Computing Foundations

Release 1.2.2

Kevin Sullivan

Mar 16, 2023

CONTENTS

1 CS6501 Spring 2023 **1**
- 1.1 To Think Abstractly . . . 1
- 1.2 Welcome. Overview. . . . 3

2 Propositional Logic **5**
- 2.1 Balanced Parentheses . . . 5
- 2.2 Simplified Propositional Logic . . . 8
- 2.3 Propositional Logic . . . 11
- 2.4 A Better Specification . . . 13
- 2.5 Formal Validation . . . 15
- 2.6 Algebraic Axioms . . . 20
- 2.7 Inference Rules . . . 24
- 2.8 Inference Rules Validation . . . 27

3 Constructive Logic **31**
- 3.1 Higher-Order Predicate Logic . . . 31
- 3.2 Propositions as Types . . . 31
- 3.3 From Truth to Proof . . . 34
- 3.4 Inference Rules . . . 36

4 Recursive Types **47**
- 4.1 Introduction . . . 47
- 4.2 Natural Numbers . . . 47
- 4.3 Polymorphic Lists . . . 49
- 4.4 Higher-Order Functions . . . 53
- 4.5 Recursive Proofs . . . 62

5 Dependent Types **67**
- 5.1 Σ Types . . . 67
- 5.2 Π Types . . . 67

6 Index **69**

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

PROPOSITIONAL LOGIC

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 seen that it's not enough to prove just a few theorems about a construction (here our syntax and semantics for propositional logic). So how will we confirm *for sure* that our model (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.

2.6.1 Algebraic Axioms

First, then, we will formalize propositional logic as an *algebra,* with Boolean-valued (as opposed to numeric) terms and operations, and then we will show that our operations and terms satisfy the axioms of propositional logic. For example, we will have to show that our specifications of $\wedge$ and $\vee$ are both commutative and associative.

These properties are analogous to the usual commutativity, associativity, distributivity, and other such properties of the natural numbers and the usual algebra on them. As we go through the analogous properties for propositional logic, take note of the common properties of both algebras.

In the rest of this chapter we will formally state and prove that our Lean model of propositional logic satisfies all of the axioms/properties required to be a correct model of the logic:

- commutativity
- associativity
- distributivity

- DeMorgan's laws
- double negation elimination
- excluded middle
- no contradiction
- implication
- and simplification
- or simplification

2.6.2 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.

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

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."

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
```

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
```

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
```

2.6.3 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
```

2.6.4 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
```

2.6.5 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
```

2.6.6 Negation

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

2.6.7 Excluded Middle

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

2.6.8 No Contradiction

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

2.6.9 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 $\vee$.

2.6.10 And Simplification

```
-- p ∧ p = p
-- p ∧ T = p
-- p ∧ F = F
-- p ∧ (p ∨ q) = p
```

2.6.11 Or Simplification

```
-- p ∨ p = p
-- p ∨ T = T
-- p ∨ F = p
-- p ∨ (p ∧ q) = p

end prop_logic_axioms
end cs6501
```

2.6.12 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,
```

(continues on next page)

(continued from previous page)

```
cases h_and,
apply rfl,
end

theorem or_intro_left_valid :
∀ (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
```

```
import .A_06_prop_logic_algebraic_axioms
namespace cs6501
```

2.8 Inference Rules Validation

This chapter pulls together in one place a formal validation of the claim that our model of propositional logic satisfies all of the =inference rules of that logic.

The first section of this chapter refactors the partial solution we developed in class, grouping definitions of the propositions that represent them, and separately a proof that each rule expresses in our model is valid. These sections also afford opportunities to intruduce a few more concepts in type theory and Lean.

To begin we import some definitions and declare a set of variables available for use in this file.

```
section rule_validation
variables
  (X Y Z: prop_expr)
  (i : prop_var → bool)
```

2.8.1 Inference Rule Statements

We start with a refactoring of the results of the last chapter, into formal statements of the inference rules and formal proofs that these rules are valid (truth- preserving under all interpretations) in our model of propositional logic.

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.

```
-- remember, we can now use X, Y, Z, i

def true_intro_rule := ⟦ ⊤ ⟧ i = tt
def false_elim_rule := ⟦⊥⟧ i = tt → ⟦X⟧ i = tt  -- X is any proposition
def and_intro_rule := ⟦ X ⟧ i = tt → ⟦ Y ⟧ i = tt → ⟦(X ∧ Y)⟧ i = tt
def and_elim_left_rule := (⟦(X ∧ Y)⟧ i = tt) → (⟦X⟧ i = tt)
def and_elim_right_rule := (⟦(X ∧ Y)⟧ i = tt) → (⟦Y⟧ i = tt)
def or_intro_left_rule := (⟦X⟧ i = tt) → (⟦(X ∨ Y)⟧ i = tt)
def or_intro_right_rule := (⟦Y⟧ i = tt) → (⟦(X ∨ Y)⟧ i = tt)
def or_elim_rule :=   (⟦(X ∨ Y)⟧ i = tt) →
                      (⟦(X => Z)⟧ i = tt) →
                      (⟦(Y => Z)⟧ i = tt) →
                      (⟦(Z)⟧ i = tt)
```

(continues on next page)

(continued from previous page)

```
-- formalize the rest
-- 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

-- note:
#reduce and_intro_rule e1 e2 i

-- prove it
theorem and_intro : and_intro_rule e1 e2 i :=
begin
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 : and_elim_left_rule X Y i :=
begin
unfold and_elim_left_rule pEval bin_op_sem,
-- case analysis
assume h_and,
cases ⟦ X ⟧ i,     -- cases analysis on X
{ -- case X (evaluates to) false
  cases ⟦ Y ⟧ i,  -- nested case analysis on Y
    cases h_and,   -- contradiction
    cases h_and,   -- contradiction
},
{ -- case X (evaluates to) true
  cases ⟦ Y ⟧ i, -- nested case analysis on Y
  cases h_and,  -- contradiction
  apply rfl,    -- ahh, equality
},
end

theorem or_intro_left : or_intro_left_rule X Y i
:=
begin
unfold or_intro_left_rule pEval bin_op_sem,
assume X_true,
rw X_true,
apply rfl,
end

theorem or_elim : or_elim_rule X Y Z i :=
begin
-- expand definitions as assume premises
unfold or_elim_rule pEval bin_op_sem,
assume h_xory h_xz h_yz,
```

(continues on next page)

(continued from previous page)

```
-- 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
```

2.8.2 Proofs

In the style of the preceding examples, give formal proofs of that the remaining inference rules are valid in our own model of propositional logic.

Identify any rules that fail to be provably valid in the presence of the bug we'd injected in bimp. Which rule validation proofs break when you re-activate that bug?

To get you started, the following proof shosws that the false elimination inference rule is valid in our logic. For any proposition, e, and interpretation, i, in our logic, if $\bot$ implies e, so from the truth of $\bot$, the truth of any expression follows. But $\bot$ can't ever be (evaluate to) true, because we've defined the logic otherwise. But wait, is there another way to formalize the axiom? If so, are the two ways equivalent?

```
theorem false_elim : false_elim_rule X i :=
begin
unfold false_elim_rule pEval,
assume h,
cases h,  -- contradiction, can't hppen, no cases!
          -- Lean determines tt = ff is impossible
end


-- Define the remaning propositions and proofs here:


end rule_validation -- section
end cs6501          -- namepsace
```

CHAPTER

THREE

CONSTRUCTIVE LOGIC

3.1 Higher-Order Predicate Logic

The term, *predicate logic*, used informally, is usually taken to refer to first-order predicate logic (often extended with theories, e.g., of natural number arithmetic). However, in this course, you will learn higher-order constructive predicate logic. First-order logic is a special restricted case.

We've organized the course so far to prepare you to quickly pick up higher-order predicate logic as it's *embedded* in the logic of the Lean prover, by definitions provided by *mathlib*, Lean's main library of mathematical definitions.

Major similarities and changes include the following:

- Propositions become types, not just logical expressions
- Truth judgments ($[\![E]\!]$ i = tt) replaced by proof judgments (e : E)
- Functions and applications are essential parts of predicate logic
- Predicates are functions from values to propositions *about them*
- We adopt all of the usual propositional logic connectives
- We adopt generalized versions of the usual inference rules
- We gain two new ones: universal and existential quantifiers
- We gain new inference rules for the $\forall$ and $\exists$ quantifiers
- Generalizing ($\forall$) over types gives us parametric polymorphism

3.2 Propositions as Types

In the last section, we build on all of the ideas of the last chapter to gain an understanding of higher order predicate logic in Lean. Each section of this chapter focuses on a dimension in which the latter is different, often a powerful generalization, of ideas from the last chapter.

As an example, in the last chapter, we represented propositions as terms of our data type, *prop_expr*. We also saw that we could formulate inference rules of propositional logic to provide a way to reason about the truth of given propositions. In this chapter, we will represent propositions as types of a special kind, instead, and proofs as values of these types. We will then adopt exacty the same inference rules we saw in the last chapter, but generalized to this far more expressive logic.

To warm up for the idea that propositions are (represented as) types (in type theory), as an example we'll first look at the types related to the natural number, 1. Then we'll analyze the types of a few propositions, namely 1 = 1 and 0 = 1. The take away message will be that propositions are special types that live in their own *type universe* and values of such types (if there are any) serve as *proofs* of such propositions.

3.2.1 Computational Types

```
-- The type of 1 is nat
#check 1

-- Type type of nat is *Type*
#check nat

-- The type of all basic computational types is *Type*
#check bool
#check string
#check list bool

-- A natural question: What's the type of Type, etc?
#check Type
#check Type 0   -- Type is shorthand for Type 0
#check Type 1   -- It's type universes all the way up
#check Type 2   -- etc.

/-
What we have so far then is a hierarchy of "computational"
types like this:

       ...        higher type universes
        |
      Type 1 (the type of objects that contains Type 0 objects)
        |
      Type 0         a type universe
        |
       nat               type
        |
        1               value

Type Universes
--------------
-/

-- What universes do various objects inhabit
#check nat              -- Type 0
#check list nat         -- list of nats : Type 0

def list_of_types : list Type := [nat, bool, string]
#check list (Type 0)    -- list of Type 0s : Type 1
#check list (Type 1)    -- list of Type 1s : Type 2
```

3.2.2 Logical Types (Propositions)

Now we'll turn to the idea that propositions are types of a logical kind. In Lean and related proof assistants and in type theory more generally, propositions are represented as *types* that inhabit a special type universe: in Lean, Prop.

```
#check 1 = 1  -- 1 = 1 is a proposition, thus of type Prop
#check ∃ (a b c : ℕ), a*a + b*b = c*c -- also of type Prop
```

```
-- Example: Here's a proof of 1 = 1
def proof_of_1_eq_1 := eq.refl 1
```

(continues on next page)

(continued from previous page)

```
-- What is its type?
#check proof_of_1_eq_1
```

In Lean, types are terms, too, and so they have types, as we have already seen. So what is the type of *1 = 1*. It's Prop. As we've said, logical types (propositions) inhabit the type universe, Prop, also known as Sort 0.

```
#check 1 = 1
#check Prop
```

Now we can clearly see the type hierarchy. Each proposition is a type, and all such types are in turn of type, Prop. We we have the following picture of the type hierarchy for the terms we've just constructed.

```
-- type is 1 = 1
#check proof_of_1_eq_1

-- type is Prop
#check 1 = 1
```

You might finally ask, what is the type of Prop? It's Type!

```
#check Prop

/-
We can draw a picture now to see how things work.

Prop (Sort 0) --> Type (Sort 1, Type 0) --> Type 1 (Sort 2) --> etc
    |                        |                      |
    |                        |                      |
  1 = 1                     nat                 list Type
    |                        |                      |
    |                        |                      |
eq.refl 1                    1              [nat, bool, string]
-/
```

3.2.3 Type Universe Levels

In the top row we have a hierarchy of type universes, starting with Sort 0 and extending up. Prop is the common name in Lean for Sort 0, and Type is the common name for Sort 1. In the second row are examples of types that inhabit each of the universes. *1 = 1* for example is a logical type (a proposition) in Prop, while nat is a computational type in Type. Finally, *list Type* is the type of lists of terms each of type Type, i.e., lists of computational types. The bottom row gives examples of values of each of the types above: *eq.refl 1* is a proof/value of (type) *1 = 1*; 1 is a value of type *nat*; and *[nat, bool, string]*, because it contains as elements values of type Type 0 is itself a value of type Type 1.

As a final comment, Lean allows one to generalize over these type universes. To do so you declare one or more *universe variable* which you can then use in decaring types. Lean can also infer universe levels.

```
-- declare two possible different type Universe levels
universes u v

--
#check @prod.mk

-- Here's a function that takes two types in arbitrary universes
```

(continues on next page)

(continued from previous page)

```
def mk_pair (α : Sort u) (β : Sort v) := (α, β)

-- Here are examples of type pairs we can form.
#check mk_pair nat bool
#check mk_pair (1=1) (2=1)
#check mk_pair (1=1) (list Type)

/-
As another example, here's a thoroughly polymorphic version of
the identity function. Given a type as an argument, it takes a
second value, a, *of that previously given type*, and returns it.
-/

def my_id (α : Sort u) (a : α) := a

-- applications to objects of various types
#reduce my_id Prop (1=1)
#reduce my_id nat 1
#reduce my_id (list Type) [nat,nat,bool]
```

3.2.4 Implicit Arguments

```
-- With {} we tell Lean that α should be inferred automatically
def my_id' { α : Sort u } (a : α) := a

-- Now we don't provide the type arguments explicitly
#reduce my_id' (1=1)
#reduce my_id' 1
#reduce my_id' [nat,nat,bool]

-- If necessary through, we can turn off implicit inference using @
#reduce @my_id' Prop (1=1)
#reduce @my_id' nat 1
#reduce @my_id' (list Type) [nat,nat,bool]
```

3.3 From Truth to Proof

In both propositional and first order predicate logic, inference rules are defined in terms of *truth judgments*. For example, in our propositional logic, we defined the *and elimination left* rule as (⟦ P ∧ Q ⟧ i == tt) → ⟦ P ⟧ i = tt. You can read it as saying, if we've judged that P ∧ Q is true under some interpretation, i, then it must be that P is also true under that interpretation.

In the logic of Lean, by contrast, inference rules are defined in terms of *proof* judgments. In Lean, if *P* is a proposition, then to express the idea that p is a proof of P, we write (p : P). We can also read this proof judgement as a type judgment: that p is a value of type P. The and elimination left rule thus changes to the following:

```
def and_elim_left_rule' := ∀ (P Q : Prop), P ∧ Q → P
```

You can read the proposition as saying the following: if P and Q are arbitrary propositions, then if you are given a *proof* (value) of (type) P ∧ Q, then you can derive a proof (value) of (type) P.

Written in the typical inference rule styles you will find in the literature, we'd see something like this: *(P Q : Prop) (p : P) (q : P)* ⊢ ⟨*p,q*⟩ *: P* ∧ *Q*.

In a textbook, it'd often be assumed or implicit that P and Q are propositions, in which case this rule would be shortened to $P, Q \vdash P \wedge Q$.

Using Lean's *variables* declaration, we can achieve the same clarity with complete formality.

```
-- Let P, Q, and R be arbitrary propositions
variables (P Q R : Prop)

-- Now we can write the rule without the forall
def and_elim_left_rule := P ∧ Q → P

/-
Lean doesn't treat the two versions exactly the
same. The type of and_elim_left_rule is of course
just Prop, because it's a proposition. However,
because we've declared *P* and *Q* to be general
in the current environment, they are understood
as arguments, allowing and_elim_left_rule to be
*applied* to any two propositions as arguments
(say P, Q), yielding the specified proposition as
a result.
-/
#reduce and_elim_left_rule'   -- just a proposition
#reduce and_elim_left_rule    -- function to proposition

#check and_elim_left_rule          -- Prop → Prop → Prop
#reduce and_elim_left_rule         -- A proposition
#reduce and_elim_left_rule P Q     -- Namely, P ∧ Q → P

#check and_elim_left_rule'        -- just Prop
#check and_elim_left_rule' P Q    -- Can't be applied

-- and_elim_left_rule applies to *any* propositions
variables (A B : Prop)
#check and_elim_left_rule A B
#reduce and_elim_left_rule A B
```

We can now assert the validity of this rule, and prove it. Of course the proof in this case will be nothing but the application of Lean's version, and.elim_left.

```
theorem and_elim_left_valid : and_elim_left_rule P Q :=
-- assume h is a proof of P → Q, show P
begin
unfold and_elim_left_rule,
intro h,
-- apply and.elim_left h
exact h.left,
end

-- The theorem now applies generally to any propositions

theorem and_elim_left_valid_2 : and_elim_left_rule A B :=
begin
apply and_elim_left_valid,
end
```

3.3.1 Practice Example

Exercise: Define two propositions (types in Prop) with made up names, each having two proof constructors also with made up names. Recall that we define types with an *inductive* definition. Here's the exact definition of the bool type, for example.

inductive bool : Type | tt | ff

The big difference now is that we want to represent logical propositions, so we will define types not in the universe, Type, but in Prop, the universe that all logical propositions inhabit. Now it's easy as the next example shows. We define two propositions, each with two "proofs," and show that we can construct a proof of the the conjunction of the two propositions.

```
-- A proposition called KevinIsFromCville with two proofs
inductive KevinIsFromCville : Prop
| DL  -- driver's license
| EB  -- electric bill

-- Another similar proposition
inductive NickIsFromNewHampshire : Prop
| DL      -- driver's license
| EB      -- electric bill
| LFODLP  -- secret code

-- Because we can prove each one we can prove the conjunction
example : KevinIsFromCville ∧  NickIsFromNewHampshire :=
begin
apply and.intro _ _,
exact KevinIsFromCville.EB,
exact NickIsFromNewHampshire.LFODLP,
end

-- Similarly, from a proof of a conjunctions we can prove each
example : KevinIsFromCville ∧  NickIsFromNewHampshire → KevinIsFromCville :=
begin
assume h,
cases h,
assumption,
end
```

3.4 Inference Rules

Next we'll see that most of the inference rules of propositional logic have analogues in constructive predicate logic, provided to us by *mathlib*, Lean's library of mathematical definitions.

Your next major task is to know and understand these inference rules. For each connective, learn its related introduction (proof constructing) and elimination (proof consuming) rules. Grasp the sense of each rule clerly. And learn how to to compose them, e.g., in proof scripts, to produce proofs of more complex propositions.

This new inference rule is just an "upgraded" version of the and-elimination-left inference rule from from the last chapter. The major task in the rest of this chapter is to "lift" your established understanding of all of the inference rules of propositional logic to the level of higher-order constructive logic. Along the way we'll see a few places where the "classical" rules don't work. We now go through each rule from propositional logic and give its analog in the predicate logic of the Lean prover.

3.4.1 true (□)

In propositional logic, we had the rule that ⊤, always evaluates to true (tt in Lean). The definition said this: *true_intro_rule* := □ □ □ *i* = *tt*.

In Lean, by contrast, there is a proposition, *true*, that has proof, called *intro*. We write *true.intro* to refer to it in its namespace.

```
#check true                    -- a proposition
example : true := true.intro   -- a proof of it
```

Now we'll see exactly how the proposition, true, with true.intro as a proof, how it is all defined. It's simple. Propositions are types, so true is a type, but one that inhabits Prop; and it has one constant constructor and that's the one and only proof, *intro*.That's it!

inductive true : Prop | intro : true

Sadly, a proof of true is pretty useless. A value of this type doesn't even provide one bit of information, as a Boolean value would. There's no interesting elimination rule for true.

3.4.2 false

In propositional logic, we had the propositional expression (prop_expr), ⊥, for *false*. In Lean, by contrast, *false* is a *proposition*, which is to say, a type, called *false*. Because we want this proposition never to be true, we define it as a type with no values/proofs at all–as an uninhabited type.

inductive false : Prop

There is no way ever to produce a proof of *false* because the type has no value constructors. There is no introduction rule false.

In propositional logic, the false elimination rule said that if an expression evaluates to ff, then it follows (implication) that any other expression evaluates to tt. The rule in Lean is called false.elim. It says that from a proof of false, a proof (or value) of *any* type in any type universe can be produced: not only proofs of other propositions but values of any types.

```
#check false
#check @false.elim  -- false.elim : Π {C : Sort u_1}, false → C

-- explicit application of Lean's false.elim rule
example : false → 0 = 1 :=
begin
assume f,
exact false.elim f,        -- So what is C (_)? It's the goal, 0 = 1.
-- exact @false.elim _ f,     -- Note that C is an implicit argument!
end

/-
We can also do case analysis on f. We will get
one case for each possible form of proof, f. As
there are no proofs of f, there are no cases at
all, and the proof is completed.
-/
example : false → 0 = 1 :=
begin
assume f,
cases f,
end
```

(continues on next page)

(continued from previous page)

```
/-
False eliminations works for "return types" in any
type universe. When the argument and return types
are both in Prop, one has an ordinary implication.
-/
example : false → nat :=
begin
assume f,
cases f,
-- contradiction,  -- this tactic works here, too
end
```

3.4.3 and ∧

From propositional logic we had three inference rules defining the meaning of *and*, one introduction and two elimination rules. These rules re-appear in both first-order predicate logic and in the higher-order logic of Lean, but now in a much richer logic. In this chapter we'll see how this is done, using *and* as an easy example.

- and_intro_rule := ⟦ X ⟧ i = tt → ⟦ Y ⟧ i = tt → ⟦(X ∧ Y)⟧ i = tt
- and_elim_left_rule := (⟦(X ∧ Y)⟧ i = tt) → (⟦X⟧ i = tt)
- and_elim_right_rule := (⟦(X ∧ Y)⟧ i = tt) → (⟦Y⟧ i = tt)

Proposition Builders

A key idea in Lean's definitions is that *and* is a *polymorphic* data type. That is to say, its akin to a function takes any two propositions (types in Prop) as arguments and yields a new Type. This new type encodes the proposition that is the conjunction of the given proposition arguments. Let's see how *and* is defined.

```
structure and (A B : Prop) : Prop :=
intro :: (left : A) (right : B)
```

The *structure* keyword is shorthand for *inductive* and can be used (only) when a type has just one constructor. The name of the constructor here is *intro*. It takes two arguments, *left*, a proof (value) of (type) *A*, and *right*, a proof of *B*.

A benefit of using the *structure* keyword is that Lean generates field access functions with the given field names. For example, if *(h : A ∧ B)*, then *(h.left : A)* and *(h.right : B)*.

Introduction: Proof Constructors

The second key idea is that the constructors of a logical type define what terms count as proofs, i.e., values of the type.

In the case of a conjunction, there is just one constructor, namely *intro,* takes two proof values as arguments and yielding a proof of the conjunction of the propositions that they prove.

Note: It's important to distinguish clearly between *and* and *intro* in your mind. The *and* connective (∧) is a proposition builder, a type builder. It takes two *propositions* (types), *(A B : Prop)* as its arguments and yields a new proposition (type) as a result, namely *(and A B),* also written as *A ∧ B*.

On the other hand, *and.intro* is a *proof/value constructor.* It takes two *proof values, (a : A)* and *(b : B)* as arguments, and yields a new proof/value/term, ⟨ *a, b* ⟩ *: A ∧ B*. There is no other way to construct a proof of a conjunction, *A ∧ B,* than to use this constructor.

Elimination: Case Analysis

Now that we've seen how to (1) construct a conjunction from two given propositions, and (2) construct a proof of one, we turn to the question, what can we *do* with such a proof if we have one.

The answer, in general, is that we can analyze how it could have been built, with an aim to show that a given proof goal follows in every case. If we can show that, then we've proved it always holds.

An already familiar example is our earlier case analysis of values of type bool. When we do case analysis on an arbitrary bool value, we have to consider the two ways (constructors) that a bool can be constructed: using the *tt* constructor or the *ff* constructor. A proof by case analysis on a bool, b, thus requires two sub-proofs: one that shows a given goal follows if *b* is *tt* and another that shows it follows if *b* is *ff*.

```
example (b : bool) :  bnot (bnot b) = b :=
begin
cases b,               -- NB: one case per constructor
repeat { apply rfl }, -- prove goal *in each case*
-- QED.                -- thus proving it in *all* cases
end
```

Turning to a proof of a conjunction, $A \wedge B$, only two small details change. First, there *and* has just one constructor. So when we do case analysis, we'll get only one case to consider. Second, the constructor now takes two arguments, rather than zero as with tt and ff. So, in that one case, we'll be entitled to assume that the two proof arguments must have been given. These will be the *left* and *right* proofs of *A* and *B* separately.

```
-- Case analysis on *proof* values
example (X Y: Prop) : X ∧ Y → X :=
begin
assume h,            -- a proof we can *use*
cases h with x y,    -- analyze each possible case
exact x,             -- also known as destructuring
end

-- We can even use "case analysis" programming notation!
example (X Y: Prop) : X ∧ Y → X
| (and.intro a b) := a
```

3.4.4 or ∨

- def or_intro_left_rule := (⟦X⟧ i = tt) → (⟦(X ∨ Y)⟧ i = tt)
- def or_intro_right_rule := (⟦Y⟧ i = tt) → (⟦(X ∨ Y)⟧ i = tt)
- def or_elim_rule := (⟦(X ∨ Y)⟧ i = tt) → (⟦(X => Z)⟧ i = tt) → (⟦(Y => Z)⟧ i = tt) → (⟦(Z)⟧ i = tt)

Just as with ∧, the ∨ connective in Lean is represented as a logical type, polymorphic in two propositional arguments.

```
inductive or (A B : Prop) : Prop
| inl (h : A) : or
| inr (h : B) : or
end hidden
```

But whereas the intended meaning of ∧ is that each of two given propositions has a proof, the intended meaning of ∨ is that *at least one of* the propositions has a proof. This difference shows up in how proofs of disjunctions are created and used.

Introduction Rules

We now have two constructors. The first, *or.inl*, constructs a proof of $A \vee B$ from a proof, (a : A). The second, *or.inr*, constructs a proof of $A \vee B$ from a proof of B.

```
-- Example using a lambda expression. Be sure you understand it.
example (A B : Prop) : A → A ∨ B := fun (a : A), or.inl a
/-
Ok, you might have notice that I've been declaring some named
arguments to the left of the : rather than giving them names
with ∀ bindings to the right. Yes, that's a thing you can do.
Also note that we *do* bind a name, *a*m to the assumed proof
of *A*, which we then use to build a proof of *A ∨ B*. That's
all there is to it.
-/
```

Elimination Rules

How do we use a proof of a conjunction, $A \vee B$? In general, what you'll want to show is that if you have a proof, h, of $A \vee B$ then you can obtain a proof of a goal proposition, let's call it C.

The proof is constructed by case analysis on h. As *(h : $A \vee B$)* (read that as *h is a proof of* $A \vee B$), there are two cases that we have to consider: *h* could be *or.inl a*, where *(a : A)*, or *h* could be *or.inr b*, where *(b : B)*.

But that's not yet enough to prove *C*. In addition, we'll need proofs that $A \to C$ and $B \to C$. In other words, to show that $A \vee B \to C$, we need to show that that true *in either case* in a case analysis of a proof of $A \vee B$. The elimination rule for $\vee$ is thus akin to what we saw in propositional logic.

```
-- or.elim : ∀ {a b c : Prop}, a ∨ b → (a → c) → (b → c) → c
-- deduce c from proofs of a ∨ b, a → c, and b → c,
#check @or.elim

example (P Q R : Prop) : P ∨ Q → (P → R) → (Q → R) → R
| (or.inl p) pr qr := pr p
| (or.inr q) pr qr := qr q
```

Examples

```
example : ∀ P Q, P ∨ Q → Q ∨ P :=
begin
assume P Q h,
cases h with p q,
exact or.inr p,
exact or.inl q,
end


example : ∀ P Q R, P ∨ (Q ∨ R) → (P ∨ Q) ∨ R :=
begin
assume P Q R h,
cases h with p qr,
left; left; assumption,
cases qr with q r,
left; exact or.inr q,
```

(continues on next page)

(continued from previous page)

```
right; assumption,
end
```

3.4.5 not (¬)

```
-- ¬¬X ⊢ X                    -- negation elimination
-- X → ⊥ ⊢ ¬X                 -- negation introduction
```

Introduction

We saw in propositional logic that if $X \to$ *false* then X must be false. That's easy to see: *true* $\to$ *false* is false, so X can't be *true*. On the other hand, *false* $\to$ *false* is true. X can only be *false*. Now, saying X *is false* in propositional logic is equivalent to saying ¬X is true, giving us our constructive ¬ introduction rule: X → ⊥ ⊢ ¬X. This is the rule of ¬ introduction: to prove $\neg X$ it will suffice to prove $X \to$ *false*.

The same idea reappears in the constructive logic of Lean. In fact, Lean simply *defines* $\neg X$ to mean $X \to$ *false*.

```
-- def not (a : Prop) := a → false
-- prefix `¬`:40 := not
#check not

example : ¬ 0 = 1 :=
begin
show 0 = 1 → false,
assume h,
contradiction,
end
```

Let's think about what $a \to$ *false* means, where a is any proposition. In Lean, a proof of an implication is a function, namely one that would turn *any* proof of a into a proof of false. So, *if there were* a proof of a then one could have a proof of *false*. That can't happen because there is no proof of false. So there must be no proof of a. Therefore a is false, and we can write that as $\neg a$.

To prove a proposition, $\neg a$, we thus just have to prove that $a \to$ *false*. To do this, we assume we have a proof of a and show that that leads to an impossibility, which shows that the assumption was wrong, thus $\neg a$ must be true. You can pronounce $\neg a$ as *not a* or *a is false*, but it can also help to think of it as saying *there provably can be no proof of a.*

```
example : 0 = 1 → false :=
begin
assume h,
cases h,
end

example : ¬ 0 = 1 :=
begin
assume h,
cases h,
end

example : 0 ≠ 1 :=
begin
assume h,
```

(continues on next page)

(continued from previous page)

```
cases h,
end
```

Elimination

In propositional logic, we have the rule of (double) negation elimination: $\neg\neg X \rightarrow X$. An easy way to think about this is that two negations cancel out: negation is an involution. As we'll now see, this rule is also defines proof by contradiction.

To see that, one can read the rule as saying that to prove X it will suffice to assume $\neg X$ and show that that leads to a a contradiction, thus proving that $\neg X$ is false, thus $\neg\neg X$. From there in classical logic it's just a final step to X.

Is this inference rule valid in Lean? Let's try to prove that it is. Our goal is to prove $\forall$ X, $\neg\neg$X $\rightarrow$ X. We first rewrite the inner $\neg X$ on the left of the $\rightarrow$ as *(X $\rightarrow$ false)*. $\neg\neg X$ becomes $\neg$*(X $\rightarrow$ false)*. Rewriting again gives *(X $\rightarrow$ false)* $\rightarrow$ *false.* Our goal, then, is to prove ((X $\rightarrow$ false) $\rightarrow$ false) $\rightarrow$ X. We get a start by assuming (h : (X $\rightarrow$ false) $\rightarrow$ false), but then find that from *h* there's no way to squeeze out a proof of *X*.

```
example : ∀ (X : Prop), ¬¬X → X :=
begin
assume X h,
-- can't do case analysis on a function
cases h,
-- we're stuck with nowhere left to go!
end
```

What we've found then is that (double) negation elimination, and thus proof by contradiction, is not valid in Lean (or in similar constructive logic proof assistants). This shows that in Lean a proposition, *X*, being proved *not false* ($\neg\neg X$) does not imply that *X* is true. From a proof of the former we can't obtain a proof of the latter. From a proof that $\neg X$ *is false* we have no way to derive a proof of *X*.

Constructive vs. Classical

In constructive logic, a proposition can thus be provably true (by a proof), it can be provably false (by a proof that there is no proof of it), or it can be provably not false, which is to say that there must exist a proof, but where one cannot be constructed from the given premise alone.

We'll see the the same constructivity requirement again when we discuss proofs of existence. In a nutshell, a constructive proof exhibits a specific "witness" to show that one does exist. To be constructive means that a proof of existence requires an actual witness.

With respect to negation elimination, it's not enough to know that there's an unspecified proof of *X* "out there." To know that *X* is true/valid, one has to *exhibit* such a proof: to have one in hand, that can actually be inspected and verified.

Consider *em* again. Given any proposition, *X*, *(em X)* is a proof of $X \vee \neg X$. Now consider the introduction rules for $\vee$ in constructive logic: to construct a proof of *X* you have to have either a proof of *X* or a proof of $\neg X$ in hand. The *em* axiom gives you a proof of $X \vee \neg X$ without requiring a proof of either disjunction. It's thus non-constructive.

Classical logic and mathematics do not adopt the constraint of constructivity, and consequently there are theorems that can be proved in classical mathematics but not in constructive mathematics. Again, it's easy to turn Lean classical simply by opening the classical namespace (the accepted signal that one will admit classical reasoning) and using *em* in your proofs.

```
-- A proof of 0 = 0 by contradition
example : 0 = 0 :=
begin
```

(continues on next page)

(continued from previous page)

```
by_contradiction, -- applies ¬¬P → P
have eq0 := eq.refl 0,
contradiction,
end
```

Excluded Middle

We've seen that in *constructive* logic, knowing that it's false that there's no proof is not the same as, and is weaker than, actually having a proof. Knowing that a proposition is not false is not the same as actually having a proof in hand. What makes constructive logic constructive is that a proof is required to judge a proposition as being true.

In classical predicate and propositional logic, by contrast, negation elimination is an axiom. To prove a proposition, X, by contradiction, one assumes $\neg X$, shows that from that one can derive a contradiction, thus proving $\neg\neg X$, and from there (here it comes) by negation elimination one finally concludes X, thereby satisfying the original goal.

In summary, negation elimination is not an axiom in constructive logic, so any proof that relies on $\forall\ X,\ \neg\neg X \rightarrow X$ gets blocked at this point. The reason it's not an axiom is that it would make the logic non-constructive: while a proof that X is not false might suggest that there exists a proof of X, it does not give you such a proof, which is what constructive logic requires.

In constructive logic there are not just two truth states for any proposition. We've seen that we can know that a proposition is true (by having a proof of it), know that it is false (by having a proof it entails a contradiction), and know that it's not false but without having constructed a proof that it's true. We can know that something is not false without having a proof of it, and without a proof we can't judge it to be true, either.

In classical logic, the *axiom ("law") of the excluded middle* rules out this middle possibility, declaring as an assumption that for any given proposition, P, there is a proof of $P \vee \neg P$.

This axiom then enables proof by contradiction. That's an easy proof. We need to prove that if $P \vee \neg P$ then $\neg\neg P \rightarrow P$. The proof is by case analysis on an assumped proof of $P \vee \neg P$. In the first case, we assume a proof of P, so the implication is true trivially. In the case where we have a proof of $\neg$P, we have a contradiction between $\neg$P and $\neg\neg$P, and so this case can't actually arise and needn't be considered any further.

In Lean, you can declare anything you want to be an additional axiom. If you're careless, you will make the logic inconsistent and thus useless. For example, don't assume $true \leftrightarrow false$. On the other hand, you may add any axiom that is independent of the given ones and you'll still have a consistent logic in which propositions that lack proofs in constructive logic now have proofs.

The law of the excluded middle, $\forall\ P,\ P \vee \neg P$ is declared as an *axiom* in the *classical* namespace, where *classical.em* (or just *em* if you *open classical*) is assumed to be a proof of $em : \forall\ P,\ P \vee \neg P$.

Now the key to understanding the power of excluded middle is that it allows you to do case analysis on any *proposition* in our otherwise constructive logic. How's that? Assume X is an arbitrary proposition. Then *(em X)* is a proof of $X \vee \neg X$.

Note that *em*, being universally quantified is essentially a function. It can be *applied* to yield a specific instance of the general rule. Ok, so what can we do with a "free proof" of $X \vee \neg X$? Case analysis! There will be just two cases. In the first case, we'd have a proof of X. In the second, we'd have a proof of $\neg X$. And those are the only cases that need to be considered.

Examples

```
#check @classical.em

theorem foo : ∀ P, (P ∨ ¬ P) → (¬¬P → P) :=
begin
assume P,
assume em,
assume notNotP,
cases em,
-- case 1
assumption,
contradiction,
end

example : 0 = 0 :=
begin
by_contradiction,
have zez := eq.refl 0,
contradiction,
end

theorem demorgan1 : ∀ P Q, ¬(P ∧ Q) ↔ (¬P ∨ ¬ Q) :=
begin
assume P Q,
--apply iff.intro _ _,
split,

-- FORWARD
assume h,


have ponp := classical.em P,
have qonq := classical.em Q,

cases ponp with p np,
cases qonq with q nq,
have pandq := and.intro p q,
contradiction,

apply or.inr nq,
apply or.inl np,

-- BACKWARDS
assume h,
cases h,

assume pandq,
have p := and.elim_left pandq,
-- cases pandq with p q,
contradiction,

assume pandq,
cases pandq with p q,
contradiction,

end
```

(continues on next page)

(continued from previous page)

```
example : ∀ (P : Prop), ¬ (P ∧ ¬P) :=
begin
assume P,
assume h,
cases h with p np,
apply false.elim (np p),
end

example : ∀ P Q R, P ∨ (Q ∧ R) → (P ∨ Q) ∧ (P ∨ R) :=
begin
assume P Q R,
assume h,
apply and.intro _ _,

cases h with p qandr,
exact or.inl p,
cases qandr with q r,
apply or.inr q,
cases h,
exact or.inl h,
cases h,
exact (or.inr h_right)

end
```

Exercises

1. Give a formal proof of the claim that excluded middle implies proof by contradiction.
2. Determine whether, and if so prove, that the two statements are equivalent: excluded middle and proof by contradiction.
3. Try to Prove each of DeMorgan's laws in Lean to identify the non-constructive ones
4. Finish the proofs of DeMorgan's laws using the axiom of the excluded middle *(em)*.

3.4.6 iff ↔

```
-- 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
```

We want $P \leftrightarrow Q$ to be equivalent to $(P \rightarrow Q) \wedge (Q \rightarrow P)$.

The introduction rule, *iff.intro,* like *and.intro*, thus takes two implications proofs, one in each direction and yields a proof of the biimplication.

The *iff* left and right elimination rules are similarly akin to those for *and*: from a proof, $h : P \leftrightarrow Q$, they derive proofs of the respective forwards and backwards implications, $P \rightarrow Q$ and $Q \rightarrow P$. Such proofs are functions, and thus can be applied to arguments in subsequent proofs steps.

The names of the *iff* left and right elimination rules in Lean are *iff.mp* and *iff.mpr.* Case analysis on a proof, $h : P \leftrightarrow Q$, using the *cases* tactic, will derive proofs of both implications at once.

```
example (P Q : Prop) : (P ↔ Q) ↔ ((P → Q) ∧ (Q → P)) :=
begin
split,

-- FORWARD
assume piffq,
cases piffq with pq qp,
exact and.intro pq qp,

-- BACKWARD
assume pqqp,
cases pqqp with pq qp,
exact iff.intro pq qp,
end
```

3.4.7 Conclusion

In this section you've encountered analogs in higher-order logic of the reasoning principles that you saw (in weaker forms) in propositional logic, but now as rules for and expressed in a higher-order predicae logic itself embedded in the higher-order logic of Lean.

Just as we ourselves specified an embedding of propositional logic in Lean, so the Lean library authors have given us a higher-order predicate logic embedded in Lean. We have already met propositions as types in Prop, functions, predicates as functions to Prop. We know the inference rules.

But what we've not yet met formally are the quantifiers of predicate logic (higher-order or not), $\forall$ and $\exists$. In Lean, these quantifiers are represented using *dependent types*. In the next chapter we take up these topics,

CHAPTER

FOUR

RECURSIVE TYPES

4.1 Introduction

In this chapter we'll look at inductive data types definitions that specify objects that have recursive structure. In particular, we'll look at the *nat* and polymorphic *list* types, with an eye to seeing some interesting algebraic commonalities.

4.2 Natural Numbers

4.2.1 Data Type

```
#check nat
```

```
-- notations for writing succ applications
def three'  := (nat.succ (nat.succ (nat.succ (nat.zero))))
def three  := nat.zero.succ.succ.succ
```

4.2.2 Operations

Having seen how the *nat* data type is defined, we now look at how to define functions taking *nat* values as arguments. As we've seen before, many such functions here will again be defined by case analysis on an incoming nat argument value. That means considering two cases separately: the incoming value is either zero or non-zero: that is, either *nat.zero,* or *nat.succ n'* for some "one-smaller" value, *n'*. For example, if the incoming argument is *succ(succ(succ zero))*, i.e., 3, then (a) it does not match *nat.zero*, but (b) it does match *nat.succ n'*, with *n'* is bound to *succ(succ zero)*, i.e., 2.

```
-- increment is just succ application
def inc (n' : nat) : nat := n'.succ
def three'' := inc(inc(inc nat.zero))
```

A predecessor (one less than) function can be defined by case analysis on a nat argument. Here we'll define *pred'* to return 0 when applied to 0, and otherwise to return the one smaller value, *n'*, when applied to any non-zero value, *nat.succ n'*.

Rather than "implementing a function" think "proving a function type." A "proof" of function type, *nat* → *nat,* is any function that converts any given nat into some resulting nat.

When proving a proposition (a type in Prop), any proof (value of that type) will do. When proving function or other data type, however, the particular value of the type that you construct is usually important. Here, for example, we don't want any function that takes and returns a nat, but one that returns the right nat for the given argument.

```
def pred' : nat → nat :=
begin
assume n,
cases n with n',
exact 0,  -- when n is zero
exact n', -- when n is succ n'
end

-- quick test
#eval pred' 6
example : pred' 6 = 5 := rfl
```

Here's the same function just specified using pattern matching notation (which, as we've seen generalizes case analysis).

```
def pred : nat → nat
| nat.zero := nat.zero  -- loop at zero
| (nat.succ n') := n'

#eval pred 5
example : pred 5 = 4 := rfl
```

Pattern matching generalizes case analysis by giving you a means to return different results based on deeper analysis of argument structures using pattern matching/unification. This example implements subtract-two, looping at zero. Notice how the third pattern matches to the sub-natural-number object nested two succ-levels deep.

```
-- this example illustrates pattern matching
-- for more fine-grained case analysis
def sub2 : nat → nat
| nat.zero := nat.zero
| (nat.succ nat.zero) := nat.zero
| (nat.succ (nat.succ n')) := n'

-- addition increments the second argument
-- the first argument number of times
def plus : nat → nat → nat
| nat.zero m := m
| (nat.succ n') m := nat.succ (plus n' m)

#eval plus 3 4

-- multiplication adds the second argument
-- to itself the first argumen number of times
def times : nat → nat → nat
| 0 m := 0
| (n'+1) m := plus m (times n' m)

#eval times 5 4
#eval times 1 20


-- substraction illustrates case analysis on
-- multiple (here two) arguments at once
def subtract :  nat → nat → nat
| 0 _ := 0
| n 0 := n
| (n' + 1) (m' + 1) := subtract n' m'
```

(continues on next page)

(continued from previous page)

```
#eval subtract 7 5
#eval subtract 7 0
#eval subtract 5 7
#eval subtract 0 7


-- exponentiation is multiplication of the second
-- argument by itself the first argument number of times
def power : nat → nat → nat
| n nat.zero := 1
| n (nat.succ m') := times n (power n m')

-- a few test cases
example : power 2 0 = 1 := rfl
example : power 2 8 = 256 := rfl
example : power 2 10 = 1024 := rfl
```

4.3 Polymorphic Lists

4.3.1 Data Type

```
#check list
```

The list data type is surprising similar to the nat data type. Where as a larger nat is constructed from only a smaller nat, a larger list is constructed from a new first element (the *head* of the new list) and a smaller list (the *tail* of the new list). This type builder enables us to represent lists of values of any type and of any finite length.

```
universe u
inductive list (T : Type u)
| nil : list
| cons (hd : T) (tl : list) : list
```

```
-- example: let's represent the list of nats, [1,2,3]
def three_list_nat'' :=
  list.cons    -- takes two arguments
    1          -- head of new list
    (          -- tail list of the new list
      list.cons    -- etc.
        2
        (
          list.cons
            3
            list.nil
        )
    )

-- it seems to have worked
#reduce three_list_nat''
```

4.3.2 Notations

```
-- notation, :: is infix for cons
-- [] notation adds nil at end
def three_list_nat''' := 1::2::3::list.nil
def three_list_nat'''' := [1,2,3]
def four_list_nat := 0::[1,2,3]       -- fun!
```

4.3.3 Operations

```
-- list prepend analogous to nat increment
def prepend' (α : Type) (a : α) (l : list α) :=
  list.cons a l

def three_list_nat' :=
  prepend' nat
    1
    (prepend' nat
      2
      (prepend' nat
        3
        list.nil
      )
    )

#eval three_list_nat'

-- here with an implicit type parameter, making it equivalent to cons
def prepend {α : Type} (a : α) (l : list α) :=
  list.cons a l

def three_list_nat :=
  prepend
    1
    (prepend
      2
      (prepend
        3
        list.nil
      )
    )

-- okay, that looks good
-- but know that to which it desugars

example := prepend' nat 2 [3,4,5]
#eval prepend' nat 2 [3,4,5]

example := prepend 2 [3,4,5]
#eval prepend 2 [3,4,5]

#eval 2::[3,4,5]
```

4.3.4 Partial Functions

Now we face some interesting issues. Our aim is to define functions that *analyze* lists and return parts of them. The problem is that there are no parts when a given list is empty.

When we defined pred, above, we defined pred of zero to be zero (rather than to be undefined). Doing that makes the function total and easily represented as a function (lambda abstractraction) in Lean. However, in a different application we really might want to define pred 0 to be undefined, not 0.

A similar set of issues arises when we consider head and tail functions on lists. When given non-empty lists there is no problem. But what to do with an empty list argument? There is no head or tail element to return, yet some value of the specified type *has to be* returned.

Let's see some fo the solutions that are available.

Default Value

```
-- a version of tail that "loops at zero"
def tail' : ∀ {α : Type}, list α → list α
| α list.nil := list.nil
| α (h::t) := t
#eval tail' [1,2,3,4,5]
```

One nice thing about this solution is that the function type is still about as natural as can be: list $\alpha \rightarrow$ list α.

Option Values

The next solution changes the type of the function, so that return value is in the form of a *variant* type, a value of which is either *none* or *some valid return value*.

```
def head'' : ∀ {α : Type}, list α → option α
| α list.nil := none
| α (h::t) := some h

#eval head'' [1,2,3]
#eval @head'' nat []
```

Precondition

Finally, we can define a version of head' that (1) typechecks as being a total function, (2) can never actually be applied fully to an empty list, in which case (3) no real result has to be specified to "prove the return type" because such a case can't happen. It would be a contradiction if it did, and so it can be dismissed as an impossibility. Magic: It *is* a total function, but it can never be fully appied to an empty list because a required proof argument, for *that* list, can never be given; so one can dismiss this case by false elimination, without having to give an actual proof of the conclusion.

Consider a head function. It returns the head element from a non-empty list, but is undefined mathematically when it's applied to an empty list. The key idea in the next design is that we can embed a *precondition* for application of the function, namely that the given list not be empty. Let's see how e might first write the function using a tactic script, to take advantage of your familiarity with using it to build proofs.

```
def head' : ∀ {α : Type} (l : list α), (l ≠ list.nil) → α
|  α l p :=
begin
cases l,
```

(continues on next page)

(continued from previous page)

```
contradiction,
exact l_hd,
end

-- When applying it a proof about the first argument has to be given
#eval head' [1,2,3] begin contradiction end    -- proof as a proof script
#eval head' [1,2,3] (by contradiction)         -- alternative syntax, fyi
#eval head' ([] : list nat) _                  -- you'll need a proof of list.nil ≠␣
↪list.nil!
```

4.3.5 Exercises

- Write a version of the pred function that can only be called for argument values greater than 0.
- Write a version of the pred function that returns an option nat value "in the usual way"
- Write a tail function that can only be called with a non-empty list, using our "by cases" notation for function definition. It should look like tail'. Note 1: Where a proof value is required, you can always use tactic mode to construct the required proof, in a begin..end block. If such a proof is a single tactic long, you can write by <tactic>. For example, try by contradiction as the *result* when your new tail function is applied to an empty list. Here's how I wrote the function type. You should provide the cases (on l). Here's the type: def tail $\{\alpha : \text{Type}\} : \forall$ (l : list α), (l $\neq$ list.nil) $\rightarrow$ list α.

```
-- implement the function, no need to (do not try) to match on α
-- it's named before the colon and is global to this definition
-- we do want to match (do case analysis) on l, so it's after :
-- def tail {α : Type} : ∀ (l : list α), (l ≠ list.nil) → list α
-- |
-- |
```

4.3.6 Solutions

```
-- let's implement a "safe" pred function using tactics
def pred' : ∀ (n : nat), (n ≠ nat.zero) → ℕ :=
begin
assume n,
cases n with n',
assume h,
contradiction,
assume h,
exact n',
end

#reduce pred' 5 _
#reduce pred' 2 _
#reduce pred' 0 _

-- here's the same predecessor function presented differently
def pred'' : ∀ (n : nat), (n ≠ nat.zero) → ℕ
| nat.zero h := by contradiction
| (nat.succ n') h := n'

-- a different safe predecessor function using an option return
```

(continues on next page)

(continued from previous page)

```
def pred''' : nat → option nat
| nat.zero := option.none
| (nat.succ n') := some n'

-- the same ideas work for safe head and tail functions on lists
universe u
def tail : ∀ {α : Type u} (l : list α), (l ≠ list.nil) → list α
| α list.nil p := by contradiction
| α (h::t) p := t

-- apply tail to [1,2,3] giving the proof as a tactic script
#eval tail [1,2,3]
begin
assume p,
contradiction,
end

-- cleaner this way
#eval tail [1,2,3] (by contradiction)
#eval tail [2,3] (by contradiction)
#eval tail [3] (by contradiction)
#eval @tail nat [] (by contradiction)      -- no can do!

-- let's try it with a tactic script
#eval @tail nat []
begin
assume h,   -- we're stuck, and that's good!
end


-- append: the list analog of natural number addition
-- please do compare/contrast list.append and nat.add
def appnd {α : Type} : list α → list α → list α
| list.nil m := m
| (h::t) m := h::appnd t m

#eval appnd [1,2,3] [4,3,2]
```

4.4 Higher-Order Functions

A higher-order function is simply a function that takes functions as arguments and/or that returns a function as a result.

4.4.1 In Logic

We've already seen this idea in logical reasoning, where function values are proofs of implications. In this chapter, we'll see that same idea in the realm of computation.

Let's start by reviewing a logical example to refresh memories. We'll review the proof that *implication is transitive*: if the truth of some proposition, P, implies the truth of Q, and if the truth of Q implies the truth of R, then the truth of P implies that of R. Thinking computationally, if we have a function, pq, that converts any proof of P into a proof of Q (a proof of P → Q), and a function, qr, that converts any proof of Q into a proof of R (a proof of Q → R), then we can build a function, pr, that converts any proof, p, of P, into a proof of R (the desired proof of P → R) by applying the proof

of P → Q to p to get a proof of Q, and by then applying the proof of Q → R to that value to get a proof of R. Here it is formally.

```
example {P Q R : Prop} : (P → Q) → (Q → R) → (P → R) :=
begin
assume pq qr,    -- assume P → Q and Q → R
assume p,        -- to show P → R, assume p a proof of P
exact qr (pq p) -- and derive the desired proof of R
end
```

This proof is a higher-order function, albeit in the realm of logic not computation with ordinary data. It takes two function arguments (one proving of P → Q and the second proving Q → R) and returns a function that, by converting any proof of P into a proof of R, proves P → R. Therefore, (P → Q) → (Q → R) → (P → R). That is, *implication is transitive*.

4.4.2 Composition

What do we get when we construct the same argument not for proofs of logical propositions but for functions on ordinary data? What we get is a higher-order function that performs *function composition*. Note the change from Prop (logic) to Type (computation) in the following definition.

```
example {α β γ : Type} : (α → β) → (β → γ) → (α → γ) :=
begin
assume αβ βγ,    -- assume f g
assume a : α,    -- assume a
exact βγ (αβ a) -- return λ a, g (f a)
end
```

Compare and contrast this definition with the statement and proof of the transitivity of implication. See that you've already been using higher-order functions albeit to reason with functions that serve as proofs of logical implications, rather than with with functions on ordinary data.

Let's write this definition a little more naturally, and give it a name: *comp*, short for *composition*.

```
def comp {α β γ : Type} (f : α → β) (g : β → γ) : α → γ :=
fun (a : α), g (f a)
```

Example

Let's see an example. Suppose we have two functions, *inc* that increments a natural number and sqr that squares one. We can form a function that first increments then squares its argument by *composing* these two functions.

```
def inc (n : ℕ) := n + 1
def sqr (n : ℕ) := n * n
def inc_then_sqr := comp inc sqr
example : inc_then_sqr 5 = 36 := rfl    -- seems to work!
```

Notation

Lean defines the infix operator ∘ as notation for function composition. Note that the order of the function arguments is reversed. (g ∘ f) is the function that applies g after applying f to its argument. That is, (g ∘ f) x = g (f x). We pronounce the function, (g ∘ f), as *g after f*.

```
def inc_then_sqr' := sqr ∘ inc        -- composition!
example : inc_then_sqr' 5 = 36 := rfl -- seems to work!
```

Example With Two Types

In this example, given functions that compute the length of a list and decrement a natural number, we construct a function that takes a list of objects and returns one less than its length. We first illustrate applications of Lean functions for length and decrement and then use both our notation and the Lean ∘ notation to construct the desired function, which we apply to the list [1,2,3] yielding the value, 2.

```
#eval list.length [1,2,3] -- apply length function to list
#eval [1,2,3].length      -- function application notation
#eval nat.pred 3          -- apply decrement function to 3

-- Apply composition of length and pred to list
#eval (comp list.length nat.pred) [1,2,3]
#eval (nat.pred ∘ list.length) [1,2,3]
```

The infix notation is best. Think of the argument, here the list [1,2,3], as moving left through list.length, yielding 3, which then moves left through nat.pred, finally yielding 2.

4.4.3 Map

In this section, we introduce the *map* function on lists. It takes (1) a function that takes objects of some type α and converts them into objects of some type β, and (2) a list of objects of type α, and returns a list of objects of type β, obtained by using the function to turn each each α object in the given list into a corresponding β object in the resulting list.

We build to a general definition of map starting with a special case: of a function that takes a list of natural numbers and returns a list in which each is increased by one, by the application of *inc*, our increment function.

We define a function that "maps" the increment function over a given list of natural numbers by case analysis on any given list. If the given list is nil, we return nil; otherwise, if the list is (h::t) we return the list with the value of (inc h) at its head and the list obtained by similarly incrementing each value in the tail of the given list as its tail.

```
def inc_list_nat : list nat → list nat
| list.nil := list.nil
| (h::t) := (inc h)::inc_list_nat t

-- it works
#eval inc_list_nat[]        -- expect []
#eval inc_list_nat [1,2,3]  -- expect [2,3,4]
```

Suppose that instead of incrementing each element of a given list to obtain a new list, we want to square each element. One way to do it is to clone the function above and replace inc with sqr.

```
def sqr_list_nat : list nat → list nat
| list.nil := list.nil
| (h::t) := (sqr h)::sqr_list_nat t
```

(continues on next page)

(continued from previous page)

```
-- It works
#eval sqr_list_nat [1,2,3,4,5]
```

Clearly we can clone and edit the preceding code to produce a version that applies *any* function of type nat → nat, instead of inc or sqr, to the head of the given list, with all of the remaining code unchanged, to map given lists of natural numbers to new lists by replacement of existing elements with new elements computed by application of the given function.

That all the code remains the same but for the *element* converting function suggests that we can instead *generalize* by making this function a *parameter* of the otherwise unchanging code.

```
def any_list_nat : (nat → nat) → list nat → list nat
| f list.nil := list.nil
| f (h::t) := f h::any_list_nat f t

-- It seems to work!
example : any_list_nat sqr [1,2,3,4,5] = [1,4,9,16,25] := rfl
example : any_list_nat inc [1,2,3,4,5] = [2,3,4,5,6] := rfl
example : any_list_nat nat.pred [1,2,3,4,5] = [0,1,2,3,4] := rfl
```

We've generalized the nat → nat function, but suppose we wanted to convert a list of *strings* to a list of their natural number lengths. We don't have the machinery to do that yet, as we can only map functions over lists of natural numbers. Otherwise we get a type error.

```
#eval any_list_nat string.length ["I", "Love", "Math"]  -- nope!
```

One solution is simply to write a new version of our mapping function specialized to map lists of strings to lists of nat values, using any given string → nat function to perform the element-wise mapping.

```
def xyz_list_nat : (string → nat) → list string → list nat
| f list.nil := list.nil
| f (h::t) := f h::xyz_list_nat f t

-- It seems to work
#eval xyz_list_nat string.length ["I", "Love", "Math"]
```

But we run into the same problem as before if we now want to map lists of strings to Boolean values, e.g., reflecting whether the length of each string is even (tt) or not (ff). Cloning code and editing it to produce another special case is really not the best solution.

```
def map_string_bool : (string → bool) → list string → list bool
| f list.nil := list.nil
| f (h::t) := f h::map_string_bool f t

-- is_even takes a nat and return tt if it's even else ff
def is_even (n : nat) : bool := n % 2 = 0
#eval is_even 2
#eval is_even 3
```

Now we can map a function that tells whether a given string is of even length or not over any given list of strings to get a corresponding list of tt/ff values.

```
def is_even_length := is_even ∘ string.length
#eval map_string_bool is_even_length ["I", "Love", "Math"]
```

Of course well run into exactly the same sort of problem, of having to engage in error-prone cloning and editing of code, if we want to now map lists of Boolean values to lists of strings (e.g., mapping each tt to "T" and each ff to "F").

And you can imagine many other examples: mapping lists of employees to list of their corresponding salaries, or mapping lists of Boolean values to lists of their negations, etc. The possibilities are endless.

The answer should now we pretty clear: we need to further generalize: not only over the function to apply to map each list element, but also over the the types of element in the input and output lists! Here, then, is a greatly generalized version.

```
def map_list {α β : Type} : (α → β) → list α → list β
| f list.nil := list.nil
| f (h::t) := f h :: map_list f t

-- It seems to work!
#eval map_list nat.succ [1,2,3]
#eval map_list is_even_length ["I", "Love", "Math"]
```

For now, we'll be satisfied with this level of generality. We will just observe that our mapping function still only works for *lists* as element containers. What if you wanted to map functions over other kinds of element "containers," e.g., to turn values of type *option* α into *option* βs? Or trees of α values into corresponding trees of β values?

The key roadblock will be that there's no way to do this using exactly the same code for, say, lists and options. So the kind of parametric polymorphism we've been using will no longer be enough. The answer will be found in a different kind of polymorphism, *ad hoc* polynorphism, of which *operator overloading* (as in C++) is an example. For instance, you can write complex number and string classes and overload the + operator in each class to do respective complex number addition and string append, but the implementations of these operations will hardly share the same code. Completely different implementations will be needed, to be selected (by the compler in C++) based on the types of the arguments to which the + operator is applied. More on this topic later.

4.4.4 Fold: list $\alpha \to \alpha$

We now turn to a very different higher-order function appliable to lists. It's called *fold* (or event better, *fold_right*) or *reduce*.

Overview

The fundamental purpose of this operation is to turn a *binary* operation on the values of any given type (e.g., nat) into an operation that can be applied to *any* number of arguments, where the arguments are packaged into a list data structure.

The way the generalized version of the binary operation works is that for the empty list it returns a base value, and for a non-empty list, *h::t*, it applies the binary operation to *h* and *to the result of applying the n-ary version to the rest of the list, *t*.*

As an example, fold will turn the addition function on natural numbers (nat.add) into an operation that can be applied to a list of any number of natural number values to compute the sum of them all. Here, for example, is such a program.

```
def reduce_sum : list nat → nat
| list.nil := 0
| (h::t) := nat.add h (reduce_sum t)

#eval reduce_sum []           -- sum of zero arguments
#eval reduce_sum [5]          -- sum of one argument
#eval reduce_sum [5,4]        -- sum of two arguments
#eval reduce_sum [5,4,3,2,1]  -- sum of five arguments
```

Generalizing the binary operation

It should be clear that we will want to generalize the binary operator from nat.add to *any* binary operation on natural numbers. For example, we might want a function that implements n-ary multiplication, reducing any list of natural numbers to the product of all the numbers in the list.

This is a little bit tricker than one might guess. To see the problem, let's clone and edit the code we've got, substituting multiplication for addition, in an attempt to implement n-ary multiplication.

```
def reduce_prod' : list nat → nat
| list.nil := 0
| (h::t) := nat.mul h (reduce_prod' t)

#eval reduce_prod' [3,2,1]    -- expect 6 got 0!
```

Operator-identity inconsistency

To see what goes wrong, let's unroll the recursion:

- reduce_prod' [3,2,1] =
- mul 3 (reduce_prod' [2,1]) =
- mul 3 (mul 2 (reduce_prod' [1])) =
- mul 3 (mul 2 (mul 1 (reduce_prod' []))) =
- mul 3 (mul 2 (mul 1 0)) = 0!

The problem is now clear, and so is the solution: we need to return a different value for the base case of an empty list when the binary operation is multiplication rather than addition. Specifically, we need to return 1 rather than zero. You can now probably guess that in general we want to return the *identity, or neutral, value* for whatever the binary operator is for the base case. Here we want to return 1.

```
def reduce_prod : list nat → nat
| list.nil := 1
| (h::t) := nat.mul h (reduce_prod t)

#eval reduce_prod []           -- expect 1
#eval reduce_prod [5,4,3,2,1] -- expect 120
```

So now we can correctly generalize fold_nat over binary operators by making the operator a parameter but by also adding as a second parameter the right identity element for whatever operator we provide as an actual parameter.

```
def fold_nat (op : nat → nat → nat):  nat → list nat → nat
| id list.nil := id
| id (h::t) := op h (fold_nat id t)


-- It seems to work!
#eval fold_nat nat.add 0 [1,2,3,4,5]  -- expect 15
#eval fold_nat nat.mul 1 [1,2,3,4,5]  -- expect 120
```

Enforcing op-id consistency

Yet a problem remains. There is nothing in our solution that prevents us from passing the wrong value for the identity element for the given binary operator. The following function application runs without any errors being reported but it gives the wrong answer, because we pass the wrong identity element for nat.mul.

```
#eval fold_nat nat.mul 0 [1,2,3]  -- oops, wrong
```

We finish this section with a step toward our ultimate solution: we will now construct a version of fold_nat (fold_nat') that *enforces consistency* between the binary function and identity element arguments by requiring, as an additional argument, a proof that the putative identity element really is one!

In particular, we'll be satisfied for now to prove that the given "identity" element really is a *right* identity. That is, we want to prove that for any n, op n id (with id on the right) is equal to n. That is, we'll require a proof of $\forall$ *(n : nat), op n id = n* as an argument to our function.

This is not our complete solution to the problem of enforcing operator-identity consistency, but we'll have to wait until after the next chapter to get to that point.

```
def fold_nat'
  (op: nat → nat → nat)
  (id :nat)
  (right_id : ∀ (n : nat), op n id = n) :
  list nat → nat
| list.nil := id
| (h::t) := op h (fold_nat' t)
```

Let's construct named proofs that 0 is an identity when it appears as the second argument to nat.add.

```
theorem zero_right_id_add : ∀ (n : nat), nat.add n 0 = n :=
begin
assume n,
simp [nat.add]
end

-- Now we can safely use fold_nat'
#eval fold_nat' nat.add 0 zero_right_id_add [1,2,3] -- good
#eval fold_nat' nat.add 1 zero_right_id_add [1,2,3] -- not good
```

From binary to n-ary operations

We now circle back to the notion that fold generalizes any given binary operator to an n-ary operator applicable to any number of arguments as long as they're arranged in a list. You can see this idea in action by just partially applying fold_nat' to a binary operator, it's identity, and the required proof, leaving the list argument TBD.

```
def n_ary_add := fold_nat' nat.add 0 zero_right_id_add

-- It seems to work!
#eval n_ary_add []            -- zero arguments
#eval n_ary_add [5]           -- one argument
#eval n_ary_add [4,5]         -- two arguments
#eval n_ary_add [1,2,3,4,5]   -- five arguments, etc!
```

Soon we'll be able similarly to turn binary multiplication into n-ary multiplication, with a definitions like this: *def n_ary_mul := fold_nat' nat.mul 1 one_right_id_mul*. The problem is we don't yet have the machinery (namely proof by induction) to construct the proof that 1 is a right identity for nat.mul. That'll come soon enough. For now, we can stub it out and get something that works but without a proof that 1 is a right identity for natural number multiplication.

```
def n_ary_mul := fold_nat' nat.mul 1 sorry
#eval n_ary_mul [1,2,3,4,5]
```

It's a good time for a few exercises to prepare for the rest of what's to come.

4.4.5 Exercises

1. Write a function, n_ary_append (without using fold) that takes a list of lists of objects of some type, α (the type will be *list (list* α*)*) and that reduces it to a single list of α using *list.append* as a binary operation. For example, it'd turn this list, [[1,2],[3,4],[5]] into the list [1,2,3,4,5]. You may use Lean's list.append function as a binary operator that combines two lists into one.

2. Write a function (without using fold) that takes a a list of lists of α and that returns the sum of the lengths of the contained lists. For example applying your function to the list, [[],[1,2,3],[1,2,3,4,5]], should return 8: the sum of 0 for the first list, 3 for the second, and 5 for the third. Your function will work by adding the length of the head of the list of lists to the result of recursively reducing the *rest* (tail) of the list of lists. You may use list.length to compute the length of any list.

3. Write a function without using fold that takes a list of lists of α and that returns true if the length of each of the elements lists is even and false otherwise.

```
-- Problem #1
/-
Let's do some test-driven development here.
(1) Define function type
(2) Write (initially failing) test cases
(3) Complete implementation and expect test cases to pass
-/
def n_ary_append {α : Type} : list (list α) → list α

-- test cases for 0, 1, 2, and more arguments
example : n_ary_append [] = [] := rfl
example : n_ary_append [[1,2,3]]] = [1,2,3] := rfl
example : n_ary_append [[1,2,3],[4,5,6]] = [1,2,3,4,5,6] := rfl
example : n_ary_append [[1,2,3],[4,5,6],[7,8,9]] = [1,2,3,4,5,6,7,8,9] := rfl

-- Problem #2
def sum_lengths {α : Type} : list (list α) → nat

example : @sum_lengths nat [] = 0 := rfl
example : sum_lengths [[1,2,3]] = 3 := rfl
example : sum_lengths [[1,2,3],[4,5,6]] = 6 := rfl
example : sum_lengths [[1,2,3],[4,5,6],[7,8,9]] = 9 := rfl

-- Problem #3
def even_lengths {α : Type} : list (list α) → bool

example : @even_lengths nat [] = tt := rfl
example : even_lengths [[1,2,3],[4,5,6],[7,8,9]] = ff := rfl
example : even_lengths [[1,2,3,4],[4,5,6],[7,8,9]] = ff := rfl
example : even_lengths [[1,2,3,4],[4,5,6,7],[7,8,9,0]] = tt := rfl
```

4.4.6 Fold: list $\alpha \to \beta$

The preceding few exercises all reduce a list of objects of one type (α) to a result of another (possibly the same) type (β). For example, even_lengths reduces a list of objects of one type, list α (not the same alpha, sorry), and returns a value of a different type, bool.

Can we devise a generalized version of fold that can handle all such reductions *in one fell swoop*? The answer is yes, but we need to think a bit harder about the nature of the binary operation to be extended to an n-ary operation by the application of the fold function.

Returning to our example, the way that the function works when the input list isn't empty is that it applies a binary operation to (a) the *head* of the given list, of type α , and (b) the *result* of folding/reducing the rest of the list, which is of type, β, yielding an overall value of the final result type, β.

Mixed-type binary operations

If we think of all this work as one operation, what is its type? Well, it's $\alpha \to \beta \to \beta$. Again, for example, it would compute a result from the head of the list (h : α) and the result of reducing the rest of the list (of type β) to yield a final result of type β.

We can think of this as the type of binary operation that fold is extending to an n-ary version, in general. Let's see this idea in action with a slightly simpler example of a folding function: one that takes a list of nat and that returns true if and only if all the numbers in the list are even.

```
def all_even' : list ℕ → bool
| list.nil := tt
| (h::t) := band (is_even h) (all_even' t)    -- band is &&

-- Seems to work
#eval all_even' [2,4,6,8]
#eval all_even' [1,4,6,8]
```

Now we'll apply the preceding reasoning to formulate what's going on in the second rule as a binary operation.

```
def all_even_op : nat → bool → bool
| n b := (is_even n) && b

def all_even : list nat → bool
| list.nil := tt
| (h::t) := all_even_op h (all_even t)

-- seems to be working
#eval all_even []       -- expect tt
#eval all_even [1]      -- expect ff
#eval all_even [0,2,4]  -- expect tt
#eval all_even [0,2,5]  -- expect ff
```

Extending such operations

Ok, so we're finally in a position to formally specify the type of any fold operation on lists. We'll call it foldr, short for "fold right," given that we combine the head of the list, on the left of h::t, with the result of folding its whole *right*-hand tail, t.

```
def foldr {α β : Type} : _
  -- op type
  -- id type
  -- list type
  -- result type
| _ := _
| _ := _

def all_even_yay : list nat → bool :=
  foldr all_even_op


#eval all_even_yay []        -- expect tt
#eval all_even_yay [1]       -- expect ff
#eval all_even_yay [0,2,4]   -- expect tt
#eval all_even_yay [0,2,5]   -- expect ff
#eval foldr nat.add 0 [1,2,3,4,5]
#eval foldr nat.mul 1 [1,2,3,4,5]
```

4.4.7 Summary

In summary, so far in this chapter, we've seen that higher-order functions are functions that consume functions as arguments and/or that return functions as results. We've produced highly general higher-order functions for (1) composition of functions, (2) mapping functions over lists to derive new lists, and (3) extending binary operators to n-ary operators whose arguments are given as lists of any length.

4.4.8 Exercises

– Define an n-ary Boolean "and" function using foldr. – Define an n-ary Boolean "or" function using foldr. – Define an n-ary $\mathbb{N}$ addition operator using foldr. – Define an n-ary $\mathbb{N}$ multiplication operator using foldr. – Define a function called map-reduce. It should accept list of objects of any type α, a function that converts α objects to β objects, and a binary operation suitable for use by our generalized fold function. As an example, you could use this function to reduce a list of strings to a Boolean value that's true if every string in a list of strings is of even length. First map the list of strings to a list of their lengths, then reduce this list to a Boolean, tt iff all lengths are even.

4.5 Recursive Proofs

4.5.1 Proof by Induction

Motivation

There was something notably questionable in the last chapter. We defined a *safe* version of *fold* by requiring a proof that the value returned for an empty list be a *right* (but not a *left*) identity element for the actual binary operator parameter given to the fold function.

We then found it easy to prove that 0 is indeed a right identity for nat.add. They key insight you need to have is that it was easy to prove because it's already given to us as an *axiom*. In particular, the first rule in the recursive definition of nat.add makes it so. Here's the definition of nat.add from Lean's core library.

```
def add : nat → nat → nat
| a  zero     := a
| a  (succ b) := succ (add a b)
```

Look at the first case/rule: any a added to zero is equal to a. This rule establishes that zero is a right identity for add. Here again is our earlier statement and proof.

Note that the *simp* tactict tries to fine, and if found applies, rules/axioms from the definition of any of the listed functions: here from just nat.add.

```
example : ∀ (n : ℕ), nat.add n 0 = n :=
begin
assume n,
simp [nat.add],
end
```

An English version of this proof might go like this: *We're to prove that for any n, n + 0 = n.* Proof: Assume n is an arbitrary but specific natural number. By the first rule/axiom defining nat.add we can rewrite n + 0 as n. That's it. As n is general, the conjecture, n + 0 = n, is true for all n. What's *not* provided by the definition of nat.add is an axiom that stipulates zero is a *left* identity for nat.add. If we try the same proof technique to prove ∀ *n, 0 + n = n*, with 0 now on the left, we can't! (When writing these propositions and proofs, use nat.add in a consistent manner instead of 0. It's a complication that's annoying, but for now just follow this simple instruction and you'll be fine.)

```
example : ∀ n, nat.add nat.zero n = n :=
begin
assume n,
simp [nat.add],
-- oops, that didn't help; we're stuck!
end
```

Looking at what remains to be proved, we might consider proof by case analysis on n. So let's try that.

```
example : ∀ n, nat.add nat.zero n = n :=
begin
assume n,
cases n with n',
-- first case: zero's also on the right
simp [nat.add],
-- second case, argument is succ of some n'
-- how to show 0 + (succ n') = (succ n')
-- but again we're stuck
simp [nat.add],
-- basically back where we started; stuck.
end
```

The problem is that all we know about n' is that it's some natural number, and that isn't enough to work with to prove the goal. That's the probem we solve now.

A Solution

What if we knew a little more? What if we knew that 0 is a left zero for n' as part of the context in which are to prove that it's a zero for (succ n')? Would that help?

It would. Suppose we know that *add 0 n' = n'* and that we want to prive that *add 0 (succ n') = (succ n')*. Key insight: We can apply the *second* axiom of addition,given by the second rule in its definition, to rewrite the term, *add 0 (succ n')* to the term *succ (add 0 n');* then we can use the fact that (by assumption) 0 is a left 0 for n' to rewrite the term *succ (add 0 n')* to *succ n'.* That's it. We've shown that 0 + succ n' = succ n'.

But what could possibly justify assuming that 0 + n' = n' in the first place? Well, let's see if it can be justified informally before getting into formalities.

Let's start by noting that by the first rule of addition, 0 is a left zero for 0. This proof gives us a base on which we can now construct a proof that 0 is a left zero for 1.

Details: we want to show that 0 + 1 = 1. That is, we want to show that 0 + succ 0 = succ 0. By the second rule/axiom of add, the left side is succ (0 + 0). *BE SURE YOU UNDERSTAND THIS STEP.* Now yy the first rule, 0 + 0 = 0, so we can rewrite succ (0 + 0) to just succ 0. With this expression on the left side, all thatremains to prove is that succ 0 = succ 0, and this is true of course by the reflexivity of the equality relation.

To recap, we proved a "base case" (that zero is a left identity for zero) using the first axiom of addition. Then we applied the second axiom to show that 0 is a left identity for 1. With this proof in hand we can apply the second axiom *again* to construct a proof that zero is left identity for 2. From this we can derive that 0 is a left identity for 3. Indeed to prove that 0 is a left identity for *any* n, we start with a proof that it's a left identity for zero using the first axiom, then we iteratively apply the second axiom n times to prove it's a left identity for *any* n.

Let's just program it to make it all clear. Out program will take any value n and return a proof that 0 is a left identity for it. It does this in the reverse order, constructing a proof for the case where n is non-zero, i.e., where n = succ n' for some n', and obtaining a proof for n' *by recursion*. The recursive calls implement iteration until the base case of n = 0 is reached, at which point a proof for that case is returned, the recursion unwinds, and we're left with a proof that 0 is a left identity for that arbitrary n. The existence of this function shows that we can construct a proof of the proposition that 0 is a left identity for any n, and so it is true *for all* n. And that's what we wanted. QED.

```
-- a proof-returning function defined by cases
-- takes any n and returns a proof of 0 + n = n
def zero_left_ident_n : ∀ n, (nat.add 0 n = n)
| nat.zero := by simp [nat.add] -- base case
| (nat.succ n') :=              -- recursive case
  begin
  simp [nat.add],               -- applies second rule and ...
                                -- removes succ on each side
                                -- by injectivity of constructors
                                -- inherent in inductive definitions
  exact (zero_left_ident_n n'), -- prove result recursively
  end

-- eyeball check of the recursive structure of these proofs!
#reduce zero_left_ident_n 0     -- the proof term is unpretty (just eyeball it)
#reduce zero_left_ident_n 1     -- the proof for 1 buids on the proof for 0
#reduce zero_left_ident_n 2     -- the proof for 2 buids on the proof for 1
                                -- and we see we can build such a proof for any n
                                -- therefore 0 is a left identity for addition
```

To sum up, what we've shown is that if we have two *little machines* we can construct a proof of the given proposition, let's call it P := (0 + n = n), for any value, n. The first machine produces a proof of P for the case where n = 0. The second machine, given a proof of P for any n' returns a proof for n' + 1. We've show that this is always possible. To construct a proof for any n, we then use the first machine to get a proof for 0, then we run the second machine n times starting on the proof for 0 to build a proof for n.

The resulting proof object has a recursive structure. Just as we've represented a non-zero natural number, n as the successor of some one-smaller natural number, n', so here we represent a proof of P for n = n' + 1 as a term that adds another layer of "proof stuff" around a proof of P for n', ultimately terminating with a proof of P for 0, with further sub-structure. apply Generalizing ~~~~~~~~~~~~

Just as we will need a proof that 0 is not only a right identity for nat.add (by the first axiom) but also a left identity (a theorem proved by induction), so will need a proof that nil is not only a right but also a left identity for the list append operation.

Here's the easy case first. From this proof you can infer that the list.append operation (with infix notation ++) has a rule/axiom that states that l ++ nil := l for any l.

```
/-
Here's the definition of list.append.
It asserts that [] is a left identity axiomatically.

def append : list α → list α → list α
| []       l := l
| (h :: s) t := h :: (append s t)
-/

-- proving right identity is trivial just as for addition
example (α : Type) : ∀ (l : list α), list.nil ++ l = l :=
begin
assume l,
simp [list.append],
end
```

```
def nil_left_ident_app (α : Type) : ∀ (l : list α), l ++ list.nil = l :=
begin
assume l,
cases l with h t,
-- base case
simp [list.append],   -- uses first rule
-- recursive case
simp [list.append],   -- why does this work?
end

-- Here's another formal demonstration of the same point
variables (α : Type) (a : α) (l : list α)
example: list.nil ++ l = l := by simp    -- first rule
example : l ++ list.nil  = l := by simp  -- by [simp] lemma in Lean library
```

4.5.2 Induction Axioms

4.5.3 Inductive Families

Coming soon.

```
inductive le (n : nat): nat → Prop
-- n is an implicit firt argument to each constructor
| refl : le /-n-/ n
| step : ∀ m, le /-n-/ m → le /-n-/ m.succ

-- you can see it in the types of the constructors
#check @le.refl
```

(continues on next page)

(continued from previous page)

```
#check @le.step


example : le 0 0 :=
begin
apply le.refl,
end

example : le 3 3 :=
begin
apply le.refl,
end

example : le 0 1 :=
begin
apply le.step,
apply le.refl,
end

example : le 0 3 :=
begin
apply le.step,
apply le.step,
apply le.step,
apply le.refl,
end

-- here's the same example using Lean's version of "le"
-- it's called nat.less_than_or_equal
example : 0 ≤ 3 :=
begin
apply nat.less_than_or_equal.step,
apply nat.less_than_or_equal.step,
apply nat.less_than_or_equal.step,
-- apply nat.less_than_or_equal.step,
apply nat.less_than_or_equal.refl,
end

-- repeat tactical goes too far; use iterate instead
example : 1 ≤ 4 :=
begin
-- repeat {apply nat.less_than_or_equal.step},
iterate 3 {apply nat.less_than_or_equal.step},
apply nat.less_than_or_equal.refl,
end
```

CHAPTER

FIVE

DEPENDENT TYPES

5.1 Σ Types

5.2 Π Types

A Pi type associates types with values. We define a predicate, Q, as an example to use in what follows. Q is true of any natural number *n* by the reflexivity of equality.

```
def Q (n : nat) := n = n
#check Q  -- "propositions are types"
```

5.2.1 Indexed Families

Note that each *n* to to which one applies *Q* gives rised to a proposition–a type–that *is about* (and in this sense, depends on) *n*. Such a types is said to be a *dependent type*.

Parametric polymorphism, by contrast, arises when *types* are parameters. For example, the *list* type builder takes an *type* (of list elements) as an argument and reduces to the type of lists of elements of that type. On the other hand, here the result type depends on a data *value*.

In effect, *Q* associates a separate type with each nat value. We say that *Q* defines a family of types *indexed by n*.

```
-- Q n is a type (proposition) dependent on n
#check Q 0
#check Q 1
#check Q 2
```

5.2.2 Dependent Function Types

The next insight to gain is that we can now define functions that return *values of dependent types*. To continue the preceding example, we define a new function that when given *n* returns not the type, n = n, but a proof of it: a value of the dependent type.

```
-- A function from n : ℕ to *proofs (values)* of *Q n*
def dep_func_prop (n : ℕ) : Q n := begin unfold Q end
```

```
#check dep_func_prop

#check dep_func_prop 0
```

(continues on next page)

(continued from previous page)

```
#check dep_func_prop 1
#check dep_func_prop 2

#reduce dep_func_prop 0
#reduce dep_func_prop 1
#reduce dep_func_prop 2

variables
  (α : Type)           -- a *data* type
  (P : α → Prop)       -- predicate on α

#check ∀ (a : α), P a
#check Π (a : α), P a
```

CHAPTER

SIX

INDEX