CSCI.6962/4962 Software Verification— Fundamental Proof Methods in Computer Science (Arkoudas and Musser)—Chapter 7

Instructor: Carlos Varela
Rensselaer Polytechnic Institute
Spring 2018

Organizing Theory Development with Athena Modules

Goal: to study **modular** development of theories.

- introducing a module
- natural numbers using modules
- extending a module
- modules for function symbols
- additional module features
- additional module procedures

Introducing a module

Modules are used to partition Athena code into separate namespaces. A module with name M is introduced with the following syntax:

$$\texttt{module}\ M\ \{\cdots\}$$

- 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, e.g.:
  - sort declarations,
  - datatype or value definitions,
  - other module definitions (called *nested* modules).

Introducing a module

A module 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]`.

Introducing a module

To retrieve one of these values after the module has been defined, we use its *qualified name*, expressed using the dot notation:

```
> 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`.

Introducing a module

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: ()
```

Introducing a module

It is possible to prefix 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.
```

Introducing a module

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:2:15: Error: Could not find a value for foo.
Module A was not successfully defined.
```

Mutually recursive definitions are still possible using `letrec`.
When a module definition fails, the lexical environment, assumption base, and symbol set are reverted to their values prior to the (failed) module definition.

Natural numbers using modules

Consider 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)
  define no-junk        := (forall n . n = zero | exists m . n = S m)

} # close module N
```

Natural numbers using modules

The module name is N, the same as the datatype, for 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)))
```

Natural numbers using modules

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))
```

Natural numbers using modules

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, e.g.:

```
open N

(!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))
```

Natural numbers using modules

- 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:

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

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

Extending a module

Modules are dynamic—one can reopen a module's scope and enter additional content into it at any time, with `extend-module`, e.g.:

```
extend-module N {

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

  conclude nonzero-S
    pick-any n
      assume (n =/= zero)
        (!dsyl (!uspec no-junk n) (n =/= zero))

} # close module N
```

Extending a module

```
extend-module N {

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

  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 these cases, we have included the proofs within the module and are thus able to use unqualified names.

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
```

Modules for function symbols

- 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 the fully qualified names
  - `N.Plus.right-zero` and
  - `N.Plus.right-nonzero`.

Modules for function symbols

We now extend `Plus` with a couple of sentences to prove:

```
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
```

We can 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`.

Additional module features

Static scoping applies as usual, so new definitions inside a module overshadow earlier definitions in everything that follows:

```
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
```

Additional module features

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
```

Additional module features

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, we can use:

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

Additional module procedures

- The unary procedure `module-size` takes the name of a module and returns the number of bindings.
- The unary procedure `module-domain` takes the name of a module and returns a list of all the identifiers defined inside the module:

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

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

x

y

Unit: ()
```

Additional module procedures

- 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.