Proving Theorems in Deduce

Jeremy G. Siek

August 12, 2024

Contents

This booklet introduces the Deduce proof language whereas the booklet "Programming in Deduce" introduces the programming language. This booklet freely uses definitions that were introduced in "Programming in Deduce", so we recommend reading that booklet first.

1 Theorems and Applying Definitions to the Goal

We begin with a simple example, proving that the length of an empty list is `0`. Of course, this is a direct consequence of the definition of `length`, so this first example is about how to use definitions. To get started, we write down the theorem we would like to prove. A theorem consists of (1) the keyword `theorem`, (2) a name for the theorem, (3) a colon, (4) the formula, (5) the keyword `proof`, (6) the proof of the formula, and (7) the keyword `end`. But for now, instead of writing the proof, we'll simply write `?` to say that we're not done yet.

```
theorem length_nat_empty: length(@empty<Nat>) = 0
proof
  ?
end
```

Run Deduce on the file. Deduce will respond with the following message to remind us of what is left to prove.

```
incomplete proof:
    length(empty) = 0
```

To tell Deduce to apply the definition of `length`, we can use the `definition` statement.

```
theorem length_nat_empty: length(@empty<Nat>) = 0
proof
  definition length
end
```

Deduce expanded the definition of `length` in the goal, changing `length(empty) = 0` to `0 = 0`. In particular, Deduce noticed that `length(empty)` matches the first clause in the definition of `length` and then replaced it with the right-hand side of the first clause. Deduce then simplified `0 = 0` to `true` and therefore accepted the `definition` statement. In general, whenever Deduce sees an equality with the same left and right-hand side, it automatically simplifies it to `true`.

Run Deduce on the file to see it respond that the file is valid.

Let's try a slightly more complex theorem, that the length of a list with just a single node is indeed `1`. Based on what we learned above, we better start by applying the definition of `length` a couple of times.

```
theorem length_node42: length(node(42, empty)) = 1
proof
  definition {length, length}
end
```

Deduce responds that we still need to prove the following obvious fact.

```
failed to prove:
    length(node(42,empty)) = 1
by
    definition {length, length}
remains to prove:
    1 + 0 = 1
```

But that is just a consequence of the definition of addition, which we can refer to as `operator +`. To carry on with proving what remains, we can use the `suffices` statement as follows. We write the formula that is left to prove after the `suffices` keyword then `by` then the `definition` statement that we're using to transform the goal. After the `suffices`, the goal changes to the `suffices` formula, which here is `1 + 0 = 1`.

```
theorem length_node42: length(node(42, empty)) = 1
proof
  suffices 1 + 0 = 1   with definition {length, length}
  definition {operator +, operator +}
end
```

Exercise

Prove that `node(1,empty) ++ node(2, empty) = node(1, node(2, empty))`.

2 Generalizing with `all` Formulas

In the proof of `length_node42` it did not matter that the element in the node was `42`. We can generalize this theorem by using an `all` formula. We begin the formula with `all x:Nat` to say that the formula must be true for all natural numbers and the variable `x` will be used as a stand-in to refer to any of them. We replace the `42` in the formula with `x` to obtain the following theorem.

```
theorem length_one_nat: all x:Nat. length(node(x, empty)) = 1
proof
  ?
end
```

Deduce responds with

```
incomplete proof:
    all x:Nat. length(node(x,empty)) = 1
```

The most straightforward way to prove an `all` formula in Deduce is with an `arbitrary` statement. When you use `arbitrary` you are promising to prove the formula for a hypothetical entity that can stand in for all entities of the specified type. The `arbitrary` statement asks you to name the hypothetical entity. Here we choose `x` but we could have chosen a different name.

```
theorem length_one_nat: all x:Nat. length(node(x, empty)) = 1
proof
  arbitrary x:Nat
  ?
end
```

Deduce responds with

```
incomplete proof:
    length(node(x,empty)) = 1
```

We don't know anything about this hypothetical `x` other than it being a natural number. But as we previously observed, we don't need any more information about `x` for this proof. We complete the proof as before, using the definitions of `length` and addition.

```
theorem length_one_nat: all x:Nat. length(node(x, empty)) = 1
proof
  arbitrary x:Nat
  definition {length, length, operator +, operator +}
end
```

Once we have proved that an `all` formula is true, we can use it by supplying an entity of the appropriate type inside square brackets. In the following we prove the `length_node42` theorem again, but this time the proof makes use of `length_one_nat`.

```
theorem length_node42_again: length(node(42, empty)) = 1
proof
  length_one_nat[42]
end
```

We can further generalize the theorem by noticing that it does not matter whether the element is a natural number. It could be a value of any type. In Deduce we can also use the `all` statement to generalize types. In the following, we add `U:type` to the `all` formula and to the `arbitrary` statement.

```
theorem length_one: all U:type, x:U. length(node(x, empty)) = 1
proof
  arbitrary U:type, x:U
  definition {length, length, operator +, operator+}
end
```

To summarize this section:

- To state that a formula is true for all entities of a given type, use Deduce's `all` formula.
- To prove that an `all` formula is true, use Deduce's `arbitrary` statement. (We'll see a second method in section 7.)
- To use a fact that is an `all` formula, instantiate the fact by using square brackets around the specific entity.

Exercise

Prove that

```
all T:type, x:T, y:T.
    node(x,empty) ++ node(y, empty) = node(x, node(y, empty))
```

Prove again that

```
node(1,empty) ++ node(2, empty) = node(1, node(2, empty))
```

but this time use the previous theorem.

3 Rewriting the Goal with Equations

Deduce provides the `rewrite` statement to apply an equation to the current goal. In particular, `rewrite` replaces each occurence of the left-hand side of an equation with the right-hand side of the equation.

For example, let us prove the following theorem using `rewrite` with the above `length_one` theorem.

```
theorem length_one_equal: all U:type, x:U, y:U.
  length(node(x,empty)) = length(node(y,empty))
proof
  arbitrary U:type, x:U, y:U
  ?
end
```

To replace `length(node(x,empty))` with `1`, we rewrite using the `length_one` theorem instantiated at `U` and `x`.

```
rewrite length_one[U,x]
```

Deduce tells us that the current goal has become

```
remains to prove:
        1 = length(node(y,empty))
```

We rewrite again, separated by a vertical bar, using `length_one`, this time instantiated with `y`.

```
rewrite length_one[U,x] | length_one[U,y]
```

Deduce changes the goal to `1 = 1`, which simplies to just `true`, so Deduce accepts the `rewrite` statement.

Here is the completed proof of `length_one_equal`.

```
theorem length_one_equal: all U:type, x:U, y:U.
  length(node(x,empty)) = length(node(y,empty))
proof
  arbitrary U:type, x:U, y:U
  rewrite length_one[U,x] | length_one[U,y]
end
```

```
add_zero: all n:Nat.  n + 0 = n
add_commute: all n:Nat. all m:Nat.  n + m = m + n
add_assoc: all m:Nat. all n:Nat, o:Nat.  (m + n) + o = m + (n + o)
left_cancel: all x:Nat. all y:Nat, z:Nat.  if x + y = x + z then y = z
add_to_zero: all n:Nat. all m:Nat. if n + m = 0 then n = 0 and m = 0
dist_mult_add: all a:Nat. all x:Nat, y:Nat. a * (x + y) = a * x + a * y
mult_zero: all n:Nat. n * 0 = 0
mult_one: all n:Nat. n * 1 = n
mult_commute: all m:Nat. all n:Nat. m * n = n * m
mult_assoc: all m:Nat. all n:Nat, o:Nat. (m * n) * o = m * (n * o)
```

Figure 1: A selection of theorems from `Nat.pf`.

4 Reasoning about Natural Numbers

The `Nat.pf` file includes the definition of natural numbers, operations on them (e.g. addition), and proofs about those operations. Here we discuss how to reason about addition. Reasoning about the other operations follows a similar pattern.

Here is the definition of addition from `Nat.pf`:

```
function operator +(Nat,Nat) -> Nat {
  operator +(0, m) = m
  operator +(suc(n), m) = suc(n + m)
}
```

Recall that we can use Deduce's `definition` statement whenever we want to rewrite the goal according to the equations for addition. Here are the two defining equations, but written with infix notation:

```
0 + m = m
suc(n) + m = suc(n + m)
```

The `Nat.pf` file also includes proofs of many equations. Figure 1 lists a selection of the theorems.

You can use these theorems by instantiating them with particular entities. For example, `add_zero[2]` is a proof of `2 + 0 = 2`. We have not yet discussed how to use the `if-then` formula in `left_cancel`, but we will get to that in section 12.

Exercise

Prove the following theorem using `left_cancel` and using `add_commute` with `rewrite`.

```
right_cancel: all x:Nat, y:Nat, z:Nat.
    if x + z = y + z then x = y
```

5 Proving Intermediate Facts with `have`

One often needs to prove some intermediate facts on the way to proving the final goal of a theorem. The `have` statement of Deduce provides a way to state and prove a fact and give it a label so that it can be used later in the proof. For example, consider the proof of

```
x + y + z = z + y + x
```

It takes several uses of `add_commute` and `add_assoc` to prove this. To get started, we use `have` to prove `step1`, which states that `x + y + z = x + z + y` (flipping the `y` and `z`).

```
theorem xyz_zyx: all x:Nat, y:Nat, z:Nat.
  x + y + z = z + y + x
proof
  arbitrary x:Nat, y:Nat, z:Nat
  have step1: x + y + z = x + z + y
    by rewrite add_commute[y][z]
  ?
end
```

Deduce prints the current goal and the **givens**, that is, the facts that we aleady know are true, which now includes `step1`.

```
incomplete proof
Goal:
    x + (y + z) = z + (y + x)
Givens:
    step1: x + (y + z) = x + (z + y)
```

We proceed four more times, using `have` to create each intermediate step in the reasoning.

```
  have step2: x + z + y = (x + z) + y by rewrite add_assoc[x][z,y]
  have step3: (x + z) + y = (z + x) + y by rewrite add_commute[z][x]
  have step4: (z + x) + y = z + (x + y) by rewrite add_assoc[z][x,y]
  have step5: z + (x + y) = z + y + x by rewrite add_commute[x][y]
```

```
theorem xyz_zyx: all x:Nat, y:Nat, z:Nat.
  x + y + z = z + y + x
proof
  arbitrary x:Nat, y:Nat, z:Nat
  have step1: x + y + z = x + z + y
    by rewrite add_commute[y][z]
  have step2: x + z + y = (x + z) + y
    by rewrite add_assoc[x][z,y]
  have step3: (x + z) + y = (z + x) + y
    by rewrite add_commute[z][x]
  have step4: (z + x) + y = z + (x + y)
    by rewrite add_assoc[z][x,y]
  have step5: z + (x + y) = z + y + x
    by rewrite add_commute[x][y]
  transitive step1 (transitive step2 (transitive step3
    (transitive step4 step5)))
end
```

Figure 2: Proof of the `xyz_zyx` theorem.

We finish the proof by connecting them all together using Deduce's `transitive` statement. The `transitive` statement takes two proofs of equations, such as `a = b` and `b = c`, and proves `a = c`.

```
  transitive step1 (transitive step2 (transitive step3
    (transitive step4 step5)))
```

Figure 2 shows the complete proof of the `xyz_zyx` theorem.

6 Chaining Equations with `equations`

Combining a sequence of equations using `transitive` is quite common but also cumbersome, so Deduce provides the `equations` statement to streamline this process. After the first equation, the left-hand side of each equation is written as `...` because it is just a repetition of the right-hand side of the previous equation. Here's another proof of the theorem about `x + y + z`, this time using an `equations` statement.

```
theorem xyz_zyx_eqn: all x:Nat, y:Nat, z:Nat.
  x + y + z = z + y + x
proof
  arbitrary x:Nat, y:Nat, z:Nat
  equations
    x + y + z = x + z + y      by rewrite add_commute[y][z]
          ... = (x + z) + y    by rewrite add_assoc[x][z,y]
          ... = (z + x) + y    by rewrite add_commute[z][x]
          ... = z + x + y      by rewrite add_assoc[z][x,y]
          ... = z + y + x      by rewrite add_commute[x][y]
end
```

Exercise

Prove that `x + y + z = z + y + x` but using fewer than 5 steps.

7 Proving `all` Formulas with Induction

Sometimes the `arbitrary` statement does not give us enough information to prove an `all` formula. In those situations, so long as the type of the `all` variable is a `union` type, we can use the more powerful `induction` statement.

For example, consider this theorem about appending a list to an empty list. Suppose we try to use `arbitrary` for both the `all U` and the `all xs`.

```
theorem append_empty: all U :type. all xs :List<U>.
  xs ++ empty = xs
proof
  arbitrary U:type
  arbitrary xs:List<U>
  ?
end
```

Deduce replies that we need to prove

```
incomplete proof:
    xs ++ empty = xs
```

But now we're stuck because the definition of append pattern matches on its first argument, but we don't know whether `xs` is an `empty` list or a `node`.

So instead of using `arbitrary xs:List<U>` to prove the `all xs`, we proceed by induction as follows.

```
theorem append_empty: all U :type. all xs :List<U>.
  xs ++ empty = xs
proof
  arbitrary U:type
  induction List<U>
  case empty {
    ?
  }
  case node(n, xs') suppose IH: xs' ++ empty = xs' {
    ?
  }
end
```

When doing a proof by induction, there is one `case` for every alternative in the `union` type. Here the union type is `List<U>`, so we have one case for `empty` and one case for `node`. Furthermore, because `node` includes a recursive argument, that is, and argument of type `List<U>`, in the case for `node` we get to assume that the formula we are trying to prove is already true for the argument. This is commonly known at the **induction hypothesis**. We must give a label for the induction hypothesis so here we choose `IH` for short.

Let us first focus on the case for `empty`. Deduce tells us that we need to prove the following.

```
incomplete proof:
    empty ++ empty = empty
```

This follows directly from the definition of append.

```
    case empty {
      definition operator++
    }
```

However, to make the proof more readable by other humans, I recommend restating the goal using the `conclude` statement.

```
    case empty {
      conclude @empty<U> ++ empty = empty
          by definition operator++
    }
```

Next let us focus on the case for `node`. Deduce tells us that we need to prove the following and that `IH` has been added to the available facts.

```
incomplete proof:
    node(n,xs') ++ empty = node(n,xs')

available facts:
    IH: xs' ++ empty = xs',
    ...
```

Looking at the goal, we notice that we can expand the definition of `append` on the right-hand side, because it is applied to a `node`. Deduce provides the `term` statement as way to use Deduce to expand definitions for us.

```
case node(n, xs') suppose IH: xs' ++ empty = xs' {
  term node(n,xs') ++ empty by definition operator++
  ?
}
```

Deduce responds with

```
remains to prove:
    node(n,xs' ++ empty)
```

We use Deduce's `have` statement to label this equality. We choose the label `step1`, state the equality, and then provide its proof after the `by` keyword.

```
case node(n, xs') suppose IH: xs' ++ empty = xs' {
  have step1: node(n,xs') ++ empty
            = node(n, xs' ++ empty)  by definition operator++
  ?
}
```

Next, we see that the subterm `xs' ++ empty` matches the right-hand side of the induction hypothesis `IH`. We use the `rewrite` statement to apply the `IH` equation to this subterm.

```
have step2: node(n, xs' ++ empty)
          = node(n,xs')                 by rewrite IH
```

To complete the proof, we combine equations (1) and (2) using the `transitive` statement.

```
conclude node(n,xs') ++ empty = node(n,xs')
    by transitive step1 step2
```

The completed proof of `append_empty` is shown in Figure 3, but we replace the intermediate `have` statements and `transitive` by an `equations` statement.

To summarize this section:

- To prove an `all` formula that concerns entities of a `union` type, use Deduce's `induction` statement.

Exercise

Prove that `length(xs ++ ys) = length(xs) + length(ys)`.

```
theorem append_empty: all U :type. all xs :List<U>.
  xs ++ empty = xs
proof
  arbitrary U:type
  induction List<U>
  case empty {
    conclude @empty<U> ++ empty = empty
        by definition operator++
  }
  case node(n, xs') suppose IH: xs' ++ empty = xs' {
    equations
      node(n,xs') ++ empty
          = node(n, xs' ++ empty)   by definition operator++
      ... = node(n,xs')             by rewrite IH
  }
end
```

Figure 3: Proof of the append_empty theorem.

8 Reasoning about and (Conjunction)

To create a single formula that expresses that two formulas are true, combine the two formulas with `and` (i.e. conjunction). The following example proves that `0 ≤ 1 and 0 ≤ 2`. This is accomplished by separately proving that `0 ≤ 1` is true and that `0 ≤ 2` is true, then using the comma operator to combine those proofs: `one_pos, two_pos`.

```
theorem positive_1_and_2: 0 ≤ 1 and 0 ≤ 2
proof
  have one_pos: 0 ≤ 1 by definition operator ≤
  have two_pos: 0 ≤ 2 by definition operator ≤
  conclude 0 ≤ 1 and 0 ≤ 2 by one_pos, two_pos
end
```

On the other hand, in Deduce you can use a conjunction as if it were one of its subformulas, implicitly. In the following we use the fact that `0 ≤ 1 and 0 ≤ 2` to prove `0 ≤ 2`.

```
theorem positive_2: 0 ≤ 2
proof
  conclude 0 ≤ 2 by positive_1_and_2
end
```

To summarize this section:

- Use `and` in Deduce to express the truth of two formulas.
- To prove an `and` formula, prove its parts and then combine them using comma.
- You can implicitly use an `and` formula as one of its parts.

9 Reasoning about `or` (Disjunction)

Two create a formula that expresses that at least one of two formulas is true (i.e. disjunction), use `or` to combine the formulas.

For example, consider the following variation on the trichotomy law, which states that for any two natural numbers `x` and `y`, either `x ≤ y` or `y < x`.

```
theorem dichotomy:  all x:Nat, y:Nat.  x ≤ y  or  y < x
proof
  ?
end
```

We can prove this using the `trichotomy` theorem from `Nat.pf`, which tells us that `x < y` or `x = y` or `y < x`.

```
theorem dichotomy:  all x:Nat, y:Nat.  x ≤ y  or  y < x
proof
  arbitrary x:Nat, y:Nat
  have tri: x < y or x = y or y < x by trichotomy[x][y]
  ?
end
```

In Deduce, you can use an `or` fact by doing case analysis with the `cases` statement. There is one `case` for each subformula of the `or`.

```
have tri: x < y or x = y or y < x by trichotomy[x][y]
cases tri
case x_l_y: x < y {
  ?
}
case x_eq_y: x = y {
  ?
}
case y_l_x: y < x {
  ?
}
```

In the first case, we consider the situation where `x < y` and still need to prove that `x ≤ y or y < x`. Thankfully, the theorem `less_implies_less_equal` in `Nat.pf` tells us that `x ≤ y`.

```
case x_l_y: x < y {
  have x_le_y: x ≤ y
      by apply less_implies_less_equal[x][y] to x_l_y
  ?
}
```

In Deduce, an `or` formula can be proved using a proof of either subformula, so here we prove `x ≤ y or y < x` with `x ≤ y`.

```
case x_l_y: x < y {
  have x_le_y: x ≤ y
      by apply less_implies_less_equal[x][y] to x_l_y
  conclude x ≤ y or y < x by x_le_y
}
```

In the second case, we consider the situation where `x = y`. Here we can prove that `x ≤ y` by rewriting the `x` to `y` and then using the reflexive property of the less-equal relation to prove that `y ≤ y`.

```
case x_eq_y: x = y {
  have x_le_y: x ≤ y by
      suffices y ≤ y  with rewrite x_eq_y
      less_equal_refl[y]
  conclude x ≤ y or y < x by x_le_y
}
```

In the third case, we consider the situation where `y < x`. So we can immediately conclude that `x ≤ y or y < x`.

```
case y_l_x: y < x {
  conclude x ≤ y or y < x by y_l_x
}
```

Figure 4 shows the completed proof of the `dichotomy` theorem.

To summarize this section:

- Use `or` in Deduce to express that at least one of two or more formulas is true.
- To prove an `or` formula, prove either one of the formulas.
- To use a fact that is an `or` formula, use the `cases` statement.

```
theorem dichotomy:  all x:Nat, y:Nat.  x ≤ y  or  y < x
proof
  arbitrary x:Nat, y:Nat
  have tri: x < y or x = y or y < x by trichotomy[x][y]
  cases tri
  case x_l_y: x < y {
    have x_le_y: x ≤ y
        by apply less_implies_less_equal[x][y] to x_l_y
    conclude x ≤ y or y < x by x_le_y
  }
  case x_eq_y: x = y {
    have x_le_y: x ≤ y by
        suffices y ≤ y   with rewrite x_eq_y
        less_equal_refl[y]
    conclude x ≤ y or y < x by x_le_y
  }
  case y_l_x: y < x {
    conclude x ≤ y or y < x by y_l_x
  }
end
```

Figure 4: Proof of the `dichotomy` theorem.

10 The `switch` Proof Statement

Similar to Deduce's `switch` statement for writing functions, there is also a `switch` statement for writing proofs. As an example, let us consider how to prove the following theorem.

```
theorem zero_or_positive: all x:Nat. x = 0 or 0 < x
proof
  ?
end
```

We could proceed by induction, but it turns out we don't need the induction hypothesis. In such situations, we can instead use `switch`. Like induction, `switch` works on unions and there is one case for each alternative of the union. Unlike induction, the goal formula does not need to be an `all` formula. Instead, you indicate which entity to switch on, as in `switch x` below.

```
arbitrary x:Nat
switch x {
  case zero {
    ?
  }
  case suc(x') {
    ?
  }
}
```

Deduce responds that in the first case we need to prove the following.

```
incomplete proof:
    true or 0 < 0
```

So we just need to prove `true`, which is what the period is for.

```
    case zero {
      conclude true or 0 < 0 by .
    }
```

In the second case, for `x = suc(x')`, we need to prove the following.

```
incomplete proof:
    false or 0 < suc(x')
```

There's no hope of proving `false`, so we better prove `0 < suc(x')`. Thankfully that follows from the definitions of < and ≤.

```
    case suc(x') {
      have z_l_sx: 0 < suc(x')
          by definition {operator <, operator ≤}
      conclude suc(x') = 0 or 0 < suc(x') by z_l_sx
    }
```

Here is the completed proof that every natural number is either zero or positive.

```
theorem zero_or_positive: all x:Nat. x = 0 or 0 < x
proof
  arbitrary x:Nat
  switch x {
    case zero {
      conclude true or 0 < 0 by .
    }
    case suc(x') {
      have z_l_sx: 0 < suc(x')
          by definition {operator <, operator ≤, operator ≤}
      conclude suc(x') = 0 or 0 < suc(x') by z_l_sx
    }
  }
end
```

To summarize this section:

- Use `switch` on an entity of union type to split the proof into cases, with one case for each alternative of the union.

11 Applying Definitions and Rewrites to the Goal

Sometimes one needs to apply a set of definitions and rewrites to the goal. Consider the following definition of `max'`. (There is a different definition of `max` in `Nat.pf`.)

```
define max' : fn Nat, Nat -> Nat
            = λx,y{ if x ≤ y then y else x }
```

To prove that `x ≤ max'(x,y)` we consider two cases, whether `x ≤ y` or not. If `x ≤ y` is true, we apply the definition of `max'` **and** we rewrite with the fact that `x ≤ y` is true, which resolves the `if-then-else` inside of `max'` to just `y`.

```
    suffices x ≤ y with definition max' and rewrite x_le_y_true
```

So we are left to prove `x ≤ y`, which we already know. Similarly, if `x ≤ y` is false, we apply the definition of `max'` and rewrite with the fact that `x ≤ y` is false.

```
    suffices x ≤ x with definition max' and rewrite x_le_y_false
```

This resolves the `if-then-else` inside of `max'` to just `x`. So we are left to prove `x ≤ x`, which of course is true. Here is the complete proof that `x ≤ max'(x,y)`.

```
theorem less_max: all x:Nat, y:Nat.  x ≤ max'(x,y)
proof
  arbitrary x:Nat, y:Nat
  switch x ≤ y {
    case true suppose x_le_y_true {
      suffices x ≤ y with definition max' and rewrite x_le_y_true
      rewrite x_le_y_true
    }
    case false suppose x_le_y_false {
      suffices x ≤ x with definition max' and rewrite x_le_y_false
      less_equal_refl[x]
    }
  }
end
```

12 Conditional Formulas (Implication) and Applying Definitions to Facts

Some logical statements are true only under certain conditions, so Deduce provides an `if-then` formula. To demonstrate how to work with `if-then` formulas, we prove that if a list has length zero, then it must be the `empty` list. Along the way we will also learn how to apply a definition to an already-known fact.

```
theorem length_zero_empty: all T:type, xs:List<T>.
  if length(xs) = 0 then xs = empty
proof
  arbitrary T:type, xs:List<T>
  ?
end
```

Deduce tells us:

```
incomplete proof
Goal:
        (if length(xs) = 0 then xs = empty)
```

To prove an `if-then` formula, we `suppose` the condition and then prove the conclusion.

```
  suppose len_z: length(xs) = 0
```

Deduce adds `len_z` to the givens (similar to `have`).

```
incomplete proof
Goal:
        xs = empty
Givens:
        len_z: length(xs) = 0
```

Next we `switch` on the list `xs`. In the case when `xs` is `empty` it will be trivial to prove `xs = empty`. In the other case, we will obtain a contradiction.

```
  switch xs {
    case empty { . }
    case node(x, xs') suppose xs_xxs: xs = node(x,xs') {
      ?
    }
  }
```

We can put the facts `len_z` and `xs_xxs` together to obtain the dubious looking `length(node(x,xs')) = 0`.

```
    have len_z2: length(node(x,xs')) = 0  by rewrite xs_xxs in len_z
```

The contradiction becomes apparent to Deduce once we apply the definition of `length` to this fact. We do so using Deduce's `definition-in` statement as follows.

```
    conclude false  by definition length in len_z2
```

We discuss contradictions and `false` in more detail in section 14.

Here is the complete proof of `length_zero_empty`.

```
theorem length_zero_empty: all T:type, xs:List<T>.
  if length(xs) = 0 then xs = empty
proof
  arbitrary T:type, xs:List<T>
  suppose len_z: length(xs) = 0
  switch xs {
    case empty { . }
    case node(x, xs') suppose xs_xxs: xs = node(x,xs') {
          have len_z2: length(node(x,xs')) = 0  by rewrite xs_xxs in len_z
      conclude false  by apply not_one_add_zero[length(xs')]
                         to definition length in len_z2
    }
  }
end
```

The next topic to discuss is how to use an `if-then` fact that is already proven. We use Deduce's `apply-to` statement (aka. modus ponens) to obtain the conclusion of an `if-then` formula by supplying a proof of the condition. We demonstrate several uses of `apply-to` in the proof of the following theorem, which builds on `length_zero_empty`.

```
theorem length_append_zero_empty:
  all T:type, xs:List<T>, ys:List<T>.
  if length(xs ++ ys) = 0
  then xs = empty and ys = empty
proof
  arbitrary T:type, xs:List<T>, ys:List<T>
  suppose len_xs_ys: length(xs ++ ys) = 0
  ?
end
```

Recall that in a previous exercise, you proved that

```
length(xs ++ ys) = length(xs) + length(ys)
```

so we can prove that `length(xs) + length(ys) = 0` as follows.

```
  have len_xs_len_ys: length(xs) + length(ys) = 0
    by transitive (symmetric length_append[T][xs][ys]) len_xs_ys
```

Note that Deduce's the `symmetric` statement takes a proof of some equality like `a = b` and flips it around to `b = a`.

Now from `Nat.pf` we have the following `if-then` fact.

```
add_to_zero: all n:Nat. all m:Nat.
    if n + m = 0 then n = 0 and m = 0
```

Here we use of `apply-to` to obtain `length(xs) = 0` and the same for `ys`.

```
  have len_xs: length(xs) = 0
      by apply add_to_zero to len_xs_len_ys
  have len_ys: length(ys) = 0
      by apply add_to_zero to len_xs_len_ys
```

We conclude that `xs = empty and ys = empty` with another use of `apply-to`, where we make use of the previous theorem `length_zero_empty`.

```
  conclude xs = empty and ys = empty
  by (apply length_zero_empty[T,xs] to len_xs),
     (apply length_zero_empty[T,ys] to len_ys)
```

Here is the complete proof of `length_append_zero_empty`.

```
theorem length_append_zero_empty:
  all T:type, xs:List<T>, ys:List<T>.
  if length(xs ++ ys) = 0
  then xs = empty and ys = empty
proof
  arbitrary T:type, xs:List<T>, ys:List<T>
  suppose len_xs_ys: length(xs ++ ys) = 0
  have len_xs_len_ys: length(xs) + length(ys) = 0
    by transitive (symmetric length_append[T][xs][ys]) len_xs_ys
  have len_xs: length(xs) = 0
      by apply add_to_zero to len_xs_len_ys
  have len_ys: length(ys) = 0
      by apply add_to_zero to len_xs_len_ys
  conclude xs = empty and ys = empty
  by (apply length_zero_empty[T,xs] to len_xs),
     (apply length_zero_empty[T,ys] to len_ys)
end
```

To summarize this section:

- A conditional formula is stated in Deduce using the `if-then` syntax.
- To prove an `if-then` formula, `suppose` the condition and prove the conclusion.
- To use a fact that is an `if-then` formula, `apply` it `to` a proof of the condition.
- To apply a definition to a fact, use `definition-in`.

Exercise

Prove that `all x:Nat. if x` $\leq$ `0 then x = 0`.

13 Reasoning about `true`

There's not much to say about `true`. It's true! And as we've already seen, proving `true` is easy. Just use a period.

```
theorem really_trivial: true
proof
  .
end
```

One almost never sees `true` written explicitly in a formula. However, it is common for a formula to simplify to `true` after some rewriting.

14 Reasoning about `false`

The formula `false` is also rarely written explicitly in a formula. However, it can arise in contradictory situations. For example, in the following we have a situation in which `true = false`. That can't be, so Deduce simplifies `true = false` to just `false`.

```
theorem contra_false: all a:bool, b:bool.
  if a = b and a = true and b = false then false
proof
  arbitrary a:bool, b:bool
  suppose prem: a = b and a = true and b = false
  have a_true: a = true by prem
  have b_true: b = false by prem
  conclude false by rewrite a_true | b_true in prem
end
```

More generally, Deduce knows that the different constructors of a union are in fact different. So in the next example, because `foo` and `bar` are different constructors, Deduce simplifies `foo = bar` to `false`.

```
union U {
  foo
  bar
}

theorem foo_bar_false: if foo = bar then false
proof
  .
end
```

The above proof is just a period because Deduce simplifies any formula of the form `if false then ...` to `true`, which is related to our next point.

So far we've discussed how a proof of `false` can arise. Next let's talk about how you can use `false` once you've got it. The answer is anything! The Principle of Explosion tells us that `false` implies anything[1]. For example, normally we don't know whether or not two arbitrary Booleans `x` and `y` are the same or different. But if we have a premise that is `false`, it doesn't matter.

[1] I promise I didn't make this up. It's a legitimate rule of logic!

```
theorem false_any: all x:bool, y:bool. if false then x = y
proof
  arbitrary x:bool, y:bool
  suppose f: false
  conclude x = y by f
end
```

To summarize this section:

- Deduce simplifies any obviously contradictory equation to `false`.
- `false` implies anything.

15 Reasoning about `not`

To express that a formula is false, precede it with `not`. For example, for any natural number `x`, it is not the case that `x < x`.

```
theorem less_irreflexive:  all x:Nat. not (x < x)
proof
  ?
end
```

Deduce treats `not` as syntactic sugar for a conditional formal with a `false` conclusion. Thus, Deduce responds to the above partial proof with the following message.

```
incomplete proof:
    all x:Nat. (if x < x then false)
```

We can proceed by induction.

```
induction Nat
case zero {
  ?
}
case suc(x') suppose IH: not (x' < x') {
  ?
}
```

In the first case, we must prove the following conditional formula.

```
incomplete proof:
    (if 0 < 0 then false)
```

So we assume the premise `0 < 0`, from which we can conclude `false` by the definitions of < and ≤.

```
case zero {
  suppose z_l_z: 0 < 0
  conclude false by definition {operator <, operator ≤} in z_l_z
}
```

In the case where `x = suc(x')`, we must prove the following

```
    incomplete proof:
        (if suc(x') < suc(x') then false)
```

So we assume the premise `suc(x') < suc(x')` from which we can prove that `x' < x'` using the definitions of < and ≤.

```
    suppose sx_l_sx: suc(x') < suc(x')
    enable {operator <, operator ≤}
    have x_l_x: x' < x' by sx_l_sx
```

We conclude this case by applying the induction hypothesis to `x' < x'`.

```
    conclude false by apply IH to x_l_x
```

Here is the completed proof that less-than is irreflexive.

```
theorem less_irreflexive:  all x:Nat. not (x < x)
proof
  induction Nat
  case zero {
    suppose z_l_z: 0 < 0
    conclude false by definition {operator <, operator ≤} in z_l_z
  }
  case suc(x') suppose IH: not (x' < x') {
    suppose sx_l_sx: suc(x') < suc(x')
    enable {operator <, operator ≤}
    have x_l_x: x' < x' by sx_l_sx
    conclude false by apply IH to x_l_x
  }
end
```

To summarize this section:

- To expression that a formula is false, use `not`.
- Deduce treats the formula `not P` just like `if P then false`.
- Therefore, to prove a `not` formula, suppose `P` then prove `false`.
- To use a formula like `not P`, apply it to a proof of `P` to obtain a proof of `false`.

16 Rewriting Facts with Equations

In section 3 we learned that the `rewrite` statement of Deduce applies an equation to the current goal. There is a second variant of `rewrite` that applies an equation to a fact. As an example, we'll prove the following theorem that is a straightforward use of `less_irreflexive`.

```
theorem less_not_equal: all x:Nat, y:Nat.
  if x < y then not (x = y)
proof
  arbitrary x:Nat, y:Nat
  suppose x_l_y: x < y
  ?
end
```

Deduce responds with the current goal, in which `not (x = y)` is expanding into `if x = y then false`.

```
incomplete proof
Goal:
        (if x = y then false)
Givens:
        x_l_y: x < y
```

So following the usual recipe to prove an `if-then`, we `suppose` the condition `x = y`.

```
  suppose x_y: x = y
```

Now we need to prove false, and we have the hint to use the `less_irreflexive` theorem.

```
incomplete proof
Goal:
        false
Givens:
        x_y: x = y,
        x_l_y: x < y
```

Here is where the second variant of `rewrite` comes in. We can use it to apply the equation `x = y` to the fact `x < y` to get `y < y`. Note the extra keyword `in` that is used in this version of `rewrite`.

```
  have y_l_y: y < y   by rewrite x_y in x_l_y
```

We arrive at the contradition by applying `less_irreflexive` to `y < y`.

```
  conclude false by apply less_irreflexive[y] to y_l_y
```

Here is the complete proof of `less_not_equal`.

```
theorem less_not_equal: all x:Nat, y:Nat.
  if x < y then not (x = y)
proof
  arbitrary x:Nat, y:Nat
  suppose x_l_y: x < y
  suppose x_y: x = y
  have y_l_y: y < y by rewrite x_y in x_l_y
  conclude false by apply less_irreflexive[y] to y_l_y
end
```

16.1 Exercise

Prove the following theorem without using `less_not_equal`.

```
greater_not_equal: all x:Nat, y:Nat. if x > y then not (x = y)
```

Note that greater-than is defined as follows in `Nat.pf`:

```
define operator > : fn Nat,Nat -> bool = λ x, y { y < x }
```

17 Reasoning about `some` (Exists)

In Deduce, you can express that there is at least one entity that satisfies a given property using the `some` formula. For example, one way to define an even number is to say that it is a number that is 2 times some other number. We express this in Deduce as follows.

```
define Even : fn Nat -> bool = λ n { some m:Nat. n = 2*m }
```

As an example of how to reason about `some` formulas, let us prove a classic property of the even numbers, that the addition of two even numbers is an even number. Here's the beginning of the proof.

```
theorem addition_of_evens:
  all x:Nat, y:Nat.
  if Even(x) and Even(y) then Even(x + y)
proof
  arbitrary x:Nat, y:Nat
  suppose even_xy: Even(x) and Even(y)
  have even_x: some m:Nat. x = 2*m
      by definition Even in even_xy
  have even_y: some m:Nat. y = 2*m
      by definition Even in even_xy
  ?
end
```

The next step in the proof is to make use of the facts `even_x` and `even_y`. We can make use of a `some` formula using the `obtain` statement of Deduce.

```
obtain a where x_2a: x = 2*a from even_x
obtain b where y_2b: y = 2*b from even_y
```

Deduce responds with

```
available facts:
    y_2b: y = 2*b,
    x_2a: x = 2*a,
```

The `a` and `b` are new variables and the two facts `y_2b` and `x_2a` are the subformulas of the `some`, but with `a` and `b` replacing `m`.

We still need to prove the following:

```
incomplete proof:
    Even(x + y)
```

So we use the definition of `Even` in a `suffices` statement

```
suffices some m:Nat. x + y = 2*m   with definition Even
?
```

To prove a `some` formula, we use Deduce's `choose` statement. This requires some thinking on our part. What number can we plug in for `m` such that doubling it is equal to `x + y`? Given what we know about `a` and `b`, the answer is `a + b`. We conclude the proof by using the equations for `x` and `y` and the distributivity property of multiplication over addition (from `Nat.pf`).

```
choose a + b
suffices 2*a + 2*b = 2*(a + b) with rewrite x_2a | y_2b
symmetric dist_mult_add[2][a,b]
```

Figure 5 shows the complete proof of `addition_of_evens`.

To summarize this section:

- The `some` formula expresses that a property is true for at least one entity.
- Deduce's `obtain` statement lets you make use of a fact that is a `some` formula.
- To prove a `some` formula, use Deduce's `choose` statement.

```
theorem addition_of_evens:
  all x:Nat, y:Nat.
  if Even(x) and Even(y) then Even(x + y)
proof
  arbitrary x:Nat, y:Nat
  suppose even_xy: Even(x) and Even(y)
  have even_x: some m:Nat. x = 2*m
      by definition Even in even_xy
  have even_y: some m:Nat. y = 2*m
      by definition Even in even_xy
  obtain a where x_2a: x = 2*a from even_x
  obtain b where y_2b: y = 2*b from even_y
  suffices some m:Nat. x + y = 2*m with definition Even
  choose a + b
  suffices 2*a + 2*b = 2*(a + b) with rewrite x_2a | y_2b
  symmetric dist_mult_add[2][a,b]
end
```

Figure 5: Proof of the `addition_of_evens` theorem.