Software Foundations in Athena

June 9, 2024

Chapter 2

Basics

2.1 Introduction

This chapter develops the same material that is presented in the corresponding chapter ("Basics", Chapter 2) of the SF book, but in Athena. This includes solutions for most of the exercises in the SF book. In addition, it introduces new material that is not present in the SF book.

2.2 Enumerated Types

Here is how we introduce a data type to represent the days of the week:

```
datatype Day := Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

This is an inductively generated type that is also a free algebra. Its defining axioms can be seen and inserted into the assumption base as follows:

```
assert (datatype-axioms "Day")
```

Let's introduce some handy variables for this sort. We'll also load the files `prolog-solver` and `nat-plus`, since we'll use that functionality:

```
load "nat-plus"
load "prolog-solver"
define [d d' d1 d2 d3] := [?d:Day ?d':Day ?d1:Day ?d2:Day ?d3:Day]
```

The following defines the `nextWeekday` function introduced in Section 2.1 of SF:

```
declare nextWeekday: [Day] -> Day

assert* nextWeekday-def :=
   [(nextWeekday Monday    = Tuesday)
    (nextWeekday Tuesday   = Wednesday)
    (nextWeekday Wednesday = Thursday)
    (nextWeekday Thursday  = Friday)
    (nextWeekday Friday    = Monday)
    (nextWeekday Saturday  = Monday)
    (nextWeekday Sunday    = Monday)]
```

Let's now `eval` a few terms to make sure the definition works properly:

```
> (eval nextWeekday Monday)

Term: Tuesday
```

```
> (eval nextWeekday nextWeekday Tuesday)

Term: Thursday
```

Using a relational approach (based on logic programming), we can evaluate/solve backwards as well, e.g., we can answer the following question: which day is the predecessor of the predecessor of Wednesday?

```
> (Prolog.auto-solve nextWeekday nextWeekday ?d = Wednesday)

List: [true {?d:Day --> Monday}]
```

Here is our first proof—showing deductively that the twice-next weekday after Monday is Wednesday:

```
> conclude (nextWeekday nextWeekday Monday = Wednesday)
    (!chain [(nextWeekday nextWeekday Monday)
           = (nextWeekday Tuesday)                  [nextWeekday-def]
           = Wednesday                              [nextWeekday-def]])

Theorem: (= (nextWeekday (nextWeekday Monday))
            Wednesday)
```

There are many different ways in which this trivial result could be proved, some of which are stylistic variants of the above proof and others are more substantially different. In terms of style, the **conclude** annotation is optional and could have been ommitted:

```
(!chain [(nextWeekday nextWeekday Monday)
       = (nextWeekday Tuesday)                  [nextWeekday-def]
       = Wednesday                              [nextWeekday-def]])
```

Also, the two equational steps could have been collapsed into one, since Athena will attempt arbitrarily long rewriting chains:

```
(!chain [(nextWeekday nextWeekday Monday) = Wednesday [nextWeekday-def]])
```

Of course, this trivial result could also be easily proved via external ATPs:

```
(!prove (nextWeekday nextWeekday Monday = Wednesday) (ab))
```

Let's introduce a more generic `next`-day function that produces the following day of any given day, week or weekend:

```
declare next: [Day] -> Day

assert* next-def := [(next Monday    = Tuesday)
                     (next Tuesday   = Wednesday)
                     (next Wednesday = Thursday)
                     (next Thursday  = Friday)
                     (next Friday    = Saturday)
                     (next Saturday  = Sunday)
                     (next Sunday    = Monday)]

> (eval next next next Friday)

Term: Monday
```

Let's now introduce an *addition* operation on days, so that, for example, adding 1 to `Monday` gives `Tuesday`, adding 2 to `Friday` gives `Sunday`, and so on:

```
declare day-add: [Day N] -> Day [+ [id int->nat]]

# We've overloaded the + symbol and also made it accept integer numerals
# in the second argument position. We define this function axiomatically as follows:
```

```
assert* day-add-def := [(d + 0 = d)
                        (d + S n = next (d + n))]

> (eval Monday + 4)

Term: Friday

> (Prolog.auto-solve ?d + 3 = Sunday)

List: [true {?d:Day --> Thursday}]
```

Now here is a conjecture: Day addition is periodic with a period of 6:

```
define conjecture := (forall d . d + 6 = d)
```

We can *test* this conjecture automatically using model-checking, up to a bound of let's say 100:

```
> (falsify conjecture 100)

List: ['success |{?d:Day := Monday}|]
```

So `Monday` is a counterexample to the conjecture. How come?

```
> (eval Monday + 6)

Term: Sunday
```

Of course, the correct period is 7, not 6:

```
define conjecture := (forall d . d + 7 = d)

# Now we're no longer able to falsify it:

> (falsify conjecture 100)

Term: 'failure
```

Can we *prove* this conjecture deductively? Yes, in a number of different ways. For example:

```
define premises := [day-add-def N.Plus.Plus-def nextWeekday-def]

datatype-cases conjecture {
  Monday    => (!chain [(Monday + 7)    = Monday    [premises]])
| Tuesday   => (!chain [(Tuesday + 7)   = Tuesday   [premises]])
| Wednesday => (!chain [(Wednesday + 7) = Wednesday [premises]])
| Thursday  => (!chain [(Thursday + 7)  = Thursday  [premises]])
| Friday    => (!chain [(Friday + 7)    = Friday    [premises]])
| Saturday  => (!chain [(Saturday + 7)  = Saturday  [premises]])
| Sunday    => (!chain [(Sunday + 7)    = Sunday    [premises]])
}
```

However, this required us to think carefully about the exact set of assumptions that are needed for the result. Alternatively, we can let Athena figure those out on its own. And to cut down our typing, we define a procedure that builds the desired statement for any given day automatically:

```
retract conjecture

define (periodic d) := (d + 7 = d)

> datatype-cases conjecture {
  Monday    => (!chain [(Monday + 7)    = Monday])
| Tuesday   => (!chain [(Tuesday + 7)   = Tuesday])
```

```
| Wednesday => (!chain [(Wednesday + 7) = Wednesday])
| Thursday  => (!chain [(Thursday + 7)  = Thursday])
| Friday    => (!chain [(Friday + 7)    = Friday ])
| Saturday  => (!chain [(Saturday + 7)  = Saturday])
| Sunday    => (!chain [(Sunday + 7)    = Sunday])
}

Theorem: (forall ?d:Day
           (= (day-add ?d:Day
                       (S (S (S (S (S (S (S zero))))))))
              ?d:Day))
```

That is cleaner, but it still contains a lot of repetition. Let's write one single **`datatype-cases`** method for `Day` to rule them all. This will be a binary method that takes any goal g of the form

$$(\forall\, d\colon \texttt{Day}\ .\ \mathit{body})$$

and an input method M that takes any particular day d' (constructor of `Day`) and derives the conclusion obtained from *body* by replacing every free occurrence of d by d':

```
define (auto-dt-for-Day goal M) :=
  datatype-cases goal {
    Monday =>    (!M Monday)
  | Tuesday =>   (!M Tuesday)
  | Wednesday => (!M Wednesday)
  | Thursday =>  (!M Thursday)
  | Friday =>    (!M Friday)
  | Saturday =>  (!M Saturday)
  | Sunday =>    (!M Sunday)
  }
```

Using this generic method, we can now express the above proof in one single line:

```
retract conjecture

> (!auto-dt-for-Day conjecture (method (x) (!chain [(x + 7) = x])))

Theorem: (forall ?d:Day
           (= (day-add ?d:Day
                       (S (S (S (S (S (S (S zero))))))))
              ?d:Day))
```

Exercise 1

Give a constructive definition of a *precedence* relation on `Day`, call it `prec`, which holds between two days d_1 and d_2 iff d_1 comes "before" d_2. Thus, for example, Monday should precede every day other than itself, but Thursday should not precede, say Tuesday.[a] Your definition should not use any auxiliary symbols other than those already defined above, and it should pass all of the following tests:

```
(eval Monday prec Tuesday)    # true
(eval Monday prec Wednesday)  # true
(eval Monday prec Thursday)   # true
(eval Monday prec Sunday)     # true
(eval Monday prec Monday)     # false
(eval Saturday prec Sunday)   # true
(eval Saturday prec Monday)   # false
(eval Saturday prec Friday)   # false
```

(*Hint*: Use the function `next`.)

[a] More precisely: If you were to assign an integer code $code(d)$ between 1 and 7 to any day d by mapping Monday to 1, Tuesday to 2, and so on, then d_1 should precede d_2 iff $code(d_1) < code(d_2)$.

2.3 Booleans

Athena already has Booleans are already built into Athena (the datatype `Boolean`), but let's go through the exercise of building them from scratch:

```
datatype Bool := T | F

assert (datatype-axioms "Bool")

define [b b' b1 b2 b3] := [?b ?b' ?b1 ?b2 ?b3]
```

We can now introduce the three main logical operations: negation, conjunction, and disjunction:

```
declare notb: [Bool] -> Bool [--]

declare andb: [Bool Bool] -> Bool [/\]
declare orb:  [Bool Bool] -> Bool [\/]
```

We overload `--` for negation, `/\` for conjunction, and `\/` for disjunction. We start by defining negation:

```
assert* notb-def := [(-- T = F)
                     (-- F = T)]

> (eval -- T)

Term: F

> (eval -- -- F)

Term: F
```

Let's prove that negation is involutive: `(forall b . -- -- b = b)`. We can do this with a simple structural analysis:

```
conclude notb-involutive := (forall b . -- -- b = b)
  datatype-cases notb-involutive {
    T => (!chain [(-- -- T) = (-- F) [notb-def]
                             = T      [notb-def]])
  | F => (!chain [(-- -- F) = (-- T) [notb-def]
                             = F      [notb-def]])
  }
```

If we are so inclined, we can collapse both two-step chains into single-step chains:

```
retract notb-involutive

conclude notb-involutive := (forall b . -- -- b = b)
  datatype-cases notb-involutive {
    T => (!chain [(-- -- T) = T [notb-def]])
  | F => (!chain [(-- -- F) = F [notb-def]])
  }
```

We proceed with conjunction and disjunction:

```
assert* andb-def := [(T /\ b = b)
                     (F /\ _ = F)]
```

```
assert* orb-def := [(T \/ _ = T)
                    (F \/ b = b)]

> (eval T /\ T /\ F /\ T)

Term: F

> (eval F \/ F \/ T \/ F)

Term: T

> (eval F \/ F \/ F \/ F)

Term: F
```

Let's prove some simple consequences, e.g., that `T` is also a right identity:

```
conclude andb-lemma-1 := (forall b . b /\ T = b)
  datatype-cases andb-lemma-1 {
    T => (!chain [(T /\ T) = T [andb-def]])
  | F => (!chain [(F /\ T) = F [andb-def]])
  }

conclude andb-lemma-2 := (forall b . b /\ F = F)
  datatype-cases andb-lemma-2 {
    T => (!chain [(T /\ F) = F [andb-def]])
  | F => (!chain [(F /\ F) = F [andb-def]])
  }
```

We go on to prove that conjunction is commutative (this is

```
conclude conjunction-comm := (forall b1 b2 . b1 /\ b2 = b2 /\ b1)
  datatype-cases conjunction-comm {
    T => pick-any b2: Bool
           (!chain [(T /\ b2)
                  = b2          [andb-def]
                  = (b2 /\ T)   [andb-lemma-1]])
  | F => pick-any b2: Bool
           (!chain [(F /\ b2)
                  = F           [andb-def]
                  = (b2 /\ F)   [andb-lemma-2]])
  }
```

Exercise 2: *(SF Exercise 3.0.1)*
Define the binary function `nandb` as specified in SF, p. 14.

Solution

```
declare nandb: [Bool Bool] -> Bool

assert* nandnb-def := [(b1 nandb b2 = -- b1 /\ b2)]

> [(eval T nandb F) (eval F nandb T) (eval F nandb F) (eval T nandb T)]

List: [T T T F]
```

Exercise 3: *(SF Exercise 3.0.2)*
Define the ternary function `andb3` so that it returns `T` iff all three inputs are `T`.

Solution

```
declare andb3: [Bool Bool Bool] -> Bool

assert* andb3-def := [((andb3 T T T) = T)
                      ((andb3 F _ _) = F)
                      ((andb3 _ F _) = F)
                      ((andb3 _ _ F) = F)]

> (eval andb3 T F T)

Term: F

> (eval andb3 T T T)

Term: T
```

2.4 Function Types

Since Athena is based on first-order logic, it does not have function types.

2.5 Modules

Athena has a simple module system, described in Chapter 7 of FPMICS.

2.6 Numbers

This section of the SF books call for building up natural numbers for scratch. Since we have already imported natural numbers from the Athena library (by loading smttnat-plus.ath), we will put all of our new work into a module named `Nat` to avoid any conflicts. We start by defining natural numbers as a free algebra (a **`datatype`**), asserting the relevant axioms, and defining some variables along with some handy conversions to and from nonnegative integers:

```
module Nat {

datatype Nat := Zero | (Succ Nat)
assert (datatype-axioms "Nat")

define [n m k n1 n2 m1 m2 k1 k2] := [?n:Nat ?m:Nat ?n1:Nat ?n2:Nat
                                     ?m1:Nat ?m2:Nat ?k:Nat ?k1:Nat ?k2:Nat]

# Some convenient notational transformations from and to integer numerals:

define (int->Nat n) :=
  check {
    (integer-numeral? n) => check {
                              (n less? 1) => Zero
                            | else => (Succ (int->Nat (n minus 1)))
```

```
                                }
  | else => n
  }

define (Nat->Int n) :=
  match n {
    Zero => 0
 | (Succ k) => (plus 1 (Nat->Int k))
 | _ => n
}

transform-output eval [Nat->Int]
```

We also introduced an output transformation for `eval` so that it prints out natural number using integer numerals.

We introduce a predecessor function `pred` in the way it is defined in the SF book, as a total function, so that the predecessor of `Zero` is taken to be `Zero`. We overload `--` to be an alias for `pred`:

```
declare pred: [Nat] -> Nat [[int->Nat]]
overload -- pred

assert* pred-def := [(-- Zero = Zero)
                     (-- Succ n = n)]

> (eval -- 4)

Term: 3

> (eval -- -- 4)

Term: 2
```

Here's the `minusTwo` function defined in SF on p. 16:

```
declare minusTwo: [Nat] -> Nat [[int->Nat]]

assert* minusTwo-def := [(minusTwo Zero = Zero)
                         (minusTwo Succ Zero = Zero)
                         (minusTwo Succ Succ n = n)]

> (eval minusTwo 5)

Term: 3
```

And the recursive definition of even parity:

```
declare even, odd: [Nat] -> Boolean [[int->Nat]]

assert* even-def := [(even 0)
                     (~ even 1)
                     (even Succ Succ n <==> even n)]

> (eval even 5)

Term: false

> (eval even 100)

Term: true
```

The definition of `odd` is simpler:

```
assert* odd-def := [(odd n <==> ~ even n)]

> (eval odd 101)

Term: true
```

At this point the SF text goes to introduce binary addition, multiplication, and subtraction on the natural numbers:

```
declare plus-nat:  [Nat Nat] -> Nat [+ [int->Nat int->Nat]]
declare mult-nat:  [Nat Nat] -> Nat [* [int->Nat int->Nat]]
declare minus-nat: [Nat Nat] -> Nat [- [int->Nat int->Nat]]

assert* plus-nat-def := [(0 + m = m)
                         ((Succ n) + m  = Succ (n + m))]

assert* mult-nat-def := [(_ * Zero = Zero)
                         (n * Succ m = n + (n * m))]

assert* minus-nat-def := [(Zero - _ = Zero)
                          (n - Zero = n)
                          ((Succ n) - (Succ m) = n - m)]

> (eval 2 + 3)

Term: 5

> (Prolog.auto-solve 2 + ?What = int->Nat 5)

List: [true {?What:Nat --> (Succ (Succ (Succ Zero)))}]

> (eval 7 - 4)

Term: 3

> (Prolog.auto-solve 5 - ?What = int->Nat 2)

List: [true {?What:Nat --> (Succ (Succ (Succ Zero)))}]

> (eval 3 * 5)

Term: 15

> (Prolog.auto-solve 3 * ?What = int->Nat 15)

List: [true {?What:Nat --> (Succ (Succ (Succ (Succ (Succ Zero)))))}]
```

Note that division is automatically performed by constraint solving through logic programming.

And here is their factorial definition in Athena:

```
declare fact: [Nat] -> Nat [[int->Nat]]

assert* fact-def := [(fact Zero = Succ Zero)
                     (fact Succ n = (Succ n) * fact n)]

> (eval fact 5)

Term: 120
```

Note that it is not necessary to define equality explicitly on the natural numbers as is done in Idris. For the

same of the example we give the definition here, but then explain why it is not necessary:

```
declare eq: [Nat Nat] -> Boolean [[int->Nat int->Nat]]

assert* eq-def := [(0 eq 0)
                   (~ 0 eq Succ _)
                   (~ Succ _ eq 0)
                   (Succ n eq Succ m <==> m eq n)]

> (eval 5 eq 5)

Term: true

> (eval 5 eq 4)

Term: false
```

The reason why this is not necessary is because Athena has a built-in definition of equality for every free algebra defined in it, which is essentially syntactic identity:

```
> (eval Succ Succ Zero = Succ Succ Zero)

Term: true

> (eval Succ Succ Zero = Succ Zero)

Term: false
```

We continue with the definition of `lte`. We overload the symbol `<=` to serve as an alias for `lte`:

```
declare lte: [Nat Nat] -> Boolean [<= [int->Nat int->Nat]]

assert* lte-def := [(0 <= _)
                    (~ Succ _ <= 0)
                    (Succ n <= Succ m <==> n <= m)]

> (eval 2 <= 3)

Term: true

> (eval 3 <= 2)

Term: false

> (eval 3 <= 3)

Term: true
```

Exercise 4: *(SF Exercise 6.0.2, pp. 19-20)*
Define a function `blt` on the natural numbers that implements the less-than relation.

Solution

```
declare blt: [Nat Nat] -> Boolean [< [int->Nat int->Nat]]

assert* blt-def := [(n < m <==> n <= m & n =/= m)]

> (eval 3 < 4)
```

```
Term: true

> (eval 3 < 2)

Term: false
```

Note that we could have also defined `blt` as follows: `(n < m <==> n <= m & ~m <= n).`

Foo bar.

2.7 Proofs by Simplification

Finally, some proofs:

```
define left-zero := (forall n . 0 + n = n)

# We can actually prove this automatically:

> (!prove left-zero (ab))

Theorem: (forall ?n:Nat
           (= (plus-nat Zero ?n:Nat)
              ?n:Nat))
```

If we were to do it manually, however:

```
> conclude left-zero
    pick-any n:Nat
      (!chain [(0 + n) = n [plus-nat-def]])

Theorem: (forall ?n:Nat
           (= (plus-nat Zero ?n:Nat)
              ?n:Nat))
```

2.8 Proof by Rewriting

We consider the following theorem:

```
define plus-id-example := (forall n m . n = m ==> n + n = m + m)

pick-any n:Nat m:Nat
  assume n-is-m := (n = m)
    (!chain-> [(n = m)
           ==> (n + n = m + n)   [fcong]
           ==> (n + n = m + m)   [n-is-m]])
```

The method `fcong` (for "function congruence") is built into Athena.[1]

Exercise 5: *(SF Exercise 8.0.1, p. 21)*
Prove: `(forall n m k .  n = m ==> m = k ==> n + m = m + k).`

Solution

```
conclude (forall n m k . n = m ==> m = k ==> n + m = m + k)
```

[1] See p. 122 of FPMICS.

```
pick-any n:Nat m:Nat k:Nat
  assume (n = m)
    assume (m = k)
     # (t = t) always holds by reflexivity, so it can be the first step
     # of a strict chain:
      (!chain-> [(n + n = n + n)
             ==> (n + m = n + m)   [(n = m)]
             ==> (n + m = m + m)   [(n = m)]
             ==> (n + m = m + k)   [(m = k)]])
```

Exercise 6: *(SF Exercise 8.0.2, p. 22)*

Prove: `(forall n m .  m = Succ n ==> m * (1 + n) = m * m).`

Solution

```
conclude (forall n m . m = Succ n ==> m * (1 + n) = m * m)
  pick-any n:Nat m:Nat
    assume hyp := (m = Succ n)
      (!chain [(m * (1 + n))
             = (m * (Succ (0 + n))) [plus-nat-def]
             = (m * (Succ n))       [plus-nat-def]
             = (m * m)              [hyp]])
```

Foo bar.

2.9 Proof by Case Analysis

Consider the following statement:

```
define plus-1-is-not-zero := (forall n . n + 1 =/= Zero)
```

In Athena this can be proved automatically. This time we use Vampire with a time limit of 100 seconds:

```
> (!vpf plus-1-is-not-zero (ab) 100)

Theorem: (forall ?n:Nat
           (not (= (plus-nat ?n:Nat
                             (Succ Zero))
                   Zero)))
```

But suppose we wanted to do it manually, and at a fairly low level of detail. Let's first write a general method that takes a premise of the form `(t = Succ _)` and derives the conclusion `(t =/= Zero)`:

```
define (not-zero premise) :=
  match premise {
    (t = (rhs as (Succ _))) =>
      (!chain-> [true ==> (Zero =/= rhs)    [(datatype-axioms "Nat")]
                      ==> (Zero =/= t)      [premise]
                      ==> (t =/= Zero)])
  }
```

With this method under our belt, we can prove `plus-1-is-not-zero` by a simple structural analysis of the universally quantified variable:

```
datatype-cases plus-1-is-not-zero {
  Zero => (!chain-> [true
```

```
                ==> (Zero + Succ Zero = Succ Zero)          [plus-nat-def]
                ==> (Zero + Succ Zero =/= Zero)             [not-zero]])
| (Succ k) =>
    (!chain-> [true
           ==> ((Succ k) + Succ Zero = Succ (k + Succ Zero)) [plus-nat-def]
           ==> ((Succ k) + Succ Zero =/= Zero)               [not-zero]])
}
```

This concludes this section (recall that we proved that negation is involutive and conjunction is commutative earlier).

2.10 Structural Recursion (Optional)

Unlike Idris, in Athena there is no requirement that all function definitions must terminate.

2.11 More Exercises

Exercise 7: *(SF Exercise 11.0.1, p. 25)*
Supposing you have a function from the Booleans to the Booleans:

```
declare f: [Bool] -> Bool
```

prove the following conjecture:

```
define conjecture := (forall b . f b = b ==> f f b = b)
```

Solution
This is trivial to prove automatically, but let's do it manually:

```
conclude conjecture
  pick-any b:Bool
    assume hyp := (f b = b)
      (!chain [(f f b)
             = (f b)    [hyp]
             = b        [hyp]])
```

Exercise 8: *(SF Exercise 11.0.2, p. 25)*
Prove the following: `(forall b1 b2 .  b1 / b2 = b1  b2 ==> b1 = b2).`

Solution
This is also trivial to prove automatically, but let's try to do it the hard way. We start with a couple of lemmas:

```
conclude bool-struc-lemma-1 := (forall b . b = T <==> b =/= F)
   datatype-cases bool-struc-lemma-1 {
     T => (!equiv assume (T = T) (!claim (T =/= F))
                 assume (T =/= F) (!reflex T))
   | F => (!equiv
             assume hyp := (F = T)
               (!chain-> [hyp
                    ==> (T = F)
                    ==> (T = F & T =/= F) [augment]
```

```
                    ==> false              [prop-taut]
                    ==> (F =/= F)          [(method (_ q) (!from-false q))]])
              assume hyp := (F =/= F)
                (!from-complements (F = T) (!reflex F) hyp))
    }

conclude bool-struc-lemma-2 := (forall b . b = F <==> b =/= T)
  pick-any b:Bool
    (!chain [(b = F) <==> (b =/= T) [bool-struc-lemma-1]])

conclude bool-struc-lemma :=
           (forall b1 b2 . b1 =/= b2 ==> b1 = T & b2 = F | b1 = F & b2 = T)
  pick-any b1:Bool b2:Bool
    assume hyp := (b1 =/= b2)
      (!cases (!chain<- [(b1 = T | b1 = F) <== true [(datatype-axioms "Bool")]])
        assume case1 := (b1 = T)
          (!chain-> [case1
                ==> (T =/= b2)                          [hyp]
                ==> (b2 =/= T)
                ==> (b2 = F)                            [bool-struc-lemma-2]
                ==> (b1 = T & b2 = F)                   [augment]
                ==> (b1 = T & b2 = F | b1 = F & b2 = T)  [alternate]])
        assume case2 := (b1 = F)
          (!chain-> [case2
                ==> (F =/= b2)                          [hyp]
                ==> (b2 =/= F)
                ==> (b2 = T)                            [bool-struc-lemma-1]
                ==> (b1 = F & b2 = T)                   [augment]
                ==> (b1 = T & b2 = F | b1 = F & b2 = T)  [alternate]]))
```

Exercise 9

Here is the phrasing of the exercise from SF:

> Consider a different, more efficient representation of natural numbers using a binary rather than unary system. That is, instead of saying that each natural number is either zero or the successor of a natural number, we can say that each binary number is either
>
> - zero,
> - twice a binary number, or
> - one more than twice a binary number.
>
> (a) First, write an inductive definition of the type `Bin` corresponding to this description of binary numbers.
>
> (b) Next, write an increment function incr for binary numbers, and a function `bin_to_nat` to convert binary numbers to unary numbers.
>
> (c) Write five unit tests `test_bin_incr_1`, `test_bin_incr_2`, etc. for your increment and binary-to-unary functions. Notice that incrementing a binary number and then converting it to unary should yield the same result as first converting it to unary and then incrementing.

Do all of the above in Athena.

Solution

```
datatype Bin := Z | (Twice Bin) | (Twice+1 Bin)
assert (datatype-axioms "Bin")

define [n m] := [?n:Bin ?m:Bin]

declare inc: [Bin] -> Bin

assert* inc-def := [(inc Z = Twice+1 Z)
                    (inc Twice n = Twice+1 n)
                    (inc Twice+1 n = Twice inc n)]

declare bin->nat: [Bin] -> Nat.Nat

assert* bin->nat-def := [(bin->nat Z = Nat.Zero)
                         (bin->nat Twice n = 2 Nat.* bin->nat n)
                         (bin->nat Twice+1 n = 1 Nat.+ 2 Nat.* bin->nat n)]

define one := (Twice+1 Z)

# This should give 5. Read it from right to
# left: inc(twice(twice(one))) = 1 + 2 * (2 * 1)

(eval bin->nat inc Twice Twice one)

declare nat->bin: [Nat.Nat] -> Bin [[int->nat]]

assert* nat->bin-def := [(nat->bin Nat.Zero = Z)
                         (nat->bin Nat.Succ x = inc nat->bin x)]

(eval nat->bin Nat.Succ Nat.Succ Nat.Succ Nat.Succ Nat.Zero)
```

Chapter 3

Induction: Proof by Induction

3.1 Proof by Induction

In the last chapter we proved that `Zero` is a neutral left element for addition. It is also a neutral element for addition on the right:

```
define plus-n-Z := (forall n . n + 0 = n)
```

However, proving this requires induction.

In Athena, this can be proved automatically with the built-in method `induction*`:

```
> (!induction* plus-n-Z)

Theorem: (forall ?n:Nat
           (= (plus-nat ?n:Nat Zero)
              ?n:Nat))
```

But here is a *declarative* proof of it (make sure to retract `plus-n-Z` first):

```
by-induction plus-n-Z {
  Zero => (!chain [(Zero + Zero) = Zero   [plus-nat-def]])
| (Succ k) => let {ih := (k + 0 = k)}
                  # ih is our inductive hypothesis; it's already in the a.b.
                (!chain [((Succ k) + 0)
                       = (Succ (k + 0))   [plus-nat-def]
                       = (Succ k)         [ih]])
}
```

This proof could be shortened, e.g., we could omit the explicit justifications:

```
by-induction plus-n-Z {
  Zero     => (!chain [(Zero + Zero) = Zero])
| (Succ k) => (!chain [((Succ k) + 0) = (Succ (k + 0)) = (Succ k)])
}
```

Here is the corresponding proof in Idris:

```
plus_n_Z : (n : Nat) -> n = n + 0
plus_n_Z Z = Refl
plus_n_Z (S k) =
  let inductiveHypothesis = plus_n_Z k in
    rewrite inductiveHypothesis in Refl
```

Note that this is a purely *procedural* proof script, not a declarative proof. It is not at all clear how the rewriting is done—what the rewriting steps are or how they are justified.

```
conclude minus-diag := (forall n . n - n = Zero)
  by-induction minus-diag {
    Zero => (!chain [(Zero - Zero) = Zero          [minus-nat-def]])
  | (Succ k) => let {ih := (k - k = Zero)}
                  (!chain [((Succ k) - (Succ k))
                         = (k - k)                 [minus-nat-def]
                         = Zero                    [ih]])
  }
```

Exercise 10: *(SF Exercise 1.0.1, p. 28)*

Prove the following results:

```
define mult-0-r :=   (forall n . n * Zero = Zero)

define plus-n-Sm :=  (forall n m . Succ (n + m) = n + Succ m)

define plus-comm :=  (forall n m . n + m = m + n)

define plus-assoc := (forall n m k . n + (m + k) = (n + m) + k)
```

Use any previously derived results as needed.

Solution

```
conclude mult-0-r := (forall n . n * Zero = Zero)
  by-induction mult-0-r {
    Zero     => (!chain [(Zero * Zero) = Zero  [mult-nat-def]])
  | (Succ k) => let {ih := (k * Zero = Zero)}
                  (!chain [((Succ k) * Zero)
                         = (Zero + (k * Zero)) [mult-nat-def]
                         = (k * Zero)          [plus-nat-def]
                         = Zero                [ih]])
  }

conclude plus-n-Sm := (forall n m . Succ (n + m) = n + Succ m)
  by-induction plus-n-Sm {
    (n as Zero) => pick-any m:Nat
                     (!chain [(Succ (Zero + m))
                            = (Succ m)          [plus-nat-def]
                            = (Zero + Succ m)   [plus-nat-def]
                            = (n + Succ m)      [(n = Zero)]])
  | (Succ k) =>
      let {ih := (forall m . Succ (k + m) = k + Succ m)}
        pick-any m:Nat
          (!chain [(Succ ((Succ k) + m))
                 = (Succ (Succ (k + m)))        [plus-nat-def]
                 = (Succ (k + Succ m))          [ih]
                 = ((Succ k) + (Succ m))        [plus-nat-def]])
  }

conclude plus-comm := (forall n m . n + m = m + n)
  by-induction plus-comm {
    Zero => pick-any m:Nat
              (!chain [(Zero + m)
                     = m                        [plus-nat-def]
                     = (m + Zero)               [plus-n-Z]])
  | (Succ k) => let {ih := (forall m . k + m = m + k)}
```

```
                  pick-any m:Nat
                    (!chain [((Succ k) + m)
                           = (Succ (k + m))     [plus-nat-def]
                           = (Succ (m + k))     [ih]
                           = (m + Succ k)       [plus-n-Sm]])
  }


conclude plus-assoc := (forall n m k . n + (m + k) = (n + m) + k)
  by-induction plus-assoc {
    Zero => pick-any m:Nat k:Nat
              (!chain [(Zero + (m + k))
                     = (m + k)                  [plus-nat-def]
                     = ((Zero + m) + k)         [plus-nat-def]])
  | (Succ n') =>
      let {ih := (forall m k . n' + (m + k) = (n' + m) + k)}
        pick-any m:Nat k:Nat
          (!chain [((Succ n') + (m + k))
                  = (Succ (n' + (m + k)))       [plus-nat-def]
                  = (Succ ((n' + m) + k))       [ih]
                  = ((Succ (n' + m)) + k)       [plus-nat-def]
                  = (((Succ n') + m) + k)       [plus-nat-def]])
  }
```

Exercise 11: *(SF Exercise 1.0.2, p. 28)*

Consider:

```
declare double: [Nat] -> Nat [[int->Nat]]

assert* double-def := [(double Zero = Zero)
                       (double Succ n = Succ Succ double n)]
```

Prove the following: `(forall n .  double n = n + n).`

Solution

```
conclude double-plus := (forall n . double n = n + n)
  by-induction double-plus {
    Zero => (!chain [(double Zero)
                   = Zero                       [double-def]
                   = (Zero + Zero)              [plus-nat-def]])
  | (Succ k) => (!chain [(double Succ k)
                       = (Succ Succ double k)   [double-def]
                       = (Succ Succ (k + k))    [(double k = k + k)] # Ind. hypothesis
                       = (Succ ((Succ k) + k))  [plus-nat-def]
                       = (Succ (k + Succ k))    [plus-comm]
                       = ((Succ k) + (Succ k))  [plus-nat-def]])
  }
```

Exercise 12: Foo Bar

Prove the following (informally, in English): `(forall n .  even n <==> ~even Succ n).`

Solution

We use structural induction. When `n` is `Zero`, we need to show `(~even Succ Zero)`, which follows directly from the definition of `even`.

For the inductive case, assume that `n` is of the form `(Succ k)`, so that our inductive hypothesis becomes

$$\texttt{(even k <==> ~even Succ k)} \tag{3.1}$$

We now need to derive the following conditional:

$$\texttt{(even Succ k <==> ~even Succ Succ k).} \tag{3.2}$$

In the left-to-right direction, assume `(even Succ k)`. Then, from the inductive hypothesis (3.1) we infer

$$\texttt{(~even k).} \tag{3.3}$$

But, by the definition of `even`, we have

$$\texttt{(even Succ Succ k <==> even k),} \tag{3.4}$$

thus (3.3) becomes the desired conclusion `(~even Succ Succ k)`.

For the right-to-left direction of the goal (3.2), assume `(~even Succ Succ k)`. By using (3.4) again (the definition of `even`), this assumption yields `(~even k)`, and applying the inductive hypothesis (3.1) to `(~even k)` yields the desired conclusion `(even Succ k)`.

Exercise 13: *(SF Exercise 1.0.3, p. 29)*

Prove the same result in Athena: `(forall n . even n <==> ~even Succ n).`

Solution

```
by-induction evenb_S {
  Zero => (!chain [(even Zero)
              <==> (~ even Succ Zero)                    [even-def]])
| (Succ k) => let {ih := (even k <==> ~ even Succ k)}
                (!chain [(even Succ k)
                    <==> (~ even k)                      [ih]
                    <==> (~ even Succ Succ k)            [even-def]])
}
```

Note that the exact formulation of `evenb_s` in SF (as given on p. 29) is equivalent to the above formulation and follows from it immediately:

```
conclude (forall n . even Succ n <==> ~ even n)
  pick-any n:Nat
    (!chain [(even Succ n) <==> (~ even n) [evenb_S]])
```

3.2 Proofs Within Proofs

This section in the SF book appears to be about forcing assertions, which in Athena would be done via the **`force`** keyword.

In terms of composite proofs: In Coq/Idris, it seems that the main way to develop proofs is via automated

rewriting and lemmas. This would be equivalent in style to using ATPs (automated theorem provers) in Athena and simply decomposing a top-level goal into a series of lemmas that could actually be handled by the ATPs. All of the "proofs" in such an approach would be 1-line proofs of the following form:

```
define goal := ...

conclude lemma-1 := (!prove ...)

conclude lemma-n := (!prove ...)

conclude goal := (!prove goal [lemma-1 ... lemma-n])
```

One can of course introduce additional theorem-proving tactics in Idris, though these again seem to be procedural instructions for how to get the built-in tactics to behave properly, not declarative proofs. Consider:

```
define plus-rearrange := (forall n1 n2 m1 m2 . (n1 + n2) + (m1 + m2) = (n2 + n1) + (m1 +
    m2))
```

This is really just a single application of commutativity of addition. Here is the Idris proof (as appears on p. 30):

```
plus_rearrange : (n, m, p, q : Nat) ->
                 (n + m) + (p + q) = (m + n) + (p + q)
plus_rearrange n m p q = rewrite plus_rearrange_lemma n m in Refl
  where
  plus_rearrange_lemma : (n, m : Nat) -> n + m = m + n
  plus_rearrange_lemma = plus_comm
```

Here is the Athena version:

```
conclude plus-rearrange
  pick-any n1:Nat n2:Nat m1:Nat m2:Nat
    (!chain [((n1 + n2) + (m1 + m2))
            = ((n2 + n1) + (m1 + m2))   [plus-comm]])
```