Proving Theorems with Athena

David R. Musser Aytekin Vargun

August 28, 2003, revised January 26, 2005

Contents

1 Introduction

The Athena language and proof system [1, 2] developed at MIT by K. Arkoudas provides a way to present and work with mathematical proofs so that they are both human-readable and machine-checkable. In this document we work through several examples showing how one can express axioms and carry out proofs of theorems using a variety of proof techniques. In section 2 we work with a few laws of order relations such as irreflexivity and transitivity (the axioms of a strict partial-order relation, <) and prove that additional laws such as asymmetry are logical consequences of the axioms.

These proofs illustrate some of Athena's primitive deduction methods [2], such as modus ponens, universal specialization, and proof by contradiction. In Sections 3 and 4 we illustrate reasoning about inductive (recursive) data structures via examples involving natural numbers and list structures. We define these types inductively via Athena's `datatype` construct, and bring into play the powerful inference method of proof-by-induction via Athena's `by-induction` construct. Since the properties we are proving, such as the associative law for appending lists together, are written as equations, we make extensive use of *term rewriting* capabilities that we have introduced as (user-defined) inference methods. These rewriting methods, which are documented at the beginning of Section 3, should be useful in many other proofs that involve equations.

2 Proofs about order relations

In this section we illustrate some basic proof methods in Athena with formalization of some properties of an order relation.

We first load the rewriting file, which defines auxiliary inference methods for carrying out term rewriting inferences (described in Section 3 and used in Sections 3 and 4), plus a method called `conclude` that allows tracing of proof attempts as an aid to debugging (illustrated in this section).

```
(load-file "rewriting.ath")
```

Unless you have a copy of `rewriting.ath` in the directory in which you started Athena, you'll need to insert in the above directive the directory path where `rewriting.ath` resides; e.g., on CSLab machines you would write

```
(load-file "/cs/musser/public.html/gsd/athena/rewriting.ath")
```

We next declare the arity of an operator we call `is<` in terms of an arbitrary domain we call `D`:[1]

```
(domain D)

(declare is< (-> (D D) Boolean))
```

We now assert as axioms the irreflexive and transitive laws for `is<`:

[1] There is already a declaration of `<` on the predefined domain `Num`.

```
(define Order-Irreflexive
  (forall ?x
    (not (is< ?x ?x))))

(define Order-Transitive
  (forall ?x ?y ?z
          (if (and (is< ?x ?y) (is< ?y ?z))
              (is< ?x ?z))))

(assert Order-Irreflexive Order-Transitive)
```

These axioms characterize `is<` as a *strict partial order* relation. From these axioms we can prove another law, asymmetry:

```
(define Order-Asymmetry
  (forall ?x ?y
          (if (is< ?x ?y)
              (not (is< ?y ?x)))))
```

The proof is by contradiction, which is expressed in Athena with its primitive method `by-contradiction`:

```
(Order-Asymmetry
 BY (pick-any x y
              (assume (is< x y)
                (!by-contradiction
                 (assume (is< y x)
                   (dseq
                    ((is< x x)
                     BY (!mp (!uspec* Order-Transitive [x y x])
                             (!both (is< x y)
                                    (is< y x))))
                    ((not (is< x x))
                     BY (!uspec Order-Irreflexive x))
                    (!absurd (is< x x) (not (is< x x)))))))))
```

The result of executing this proof is

```
Theorem: (forall ?x:D
           (forall ?y:D
             (if (is< ?x ?y)
                 (not (is< ?y ?x)))))
```

The contradiction we get, expressed in the arguments to `absurd`, is `(is< x x)` and `(not (is< x x))`. The method `uspec*` used in proving the first

of these relations is an extension of the primitive universal specialization method, `uspec`:

uspec* P $[t_1 \ldots t_n]$

replaces the first n quantified variables in P with the terms $t_1, \ldots, t_n$.

Thus, we've shown that the asymmetry law is a theorem when one has a partial order relation; one does not have to assert it as a separate axiom.

In the above proof we used Athena's `BY` method, which has the syntax[2]

(⟨*proposition*⟩
  `BY` ⟨*deduction*⟩)

which sets up *proposition* as the one to be proved by the deduction that follows `BY`. That is, Athena checks that the result proved by the deduction is the same proposition as ⟨*proposition*⟩; if it is different, Athena reports an error. Using `BY` at intermediate steps within the proof helps document the subgoals into which the deduction divides the overall proof. If something is wrong at any stage of the deduction, however, it can be difficult to tell where the problem is. For this reason, we have devised a method called `conclude` that provides for tracing of the progress of the proof. The `conclude` method is called with the syntax

(`!(conclude` ⟨*proposition*⟩`)`
  ⟨*deduction*⟩`)`

which, like `BY`, sets up ⟨*proposition*⟩ as a proposition to be proved by the deduction that follows. With `conclude`, however, we can turn proof tracing on with[3]

```
(set! tracing true)
```

Now if we redo the above proof using `conclude`,

[2] This is Athena's only infix operator.

[3] Note that the exclamation point *follows* `set` rather than preceding it; this is not a method call but simply a call of a function named `set!`. The inclusion of an exclamation point in the name of an Athena function means that it is an imperative procedure executed for its effect rather than for computing value, a convention borrowed from the Scheme language.

```
(!(conclude Order-Asymmetry)
  (pick-any x y
            (assume (is< x y)
              (!by-contradiction
               (assume (is< y x)
                 (dseq
                  (!(conclude (is< x x))
                    (!mp (!uspec* Order-Transitive [x y x])
                           (!both (is< x y)
                                  (is< y x))))
                  (!(conclude (not (is< x x)))
                    (!uspec Order-Irreflexive x))
                  (!absurd (is< x x) (not (is< x x)))))))))
```

the output is

```
  Proving at level 1:
    (forall ?x:D
  (forall ?y:D
    (if (is< ?x ?y)
        (not (is< ?y ?x)))))
        Proving at level 2:
        (is< ?v318 ?v318)
        Done at level 2
        Proving at level 2:
        (not (is< ?v318 ?v318))
        Done at level 2
    Done at level 1

Theorem: (forall ?x:D
           (forall ?y:D
             (if (is< ?x ?y)
                 (not (is< ?y ?x)))))
```

With such tracing, if there is an error at any step of the proof, it is easier to see where the problem is. In all subsequent proofs we will use the `conclude` method. However, in showing output we will assume tracing is turned off:

```
(set! tracing false)
```

Now suppose we define a binary relation `E` as follows:

```
(declare E (-> (D D) Boolean))

(define E-Definition
  (forall ?x ?y
          (iff (E ?x ?y)
               (and (not (is< ?x ?y))
                    (not (is< ?y ?x))))))
```

The name `E` for this relation is motivated by fact that we can show that if `E` is assumed to be transitive then in combination with the partial order axioms for `is<` we can prove that `E` is in fact an equivalence relation; i.e., that it also obeys the other two axioms of an equivalence relation besides the transitive law, namely the reflexive and symmetric laws.

```
(define E-Transitive
  (forall ?x ?y ?z
          (if (and (E ?x ?y) (E ?y ?z))
              (E ?x ?z))))

(assert E-Definition E-Transitive)

(define E-Reflexive
  (forall ?x
    (E ?x ?x)))

(!(conclude E-Reflexive)
  (pick-any x
    (dseq
     (!(conclude (not (is< x x)))
       (!uspec Order-Irreflexive x))
     (!(conclude (E x x))
       (!mp (!right-iff (!uspec* E-Definition [x x]))
            (!both (not (is< x x))
                   (not (is< x x))))))))

(define E-Symmetric
  (forall ?x ?y
    (if (E ?x ?y)
        (E ?y ?x))))

(!(conclude E-Symmetric)
  (pick-any x y
            (assume (E x y)
              (dlet ((both-not
```

```
                        (and (not (is< x y))
                             (not (is< y x)))))
                  (dseq
                   (!(conclude both-not)
                     (!mp (!left-iff (!uspec* E-Definition [x y]))
                          (E x y)))
                   (!(conclude (E y x))
                     (!mp (!right-iff (!uspec* E-Definition [y x]))
                          (!both (!right-and both-not)
                                 (!left-and both-not)))))))))

(define E-Equivalent
  (and E-Reflexive
       (and E-Symmetric
            E-Transitive)))

(!(conclude E-Equivalent)
  (!both E-Reflexive
         (!both E-Symmetric
                E-Transitive)))
```

The output from this last deduction is

```
Theorem: (and (forall ?x:D
                (E ?x ?x))
              (and (forall ?x:D
                     (forall ?y:D
                       (if (E ?x ?y)
                           (E ?y ?x))))
                   (forall ?x:D
                     (forall ?y:D
                       (forall ?z:D
                         (if (and (E ?x ?y)
                                  (E ?y ?z))
                             (E ?x ?z)))))))
```

3 Proofs about natural numbers

Now we turn to some simple proofs by induction. We first declare a natural numbers type as an inductive datatype.

```
(datatype Nat zero (succ Nat))
```

We next define a `Plus` function, except that instead of actually defining it as a function, we specify its behavior axiomatically in terms of a `Plus` symbol.

The `Plus` function will take two `Nat` values as parameters and return their sum as a `Nat` value.

```
(declare Plus (-> (Nat Nat) Nat))
```

We next define the propositions we intend to use to define the meaning of `Plus` axiomatically.

```
(define Plus-zero-axiom
  (forall ?n (= (Plus ?n zero) ?n)))

(define Plus-succ-axiom
  (forall ?n ?m
          (= (Plus ?n (succ ?m))
             (succ (Plus ?n ?m)))))
```

After defining these propositions, we add them as axioms to Athena's assumption base. We can only use the axioms in the assumption base to prove new propositions.

```
(assert Plus-zero-axiom Plus-succ-axiom)
```

3.1 Term rewriting methods

Note that these axioms are stated using equations. When working with equations, one often structures a proof as sequence of *rewriting* steps, in which a term is shown to be equal to another term by means of a substitution that is justified by an universally quantified equation (or just a simple unquantified equation) in the assumption base. For example, we can rewrite the term

$$t = \texttt{(succ (Plus zero (succ (succ zero))))} \quad \text{(representing } (0+2)+1\text{)}$$

to

$$u = \texttt{(succ (succ (Plus zero (succ zero))))} \quad \text{(representing } (0+1)+2\text{)}$$

using our universally quantified equation

```
(define Plus-succ-axiom
  (forall ?n ?m
          (= (Plus ?n (succ ?m))
             (succ (Plus ?n ?m)))))
```

This can be done by substituting `zero` for `?n` and `(succ (succ zero))` for `?m`, producing the specialized equation

```
(= (Plus zero (succ (succ zero))) (succ (Plus zero (succ
                          zero))))
```

and then replacing the occurrence of the left hand side of this equation

```
(Plus zero (succ (succ zero)))
```

within t with the right hand side

```
(succ (Plus zero (succ zero)))
```

to produce u. We say that the equation is being used "left-to-right" or as a "left-to-right rewriting rule." (If the equation has no quantifiers, no substitution need be computed; the identity substitution is used.)

Athena does not currently have such rewriting methods built in, so we have created the following rewriting methods (implemented in the `rewriting.ath` file).

setup c t

initializes cell c to hold term t. Actually it internally holds the equation $(= t\ t)$, and the reduce and expand methods transform the right hand side of this equation. There are two predefined cells named left and right.

reduce c u E

attempts to transform the term t in cell c to be identical with the given term u by using equation E left-to-right.

expand c u E

attempts to transform the term u to the term t in cell c by using equation E left-to-right.

combine left right

attempts to combine the equation internally stored in cell left, say $(= t\ t')$, with the equation internally stored in cell right, say $(= u\ u')$, to deduce $(= t\ u)$ (which succeeds if t' and u' are identical).

As a simple example consider the following deduction in which we prove an equality by reducing both its left and its right hand side term to the same term, `(succ (succ (succ zero)))`.

```
(!(conclude (= (Plus zero (succ (succ (succ zero))))
               (succ (succ (succ (Plus zero zero))))))
 (dseq
  (!setup left   (Plus zero (succ (succ (succ zero)))))
  (!setup right  (succ (succ (succ (Plus zero zero)))))
  (!reduce left (succ (Plus zero (succ (succ zero))))
           Plus-succ-axiom)
  (!reduce left (succ (succ (Plus zero (succ zero))))
           Plus-succ-axiom)
  (!reduce left (succ (succ (succ (Plus zero zero))))
           Plus-succ-axiom)
  (!reduce left (succ (succ (succ zero ))) Plus-zero-axiom)
  (!reduce right (succ (succ (succ zero))) Plus-zero-axiom)
  (!combine left right)))
```

This produces

```
Theorem: (= (Plus zero
                   (succ (succ (succ zero))))
              (succ (succ (succ (Plus zero zero)))))
```

It's possible to trace the execution of rewriting methods with

```
(set! tracing-rewrites true)
```

Then reentering the above deduction produces

```
Rewriting
 (Plus zero
   (succ (succ (succ zero))))
 -->
 (succ (Plus zero
         (succ (succ zero))))
 -->
 (succ (succ (Plus zero
               (succ zero))))
 -->
 (succ (succ (succ (Plus zero zero))))
 -->
 (succ (succ (succ zero)))
```

```
    Rewriting
    (succ (succ (succ (Plus zero zero))))
    -->
    (succ (succ (succ zero)))

Theorem: (= (Plus zero
                  (succ (succ (succ zero))))
            (succ (succ (succ (Plus zero zero)))))
```

When we show output in subsequent examples we assume tracing of rewrites is turned off:

```
(set! tracing-rewrites false)
```

It should be noted that these rewriting methods, although not in the base Athena system, are guaranteed to be logically sound (relative to the soundness of Athena's logical system) because they are programmed in terms of Athena's primitive methods and all of the means that Athena provides for composing methods preserve soundness.

3.2 Proof by induction

The proof of the equational property above would be much simpler if had as a general property of `Plus`,

```
(define Plus-zero-property
  (forall ?n (= (Plus zero ?n) ?n)))
```

Whereas `Plus-zero-axiom` states that `zero` serves as a right-identity element for the `Plus` operator, this proposition states that it is also a left-identity element. The validity of this property is, however, *not* a consequence simply of the equational axioms we've stated about `Plus` and `zero`; it depends on the fact that `Nat` is inductively defined by the datatype declaration `(datatype Nat zero (succ Nat))`. According to the built-in semantics of `datatype`, the only values of type `Nat` are those that can be expressed with syntatically-correct and type-correct combinations of `zero` and `succ` terms:

```
zero
(succ zero)
(succ (succ zero))
(succ (succ (succ zero)))
...
```

For such inductively-defined types, we can prove propositions using the method of proof by induction.

3.3 Steps of a proof of an Nat inductive property.

There are two main steps in a proof by induction.

- The basis case. In our `Nat` datatype example, this is a special case in which the proposition to be proved is instantiated with `zero`.
- The induction step. In our `Nat` datatype example, we instantiate the proposition with (succ n) and attempt to prove it, assuming the proposition is true for n.

These separate proofs can be combined using Athena's `by-induction` construct to conclude the proposition is true for all n.

3.4 Basis case proof

Before doing the full proof, let's just check the basis case first.

```
(!(conclude (= (Plus zero zero) zero))
  (dseq
   (!setup left (Plus zero zero))
   (!setup right zero)
   (!reduce left zero Plus-zero-axiom)
   (!combine left right)))
```

In the proof itself, the `setup` calls set up `left` and `right` to hold the left and right hand sides of the equation to be proved. In this simple case only one application of `reduce`, applied to `left`, is sufficient to produce the same term, `zero`, as we have placed in `right`. Note that `reduce` has to specialize the quantified variable `?n` in Plus-zero-axiom to `zero` in order to use the result `(= (Plus zero zero) zero)` to substitute `zero` for `(Plus zero zero)`.

3.5 Induction step proof

We next proceed with the induction step of the proof:

```
(!(conclude (forall ?n (if (= (Plus zero ?n) ?n)
                           (= (Plus zero (succ ?n))
                              (succ ?n)))))
  (pick-any n
    (assume (= (Plus zero n) n)
      (dseq
       (!setup left (Plus zero (succ n)))
       (!setup right (succ n))
       (!reduce left (succ (Plus zero n)) Plus-succ-axiom)
       (!reduce left (succ n) (= (Plus zero n) n))
       (!combine left right)))))
```

Note that in the last call of `reduce` we used an unquantified equation to do the reduction; we could do that since the left hand side, `Plus zero n`, is an exact match for a subterm of `(succ (Plus zero n))`.

3.6 The full proof

After the basis case and inductive step proofs given above have been executed, the assumption base will contain the necessary ingredients for completion of the proof using Athena's `by-induction` deduction:

```
(by-induction
 Plus-zero-property
 (zero
  (!claim (= (Plus zero zero) zero)))
 ((succ n)
  (!mp (!uspec (forall ?n (if (= (Plus zero ?n) ?n)
                              (= (Plus zero (succ ?n))
                                 (succ ?n))))
               n)
       (= (Plus zero n) n))))
```

This produces

```
Theorem: (forall ?n:Nat
           (= (Plus zero ?n) ?n))
```

While this approach — successively doing the basis case, the induction step, and then applying `by-induction` — works, it seems a bit awkward. Why is it necessary to apply `uspec` and `mp` in the `succ` clause of the `by-induction` application? It's because `by-induction` actually sets up the induction hypothesis and puts it into the assumption base automatically, so one doesn't really to need to do it manually the way we did in proving the induction step as a separate proof. We can simply package the full proof into an application of `by-induction`, as follows:

```
(by-induction
 Plus-zero-property
 (zero
  (!(conclude (= (Plus zero zero) zero))
    (dseq
     (!setup left (Plus zero zero))
     (!setup right zero)
     (!reduce left zero Plus-zero-axiom)
     (!combine left right))))
 ((succ n)
  (!(conclude (= (Plus zero (succ n)) (succ n)))
    (dlet ((induction-hypothesis
            (= (Plus zero n) n)))
      (dseq
       (!setup left (Plus zero (succ n)))
       (!setup right (succ n))
       (!reduce left (succ (Plus zero n)) Plus-succ-axiom)
       (!reduce left (succ n) induction-hypothesis)
       (!combine left right))))))
```

Again, this produces:

```
Theorem: (forall ?n:Nat
           (= (Plus zero ?n) ?n))
```

In this proof, we defined and used the induction hypothesis, but we didn't have to assume it. Again, this is because `by-induction` automatically adds induction hypotheses to the assumption base according to the inductive structure of the type.

The possible drawback of packaging the full proof into a call of `by-induction` is that if there is an error somewhere, it can be harder to pinpoint it than if the basis case and inductive step proofs are done separately. However, we can avoid this problem by using `conclude` method calls as in the above proof (rather than `BY`) so that we can turn tracing on to show progress through the proofs of the individual subgoals.

In developing a proof within `by-induction`, we may want to test out the basis case proof before starting on the induction step proof. This is possible; we just have to include the inductive step clause with correct syntax. For example,

```
(by-induction
 Plus-zero-property
 (zero
  (!(conclude (= (Plus zero zero) zero))
    (dseq
     (!setup left (Plus zero zero))
     (!setup right zero)
     (!reduce left zero Plus-zero-axiom)
     (!combine left right))))
 ((succ n)
  (!claim true)))
```

parses and executes; it produces an error

```
Error, top level, 11.3: A Inductive subdeduction failed to establish
the right conclusion. The required conclusion was:
(= (Plus zero
         (succ ?v296))
   (succ ?v296))
but the derived one was:
true
(where the fresh variable ?v296 has replaced the pattern variable n).
```

but the error is just with the inductive step part, indicating that the basis case proof went through.

4 Proofs about lists

In this section we axiomatize list datatypes and prove a couple of useful properties based on the axioms using proof by induction. Although this example exhibits strong similarities to the inductive structure and proof we introduced in the previous section about natural numbers, there are some additional twists that reveal a few more of the capabilities of Athena.

4.1 Append Nil Property

Athena's type system allows polymorphism: we can declare polymorphic lists as a datatype. In fact, the following declaration is predefined:

```
(datatype (List-Of T) Nil (Cons T (List-Of T)))
```

Here `T` is a type variable that can be instantiated with any type, so that, for example,

- `(Cons zero (Cons (succ zero) Nil))` is a term of type `(List Nat)` (if `Nat` has been introduced as a datatype as in the previous section);
- `(Cons true Nil)` is a term of type `(List Boolean)`;
- however, `(Cons zero (Cons true Nil))` is an error ("Ill-sorted term") since all elements of a `List` must have the same type.

We define an `Append` function on `List`s, except that instead of actually defining it as a function, we specify its behavior axiomatically in terms of an `Append` symbol. `Append` takes two `List`s as parameters and returns a single list.

```
(declare Append ((T) -> ((List-Of T) (List-Of T)) (List-Of T)))
```

Note that Athena allows a list of type parameters to precede the function arrow; in this case there is only one, `T`.

The semantics we want for (Append p q) is it contains all of the elements of p followed by all of the elements of q. We next define the propositions we intend to use to define the meaning of Append axiomatically.

```
(define Append-Nil-axiom
  (forall ?q
    (= (Append Nil ?q) ?q)))

(define Append-Cons-axiom
  (forall ?x ?r ?q
          (= (Append (Cons ?x ?r) ?q)
             (Cons ?x (Append ?r ?q)))))))
```

After defining these propositions, we add them as axioms to Athena's assumption base.

```
(assert Append-Nil-axiom Append-Cons-axiom)
```

There are of course other properties of `Append` that could be stated and asserted as axioms. For example, the Append-Nil-axiom above says that the `Nil` list acts as a left-identity element for the `Append` operator. As in the case of `zero` in the `Nat` example in the previous section, it is natural to ask, is `Nil` also a right-identity element? I.e., is it also true that

```
(define Append-Nil-property
  (forall ?q (= (Append ?q Nil) ?q)))
```

In fact, we can prove that this proposition is a consequence of the two axioms for `Append`, using proof by induction. In Athena such proofs are supported by `by-induction`, which works just as well with the `List` datatype as we saw it did with `Nat` in the previous section.

4.2 Steps of a proof of a List inductive property

Again, there are two main parts of such a proof by induction.

- The basis case. In our `List` datatype example, this is a special case in which the proposition is instantiated with `Nil`.
- The induction step. In our `List` datatype example, we instantiate the proposition with (Cons x q) and attempt to prove it, assuming the proposition is true for q.

These proofs can be combined using Athena's `by-induction` construct to conclude the proposition is true for all q.

```
(by-induction
 Append-Nil-property
 (Nil
  (!(conclude (= (Append Nil Nil) Nil))
    (dseq
     (!setup left (Append Nil Nil))
     (!setup right Nil)
     (!reduce left Nil Append-Nil-axiom)
     (!combine left right))))
 ((Cons x p)
  (!(conclude (= (Append (Cons x p) Nil) (Cons x p)))
    (dlet ((induction-hypothesis
            (= (Append p Nil) p)))
      (dseq
       (!setup left  (Append (Cons x p) Nil))
       (!setup right (Cons x p))
       (!reduce left (Cons x (Append p Nil)) Append-Cons-axiom)
       (!reduce left (Cons x p) induction-hypothesis)
       (!combine left right))))))
```

4.3 Another proof about Append

The final example is the proof of another property of `Append`, again by induction.

```
(define Append-Associative
  (forall ?p ?q ?r
          (= (Append (Append ?p ?q) ?r)
             (Append ?p (Append ?q ?r)))))
```

The new twist here is that we have two additional quantified variables (`?q` and `?r`) besides the one (`?p`) on which we do the induction. These quantified variables must be dealt with in the proof. The Athena deduction that allows us to deal with them properly is `pick-any`.

```
(by-induction
 Append-Associative
 (Nil
  (!(conclude (forall ?q ?r (= (Append (Append Nil ?q) ?r)
                               (Append Nil (Append ?q ?r)))))
    (pick-any q r
              (dseq
               (!setup left (Append (Append Nil q) r))
               (!setup right (Append Nil (Append q r)))
               (!reduce left (Append q r) Append-Nil-axiom)
               (!reduce right (Append q r) Append-Nil-axiom)
               (!combine left right)))))
 ((Cons x p)
  (!(conclude (forall ?q ?r (= (Append (Append (Cons x p) ?q) ?r)
                               (Append (Cons x p) (Append ?q ?r)))))
    (dlet ((induction-hypothesis
            (forall ?q ?r (= (Append (Append p ?q) ?r)
                             (Append p (Append ?q ?r))))))
      (pick-any q r
                (dseq
                 (!setup left  (Append (Append (Cons x p) q) r))
                 (!setup right (Append (Cons x p) (Append q r)))
                 (!reduce left  (Append (Cons x (Append p q)) r)
                          Append-Cons-axiom)
                 (!reduce left  (Cons x (Append (Append p q) r))
                          Append-Cons-axiom)
                 (!reduce left  (Cons x (Append p (Append q r)))
                          induction-hypothesis)
                 (!reduce right (Cons x (Append p (Append q r)))
                          Append-Cons-axiom)
                 (!combine left right)))))))
```

References

[1] Konstantine Arkoudas. *Denotational Proof Languages*. PhD thesis, MIT, 2000. 1

[2] Konstantine Arkoudas. An athena tutorial, 2004. http://www.cag.csail.mit.edu/~kostas/dpls/athena. 1