Understanding Athena Proofs

David R. Musser

Rensselaer Polytechnic Institute, Troy, NY 12180
musser@cs.rpi.edu

Abstract. In the Athena interactive theorem proving environment, proofs are usually written in terms of a few high-level inference methods: equality and implication chaining, induction, case analysis, contradiction, and abstraction and specialization. This document briefly describes how these methods are expressed in Athena and illustrates their application in terms of a few proofs from the Athena library.[1]

1 Introduction

Athena [1] is an interactive and programmable theorem proving environment, with separate but intertwined languages for computation and deduction. For conventional programming, the built-in computational domains include not only the usual ones of most languages—booleans, numbers, and strings—but also those typical of symbolic computation such as lists, terms and sentences of (first-order, multi-sorted) logic, substitutions, etc. The principal mechanism for program composition is the procedure call. Procedures are higher-order, i.e., they may take procedures as arguments and return procedures as results.

The principal tool for constructing proofs is the *method* call. A method call represents an inference step, and can be primitive or complex (derived). Like procedures, methods can accept arguments of arbitrary types, including other methods and/or procedures, and thus are also higher-order. Evaluation of a procedure call, if it does not raise an error or diverge, can result in a value of any type, but evaluation of a method call—again, if it does not raise an error or diverge—can result only in a *theorem*: a sentence of logic that is derived by inference from axioms and other theorems.

While there are generally many ways to express a proof, an Athena method (or a stand-alone, "straight-line" program in the proof language) is one such expression, and a key attribute of the Athena proof language is that such expressions of proofs are not only machine-checkable but also human-readable. The readability of Athena proofs rests mainly on the naturalness with which one can express important proof methods. In part, this is due to a fundamental mechanism of Athena: its *assumption base*. When a sentence is assumed or proved, it is entered into the assumption base, which is a set of sentences that each of Athena's primitive inference forms and methods interacts with, checking one or more of its inputs to see if they are present in the set and/or making new entries.

Assumption bases play a fundamental role in Athena proofs. To anyone acquainted with the basics of how conventional programs are interpreted or compiled, manipulation of the assumption

[1] Pending the completion of a comprehensive tutorial by Konstantine Arkoudas, Athena's creator and primary developer, this document is intended as a basic guide to understanding existing Athena proofs, but lacks many details needed for becoming proficient in writing new proofs.

base during proofs will seem familiar, because it is similar to the way that the *run-time stack* is used to hold results of intermediate computations. As with the stack, sentences may be added to the assumption base temporarily and later removed. An important case is the construct

assume A
  D

during whose evaluation A is added to the assumption base and remains present while evaluating deduction D. If D succeeds in proving a theorem C then the new sentence added to the assumption base is $(A \Rightarrow C)$, and A is removed.[2] (A is also removed if D raises an error condition.) This is logically sound because A was added to the assumption base only for the purpose of proving $(A \Rightarrow C)$, and validity of that sentence itself does not depend on A.

Consider also `mp`, which is Athena's version of the *modus ponens* inference rule: `(!mp` P Q`)` checks that both P and Q are in the assumption base, and that P is an implication, $(Q \Rightarrow R)$, with Q as its antecedent. If these conditions are satisfied, then the consequent, R, of the implication is established as a theorem and entered into the assumption base. If any of the conditions fails, an error is reported.

The `mp` method is one of Athena's built-in inference rules that form the foundation of its reasoning capability, but in most cases users do not need to invoke it directly. Instead, they will invoke higher-level methods: *equality and implication chaining, induction, case analysis*, and *proof by contradiction*. In the next section we give a brief description of how Athena supports each of these methods. Section 4 describes Athena facilities for *abstraction and specialization*.

2 Higher-level Athena proof methods

The best way to understand the Athena proof methods described in this section is to see applications of them in the context of a meaningful application. In simple cases, a proof can be done with only one of these methods, but most proofs require combinations of them, such as the use of equality or implication chaining in each of the basis and inductive steps of an application of induction. For this reason, in describing each method we will refer the reader to illustrative segments of an actual file from the Athena library [2], `nat-minus.ath`, whose listing is given in Section 3.

2.1 Equality and implication chaining

One of the most ubiquitous proof methods is proving equations by chaining together a sequence of terms connected by equalities. In Athena, we can express such proofs with the `chain` method:

`(!chain [`t_0 `=` t_1 `[`J_1`]` `=` t_2 `[`J_2`]` `=` $\cdots$ `=` t_n `[`J_n`]])`

proves the equation $t_0 = t_n$, where each J_i is a *justification* for the preceding equality $t_{i-1} = t_i$. Each justification J_i must be a sentence—in this case a previously assumed or proved universally

[2] For simplicity, we speak here of sentences being "added" or "removed" as if assumption bases were stateful objects subject to destructive modification. In reality, assumption bases are *functional*. Thus, for instance, the body of the `assume` deduction above is evaluated in a *new* assumption base, obtained by augmenting the outer assumption base with the hypothesis A.

quantified equality or conditional equality—or a method capable of proving $t_{i-1} = t_i$.[3] Thus, each justification that is a sentence must already be in the assumption base, and each one that is a method must work with assumption base entries to prove the equality step.

The first proof in file `nat-minus.ath` contains a simple application of `chain` to prove an equation `((x - zero) + zero) = x`, in lines 23-25. Another example is in lines 40-44. In these examples, `sym` is a primitive method for applying the symmetric axiom of equality: `(!sym (t = u))` proves `(u = t)` provided `(t = u)` is in the assumption base.

One can also express *implication chains* with the `chain` method. To prove the implication $S_0 \Rightarrow S_n$, one can write a `chain` call

$$\texttt{(!chain [}S_0 \Rightarrow S_1 \texttt{ [}J_1\texttt{]} \Rightarrow S_2 \texttt{ [}J_2\texttt{]} \Rightarrow \cdots \Rightarrow S_n \texttt{ [}J_n\texttt{]])}$$

where the S_i are sentences and the justification J_i proves $(S_{i-1} \Rightarrow S_i)$.

If S_0 has already been proved, the variant

$$\texttt{(!chain-last [}S_0 \Rightarrow S_1 \texttt{ [}J_1\texttt{]} \Rightarrow S_2 \texttt{ [}J_2\texttt{]} \Rightarrow \cdots \Rightarrow S_n \texttt{ [}J_n\texttt{]])}$$

proves S_n.

Lines 105-108, in the proof of `alt-<=-characterization`, contain an implication chain expressed with `chain`. Lines 124-132 contain a couple of examples using `chain-last` to prove the final sentence in the implication chain, among others in the file.

2.2 Induction

For natural numbers, ordinary mathematical induction takes the form of dividing a proof of $(\forall n.\ (P\ n))$ into two cases: (i) $(P\ 0)$ and (ii) $(\forall n.\ (P\ n) \Longrightarrow (P\ n+1))$. Case (i) is called the *Basis Case*, case (ii) is called the *Induction Step*, and within it $(P\ n)$ is called the *Induction Hypothesis*. Proof of the Basis Case and the Induction Step suffices, basically because every natural number is either 0 or can be constructed from 0 by a finite number of increments by 1. This property of natural numbers can be stated in Athena by defining natural numbers as the following datatype:

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

The symbols `zero` and `S` ("successor") are called the *constructors* of `N`. Given this declaration, the only ground terms allowed by Athena's type system for `N` values are `zero`, `(S zero)`, `(S (S zero))`, etc. Furthermore, from this ***datatype*** declaration, Athena derives the usual natural number induction principle and makes it available via its ***by-induction*** form. Schematically,

```
by-induction (forall ?n . P ?n) {
  zero =>
    conclude (P zero)
      ...
| (S n) =>
```

[3] Actually, the justification can be a list of such sentences or methods which together justify the equality. Also, in place of a sentence or method one can write an arbitrary Athena expression that evaluates to it.

```
    let {ind-hyp := (P n)}
      conclude (P (S n))
        ...
}
```

Here we have given a name `ind-hyp` to the induction hypothesis `(P n)`, but whether we name it or not, it is available in the induction-step case (i.e., ***by-induction*** places it in the assumption base) for use in the proof of `(P (S n))`.

The proof of `second-equal` in lines 50-57 is a simple application of ***by-induction***. The first proof in the file is a more complex example, and contains within it an application of ***datatype-cases***, an inference form that is the same as ***by-induction*** except that the induction hypothesis is not placed in the assumption base. It is thus useful, as in this example, when one merely needs to break down a proof by cases according to the datatype's constructors.

More generally, Athena can extend ***by-induction*** to provide an induction principle based on the form of constructors as given in any ***datatype*** declaration. An example only slightly more complex than `N` is a *list* type defined by

```
1 datatype (List T) := nil | (:: T (List T))
```

This defines a *polymorphic* datatype: `T` is a sort parameter, and `(List T)` is the sort of homogeneous lists of sort `T` elements; e.g., `(List Boolean)`, `(List N)`, `(List (List N))`, etc., are separate sorts that share the `nil` and `::` constructors. A use of the corresponding induction principle might take the form:

```
by-induction (forall ?L ?x . (Q ?L ?x) {
  nil =>
    conclude (forall ?x . (Q nil ?x))
      ...
| (y :: L) =>
  let {ind-hyp := (forall ?x . (Q L ?x))}
    conclude (forall ?x . (Q (y :: L) ?x))
      ...
}
```

Note that the induction cases retain the universal quantification of variables other than the induction variable, which can be seen also in many of the proofs in `nat-minus.ath`.

2.3 Case analysis

An induction proof (or a datatype-cases proof) is one important kind of case analysis, breaking a proof into one or more basis cases (corresponding to ***datatype*** constructors that take no arguments of the type) and one or more induction step cases (corresponding to constructors that take at least one argument of the type). More generally, case analysis can be applied independently of a datatype, when we have the same proof goal under each of several different assumptions whose disjunction is known to cover all possibilities. For example, assuming the disjunction

$(P_1 \mid P_2 \mid \ldots \mid P_n)$

is in the assumption base, if we write

```
(!cases (P1 | P2 | ... | Pn)
  assume P1
    D1
  assume P2
    D2
  ...
  assume Pn
    Dn)
```

and fill in each of the D_i deductions to conclude the same goal sentence Q, then Q is the result of the entire `cases` deduction.

For an example, see lines 93-97 in the proof of `second-greater-than-or-equal`.

An important special case is just two disjuncts with one the negation of the other. We can write

```
(!two-cases
  assume A
    D1
  assume (~ A)
    D2)
```

since it is equivalent to

```
(!cases (!ex-middle A)
  assume A
    D1
  assume (~ A)
    D2)
```

where `(!ex-middle` A`)` deduces `(`A `|` `~` A`)` for any sentence A. (Athena's logic is classical.) There is no example of `two-cases` in `nat-minus.ath`, but there are many in `nat-less.ath`.

2.4 Proof by contradiction

In Athena, the main form of a proof by contradiction is

```
(!by-contradiction P
  assume (~ P)
    D)
```

where D deduces `false`. Commonly, D obtains `false` with a call of the `absurd` method, of the form

```
(!absurd Q R)
```

where Q and R have been deduced from `(~` P`)` and other members of the assumption base and R is the negation of Q.

Lines 121-132, in the proof of `<-left`, are an application of `by-contradiction`.

Sometimes one does not need the negation of P to obtain the contradiction. Then one can deduce P more simply with the `from-complements` method:

```
(!from-complements P Q R)
```

where R is the negation of Q. See the proof of `Plus-Cancel`, lines 33-36, for an example.

3 Axioms and theorems for a natural number subtraction function

The following is a listing of the `nat-minus.ath` file in the Athena library, giving axioms and theorems about a subtraction operation on natural numbers. It is not self-contained, as it uses axioms and theorems about inequality operations `<` and `<=` from the file `nat-less.ath`, which in turn depends on axioms and theorems about the natural number `+` function from file `nat-plus.ath`. In most cases, the form of the external axioms and theorems can be inferred from their name and how they are used in a proof. In any case, one can easily look them up in the library files or retrieve them in an Athena session. For example in the first proof in the listing, the first justification `zero-right` in line 25 is one of the asserted axioms (line 15), but `right-zero` is not defined in this file. To see its value, we can search for it in `nat-less.ath` and `nat-plus.ath` (which is where, in fact, it is defined), or we can load `nat-less.ath` into an Athena session, and enter it at the prompt:

```
1 >right-zero
2
3 input prompt:1:1: Error: Could not find a value for right-zero.
```

So that doesn't do it, but notice that `right-zero` is used in the context of `extend-module N {...}`, so we should try

```
1 >N.right-zero
2
3 Sentence: (forall ?n:N
4               (= (N.Plus ?n:N zero)
5                  ?n:N))
```

Note that Athena's output is currently always in prefix mode, which is the internal form stored even when infix notation is used for input.[4]

```
1  # Subtraction of natural numbers.
2
3  load "nat-less.ath"
4
5  extend-module N {
6
7   declare Minus: [N N] -> N
8   (overload (- Minus))
9   (set-precedence Minus 200)
10
11  module Minus {
12
13    assert zero-left := (forall ?x . zero - ?x = zero)
14    assert zero-right := (forall ?x . ?x - zero = ?x)
15    assert injective :=  (forall ?x ?y . (S ?x) - (S ?y) = ?x - ?y)
16
```

[4] Any binary function symbol, sentential connective, or procedure can be input in infix mode. Note also that function symbols (and program identifiers) can include special characters, so that `Less=` and `<-left`, for example, are legal identifiers.

```
define Plus-Cancel := (forall ?y ?x . ?y <= ?x ==> ?x = (?x - ?y) + ?y)

by-induction Plus-Cancel {
  zero =>
  pick-any x:N
    assume (zero <= x)
      (!sym (!chain [((x - zero) + zero)
                     = (x + zero)           [zero-right]
                     = x                    [right-zero]]))
| (S y) =>
  let {ind-hyp := (forall ?x . y <= ?x ==> ?x = (?x - y) + y)}
    datatype-cases
       (forall ?x . (S y) <= ?x ==> ?x = (?x - (S y)) + (S y))
    {
      zero =>
        assume A := ((S y) <= zero)
          (!from-complements
            (zero = (zero - (S y)) + (S y))
            A
            (!chain-last [true ==> (~ A)  [Less=.not-S-zero]]))
    | (S x) =>
        assume A := ((S y) <= (S x))
          let {C := (!chain-last [A ==> (y <= x) [Less=.injective]])}
            (!sym (!chain
                   [(((S x) - (S y)) + (S y))
                    = ((x - y) + (S y))       [injective]
                    = (S ((x - y) + y))       [right-nonzero]
                    = (S x)                   [C ind-hyp]]))
    }
}

define second-equal := (forall ?x . ?x - ?x = zero)

by-induction second-equal {
  zero =>
  (!chain [(zero - zero) = zero   [zero-left]])
| (S x) =>
  let {ind-hyp := (x - x = zero)}
    (!chain [((S x) - (S x)) = (x - x)   [injective]
                             = zero      [ind-hyp]])
}

define second-greater :=
  (forall ?x ?y . ?x < ?y ==> ?x - ?y = zero)

by-induction second-greater {
  zero =>
  conclude (forall ?y . zero < ?y ==> zero - ?y = zero)
    pick-any y
      assume (zero < y)
```

```
          (!chain [(zero - y) = zero [zero-left]])
  | (S x) =>
    let {ind-hyp := (forall ?y . x < ?y ==> x - ?y = zero)}
    datatype-cases (forall ?y . (S x) < ?y ==> (S x) - ?y = zero)
    {
      zero =>
      assume A := ((S x) < zero)
        (!from-complements ((S x) - zero = zero)
         A
         (!chain-last [true ==> (~ A) [Less.S-not-<-zero]]))
    | (S y) =>
      assume A := ((S x) < (S y))
        let {C := (!chain-last [A ==> (x < y)  [Less.injective]])}
        (!chain [((S x) - (S y))
                 = (x - y)        [injective]
                 = zero           [C ind-hyp]])
    }
  }

  define second-greater-or-equal :=
    (forall ?x ?y . ?x <= ?y ==> ?x - ?y = zero)

  conclude second-greater-or-equal
    pick-any x:N y:N
      assume A := (x <= y)
        let {C := (!chain-last [A ==> (x < y | x = y) [Less=.definition]])}
        (!cases C
         (!chain [(x < y) ==> (x - y = zero)  [second-greater]])
         assume (x = y)
          (!chain [(x - y) = (x - x)   [(x = y)]
                           = zero      [second-equal]]))

  define alt-<=-characterization :=
    (forall ?x ?y . ?x <= ?y <==> (exists ?z . ?y = ?x + ?z))

  conclude alt-<=-characterization
    pick-any x y
      (!equiv
       (!chain [(x <= y)
                ==> (y = (y - x) + x)   [Plus-Cancel]
                ==> (y = x + (y - x))   [commutative]
                ==> (exists ?z . y = x + ?z) [existence]])
       assume A := (exists ?z . y = x + ?z)
         pick-witness z for A witnessed
           (!chain-last
            [witnessed ==> (x <= y)   [Less=.k-Less=]]))

  define <-left :=
    (forall ?x ?y . zero < ?y  & ?y <= ?x ==> ?x - ?y < ?x)

```

```
  conclude <-left
    pick-any x y
      assume A := (zero < y & y <= x)
        let {goal := ((x - y) < x)}
        (!by-contradiction goal
         assume (~ goal)
          (!absurd
           (!chain-last [(zero < y)
                         ==> (zero + x < y + x)   [Less.Plus-k]
                         ==> (x < y + x)          [left-zero]])
           (!chain-last [(~ goal)
                         ==> (x <= x - y)           [Less=.trichotomy1]
                         ==> (x + y <= (x - y) + y) [Less=.Plus-k]
                         ==> (x + y <= x) [(y <= x) Plus-Cancel]
                         ==> (~ x < x + y) [Less=.trichotomy4]
                         ==> (~ x < y + x) [commutative]]))))

  define Plus-Minus-property :=
    (forall ?x ?y ?z . ?x = ?y + ?z ==> ?x - ?y = ?z)

  conclude Plus-Minus-property
    pick-any x y z
      assume A := (x = y + z)
        let {C1 :=
             (!chain-last
              [A ==> (y <= x)                          [Less=.k-Less=]
                 ==> (x = (x - y) + y)                 [Plus-Cancel]]);
             C2 := (!chain-last [A ==> (x = z + y)     [commutative]])}
        (!chain-last
         [((x - y) + y) = x         [C1]
                        = (z + y)   [C2]
                        ==> ((x - y) = z)  [=-cancellation]])

  define cancellation :=  (forall ?x ?y . (?x + ?y) - ?x = ?y)

  conclude cancellation
    pick-any x y
      (!chain-last
       [(x + y = x + y)  ==> ((x + y) - x = y) [Plus-Minus-property]])

 } # N.Minus
} # N
```

4 Abstraction and specialization

In Athena, one can introduce axioms and theorems at an abstract level via *structured theories*, as explained below.[5] Proofs are encapsulated in parametrized methods that can be used for proving

[5] Parts of this section are taken from [4].

theorems that are different specializations of an abstract theorem via different mappings of function symbols. Algebraic theories that have been developed in the Athena library as structured theories include *semigroup*, *monoid*, *group*, *ring*, *integral domain*, etc. See [2, group.ath]. Other theories collected in the Athena library include familiar relational theories: *binary-relation, reflexive, symmetric, transitive, preorder, strict weak order, total order, transitive closure*, etc. See Figure 1 for development of a few of these relational theories by successive *structured theory refinements*.

In these definitions we use the `theory` procedure, whose current form is based on one first presented in [5]. Specifically,

(`theory` *superiors axioms name*)

defines a structured theory with name *name*, (new) *axioms*, and *superiors*—theories of which the new theory is a direct refinement. The set of axioms of the theory so defined is the union of new axioms and, recursively, the sets of axioms of the superiors. In `Irreflexive` there is a single axiom, also named `Irreflexive`, and one superior, `Binary-Relation.Theory`, which has no axioms. `Strict-Partial-Order` refines `Irreflexive.Theory` and `Transitive.Theory`, and introduces no additional axioms.

The definition of `Transitive-Closure.Theory` refines an *adapted* theory,

`[Strict-Partial-Order.Theory 'TC [R R+]]`,

which has the same axioms as `Strict-Partial-Order` but with `R` renamed as `R+`. These adapted axioms can be referred to by compound names of the form `['TC` *s*`]`, where *s* is a `Strict-Partial-Order.Theory` sentence. For example, from the `Transitive` axiom in `Strict-Partial-Order.Theory`,

```
(forall ?x ?y ?z . ?x R ?y & ?y R ?z ==> ?x R ?z),
```

we obtain `['TC Transitive]` as the name of the axiom

```
(forall ?x ?y ?z . ?x R+ ?y & ?y R+ ?z ==> ?x R+ ?z).
```

With `Transitive-Closure.Theory`'s other superior theory, `Irreflexive. Theory`, there is no renaming, so its axiom with the original `R` symbol is included. Thus `Transitive-Closure.Theory` is defining axioms that relate the three operators `R`, `R+` and `R*`.

Although we regard the sentences listed in a theory as axioms, we do not assert them into the assumption base. Instead, if we have *proved* that a homomorphic image of each of these sentences is a theorem, then we will be able to use proof methods associated with the theory to prove new theorems, rather than having to write their proofs for every structure that models the theory.

4.1 Proofs at an abstract level

To illustrate the way that abstract-level proofs can be encapsulated in a parametrized method, let us state a couple of Transitive-Closure theorems, define a proof method that can prove any adapted version of the theorems, and show how the theorem statement and proof method can be incorporated into the Transitive-Closure theory.

```
1 extend-module Transitive-Closure {
2 define RR+-inclusion :=  (forall ?x ?y . ?x R ?y ==> ?x R+ ?y)
```

```
module Binary-Relation {
 declare R: (T) [T T] -> Boolean
 define Theory := (theory [] [] 'Binary-Relation)}
module Irreflexive {
 open Binary-Relation
 define Irreflexive := (forall ?x . ~ ?x R ?x)
 define Theory := (theory [Binary-Relation.Theory]
                          [Irreflexive] 'Irreflexive)}
module Transitive {
 open Binary-Relation
 define Transitive := (forall ?x ?y ?z .
                        ?x R ?y & ?y R ?z ==> ?x R ?z)
 define Theory := (theory [Binary-Relation.Theory]
                          [Transitive] 'Transitive)}
module Strict-Partial-Order {
 open Irreflexive
 open Transitive
 define Theory :=
   (theory [Irreflexive.Theory Transitive.Theory]
           [] 'Strict-Partial-Order)}
module Transitive-Closure {
 open Irreflexive
 open Strict-Partial-Order
 declare R+, R*: (S) [S S] -> Boolean
 declare R**: (S) [N S S] -> Boolean
 define R**-zero :=
  (forall ?x ?y . (R** zero ?x ?y) <==> ?x = ?y)
 define R**-nonzero :=
  (forall ?x ?n ?y .
   (R** (S ?n) ?x ?y) <==>
   (exists ?z . (R** ?n ?x ?z) & ?z R ?y))
 define R+-definition :=
  (forall ?x ?y . ?x R+ ?y <==>
  (exists ?n . (R** (S ?n) ?x ?y)))
 define R*-definition :=
  (forall ?x ?y . ?x R* ?y <==> (exists ?n . (R** ?n ?x ?y)))
 define Theory :=
  (theory [Irreflexive.Theory
           [Strict-Partial-Order.Theory 'TC [R R+]]]
          [R**-zero R**-nonzero R+-definition R*-definition]
    'Transitive-Closure)}
```

Fig. 1. Structured theory definitions. Omitted from this figure are the theorems and proofs included in the `Strict-Partial-Order` and `Transitive-Closure` modules; see [2, order.ath and transitive-closure.ath].

```
define TC-Transitivity1 :=
  (forall ?x ?y ?z . ?x R+ ?y & ?y R ?z ==> ?x R+ ?z)
define theorems := [RR+-inclusion TC-Transitivity1]}
```

```
extend-module Transitive-Closure {
define proofs :=
  method (theorem adapt)
    let {given := lambda (P) (get-property P adapt Theory);
         lemma := method (P) (!property P adapt Theory);
         chain := method (L) (!chain-help given L 'none);
         chain-last := method (L) (!chain-help given L 'last);
         [R R+ R*] := (adapt [R R+ R*])}
    match theorem {
      (val-of RR+-inclusion) =>
      pick-any x y
        (!chain
         [(x R y)
          ==> (x R y | (exists ?z . x R+ ?z & ?y' R y))
                                      [alternate]
          ==> (x R+ y)                [R+-definition]])
    | (val-of TC-Transitivity1) =>
      pick-any x y z
        assume A := (x R+ y & y R z)
          (!chain-last
           [A ==> (exists ?y . x R+ ?y & ?y R z) [existence]
              ==> (x R z | (exists ?y . x R+ ?y & ?y R z))
                                                  [alternate]
              ==> (x R+ z)                    [R+-definition]])
    }
(evolve Theory [theorems proofs])}
```

The `evolve` procedure call extends the list of sentences associated with Transitive-Closure theory to include these theorems and also records the proof method containing the corresponding proofs. Having done so, we can at any time retrieve an instance one of the Transitive-Closure axioms or theorems with Athena's `get-property` procedure. In general, (`get-property` *P adapt theory*) searches the theory refinement hierarchy for *P*, using *theory* as the starting point for the search. The top-level axioms of *theory* are searched, followed by a recursive search of each of its superiors. If `get-property` finds *P* it applies the *adapt* mapping to it and returns the result. It is an error if the search ends without finding *P*, or if applying *adapt* results in an ill-typed sentence.

While `get-property` is just a procedure, there is a corresponding method—`property`, which takes the same arguments and conducts the same search and adaptation—that tries to prove the resulting sentence using the associated proof method.

The proof method introduces local, adapted versions of methods `lemma`, `chain` and `chain-last`, and a procedure, `given`, which are handy tools for simplifying the way that external properties can be cited in proof steps that need them. The `lemma` method simply calls the `property` method, passing it `P` (`lemma`'s argument), `adapt` (one of the enclosing proof method's arguments), and the theory to be used as the starting point for searches for `P`. Thus it finds the cited property in

the theory structure and proves the instance produced by applying `adapt` to it, using the proof method that accompanies `P`.

The `given` procedure is similar but only retrieves the cited property; it is used in an abstract-level proof when the property is known either to be an axiom of the theory or to have already been proved using `lemma` (or `property` directly). Thus, in either case, the property is already in the assumption base.

Lastly, the local definition of `chain` (and, if necessary, `chain-last` and `chain-first`) is defined in terms of the predefined method `chain-help`, which tries to apply `given` to sentences in chain justifications (if that fails, it uses the sentence itself). This is what allows one to write chain-step justifications like `[R+-definition]` instead of the more verbose `[(given R+-definition)]` or `[(get-property R+-definition adapt Transitive-Closure.Theory)]`.

5 Conclusion

In this guide to understanding Athena proofs, we've sampled several of the most important proof methods and illustrated them with excerpts from the Athena library [2]. Many other fairly basic examples are available in the library, and more advanced examples can be found in a development of theories concerning the actor model of distributed computation [3, 4].

References

1. K. Arkoudas. Athena. `http://proofcentral.org/athena`.
2. K. Arkoudas and D. Musser. Athena libraries. `http://proofcentral.org/athena/lib`.
3. D. Musser and C. Varela. Actor model Athena files. `http://proofcentral.org/athena/examples/actor-model`.
4. D. Musser and C. Varela. Structured reasoning about actor systems, 2012. Submitted for publication.
5. Aytekin Vargun. *Code-Carrying Theory.* PhD thesis, Rensselaer Polytechnic Institute, 2006. `http://www.cs.rpi.edu/~vargua/thesis/cct-thesis.pdf`.