Modeling Asymptotic Complexity Using ACL2

William D. Young
Department of Computer Science
University of Texas at Austin
Austin, Texas
byoung@cs.utexas.edu

The theory of asymptotic complexity provides an approach to characterizing the behavior of programs in terms of bounds on the number of computational steps executed or use of computational resources. We describe work using ACL2 to prove complexity properties of programs implemented in a simple imperative programming language embedding via an operational semantics in ACL2. We simultaneously prove functional properties of a program and its complexity. We illustrate our approach by describing proofs about a binary search algorithm, proving both that it implements binary search on a sorted list and that it is $O(\log_2(n))$, where n is the length of the list.

1 Introduction

The theory of asymptotic complexity provides a systematic approach to characterizing the limiting behavior of a function as its argument tends toward infinity. A family of notations, collectively called *Bachmann-Landau* or asymptotic notations, allow characterizing upper bounds, lower bounds, and tight bounds of one function in terms of another function. These have been used in analytic number theory for more than a century, but became ubiquitous in computing following the groundbreaking work of Hartmanis and Stearns[14]. These notations provide convenient abstract characterizations of the difficulty of computing specific algorithms, independent of any particular hardware platform.

The most common asymptotic notation is the big-O notation for estimating an upper bound on the time or space complexity of an algorithm. It can be defined as follows[23]:

> **Definition:** Let f and g be functions $f, g : N \to R^+$. We say that $f(n) = O(g(n))$ if positive integers c and n_0 exist 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 *upper bound* for $f(n)$), or more precisely, that $g(n)$ is an *asymptotic upper bound* for $f(n)$, to emphasize that constant factors are suppressed.

As an example, if $f(n) = 3n^3 + 7n^2 - 5$, then we can easily prove that $f(n) = O(n^3)$, choosing $c = 4$ and $n_0 = 7$ (though infinitely many other choices of c and n_0 would work as well). In the context of computational complexity, the function being bounded typically describes the number of computational steps or the number of units of storage required in execution of an algorithm. Thus, for example, we might say that "binary search is $O(\log_2(n))$," where n is the length of the sorted list being searched, for the following reason: Executing the algorithm takes no more than $c \cdot \log_2(n)$ steps whenever $n \geq n_0$, for some positive integers c, n_0. The proof entails exhibiting specific witnesses for c and n_0.

The goal of the research described in this paper is to formalize and prove big-O properties of algorithms using ACL2. Since the definition of big-O is logically higher-order, we make certain concessions

Rob Sumners, Cuong Chau (Eds.): 17th ACL2 Workshop (ACL2 2022)
EPTCS 359, 2022, pp. 83–98, doi:10.4204/EPTCS.359.9

to the first-order nature of the ACL2 logic. Specifically, we proved big-O properties of specific algorithms, without defining big-O abstractly. For example, we proved that an implementation of binary sort has complexity bounded by $\log_2(n)$, where n is the length of the list being searched.[1] To do this we defined *logarithmic complexity* as a first-order predicate rather than as an instance of the higher-order function $O(-)$.

We approached this by defining a simple imperative language and an interpreter for that language within ACL2. The interpreter defines a traditional operational semantics for programs within the language, using a clock argument to guarantee termination. We prove functional properties of our programs using the interpreter semantics in the standard way. The interpreter also keeps a count of computational steps as execution proceeds. Then we prove for specific programs in our language that the count of steps taken during execution of the program is bounded (in the big-O sense) by another specific function. As an example, we prove that a program in our language that implements binary search on a sorted list of length n, is $O(\log_2(n))$.[2]

The paper is structured as follows. The following section cites some related work. Section 3 describes our simple imperative language and its interpreter semantics. In Section 4, we outline our approach to proving simultaneously the functional correctness of a program and its computational complexity. This is illustrated by the proof of a binary search algorithm on a sorted list. We prove that the algorithm has complexity of $O(\log_2(n))$, where n is the length of the list. Finally, in Section 5 we draw some conclusions and point to potential extensions of this work.

2 Related Work

The notion of embedding a language within the Boyer-Moore family of logics, of which ACL2 is the latest instantiation, has a very long history[5] that includes modeling software[20], hardware[6], and systems[2]. See the ACL2 homepage publications list for a selection.[3] We are unaware of any work with ACL2 specifically aimed at modeling the asymptotic complexity of algorithms, though Boyer and Moore certainly did careful analysis of the complexity of algorithms they developed such as their fast string search algorithm[3] and fast majority vote algorithm[4].

However, research on modeling and proving asymptotic complexity results has been done using other automated formal reasoning systems. That work encompasses three distinct area of research: expressing within a mechanized formal logic big-O and related notions,[4] extracting complexity results automatically from programs, and proving complexity results about specific algorithms/programs.

Exemplifying the first approarch is the work of Iqbal et al.[18] who formalized big-O and related notions in the HOL-4 logic and checked proofs of classical properties such as transitivity, symmetry and reflexivity using the HOL-4 prover. They do not appear to have used the definitions in the analysis of any specific programs. Most other efforts in this space have gone beyond merely modeling and manipulating notations to analyzing algorithms or programs. Modeling and verification tools like HOL

[1] We also carried out a proof that linear search has linear complexity and other proofs of the correctness of some specific algorithms coded in our iterative language. These proofs are not described here.

[2] We used a function that approximates $\log_2(n)$ by counting the number of times n can be divided in half, using integer division. One reviewer pointed out that there are at least two definitions of `ilog` in community books. Our definition is off by 1 from these, but otherwise provably equivalent.

[3] `https://www.cs.utexas.edu/users/moore/publications/acl2-papers.html`.

[4] Others notations in this family include Omega (lower bound), Theta (exact bound), and little-O (upper bound excluding exact bound).

and Coq that implement reasoning for higher-order logic allow modeling big-O and related notions, which are inherently higher order, more directly than does ACL2.

Inferring the complexity of algorithms is theoretically undecidable, but possible for many programs. Steps in that direction (for worst-case analysis) were described by Wegbreit[24], Le Métayer[19], and Benzinger[1]. Average-case analysis is somewhat harder since it requires a distribution function for the data and probabilistic reasoning. Hickey and Cohen[15] describe a performance compiler that generates recurrence relations characterizing the average-case complexity of programs. Similar work is described by Flajolet et al.[11]

Amortized analysis aims to model average case behavior over a suite of operations, where some may be considerably more expensive than the average. Hofmann and Jost[16] applied automatic type-based amortized analysis to heap usage of functional programs. Carbonneaux et al.[7] generate Coq proof objects to certify inferred resource bounds. Nipkow and Brinkop[21] have formalized in Isabelle/Hol a framework for the analysis of amortized complexity of functional data structures.

Perhaps closest to our work is that of Guéneau[12, 13]. Using and extending an existing program verification framework, CFML[8], Guénaeu uses Separation Logic with Time Credits to model big-O and its related notions and prove the asymptotic complexity of OCaml programs using the Coq automated reasoning system. A very extensive bibliography on research in modeling and proving asymptotic complexity results with automated reasoning systems is available in Guénau's dissertation[13].

3 A Simple Imperative Programming Language

Our primary goal is to prove big-O complexity results for specific algorithms using ACL2. To do so, we define a simple imperative programming language embedded in the ACL2 logic.

3.1 Expression Sublanguage

$(\text{var } s)$	variable
$(\text{lit} . v)$	literal
$(\text{lit} . (v_0, \ldots v_k))$	list literal
$(== e_1\ e_2)$	equals
$(+ e_1\ e_2)$	addition
$(- e_1\ e_2)$	subtraction
$(* e_1\ e_2)$	multiplication
$(// e_1\ e_2)$	integer division
$(< e_1\ e_2)$	less than
$(<= e_1\ e_2)$	less than or equal
$(> e_1\ e_2)$	greater than
$(>= e_1\ e_2)$	greater than or equal
$(\text{len } e)$	len of a list
$(\text{ind } e_1\ e_2)$	index into a list.

Figure 1: Expression Language

Our language includes an expression sublanguage as shown in Figure 1. Here s denotes a symbol, v either a number or symbol, and e an expression within this sublanguage. The semantics of our expression

sublanguage is defined by an evaluator function `(exeval x status vars)`. Here `x` is the expression to be evaluated, `status` is a boolean indicating whether evaluation should proceed and `vars` is a variable alist associating variable names to values (symbols, numbers, or lists of values). Expression evaluation returns a triple `(status, value, steps)`, where `status` is a boolean indicating whether or not the evaluation raised an error, `value` is the result of the evaluation (assuming no error occurred), and `steps` records the number of steps taken by the evaluation. In Figure 2 we exhibit part of the definition of `exeval`; clauses omitted are similar to those shown.

Note that arguments within any subexpression are "type checked" to ensure that evaluation will succeed. Thus, arithmetic and relational functions are applied only to numbers; `len` is applied only to lists; etc. It would be straightforward to relax some of the restrictions, e.g., comparing symbols alphabetically. Also, evaluation is strict in error detection; an error at any level propogates to the top. The presumption is that we only care about the result and the complexity of the computation if it terminates normally, without error.

The number of steps assessed for any evaluation is somewhat arbitrary. For example, we can perform the following computation:

```
ACL2 !>(exeval '(+ (var x) (* (var y) (lit . 10)))    ; expression
               t                                       ; status
               '((x . 7) (y . 5)))                     ; alist
(T 57 5)
```

The resulting triple indicates that the evaluation proceeded without error and returned a value of 57 in 5 steps (1 each to evaluate two variables and one literal, and 1 each for the multiplication and addition). We could easily adjust the numbers of steps assessed for each operation, e.g., make multiplication more expensive than addition. Such changes would not affect the complexity of any programs, as long as these operations remained $O(1)$. We also assume that both indexing into a list (`ind`) and computing the length of a list (`len`) are $O(1)$ operations. Again, that could easily be adjusted, though treating `len` as an $O(n)$ operation, as it might be in Lisp, could change the big-O complexity of programs using it.

3.2 Statement Sublanguage

(skip)	no-op
(assign v e)	assign expression e to variable v
(return e)	assign expression e to variable `'return`
(if-else e s_1 s_2)	if expression e, do statement s_1, else do s_2
(while e s)	while expression e is true, do statement s
(seq s_1 s_2)	do statement s_1, then s_2

Our simple imperative language contains the six types of statements shown above. We provide an operational semantics for this language in the form of an interpreter: `(run stmt status vars steps count)` Here `stmt` is a statement to execute, `status` a symbol indicating whether the computation should continue, `vars` a variable alist, `steps` a tally of the steps so far, and `count` our "clock" argument to guarantee that execution terminates. Function `run` returns a triple `(status, vars, steps)`. Unlike the expression sublanguage, here `status` is not a boolean, but a symbol. A value of `'OK` indicates that execution should continue; `'RETURNED` indicates that a result was assigned to variable `result`; any other

```
(defun exeval (x status vars)
  (if (not status)
      ; Execution proceeds only if status is T
      (exeval-error)
    ; No legal expression is an atom
    (if (atom x)
        (exeval-error)
      (case (operator x)
            ; Variables
            (var (if (definedp (param1 x) vars)
                     (mv t (lookup (param1 x) vars) 1)
                   (exeval-error)))
            ; Literals
            (lit (if (valuep (cdr x))
                     (mv t (cdr x) 1)
                   (exeval-error)))
            ; Addition
            (+ (mv-let (stat1 val1 steps1)
                       (exeval (param1 x) status vars)
                       (mv-let (stat2 val2 steps2)
                               (exeval (param2 x) status vars)
                               (if (and stat1 stat2
                                        (acl2-numberp val1)
                                        (acl2-numberp val2))
                                   (mv t (+ val1 val2)
                                       (+ 1 steps1 steps2))
                                 (exeval-error)))))
               ...

            ; Index into a list (ind i lst)
            (ind (mv-let (stat1 val1 steps1)
                         (exeval (param1 x) status vars)
                         (mv-let (stat2 val2 steps2)
                                 (exeval (param2 x) status vars)
                                 (if (and stat1 stat2
                                          (natp val1)
                                          (listp val2)
                                          (< val1 (len val2)))
                                     (mv t (nth val1 val2)
                                         (+ 1 steps1 steps2))
                                   (exeval-error)))))
            (otherwise (exeval-error))))))
```

Figure 2: Part of the Definition of exeval

value signals an error, typically 'ERROR or 'TIMED-OUT (indicated that count was insufficient for the recursive depth of the call tree). Note that it is possible in ACL2 to define an interpreter for this language without the clock argument. Several approaches are described in [10].

The definition of the interpreter is in Figure 3. Here, run-skip, run-return and run-assignment are non-recursive functions which evaluate skip, return and assign statements, respectively. These are shown in Figure 4. These don't require the status argument since they're never called unless status is 'OK; they don't require count since they are non-recursive.

As with expression evaluation, all arguments are "type checked" and error checking is strict. All of our big-O results assume that execution of the program terminates without error. Again, the counting of steps is somewhat arbitrary, but the choices seem to us to be defensible.[5] For example, running the statement:

```
(if-else test true-branch false-branch)
```

requires one plus the number of steps to evaluate the test plus the number of steps to execute true-branch or false-branch, as appropriate. These easily could be adjusted.

3.3 Programs in the Language

Our simple language is almost certainly Turing complete, though we haven't proved that. We define programs in the language as ACL2 literals. To see how it works, consider the following Python program that computes binary search.

```
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
```

In Figure 5 is the version of this program in our simple imperative language.[6] It's pretty easy to see that this is a straightforward translation of the Python program. This translation is likely not difficult to automate, though we haven't tried to do so. Performing any sort of optimizations automatically would be more challenging.

We can run our program with concrete data by invoking the interpreter on this constant with appropriate arguments. For example,

[5] We note, for example, that similar analysis by Guénau, et al.[13] assesses one time unit for calling a function or entering a loop body, and zero units for other operations.

[6] Here (seqn x y ... w z) is an abbreviation for (seq x (seq y ... (seq w z))...).

```
(defun run (stmt status vars steps count)
  (declare (xargs :measure (two-nats-measure count (acl2-count stmt))))
  (if (not (okp status))
      (mv status vars steps)
    (if (zp count)
        (mv 'timed-out vars steps)
      (case (operator stmt)
            (skip   (run-skip stmt vars steps))
            (assign (run-assignment stmt vars steps))
            (return (run-return stmt vars steps))
            (seq    (mv-let (new-stat new-vars steps1)
                            (run (param1 stmt) status vars steps count)
                            (run (param2 stmt) new-stat
                                 new-vars steps1 count)))
            (if-else (mv-let (eval-stat eval-val eval-steps)
                             (exeval (param1 stmt) t vars)
                             (if (not eval-stat)
                                 (run-error vars)
                               (if eval-val
                                   (run (param2 stmt) status vars
                                        (+ 1 steps eval-steps) count)
                                 (run (param3 stmt) status vars
                                      (+ 1 steps eval-steps) count)))))
            (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))))))
```

Figure 3: The Interpreter for our Imperative Language

```
(defun run-skip (stmt vars steps)
  ; Statement is (skip)
  (declare (ignore stmt))
  (mv 'ok vars steps))

(defun run-assignment (stmt vars steps)
  ; Statement has form (assign var expression)
  (mv-let (eval-stat val eval-steps)
          (exeval (param2 stmt) t vars)
          ; lhs must be a variable
          (if (and (varp (param1 stmt))
                   eval-stat)
              (mv 'ok
                  (store (cadr (cadr stmt)) val vars)
                  (+ 1 eval-steps steps))
            (run-error vars))))

(defun run-return (stmt vars steps)
  ; Statement has form (return expression)
  (mv-let (eval-stat val eval-steps)
          (exeval (param1 stmt) t vars)
          (if eval-stat
              (mv 'returned
                  (store 'result val vars)
                  (+ 1 eval-steps steps))
            (run-error vars))))
```

Figure 4: Run, Assign, and Return

```
(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)))))
```

Figure 5: Binary Search in Our Imperative Language

```
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)
```

This shows that the computation returned after 77 steps with an updated variable alist, including the index/answer in variable RESULT. Note that several var/value pairs were added to the alist during execution, corresponding to the procedure's locals and the result value. Below is an example using variables in the alist:

```
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)
```

In this case, RESULT contains -1, indicating that the search failed after 91 steps. Finally, here's the same computation, but with an insufficient clock value:

```
ACL2 !>(run (binarysearch '(var key) '(var lst))
            'OK '((key . 4) (lst . (0 1 3 5 7 9 10))) 0 1)
(TIMED-OUT ((KEY . 4)
            (LST 0 1 3 5 7 9 10)
            (LOW . 0)
            (HIGH . 2)
            (MID . 3))
           33)
```

Our proofs about big-O properties of functions assume that the computation does not raise an error or time out.

4 Proving Correctness and big-O

Our proofs do more than verifying the big-O behavior of the program. We simultaneously prove the functional correctness of the program. This is not particularly novel, but assures that the program for which we are establishing complexity results is also functionally correct. We describe this aspect of our proof effort below in Section 4.1. The more novel aspect of our work is verifying the big-O behavior of programs. This is described in Section 4.2.

4.1 Functional Correctness

We characterize the functional behavior of a program by defining a recursive functions that closely mimics the execution of our imperative program. For example, for the proof of our binary search program, we defined a function `(recursiveBS key lst)` that behaves "in the same way" as our iterative version. This is illustrated in Figure 6. Notice that the computation carries along the local parameters of the Python computation as well as a count of the number of recursive calls. The locals are needed to show in the inductive proof that the recursive and non-recursive versions coincide, since these variables are all stored in the iterative state. Our proof shows that, in each step of the execution, the values of variables stored in the state are exactly those computed by our recursive version. The count of recursive calls is needed because the time complexity of the computation is ultimately a function of the number of recursive calls made.

```
(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.
  (declare (xargs :measure (nfix (1+ (- high low)))))
  (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)
  ;; This is the recursive version of binary search
  (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)))
```

Figure 6: Recursive Equivalent of our Binary Search Program

As an example, we prove that 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)))))
```

This characterizes explicitly what state is computed by execution of our iterative program; the values in the resulting variable alist are precisely those computed by its recursive counterpart. Notice that the number of steps taken is a function of the computed number of recursive calls.

Finally, we define a second, simpler recursive version `recursiveBS2`, which omits the local variables, and prove that the two recursive versions return the same "result." By transitivity, the iterative version and the simpler recursive version return the same result. Finally, we can see that our iterative program actually computes binary search by examining our "simpler" recursive function, shown here:

```
(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))))
```

Our iterative function computes binary search if this recursive version does. Hence, it remains to show that this recursive function actually searches a list. We proved the following theorem:

```
(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))))))
```

4.2 Computational Complexity

Since the interpreter for our simple imperative language counts execution steps, it should be straightforward to characterize the big-O behavior of the program in terms of the step count. While that's true in theory, it proved to be rather difficult in practice. In this section, we illustrate the process of illustrating how we prove that the count of the number of steps is bounded by our big-O bounding function—in the case of binary search, $\log_2$.

We defined an integer approximation of $\log_2(n)$; Assuming n is a positive integer, $\log_2(n)$ is approximately how often you can halve n before you reach 0. This suggests the following function:

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

This was the definition we used in proofs; notice it is well-suited for reasoning about binary search.[7]

Recall our definition of big-O from Section 1. Rather than try to define the intrinsically higher-order function $O(-)$, we define an instance of it, *logarithmic complexity*, appropriate for complexity proofs about our binary search function. In Figure 7 is our rendering of this in ACL2. `function-logarithmic1` defines the following predicate. Suppose our program (which is a quoted constant) is run on the interpreter with an `'OK` initial status, variable alist, zero previous steps, and an adequate clock. Then the number of steps taken has the appropriate relation to the two positive integer constants c and n_0. Here, parameter `log-of` specifies of which structure's size we're taking the log. `function-logarithmic2` merely adds the existential quantification with respect to variables c and n_0.

Finally, we need to prove that our specific program satisfies this predicate. This is the content of lemma `binarysearch-logarithmic-lemma` shown in Figure 8. This lemma asserts the following: if we assume that

1. variable `'key` is associated with a number `keyval` in our variable alist;
2. variable `'lst` is associated with a sorted list `lstval` of numbers in our variable alist; and
3. the computation does not time-out,

then our binary search function is bounded above, in the appropriate big-O sense, by `(log2 (len lstval))`. The proof of this requires supplying specific values of c and n_0; we supplied $c = 51; n_0 = 26$, though many other values would have worked as well. We have performed proofs showing that linear search is linear in the length of the input list and similar results for several other programs not described here.

4.3 Subtleties of this Approach

Because we carefully count every step in the execution of our program, we obtain a very fine-grained characterization of the program's temporal behavior. This may be useful for purposes other than asymptotic complexity. However, it means that the counting is exquisitely sensitive to how the program is written. For example, our Python implementation of binary search contains the following if statement.

[7] Note that the base case probably should have been `(or (zp n) (equal n 1))`. Without that the value is off by one for $n > 1$, but doesn't affect any of the complexity results. With that change, our version is equivalent to the definition of `ilog` in at least two of the community books.

```
(defun-sk function-logarithmic1 (program log-of c n0 vars count)
  ;; This says that program (which is just a literal) can be run
  ;; against the variable-alist vars with count.  It says that
  ;; the number of steps taken to run the program are logarithmic
  ;; in the size of parameter log-of.  Params c and n0 are the two
  ;; variables of the definition of asymptotic complexity.
  (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))))
```

Figure 7: Definition of Function Logarithmic

```
(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))))
```

Figure 8: Binary Search Logarithmic

```
if key == lst[mid]:
    return mid
elif key < lst[mid]:
    high = mid - 1
else:
    low = mid + 1
```

In this version, assuming that `key` is not found initially, the switch to upper half or lower half take exactly the same number of steps. Assume instead that this were coded (as it was initially) as follows:

```
if key < lst[mid]:
    high = mid - 1
elif key == lst[mid]:
    return mid
else:
    low = mid + 1
```

In this case, selecting the upper half of the list takes more steps than selecting the lower half, since there's an additional test to evaluate. This means that the overall step count is dependent on the *sequence* of choices made rather than the *number* of choices made, as in the prior version. This adds complexity to the overall proof, even though the programs are functionally identical. This isn't too surprising since the exact timing of any program depends on how it's written. But in this case and many others, a difference orthogonal to the big-O behavior substantially complicates the proofs.

A related issue is whether or not the code is optimized. Consider the following Python code:

```
while test:
   x = 0
   ...
```

Any sensible compiler would move the assignment to `x` out of the loop. The step counts could vary significantly depending on whether or not an optimization is performed, and could alter the big-O behavior of a program. We reasoned about the program as written without regard to optimization. It will be unsurprising that properties of programs—both functional and complexity-related—depend on software tools such as compilers.[2] However, these sorts of subtleties make our proofs more fragile and significantly more labor intensive than we would like.

5 Conclusions and Future Work

We have developed techniques for proving big-O properties of imperative programs using ACL2. This involves embedding a simple imperative language in ACL2 with semantics defined with a standard clocked interpreter. We use the interpreter semantics to prove the functional correctness of our code using standard techniques. Our interpreter also returns a count of the number of steps in the execution of the program. We use this count to prove that the number of steps is bounded by a certain function in the big-O sense. We illustrated this with a proof that a certain program implements binary search on a sorted list. We show that the program is $O(\log_2(n))$, where n is the length of the list. Because we analyze both the functional behavior of a program and its step count concurrently, we derive strong assurance that the program is both correct and is a member of the expected big-O class.

However, reasoning about both functional and complexity results imposes a pretty significant overhead if the primary goal is a big-O result. It might be possible to bound the program behavior without carefully counting each step. For binary search, for example, we might prove that the recursive algorithm can't make more than $\log_2(n)$ recursive calls on a list of length n. ACL2 should be an excellent tool for carrying out such a proof. However, if the goal is to prove complexity with respect to iterative implementions, we would still like to verify that the recursive version computes the same results as our iterative version.

Also, our current approach is highly sensitive to the way in which the program is written and to what optimizations have been performed. Overall, we have found our approach to proving big-O properties of programs to be rather fragile and proof intensive. We hope to find approaches which require less effort and alleviate some of these issues.

As noted by one of the anonymous reviewers, *clock functions* are often used in ACL2 total correctness proofs relating to what Ray and Moore[22] call "operationally modeled programs." Such clock functions specify the number of steps required to reach a halting state in programs for "specialized architectures and machine codes."[22] The reviewer suggested taking the "clock function used for functional correctness and then prov[ing] properties of that clock function." We take this suggestion to mean that, rather than developing a new imperative language and proving properties about programs as we have done, to reason in terms of the clock function for some of the several architectures or languages already defined in ACL2. There's no obvious reason to believe that wouldn't work. However, we're not convinced, as the reviewer suggests that "the method would have been applicable for any operational semantics defined in ACL2 and [wouldn't] require tweaking and writing a different evaluator." Unless the method were defined in generality, it would still be specific to some particular architecture or language, such as Moore's Java model[20]. It might be possible to define a *generic* language interpreter and prove complexity properties for programs in that language, but that seems to us more of the same.

We consider the research described in this paper to be preliminary. In particular, it would be great to find more robust and automated techniques for finding values for the existentially quantified variables c and n_0 from the Big-O definitions. Another potentially productive step would be constructing tools to aid in the process of upper-bounding the number of steps of a computation. The approach described in this paper is labor intensive. We regard this work as a proof of principle that ACL2 can be used in developing proofs of computational complexity. But there's likely an easier way.

References

[1] Ralph Benzinger (2004): *Automated Higher-order Complexity Analysis*. Theoretical Computer Science 318(1-2), pp. 79–103, doi:10.1016/j.tcs.2003.10.022.

[2] William R. Bevier, Warren Hunt, Jr., J Strother Moore & William Young (1989): *An Approach to Systems Verification*. Journal of Automated Reasoning 5(4), pp. 411–428, doi:10.1007/BF00243131.

[3] Robert S. Boyer & J Strother Moore (1977): *A Fast String Searching Algorithm*. Communications of the ACM 20(10), pp. 762–772, doi:10.1145/359842.359859.

[4] Robert S. Boyer & J Strother Moore (1991): *MJRTY: A Fast Majority Vote Algorithm*. In Robert S. Boyer, editor: Automated Reasoning: Essays in Honor of Woody Bledsoe, Kluwer Academic Publishers, Dordrecht, The Netherlands, pp. 105–117, doi:10.1007/978-94-011-3488-0_5.

[5] Robert S. Boyer & J Strother Moore (1996): *Mechanized Formal Reasoning about Programs and Computing Machines*. In R. Veroff, editor: Automated Reasoning and Its Applications: Essays in Honor of Larry Wos, MIT Press, Boston, pp. 141–176, doi:10.1.1.27.8666.

[6] Bishop Brock, Matt Kaufmann & J Strother Moore (1996): *ACL2 Theorems about Commercial Microprocessors*. In: *Formal Methods in Computer-Aided Design (FMCAD'96)*, Springer-Verlag, pp. 275–293, doi:10.1007/BFb0031816.

[7] Q. Carbonneaux, J. Hoffmann, T.W. Reps & Z. Shao (2017): *Automated Resource Analysis with Coq Proof Objects*. In: *Computer Aided Verification (CAV 2017)*, Springer-Verlag, pp. 64–85, doi:10.1007/978-3-319-22102-1_9.

[8] A. Charguéraud (2019): *The CFML Tool and Library*. `https://www.charglueraud.org/softs/cfml/`.

[9] A. Charguéraud & F. Pottier (2019): *Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits*. *Journal of Automated Reasoning* 62(3), pp. 331–365, doi:10.1007/s10817-017-9431-7.

[10] John Cowles, David Greve & William Young (2007): *The While Language Challenge: First Progress*. In: *Proceedings of the Seventh International Workshop on the ACL2 Theorem Prover and its Applications*.

[11] Philippe Flajolet, Bruno Salvy & Paul Zimmermann (1991): *Automatic Average-Case Analysis of Algorithms*. *Theoretical Computer Science* 79, pp. 37–109, doi:10.1016/0304-3975(91)90145-R.

[12] A. Guéneau, A. Charguéraud & F. Pottier (2018): *A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification*. In A. Ahmed, editor: *Programming Languages and Systems (ESOP 2018)*, Springer, New York, doi:10.1007/978-3-319-89884-1_19.

[13] Armaël Guéneau (2020): *Mechanical Verification of the Correctness and Asymptotic Complexity of Programs: The Right Answer at the Right Time*. Ph.D. thesis, University of Paris.

[14] Juris Hartmanis & Richard E. Stearns (1965): *On the computational complexity of algorithms*. *Transactions of the American Mathematical Society* 117, pp. 285–306, doi:10.1090/S0002-9947-1965-0170805-7.

[15] T. Hickey & J. Cohen (1988): *Automated Program Analysis*. *Journal of the ACM* 35(1), pp. 185–220, doi:10.1145/99370.99381.

[16] M. Hofmann & S. Jost (2003): *Static Prediction of Heap Space Usage for First-Order Functional Programs*. In: *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, ACM, pp. 185–197, doi:10.1.1.175.6871.

[17] Warren Hunt, Matt Kaufmann, J Strother Moore & Anna Slobodova (2017): *Verified Trustworthy Software Systems*, chapter Industrial Hardware and Software Verification with ACL2. Philosophical Transactions A, vol. 374, Royal Society Publishing, doi:10.1098/rsta.2015.0399.

[18] N. Iqbal, O. Hasan, U. Siddique & F. Awwad (2019): *Formalization of Asymptotic Notations in HOL-4*. In: *4th International Conference on Computer and Commumication Systems*, IEEE, pp. 383–387, doi:10.1007/978-3-030-78409-6_5.

[19] Daniel Le Métayer (1988): *ACE: an automatic complexity evaluator*. *ACM Transactions on Programming Languages and Systems* 10(2), pp. 248–266, doi:10.1145/42190.42347.

[20] J Strother Moore (1999): *Correct System Design—Recent Insights and Advances*, chapter Proving Theorems about Java-like Byte Code, pp. 139–162. LNCS: State of the Art Survey, vol. 1710, Springer-Verlag, doi:10.1007/3-540-48092-7_7.

[21] T. Nipkow & H. Brinkop (2019): *Amortized Complexity Verified*. *Journal of Automated Reasoning* 62, pp. 367–391, doi:10.1007/s10817-018-9459-3.

[22] S. Ray & J S. Moore (2004): *Proof Styles in Operational Semantics*. In A. J. Hu & A. K. Martin, editors: *Proceedings of the 5th International Conference on Formal Methods in Computer-aided Design (FMCAD 2004): LNCS 3312*, Springer-Verlag, New York, pp. 67–81, doi:10.1007/978-3-540-30494-4_6.

[23] Michael Sipser (2006): *Introduction to the Theory of Computation (Second Edition)*. Thompson Course Technology, Boston.

[24] Ben Wegbreit (1975): *Mechanical Program Analysis*. *Communications of the ACM* 18(9), pp. 528–539, doi:10.1145/361002.361016.