formal* validation is the use of mathematical logic to validate given specifications.

In particular, one can formally validate a specification by proving that it has certain required properties. To illustrate the point, n this chapter we add a fourth section,(4): a formal proof of the proposition that in our syntax and semantics, ∧ is commutative; that for *any* expressions in propositional logic, e1 and e2, and for *any* interpretation, the values of e1 ∧ e2 under i, and of e2 ∧ e1 under i, are equal.

Another way to look at this chapter is that it extends the set of elements of a good formalization of a mathematical concept from three above to four. Now the modular unit of definition specifies (1) the data (sometimes language syntax, sometimes

not), (2) the operations on that data, (3) the available concrete notations, and now also (4) proofs that essential properties hold. Abstract Syntax ————————

```
namespace cs6501

-- variables, indexed by natural numbers
inductive prop_var : Type
| mk (n : ℕ)

-- Abstract syntactic terms for unary operators
inductive unop
| opNot

-- Abstract syntactic terms for binary operators
inductive binop
| opAnd
| opOr
| opImp
| opIff
| opXor

-- make constructor names globally visible
open unop
open binop

-- syntax
inductive prop_expr : Type
| pLit (b : bool)                          -- literal expressions
| pVar (v: prop_var)                       -- variable expressions
-- | pNot (e : prop_expr)                     -- unary operator expression
| pUnOp (op :unop) (e : prop_expr)                    -- unary operator expression
| pBinOp (op : binop) (e1 e2 : prop_expr) -- binary operator expressions

open prop_expr
```

2.5.1 Concrete Syntax / Notation

```
-- notations (concrete syntax)
notation ` ⊤ ` := pLit tt    -- Notation for True
notation ` ⊥ ` := pLit ff    -- Notation for False
def True := pLit tt          -- deprecated now
def False := pLit ff         -- deprecated now
notation (name := pVar) ` [ ` v  ` ] ` :=  pVar v
notation (name := pNot) ¬e := pUnOp opNot e
notation (name := pAnd) e1 ` ∧ ` e2 :=  pBinOp opAnd e1 e2
notation (name := pOr) e1 ` ∨ ` e2 :=  pBinOp opOr e1 e2
precedence ` => `: 23                                          -- add operator␣
↪precedence
notation (name := pImp) e1 ` => `  e2 := pBinOp opImp e1 e2  -- bug fixed; add back␣
↪quotes
notation (name := pIff) e1 ` ↔ ` e2 := pBinOp opIff e1 e2
notation (name := pXor) e1 ` ⊕ ` e2 := pBinOp opXor e1 e2
-- Let's not bother with notations for nand and nor at this point
```

2.5.2 Semantics

The *semantic domain* for our language is not only the Boolean values, but also the Boolean operations. We map variables to Boolean values (via an interpretation) and we define a fixed mapping of logical connectives (¬, ∧, ∨, etc.) to Boolean operations (bnot, band, bor, etc.) With these elementary semantic mappings in place we can finally map *any* propositional logical expression to its (Boolean) meaning in a *compositional* manner, where the meaning of any compound expression is composed from the meanings of its parts, which we compute recursively, down to individual variables and connectives.

The Lean standard libraries define some but not all binary Boolean operations. We will thus start off in this section by augmenting Lean's definitions of the Boolean operations with two more: for implication (we follow Lean naming conventions and call this bimp) and bi-implication (biff).

```
-- Boolean implication operation (buggy!)
def bimp : bool → bool → bool
| tt tt := tt
-- A fault to inject first time through
-- Now corrected in following to lines
-- | tt ff := tt
-- | ff tt := ff
| tt ff := ff
| ff tt := tt
| ff ff := tt

-- Boolean biimplication operation
def biff : bool → bool → bool
| tt tt := tt
| tt ff := ff
| ff tt := ff
| ff ff := tt
```

Next we define a fixed interpretation for our syntactic logical connectives, first unary and then binary. We give these mappings in the form of functions from unary and binary operators (which act to compose logical expressions into new expressions), to Boolean operations (which compose Boolean values into Boolean results).

```
-- interpretations of unary operators
def un_op_sem : unop → (bool → bool)
| opNot := bnot

-- interpretations of binary operators
def bin_op_sem : binop → (bool → bool → bool)
| opAnd := band
| opOr  := bor
| opImp := bimp
| opIff := biff
| opXor := bxor
```

And now here's our overal expression semantic evaluation function. It works as described, computing the value of sub-expressions and composing the Boolean results into final Boolean meanings for any given expression under any give interpretation.

```
-- semantic evaluation (meaning of expressions)
def pEval : prop_expr → (prop_var → bool) → bool
| (pLit b)          i := b
| ([v])             i := i v                    -- our [] notation on the left
| (pUnOp op e)      i := (un_op_sem op) (pEval e i)     -- our ¬ notation; Lean's bnot
| (pBinOp op e1 e2) i := (bin_op_sem op) (pEval e1 i) (pEval e2 i) -- BUG FIXED :-)!
```

2.5.3 Formal Validation

```
-- proof of one key property: "commutativity of ∧" in the logic we've specified,, as␣
↪required
def and_commutes :
  ∀ (e1 e2 : prop_expr)
    (i : prop_var → bool),
    (pEval (e1 ∧ e2) i) = (pEval (e2 ∧ e1) i) :=
begin
-- assume that e1 e2 and i are arbitrary
assume e1 e2 i,

-- unfold definition of pEval for given arguments
unfold pEval,

-- unfold definition of bin_op_sem
unfold bin_op_sem,

-- case analysis on Boolean value (pEval e1 i)
cases (pEval e1 i),

-- within first case, nested case analysis on (pEval e2 i)
cases (pEval e2 i),

-- goal proved by reflexivity of equality
apply rfl,

-- second case for (pEval e2 i) within first case for  (pEval e1 i)
apply rfl,

-- onto second case for (pEval e1 i)
-- again nested case analysis on (pEval e2 i)
cases (pEval e2 i),

-- and both cases are again true by reflexivity of equality
apply rfl,
apply rfl,
-- QED
end
```

2.5.4 Examples

```
-- tell Lean to explain given notations
#print notation ¬
#print notation ∧
#print notation ↑

-- variables
def v₀ := prop_var.mk 0
def v₁ := prop_var.mk 1
def v₂ := prop_var.mk 2

-- variable expressions
def X := [v₀]
def Y := [v₁]
```

(continues on next page)

(continued from previous page)

```
def Z := [v₂]

-- operator expressions
def e1 := X ∧ Y
def e2 := X ∨ Y
def e3 := ¬Z
def e4 := e1 => e2
def e5 := e1 ↔ e1
def e6 := X ⊕ ¬X

-- an interpretation
def an_interp : prop_var → bool
| (prop_var.mk 0) := tt -- X
| (prop_var.mk 1) := ff -- Y
| (prop_var.mk 2) := tt -- Z
| _ := ff               -- any other variable

-- evaluation
#reduce pEval X an_interp  -- expect false
#reduce pEval Y an_interp  -- expect false
#reduce pEval e1 an_interp  -- expect false
#reduce pEval (X => Y) an_interp  -- oops

-- applying theorem!
#reduce and_commutes X Y an_interp
-- result is a proof that ff = ff
```

2.5.5 Exercises

1. Formally state and prove that in our language, or (∨) is commutative (1 minute!)
2. Formally state and prove that in our language, not (¬) is involutive (a few minutes). Hints: Put parens around (¬¬e). Open the Lean infoview with CTRL/CMD-SHIFT-RETURN/ENTER. If you get hung up on Lean syntax, ask a friend (or instructor) for help to get unstuck.

2.5.6 Solutions

```
-- The proof that ∨ is commutative is basically identical to that for ∧
def or_commutes :
  ∀ (e1 e2 : prop_expr)
    (i : prop_var → bool),
    (pEval (e1 ∨ e2) i) = (pEval (e2 ∨ e1) i) :=
begin
-- Suppose e1 e2 and i are arbitrary expressions and interpretation
assume e1 e2 i,
-- unfold definitions of pEval and bin_op_sem applied to their arguments
unfold pEval,
unfold bin_op_sem,
-- proof by simple case analysis on possible results of evaluating e1 and e2 under i
cases (pEval e1 i),
cases (pEval e2 i),
apply rfl,
apply rfl,
```

(continues on next page)

(continued from previous page)

```
cases (pEval e2 i),
apply rfl,
apply rfl,
-- QED: By showing it's true for arbitrary e1/e2/i we've shown it's true for *all*
end

-- Prove not is involutive
theorem not_involutive: ∀ e i, (pEval e i) = (pEval (¬¬e) i) :=
begin
assume e i,
unfold pEval,
unfold un_op_sem,
cases (pEval e i),
repeat { apply rfl },
end

end cs6501
```

```
import .A_05_prop_logic_properties
namespace cs6501
```

2.6 Algebraic Axioms

We've now seen that it's not enough to prove a few theorems about a construction (here our syntax and semantics for propositional logic).

So how will we confirm *for sure* that our model (or implementation) of propositional logic is completely valid?

We'll offer two different methods. First, in this chapter, we'll prove that our specification satisfies the *algebraic axioms* of propositional logic. Second, in the next chapter, we'll prove that the *inference rules* of propositional logic are valid in our model.

Along the way we'll take the opportunity to see more of what Lean can do for us: - Scott/semantic bracket notation for *meaning-of* - declare automatically introduced variables - use of implicit arguments to further improve notation - universally quantified variables are function arguments - "sorry" to bail out of a proof and accept proposition as axiom

To avoid duplication of code from the last chapter, we'll import all of the definitions in its Lean file for use here. Algebraic Properties ————————

First, a propositional logic can be understood as an *algebra* with Boolean-valued (as opposed to numeric) terms, variables, and constants. Constants and variables are are terms, but terms are also constructed from smaller terms using connectives: $\wedge$, $\vee$, $\neg$, and so on. The axioms of propopsitions define how these operations work, on their own and when combined.

These algebraic properties of propositional logic are very much akin to the usual commutativity, associativity, distributivity, identity, and other such properties of the natural numbers and the addition and multiplication operations on them. As we go through the analogous properties for propositional logic, note the common properties of both algebras.

2.6.1 Commutativity

The first two axioms that that the and and or operators (∧, ∨) are commutative. One will often see these rules written in textbooks and tutorials as follows:

- (p ∧ q) = (q ∧ p)
- (p ∨ q) = (q ∨ p)

This kind of presentation hides a few assumptions. First, it assumes p and q are taken to be arbitrary expressions in propositional logic. Second, it assumes that what is really being compared here are not the expressions per se but their semantic meanings. Third it assumes that equality of meanings hold under all possible interpretations.

To be completely formal in Lean, we need to be explicit about these matters. We need to define variables, such as p and q, to be arbitrary expressions. Second, we need to be clear that the quantities that are equal are not the propositions themselves but their *meanings* under all possible interpretations.

We have already seen, in the last chapter, how to do this. For example, we defined the commutative property of ∧ as follows. In mathematical natural language we can read this as "for any expressions, p and q, the meaning of $p \wedge q$ is equal to that of $q \wedge p$ under all interpretations."

```
example :
∀ (p q : prop_expr) (i : prop_var → bool),
  pEval (p ∧ q) i = pEval (q ∧ p) i :=
and_commutes  -- proof from last chapter
```

2.6.2 Another Notation

As we've seen, mathematical theories are often augmented with concrete syntactic notations that make it easier for people to read and write such mathemantics. We would typically write *3 + 4*, for example, in lieu of *nat.add 3 4*. For that matter, we write *3* for (succ(succ(succ zero))). Good notations are important.

One area in our specification that could use an improvement is where we apply the *pEval* semantic *meaning-of* operator to a given expression. The standard notation for sucg a "meaning-of" operator is a pair of *denotation* or *Scott* brackets.

We thus write ⟦ *e* ⟧ as "the meaning of *e*" and define this notation to desugar to *pEval e*. We thus write ⟦ *e* ⟧ *i* to mean the truth (Boolean) value of *e* under the interpretation i. Thus, the expression, ⟦ *e* ⟧ *i*, desugars to *pEval e i*, which in turn reduces to the Boolean meaning of *e* under *i*.

With this notation in hand, we'll be able to write all of the algebraic axioms of propositional logic in an easy to read, mathematically fairly standard style. So let's go ahead and define this notation, and then use it to specify the commutative property of logical ∧ using it. Here's the notation definition. When an operation, such as pEval is represented by tokens on either side of an argument, we call this an outfix notation.

```
notation (name := pEval) ` ⟦ ` p ` ⟧ `  :=  pEval p
```

2.6.3 Variable Declarations

It's common when specifying multiple of properties of a given object or collection of objects to introduce the same variables at the beginning of each definition. For example, we started our definition of the commutative property with ∀ *(p q : prop_expr) (i : prop_var → bool)*. Lean allows us to avoid having to do this by declaring such variables once, in a *section* of a specification, and then to use them in multiple definitions without the need for redundant introductions. Let's see how it works.

```
-- start a section
section prop_logic_axioms

-- Let p, q, r, and i be arbitrary expressions and an
-- interpretation
variables (p q r : prop_expr) (i : prop_var → bool)
```

Now we can write expressions with these variables without explicitly introducing them. As an aside, in this example, we add prime marks to the names used in imported chapter to avoid conflicts with names used in that file.

```
def and_commutes' := (⟦(p ∧ q)⟧ i) = (⟦(q ∧ p)⟧ i)
def or_commutes' :=  ⟦(p ∧ q)⟧ i = ⟦(q ∧ p)⟧ i
```

2.6.4 Specialization of Generalizations

Observe: We can *apply* these theorems to particular objects to specalize the generalized statement to the particular objects.

```
#reduce and_commutes' p q i
```

We can use notations not only in writing propositions to be proved but also in our proof-building scripts. In addition to doing that in what follows, we illustrate two new elements of the Lean proof script (or tactic) language. First, we can sequentially compose tactics into larger tactics using semi-colon. Second we can use *repeat* to repeated apply a tactic until it fails to apply. The result can be a nicely compacted proof script.

```
-- by unfolding definitions and case analysis
example : and_commutes' p q i :=
begin
unfold and_commutes' pEval bin_op_sem,
cases ⟦ p ⟧ i,
repeat { cases ⟦ q ⟧ i; repeat { apply rfl } },
end
```

2.6.5 Associativity

```
def and_associative_axiom :=  ⟦(p ∧ q) ∧ r⟧ i = ⟦(p ∧ (q ∧ r))⟧ i
def or_associative_axiom :=   ⟦(p ∨ q) ∨ r⟧ i = ⟦(p ∨ (q ∨ r))⟧ i
```

2.6.6 Distributivity

```
def or_dist_and_axiom := ⟦p ∨ (q ∧ r)⟧ i = ⟦(p ∨ q) ∧ (p ∨ r)⟧ i
def and_dist_or_axiom := ⟦p ∧ (q ∨ r)⟧ i = ⟦(p ∧ q) ∨ (p ∧ r)⟧ i
```

2.6.7 DeMorgan's Laws

```
def demorgan_not_over_and_axiom := ⟦¬(p ∧ q)⟧ i = ⟦¬p ∨ ¬q⟧ i
def demorgan_not_over_or_axiom :=  ⟦¬(p ∨ q)⟧ i = ⟦¬p ∧ ¬q⟧ i
```

2.6.8 Negation

```
def negation_elimination_axiom := ⟦¬¬p⟧ i = ⟦p⟧ i
```

2.6.9 Excluded Middle

```
def excluded_middle_axiom := ⟦p ∨ ¬p⟧ i = ⟦⊤⟧ i    -- or just tt
```

2.6.10 No Contradiction

```
def no_contradiction_axiom := ⟦p ∧ ¬p⟧ i = ⟦⊥⟧ i    -- or just tt
```

2.6.11 Implication

```
def implication_axiom := ⟦(p => q)⟧ i = ⟦¬p ∨ q⟧ i  -- notation issue

example : implication_axiom p q i :=
begin
unfold implication_axiom pEval bin_op_sem un_op_sem,
cases ⟦ p ⟧ i; repeat { cases ⟦ q ⟧ i; repeat { apply rfl } },
end
```

The next two sections give the axioms for simplifying expressions involving $\wedge$ and $\vee$.

2.6.12 And Simplification

$p \wedge p = p$ $p \wedge T = p$ $p \wedge F = F$ $p \wedge (p \vee q) = p$

2.6.13 Or Simplification

$p \vee p = p$ $p \vee T = T$ $p \vee F = p$ $p \vee (p \wedge q) = p$

```
end prop_logic_axioms
end cs6501
```

2.6.14 Homework

1. Formalize the and/or simplification rules.

2. Use Lean's *theorem* command to assert, give, and name proofs that our Lean model satisfies all of the algebraic axioms of propositional logic, as formalized above.

Solving this problem is repetitive application of what we've done already in a few examples, but it's still worth writing and running these proofs scripts a few times to get a better feel for the process.

3. Collaboratively refactor the "code" we've developed into a mathematical library component with an emphasis on good design.

What does that even mean? Good with respect to what criteria, deciderata, needs, objectives?

- data type definitions
- operation definitions
- notation definitions
- formal validation
- some helpful examples

2.7 Inference Rules

In this chapter we will present another approach to validating our model of propositional logic: by verifying that it satisifies the *inference rules* of this logic.

An inference rule is basically a function that takes zero or more arguments, usually including what we call *truth judgements* or *proofs* of certain propositions, and that returns truth judgments or proofs of other propositions, which are said to be *derived* or to be *deduced* from the arguments.

For example, in both propositional and *first-order predicate* logic, we have a rule, *and introduction*, that takes as arguments, or *premises*, truth judgments for any two arbitrary propositions, X and Y, and that returns a truth judgment for $X \wedge Y$.

A truth judgment is a determination that a proposition, say X, is logically true, and can be written (in paper and pencil logic as) X *: true*. The *and introduction* rule thus states that X *: True,* Y *: True* $\vdash X \wedge Y$ *: True.*

This is usually shortened to $X, Y \vdash X \wedge Y$ based on the assumption that everything to the left of the turnstile is assumed to have already been judged to be true. Such a rule can be pronounced as follows: in a context in which you have already judged X and Y to be true you can always conclude that X $\wedge$ Y is true.

What's different is that these rules are syntactic and don't presume that we have an algorithm for determining truth. We do for propositional logic, but not predicate logic. Learning the basic inference rules of the logic is thus essential for reasoning about the truth of given propositions expressed in predicate logic.

```
import .A_06_prop_logic_algebraic_axioms
namespace cs6501
```

2.7.1 Inference Rules

Key idea: These are rules for reasoning about evidence. What *evidence* do you need to derive a given conclusion? These are the "introduction" rules. From a given piece of evidence (and possibly with additional evidence) what new forms of evidence can you derive? These are "elimination" rules of the logic.

```
-- 1. ⊢ ⍰                      -- true introduction
-- 2. ⊥, X ⊢ X                 -- false elimination

-- 3. X, Y ⊢ X ∧ Y             -- and_introduction
-- 4. X ∧ Y ⊢ X                -- and_elimination_left
-- 5. X ∧ Y ⊢ Y                -- and_elimination_right

-- 6. X ⊢ X ∨ Y                -- or introduction left
-- 7. Y ⊢ X ∨ Y                -- or introduction right
-- 8. X ∨ Y, X → Z, Y → Z ⊢ Z -- or elimination

-- 9. ¬¬X ⊢ X                  -- negation elimination
-- 10. X → ⊥ ⊢ ¬X               -- negation introduction

-- 11. (X ⊢ Y) ⊢ (X → Y)       -- a little complicated
-- 12. X → Y, X ⊢ Y            -- arrow elimination

-- 13. X → Y, Y → X ⊢ X ↔ Y    -- iff introduction
-- 14. X ↔ Y ⊢ X → Y           -- iff elimination left
-- 15. X ↔ Y ⊢ Y → X           -- iff elimination right
```

```
open cs6501

theorem and_intro_valid : ∀ (X Y : prop_expr) (i : prop_var → bool),
    (⟦X⟧ i = tt) → (⟦Y⟧ i = tt) → (⟦(X ∧ Y)⟧ i = tt) :=
begin
assume X Y i,
assume X_true Y_true,
unfold pEval bin_op_sem, -- axioms of eq
rw X_true,
rw Y_true,
apply rfl,
end

theorem and_elim_left_valid :
∀ (X Y : prop_expr) (i : prop_var → bool),
(⟦(X ∧ Y)⟧ i = tt) → (⟦X⟧ i = tt) :=
begin
unfold pEval bin_op_sem,
assume X Y i,
assume h_and,
cases ⟦ X ⟧ i,
cases ⟦ Y ⟧ i,
cases h_and,
cases h_and,
cases ⟦ Y ⟧ i,
cases h_and,
apply rfl,
end

theorem or_intro_left_valid :
```

(continues on next page)

(continued from previous page)

```
∀ (X Y : prop_expr) (i : prop_var → bool),
(⟦(X)⟧ i = tt) → (⟦X ∨ Y⟧ i = tt) :=
begin
unfold pEval bin_op_sem,
assume X Y i,
assume X_true,
rw X_true,
apply rfl,
end

theorem or_elim_valid : ∀ (X Y Z : prop_expr) (i : prop_var → bool),
(⟦ (X ∨ Y) ⟧ i = tt) →
(⟦ (X => Z) ⟧ i = tt) →
(⟦ (Y => Z) ⟧ i = tt) →
(⟦ Z ⟧ i = tt) :=
begin
-- expand definitions as assume premises
unfold pEval bin_op_sem,
assume X Y Z i,
assume h_xory h_xz h_yz,

-- the rest is by nested case analysis
-- this script is refined from my original
cases (⟦ X ⟧ i), -- case analysis on bool (⟦ X ⟧ i)
repeat {
  repeat {      --  case analysis on bool (⟦ Y ⟧ i)
    cases ⟦ Y ⟧ i,
    repeat {    -- case analysis on bool (⟦ Z ⟧ i)
      cases ⟦ Z ⟧ i,
      /-
      If there's an outright contradiction in your
      context, this tactic will apply false elimination
      to ignore/dismiss this "case that cannot happen."
      -/
      contradiction,
      apply rfl,
    },
  },
},
end
```

2.7.2 Practice

In the style of the preceding examples, formally state, name, and prove that each of the remaining inference are also valid in our logic. Identify any rules that fail to be provable in the presence of the bug we injected in bimp. You can do this by completing the proofs and seeing how they break when bugs are added to our definitions.

```
-- Write your formal propositions and proofs here:



end cs6501
```

CHAPTER

THREE

INDEX