Using MBE to Speed a Verified Graph Pathfinder

David Greve and Matthew Wilding

Rockwell Collins Advanced Technology Center
Cedar Rapids, IA 52498 USA

{dagreve, mmwildin}@rockwellcollins.com

Abstract

The experimental feature `MBE` (for "must be equal") allows ACL2 users to introduce two versions of the body for a function, one for reasoning and one for execution [2]. The user must show that the two function bodies are equal when the function arguments satisfy the function guards. We demonstrate that `MBE` allows us to overcome an ACL2 logic weakness identified in an example presented at the 2nd ACL2 Workshop [4].

1 Background

J Moore demonstrates an ACL2 program in [3] that finds paths through a graph. The pathfinding algorithm has a time complexity that is linear in the number of edges in the graph since it marks visited nodes and does not investigate paths that include nodes that have already been visited. Moore's implementation, however, because it uses list access and update functions to implement graph operations, has quadratic time complexity. The `stobj` feature of ACL2 allows programmers to code efficient imperative-style datastructure operations that have simple, functional semantics [1]. Matt Wilding reimplements Moore's program in [4] using stobjs so that the basic graph datastructure operations — marking a node, looking up the edges emanating from a node, etc. — are constant-time.

An issue identified in [4] is the need in ACL2 to add complexity to some programs in order to prove termination. Sometimes this additional complexity seems unnecessary. For example, the main function that implements the search program in [4] is listed below.

```
(defun linear-find-next-step-st (c b st)
  (declare (xargs :stobjs st  :measure (measure-st c st)
                  :guard (and (graphp-st st) (bounded-natp b (maxnode))
                              (numberlistp c (maxnode)))
                  :verify-guards nil))
  (if (endp c) st
    (let ((cur (coerce-node (car c)))
          (temp (NUMBER-UNMARKED st)))
      (cond
       ((equal (marksi cur st) 1)
        (linear-find-next-step-st (cdr c) b st))
       ((equal cur b)
        (let ((st (setstatus 0 st)))
          (setstack (myrev (cons (car c) (stack st))) st)))
       (t (let ((st (setmarksi cur 1 st)))
            (let ((st (setstack (cons (car c) (stack st)) st)))
              (let ((st (linear-find-next-step-st (gi cur st) b st)))
                (if (or (<= temp (NUMBER-UNMARKED st))  ; always nil
                        (equal (status st) 0))
                    st
                  (let ((st (setstack (cdr (stack st)) st)))
                    (linear-find-next-step-st (cdr c) b st))))))))))
```

This function and its verification is described in detail in [4]. However, note instances of `number-unmarked` in the body of `linear-find-next-step-st`, which counts the number of unmarked nodes in a graph. It can be proved using ACL2 that `linear-find-next-step-st` never increases the number of unmarked nodes, so a test that this is so as part of the body of `linear-find-next-step-st` is in a very real way extraneous. However, without this test we are unable to prove termination of the function as required in ACL2.

Of course, for efficiency reasons we would like to use a version of this function without the irrelevant checks. The function `linear-find-next-step-st-fast` is introduced in [4] without proving the justifying termination.

```
(defun linear-find-next-step-st-fast (c b st)
  (declare (xargs :stobjs st
                  :measure (measure-st c st)
                  :guard (and (graphp-st st) (bounded-natp b (maxnode))
                              (numberlistp c (maxnode)))))
  (if (endp c) st
    (cond
     ((equal (marksi (car c) st) 1)
      (linear-find-next-step-st-fast (cdr c) b st))
     ((equal (car c) b)
      (let ((st (setstatus 0 st)))
        (setstack (myrev (cons b (stack st))) st)))
     (t (let ((st (setmarksi (car c) 1 st)))
          (let ((st (setstack (cons (car c) (stack st)) st)))
            (let ((st (linear-find-next-step-st-fast (gi (car c) st) b st)))
              (if (equal (status st) 0)
                  st
                (let ((st (setstack (cdr (stack st)) st)))
                  (linear-find-next-step-st-fast (cdr c) b st))))))))))
```

We believe that it is not possible to justify the termination of this function within the logic of ACL2 despite the fact that it is possible to prove that the "extraneous" test of `linear-find-next-step-st` can be dispensed with when the guards are met [4]. Of course, this additional test execution of the function.

2 Using `MBE`

In January 2003, Matt Kaufmann asked the authors whether an experimental ACL2 feature called `MBE` (which stands for "must be equal") would overcome the weakness encountered in [4]. `MBE` allows the introduction of "executable" versions of functions whose equivalence to the "logical" versions can be justified under the assumption that the function argument types are consistent with their specified guards. This challenge led us to determine whether `MBE` can be used to implement a pathfinder program that is verified using ACL2 and is as fast as `linear-find-next-step-st-fast` with no unproved assumptions.

The term `(mbe :logic logic-code :exec exec-code)` is logically equivalent to `logic-code`. However, when executed in raw Lisp (which ACL2 uses when a function's guards are verified) it evaluates to `exec-code`. This implementation is sound because the guard proof obligations of each function that contains this term includes the requirement that `logic-code` is logically equivalent to `exec-code` when the function's guards are met.

We implement the guard pathfinding program — again! — using `MBE`. We use the bodies of the two previously-presented functions, except that the recursive calls are renamed to refer to the new function name, `linear-find-next-step-st-mbe`.

```
(defun linear-find-next-step-st-mbe (c b st)
  (declare (xargs :stobjs st
                  :measure (measure-st c st)
                  :guard (and (graphp-st st)
                              (bounded-natp b (maxnode))
                              (numberlistp c (maxnode)))
                  :verify-guards nil))
  (mbe

    :logic  body of linear-find-next-st with recursive calls renamed

    :exec  body of linear-find-next-st-fast with recursive calls renamed

      ))
```

We are able to guard-check `linear-find-next-step-st-mbe`, which includes the obligation that the `:logic` and `:exec` arguments are equal when the function arguments satisfy the guards. This proof follows directly from the proofs in [4], as does the proof of correctness of the algorithm itself. This book certifies in experimental ACL2 2.8 in about 2 minutes, and accompanies this paper in the workshop proceedings. Also associated with this paper is a file containing functions that support running the pathfinding program.

3 Conclusion

We believe that we have implemented this algorithm so that it is about as fast as it can be. A million-edge complete graph is searched exhaustively in under a second on our Sun workstations, which is comparable to the times reported in [4] for the most-optimized (and unjustified) implementation. We have verified the correctness of this implementation wholly using the theorem-prover-supported logic of ACL2 in a manner similar to what was done in part using an unchecked informal argument in [4]. This is made possible by the new `MBE` feature.

References

[1] Robert S. Boyer and J Strother Moore. Single-threaded objects in ACL2. *PADL 2002*, 2002.

[2] M. Kaufmann and J Moore. What's new in ACL2. In *Proceedings of the Fourth International Workshop on the ACL2 Theorem Prover and Its Applications*, Boulder, Colorado, July 2003.

[3] J Strother Moore. An exercise in graph theory. In *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[4] Matthew Wilding. Using a single-threaded object to speed a verified graph pathfinder. In *ACL2 Workshop 2000*, November 2000. available as University of Texas Dept. of CS TR 00-29.