Implementing a Cost-Aware Evaluator for ACL2 Expressions

Ruben Gamboa
Department of Computer Science
University of Wyoming
Laramie, WY 82071
ruben@cs.uwyo.edu

John Cowles
Department of Computer Science
University of Wyoming
Laramie, WY 82071
cowles@cs.uwyo.edu

ABSTRACT

One of ACL2's most interesting features is that it is *executable*, so users can run the programs that they verify, and debug them during verification. In fact, the ACL2 implementors have gone well out of their way to make sure ACL2 programs can be executed *efficiently*. Nevertheless, ACL2 does not provide a framework for reasoning about the cost of function invocations. This paper describes how such a framework can be added to ACL2, by using ACL2 macros and supporting code to access the prover state. The approach is illustrated with a cost analysis of red-black tree operations.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms

Evaluators, function cost

Keywords

ACL2 evaluator

1. INTRODUCTION

ACL2 can be thought of as both a theorem prover and a programming language. This is useful for a number of reasons. Since (most) ACL2 terms can be evaluated directly, it is possible to check manually whether a particular instance of a theorem holds, *before* trying to prove that the theorem is actually true. In addition, ACL2 can be used directly as a programming language, e.g., to build hardware simulators [11], leaving open the door to a later verification effort.

As a result, the ACL2 developers have spent a great deal of effort making sure that programs written in ACL2 are efficient. For example, they have added many features to ACL2 designed strictly to improve the efficiency of executable theories, such as

- guards, which justify the direct execution of functions by the underlying "raw" Lisp implementation,
- stobjs, which provide $O(1)$ destructive access to data structures while maintaining an applicative semantics for formal reasoning, and
- mbe, which allows users to provide a different (more efficient) definition of a function than the one used for logical reasoning.

And many researchers have dealt explicitly with this aspect of ACL2. For example, Ruiz Reina and others' recent formalization of a quadratic unification algorithm is significant in large part precisely because the algorithm in question is quadratic [10]. Similarly, Moore demonstrated the correctness of an $O(n)$ time graph pathfidning algorithm [8]. Wilding later showed that the algorithm, while $O(n)$ in essence, was closer to $O(n^2)$ in practice because an ACL2 implementation of the graph data structure had expensive primitive operations. This was sufficient motivation for the development of ACL2 stobjs and mbe. Using these features, Wilding and Greve showed that the ACL2 implmentation did indeed result in an $O(n)$ pathfinding algorithm [12, 5].

It is significant that none of the results above dealt with the efficiency of the programs in a formal way. It is likely that this is due to the lack of a framework for reasoning about function cost in ACL2. In particular, when a function is defined in ACL2, the current ACL2 theory is extended with axioms that describe the behavior of the function (as well as axioms justifying induction and other derived inference rules—see [6, 4] for details. However, no axioms regarding the cost of executing the new functions are introduced.

This stands in contrast with Nqthm, also known as the Boyer-Moore Theorem Prover, where each new function definition implicitly enhanced the (partial) definition of the global function `v&c$`, which could then be used to reason about the cost of evaluating an Nqthm expression containing the new function symbol (and possibly other, previously defined symbols as well.) In addition, `v&c$` served as an evaluator which could be used to reason about the value of such expressions.

Nqthm's `v&c$` turns out to be far more powerful than a simple evaluator, even a cost-aware evaluator, for terms in the Nqthm logic. For example, readers who learned ACL2 with no prior knowledge of Nqthm will be surprised that the

ACL2 '06 Seattle, Washington USA

following definition (named after Betrand Russell) in Nqthm is *valid* [2]:

```
(RUS) = (CAR (V&C$ T '(ADD1 (RUS)) NIL)
```

Read informally, this definition suggests that the value of `RUS` is one more than the value the universal evaluator assigns to the function `RUS`. This *apparent* inconsistency is resolved in Nqthm by axiomatizing the evaluator such that it returns either false, or a pair containing the value and cost of the evaluation. Intuitively, `v&c$` returns the value and cost of well-defined expressions, but false for expressions whose evaluation cost is infinite, such as `RUS` above.

`v&c$` is even more powerful than the example above suggests. Boyer and Moore show how `v&c$` and its cousin `eval$` can be used to define "quantifier" functions, similar to Lisp's `mapcar` [1]. Kunen took this a step further, showing how `v&c$` and `eval$` can be used to build a nonconstructive extension to primitive recursive arithmetic (PRA,) in which it is possible to prove the consistency of PRA [7].

We suspect that the surprising power of `v&c$` is responsible, at least in part, for the ACL2 designers' decision to leave it out of ACL2. Evaluators do fill an important need in ACL2 (as in Nqthm,) because they can be used to justify `:meta` inference rules. To fill this need, ACL2 provides `defevaluator`, which builds a custom evaluator for a specific ACL2 subtheory. The evaluator is added to the ACL2 theory using the regular theory extension mechanisms. In fact, `defevaluator` is a clever macro that creates ACL2 `encapsulate`, `defun`, and `defthm` events needed to introduce the evaluator. This has two practical consequences. First, `defevaluator` avoids the possibility of introducing unsoundness into an ACL2 theory. From the perspective of soundness, it can be seen as nothing more than syntactic sugar for a particular theory extension. Second, `defevaluator`, unlike Nqthm's `v&c$` can not make the theorem prover logically more powerful.

In this paper we present `defeval$`, a framework implemented in the same way as `defevaluator`—i.e., as an ACL2 macro with some supporting code—which produces axioms similar to `v&c$`'s for reasoning about the cost, as well as value, of expressions in ACL2.

2. THE GENERATED THEORY

In this section, we present the basic definitions and theorems that make up a theory constructed with `defeval$`. Like `v&c$`, given an expression and an alist, `defeval$` returns either `nil` (when the expression contains terms that the evaluator does not know how to evaluate) or a pair consisting of the value of the expression and the cost of computing it. The next section will show how this theory can be constructed automatically in ACL2.

First of all, recall that `defeval$` can be used to create an evaluator for a specific theory. This can be done with the following event:

```
(defeval$ evl reverse)
```

This event introduces the function `evl`, which is an evaluator for ACL2 expressions involving `reverse`. The arguments to `evl` include the term to be evaluated and an alist mapping variables to concrete ACL2 terms. For example, after the introduction of this event, the following will be theorems of ACL2:

```
(equal (car  (evl '(reverse x) '((x . (1 2 3)))))
       '(3 2 1))
(equal (cadr (evl '(reverse x) '((x . (1 2 3)))))
       24)
```

Note: These theorems are similar to the ones supported by Nqthm's `v&c$`.

Why should we believe that `(car (evl ...))` returns the value of the expression and that `(cadr (evl ...))` returns its cost? This is where the virtue of using an evaluator to compute cost becomes apparent: It is possible to characterize the behavior of the evaluator in ways that make its output immediately obvious. For example, the `defeval$` introducing `evl` will also introduce the following theorems:

```
(defthm evl-constraint-1
  ; The cost of evaluating a symbol is 0, and its
  ; value is either itself (for constants) or the
  ; lookup in the alist.
  (implies (symbolp term)
           (equal (evl term alist)
                  (if (or (booleanp term)
                          (acl2-numberp term))
                      (list term 0)
                    (list (cdr (assoc-eq term
                                         alist))
                          0)))))

(defthm evl-constraint-2
  ; The cost of evaluating a quoted term is 0, and
  ; the value is the quoted term.
  (implies (equal (car term) 'quote)
           (equal (evl term alist)
                  (list (nth 1 term) 0))))

(defthm evl-constraint-3-reverse
   ; The cost of evaluating (reverse x) is 1, plus
   ; the cost of evaluating x, plus the cost of
   ; evaluating the body of reverse.  The value is
   ; the reverse of the value of x.
   (implies (and (evl-valid-expr term)
                 (equal (car term) 'reverse))
      (equal (evl term alist)
        (list (reverse (car (evl (nth 1 term)
                                 alist)))
              (+ 1
                 (cadr (evl (nth 1 term) alist))
                 (cadr (evl '(if (stringp x)
                                 (coerce
                                  (revappend
                                   (coerce x 'list)
                                   nil)
                                  'string)
                                (revappend x nil))
                            `((x . ,(car
                                     (evl
                                      (nth 1 term)
                                      alist))))
                            )))))))
```

The function `evl-valid-expr` checks to make sure that `term` contains only function symbols that are known to `evl`.

These theorems make it clear that the `car` of `evl` really is the expected value of the expression, and that the `cadr` of `evl` really is computing the cost of executing the function,

where the cost is measured by counting function openings. Note: The generated theory includes many theorems analogous to `evl-constraint-3-reverse`, one for each function used directly or indirectly in the execution of `reverse`, e.g., `if`, `stringp`, `coerce`, `revappend`, etc.

The theorem `evl-constraint-3-reverse` provides a hint to the definition of `evl`. In particular, it should look something like this, where `...body...` denotes the body of the definition of reverse:

```
(defun evl (term alist)
  (if (atom term)
      ...
    (if (equal (car term) 'reverse)
        (let ((arg1 (evl (nth 1 term) alist))
              (body (evl ...body...
                         `((x . ,(car arg1))))))
          (list (car body)
                (+ 1
                   (cadr arg1)
                   (cadr body))))
      ...)))
```

The problem, however, is that `evl` as defined above is not obviously terminating. Consider: There are two recursive invocations of `evl`, one to evaluate the argument to `reverse`, and the other to evaluate the body (which is indicated by `...body...` above.) The first call obviously operates on a "smaller" argument, so that would lead to termination. The second call, however, does not. The body of `reverse` looks "bigger" than the term involving `reverse`, and the `alist` is of little help—e.g., it may not change at all in the recursive call. Aside: ACL2's `defevaluator` macro sidesteps this issue by *not* considering the body of the functions defined. This works in that context, because the evaluator is only concerned with the value of expressions, not their computational cost.

Of course, there really *is* a measure that is reduced by each invocation, namely the canonical measure as defined in [6]. This measure essentially counts the number of steps required to complete the execution of the current term. The proof obligations that ACL2 examines when new functions are introduced justify the fact that this measure is indeed decreased by the act of opening up a function call. However, we found no easy way to use this measure directly to define functions such as `evl`.

Instead, we proceed by defining a resource-bounded evaluator `(evl-step term alist n)` that evaluates the expression `term` up to at most `n` steps. This function obviously terminates, since `n` is decreased in each recursive call. Next, `defeval$` defines a function `(evl-reverse-n term alist)` that returns an estimate (usually larger than necessary) of the value of `n` required for `evl-step` to completely evaluate `term` under `alist`. Finally, these two functions are combined to define `evl`.

3. GENERATING THE THEORY

In the previous section we saw the theory that is developed by `defeval$` to reason about the cost of evaluating a given set of functions. In the following sections we consider how this theory can be generated automatically. First we discuss how we can find all the functions that are used directly or indirectly by the functions we are interested in evaluating; this is done in section 3.1. In section 3.2 we develop the theory of the bounded evaluator, including the definition and the key theorems that are required. Section 3.3 describes how we find upper bounds for the limit argument used by the bounded evaluator to process a given term. Then in section 3.4 we turn our attention to the final evaluator. The macro `defeval$` wraps all this together, and it is described in section 3.5.

3.1 The Supporting Theory

`defeval$` allows the user to specify one or more functions in which the user is interested. For example, to reason about the cost of evaluating expressions in terms of `length` and `member`, a user may issue the following event:

```
(defeval$ evl length member)
```

This is similar to the way an evaluator is defined using ACL2's built-in `defevaluator` function, but there are some key differences. Consider the equivalent definition using `defevaluator`:

```
(defevaluator evl evl-list
  ((length x) (member x y)))
```

First of all, `defevaluator` requires the user to name two functions: `evl` which works on terms and `evl-list` which evaluates *lists* of terms. Second, `defevaluator` requires that the user specify the arity of the functions to be evaluated, as shown above.

The third difference is more important (and serves to explain the previous two differences.) `defevaluator` is concerned only with the value of expressions, so it can use the original functions directly. `defeval$`, on the other hand, is also concerned with the cost of function execution, so it needs to know the *bodies* of the functions. Since it is unfeasible to ask the user to provide the bodies, as well as the arities of the desired functions, `defeval$` has no choice but to access the definitions from the ACL2 state. Given that, `defeval$` can gather all the other necessary information at the same time.

ACL2 stores the required information as properties of the function symbol. You can view all the properties for a symbol with the command `getprops` as follows:

```
(getprops 'reverse 'current-acl2-world (w state))
```

This will list all the information that ACL2 stores about the symbol `reverse`, including its definition, lemmas, etc. We are particularly interested in the following properties:

- **formals:** A list containing the function's formal arguments, e.g., `(X)` for `reverse`.
- **body:** The body of the definition of the function, after expanding macros and applying some minor normalizations (such as propagating `if` expressions upwards.) This property is `nil` for built-in functions. Note: The definition body may have free variables, but only if they are listed in the list of formals.
- **recursivep:** Non-nil if the function is recursive. More precisely, it is a list containing the clique of functions making up a (possibly mutually) recursive definition.
- **induction-machine:** For recursive calls, this holds a list of tests and the associated recursive calls that are

made when each test succeeds. This is created when the function definition is accepted (according to some well-founded measure,) and it used internally by ACL2 to build induction schemes.

- **absolute-event-number:** A (sort of) timestamp that marks where the definition was added to the ACL2 history.

Individual properties can be retrieved using the function `getprop`.

The property `body` is the key to finding the functions used directly or indirectly by each function the user wishes to evaluate. This can be done with a pair of recursive functions—one for terms, the other for list of terms—that traverse down the body of the original functions, finding new functions to consider. The termination of these functions is trivial, in that one eventually runs out of function symbols to consider. Nevertheless, this argument would be hard to carry out in ACL2, so we declare these functions in program mode.

It will be important later that functions be considered in inclusion order. That is, if function `fn1` is used (directly or indirectly) in the definition of function `fn2` (and the functions are not mutually recursive,) then `fn1` should be considered before `fn2`. To ensure this we collect the functions in the order in which they were originally defined, using the property `absolute-event-number` to make the comparisons.

3.2 The Bounded Evaluator

Once all functions have been identified, it is possible to construct the bounded evaluator `(evl-step term alist n)` which returns either a list containing the value and cost of `term` under `alist`, or the value `nil` when the evaluation of `term` requires more than `n` steps. The definition contains boilerplate to deal with the `zp` bound, atomic terms, quoted terms, and if-terms. The remaining entries in the evaluator are taken from the list of functions as follows:

- Terms corresponding to built-in functions are evaluated by evaluating the arguments, then (if their evaluation succeeds) calling the built-in directly. The cost is measured as 1 (for the function opening,) plus the cost of evaluating the arguments.
- Terms corresponding to non-built-in functions are evaluated by evaluating the arguments, then (if their evaluation succeeds) evaluating the body while binding the formal arguments to the result of evaluating the arguments of the term. The cost is measured as 1, plus the cost of evaluating the arguments, plus the cost of evaluating the body.

The remaining generated events prove that `evl-step` has the right properties, namely that when it returns a non-nil value—an event we refer to as "succeeding" in the sequel—it returns the correct value and execution cost of the expression, as suggested by the theorems listed in section 2. The development of this theory is a non-trivial exercise.

An important first step is to show that `evl-step` is monotonic. That is, when `(evl-step term alist n)` succeeds, so will `(evl-step term alist m)` for all values of `m` larger than `n`. Moreover, the value and cost returned by these two invocations of `evl-step` are identical.

The problem with this lemma is that it is hard to apply automatically. Consider a term `(evl-step term alist m)` that we wish to show is not `nil`. According to the monotonicity lemma, all we need is a value of `n`, smaller than `m`, such that `(evl-step term alist n)` but it is not always obvious which value of `n` to try.

Instead, we discovered that terms of the form

```
(evl-step term alist (+ -3 n))
(evl-step term alist (+ -1 n))
```

and so on were ubiquitous in subsequent proofs. What we really needed are lemmas that show that when `(+ -3 n)` is known to be sufficient, so is `(+ -1 n)`. The trick is to find the constant values in these lemmas automatically. For reasons that will become obvious, we only have to consider values from 0 to $-k$, where k is the maximum depth of any given function body, where "depth" is defined using the typical measure on trees. Once the value of k is found, it is straightfoward to generate all the (roughly) $k^2/2$ lemmas involved.

Next we consider lemmas analogous to `evl-constraint-*`, as defined in section 2. Constraints 1 and 2, dealing with atomic and quoted terms, are completely straightforward. So is constraint 3 for if-terms. These cases share a common thread: They are generated with boilerplate in the definition of `eval-step`, so they can be dispensed with trivially, regardless of the functions being evaluated.

The remaining function symbols can be classified into three categories: built-ins, non-recursive, and recursive. The built-ins are also trivial to deal with, since the constraint 3 associated with each built-in is identical to its definition in `evl-step`. Non-recursive functions take only a little more effort. Suppose `f` is a non-recursive function with body `(g ...)`. According to the definition of `evl-step`, the values (not costs) returned by the following terms are identical:

```
(evl-step '(f x1 ... xk) alist n)
(evl-step '(g ...)
          '((x1 . V1) ... (xk . Vk))
          (+ -1 n))
```

Here we assume that `Vi` is the value of `xi` under `alist`. Now, if we know that `evl-step` works correctly for terms based on `g`, we can conclude that the last term returns the value of `(g ...)`, which is precisely equal to `f`, by hypothesis. ACL2 can verify all this automatically, as long as it can take the key step rewriting `(evl-step '(g ...) ...)` to `(g ...)`. This key step, of course, is constraint 3 for the function `g`. Here is where we require that the function symbols identified in section 3.1 be ordered according to introduction via definition. By ordering the functions appropriately, we can be sure that the lemmas showing that `evl-step` works correctly for `g`-terms is proved before the theorem for `f`-terms, where it is needed.

That leaves recursive functions, where the story is understandably more complicated. We illustrate the approach with a running example. Suppose that `f` is a recursively-defined function, and consider an arbitrary term `(f ...)`. As part of evaluating this term, we will be forced to evaluate a term such as `(f (cdr x) ...)`, where `cdr` is some "destructor" function, and all the variables in this term are formals of `f`. It is not easy to see why this latter term should evaluate correctly. Clearly what is needed is some sort of inductive argument, and it turns out that an induction based on the definition of `f` fits the bill.

First a key lemma is required. Suppose we are considering the following term:

```
(evl-step '(f arg1 ... argk) alist n)
```

If this term is not `nil`, then the following term is also not `nil`:

```
(evl-step '(f x1 ... xk)
          '((x1 . V1) ... (xk . Vk))
          n)
```

As before, we assume here that `Vi` is the value of `argi` under `alist`, and that the `xi` are the formal parameters to `f`.

This step means that we need only consider terms of this last form, for all possible values of `Vi`. These terms can be described succintly with the backquote notation as follows:

```
(evl-step '(f x1 ... xk)
          `((x1 . ,x1) ... (xk . ,xk))
          n)
```

Here, `xi` is a *constant* value naming a variable, and `,xi` is the *free variable* denoting any possible value.

Now, suppose the induction scheme suggested by `f` replaces `x1` with `(cdr x1)` as suggested above. Then the induction hypothesis states that the following term is evaluated properly by `evl-step`:

```
(evl-step '(f x1 ... xk)
          `((x1 . ,(cdr x1)) ... (xk . ,xk))
          n)
```

This is, of course, not the term that appears naturally in the proof. In particular, when `evl-step` is opened, it will inevitably lead to the following subterm:

```
(evl-step '(f (cdr x1) ... xk)
          `((x1 . ,x1) ... (xk . ,xk))
          n)
```

What is needed is a lemma showing that these last two invocations of `evl-step` are equivalent. Then all the pieces are in place for an inductive proof of the correctness of `evl-step` for terms involving `f`—using the same induction scheme as suggested by `f`.

What is required is a mechanism to generate these lemmas in sequence. The property `induction-machine` of symbol `f` has just the needed information. The value of this property is a list of terms containing tests and recursive calls in the context of these so-called ruling tests. For example, consider the (normalized) body of binary-append:

```
(if (consp x)
    (cons (car x)
          (binary-append (cdr x) y))
  y)
```

The induction machine for this function contains the following[1]:

- Test: `(endp x)`
  - Call: none
- Test: `(not (endp x))`
  - Call: `(binary-append (cdr x) y)`

In general there may be more than one recursive call for each group of governing tests. For example, consider the function `fringe-aux` defined as follows:

```
(defun fringe-aux (tree fringe)
  (if (consp tree)
      (fringe-aux (car tree)
                  (fringe-aux (cdr tree) fringe))
    (cons tree fringe)))
```

Its induction machine is given by the following:

- Test: `(consp tree)`
  - Call:

    ```
    (fringe-aux (car tree)
                (fringe-aux (cdr tree)
                            fringe))}
    ```
  - Call: `(fringe-aux (cdr tree) fringe)`
- Test: `(not (consp tree))`
  - Call: none

We always generate a lemma for each recursive call, regardless of how many calls exist for a given set of governing tests. In fact, since one of the recursive calls may appear inside another recursive call as in the `fringe-aux` example, we take care to generate the lemmas in the proper order (which happens to be the order opposite to the one in which the calls appear in `induction-machine`.)

Once these lemmas are established, only minor plumbing is required to use them in an induction, as suggested by `f`, to show that `evl-step` correctly evaluates `f`-terms.

3.3 A Sufficient Bound

Once `evl-step` is defined, it becomes necessary to find a value of `n` for each `term` and `alist` such that `(evl-step term alist n)` is not `nil`. Atomic and quoted terms are handled trivially, since any positive (integer) value of `n` is sufficient to evaluate them. For terms involving built-ins, this is fairly straightforward: It is only necessary to find an `n` 1 larger than is necessary to evaluate the arguments. A similar argument takes care of `if`-terms, as it is only necessary to evaluate the condition and one of the then/else subterms.

For the remaining function symbols `f`, we define a function `evl-f-n` with the same arity as `f`, such that the term `(evl-step '(f t1 ... tk) alist n)` is not `nil` whenever

- `n` is a positive integer,
- `n` is large enough that each of the `ti` can be evaluated under `alist` in `n`-1 steps, and
- `n` is at least equal to `(evl-f-n V1 ... Vk)` where each `Vi` is the result of evaluating `ti` under `alist`.

In the remainder of this section we show how we can construct such a function, and how we can automatically generate the proof that this function works as described.

The idea is to build this function by looking carefully at the body of `f`. Let *term* be a term in ACL2 (e.g., the body of `f`.) Then the value $C(term)$ defined as follows is large enough such that `evl-step` will be able to evaluate *term* if given at least $C(term)$ steps:

[1]The reader may notice that the tests and calls in the induction machine are based on the unnormalized body of `binary-append`. That explains why the induction machine mentions `endp` while the normalized body mentions `consp` instead.

- If *term* is an atom or of the form `(quote` u`)`, then $C(term) = 1$
- If *term* is `(if cond then else)`, then $C(term)$ is

  (if *cond*
    (+ 1 (max C(*cond*) C(*then*)))
   (+ 1 (max C(*cond*) C(*else*))))

- If *term* is `(f t1...tk)` where `f` is a built-in function, $C(term)$ is `(+ 1 (max` C`(t1)...`C`(tk)))`
- Otherwise *term* is `(f t1...tk)` for some previously (or recursively) defined function `f`. Then $C(term)$ is `(+ 1 (max` C`(t1)...`C`(tk) (evl-f-n V1...Vk)))` where `Vi` is the value of `ti` under the given `alist`.

The function `evl-f-n` can be defined as $C(body) + 1$, where *body* is the body of `f`. The extra 1 is necessary to open up the definition of `f`.

The proof that `evl-f-n` works as claimed follows from the same induction scheme suggested by `f`. The key lemma is the following: If `n` is large enough that `(evl-step ti alist (- n 1))` succeeds for all `ti`, and `n` is large enough that `(evl-step` *body* `'((t1 . ,V1)...((tk . ,Vk))) (- n 1))` also succeeds, then `evl-step '(f t1...tk) alist n)` must succeed.

A few hints are required to get this theorem through ACL2. In particular, it is necessary to ask ACL2 to `:expand` the `evl-step` terms that appear in the recursive subgoals. Our approach to generate these hints is to build a lemma for each recursive call in the induction machine of `f`. Then, when induction is invoked, all the subgoals generated by induction can be proved automatically. This technique was common in Nqthm, since it was impossible in Nqthm to provide a hint for any subgoal except the topmost goal. An alternative that we have not explored is to use computed hints for this purpose, although we imagine they would work as well.

Implementation Note: When `f` is recursive, `evl-f-n` will also be recursive. In fact, both functions should have the same induction machine. When ACL2 builds the induction machine, it finds governing terms (i.e., the tests that determine when a recursive call is invoked) by traversing only the topmost layer of `if`-terms. Therefore, we are careful in the definition of $C(term)$ to follow the exact same `if`-structure as in the original function. What this means is that we have to move any nested `if` terms up through the `max` and `+` functions above.

3.4 The Final Evaluator

We are now almost ready to construct the final evaluator. First of all, the individual functions `evl-f-n` can be collected into a single function `evl-n` that can process an entire term at once, regardless of its top-most symbol. Since this process requires that we find the `Vi` that corresponds to a particular `ti`, e.g., to find the arguments of a particular `evl-f-n`, we need to be able to evaluate subterms. Therefore, we construct a total evaluator `(evl-aux term alist)` that is identical to the one produced by ACL2's `defevaluator`. It is easy to prove—i.e., no hints are required—that when `evl-step` succeeds, the value it returns is the same computed by `evl-aux`.

We would then like to prove the following theorem:

```
(defthm proposed-evl-n-is-enough
  (implies (and (integerp n)
                (<= (evl-n term alist) n))
           (evl-step term alist n)))
```

However, this proposed theorem is false. The reason is that there are two ways for `evl-term` to return `nil`. One is that the value of `n` is too small to finish the computation of `term`. This should not happen here, given the construction of `evl-n`. But the other way is that `term` includes a function symbol that `evl-n` does not recognize. To get around this problem, we define the function `evl-valid-expr`, which checks whether `term` is defined solely in terms of recognized functions. The correct theorem is as follows:

```
(defthm proposed-evl-n-is-enough
  (implies (and (evl-valid-expr term)
                (integerp n)
                (<= (evl-n term alist) n))
           (evl-step term alist n)))
```

The proof of this theorem requires a hint to force `n` to decrease correctly on the inductive calls—ACL2 users will be familiar with this phenomenon. The function that suggests this hint is also generated automatically, following the same recursive pattern as `evl-n`.

We can now define the desired evaluator:

```
(defun evl (term alist)
  (evl-step term alist (evl-n term alist)))
```

It follows from the properties of `evl-step` that, as long as `term` is valid, `evl` always succeeds.

Moreover, the correctness properties previously proved about `evl-step` transfer easily to `evl`. It is only necessary to provide a hint to ACL2 to `:use` the appropriate instances, rewriting `evl` applied to the arguments and body of a function into the equivalent `evl-step` terms.

3.5 Putting it Together

All the events described above are generated by the macro `defeval$-form` with the following signature:

```
(defeval$-form evfn fns world-name world-alist)
```

The argument `evfn` denotes the desired name for the new evaluator, and `fns` is a list of functions symbols that `evfn` should be able to process. The remaining two arguments are used to access the current ACL2 world, which needs to be inspected by `defeval$-form` (for example, to find the definitions of the functions in `fns`.) Ordinarily, `world-name` should be `'current-acl2-world` and `world-alist` should be `(w state)`. What `defeval$-form` returns is an event that, when executed, will define constrained and executable versions of the function `evl`. We automatically execute the generated event by sending it to ACL2's `ld` command. This is done automatically by the macro `defeval$`:

```
(defmacro defeval$ (evfn &rest fns)
  (list 'ld
        (list 'list
              (list 'defeval$-form
                    (list 'quote evfn)
                    (list 'quote fns)
                    (list 'quote
                          'current-acl2-world)
                    (list 'w 'state)))))
```

This allows the user to isssue a simple command, such as the following:

```
(defeval$ evl reverse)
```

The macro `deveval$` will generate the necessary events and submit them to ACL2 for verification.

4. AN EXTENDED EXAMPLE

In this section we illustrate the use of `defeval$` by presenting a formal, albeit brief, analysis of the correctness and completeness of the insert and lookup operations on red-black trees. In section 4.1 we present a brief overview of red-black trees and their formalization in ACL2. The correctness of this formalization is addressed in section 4.2. We address the time complexity of red-black trees in section 4.3. Finally, in section 4.4 we assess how well `defeval$` performed in this case study.

4.1 Red-Black Trees

Red-black trees are binary search trees with the following additional restrictions [3]:

- Each node in the tree is colored either red or black.
- Every (NIL) leaf is black.
- The children of a red node are both black.
- Every path from a node to one of its descendant leaves contains the same number of black nodes.

It is clear from the restrictions above that red-black trees are nearly balanced: The maximum length of a branch from the root to a leaf is no more than twice the minimum length of such a path. It follows that tree operations that depend on the height of the tree, such as lookup, are $O(\log n)$, where n is the number of nodes in the tree.

We model red-black trees in ACL2 using Bishop Brock's `defstructure` command:

```
(defstructure rb key value color left right
              (:options :slot-writers))
```

This defines the constructor function `(rb key value color left right)` that creates a new node in a tree, as well as updater functions, such as `set-rb-left`.

Lookup operations with red-black trees are no different than lookup operations with ordinary binary search trees, and they can be defined in ACL2 as follows:

```
(defun rb-lookup (key tree)
  (cond ((not (rb-p tree))
         nil)
        ((equal key (rb-key tree))
         (rb-value tree))
        ((lexorder key (rb-key tree))
         (rb-lookup key (rb-left tree)))
        (t
         (rb-lookup key (rb-right tree)))))
```

Notice that we use the function `lexorder` to compare nodes; `lexorder` is a total ordering of all the objects in the ACL2 universe.

The insert operation is more complex than lookup, because it needs to preserve the key properties that make up a red-black tree. This can be done efficiently. The key observation is that the red-black tree properties can be done by adding a local "rotation" step after the regular binary search tree insertion step:

```
(defun rb-insert-aux (key value tree)
  (cond ((not (rb-p tree))
         (rb key value 'red nil nil))
        ((equal key (rb-key tree))
         (set-rb-value value tree))
        ((lexorder key (rb-key tree))
         (rb-balance-left
          (set-rb-left
           (rb-insert-aux key
                          value
                          (rb-left tree))
           tree)))
        (t
         (rb-balance-right
          (set-rb-right
           (rb-insert-aux key
                          value
                          (rb-right tree))
           tree)))))
```

Note the functions `rb-balance-left` and `rb-balance-right` called immediately after the node is inserted into the appropriate tree. Replacing these functions with the identity function results in ordinary binary tree insertion.

The rotations above are strictly local operations (although possibly $O(\log n)$ rotations will be required for each insertion.) In an imperative language, these rotations are accomplished by a swapping a small number of pointers. In ACL2 we can achieve the same effect with a trick of Okasaki's [9]. The idea is to reconstruct the "top" two levels of the red-black tree. There are four cases to consider, two each for a left and a right rotation. The following case is representative:

```
(defmacro rb-rotate (a1 a2 a3 t1 t2 t3 t4)
  `(rb (rb-key ,a2) (rb-value ,a2) 'red
       (rb (rb-key ,a1) (rb-value ,a1)
           'black ,t1 ,t2)
       (rb (rb-key ,a3) (rb-value ,a3)
           'black ,t3 ,t4)))

(defun rb-balance-left-case-1 (tree)
  (rb-rotate (rb-left (rb-left tree))
             (rb-left tree)
             tree
             (rb-left (rb-left (rb-left tree)))
             (rb-right (rb-left (rb-left tree)))
             (rb-right (rb-left tree))
             (rb-right tree)))
```

Note that the rotations are intended to preserve the order in the tree, so it is assumed that the following properties hold

- $k(a_1) \prec k(a_2) \prec k(a_3)$
- $max(t_1) \prec k(a_1) \prec min(t_2)$
- $max(t_2) \prec k(a_2) \prec min(t_3)$
- $max(t_3) \prec k(a_3) \prec min(t_4)$
- t_i is ordered

Here $k(a)$ denotes the key of node a and $min(t)$ ($max(t)$) denotes the minimum (maximum) key of tree t.

The remaining three cases are also based on the `rb-rotate` macro. It is easy to see that this function simply rearranges the top part of the tree while preserving the ordered nature of the tree. It is less obvious that this sequence of rotations does in fact result in a red-black tree.

4.2 Correctness

To prove the correctness of the algorithm requires us to show that insertions into a red-black tree result in a search tree that also preserves the red-black conditions as outlined above. The search tree property can be defined as follows:

```
(defun ordered-tree-p (tree)
  (if (rb-p tree)
      (and (ordered-tree-p (rb-left tree))
           (ordered-tree-p (rb-right tree))
           (or (not (rb-p (rb-left tree)))
               (strict-lexorder
                (tree-max (rb-left tree))
                (rb-key tree)))
           (or (not (rb-p (rb-right tree)))
               (strict-lexorder
                (rb-key tree)
                (tree-min (rb-right tree)))))
    t))
```

As its name suggests, the function `strict-lexorder` is an anti-reflexive version of `lexorder`. It is straightforward to prove via induction that the lookup function works correctly (e.g., as compared by a naive lookup that searches all nodes in the tree) when a tree is ordered, as defined above.

A similar straightforward induction will show that inserting a node into an ordered tree results in an ordered tree, as long as we ignore the effect of the rotations. To include the effect of the rotations, we proceed as follows. First, it is easy to see that rotating a tree does not change its minimum or maximum element. This means that if a child of a parent node is rotated, this does not affect the ordered property of the parent tree—provided we can show that the child subtree itself is ordered.

So what remains to be shown is that if a tree is ordered, the rotation of this tree is also ordered. Recall the definition of the macro `rb-rotate`:

```
(defmacro rb-rotate (a1 a2 a3 t1 t2 t3 t4)
  `(rb (rb-key ,a2) (rb-value ,a2) 'red
       (rb (rb-key ,a1) (rb-value ,a1)
           'black ,t1 ,t2)
       (rb (rb-key ,a3) (rb-value ,a3)
           'black ,t3 ,t4)))
```

Earlier we said that `rb-rotate` assumes that the arguments meet the following properties:

- $k(a_1) \prec k(a_2) \prec k(a_3)$
- $max(t_1) \prec k(a_1) \prec min(t_2)$
- $max(t_2) \prec k(a_2) \prec min(t_3)$
- $max(t_3) \prec k(a_3) \prec min(t_4)$
- t_i is ordered

What we did was to show formally that when these conditions hold the result of `rb-rotate` is an ordered tree. The reader can easily verify this claim.

Once this lemma is verified, it remains only to show (via case analysis) that the calls to `rb-rotate` generated by each of the four possible rotations satisfies the conditions above.

Similar arguments can be used to prove that the tree retains the other properties of a red-black tree.

4.3 Complexity

We now use `defeval$` to address the complexity of the red-black tree operations in two steps. First, we count the number of function openings required to insert or lookup elements in a red-black tree in terms of the maximum height of the tree. Later, we relate the height of the tree to the number of elements in the tree.

One of the attractive aspects of ACL2 is that it supports direct execution of functions defined in it. So we can execute the cost-aware evaluator defined by `defeval$` to get an (informal) idea of the execution cost. Our approach here was to consider red-black trees of size ranging from 10 to 500 nodes—the trees were generated by inserting the first n natural numbers into the empty tree. Then we used `defeval$` to count the number of function openings required to insert one more value into the tree. The value chosen is smaller than all the values in the tree, so this measures the cost of inserting the value into the leftmost branch. The results can be seen in figure 1, which suggests that the actual execution cost is logarithmic, as expected.

To prove that the cost of insertion is indeed logarithmic, we examine the cost of each of the primitive functions in turn. The constructor function `rb`, and the accessor and updater functions, e.g., `rb-key` and `set-rb-key`, are particularly simple. These functions are defined as fixed combinations of `cons`, `car`, and `cdr`, so they have a fixed computation cost. For example, the cost of `rb-key` is equal to $3+C_1$, where C_1 is the cost of evaluating the (first and only) argument to `rb-key`.

The cost of each individual rotation, e.g., the function `rb-balance-left-case-1` is also easy to consider. Since each rotation is implemented as a fixed computation tree, its cost is also constant. For `rb-balance-left-case-1`, this cost is equal to 136, a constant that was first discovered using the executable version of the evaluator.

Things get murky when the rotations are combined. For example, the following is the definition of the left rotation:

```
(defun rb-balance-left (tree)
  (cond ((or (not (rb-p tree))
             (equal (rb-color tree) 'red)
             (not (rb-p (rb-left tree)))
             (not (equal (rb-color (rb-left tree))
                         'red)))
         tree)
        ((and (rb-p (rb-left (rb-left tree)))
              (equal (rb-color (rb-left
                                (rb-left tree)))
                     'red))
         (rb-balance-left-case-1 tree))
        ((and (rb-p (rb-right (rb-left tree)))
              (equal (rb-color (rb-right
                                (rb-left tree)))
                     'red))
         (rb-balance-left-case-2 tree))
```

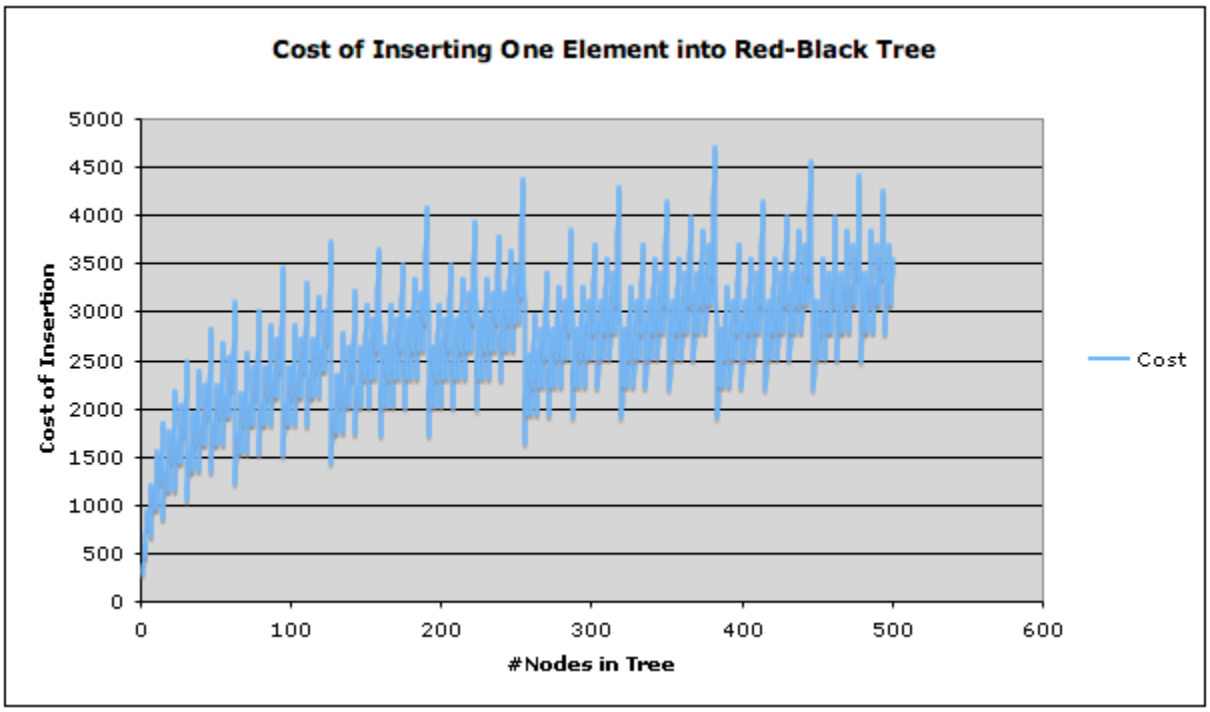

Figure 1: Cost of Inserting to a Red-Black Tree

```
    (t tree)))
```

The problem is that the cost of this function depends not only on the cost of the individual rotations, but also on the cost of deciding which rotation, if any, to apply. And the cost of making that decision depends on the properties of the specific tree in question. So we can only find an upper bound for this function.

A similar situation occurs with `rb-insert-aux`. Our approach was to define a recurrence relation that overestimates the cost of `rb-insert-aux`. To discover the right recurrence relation, we only had to consider the cost of evaluating expressions such as the following:

```
(or (not (rb-p tree))
    (equal (rb-color tree) 'red)
    (not (rb-p (rb-left tree)))
    (not (equal (rb-color (rb-left tree)) 'red)))
```

The only surprise was the cost of `lexorder`, which we expected to be close to 1 but turned out to be 54 for symbols. To keep the numbers manageable, we decided to define our recurrence relation in terms of the cost of evaluating these expressions, rather than as hard bounds. The reason for this is that it made it easier for us to remember where the values came from; i.e., the cost of `lexorder` is more mnemonic than the magic constant 54.

The recurrence relation for `rb-insert-aux` is easily solved, and it turns out to be linear in the number of nodes examined, which can be no more than the maximum height of the tree. At this point, we are finished with the first part of the analysis. We know that the time needed to perform an insertion is linear on the height of the tree.

The second part of the analysis—a connection between the size of the tree and its maximum height—follows from the properties of red-black trees. In particular, since all paths from the root to a leaf have the same number of black nodes and there can be at most one red node between any two black nodes in a path, it follows easily that the maximum height of the tree is at most twice the minimum height of the tree. Since the minimum height of the tree can be used to find a lower bound for the number of nodes in the tree, we have that the maximum height of the tree also binds the number of nodes. Turning things around, if the number of nodes is less than 2^n, the tree can be of height at most $2n$.

The two results can now be combined to show that insert is an $O(\log n)$ operation. Similar results can also be developed (and more easily) for lookup.

4.4 Assessment

A few problems with our `defeval$` became apparent during the formalization. The most serious one is that while reasoning about the cost of complex functions, it is impossible to stop the defined evaluator from considering the cost of simpler functions. In particular, when reasoning about the correctness of a complex function, it is common practice to establish the correctness of the sub-functions on which it depends and then to disable the definitions of these sub-functions. The lemmas about the sub-functions can then be used while establishing the correctness of the more complex functions.

But we have been unable to make such a compositional approach—disabling the constraints that define the behavior of the evaluator—work with the cost-aware evaluator. I.e., we can either enable or disable the evaluator, but we can not enable it for terms with topmost symbol `rb-balance-left` and disable for terms with topmost symbol `rb-left`. Instead, we have had to rely on detailed intervention during the proof effort. We are hopeful that this can be simplified in the future by the strategic use of computed hints.

The other problem that we discovered is that often the execution cost, as measured by the number of function invocations, is dominated by the cost of ACL2 built-ins. For example, `endp` is defined in terms of `atom`, which is defined in terms of `not`—itself defined in terms of `if`—and `consp`. So reasoning about the cost of functions that use `endp` as part of a recursive definition (i.e., almost all functions that operate on lists) means that we have to address this chain of function openings. This detracts, obviously, from the important focus, which is the cost of the function that operates on the list. We were surprised, when reasoning about the cost of red-black tree operations, by the high cost of the operations. Eventually, we traced some of this cost to the function `lexorder`. But since the cost of the `lexorder` is bundled with the cost of the red-black operations, we found it difficult to use the executable version of the evaluator to guess the right values for the cost of intermediate expressions—and code inspection was next to worthless. That is why we turned to using ACL2's proof checker, so we could manipulate the terms directly and see where the cost was coming from. We hope to simplify this in a forthcoming version of `defeval$` by letting the user specify the cost of some functions directly, e.g., letting the user assign unit cost to `lexorder`, as if it were a true built-in.

5. CURRENT STATUS AND FUTURE DIRECTIONS

Currently, `defeval$` is defined as an ACL2 function in the file `defeval.lisp`. To use it, it is only necessary to load this file into your ACL2 session, which will define the macro `defeval$` and its supporting functions.

Our biggest challenge has been ensuring that the generated events are proved cleanly by ACL2. Particularly challenging has been the treatment of recursive functions. What we have today works for many ACL2 terms, but we are sure that it will fail for some functions whose recursive definition eludes our code generation.

Our intent is to continue experimenting with `defeval$`, in the expectation that it will be included in future releases of ACL2. We believe that the events generated by `defeval$` are always theorems, although we have less confidence that they are always provable by ACL2. Moreover, the time to prove these events can be significant. Given these facts, we anticipate that the final version of `defeval$` will simply add the generated events into the current ACL2 theory, without verifying them via `defthm` first. We are looking forward to feedback in this issue.

We are also experimenting with some different cost functions. The first involves ACL2's special treatment of stobjs and `mbe`. For stobjs, we are detecting the use of primitives, such as `update-field`, and treating as built-ins, instead of treating them as defined functions. This reflects the facts that, while they have logical definitions, ACL2 executes them using Common Lisp's destructive operations which are $O(1)$. A similar issue is raised by `mbe`. Unfortunately, determining the executable version of the function is non-trivial. We have not found a satisfactory way to resolve this yet, and we look forward to suggestions from the community.

Finally, we observe that we can modify the sequence of events generated by `defeval$` to compute costs according to different criteria. Note in particular that the cost function plays absolutely no role in the development of the functions `evl-f-n` and `evl-n`. Therefore, once we decide on a suitable cost function, the same lemmas and theorems necessary to show termination of `evl-step` will go through.

This allows us to consider some interesting possibilities. We can leave it up to the user to specify a cost function that determines the cost of a particular function invocation. This allows the user to override the cost of built-in functions, for example. This can be used to measure different aspects of function invocation. One cost function can be used to count the number of `cons` evaluations, giving an idea of the space complexity of a given execution. Another alternative is to consider only the maximum cost of argument evaluation, not their sum. This corresponds to an (ideal) parallel execution. We are currently exploring this possibility in the context of powerlists.

6. REFERENCES

[1] R. S. Boyer and J S. Moore. The addition of bounded quantification and partial functions to a computational logic and its theorem prover. *Journal of Automated Reasoning*, 4, 1988.

[2] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, San Diego, 1988.

[3] T. H. Corman, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, chapter 32. McGraw-Hill, New York, 1990.

[4] R. Gamboa and J. Cowles. Theory extension in ACL2(r), 2006 (under review).

[5] D. Greve and M. Wilding. Using MBE to speed a verified graph pathfinder. In *Proceedings of the Fourth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-2003)*, 2003.

[6] M. Kaufmann and J S. Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[7] K. Kunen. Nonconstructive computational mathematics. *Journal of Automated Reasoning*, 21, 1998.

[8] J S. Moore. An exercise in graph theory. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 5. Kluwer Academic Press, 2000.

[9] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[10] J. L. R. Reina, J. A. Alonso, M. J. Hidalgo, and F. Martín. A formally verified quadratic unification algorithm. In *Proceedings of the Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2004)*, 2004.

[11] M. Wilding. Single-threaded processor models: Enabling proof and high-speed execution. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 8. Kluwer Academic Press, 2000.

[12] M. Wilding. Using a single-threaded object to speed a verified graph pathfinder. In *Proceedings of the Second International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-2000)*, 2000.