CSCI.6962/4962 Software Verification— Fundamental Proof Methods in Computer Science (Arkoudas and Musser)—Sections 8.2-8.7

Instructor: Carlos Varela
Rensselaer Polytechnic Institute
Spring 2018

Natural Number Orderings

Goal: to reason about properties of natural number ordering relations, and its application to ordered lists and binary search trees.

- *properties of natural number ordering functions*
  - *trichotomy properties*
  - *transitive and asymmetric properties*
  - *less-equal properties*
  - *combining ordering and arithmetic*
- natural number subtraction
- ordered lists
- binary search trees

Natural number subtraction

- Customary subtraction is not closed on the natural numbers: when the second argument is larger than the first, the result is not a natural number but a negative integer.
  - For instance, subtracting 5 from 3 yields -2.
- Here we continue to deal only with natural numbers and stipulate that when the second argument is larger than the first, the result is 0.
- For all other arguments n and m, the result will be, as expected, the difference $n - m$.
- The function just described is sometimes called the *monus* operation.

Natural number subtraction

The definition can be given by three axioms:

- the first two handling the cases when one of the two arguments is 0, and
- the third for the case when both arguments are greater than zero.

```
declare -: [N N] -> N [200 [int->nat int->nat]]

module Minus {

  assert* axioms := [(zero - x = zero)
                     (x - zero = x)
                     (S x - S y = x - y)]

  define [zero-left zero-right both-nonzero] := axioms
```

Natural number subtraction

An important property of subtraction is the following form of *cancellation*:

- When the second argument y is no larger than the first, x, then

$$x = (x - y) + y.$$

- That is, if we first subtract y from x and then add y back to the result, we get x again.

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

Natural number subtraction

We prove this cancellation property by induction, using `force` for the inductive step:

```
by-induction Plus-Cancel {
  zero =>
    conclude (forall x . zero <= x ==> x = (x - zero) + zero)
      pick-any x
        assume (zero <= x)
          (!sym (!chain [((x - zero) + zero)
                         = (x + zero)                [zero-right]
                         = x                         [Plus.right-zero]]))
| (y as (S y')) =>
   let {ind-hyp := (forall x . y' <= x ==> x = (x - y') + y')}
     (!force (forall x . S y' <= x ==> x = (x - S y') + S y'))
}
```

Natural number subtraction

Now, we can prove the inductive step to replace the use of `force`:

```
let {IH := (forall x . y' <= x ==> x = (x - y') + y')}
  datatype-cases (forall x . S y' <= x ==> x = (x - S y') + S y') {
    zero => conclude (S y' <= zero ==> zero = (zero - S y') + S y')
              assume hyp := (S y' <= zero)
                (!from-complements (zero = (zero - S y') + S y')
                  hyp (!uspec Less=.not-S-zero y')) # ~ hyp
  | (S x') =>
     conclude (S y' <= S x' ==> S x' = (S x' - S y') + S y')
       assume hyp := (S y' <= S x')
         let {C := (!chain-> [hyp ==> (y' <= x')   [Less=.injective]])}
           (!sym
             (!chain-> [((S x' - S y') + S y')
                        = ((x' - y') + S y')          [both-nonzero]
                        = (S ((x' - y') + y'))        [Plus.right-nonzero]
                        = (S x')                      [C IH]]))
  }
```

Case Analysis Based on Conditional Definitions

Generalizing, we arrive at the following useful principle:

> When the proof goal involves a function f that is conditionally defined according to a number of mutually exclusive conditions $C_1, \ldots, C_n$, try to structure the proof as a case analysis that distinguishes n appropriate instances $C_1', \ldots, C_n'$ of the conditions $C_1, \ldots, C_n$.

In this case the function was `N.-` and the relevant conditions were two, depending on whether the second argument to `N.-` is `zero` or not. We will soon encounter more occasions illustrating this powerful heuristic.

Natural number subtraction

Another fundamental property of subtraction is that if the second argument is greater than zero but not greater than the first, then the result of the subtraction will be strictly less than the first argument:

$$0 < y \leq x \Rightarrow (x - y) < x.$$

We can state it in Athena as follows:

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

Natural number subtraction

Here is a proof:

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

Natural number subtraction

Other properties about subtraction follow:

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

define Plus-Minus-property := (forall x y z . x = y + z ==> x - y = z)
define cancellation :=  (forall x y . (x + y) - x = y)
define Times-Distributivity :=
   (forall x y z . x * y - x * z = x * (y - z))
```

Natural number subtraction

Consider the `Plus-Minus-property`:

$$\forall\, x, y, z \,.\, x = y + z \Rightarrow\ x - y = z.$$

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

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

       (!chain-> [((x - y) + y) = x             [C1]
                                = (z + y)       [C2]
                ==> (x - y = z)                 [Plus.=-cancellation]])
```

Natural number subtraction

From the `Plus-Minus-property` we easily obtain another cancellation property:

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

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

} # close module Minus
} # close module N
```

Ordered lists

We consider lists of natural numbers and use the order operators < and <= on natural numbers, which we have already axiomatized and developed proofs about.

First, we define a *membership* predicate on lists as follows:

```
extend-module List {

  declare in: (S) [S (List S)] -> Boolean [[id (alist->clist id)]]

  module in {
    assert* def := [(~ x in nil)
                    (x in h::t <==> x = h | x in t)]

    define [empty nonempty] := def
  }
```

Ordered lists

Some properties about list membership:

```
define head    := (forall x L . x in x::L)
define tail    := (forall x y L . x in L ==> x in y::L)
define of-join := (forall L M x . x in L ++ M <==> x in L | x in M)
```

Ordered lists

We introduce ordered lists of natural numbers not as a new datatype but as a subset of `(List N)` values *defined by a predicate*, `List.ordered`.

- We define this predicate constructively, by first introducing a binary predicate, which compares an `N` value with a `(List N)` value (continuing within an extension of the `List` module):

```
declare <=L: [N (List N)] -> Boolean [[int->nat (alist->clist int->nat)]]
```

- We define (x `<=L` L) to hold iff x is less than or equal to the first element of L or if L is empty:

```
module <=L {
  assert* def := [(x <=L nil)
                  (x <=L h::_ <==> x <= h)]
  define [empty nonempty] := def
```

Ordered lists

```
  define left-transitive :=
    (forall L x y . x <= y & y <=L L ==> x <=L L)

  define before-all-implies-before-first :=
    (forall L x . (forall y . y in L ==> x <= y) ==> x <=L L)

  define append :=
    (forall L M x . x <=L L & x <=L M ==> x <=L L ++ M)

} # close module <=L
```

Ordered lists

Now we can define the ordered predicate as follows:

```
declare ordered: [(List N)] -> Boolean [[(alist->clist int->nat)]]

module ordered {

  assert* def := [(ordered nil)
                  (ordered h::t <==> h <=L t & ordered t)]

  define [empty nonempty] := def

} # close module ordered
```

Ordered lists

Some properties:

```
extend-module ordered {

  define head := (forall L x . ordered x::L ==> x <=L L)
  define tail := (forall L x . ordered x::L ==> ordered L)

  define first-to-rest-relation :=
    (forall L x y .  ordered x::L & y in L ==> x <= y)

  define cons :=
    (forall L x . ordered L & (forall y . y in L ==> x <= y)
                  ==> ordered x::L)

  define append :=
    (forall L M . ordered L &
                  ordered M &
                  (forall x y . x in L & y in M ==> x <= y)
                 ==> ordered L ++ M)
```

Ordered lists

A proof of the `first-to-rest-relation` using induction on lists:

```
by-induction first-to-rest-relation {
  nil =>
    pick-any x y
      assume (ordered x::nil & y in nil)
        let {not-in := (!chain-> [true ==> (~ y in nil) [in.empty]])}
          (!from-complements (x <= y) (y in nil) not-in)
| (L as (z :: M)) =>
  let {ind-hyp := (forall x y . ordered x::M & y in M ==> x <= y)}
    conclude (forall x y . ordered x::L & y in L ==> x <= y)
      pick-any x:N y:N
        <D>
} # by-induction

  } # close module ordered
} # close module List
```

Ordered lists

Deduction <D> for inductive step (L as (z :: M)) to prove ordered x::L & y in L ==> x <= y follows:

```
assume (ordered x::z::M & y in z::M)
  let {p0 := <D0>;    # x <= z  &  z <=L M & ordered M
       p1 := <D1>;    # ordered M
       p2 := <D2>}    # ordered x::M
(!cases (!chain<- [(y = z | y in M) <==
                   (y in z::M)                  [in.nonempty]])
   assume (y = z)
     (!chain-> [p0 ==> (x <= z)                 [left-and]
                ==> (x <= y)                    [(y = z)]])
   (!chain [(y in M)
        ==> (p2 & y in M)                       [augment]
        ==> (x <= y)                            [ind-hyp]]))
```

Ordered lists

Deductions <D0> and <D1> extract two consequences from

```
ordered x::z::M
```

- `x <= z & z <=L M & ordered M`
- `ordered M`

```
            p0 := (!chain->
                     [(ordered x::z::M)
                   ==> (x <=L z::M &
                        ordered z::M)                    [nonempty]
                   ==> (x <=L z::M & z <=L M &
                        ordered M)                       [nonempty]
                   ==> (x <= z  &
                        z <=L M &
                        ordered M)                       [<=L.nonempty]]);

           p1 := (!chain-> [p0 ==> (ordered M)    [prop-taut]]);
```

Ordered lists

Deduction <D2> infers one more consequence from `ordered x::z::M`

- `ordered x::M`

```
        p2 := (!chain-> [p0
                         ==> (x <= z & z <=L M)     [prop-taut]
                         ==> (x <=L M)              [<=L.left-transitive]
                         ==> (x <=L M & p1)         [augment]
                         ==> (ordered x::M)         [nonempty]])
```

`p0` and `p2` were directly used in deduction <D> above.

Binary search trees

- Another important datatype related to ordering properties is that of *binary search trees*, a subset of binary trees satisfying a condition relating elements of the tree by their order.
- First, let's define binary trees:

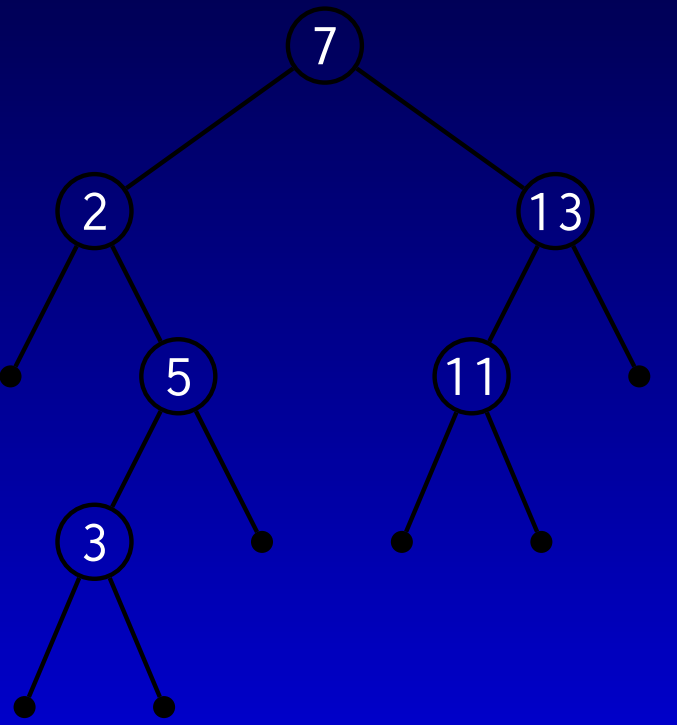
```
datatype (BinTree S) :=  null
                       | (node (BinTree S) S (BinTree S))

assert (datatype-axioms "BinTree")
```

- We wish to interpret `null` as an empty tree (no elements), and (node L x R) as a tree with element x at its root, L as its "left subtree," and R as its "right subtree."

Binary search trees

A sample binary tree with `Int` elements follows:

```
define tree1 :=
  (node (node null
              2
              (node (node null
                          3
                          null)
                    5
                    null))
        7
        (node (node null
                    11
                    null)
              13
              null))
```



Binary search trees

The first function we define on binary trees is a predicate for testing whether a value of sort `S` is present in a `(BinTree S)`.

```
module BinTree {
  define ++ := List.++
  define [x x' y T T' L R] :=
         [?x ?x' ?y ?T:(BinTree 'S1) ?T':(BinTree 'S2)
          ?L:(BinTree 'S3) ?R:(BinTree 'S4)]

  declare in: (S) [S (BinTree S)] -> Boolean

  module in {
    assert* def :=
      [(~ x in null)
       (x in (node L y R) <==> x = y | x in L | x in R)]

    define [empty nonempty] := def
} # close module in
} # close module BinTree
```

Binary search trees

A few simple lemmas in `BinTree.in`:

```
define root  := (forall x L y R . x = y  ==> x in (node L y R))
define left  := (forall x L y R . x in L ==> x in (node L y R))
define right := (forall x L y R . x in R ==> x in (node L y R))
```

Binary search trees

We define a function `inorder` from `(BinTree S)` to `(List S)` that produces a list of the tree elements ordered so that the root element appears between the elements of the left subtree and those of the right subtree, and recursively the elements are in this order within each subtree.

```
extend-module BinTree {

  declare inorder: (S) [(BinTree S)] -> (List S)

  module inorder {

    assert* def :=
        [(inorder null = nil)
         (inorder (node L x R) = (inorder L) ++ x::inorder R)]

    define [empty nonempty] := def
  }
```

Binary search trees

As an example, we can apply `inorder` to the binary tree `tree1`:

```
> (eval inorder tree1)

Term: (:: 2
          (:: 3
              (:: 5
                  (:: 7
                      (:: 11
                          (:: 13
                              nil:(List Int)))))))
```

Binary search trees

We first relate `BinTree.inorder` to `BinTree.in`:

```
overload in List.in

extend-module inorder {
  define in-correctness := (forall T x . x in inorder T <==> x in T)
```

- Note the overloading, which allows `in` to denote both `List.in` and `BinTree.in`, disambiguated by the sort of its second argument. In `in-correctness`, therefore, the first occurrence of `in` stands for `List.in` while the second stands for `BinTree.in`.

To prove `BinTree.in-correctness`, we can separately prove each of the implications by induction, then combine the results with `equiv`.

```
define in-correctness-1 := (forall T x . x in inorder T ==> x in T)
define in-correctness-2 := (forall T x . x in T ==> x in inorder T)
```

Binary search trees

Principle .1: Mathematical Induction for Binary Trees

To prove $\forall\ T\ .\ P(T)$ where T ranges over binary trees, it suffices to prove:

1. *Basis Case*: $P(\mathit{null})$.
2. *Induction Step*:
   $$\forall\ L\ R\ .\ P(L) \wedge P(R) \Rightarrow \forall\ x\ .\ P(\mathit{node}(L, x, R)).$$

In the induction step, the assumptions $P(L)$ and $P(R)$ are called the *induction hypotheses*.

Binary search trees

- As we have seen with natural numbers and lists, Athena is able to derive from the definition of the `BinTree` datatype the cases that must be handled by an inductive proof over binary trees.
- During evaluation, it also automatically enters both induction hypotheses into the assumption base so that they are available without having to explicitly assume them in the inductive step case.

Binary search trees

Here is a proof of `in-correctness-1`:

```
by-induction in-correctness-1 {
  null =>
  pick-any x
    assume (x in inorder null)
    let {in-nil := (!chain->
                      [(x in inorder null)
                   ==> (x in nil)                    [empty]]);
         -in-nil := (!chain-> [true
                             ==> (~ x in nil)        [List.in.empty]])}
      (!from-complements (x in null) in-nil -in-nil)
| (T as (node L y R)) =>
  <D_IS>
}
} # close module inorder
} # close module BinTree
```

Binary search trees

The inductive step proof <D_IS> for (T as (node L y R)) follows:

```
let {ind-hyp1 := (forall x . x in inorder L ==> x in L);
     ind-hyp2 := (forall x . x in inorder R ==> x in R)}
pick-any x
  assume hyp := (x in inorder T)
    let {D := (!chain-> [hyp
                         ==> (x in (inorder L) ++
                                   y::inorder R)  [nonempty]
                         ==> (x in inorder L |
                              x in y::inorder R)    [List.in.of-join]
                         ==> (x in inorder L |
                              x = y |
                              x in inorder R)       [List.in.nonempty]])}

    (!cases D <DL> <DN> <DR>)
```

Binary search trees

And the deductions for left, root, and right (<DL>, <DN>, and <DR>) cases follow respectively:

```
        (!chain [(x in inorder L)
             ==> (x in L)                          [ind-hyp1]
             ==> (x in T)                          [in.left]])
        (!chain [(x = y) ==> (x in (node L y R))  [in.root]])
        (!chain [(x in inorder R)
             ==> (x in R)                          [ind-hyp2]
             ==> (x in T)                          [in.right]])
```

Binary search trees

Finally, we focus on binary search trees, a special type of binary trees: those satisfying a predicate, `BST`. We define `BST` for natural numbers. We first give a constructive definition of `BST` in terms of two binary predicates: `no-smaller` and `no-larger`.

- The first takes a binary tree of natural numbers T and a natural number n and returns `true` iff every element of T is not smaller than (i.e., larger than or equal to) n.
- The `no-larger` predicate is similar except that it holds iff every element of T is smaller than or equal to n.

Then we derive from that definition a more abstract characterization theorem that has two parts, which are directly asserted as axioms (their derivations from `BinTree.BST.definition` are left as exercise 8.30).

Binary search trees

```
extend-module BinTree {
  define [x y z T L R] := [?x:N ...  ?T:(BinTree N) ...]
  define [< <=] := [N.< N.<=]

  declare BST: [(BinTree N)] -> Boolean
  declare no-smaller, no-larger: [(BinTree N) N] -> Boolean

  assert* no-smaller-def :=
    [(null no-smaller _)
     ((node L y R) no-smaller x <==> x <= y &
                                     L no-smaller x &
                                     R no-smaller x)]
  assert* no-larger-def :=
    [(null no-larger _)
     ((node L y R) no-larger x <==> y <= x &
                                    L no-larger x &
                                    R no-larger x)]
} # close module BinTree
```

Binary search trees

```
extend-module BinTree {
  module BST {
    assert* definition :=
      [(BST null)
       (BST (node L x R) <==> BST L & L no-larger x &
                              BST R & R no-smaller x)]

    # The two parts of the characterization result, here as axioms:
    assert empty := (BST null)
    assert nonempty :=
      (forall L y R .
        BST (node L y R) <==>
         BST L &
         (forall x . x in L ==> x <= y) &
         BST R &
         (forall z . z in R ==> y <= z))
  } # close module BST
} # close module BinTree
```

Summary and a connecting theorem

In this chapter, we have focused on:

- Defining inequality relations on natural numbers with (constructive) axioms and using proofs of selected theorems about these relations for further practice with induction, case analysis, implication chaining, and proof by contradiction.
- Defining subtraction of natural numbers and proving related theorems.
- Extending the discussion of the `List` datatype to properties of ordered lists over the natural numbers.
- Introducing binary trees over the natural numbers and a corresponding induction principle.
- Defining binary search trees over natural numbers, in preparation for our first example of algorithm specification and correctness.

Summary and a connecting theorem

We conclude this chapter with the statement of a theorem connecting three of the main ordering concepts we have studied—namely, that the `inorder` function applied to a binary search tree produces an ordered list (proved only for natural numbers).

```
extend-module BinTree {
  extend-module BST {

    define is-ordered := (forall T . BST T ==> List.ordered inorder T)

  } # close module BST
} # close module BinTree
```