Modeling Asymptotic Complexity Using ACL2

ACL2 Workshop 2022

William D. Young
Department of Computer Science
University of Texas at Austin

Last updated: May 25, 2022 at 11:30

Asymptotic Complexity

Asymptotic complexity: a systematic approach to characterizing the limiting behavior of a function as its argument tends toward infinity.

A collection of notations, collectively called *Bachmann-Landau* notations allow characterizing the behavior of one function in terms of another:

- $O(g(n))$ (Big-O): the set of functions asymptotically upper bounded by $g(n)$.
- $\Omega(g(n))$ (Big Omega): functions lower bounded by $g(n)$.
- $\Theta(g(n))$ (Big Theta): functions upper and lower bounded by $g(n)$.

There are also corresponding "little" notations that provide strict bounds.

Big-O

Most common is the big-O notation for estimating an upper bound on the time or space complexity of an algorithm.

Definition: Let f and g be functions $f, g : N \to R^+$. We say that $f(n) = O(g(n))$ if there exist positive integers c and n_0 such that for every integer $n \geq n_0$,

$$f(n) \leq c \cdot g(n).$$

When $f(n) = O(g(n))$ we say that $g(n)$ is an *asymptotic upper bound* for $f(n)$).

The Goal of this Research

The goal of this research: formalize and prove Big-O properties of algorithms using ACL2.

- How to characterize the algorithms;
- How to express the higher-order notion of Big-O in ACL2;
- How to count "steps" in the execution;
- How to prove that the number of steps is $O(g(n))$, for some $g(n)$.

The Language

We embed a simple imperative language in ACL2 via an operational semantics.

Consists of:

- expression sublanguage: literals, variables, arithmetic and logical expressions;
- statements: skip, assign, return, if-else, while, sequence.

The Operational Semantics

The semantics is provided by an typical interpreter function:

```
(run stmt status vars steps count)
```

where:

- `stmt`: the statement to execute;
- `status`: the current state of the execution (only proceeds if status is `'OK`);
- `vars`: a variable alist;
- `steps`: a running count of the number of execution steps;
- `count`: the clock argument to guarantee termination.

It returns a triple:

```
(status, vars, steps)
```

Semantics

```
(defun run (stmt status vars steps count)
  (if (not (okp status))
      (mv status vars steps)
    (if (zp count)
        (mv 'timed-out vars steps)
      (case
        (operator stmt)
            ...
        (while (mv-let (test-stat test-val test-steps)
                       (exeval (param1 stmt) t vars)
                       (if (not test-stat)
                           (run-error vars)
                         (if test-val
                             (mv-let (body-stat body-vars body-steps)
                                     (run (param2 stmt) status vars
                                          (+ 1 steps test-steps)
                                          count)
                                     (run stmt body-stat body-vars
                                               body-steps
                                               (1- count)))
                           (mv 'ok vars (+ 1 test-steps steps))))))
        (otherwise (run-error vars))))))
```

Binary Search: Python Version

```
def BinarySearch( key, lst ):
    low = 0
    high = len(lst) - 1
    while (high >= low):
        mid = (low + high) // 2
        if key == lst[mid]:
            return mid
        elif key < lst[mid]:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

Binary Search: Our Version

Here's a hand translation of the Python Binary Search routine into our simple iterative language:

```
(defun binarysearch (key lst)
  `(seqn (assign (var low) (lit . 0))
         (assign (var high) (- (len ,lst) (lit . 1)))
         (while (<= (var low) (var high))
           (seq (assign (var mid)
                        (// (+ (var low) (var high)) (lit . 2)))
                (if-else (== ,key (ind (var mid) ,lst))
                         (return (var mid))
                         (if-else (< ,key (ind (var mid) ,lst))
                                  (assign (var high)
                                          (- (var mid) (lit . 1)))
                                  (assign (var low)
                                          (+ (var mid) (lit . 1)))))))
         (return (lit . -1)))))
```

Executing our Program

```
ACL2 !>(run (binarysearch '(lit . 4)
                          '(lit . (0 1 2 3 4 5 6 7)))
            'OK nil 0 10)
(RETURNED ((LOW . 4)
           (HIGH . 4)
           (MID . 4)
           (RESULT . 4))
          77)

ACL2 !>(run (binarysearch '(var key) '(var lst))
            'OK '((key . 4) (lst . (0 1 3 5 7 9 10))) 0 10)
(RETURNED ((KEY . 4)
           (LST 0 1 3 5 7 9 10)
           (LOW . 3)
           (HIGH . 2)
           (MID . 2)
           (RESULT . -1))
          91)
```

Our Proof Approach

We prove two things simultaneously:

1. **Functional correctness:** the program actually computes the correct result;
2. **Asymptotic complexity:** the program is a member of a certain Big-O class.

Functional Correctness

```
(defun recursiveBS-helper (key lst low mid high calls)
  ;; This performs a recursive binary search for key in
  ;; lst[low..high].  It returns a 5-tuple (success low mid high calls).
  ;; We need all of those values to do the recursive proof.
  (if (or (< high low) (not (natp low)) (not (integerp high)))
      (mv nil low mid high calls)
    (let ((newmid (floor (+ low high) 2)))
      (if (equal key (nth newmid lst))
          (mv t low newmid high calls)
        (if (< key (nth newmid lst))
            (recursiveBS-helper key lst low
                                newmid (1- newmid) (1+ calls))
          (recursiveBS-helper key lst (1+ newmid)
                              newmid high (1+ calls)))))))

(defun recursiveBS (key lst)
  (mv-let (success low mid high calls)
          (recursiveBS-helper key lst 0 nil (1- (len lst)) 0)
          (declare (ignore low high calls))
          (if success mid -1)))
```

Relating the Iterative and Recursive Versions

As an example, if `(member-equal keyval lstval)`, where `keyval` and `lstval` are values stored in the alist in appropriate variables, then the following is true:

```
(equal (run (binarysearch key lst) 'ok vars 0 count)
       (mv-let (success endlow endmid endhigh endcalls)
               (recursiveBS-helper keyval lstval
                                   0 nil (1- (len lstval)) 0)
               (mv 'returned
                   (store 'result endmid
                          (store 'mid endmid
                                 (store 'high endhigh
                                        (store 'low endlow vars))))
                   (+ 25 (* 26 endcalls)))))
```

Notice this shows that the iterative and recursive versions are in lock-step.

A Simpler Recursive Version

We define a simpler recursive version of binary search, without the local variables:

```
(defun recursiveBS2-helper (key lst low high)
  (if (or (< high low)
          (not (natp low))
          (not (integerp high))
          )
      -1
    (let ((newmid (floor (+ low high) 2)))
      (if (equal key (nth newmid lst))
          newmid
        (if (< key (nth newmid lst))
            (recursiveBS2-helper key lst low (1- newmid))
          (recursiveBS2-helper key lst (1+ newmid) high))))))

(defun recursiveBS2 (key lst)
  (recursiveBS2-helper key lst 0 (1- (len lst))))
```

Two Values are Equivalent

We prove that the two recursive versions are equivalent:

```
(defthm recursiveBS-versions-equivalent
  (implies (and (number-listp lst)
                (acl2-numberp key))
           (equal (recursiveBS key lst)
                  (recursiveBS2 key lst))))
```

And that the simpler version actually searches:

```
(defthm recursiveBS2-searches
  (implies (and (acl2-numberp key)
                (number-listp lst)
                (sorted lst))
           (let ((index (recursiveBS2 key lst)))
             (and (implies (member-equal key lst)
                           (equal (nth index lst) key))
                  (implies (not (member-equal key lst))
                           (equal index -1))))))
```

Reminder: Definition of Big-O

Recall our earlier definition of Big-O:

> **Definition:** Let f and g be functions $f, g : N \rightarrow R^+$. We say that $f(n) = O(g(n))$ if there exist positive integers c and n_0 such that for every integer $n \geq n_0$,
>
> $$f(n) \leq c \cdot g(n).$$

But this is higher order!

So instead of defining `function-Big-O`, we defined `function-logarithmic`:

```
(defun log2 (n)
  (if (zp n)
      0
    (1+ (log2 (floor n 2)))))

(defun-sk function-logarithmic1 (program log-of c n0 vars count)
  (forall (n)
          (implies (and (equal n (len log-of))
                        (<= n0 n))
                   (mv-let (run-stat run-vars run-steps)
                           (run program 'ok vars 0 count)
                           (declare (ignore run-stat run-vars))
                           (and (<= 0 run-steps)
                                (<= run-steps (* c (log2 n))))))))

(defun-sk function-logarithmic2 (program log-of vars count)
  (exists (c n0)
          (and (posp c)
               (posp n0)
               (function-logarithmic1 program log-of c n0 vars count))))
```

Proving It's Logarithmic

This is the theorem that shows our iterative program is $O(\log_2(n))$:

```
(defthm binarysearch-logarithmic-lemma
  (let ((keyval (lookup 'key vars))
        (lstval (lookup 'lst vars)))
    (implies
       (and (acl2-numberp keyval)
            (number-listp lstval)
            (sorted lstval)
            (integerp count)
            (not (timed-outp
                   run-status (run (binarysearch '(var key) '(var lst))
                                                 'ok vars 0 count)))))
       (function-logarithmic2 (binarysearch '(var key) '(var lst))
                              (lookup 'lst vars) vars count))))
```

I proved a similar theorem for linear search and some other simple programs.

Subtleties of the Approach

- Counting steps may be useful for other purposes.
- But, it's very sensitive to the way the program is written.
- Counting is at the source code level; maybe object code would be better.
- Object level programs could already be optimized.
- The proofs are fragile and tedious.

It would be great to find a more robust and less labor intensive methodology.

The hardest part was proving the equivalence of the iterative and recursive versions of the program.