7 Organizing Theory Development with Athena Modules

MODULES are used to partition Athena code into separate namespaces. We begin by describing how to introduce a module and how to refer to names introduced inside a module. Some examples are drawn from a recasting of some of the natural number development in Chapter 3 using modules. We further introduce nested modules and illustrate their use as a convenient way of encapsulating axioms and theorems for function symbols. Brief examples are then given to illustrate a few additional module features, and the chapter closes with a note on indentation conventions for modules.

7.1 Introducing a module

A module with name M is introduced by the following syntax:

module M $\{\cdots\}$

The region between the matching curly braces is called the *scope* of the module. The contents of that scope can consist of any nonempty sequence of any of the Athena directives we have seen so far: sort declarations, datatype or value definitions, and so on, possibly including other module definitions (called *nested* modules). Here is an example:

```
> module A {

    define x := 'foo

    define L := [1 2 3]

  }

Term A.x defined.

List A.L defined.

Module A defined.
```

Module `A` does two simple things: It defines `x` as the term `'foo` and `L` as the list `[1 2 3]`. To retrieve one of these values after the module has been defined, we use its *qualified name*, expressed using the dot notation: We write the name of the module, followed by a dot, followed by the name that was given to the value inside the module.

```
> A.x

Term: 'foo

> A.L
```

```
List: [1 2 3]

> A.z

input prompt:1:1: Error: Could not find a value for A.z.
```

An error was reported when we tried to evaluate `A.z`, since the name `z` was not defined inside `A`. The definitions given inside a module M can be inspected by the procedure `module->string`, which takes as input the name of a module and returns a properly formatted and indented printed representation of the module's bindings:

```
> (print (module->string "A"))

A --> module { ### 2 bindings
        x --> 'foo
        L --> [
                1
                2
                3
               ]
      }

Unit: ()
```

As this example illustrates, all names introduced inside a module are by default visible to the outside, once the module has been successfully defined. Accordingly, it is not necessary to specify any "export" directives; everything is automatically exported. However, there is a way to override this default behavior by prefixing a definition with the **`private`** keyword. A private definition will not be visible outside of the module in which it appears.

```
> module A {

    private define a := 99

    define b := (a plus 1)
  }

Term A.b defined.

Module A defined.

> A.b

Term: 100

> A.a

input prompt:1:1: Error: Could not find a value for A.a.
```

When a module is defined, its contents are processed sequentially, from top to bottom, so forward references are not meaningful and are flagged as errors:

```
> module A {
    define x := foo

    define foo := 3
  }

input prompt:3:15: Error: Could not find a value for foo.
Module A was not successfully defined.
```

This is similar to the top-down way in which **load** processes the contents of a file. Of course, mutually recursive definitions are still possible using **letrec**. Also, as this last example demonstrates, when an error occurs during the processing of a module's definition, the definition is considered to have failed, and therefore no module will be created as a result of the (attempted) definition. If the module whose definition failed is embedded inside other module definitions, then all those outer module definitions will fail as well.[1]

7.2 Natural numbers using modules

To show some of these module features in actual use, and to help introduce other features, let us redo some of the development of the natural number datatype and functions:

```
datatype N := zero | (S N)

assert (datatype-axioms "N")

module N {
  declare one, two: N

  define [m n n' x y z k] := [?m:N ?n:N ?n':N ?x:N ?y:N ?z:N ?k:N]

  assert one-definition := (one = S zero)
  assert two-definition := (two = S one)

  define S-not-zero     := (forall n . S n =/= zero)
  define one-not-zero   := (one =/= zero)
  define injective      := (forall m n . S m = S n <==> m = n)

} # close module N
```

1 When a module definition fails, the global lexical environment, the global assumption base, and the global symbol set are reverted to the values they had prior to the definition of the outermost module containing the failing definition. That is, Athena will transitively roll back any intermediate effects to these three semantic parameters that might have come about as a result of processing the module definition(s) up to the point of failure. However, side effects to the store (and obviously, any output side effects) will not be undone.

Here the module name is N, the same identifier we used to name the natural number datatype. We could have used a different identifier to name the module; using the same name is just a matter of convenience. If we write a proof outside the scope of module N, we use qualified names; for example:

```
> conclude N.S-not-zero
    pick-any n
      (!chain->
        [true ==> (zero =/= S n) [(datatype-axioms "N")]
              ==> (S n =/= zero) [sym]])

Theorem: (forall ?n:N
           (not (= (S ?n)
                   zero)))
```

We can in turn use this sentence, since it is now a theorem, in the proof of N.one-not-zero.

```
> (!by-contradiction N.one-not-zero
     assume (N.one = zero)
       let {is := conclude (S zero = zero)
                    (!chain [(S zero)
                             = N.one                [N.one-definition]
                             = zero                 [(N.one = zero)]]);
            is-not := (!chain-> [true
                                 ==> (S zero =/= zero) [N.S-not-zero]])}

     (!absurd is is-not))

Theorem: (not (= N.one zero))
```

If we had included the proofs within the scope of the module, we could have use unqualified names. Another way to write the proofs using unqualified names is to use an **open** directive:

```
open N

conclude S-not-zero
  pick-any n
    (!chain->
      [true ==> (zero =/= S n) [(datatype-axioms "N")]
            ==> (S n =/= zero) [sym]])

(!by-contradiction one-not-zero
   assume (one = zero)
     let {is := conclude (S zero = zero)
                  (!chain
                    [(S zero)
                     = one          [one-definition]
                     = zero         [(one = zero)]]);
          is-not := (!chain->
```

```
                    [true
                  ==> (S zero =/= zero) [S-not-zero]])}
    (!absurd is is-not))
```

open M brings all the names introduced inside module M into the current scope, so that they may be used without qualification. Multiple **open** directives may be issued, and several modules $M_1, \ldots, M_n$ can be opened with a single **open** directive:

$$\textbf{open}\ M_1, \ldots, M_n$$

However, every time we **open** a module we increase the odds of naming confusion and conflicts, so excessive use of this directive is discouraged.

7.3 Extending a module

Modules are dynamic, in the sense that one can reopen a module's scope and enter additional content into it at any time. This is done with **extend-module**; for example,

```
extend-module N {

  define nonzero-S :=
    (forall n . n =/= zero ==> exists m . n = S m)

  define S-not-same := (forall n . S n =/= n)

  by-induction nonzero-S {
    zero => assume (zero =/= zero)
              (!from-complements (exists m . zero = S m)
                                 (!reflex zero)
                                 (zero =/= zero))
  | (S k) => assume (S k =/= zero)
                 let {_ := (!reflex (S k))}
                   (!egen (exists m . S k = S m) k)
  }

  by-induction S-not-same {
    zero => conclude (S zero =/= zero)
              (!instance S-not-zero zero)
  | (S m) => let {ihyp := (S m =/= m)}
               (!chain-> [ihyp
                      ==> (S S m =/= S m) [injective]])
  }

} # close module N
```

In this case we have included the proofs within the module and are thus able to use unqualified names.

7.4 Modules for function symbols

Although we could continue to put natural number properties directly into module `N`, another alternative is to create a *nested module* within module `N` for properties of each function symbol that we specify. For example, we can begin to do so for addition of natural numbers, as follows:

```
extend-module N {
  declare +: [N N] -> N
  module Plus {

    assert right-zero    := (forall n . n + zero = n)
    assert right-nonzero := (forall m n . n + S m = S (n + m))

  } # close module Plus
} # close module N
```

Within module `Plus`, we can use the unqualified names `right-zero` and `right-nonzero`. Outside of module `Plus`, but within module `N`, we must use names qualified with `Plus`: `Plus.right-zero` and `Plus.right-nonzero`. And outside of `N`, we must use fully qualified names `N.Plus.right-zero` and `N.Plus.right-nonzero`.

We now extend `Plus` with a couple of sentences that we intend as theorems:

```
extend-module N {
  extend-module Plus {
    define left-zero    := (forall n . zero + n = n)
    define left-nonzero := (forall n m . (S m) + n = S (m + n))
  } # close module Plus
} # close module N
```

Exercise 7.1: Prove

`N.Plus.left-zero`

using `N.Plus.right-zero` and `N.Plus.right-nonzero`. □

Exercise 7.2: Derive `N.Plus.left-nonzero` from the same two axioms. □

We could continue with additional extensions and corresponding proofs. See the file `lib/main/nat-plus.ath`, where such a development is carried out. For multiplication, see the development of a `Times` module in `lib/main/nat-times.ath`.

7.5 Additional module features

Lexical scoping applies as usual, so new definitions inside a module overshadow earlier definitions in everything that follows, including any subsequent modules:

```
module M1 {
  define x := 2

  module M2 {

    define y := (x plus 3)
    define x := 99

    module M3 {

      define y := (x plus 1)

    } # close module M3

  } # close module M2

} # close module M1

> M1.M2.M3.y

Term: 100
```

Note that **open** is transitive. If a module M_1 is opened inside M_2, and then M_2 is opened inside M_3, M_1 will also be opened inside M_3:

```
module M1 {
  define a1 := 1
}

module M2 {
  open M1
  define a2 := (a1 plus 2)
}

module M3 {
  open M2
  define a3 := (a2 plus a1)
}

> M3.a3

Term: 4
```

The value of a name *I* defined at the top level can always be retrieved by writing `Top.`*I*. For instance:

```
define foo := 2

module M {
  define foo := 3;
  define y := (foo plus Top.foo)
}

> M.y

Term: 5
```

`Top` is not quite a proper module (it would have to contain itself otherwise!), but it can be treated as such for most practical purposes. For instance,

```
(print (module->string "Top"))
```

produces the expected results. To ensure that `Top` is always treated distinctly and consistently, a user-defined module cannot be named `Top`.[2]

7.6 Additional module procedures

We close by describing a few more primitive procedures that may be useful in working with modules.

1. The unary procedure `module-size` takes the name of a module, as a string, and returns the number of bindings in the corresponding module.
2. The unary procedure `module-domain` takes the name of a module, as a string, and returns a list of all and only those identifiers, represented as strings, that are defined inside the module. For instance:

```
module M {
  define [x y] := [1 2]
}

> (map-proc println (module-domain "M"))

x

y

Unit: ()
```

2 Of course `Top` can still be used to name a value, as module names and value-denoting identifiers comprise different namespaces.

3. The binary procedure `apply-module` is a programmatic version of the dot notation. It takes the name of a module M and any identifier I defined inside M and produces the corresponding value, $M.I$. The module name can itself be nested, written in dot notation:

```
module M1 { module M2 { define L := [] }}

> (apply-module "M1.M2" "L")

List: []
```

All of the above procedures generate an error if their first string argument does not represent a module.

7.7 A note on indentation

In the brief examples of modules in this chapter, we have indented the contents, which is consistent with a general policy of indenting code inside curly braces. In practice, however, strictly adhering to this policy for modules can become inconvenient, as the contents of a module often extend over hundreds or thousands of lines. Consequently, in the rest of the book and in the Athena libraries, we indent the contents of modules in short examples, but otherwise we just start at the first character of the line.