CSCI.6962/4962 Software Verification—Fundamental Proof Methods in Computer Science (Arkoudas and Musser)—Chapter 2.8-2.10

Instructor: Carlos Varela

Rensselaer Polytechnic Institute

Spring 2018

Introduction to Athena

Goal: to become familiar with Athena language

- *interacting with Athena*
- *domains and function symbols*
- *terms*
- *sentences*
- *definitions*
- *assumption bases*
- *datatypes*
- polymorphism
- meta-identifiers
- expressions and deductions

Datatype for natural numbers

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

This defines a datatype `N` with two constructors:

- `zero` is a constant constructor.
- `S` is a unary constructor.
  - Because the argument of `S` is of the same sort as its result, `N`, we say that `S` is a *reflexive* constructor.
  - Thus, `S` requires an element of `N` as input in order to construct another such element as output.

Datatype for natural numbers

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

A datatype defines a unique recursive set. In the case of `N`, that set is given by the following rules:

1. `zero` is an element of `N`.
2. For all n, if n is an element of `N`, then `(S` n`)` is an element of `N`.
3. Nothing else is an element of `N`.

The last clause ensures minimality of the defined set, i.e., the *only* elements of `N` are those that can be obtained by the first two clauses, namely, *by a finite number of constructor applications* (No "junk").

Natural numbers—Peano's axioms

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

The following are the free-generation axioms for `N`:

```
(forall ?n . zero =/= S ?n)

(forall ?n ?m . S ?n = S ?m ==> ?n = ?m)

(forall ?n . ?n = zero | exists ?m . ?n = S ?m)
```

The first two are no-confusion axioms:

1. `zero` is different from every application of `S` (i.e., `zero` is not the successor of any natural number),
2. applications of `S` to different arguments produce different results (i.e., `S` is injective).

The third axiom says `zero` and `S` span the domain `N` (no-junk.)

Structures

Not all inductively defined sets are freely generated, e.g.:

```
datatype Set := null | (insert Int Set)
```

Two sets are identical iff they have the same members, e.g., $\{1, 3\} = \{3, 1\}$. We must then be able to prove

```
((insert 3 (insert 1 null)) = (insert 1 (insert 3 null))).
```

But one of the no-confusion axioms is that `insert` is injective:

```
(forall ?i1 ?s1 ?i2 ?s2 . (insert ?i1 ?s1) = (insert ?i2 ?s2)
              ==> ?i1 = ?i2 & ?s1 = ?s2).
```

These are inconsistent as they would allow us to conclude `(1 = 3)`.

Structures

Instead we define `Set` as a `structure` rather than a `datatype`:

```
structure Set := null | (insert Int Set)
```

- A *structure* is a datatype with a coarser identity relation.
- A structure is also inductively generated by its constructors, so structural induction (via `by-induction`) is also available.
- The only difference is that there may be some “confusion,”: the constructors might not be injective. We might even obtain the same value by applying two distinct constructors.
- We need to assert a proper identity relation for a structure, e.g.:

```
(forall ?s1 ?s2 . ?s1 = ?s2 <==>
                      ?s1 subset ?s2 & ?s2 subset ?s1)
```

Structure/datatype axioms

Unary procedures `datatype-axioms` and `structure-axioms` return all the inductive axioms for a datatype or a structure, e.g.:

```
> (structure-axioms "Set")

List: [
(forall ?y1:Int
  (forall ?y2:Set
    (not (= null
            (insert ?y1:Int ?y2:Set)))))

(forall ?v:Set
  (or (= ?v:Set null)
      (exists ?x1:Int
        (exists ?x2:Set
          (= ?v:Set
             (insert ?x1:Int ?x2:Set))))))
]
```

Polymorphic domains

A domain can be polymorphic.

For example, consider sets over an arbitrary universe, call it `S`:

```
> domain (Set S)

New domain Set introduced.
```

- In general, $(I \;\; I_1 \cdots I_n)$ introduces polymorphic domain I.
- The identifiers $I_1, \ldots, I_n$ are parameters that serve as *sort variables* in this context, indicating that I is a *sort constructor* that takes any n sorts $S_1, \ldots, S_n$ as arguments and produces a new sort as a result, namely $(I \;\; S_1 \cdots S_n)$.
    - For instance, `Set` is a unary sort constructor that can be applied to an arbitrary sort, say the domain `Int`, to produce the sort `(Set Int)`.

Polymorphic domains

- For uniformity, monomorphic sorts such as `Person` and `N` can be regarded as nullary sort constructors.
- Polymorphic datatypes and structures can also serve as sort constructors.
- The following are example sorts (among infinitely many) over $\boldsymbol{SC} = \{\mathtt{Int}, \mathtt{Boolean}, \mathtt{Set}\}$ and $\boldsymbol{SV} = \{\mathtt{S1}, \mathtt{S2}\}$:

```
Int,
(Set Boolean),
S1,
(Set S2),
(Set (Set Int)),
(Set (Set S1)).
```

Sort relations and identity

- A *ground* (or *monomorphic*) sort is one that contains no sort variables. A sort that is not ground is said to be *polymorphic*.
  - `Int`, `(Set Boolean)`, and `(Set (Set Int))` are ground.
  - `S1`, `(Set S2)`, and `(Set (Set S1))` are polymorphic.
- A *sort valuation* τ is a function from sort variables to sorts. It can be extended to a function $\widehat{\tau}$ from sorts over ***SC*** and ***SV*** to sorts over ***SC*** and ***SV***.
- A sort S_1 is an *instance of* (or *matches*) a sort S_2 iff there exists a sort valuation τ such that $\widehat{\tau}(S_2) = S_1$. Two sorts S_1 and S_2 are *unifiable* iff there exists a sort valuation τ such that $\widehat{\tau}(S_1) = \widehat{\tau}(S_2)$.
- Two sorts are considered *identical* iff they differ only in their variable names, e.g., `(Set S1)` is identical to `(Set S2)`.

Polymorphic function symbols

The general syntax form for declaring a polymorphic function symbol f is

$$\texttt{declare } f\texttt{: (}I_1, \ldots, I_n\texttt{) [}S_1 \cdots S_n\texttt{] -> } S$$

- $I_1, \ldots, I_n$ are distinct identifiers that serve as sort variables,
- S_i is the sort of the i^{th} argument, and
- S is the sort of the result.

For example:

```
declare in: (S) [S (Set S)] -> Boolean

declare union: (S) [(Set S) (Set S)] -> (Set S)

declare =: (S) [S S] -> Boolean

declare empty-set: (S) [] -> (Set S)
```

Polymorphic terms

Athena automatically infers the most general possible polymorphic sorts for every variable occurrence, e.g.:

```
> ?x
Term: ?x:'T175

> (?x in ?y)
Term: (in ?x:'T203
              ?y:(Set 'T203))

> (?a = ?b)
Term: (= ?a:'T206 ?b:'T206)

> (?x in ?y:(Set (Set 'T)))
Term: (in ?x:(Set 'T209)
              ?y:(Set (Set 'T209)))
```

Polymorphic sentences

A polymorphic sentence contains at least one polymorphic term, or a quantified variable with a nonground sort, e.g.:

```
> (forall ?x . ?x = ?x)
Sentence: (forall ?x:'S
            (= ?x:'S ?x:'S))

> (forall ?x ?y . ?x union ?y = ?y union ?x)
Sentence: (forall ?x:(Set 'S)
            (forall ?y:(Set 'S)
              (= (union ?x:(Set 'S)
                        ?y:(Set 'S))
                 (union ?y:(Set 'S)
                        ?x:(Set 'S)))))

> (~ exists ?x . ?x in empty-set)
Sentence: (not (exists ?x:'S
                 (in ?x:'S
                     empty-set:(Set 'S))))
```

Parametric polymorphism

A polymorphic function symbol f can be thought of as a collection of monomorphic function symbols, each of which can be viewed as an instance of f. For example:

```
declare in: (S) [S (Set S)] -> Boolean
```

can be thought of as

```
declare in_Int: [Int (Set Int)] -> Boolean

declare in_Real: [Real (Set Real)] -> Boolean

declare in_Boolean: [Boolean (Set Boolean)] -> Boolean

declare in_(Set Int): [(Set Int) (Set (Set Int))] -> Boolean
```

and so on for infinitely more ground sorts.

Parametric polymorphism

A polymorphic sentence such as:

```
(forall ?x . ?x = ?x)
```

can also be seen as a collection of (potentially infinitely many) monomorphic sentences, namely:

```
(forall ?x:Int . ?x = ?x),
(forall ?x:Boolean . ?x = ?x),
(forall ?x:(Set Int) . ?x = ?x),
```

and so on.

- This expressivity is the power of parametric polymorphism.
- A single polymorphic sentence can express infinitely many propositions about infinitely many sets of objects.

Polymorphic datatypes

Since datatypes are special kinds of structures, which are special kinds of domains, they can also be polymorphic, e .g.:

```
datatype (List S) := nil | (:: S (List S))

datatype (Pair S T) := (pair S T)
```

- In general, $(I \;\; I_1 \cdots I_n)$ introduces polymorphic datatype I.
- The identifiers $I_1, \ldots, I_n$ serve as local *sort variables*, indicating that I is a *sort constructor* that takes any n sorts $S_1, \ldots, S_n$ as arguments and produces a new sort as a result, namely $(I \;\; S_1 \cdots S_n)$.
  - For instance, `Pair` is a binary sort constructor that can be applied to any two arbitrary sorts, to produce a new sort, e.g., `(Pair Int (List Boolean))`.

Integers and reals

Athena comes with two predefined numeric domains:

- `Int` for integers, e.g., `47`, `(- 5)`, `0`
- `Real` for real numbers, e.g., `3.14`, `0.158`, `2.3`.

There are five predeclared binary function symbols:

- `+` (addition),
- `-` (subtraction),
- `*` (multiplication),
- `/` (division), and
- `%` (remainder).

Integers and reals

Function symbols are overloaded so that they can be used both with integers and with reals, or indeed with any combination thereof:

```
> (?x + 2)

Term: (+ ?x:Int 2)

> (2.3 * ?x)

Term: (* 2.3 ?x:Real)
```

These symbols adhere to the usual precedence and associativity conventions:

```
> (2 * 7 + 35)

Term: (+ (* 2 7)
         35)
```

Integers and reals

The subtraction symbol can be used both with one and with two arguments:

```
> (- 2)

Term: (- 2)

> (7 - 5)

Term: (- 7 5)
```

- As a unary symbol it represents integer/real negation, and as a binary symbol it represents subtraction.
- Likewise, + can be used both as a unary and as a binary symbol.

Integers and reals

There are also function symbols for usual comparison operators:

- < (less than),
- > (greater than),
- <= (less than or equal to), and
- >= (greater than or equal to).

```
> (forall ?x . ?x + 1 > ?x)

Sentence: (forall ?x:Int
            (> (+ ?x:Int 1)
               ?x:Int))
```

Integers and reals

Function symbols for comparison operators are likewise overloaded:

```
> (forall ?x:Real . ?x + 1 > ?x)

Sentence: (forall ?x:Real
            (> (+ ?x:Real 1)
               ?x:Real))

> (forall ?x . ?x + 1.0 > ?x)

Sentence: (forall ?x:Real
            (> (+ ?x:Real 1.0)
               ?x:Real))
```

Numeric procedures

There are also predefined procedures for performing the usual computations with numbers: `plus`, `minus`, `times`, `div`, `mod`, `less?`, `greater?`, and `equal?`.

These are not function symbols, that is, they do not make terms but actually perform the underlying computations:

```
> (1 plus 2 times 3)

Term: 7

> (100 div 2)

Term: 50
```

`equal?` is not equal to `=`

The *procedure* `equal?` (also defined as `equals?`) is a generic equality test that can be applied to any two values, not just numbers.

```
> (3.0 div 1.5 equal? 2)

Term: true
```

Do not confuse it with the *polymorphic function symbol* `=`:

```
> (x = x)

Term: (= ?x:'T10961 ?x:'T10961)

> (x equal? x)

Term: true
```

`equal?` is not equal to `=`

`equal?` can be used to compare terms and sentences:

```
> (equal? (= x x) (= y y))

Term: false

> (equal? (forall x . x = x) (forall y . y = y))

Term: true
```

If two sentences are alpha-variants—i.e., if you can get one from the other by alpha-renaming—, then they are considered equal.

Meta-identifiers

- Domains such as `Int` and `Real`, and datatypes such as `Boolean` are pre-defined in Athena. Another built-in domain is `Ide`, the domain of *meta-identifiers*.
- There are infinitely many constants pre-declared for `Ide`. These are all of the form `'`I, where I is a regular Athena identifier. For example:

```
'foo
'x
'233
'*
'sd8838jd@!
```

These are called meta-identifiers.

Meta-identifiers

Examples:

```
> 'x

Term: 'x

> (println (sort-of 'x))

Ide

Unit: ()

> (exists ?x . ?x = 'foo)

Sentence: (exists ?x:Ide
            (= ?x:Ide 'foo))
```

Meta-identifiers can represent the variables of some object language whose abstract syntax is modeled by an Athena structure.

Meta-identifier Example

Consider the untyped λ-calculus. An expression is either:

- a variable (identifier), or
- an abstraction, or
- an application.

That abstract grammar could be represented by the following structure:

```
structure Exp := (Var Ide) | (Lambda Ide Exp) | (App Exp Exp)
```

Then the term

```
(Lambda 'x (Var 'x))
```

would represent the identity function.

Exercise: Why should `Exp` not be a datatype?

Expressions and deductions

The most basic kind of expression is a *procedure call* (application):

$$(E \; F_1 \cdots F_n)$$

- E is an expression whose value must be a procedure
- the arguments $F_1 \cdots F_n$, $n \geq 0$, are phrases whose values become the inputs to that procedure.

The most basic kind of deduction is a *method call* (application):

$$(\texttt{apply-method} \; E \; F_1 \cdots F_n) \text{ or } (!E \; F_1 \cdots F_n)$$

- E is an expression that must denote a *method* M
- the arguments $F_1 \cdots F_n$, $n \geq 0$, are phrases whose values become the inputs to M.

Simplest methods

The nullary method `true-intro` always results in the constant `true`, no matter what the assumption base is:

```
> (!true-intro)
Theorem: true
```

The unary reiteration method `claim` takes an arbitrary sentence p as input, and if p is in the assumption base, then it simply returns it back as the output:

```
assert true
> (!claim true)
Theorem: true
```

The result of any deduction D is always reported as a *theorem*, because the result of D is guaranteed to be a logical consequence of the assumption base in which D was evaluated.

Conjunction introduction

Conjunction introduction is performed by the binary method `both`.

- takes any two sentences p and q, and provided that both of them are in the assumption base (up to alpha-equivalence),
- produces the conclusion `(and` p q`)`

```
declare A, B, C: Boolean

assert A, B

> (!both A B)

Theorem: (and A B)
```

Conjunction elimination

Conjunction elimination is performed by the two unary methods `left-and` and `right-and`.

```
clear-assumption-base

assert (A & B)

> (!left-and (A & B))
Theorem: A

> (!right-and (A & B))
Theorem: B

> (!right-and (C & B))
Error, standard input, 1.2: Failed application of right-and---the sentence
(and C B) is not in the assumption base.
```

Double negation elimination

Another unary primitive method is dn, which performs double-negation elimination.

- takes a premise p of the form (not (not q)), and provided that p is in the assumption base,
- returns q

```
assert p := (~ ~ A)

> (!dn p)

Theorem: A
```

Nested method calls

```
clear-assumption-base
assert conj := (A & (B & C))

> (!left-and (!right-and conj))
Theorem: B
```

In general, every time a deduction appears as an argument to a method call, the conclusion of that deduction will appear (temporarily) in the assumption base in which the method will be applied:

```
> (ab)
List: [
(and A
     (and B C))

B
]
```

Let expressions and deductions

The most common form of the `let` construct is:

$$\texttt{let}\ \{I_1 \texttt{ := } F_1;\ \cdots\ ; I_n \texttt{ := } F_n\}\ \ F$$

- $I_1, \ldots, I_n$ are identifiers
- $F_1, \cdots, F_n$ and F are phrases.
- If F, the *body* of the `let` phrase, is an expression, then so is the whole `let` phrase. And if the body F is a deduction, then the whole `let` is also a deduction.

Let deduction

An example of a `let` deduction is:

```
assert hyp := (male peter & female ann)

> let { left  := (!left-and hyp);
        right := (!right-and hyp)
      }
    (!both right left)

Theorem: (and (female ann)
              (male peter))
```

Conclusion-annotated deductions

The general syntax is: `conclude` $[I\ \texttt{:=}\,]\,E\quad D$.

- D is an arbitrary deduction
- E is its intended (named) conclusion

The deduction D is evaluated and the conclusion E is checked:

```
assert p := (A & B)

> conclude A
    (!left-and p)
Theorem: A

> conclude B
    (!left-and p)
standard input:1:2: Error: Failed conclusion annotation.
The expected conclusion was: B
but the obtained result was: A.
```

Conditional expressions

The syntax of a `check` expression is

$$\texttt{check}\ \{F_1 \texttt{ => } E_1 \ \texttt{|}\ \cdots\ \texttt{|}\ F_n \texttt{ => } E_n\}$$

- F_i => E_i pairs are its *clauses*, with each clause consisting of a *condition* F_i and a corresponding *body* expression E_i.
- To evaluate a `check` expression,
  - we evaluate the conditions $F_1, \ldots, F_n$, in that order.
  - if F_i produces `true`, we evaluate and return the corresponding E_i.
  - The last condition, F_n, may be the keyword `else`, which is treated as though it were `true`.
  - It is an error if no F_i produces `true` and there is no `else` clause at the end.

Conditional deductions

The syntax of a check deduction is

$$\texttt{check}\ \{F_1\ \texttt{=>}\ D_1\ \mid\ \cdots\ \mid\ F_n\ \texttt{=>}\ D_n\}$$

The evaluation process is the same as for check expressions, but with deductions, e.g.:

```
assert A

> check {(holds? false) => 1 | (holds? A) => 2 | else => 3}

Term: 2
```

Pattern-matching expressions and deductions

A pattern-matching expression has the form

$$\texttt{match}\ F\ \{\pi_1\ \texttt{=>}\ E_1\ \texttt{|}\ \cdots\ \texttt{|}\ \pi_n\ \texttt{=>}\ E_n\}$$

- the phrase F is called the *discriminant*
- the π_i `=>` E_i pairs are the *clauses*, with each clause consisting of a *pattern* π_i and a corresponding *body* expression E_i.

It is evaluated in a given environment ρ and assumption base β :

- We first evaluate the discriminant F, obtaining from it a value V
- We then try to *match* V against $\pi_1, \ldots, \pi_n$, in that order. If we succeed in matching V against some π_i, resulting in a number of bindings, we go on to evaluate the corresponding body E_i (or D_i) in ρ augmented with these bindings, and in β.

Pattern-matching expressions and deductions

Examples:

```
> match [1 2] {
    [] => 99
  | (list-of h _) => h
  }

Term: 1

> match [1 2] {
    [] => (!claim false)
  | (list-of _ _) => (!true-intro)
  }

Theorem: true
```

Defining procedures

We can define our own procedures with the `lambda` construct, and then use them as if they were primitive procedures, e.g.:

```
> define square := lambda (n) (n times n)

Procedure square defined.

> square

Procedure: square (defined at standard input:1:32)

> (square 4)

Term: 16

> (map lambda (n) (n times n)
       [1 2 3 4 5])

List: [1 4 9 16 25]
```

Defining methods

Methods abstract over deductions, similarly to procedures over computations.

```
method (p q)
  let {_ := (!left-and (p & q));
       _ := (!right-and (p & q))}
    (!both q p)
```

This method can be applied to two arbitrary conjuncts p and q and will produce the conclusion

$$(q \ \& \ p),$$

provided that the premise $(p \ \& \ q)$ is in the assumption base.

Defining methods

While the method could be applied anonymously, it is more convenient to give it a name first:

```
clear-assumption-base
define commute-and :=
  method (p q)
    let {_ := (!left-and (p & q));
         _ := (!right-and (p & q))}
      (!both q p)
assert (B & C)

> (!commute-and B C)
Theorem: (and C B)

> (!commute-and A B)
standard input:3:15: Error: Failed application of left-and---the sentence
(and A B) is not in the assumption base.
```

Defining composable methods

- Method closures have static name scoping but *dynamic assumption scoping*, i.e., the method will evaluate in the assumption base present at the time of method application, not method definition.
- Therefore, it is best to use for method arguments, the premises it needs, so that when nesting method calls, the assumption base will have the right lemmas.
- For example, the `commute-and` method is better as

```
define commute-and' :=
  method (premise)
    match premise {
      (p & q) => let {_ := (!left-and premise);
                       _ := (!right-and premise)}
                   (!both q p)
    }
```

Defining composable methods

- For instance, suppose the assumption base contains `(~ (~ (A & B)))` and we want to derive `(B & A)`.
- Using the second version, we can express the proof in a single line by composing double negation and conjunction commutation:

```
assert premise :=  (~ ~ (A & B))

> (!commute-and' (!dn premise))

Theorem: (and B A)
```

Such composition is not possible with the former version.

Alternative procedure/method definition syntax

At the top level it is not necessary to define procedures with `lambda`. An alternative notation is the following:

```
> define (square n) := (n times n)

Procedure square defined.
```

or in more traditional Lisp notation:

```
(define (square n)
   (times n n))
```

Likewise with methods:

```
> define (commute-and p q) :=
    let {_ := (!left-and (p & q));
         _ := (!right-and (p & q))}
      (!both q p)

Method commute-and defined.
```

Alternative definition syntax

How can Athena tell the difference from a procedure in a method defined with syntax:

$$\texttt{define (}M \;\; I_1 \cdots I_n\texttt{) := } D,$$

In most cases, a deduction is indicated just by the leading keyword:

`apply-method` (usually written `!`)

`assume`	`with-witness`
`pick-any`	`suppose-absurd`
`pick-witness`	`conclude`
`pick-witnesses`	`by-induction`
`generalize-over`	`datatype-cases`