CSCI.6962/4962 Software Verification—Fundamental Proof Methods in Computer Science (Arkoudas and Musser)—Chapter 2.1-2.7

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
Athena

Athena is a language for expressing proofs and computations.

- it is *higher-order*, i.e., procedures are first-class values (can be passed or returned).
- it is *dynamically typed*, i.e., type checking at run-time.
- it has *cells* and *vectors* for state update.
- it uses *lists*, which are *heterogeneous* (arbitrary types for values).
- similarly to the $\lambda$-calculus, uses *procedure calls* for flow control.
Athena’s Fundamental Data Values

Athena’s fundamental data values are *terms* and *sentences*.

- A *term* is a symbolic structure, essentially a tree whose every node contains either a *function symbol* or a *variable*. For example:
  - $3 + 5$, $x/2$, $78$, etc.
  - *Joe*, *Joe’s father*,...

Terms denote individual objects in some domain of interest.

- A *sentence* is essentially a formula of first-order logic: either an atomic formula, or a Boolean combination of formulas, or a quantification.

Sentences express propositions about domains of interest. They serve as the conclusion of proofs.
Athena’s Fundamental Syntactic Categories

Athena’s fundamental syntactic categories are *deductions* and *expressions*.

A phrase $F$ is either an expression or a deduction.

\[
E ::= \cdots \quad \text{(Expressions, for computing)}
\]
\[
D ::= \cdots \quad \text{(Deductions, for proving)} \tag{1}
\]
\[
F ::= E \mid D \quad \text{(Phrases)}
\]

Deductions, if successful, can only produce one type of value: a *sentence*, e.g., all prime numbers greater than 2 are odd.

Guaranteed to be logical consequence of assumptions at evaluation time.
Athena’s Phrase Evaluation

Athena evaluates *phrases* to produce *values*.

A phrase $F$ is evaluated as follows:

Input: Phrase $F$ → Evaluation (w.r.t. given $\rho, \beta, \sigma, \gamma$) → Output: Value $V$

- $\rho$ a lexical environment
- $\beta$ an assumption base
- $\sigma$ a store
- $\gamma$ a symbol set
Interacting with Athena

Athena can be used either in batch mode or interactively. Typing

\begin{verbatim}
load "file.ath"
\end{verbatim}

at the input prompt will process file.ath sequentially. The .ath extension can be omitted. The file can also be given as a command line argument to Athena. ; ; will signal end of input in multiple-line entry in interactive mode.
Domains and function symbols

A domain is simply a set of objects that we want to talk about. We can introduce one with the domain keyword. For example,

> domain Person

New domain Person introduced.

Multiple domains can be introduced with the domains keyword:

domains Element, Set

Domains are sorts. Function symbols denote operations on sorts, e.g.:

> declare father: [Person] -> Person

New symbol father declared.

[Person] -> Person is the signature of father.
Function symbols

Multiple function symbols with same signature can be declared separated by commas:

```
declare union, intersection: [Set Set] -> Set
```

```
declare father, mother: [Person] -> Person
```

A function symbol of arity zero is called a constant symbol, or simply a constant.

```
> declare joe: Person
```

New symbol joe declared.

```
> declare null: Set
```

New symbol null declared.

Function symbols are first-class data values. They are not procedures.
Function symbols

Multiple constant symbols of the same sort can be introduced by separating them with commas:

\[
\begin{align*}
declare & \text{peter, tom, ann, mary: Person} \\
declare & \text{e, e1,e2: Element} \\
declare & \text{S, S1, S2: Set}
\end{align*}
\]

true and false are constants of the built-in sort Boolean. The two numeric domains Int (integers) and Real (reals) are also built-in.

A function symbol whose range is Boolean is also called a *relation* (or *predicate*) symbol, or just “predicate” for short. Some examples:

\[
\begin{align*}
declare & \text{in: [Element Set] -> Boolean} \\
declare & \text{male, female: [Person] -> Boolean} \\
declare & \text{siblings: [Person Person] -> Boolean} \\
declare & \text{subset: [Set Set] -> Boolean}
\end{align*}
\]
Procedures

An Athena procedure is a lambda abstraction written by users to compute, e.g.:

```plaintext
define (fact n) :=
  check {
    (less? n 1) => 1
    | else => (times n (fact (minus n 1)))
  }
```

where `less?`, `times`, and `minus` are primitive procedures:

```plaintext
> (less? 7 8)
Term: true

> (times 2 3)
Term: 6

> (minus 5 1)
Term: 4
```
Terms

A term is a syntactic object that represents an element of some sort. The simplest term is a constant symbol:

> joe

Term: joe

We can ask Athena to print the sort of this (or any other) term:

> (println (sort-of joe))

Person

Unit: ()

Athena knows that joe denotes an individual in the domain Person.
**Terms**

A *variable* is also a term.

The following are all legal variables:

- `?x:Person`
- `?S25:Set`
- `?foo-bar:Int`
- `?b_1:Boolean`
- `?@sd%&:Real`

Constant symbols and variables are primitive or *simple* terms, with no internal structure.
Terms

More complex terms can be formed by applying a function symbol $f$ to $n$ given terms $t_1 \cdots t_n$, where $n$ is the arity of $f$.

Some examples of complex terms:

(father joe)
(father (father joe))
(in e S)
(union null S2)
(male (father joe))
(subset null (union ?X null))
Terms

root and children primitive procedures return the root symbol of an application and its children (as a list of terms, ordered from left to right). For example:

```
define t := (father (mother joe))

> (root t)

Symbol: father

> (children t)

List: [(mother joe)]
```
Lists

A list of $n \geq 0$ values $V_1 \cdots V_n$ can be formed simply by enclosing the values inside square brackets: $[V_1 \cdots V_n]$. For instance:

> [tom ann]

List: [tom ann]

> []

List: []

> [tom [peter mary] ann]

List: [tom [peter mary] ann]

Lists are heterogeneous, i.e., they may contain elements of different types.
Some operations on lists

Some operations on lists are \texttt{add}, \texttt{head}, \texttt{tail}, \texttt{length}, \texttt{rev}, and \texttt{join}.

\begin{verbatim}
> (add 1 [2 3])
List: [1 2 3]

> (head [1 2 3])
Term: 1

> (tail [1 2 3])
List: [2 3]

> (rev [1 2 3])
List: [3 2 1]

> (length [1 2 3])
Term: 3

> (join [1 2] ['a 'b] [3])
List: [1 2 'a 'b 3]
\end{verbatim}
Sort checking and inference

Athena checks for correctness of sorts in complex terms:

```lisp
> (father true)

standard input:1:2: Error: Unable to infer a sort for the term: (father true)

(Failed to unify the sorts Boolean and Person.)
```

It also performs Hindley-Milner-style sort inference. For instance:

```lisp
> (in ?x ?S)

Term: (in ?x:Element ?S:Set)
```

Notice that only terms have sorts. But terms are only one type of Athena value.

Other types include: sentences, lists, procedures, methods, the unit value, and more.
Infix form and precedence

Infix form is allowed in Athena, for example:

\[(e \text{ in } S)\]

\[(\text{null union } S2)\]

\[(\text{male father joe})\]

\[(\text{null subset } ?x \text{ union null})\]

Athena always prints output terms in full prefix form, as so-called “s-expressions”:

\[
> (\text{null union } ?s)
\]

Term: \((\text{union null } ?s: \text{Set})\)

By default, every predicate is given a precedence of 100, while other binary or unary function symbols are given a precedence of 110.
Sentences

• “the bread and butter of Athena”—every successful proof derives a sentence.

• There are three kinds:
  • Atomic sentences
  • Boolean combinations
  • Quantified sentences
Atomic Sentences

Atomic sentences, or just *atoms*. These are simply terms of sort Boolean.
Examples are:

(siblings peter (father joe))
(subset ?s1 (union ?s1 ?s2))
**Boolean Combinations**

Obtained from other sentences through one of the five *sentential constructors* not, and, or, if, and iff, or their synonyms, as shown in the following table.

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Synonym</th>
<th>Prefix mode</th>
<th>Infix mode</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>not</td>
<td>~</td>
<td>(not (p))</td>
<td>((\sim p))</td>
<td>Negation</td>
</tr>
<tr>
<td>and</td>
<td>&amp;</td>
<td>(and (p q))</td>
<td>((p &amp; q))</td>
<td>Conjunction</td>
</tr>
<tr>
<td>or</td>
<td></td>
<td>(or (p q))</td>
<td>((p</td>
<td>q))</td>
</tr>
<tr>
<td>if</td>
<td>==&gt;</td>
<td>(if (p q))</td>
<td>((p \Rightarrow q))</td>
<td>Conditional</td>
</tr>
<tr>
<td>iff</td>
<td>&lt;=&gt;</td>
<td>(iff (p q))</td>
<td>((p \iff q))</td>
<td>Biconditional</td>
</tr>
</tbody>
</table>
Quantified Sentences

A quantified sentence is of the form \((Q \ x:S \ . \ p)\) where \(Q\) is a quantifier, \(x:S\) is a variable of sort \(S\), and \(p\) is a sentence.

<table>
<thead>
<tr>
<th>Quantifier</th>
<th>Prefix</th>
<th>Infix</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall</td>
<td>(forall (x:S \ p))</td>
<td>(forall (x:S \ . \ p))</td>
<td>(p) holds for every (x:S)</td>
</tr>
<tr>
<td>exists</td>
<td>(exists (x:S \ p))</td>
<td>(exists (x:S \ . \ p))</td>
<td>(p) holds for some (x:S)</td>
</tr>
</tbody>
</table>

Examples are:

\[(forall \ ?x \ . \ ?x =/= father \ ?x)\]
\[(forall \ ?S1 \ ?S2 \ . \ ?S1 = ?S2 <==>
  \ ?S1 subset ?S2 & ?S2 subset ?S1)\]
**Definitions**

Definitions let us give a name to a value and then subsequently refer to the value by that name.

Top-level directive `define`’s syntax form is:

\[
\text{define } I := F
\]

where \( I \) is any identifier and \( F \) is a phrase denoting the value that we want to define.

> define p := (forall ?s . ?s subset ?s)

Sentence p defined.

> (p & p)

Sentence: (and (forall ?s:Set

  (subset ?s:Set ?s:Set))

  (forall ?s:Set

    (subset ?s:Set ?s:Set)))
Assumption bases

- At all times Athena maintains a global set of sentences called the *assumption base*.

- We can think of the elements of the assumption base as our premises—sentences that we regard (at least provisionally) as true.

- Initially the system starts with a small assumption base.

- Every time an axiom is postulated or a theorem is proved at the top level, the corresponding sentence is inserted into the assumption base.
Datatypes

- A datatype is a special kind of domain.
- It is special in that it is *inductively generated*, i.e., every element of the domain can be built up in a finite number of steps by applying *constructor* s of the datatype.
- A datatype $D$ is specified by giving its name, possibly followed by some sort parameters, and then a nonempty sequence of constructor profiles separated by the symbol $|$.
- A constructor profile without selectors is of the form

$$ (c \ S_1 \cdots S_n), $$

consisting of the name of the constructor, $c$, along with $n$ sorts $S_1 \cdots S_n$, where $S_i$ is the sort of the $i^{th}$ argument of $c$. 
Datatype examples

Boolean is a pre-defined datatype that has two constant constructors, true and false.

```
datatype Boolean := true | false
```

The intended effect of this datatype definition could be approximated in terms of mechanisms with which we are already familiar as follows:

```
domain Boolean

declare true, false: Boolean

assert (true =/= false)

assert (forall ?b:Boolean . ?b = true | ?b = false)
```

These are *free-generation* axioms: the first is known as *no-confusion*, and the second is known as *no-junk* in universal algebra.