CSCI.6962/4962 Software Verification— Fundamental Proof Methods in Computer Science (Arkoudas and Musser)—Chapter 11

Instructor: Carlos Varela
Rensselaer Polytechnic Institute
Spring 2018

A Binary Search Algorithm

Goal: to introduce reasoning about algorithms using a binary search algorithm as an example.

- defining the algorithm
- first correctness properties
- correctness of an optimized binary search algorithm

Defining the algorithm

Binary search is an algorithm capable of searching a binary tree containing n items with only $O(\log n)$ comparisons, assuming the tree is balanced.

- Roughly speaking, balanced means that each path from the root to a leaf node is about $\log_2 n$ in length.

To define the binary search function on binary trees of natural numbers, we start as usual with its declaration:

```
extend-module BinTree {

define [x y z T L R] :=
  [?x:N ?y:N ?z:N ?T:(BinTree N) ?L:(BinTree N) ?R:(BinTree N)]

  declare binary-search: [N (BinTree N)] -> (BinTree N)

  module binary-search {
# ...
```

Defining the algorithm

To define how `binary-search` is computed, we call upon a library procedure named `fun` to translate a simpler input into defining axioms:

```
assert axioms :=
  (fun
   [(binary-search x (node L y R)) =
        [(node L y R)          when (x = y)
         (binary-search x L)   when (x < y)
         (binary-search x R)   when (x =/= y & ~ x < y)]
    (binary-search x null) = null])

define [at-root go-left go-right empty] := axioms
```

Defining the algorithm

The effect of the `assert` of the `fun` together with the `define` is exactly the same as if we had written

```
assert* at-root :=
  (x = y ==> (binary-search x (node L y R)) = (node L y R))

assert* go-left :=
  (x < y ==> (binary-search x (node L y R)) = (binary-search x L))

assert* go-right :=
  (x =/= y & ~ x < y
   ==> (binary-search x (node L y R)) = (binary-search x R))

assert* empty := ((binary-search x null) = null)
```

The `fun` procedure

The procedure `fun` is used to define either a predicate or a nonpredicate. In either case, it takes one argument, a list containing *defining clauses*, and translates it into a list `[`s_1 $\ldots$ s_k`]` where each s_i is a sentence.

- A defining clause for a nonpredicate function f is a triplet (three consecutive list elements):
  - a term l of the form `(`f t_1 $\ldots$ t_n`)`, called a *left-hand-side* term;
  - the symbol =; and
  - a *right-hand-side term* or *list*.
- In the left-hand-side term, the number of arguments n must be the same as f's arity, and each argument t_i must be either a variable or a term of the sort required by f's i^{th} argument position (so that the whole term is legal).

The `fun` procedure

- In the case of a right-hand-side term r, r must be a term whose sort is the return sort of f and the corresponding sentence is the closure of $(l$ `=` $r)$.
- In the case of a right-hand-side list, it must consist of a sequence of triplets, each triplet consisting of a right-hand-side term r, followed by the meta-identifiers `'when`, followed by a guard p. For each such triplet, the corresponding sentence is the closure of $(p$ `==>` l `=` $r)$. (We can allow writing `when` instead of `'when` with `define when := 'when`.)

The `fun` procedure

The `fun` syntax is similar to multipart definitions in conventional mathematical notation, where one might write, e.g.,

$$\textit{binary-search}(x, \textit{node}(L, y, R)) = \begin{cases} \textit{node}(L, y, R) & \text{where } x = y \\ \textit{binary-search}(x, L) & \text{where } x < y \\ \textit{binary-search}(x, R) & \text{otherwise} \end{cases}$$

Instead of "otherwise", we write explicitily "where $x \neq y$ and $\sim x < y$".

The `fun` procedure

Writing the third condition as the conjunction of the negations of the preceding conditions, makes it explicit that the condition is disjoint from all of the previous ones. This helps with proof structures such as:

```
(!two-cases
  assume (x = y)
    # here at-root can be applied
      ...
  assume (x =/= y)
    (!two-cases
      assume (x < y)
        # here go-left can be applied, since (x < y) is
        # in the assumption base
        ...
      assume (~ x < y)
        # here go-right can be applied, since both (x =/= y)
        # and (~ x < y) are now in the assumption base
        ...
    ))
```

Efficiency considerations

While the order of presentation makes no difference for the use of axioms in proofs, it can have a significant effect on efficiency when the axioms are used for computation.

In Athena, axioms defining a function f are compiled into code by the name f` or f``.

Let's look at the code that is compiled from the axioms generated by our `fun` definition of `binary-search`:

```
define (binary-search` t1 t2) :=
  match [t1 t2] {
      [x3 (node x1 x4 x2)] =>
        check {(=` x3 x4) => (node x1 x4 x2)
               | (N.Less.<`` x3 x4) => (binary-search` x3 x1)
               | else => (binary-search` x3 x2)}
    | [x1 null:(BinTree N)] => null:(BinTree N)
    }
```

Efficiency considerations

- The axioms have been compiled into a `match` expression with two clauses, one for each of the two constructors of the `BinTree` datatype, which constitute a case analysis for the structure of the second argument to `binary-search`.
- The clause corresponding to the `node` constructor includes a further case analysis implemented by a `check`, which first tests whether the key is equal to the root value, and then whether it is less than it.
- But note that the last clause of the `check` expression does not include a proper condition, having a catch-all `else` guard instead.

Efficiency considerations

- Whereas a completely straightforward translation of the corresponding clause of the `fun` definition, namely

  `(binary-search x R) when (x =/= y & ~ x < y)`

  would result in:

```
check {(=` x3 x4) => (node x1 x4 x2)
     | (N.Less.<`` x3 x4) => (binary-search` x3 x1)
     | (&& (negate (=` x3 x4)) (negate (N.Less.<`` x3 x4)))  =>
          (binary-search` x3 x2)}
```

- But because each conjunct in the last guard negates a condition already tested in previous clauses, the guard simplifies to `(&& true true)`, and thus it can be replaced by `else`.

Efficiency considerations

- These simplifications depend on the order of the given conditions.
- For most purposes, the following guarantees about the compiled Athena code suffice:
  1. the order in which `check` clauses (or potentially `match` clauses with `where` guards) are listed is the same as the order in which the axioms with the corresponding conditions are presented; and
  2. each guard in a clause is simplified by removing the negation of any guard of a preceding clause.
- These guarantees ensure that lines in a `fun` serve, for purposes of computation, essentially like an "if-then-else" or "cond" construct in conventional high-level languages.

Correspondence to other languages

In a programming language that allows datatype constructor patterns as formal parameters of function definitions, such as ML or Haskell, one can write the binary search definition similarly to using `fun`, e.g., in ML:

```
datatype BinTree = null |
                   node of BinTree * int * BinTree
fun binarySearch(x,node(L,y,R)) =
     if x = y then
        node(L,y,R)
     else if x < y then
        binarySearch(x,L)
     else
        binarySearch(x,R)
  | binarySearch(x,null) = null
```

Correspondence to other languages

In languages which lack such pattern matching, we would be restricted to using only variables as formal parameters. Then, in the body of the function, we would need to replace occurrences of pattern variables with the corresponding accessor function call or other component access notation, e.g., in C++:

```
struct node;
typedef node* BinTree;

struct node {
  BinTree left; int key; BinTree right;
  node(BinTree L, int y, BinTree R) :
      left(L), key(y), right(R) {}
};
```

Correspondence to other languages

```
BinTree binarySearch(int x, BinTree T)
{
  if (T == NULL)
    return NULL;
  if (x == T->key)
    return T;
  if (x < T->key)
    return binarySearch(x, T->left);
  return binarySearch(x, T->right);
}
```

Note that in this case, the test for the empty tree must be done first, since otherwise the expression `T->key`, etc., would cause an error when `T` is a null pointer.

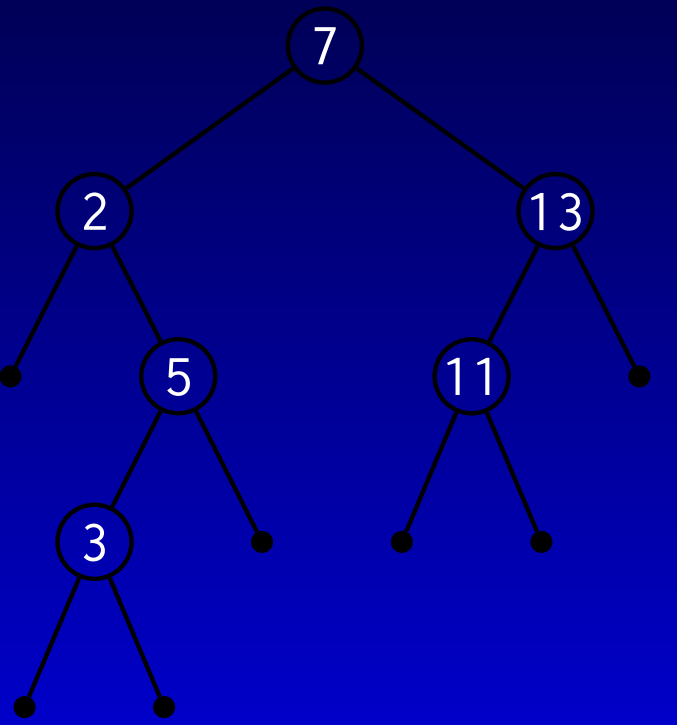
Interface design

- One additional factor in algorithm design is the interface.
- In the case of `binary-search`, it would be simpler to return a Boolean value, `true` if the target value is found, `false` otherwise. (Most significantly, that interface would make some of the issues we have to deal with in the upcoming proofs disappear.)
- By returning a node or `null`, however, we still have an easy test for success—`((binary-search` x T`) =/= null)`—and we make it possible to continue with other computations at the position in the tree where the search value is found, without recomputing that position.

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))
```



Testing with evaluation

We can use input expansion and output transformation to work with trees such as the one defined earlier.

```
expand-input binary-search [int->nat int-tree->nat-tree]
transform-output eval [nat-tree->int-tree]

> (eval (binary-search 5 tree1))

Term: (node (node null:(BinTree Int)
                  3
                  null:(BinTree Int))
            5
            null:(BinTree Int))

> (eval (binary-search 12 tree1))

Term: null:(BinTree Int)
```

First correctness properties

The first correctness properties we consider concern what the result returned by (`binary-search` x T) implies about the presence or absence of x in T.

```
extend-module binary-search {

  define found :=
    (forall T . BST T ==>
      forall x L y R . (binary-search x T) = (node L y R) ==>
                           x = y & x in T)

  define not-found :=
    (forall T . BST T ==>
      forall x . (binary-search x T) = null ==> ~ x in T)
```

First correctness properties

The proof of found is by induction on binary trees, with basis case:

```
define tree-axioms := (datatype-axioms "BinTree")

define (binary-search-found-base) :=
 conclude (BST null ==>
             forall x L y R .
               (binary-search x null) = (node L y R)
               ==> x = y & x in null)
    assume (BST null)
      pick-any x:N L:(BinTree N) y:N R:(BinTree N)
        assume i := ((binary-search x null) = (node L y R))
          let {A := (!chain [null:(BinTree N)
                             = (binary-search x null)    [empty]
                             = (node L y R)              [i]]);
               -A := (!chain-> [true
                             ==> (null =/= (node L y R)) [tree-axioms]])}
          (!from-complements (x = y & x in null) A -A)
```

First correctness properties

Now we can check the basis case proof before going on to the induction step:

```
> (!binary-search-found-base)

Theorem: (if (BinTree.BST null:(BinTree N))
             (forall ?x:N
               (forall ?L:(BinTree N)
                 (forall ?y:N
                   (forall ?R:(BinTree N)
                     (if (= (BinTree.binary-search ?x:N
                                                   null:(BinTree N))
                            (node ?L:(BinTree N)
                                  ?y:N
                                  ?R:(BinTree N)))
                         (and (= ?x:N ?y:N)
                              (BinTree.in ?x:N
                                          null:(BinTree N)))))))))
```

First correctness properties

We therefore continue with the induction step. To simplify the notation somewhat, we introduce a property procedure, `found-property`, which allows us to express the desired `found` result as follows:

```
(forall T . BST T ==> found-property T).
```

```
define [x1 y1 L1 R1] := [?x1:N ?y1:N ?L1:(BinTree N) ?R1:(BinTree N)]

define (found-property T) :=
  (forall x L1 y1 R1 .
     (binary-search x T) = (node L1 y1 R1) ==> x = y1 & x in T)
```

After applying the definition of `BST` to `(node L y R)` and combining the result with the induction hypotheses, the proof proceeds by case analysis, setting up cases according to whether `x` is found at the root, in the left subtree, or in the right subtree, and applying the corresponding `binary-search` axiom in each case.

First correctness properties

```
define binary-search-found-step :=
 method (T)
   match T {
     (node L:(BinTree N) y:N R:(BinTree N)) =>
       let {[ind-hyp1 ind-hyp2] := [(BST L ==> found-property L)
                                    (BST R ==> found-property R)]}
       assume hyp := (BST T)
         conclude (found-property T)
          let {p0 := (BST L &
                      (forall x . x in L ==> x <= y) &
                      BST R &
                      (forall z . z in R ==> y <= z));
               _ := (!chain-> [hyp ==> p0                [BST.nonempty]]);
               fpl := (!chain-> [p0 ==> (BST L)            [prop-taut]
                                    ==> (found-property L) [ind-hyp1]]);
               fpr := (!chain-> [p0 ==> (BST R)            [prop-taut]
                                    ==> (found-property R) [ind-hyp2]])}
```

First correctness properties

```
pick-any x:N L1 y1:N R1
   let {subtree := (node L1 y1 R1)}
    assume hyp' := ((binary-search x T) = subtree)
     conclude (x = y1 & x in T)
       (!two-cases
         assume (x = y)
           (!both conclude (x = y1)
                  (!chain->
                 [T = (binary-search x T)    [at-root]
                    = subtree                [hyp']
                    ==> (y = y1)             [tree-axioms]
                    ==> (x = y1)             [(x = y)]])
                conclude (x in T)
                  (!chain-> [(x = y)
                             ==> (x in T)    [in.root]]))
```

First correctness properties

```
                    assume (x =/= y)
                       (!two-cases
                         assume (x < y)
                           (!chain-> [(binary-search x L)
                                      = (binary-search x T) [go-left]
                                      = subtree             [hyp']
                                     ==> (x = y1 & x in L)  [fpl]
                                     ==> (x = y1 & x in T)  [in.left]])
                         assume (~ x < y)
                           (!chain-> [(binary-search x R)
                                      = (binary-search x T) [go-right]
                                      = subtree             [hyp']
                                     ==> (x = y1 & x in R)  [fpr]
                                     ==> (x = y1 & x in T)  [in.right]])))
   }
```

First correctness properties

The complete proof is then:

```
> by-induction found {
    null => (!binary-search-found-base)
  | (T as (node _ _ _)) => (!binary-search-found-step T)}
Theorem: (forall ?T:(BinTree N)
           (if (BinTree.BST ?T:(BinTree N))
               (forall ?x:N
                 (forall ?L:(BinTree N)
                   (forall ?y:N
                     (forall ?R:(BinTree N)
                       (if (= (BinTree.binary-search ?x:N
                                                     ?T:(BinTree N))
                              (node ?L:(BinTree N)
                                    ?y:N
                                    ?R:(BinTree N)))
                           (and (= ?x:N ?y:N)
                                (BinTree.in ?x:N
                                            ?T:(BinTree N)))))))))))
```

First correctness properties

Turning to the proof of `not-found`, we will write the basis case and induction step proofs inline.

- The basis case is an immediate consequence of `BinTree.in.empty`.
- In the induction step, we again start by applying the definition of `BST` to `T` and combining the result with the induction hypotheses. The conclusion we seek being a negative, we naturally use proof by contradiction, but there are several different cases to be considered, and we must find a contradiction within each.

We define a property procedure (`not-found-prop`) for this proof, too.

```
define (not-found-prop T) :=
  (forall x . (binary-search x T) = null ==> ~ x in T)
```

First correctness properties

```
by-induction not-found {
  null =>
    assume (BST null)
      conclude (not-found-prop null)
        pick-any x:N
          assume ((binary-search x null) = null)
            (!chain-> [true ==> (~ x in null)    [in.empty]])
| (T as (node L y:N R)) =>
    let {p1 := (not-found-prop L);
         p2 := (not-found-prop R);
         [ind-hyp1 ind-hyp2] := [(BST L ==> p1) (BST R ==> p2)]}
    assume hyp := (BST T)
      conclude (not-found-prop T)
```

First correctness properties

```
let {smaller-in-left := (forall x . x in L ==> x <= y);
     larger-in-right := (forall z . z in R ==> y <= z);
     p0 := (BST L &
            smaller-in-left &
            BST R &
            larger-in-right);
     _ := (!chain-> [hyp ==> p0                    [BST.nonempty]]);
     _ := (!chain-> [p0 ==> smaller-in-left        [prop-taut]]);
     _ := (!chain-> [p0 ==> larger-in-right        [prop-taut]]);
     _ := (!chain-> [p0
                 ==> (BST L)                       [prop-taut]
                 ==> (not-found-prop L)            [ind-hyp1]]);
     _ := (!chain-> [p0
                 ==> (BST R)                       [prop-taut]
                 ==> (not-found-prop R)            [ind-hyp2]])}
```

First correctness properties

```
pick-any x
  assume hyp' := ((binary-search x T) = null)
    (!by-contradiction (~ x in T)
      assume (x in T)
        let {disj := (!chain-> [(x in T)
                                ==> (x = y  |
                                     x in L |
                                     x in R)      [in.nonempty]])}
          (!two-cases
            assume (x = y)
             (!absurd
               (!chain [null:(BinTree N)
                        = (binary-search x T)     [hyp']
                        = T                       [at-root]])
               (!chain-> [true
                          ==> (null =/= T)        [tree-axioms]])))
```

First correctness properties

```
                  assume (x =/= y)
                    (!two-cases
                     assume (x < y)
                       (!cases disj
                         assume (x = y)
                           (!absurd (x = y) (x =/= y))
                         assume (x in L)
                           (!absurd
                             (x in L)
                             (!chain->
                              [(binary-search x L)
                             = (binary-search x T)     [go-left]
                             = null                    [hyp']
                            ==> (~ x in L)             [p1]]))
```

First correctness properties

```
                    assume (x in R)
                      (!absurd
                        (x < y)
                        (!chain->
                          [(x in R)
                         ==> (y <= x)     [larger-in-right]
                         ==> (~ x < y)    [N.Less=.trichotomy4]]))))
                assume (~ x < y)
                  (!cases disj
                   assume (x = y)
                     (!absurd (x = y) (x =/= y))
```

First correctness properties

```
                    assume (x in L)
                      (!absurd
                       (x =/= y)
                       (!chain-> [(x in L)
                                  ==> (x <= y)          [smaller-in-left]
                                  ==> (x < y | x = y) [N.Less=.definition]
                                  ==> (~ x < y &
                                       (x < y | x = y)) [augment]
                                  ==> (x = y)           [prop-taut]]))
                    assume (x in R)
                      (!absurd
                       (x in R)
                       (!chain->
                        [(binary-search x R)
                       = (binary-search x T)      [go-right]
                       = null                     [hyp']
                      ==> (~ x in R)              [p2]]))))))
} # by-induction
```

An optimized binary search algorithm

The binary search algorithm first compares for equality of the search value with the current node's key.

Notice, however, that until the equality comparison succeeds, the algorithm expends two comparisons per tree level. And in a balanced n-element tree, most searches will require descending through almost $\log_2 n$ levels (since about $n/2$ elements reside in the bottom level, $n/4$ at the next lowest level, etc.) Thus, on average, a total of almost $2 \log_2 n$ comparisons will be required. An alternative definition is:

```
assert axioms :=
  (fun
   [(binary-search x (node L y R)) =
        [(binary-search x L)  when (x < y)
         (binary-search x R)  when (y < x)
         (node L y R)         when (~ x < y & ~ y < x)]
    (binary-search x null) = null])
define [go-left go-right at-root empty] := axioms
```

Correctness of optimized algorithm

With this control structure, if the search value is in the left subtree, only one comparison is required to descend a level; if it is in the right subtree, two are required.

- Assuming it is equally likely to take one path or the other at each level, a total of about $1.5 \log_2 n$ comparisons are required, on average—a 25 percent reduction from the traditional version of the algorithm.
- This is a significant optimization, but before adopting it as the preferred version of binary search, we should be sure that it is correct.

This is a small example, but it is illustrative of one of the most important applications of logic and proof in computer science, correctness of algorithm optimizations.

Correctness of optimized algorithm

One way to prove the optimized algorithm is to modify the proofs for `found` and `not-found` by restructuring the case analyses to match the new `when` conditions.

Another way to establish `found` and `not-found` using the optimized version of `binary-search` is to start with the new axioms of the optimized version and prove the axioms of the original version as lemmas. Then the original proofs will still work.

Correctness of optimized algorithm

That is, from the new axioms first prove

```
define at-root' :=
  (forall x L y R .
    x = y ==> (binary-search x (node L y R)) = (node L y R))

define go-right' :=
  (forall x L y R .
    x =/= y & ~ x < y ==>
    (binary-search x (node L y R)) = (binary-search x R))
```

(`go-left` is unchanged). Then reprove `not-found` with

```
let {[at-root go-right] := [at-root' go-right']}
  conclude not-found
    # ...  same proof as above.
```

and similarly for `found`.