Athena Basics: a short tutorial

October 27, 2022

1.1 Terms and Sentences

1.1.1 Terms

The main ingredients of proofs are *terms* and *sentences*, so this is where we begin. A term is a syntactic object (essentially a tree, as we will see soon) that represents an element of some *sort*. The simplest sorts are *domains*, which are introduced as follows:

```
>domain Person

New domain Person introduced.
```

There are a few built-in sorts, such as `Int`, `Real`, and `Ide`. Some sorts, called *datatypes*, have a special inductive structure. The built-in sort `Boolean` is a very simple example of a datatype. We'll discuss datatypes in another tutorial.

Once a sort has been introduced, we can build terms of that sort. The simplest kind of term is a *variable*, an identifier starting with a question mark and possibly followed by a sort declaration:

```
>?x:Person

Term: ?x:Person

>?Foo_Bar:Person

Term: ?Foo_Bar:Person
```

Variables that are not explicitly decorated with sorts are *polymorphic*; their sorts are automatically inferred by the surrounding context:

```
>?x

Term: ?x:'T26184

>(= ?x ?y)

Term: (= ?x:'T26192 ?y:'T26192)

>(= ?x:Person ?y)

Term: (= ?x:Person ?y:Person)
```

A sort like `'T26184` is a *sort variable* that represents an arbitrary sort.

More complex terms are built with *function symbols*, or *symbols* for short. A function symbol is introduced with a *signature declaration*:

```
declare father, mother: [Person] -> Person      # Two unary function symbols

declare sibling-of: [Person Person] -> Boolean  # A binary relation symbol

declare joe, mary: Person                       # Two constant symbols
```

A function symbol whose range (co-domain) is `Boolean` is called a *relation* or *predicate symbol*. Every constant function symbol is a term. Moreover, if f is a symbol with a signature `[`$S_1 \cdots S_n$`] -> `S and if t_i is a term of sort S_i, then `(`$f\ \ t_1 \cdots t_n$`)` is a term of sort S. These terms are called *applications*, since we are "applying" the symbol f to the terms $t_1 \cdots t_n$, which are viewed as the arguments supplied to f. Here is an example of applying `sibling-of` to `joe` and `mary` to make a more complex term:

```
>(sibling-of joe mary)

Term: (sibling-of joe mary)
```

If the signature of f is `[`$S_1 \cdots S_n$`] -> `S, we say that f has *arity* n; this is the number of terms that f expects as arguments. Symbols of arity zero (like `joe` and `mary`) are *constants*. If f is binary ($n = 2$), we can apply it using infix notation. Also, successive applications of unary symbols do not need intermediate parentheses. Finally, relation symbols typically bind less tightly (have smaller precedence) than other function symbols.

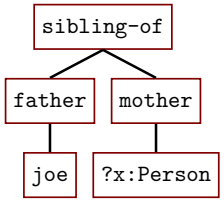
```
>(joe sibling-of mary)

Term: (sibling-of joe mary)

>(father mother joe)

Term: (father (mother joe))

>(father joe sibling-of mother ?x)

Term: (sibling-of (father joe)
                  (mother ?x:Person))
```

Note that terms are always output in prefix, as s-expressions, even if they were written in infix.

Terms can be viewed as trees. Variables and constants are leaves, whereas a term of the form `(`$f\ t_1 \cdots t_n$`)` for $n > 0$ can be understood as a tree with the symbol f at the root and with the trees corresponding to $t_1, \ldots, t_n$ as the subtrees of f, from left to right. For example, the term

```
(sibling-of (father joe) (mother ?x:Person))
```

can be understood as the following tree:



The primitive procedures `root` and `children` take a term t and return the root node of t and the subtrees of that root, respectively:

```
define t := (father joe sibling-of mother ?x)

>(root t)

Symbol: sibling-of

>(children t)

List: [
(father joe)

(mother ?x)
]
```

We'll have more to say about procedures and lists soon.

Some symbols are predefined. These include `true` and `false` (constants of sort `Boolean`), numerals like `3` and `(- 875)` (constants of sort `Int`), floats like `3.14` (constants of sort `Real`), so-called *meta-identifiers* like `'foo` (constants of sort `Ide`), function symbols for numeric operations like `+`, `-`, `*`, and so on, and also for numeric comparisons (`<`, `>`, `<=`, `>=`).

```
>(?x + 4)

Term: (+ ?x:Int 4)

>(?a + 2 * 3.14)

Term: (+ ?a:Real
         (* 2 3.14))
```

A binary equality symbol is also built-in; this is a *polymorphic* function symbol.

```
>(?x = joe)

Term: (= ?x:Person joe)
```

Polymorphic function symbols can be introduced by prefixing the signature with one or more *sort variables* listed within parentheses and separated by commas:

```
>declare P: (T) [T T] -> Boolean

New symbol P declared.

>(joe P mary)

Term: (P joe mary)

>(2 P 3)

Term: (P 2 3)
```

1.1.2 Sentences

Every proof derives a unique *sentence*. An atomic sentence (or "atom") is a term of sort `Boolean`, while a complex sentence is a negation (formed with the sentential constructor `not`), or a conjunction (formed with `and`), or a disjunction (formed with `or`), or a conditional or biconditional (formed with `if` and `iff`, respectively). The constructors `and`, `or`, `if`, and `iff` can be used in infix or prefix style:

```
>(?x:Int > 3) # an atomic sentence (a term of sort Boolean)

Term: (+ ?x:Int 4)

>(and (joe sibling-of mary) (?x > 4)) # a conjunction, written in prefix style

Sentence: (and (sibling-of joe mary)
               (> ?x:Int 4))


>(joe sibling-of mary or ?x > 4)      # a disjunction, in infix notation

Sentence: (or (sibling-of joe mary)
              (> ?x:Int 4))

>(if (?x > 4) (?x > 3))               # a conditional, in prefix notation

Sentence: (if (> ?x:Int 4)
              (> ?x:Int 3))

>(?x > 0 iff (- ?x) < 0)              # a biconditional, in infix notation
```

```
Sentence: (iff (> ?x:Int 0)
               (< (- ?x:Int)
                  0))
```

Also, the tilde ~ can be used interchangeably with `not`, `&` with `and`, `|` with `or`, `==>` with `if`, and `<==>` with `iff`:

```
>(2 > 3 & ~ joe = mary)

Sentence: (and (> 2 3)
               (not (= joe mary)))

>(?x < ?y | ?x = ?y | ?x > ?y)

Sentence: (or (< ?x:Real ?y:Real)
              (or (= ?x:Real ?y:Real)
                  (> ?x:Real ?y:Real)))

>(?x < ?y <==> ?y > ?x)

Sentence: (iff (< ?x:Real ?y:Real)
               (> ?y:Real ?x:Real))
```

The predefined procedure `=/=` can be used a short-hand to build negated equalities:

```
>(joe =/= mary ==> mary =/= joe)

Sentence: (if (not (= joe mary))
              (not (= mary joe)))
```

The sentences we have seen so far are called *zero-order* sentences, or propositional-logic sentences. First-order logic involves the use of the universal and existential *quantifiers*, which in Athena are named `forall` and `exists`:

```
>(forall ?x ?y (iff (sibling-of ?x ?y)
                    (sibling-of ?y ?x)))

Sentence: (forall ?x:Person
            (forall ?y:Person
              (iff (sibling-of ?x:Person ?y:Person)
                   (sibling-of ?y:Person ?x:Person))))

>(forall ?x (exists ?y (< ?x ?y)))

Sentence: (forall ?x:Real
            (exists ?y:Real
              (< ?x:Real ?y:Real)))
```

Quantified sentences can be written in more conventional notation as follows:

```
>(forall ?x ?y . ?x sibling-of ?y <==> ?y sibling-of ?x)

Sentence: (forall ?x:Person
            (forall ?y:Person
              (iff (sibling-of ?x:Person ?y:Person)
                   (sibling-of ?y:Person ?x:Person))))
```

To avoid having to type too many question marks, we can simply define some names to denote variables:

```
define [x y z] := [?x ?y ?z]
```

We can then write:

```
>(forall x y . x sibling-of y <==> y sibling-of x)

Sentence: (forall ?x:Person
            (forall ?y:Person
              (iff (sibling-of ?x:Person ?y:Person)
                   (sibling-of ?y:Person ?x:Person))))

>(forall x . exists y . x < y)

Sentence: (forall ?x:Real
            (exists ?y:Real
              (< ?x:Real ?y:Real)))

>(forall x y z . x = y & y = z ==> x = z)

Sentence: (forall ?x:'T10987
            (forall ?y:'T10987
              (forall ?z:'T10987
                (if (and (= ?x:'T10987 ?y:'T10987)
                         (= ?y:'T10987 ?z:'T10987))
                    (= ?x:'T10987 ?z:'T10987)))))
```

1.2 Computing with Athena

Athena can be used as a general-purpose programming language. *Procedures* can be written to perform arbitrary computation with terms, which are the fundamental data type of the language, but also with values of other types, such as lists, functional dictionaries and hash tables. Procedures can be recursive, they can perform conditional branching, pattern matching on terms, sentences, lists, and values of other types, and so on. Generally speaking, a procedure f that takes n arguments is applied in prefix style: $(f\ e_1 \cdots e_n)$, where each e_i is an arbitrary Athena phrase. However, just as was the case with function symbols, binary procedures can be used in infix style, and successive applications of unary procedures need not be separated by parentheses. The associativity and precedence level of any procedure can be set by the user. The rest of this tutorial gives a whirlwind tour of some of Athena's computation facilities. There's quite a bit more that cannot be covered here, such as backtracking (via `try`), loops (`while`), cells, substitutions, dictionaries, and more; refer to XX for more details.

1.2.1 Input and Output

Output can be written to stdout using the predefined procedure `print`, which takes an arbitrary number of arguments (of arbitrary types), prints them out, and returns the *unit value*:[1]

```
>(print "\nHello world!\n")

Hello world!

Unit: ()
```

A string can be written to a text file with the procedure `write-file`, whose first argument is a

[1] The unit value, `()`, is a unique expression and value in Athena that is typically returned as the result of evaluating expressions with side effects, such as `cell` assignments, `while` loops, and I/O operations.

file path (represented as a string) and whose second argument is the string to be written, while the unary procedure `read-file` returns a string representing the contents of a text file:

```
>(write-file "foo.txt" "Foo bar")

Unit: ()

>(print (read-file "foo.txt"))
Foo bar
Unit: ()
```

Note that `write-file` writes in the given file in append mode. If you'd like to clobber the file's existing contents (if any) before writing to it, you can write your own procedure for it as follows:

```
>define (overwrite file str) :=
   let {_ := (delete-file file)}
     (write-file file str)

Procedure overwrite defined.
```

We'll talk more about user-defined procedures soon. Finally, the nullary procedure `read` receives as input from the user a single character at a time:

```
>(read) # Press the letter 'a'

Character: `a
```

1.2.2 Lists

Lists are a fundamental data type in Athena. A list is written simply by enclosing its elements inside square brackets, separated by white space:

```
define L := [1 2 3 4 5]

List L defined.
```

Primitive procedures for manipulating lists include `length` (self-explanatory); `head` (gives the first element of a list); `tail` (gives the tail of a list); `rev` (returns the reverse of the input list); `join`, which takes an arbitrary number of lists as arguments and concatenates them; and `add`, which takes an element h and a list t and returns the list obtained by adding h to the front of t:

```
>(length L)

Term: 5

>(head L)

Term: 1

>(tail L)

List: [2 3 4 5]

>(rev L)

List: [5 4 3 2 1]

>(join ['a 'b] L ['c 'd])
```

```
List: ['a 'b 1 2 3 4 5 'c 'd]

>(add 1 (add 2 []))

List: [1 2]
```

We can work with lists using pattern matching:

```
>match L {
   (list-of h t) => (print "First element: " h "and the tail: " t "\n")
 }

First element: 1 and the tail: [2 3 4 5]

Unit: ()
```

Here's how we might write our own list reversal procedure using pattern matching:

```
define reverse :=
  lambda (lst)
    match lst  {
      [] => []
    | (list-of h t) => (join (reverse t) [h])
    }

>(reverse [1 2 3])

List: [3 2 1]
```

Instead of using **lambda**, procedures can be defined in more conventional notation as follows:

```
define (reverse lst) :=
  match lst  {
    [] => []
  | (list-of h t) => (join (reverse t) [h])
  }
```

While this implementation works, its runtime complexity is quadratic in the size of the input list. We can attain linear-time performance by using an internal helper procedure, defined with **letrec**, since it is recursive:

```
define (reverse lst) :=
  letrec {loop := lambda (lst accum)
                    match lst {
                      [] => accum
                    | (list-of h t) => (loop t (add h accum))
                    }}
    (loop lst [])

>(reverse [1 2 3])

List: [3 2 1]
```

Athena strings are just lists of characters, so any procedure that operates on lists can be just as well applied to strings:

```
>(tail "foo bar")

List: [`o `o `\blank `b `a `r]

>(println (rev "foo bar"))
```

```
rab oof

Unit: ()
```

1.2.3 Computing with Terms and Sentences

A number of primitive procedures for computing with terms are built-in, such as numeric computation primitives: `plus`, `minus`, `times`, `div`, `mod`, `less?`, `leq?`, `greater?`, `geq?`, and so on. (As we saw, the identifiers `+`, `-`, `*`, `/`, and `%` are used as function symbols, not as procedures for numeric computation.)

```
>define (factorial n) :=
   check {
     (n less? 2) => 1
   | else => (n times (factorial (n minus 1)))
   }

Procedure factorial defined.

>(factorial 5)

Term: 120
```

We can use `let` expressions to name results of intermediate computations:

```
>let {p := (true & false);
      q := (false & true)}
  (p <==> q)

Sentence: (iff (and true false)
               (and false true))
```

1.2.4 Pattern Matching on Terms and Sentences

We can perform pattern matching on terms and sentences along the following lines:

```
define (verbalize t) :=
  match t {
    (father t') => (join "the father of " (verbalize t'))
  | (mother t') => (join "the mother of " (verbalize t'))
  | (t1 sibling-of t2) => (join (verbalize t1) " is a sibling of " (verbalize t2))
  | _ => (val->string t) # handle all remaining cases
  }

define t := (father joe sibling-of mother mary)

>(print "\n" (verbalize t) "\n")

the father of joe is a sibling of the mother of mary

Unit: ()
```

We can likewise use pattern matching on sentences. For instance, the following procedure carries out one step of the core transformation that is needed to convert a sentence to CNF (conjunctive normal form):

```
define (cnf-once p) :=
  match p {
    (~ (~ q)) => q
```

```
  | (~ (p1 & p2)) => (~ p1 | ~ p2)
  | (~ (p1 | p2)) => (~ p1 & ~ p2)
  | (~ (p1 | p2)) => (~ p1 & ~ p2)
  | (~ true) => false
  | (~ false) => true
  | (p1 | (p2 & p3)) => ((p1 | p2) & (p1 | p3))
  | ((p1 & p2) | p3) => ((p1 | p3) & (p2 | p3))
  | _ => p
  }

>(cnf-once ~ (true | false))

Sentence: (and (not true)
               (not false))
```

Quantified sentences can also be matched.

Pattern matching can be used at the top level to define multiple values simultaneously, and inside `let` expressions as well:

```
>define [a b] := [1 2]

Term b defined.

Term a defined.

>let {[a b] := [1 2]} (a plus b)

Term: 3
```

Athena has a very rich pattern language and these examples barely scratch the surface; refer to XX for more details and examples.

1.2.5 Precedence, Associativity, Input Expansion and Output Transformation

Every unary and binary function symbol has a certain *precedence*, which can be obtained by applying the procedure `get-precedence` to the symbol. Binary symbols also have a certain associativity, which can be left or right:

```
>(get-precedence +)

Term: 200

>(println (get-assoc *))

default (right)

Unit: ()
```

When we introduce a function symbol, we can optionally include some additional information at the end of the declaration, inside square brackets:

```
declare f: [Int Int] -> Int [130 left-assoc]
```

The above says that `f` has a precedence of 130 and is left-associative. We can also (optionally) specify *input expanders* inside the square brackets. This is a list of n unary procedures $g_1 \cdots g_n$, themselves enclosed within square brackets, where n is the arity of the symbol being declared. Then every time the symbol is applied to n values $v_1 \cdots v_n$, each v_i will first be run through g_i. For example:

```
define (word->numeral n) :=
```

```
  match n {
    "one" => 1 | "two" => 2 | "three" => 3 | "four" => 4 | _ => n
  }

declare h: [Int Boolean] -> Int [120 [word->numeral id]]

>(h "three" false)

Term: (h 3 false)
```

In Athena, `id` is the (predefined) identity procedure, `lambda (x) x`. So here we apply `word->numeral` to the first argument of `h`, while leaving the second argument alone.

Input expansion is a very useful notational device that allows us to change the way we write certain terms in entirely arbitrary ways. Likewise, we can transform the output produced by a certain user-defined procedure. For instance, if a procedure outputs natural numbers in Peano notation, we can make it convert these into conventional numeral notation. Here we simply convert `false` to `0` and `true` to `1`:

```
define (bool->number b) :=
  match b {
    false => 0
  | true => 1
  | _ => b
  }

define (even? x) := (x mod 2 equals? 0)

>(even? 500)

Term: true

>transform-output even? [bool->number]

OK.

>(even? 500)

Term: 1
```

Note that `equals?` is a predefined binary procedure that accepts two values and returns `true` if the values are identical and `false` otherwise.