Function Memoization and Unique Object Representation for ACL2 Functions

Robert S. Boyer
Department of Computer Sciences
The University of Texas
Austin, Texas USA
boyer@cs.utexas.edu

Warren A. Hunt, Jr.
Department of Computer Sciences
The University of Texas
Austin, Texas USA
hunt@cs.utexas.edu

ABSTRACT

We have developed an extension of ACL2 that includes the implementation of hash-based association lists and function memoization; this makes some algorithms execute more quickly. This extension, enabled partially by the implementation of *Hash*-`CONS`, represents ACL2 data objects in a canonical way, thus the comparison of any two such objects can be determined without the cost of descending through their CONS structures. A restricted set of ACL2 user-defined functions may be memoized; the underlying implementation may conditionally retain the values of such function calls so that if a repeated function application is requested, a previously computed value may instead be returned. We have defined a fast association list access and update functions using hash tables. We provide a file reader that identifies and eliminates duplicate representations of repeated objects, and a file printer that produces output with no duplicate subexpressions.

General Terms

Function Memoization, Hash CONS, ACL2, Lisp

Keywords

ACL2 Workshop, Hash CONSing

1. INTRODUCTION

We have developed a canonical representation for ACL2 data objects and a function memoization mechanism to facilitate reuse of previously computed results. We include procedures to read and print ACL2 expressions in such a way that repetition of some ACL2 objects is eliminated, thereby permitting a kind of on-the-fly file compression. Our implementation does not alter the semantics of ACL2 except to add a handful of definitions.

The executable portion of the ACL2 logic is a first-order logic of recursive functions [8, 9]. Data objects of the ACL2 logic include complex rationals, symbols, characters, and strings. In addition, a pair of any two objects may be created with the `CONS` function, thus there are five distinct data types. ACL2 functions require ACL2 objects as arguments and return ACL2 objects. The functional nature of ACL2 logic permits the canonical representation of ACL2 objects; that is, we may represent two logically equal objects by using only one copy.

The ACL2 logic is a formalization of a superset of a subset of Common Lisp, and the basic axioms of ACL2 provide definitions for almost 200 Common Lisp functions. The ACL2 theorem-proving system uses Common Lisp itself to provide the underlying representation for ACL2 data objects and function definitions. In the implementation of the ACL2 logic, ACL2 data objects are represented by Common Lisp objects of the same type, and the ACL2 pairing (`CONS`) operation is internally implemented by the Common Lisp `CONS` procedure. In Common Lisp, `CONS` is guaranteed to provide a new pair, distinct from any previously created pair. We have defined a new ACL2 function `HONS` that is logically identical to the ACL2 `CONS` function, but whose implementation usually reuses an existing pair if its components are identical to the components of an existing pair. A record of ACL2 `HONS` objects is kept, and when an ACL2 function calls `HONS` we search for an existing identical pair before allocating a new pair; this operation been called Hash CONSing.

It appears that Hash CONSing was first conceived by A. P. Ershov [5] in 1957, to speed up the recognition of common subexpressions. Ershov showed how to collapse trees to minimal DAGs by traversing trees bottom up, and he used hashing to eliminate the re-evaluation of common subexpressions. Later, Eiichi Goto [6] implemented a Lisp system with a built-in Hash CONS operation: his **h**-CONS cells were rewrite protected and free of duplicate copies, and Goto used this Hash CONS operation to facilitate the implementation of a symbolic algebra system he developed.

Memoizing functions also has a long history. In 1967,

ACL2 '06 Seattle, Washington USA

Donald Michie proposed using memoized functions to improve the performance of machine learning [10, 11]. Rote learning was improved by a learning function not forgetting what it had previously learned; this information was stored as a memoized function values.

The use of Hash CONSing has appeared many times. For instance, Henry Baker using Hash CONS to improve the rewriting performance [1] of the Boyer (and Moore) rewriting benchmark [7]. Baker used both Hash CONSing and function memoization improve the speed of the Takeuchi function [2], exactly in the spirit of our implementation, but without the automated, system-wide integration we report here.

Our implementation permits memoization of user-defined ACL2 functions. During execution a user may enable or disable function memoization on an individual function basis, may clear memoization tables, or even may keep a stack of memoization tables. This facility takes advantage of our implementation where we keep one copy of each distinct ACL2 data object. Due to the functional nature of ACL2, it is sufficient to have at most one copy of any data structure; thus, a user may arrange to keep data canonicalized. Our implementation extends to the entire ACL2 system the benefits enjoyed by BDDs citeBryant1986: canonicalization, memoization, and fast equality check.

We have defined various algorithms using these features, and we have observed, in some cases, substantial performance increases. For instance, we have implemented unordered set intersection and union operations that operate in time roughly linear in the size of the sets. Without using arrays, we defined a canonical representation for Boolean functions using ACL2 objects. We have investigated the performance of rewriting and tree consensus algorithms to good effect, and we believe function memoization offers interesting opportunities to simplify algorithm definition while simultaneously providing performance improvements.

Our presentation is split into pieces, which we start by providing an example. We present our logical extensions to ACL2 by exhibiting a collection of ACL2 definitions. For a few of these definitions, we have defined Common Lisp implementations that provide for unique object representation and function memoization. We recommend that the reader try to keep the logical definitions separate from their underlying Common Lisp implementation. We present several algorithms designed to take advantage of these features, and we compare their performance with existing ACL2 algorithms. Our preliminary results suggest that substantial performance improvement opportunities exist.

2. EXAMPLES

We begin with an example that demonstrates the utility of function memoization. This definition of the Fibonacci function exhibits an exponential increase in its runtime as its input argument value increases.

```
(defun fib (x)
  (declare (xargs :guard (natp x)))
  (mbe
   :logic
   (cond ((zp x) 0)
         ((= x 1) 1)
         (t (+ (fib (- x 1)) (fib (- x 2)))))
   :exec
   (if (< x 2)
       x
     (+ (fib (- x 1)) (fib (- x 2))))))
```

By using the ACL2 function `TIME$`, we measure the time to execute a call to the `FIB` function. Below is the output collected running OpenMCL (a Common Lisp implementation) on a 1 GHz Apple G4 PowerBook. The first call is made before memoization is enabled. Note that we have eliminated some of the output provided by the `TIME$` function when used with OpenMCL. `MEMOIZE` is actually an ACL2 macro that expands to the actual ACL2 function used to identify a function for memoization (see Section6). This function also accepts a well-formed term that must be true for the system to memoize a call of the memoized function; to ensure that the arguments supplied to term are safe, we perform a check that if the guards to the memoized function are satisifed, then the arguments to this term will execute without error. Thus, each time we memoize a function a guard check is performed and ACL2 prints a guard-related message.

```
ACL2 !>(time$ (fib 40))
(FIB 40) took 16.072 seconds to run.
102334155

ACL2 !>(memoize 'fib)
The guard for FIB trivially implies
the guard conjecture for T.
Summary
Form:  CHECK-CONDITION-GUARD
Rules: NIL
Warnings:  None
Time: 0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
FIB

ACL2 !>(time$ (fib 40))
(FIB 40) took 0 milliseconds (0.000 seconds) to run.
102334155

ACL2 !>(time$ (fib 100))
(FIB 100) took 0 milliseconds (0.000 seconds) to run.
354224848179261915075
ACL2 !>(unmemoize 'fib)
FIB
ACL2 !>
```

We see that once the function `FIB` is identified as a function for which function calls should be memoized, that the execution times are substantially reduced. Finally, we can prevent `FIB` from being further memoized; in fact, `UNMEMOIZE` eliminates the previously memoized values.

This contrived example is just that, contrived. A more sensible implementation would make provisions for recording previously computed values or computing with a linear-time, tail-recursive algorithm.

```
(defun f1 (fx-1 fx n-more)
  (declare (xargs :guard (and (natp fx-1)
                              (natp fx)
                              (natp n-more))))
  (if (zp n-more)
      fx
    (f1 fx (+ fx-1 fx) (1- n-more))))

(defun fib2 (x)
  (declare (xargs :guard (natp x)))
  (if (zp x)
      x
    (f1 0 1 (1- x))))
```

```
(defthm fib2-is-fib
  (implies (natp x)
           (equal (fib2 x)
                  (fib x))))
```

We can prove that function FIB2 is equal to FIB, thus we can maintain a simple recursive definition while still providing an implementation that is roughly linear in time and space to the input argument. Thus, function memoization, by itself, is nothing more than a convenient dynamic programming mechanism; however, when we combine canonical data representation with memoization, we sometimes observe substantial performance improvements.

We next consider another somewhat contrived example, but this example exhibits the cooperation of function memoization with canonical object representation. Consider the second and third function definitions below; these two functions are provably equivalent.

```
(defun my-len (x)
  (declare (xargs :guard t))
  (if (atom x)
      0
    (1+ (my-len (cdr x)))))

(defun make-list-of-numbers (n)
  (declare (xargs :guard (natp n)))
  (if (zp n)
      nil
    (hons (my-len (make-list-of-numbers (1- n)))
          (make-list-of-numbers (1- n)))))

(defun make-list-of-numbers2 (n)
  (declare (xargs :guard (natp n)))
  (if (zp n)
      nil
    (let ((rest (make-list-of-numbers2 (1- n))))
      (hons (my-len rest) rest))))

(defthm make-list-of-number-functions-are-the-same
  (equal (make-list-of-numbers  n)
         (make-list-of-numbers2 n)))

(defmacro bvl (variable new-value)
  (declare (xargs :guard t))
  `(mv-let
    (erp result state)
    (assign ,variable ,new-value)
    (declare (ignore result))
    (value (not erp))))
```

Our measurements show function MAKE-LIST-OF-NUMBERS to have an exponential execution time cost. The equivalent function MAKE-LIST-OF-NUMBERS2 only computes the remainder of the list once, making the execution linear. Just the execution of (MAKE-LIST-OF-NUMBERS 22) requires several seconds, while the execution of (MAKE-LIST-OF-NUMBERS2 20000) finishes in 13 seconds. The macro BVL prevents the evaulation result from being printed, but still associates the result with name in the VARIABLE argument.

```
ACL2 !>(time$ (make-list-of-numbers 22))
(MAKE-LIST-OF-NUMBERS 22) took 3.580 seconds to run.
(21 20 19 18 17 16
    15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0)

ACL2 !>(bvl list-20000
            (time$ (make-list-of-numbers2 20000)))
(MAKE-LIST-OF-NUMBERS2 20000) took 13.141 seconds to run.
 T
```

```
ACL2 !>(memoize 'make-list-of-numbers)
The guard for MAKE-LIST-OF-NUMBERS trivially implies
the guard conjecture for T.
Summary
Form:  CHECK-CONDITION-GUARD
Rules: NIL
Warnings:  None
Time: 0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
MAKE-LIST-OF-NUMBERS

ACL2 !>(bvl list3-20000
            (time$ (make-list-of-numbers 20000)))
(MAKE-LIST-OF-NUMBERS 20000) took 13.251 seconds to run.
 T

ACL2 !>(unmemoize 'make-list-of-numbers)
MAKE-LIST-OF-NUMBERS
```

By memoizing the function MAKE-LIST-OF-NUMBERS, we see execution time comparable to that of MAKE-LIST-OF-NUMBERS2. If we memoize both MY-LEN and MAKE-LIST-OF-NUMBERS, then the execution time of MAKE-LIST-OF-NUMBERS is further reduced. Before we memoize MY-LEN, the length of the list so far created is measured again and again.

```
ACL2 !>(memoize 'make-list-of-numbers)
The guard for MAKE-LIST-OF-NUMBERS trivially implies
the guard conjecture for T.

Summary
Form:  CHECK-CONDITION-GUARD
Rules: NIL
Warnings:  None
Time: 0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
MAKE-LIST-OF-NUMBERS

ACL2 !>(memoize 'my-len)
The guard for MY-LEN trivially implies
the guard conjecture for T.

Summary
Form:  CHECK-CONDITION-GUARD
Rules: NIL
Warnings:  None
Time: 0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
MY-LEN
ACL2 !>(bvl list3-20000
            (time$ (make-list-of-numbers 20000)))
(MAKE-LIST-OF-NUMBERS 20000) took 0.153 seconds to run.
 T
```

Thus, when defining a function, it may possible to use memoization to observe whether further development effort might provide better execution performance. In our final example, by memoizing both MAKE-LIST-OF-NUMBERS and MY-LEN we have reduced the runtime to less than that of MAKE-LIST-OF-NUMBERS2.

3. ACL2 FUNCTION DEFINITIONS

This section concerns only the logical definition of functions using the ACL2 definition mechanism. In Section 4, we will discuss the Common Lisp implementation of several of these functions, but their underlying Common Lisp implementation is not necessary to understand these functions. Functions that have special Common Lisp implementations are marked as having `an under the hood implementation`.

Our first three definitions are logically trivial.

```
(defun hons (x y)
```

```
  (declare (xargs :guard t))
  ;; Has an "under the hood" implementation.
  (cons x y))

(defun hons-equal (x y)
  (declare (xargs :guard t))
  ;; Has an "under the hood" implementation.
  (equal x y))

(defun hons-copy (x)
  (declare (xargs :guard t))
  ;; Has an "under the hood" implementation.
  x)
```

For our collections of functions that operate on associations lists (alists), we have defined the predicate `HONS-ALISTP` that recognized well-formed alists. We note that any association list recognized by `ALISTP` is also recognized by `HONS-ALISTP`, as it allows any atom to terminate the end of an association list.

```
(defun hons-alistp (x)
  (declare (xargs :guard t))
  (if (atom x)
      t
    (and (consp (car x))
         (hons-alistp (cdr x)))))

(defthm alistp-implies-hons-alistp
  (implies (alistp l)
           (hons-alistp l)))
```

We access key-value pairs in association list `Y` with key `X` using the function `ASSOC-HONS-EQUAL`. We define another function `HONS-GET`, which is just defined to call `ASSOC-HONS-EQUAL`; its purpose involves its underlying definition.

```
(defun assoc-hons-equal (x y)
  (declare (xargs :guard (hons-alistp y)))
  (cond ((atom y) nil)
        ((hons-equal x (car (car y))) (car y))
        (t (assoc-hons-equal x (cdr y)))))

(defun hons-get (x l)
  (declare (xargs :guard (hons-alistp l)))
  ;; Has an "under the hood" implementation.
  (assoc-hons-equal x l))
```

To update an association list recognized by the predicate `HONS-ALISTP`, we define two semantically equivalent functions, `HONS-ACONS` and `HONS-ACONS!`, that add a new key-value pair to associations list `L`.

```
(defun hons-acons (key value l)
  (declare (xargs :guard t))
  ;; Has an "under the hood" implementation.  Note:
  ;; under the hood, the key will be made unique,
  ;; but the alist and its top-level pairs are built
  ;; with CONS, not HONS.
  (cons (cons (hons-copy key) value) l))

(defun hons-acons! (key value l)
  (declare (xargs :guard t))
  ;; Has an "under the hood" implementation.  The
  ;; (HONS KEY VALUE) below will cause VALUE to have
  ;; a unique representation, which, for large
  ;; structures, may require a substantial amount of
  ;; work.
  (hons (hons (hons-copy key) value) l))
```

We define a read object function `HONS-READ-OBJECT` that has a semantics identical to `READ-OBJECT`, but that uses `HONS` instead of `CONS` to construct all pairs.

```
(defun hons-read-object (channel state)
  (declare
   (xargs :stobjs state
          :guard
          (and (state-p state)
               (symbolp channel)
               (open-input-channel-p
                channel :object state))))
  ;; Has an "under the hood" implementation.
  (read-object channel state))
```

Finally, we define two functions that remove key-value pairs that have duplicate keys in association lists recognized by `HONS-ALISTP`; their internal implementation provides better compression speed than just defining these functions as written below.

```
(defun hons-shrink-alist! (alst ans)
  (declare
   (xargs :guard (and (hons-alistp alst)
                      (hons-alistp ans))))
  ;; Has an "under the hood" implementation.
  (cond
   ((atom alst) ans)
   (t (let ((p (hons-get (car (car alst)) ans)))
        (cond
         (p (hons-shrink-alist! (cdr alst) ans))
         (t (hons-shrink-alist!
             (cdr alst)
             (hons-acons! (car (car alst))
                          (cdr (car alst))
                          ans))))))))

(defun hons-shrink-alist (alst ans)
  (declare
   (xargs :guard (and (hons-alistp alst)
                      (hons-alistp ans))))
  ;; Has an "under the hood" implementation.
  (cond
   ((atom alst) ans)
   (t (let ((p (hons-get (car (car alst)) ans)))
        (cond
         (p (hons-shrink-alist (cdr alst) ans))
         (t (hons-shrink-alist
             (cdr alst)
             (hons-acons (car (car alst))
                         (cdr (car alst))
                         ans))))))))
```

We have defined a number of other functions that have ACL2 system-level definitions, but these functions allow a user to influence the operation of the underlying definitions. We present these definitions later (Section 5).

4. HONS SYSTEM IMPLEMENTATION

The implementation of the HONS system involves several facets: canonical representation of ACL2 data, function memoization, and the use of Lisp hash tables to improve the performance of association list operations. We discuss each of these in turn, and we mention some subtle interrelationships. Although it is not necessary to understand the discussion in this section, it may permit some users to better use the `HONS` system. This section may be confusing for some ACL2 users as it makes references to Lisp implementation features.

The mechanical implementation of the ACL2 system is actually written as a Lisp program that is layered on top of a Common Lisp system implementation. ACL2 data constants are implemented with their corresponding counterparts in Common Lisp; that is, ACL2 CONS pairs, strings,

characters, numbers, and symbols, are implemented with their specific Common Lisp counterparts. This choice permits a number of ACL2 primitive functions to be implemented with their corresponding Common Lisp functions, but, there are indeed differences. ACL2 is a logic, and as such, it does not specify anything to do with physical storage or execution performance. When ACL2 is used, the knowledgeable user can write functions to facilitate the reuse of some previously computed values. For instance, the previously introduced function `MAKE-LIST-OF-NUMBERS2` is more efficient than the equivalent function `MAKE-LIST-OF-NUMBERS`; the `LET` form requires only one recursive call instead of the two calls prescribed in `MAKE-LIST-OF-NUMBERS`.

There are three mechanisms that are provided by the `HONS` system: hash `CONS`, function memoization, and *fast* association list operations. The function memoization mechanism takes advantage of the canonical representation of data objects provided by the `HONS` operation as does the fast association list operation mechanism. Each time a HONS pair is created, its arguments are themselves converted, if necessary, to uniquely represented objects.

The ACL2 universe is recursively closed under the `CONS` pairing operation and the atoms. Hash `CONS` is logically identical to `CONS`, but a set of tables are used to record each HONS pair. In fact, our implementation provides a stack of such tables; thus a new environment of `HONS` tables can be requested, used, and then released. When a `HONS` pair is requested, the implementation checks, in the current set of tables, whether the requested pair already exists. If not, a new pair is created and a record of that pair is made; otherwise, a reference to the previously created pair is returned. Thus, any data object created with a `HONS` operation has a unique representation, as does every subcomponent. We also arrange for strings to have a unique representation – only one copy of each different string is kept – and when any previously unseen string is an argument to a `HONS` operation, we add the string to our unique table of strings. A system-wide benefit of having a canonical representation for data is that equality comparisons between any two data objects can be done in constant time.

The definition of the `HONS` in no way changes the operation of `CONS` – a `HONS` is implemented with a `CONS`. When asked to create a `HONS`, our implementation checks to see if there is a `CONS` with the same `CAR` and `CDR` as the `HONS` being requested. Thus, the only difference between a `HONS` and a `CONS` is a notation in some invisible (to the ACL2 logic) tables where we record what `CONS` elements are also `HONS` elements. Since a `HONS` is nothing but a `CONS`, the operation of `CAR` and `CDR` is unchanged. In fact, we have attempted to design our `HONS` implementation so that at any time it is safe to clear the table identifying which `CONS` elements are also considered `HONS` elements.

User-defined functions with defined and verified guards can be memoized. When a function is memoized, a user-supplied condition restricts the domain when memoization is attempted. When the condition is satisfied, memoization is attempted (assuming that memoization for the function is presently enabled); otherwise, the function is just evaluated. Each memoized function has a hash table that is used to keep track of a unique list of function arguments paired with their values. If appropriate, for each function the corresponding table is checked to see if a previous call with exactly the same arguments already exists in the table: if so, then the associated value is returned; if not, then the function is evaluated and a new key-value pair is added to the table of memoized values so that some future call will benefit from the memoization. With ACL2 user functions memoization can be dynamically enabled and disabled. There is an ACL2 function that clears a specific memoization table. And finally, just as with the `HONS` table, a stack of these function memoization tables are maintained; that is, it is possible to memoize a function, use it a bit, set the memoized values aside, start a new table, use it, and then return to the original table.

A part of our `HONS` system provides a fast lookup operation for association lists. When a pair is added to an association list using the function `HONS-ACONS` or `HONS-ACONS!`, our system also records the key-value pair in an associated hash table. So long as a user only used these two functions when placing key-value pairs on an association list, the key-value pairs in corresponding hash table will be synchronized with the key-value pairs in the association list. Later, if `HONS-GET` is used to lookup a key, then instead of performing a linear search of the association list we consult the associated hash table. If a user does not strictly follow this discipline, then a linear search may be required. In some sense, these association lists are much like ACL2 arrays, but without the burden of explicitly naming the arrays. The ACL2 array `COMPRESS` function is provided by the functions `HONS-SHRINK-ALIST` and `HONS-SHRINK-ALIST!`. There are user-level ACL2 functions that allow the associated hash tables to be cleared and removed.

It has occurred to us that a global replacement of `CONS` by `HONS` throughout the implementation of the ACL2 theorem prover might be advantageous. However, there are some subtle issues regarding I/O and state that need further consideration. In addition, many of the algorithms inside the ACL2 system should be recoded if one wanted to take maximum advantage of function memoization. For instance, we developed simple inside-out and an outside-in rewriters; subtle changes in the definition of the rewrite functions were required to eliminate memoization interference from their termination clock input arguments.

We believe, that the use of `HONS` could benefit the ACL2 theorem prover. One profiling benchmark of the ACL2 regression suite revealed that 13% of the execution time was spent in the function `EQUAL`; this was the largest of all profiled functions. Equality tests for two `CONS` trees requires a complete exploration of both trees, and the ACL2 theorem prover often makes such comparisons. For all `CONS` trees that are also `HONS` trees then an equality test can be done with a few machine instructions. Our `HONS-EQUAL` equality check function optimizes equality checking when there is a mixture of `CONS` and `HONS` objects.

5. SYSTEM CONTROL FUNCTIONS

The `HONS` system provides a number of ACL2 user functions that are logically identity functions, but that provide system-level side effects such as enabling or disabling function memoization. These functions allow a user to more tightly control the use of the underlying resources used to implement fast association lists, function memoization, and canonical data representation.

For the active `HONS`, function memoization, and fast association list support data structures, we have functions that permit these data structures to be cleared and initialized.

All of the ACL2 functions and macros presented in this section have *under the hood* implementations.

```
(defun clear-hash-tables ()
  (declare (xargs :guard t))
  ;; Clears the underlying hash tables that are
  ;; used to determine whether a new HONS (CONS)
  ;; pair already exists.
  nil)

(defun clear-hons-acons-table ()
  (declare (xargs :guard t))
  ;; Clears table that is used to identify an
  ;; association list with a hash table for
  ;; hash-based access.
  nil)

(defun clear-memo-tables ()
  (declare (xargs :guard t))
  ;; For all memoized functions, clears tables of
  ;; memoized values.
  nil)

(defun init-hash-tables ()
  (declare (xargs :guard t))
  ;; Like CLEAR-HASH-TABLES, but actually removes
  ;; the underying hash tables and creates new hash
  ;; tables.
  nil)

(defun init-hons-acons-table ()
  (declare (xargs :guard t))
  ;; Like CLEAR-MEMO-TABLES, but removes underlying
  ;; hash table and creates a new "HONS-ACONS" hash
  ;; table.
  nil)

(defun flush-hons-get-hash-table-link (x)
  (declare (xargs :guard t))
  ;; Breaks the link between association list X and
  ;; its corresponding a hash table if such a link
  ;; exists, thus permitting the garbage collection
  ;; of that hash table.
   x)
```

To permit the creation of a fresh environment for all underlying tables, we provide the `HT-LET` macro. The side effects of using this macro are dramatic, as a completely new environment is created where the `HONS` table is saved and a new one is created; all key-value tables for function memoization are also set aside and new, empty tables are created. Upon a user's thread of control leaving this macro, the original tables are restored.

```
(defmacro ht-let (x)
  ;; HT-LET causes the evaluation of X to take place
  ;; in an environment similar to that produced by a
  ;; call of CLEAR-HASH-TABLES, CLEAR-MEMO-TABLES,
  ;; and CLEAR-HONS-ACONS-TABLE, i.e., the HONSing
  ;; hash table, the function memoization hash
  ;; tables, and the HONS-ACONS tables are cleared.
  ;; Upon conclusion of the evaluation of X, the
  ;; previously existing tables are restored.  The
  ;; user may wish to HONS-COPY in and HONS-COPY out
  ;; some terms.
  x)
```

The functions `MEMOIZE-WITH-CONDITION-FN` and `UNMEMOIZE` have rather innocent looking semantics, but they enable and disable memoization. The argument `CONDITION` is an ACL2 term that is evaluated to see if memoization should be attempted; this term must satisfy the same guards as the guards provided by the function `FN` that is to be memoized. Function `MEMOIZE-WITH-CONDITION-FN` could cause errors due to compilation problems with the user-supplied memoization condition; `CONDITION` is not analyzed or compiled, nor are its guards checked, until this call is made. A macro `MEMOIZE` is provided in an associated ACL2 book that makes the use of the `MEMOIZE-WITH-CONDITION-FN` easier for the usual cases. The arguments `HINTS` and `OTF-FLG` are provided to provide hints to the ACL2 theorem prover when it attempts to prove that the term `CONDITION` satisfies the guards for function `FN`.

```
(defun memoize-with-condition-fn
  (fn condition hints otf-flg)
  ;; It is an error to call memoize on something
  ;; that is not a user-defined ACL2 function
  ;; symbol.  It is also an error to call memoize on
  ;; a function that is currently memoized.
  (declare (xargs :guard (and (symbolp fn)
                              (pseudo-termp condition)))
           (ignore condition hints otf-flg))
  ;; Has an "under the hood" implementation.
  fn)

(defun unmemoize (fn)
  ;; It is an error to call unmemoize on something
  ;; not memoized.
  (declare (xargs :guard (symbolp fn)))
  fn)

(defmacro memo-on (fn x)
  ;; It is an error to execute memo-on unless FN is
  ;; already memoized.  MEMO-ON causes X to be
  ;; evaluated in an environment in which FN is
  ;; memoized.
  (declare (ignore fn))
  x)

(defmacro memo-off (fn x)
  ;; It is an error to execute memo-off unless FN is
  ;; already memoized.  MEMO-OFF causes X to be
  ;; evaluated in an environment in which FN is not
  ;; memoized.
  (declare (ignore fn))
  x)
```

This concludes the definition of ACL2 system-level functions used to define and implement the `HONS` system.

6. BOOKS FOR THE HONS SYSTEM

To more easily take advantage of the our `HONS` system, we have defined a book that contains additional definitions and lemmas. Here we present a few of these definitions so as to make more clear some of our use idioms. Some of the comments in the `HONS-HELP` book refer to underlying implementations issues, which provide a potential user with additional intuition about some of the system issues.

To simplify the memoization of functions, the `HONS-HELP` book defines the function `MEMOIZE`. This function, in turn, uses the `MEMOIZE-WITH-CONDITION` macro that calls the ACL2 system-level function `MEMOIZE-WITH-CONDITION-FN` described earlier.

```
(defmacro memoize-with-condition
  (fn condition &key hints otf-flg)
  `(memoize-with-condition-fn
    ,fn ,condition ,hints ,otf-flg))

(defun memoize (fn)
  ;; It is an error to call memoize on something not
  ;; an ACL2 user function.  It is also an error to
  ;; call memoize on a function that is currently
  ;; memoized.
  (declare (xargs :guard (symbolp fn)))
  ;; fn
  (memoize-with-condition fn ''t))
```

To simplify the creation of data structures, we have defined the macros HONS-LIST and HONS-LIST* that operate just like the ACL2 macros LIST and LIST*. Actually, there is another book, HONS-HELP2, that defines a number of aliases; in this book we define aliases for functions and macros that begin with HONS-. For instance, HIST is defined as an alias for HONS-LIST and HIST* is defined as an alias for HONS-LIST*. Thus, a user can evaluate (HIST* 1 2 3) and see (1 2 . 3) printed.

Since it is not possible to distinguish between data structures that include structure sharing, we have defined a mechanism to count the number of actual HONS (CONS) elements. We first define a tail-recursive version of LEN.

```
(defun hons-len1 (x acc)
  (declare (xargs :guard (natp acc)))
  (if (atom x)
      acc
    (hons-len1 (cdr x) (+ 1 acc))))

(defun hons-len (x)
    (declare (xargs :guard t))
    (hons-len1 x 0))
```

We next define a function that collects each unique subtree as a key into an association list.

```
(defun cons-subtrees (x al)
  ;; (CONS-SUBTREES X NIL) is an alist that
  ;; associates each subtree of X with the constant
  ;; T, without duplication.
  (declare (xargs :guard (hons-alistp al)))
  (cond ((atom x) al)
        ((hons-get x al) al)
        (t (cons-subtrees
            (car x)
            (cons-subtrees (cdr x)
                           (hons-acons x t al))))))
```

Finally, we just count the length of the resulting association list after *throwing away* the associated hash table. The symbol 'number-subtrees is used to reduce the chances of another association list being identical to the one constructed for the purposes of counting the number of entries created by CONS-SUBTREES.

```
(defun number-subtrees (x)
  (declare (xargs :guard t))
  (hons-len (flush-hons-get-hash-table-link
             (cons-subtrees x 'number-subtrees))))
```

We have defined some set operations that appear to have quadratic performance, but because of our use of our fast association list mechanism, these set operations generally exhibit linear performance in the size of the sets. Objects in the sets may be any object, but better performance will be provided if pairs are constructed with HONS. The function BUILD-FAST-ALIST-FROM-LIST just builds an (fast) association list from a list of data objects.

```
(defun build-fast-alist-from-list (l acc)
  (declare (xargs :guard (hons-alistp acc)))
  (hons-put-list l t acc))

(defun hons-intersection1 (l al acc)
  (declare (xargs :guard (hons-alistp al)))
  (cond ((atom l) acc)
        ((hons-get (car l) al)
         (hons-intersection1 (cdr l) al
                             (cons (car l) acc)))
        (t (hons-intersection1 (cdr l) al acc))))

(defun hons-intersection (l1 l2)
  (declare (xargs :guard t))
  (let ((temp-table
         (build-fast-alist-from-list
          l2 '*hons-intersection-alist*)))
    (let ((ans (hons-intersection1 l1 temp-table nil)))
      (let ((temp-table
             (flush-hons-get-hash-table-link
              temp-table)))
        (declare (ignore temp-table))
        ans))))

(defun hons-set-diff1 (l al acc)
  (declare (xargs :guard (hons-alistp al)))
  (cond ((atom l) acc)
        ((hons-get (car l) al)
         (hons-set-diff1 (cdr l) al acc))
        (t (hons-set-diff1 (cdr l) al
                           (cons (car l) acc)))))

(defun hons-set-diff (l1 l2)
  (declare (xargs :guard t))
  (let ((temp-table (build-fast-alist-from-list
                     l2 '*hons-set-diff-alist*)))
    (let ((ans (hons-set-diff1 l1 temp-table nil)))
      (let ((temp-table
             (flush-hons-get-hash-table-link
              temp-table)))
        (declare (ignore temp-table))
        ans))))

(defun hons-union (l1 l2)
  (declare (xargs :guard t))
  (let ((temp-table (build-fast-alist-from-list
                     l2 '*hons-union-alist*)))
    (let ((ans (hons-set-diff1 l1 temp-table l2)))
      (let ((temp-table
             (flush-hons-get-hash-table-link
              temp-table)))
        (declare (ignore temp-table))
        ans))))
```

With a list of 100,000 numbers we can compute the set intersection, union, and set difference in under one second. Below, we have already initialized the ACL2 top-level *variable* LOTS-OF-NUMBERS to a list containing 100,000 different natural numbers. Note that we have used more abbreviations from the HONS-HELP2 book; i.e., HEN for HONS-LEN, HSET-DIFF for HONS-SET-DIFF, HUNION for HONS-UNION, and HINTERSECTION for HONS-INTERSECTION.

```
ACL2 !>(time$ (hen (hintersection
                    (@ lots-of-numbers)
                    (@ lots-of-numbers))))
(HEN (HINTERSECTION (@ LOTS-OF-NUMBERS)
                    (@ LOTS-OF-NUMBERS)))
```

```
took 702 milliseconds (0.702 seconds) to run.
100000

ACL2 !>(time$ (hen (hunion
                   (@ lots-of-numbers)
                   (@ lots-of-numbers))))
(HEN (HUNION (@ LOTS-OF-NUMBERS)
             (@ LOTS-OF-NUMBERS)))
took 652 milliseconds (0.652 seconds) to run.
100000

ACL2 !>(time$ (hen (hset-diff
                   (@ lots-of-numbers)
                   (@ lots-of-numbers))))
(HEN (HSET-DIFF (@ LOTS-OF-NUMBERS)
                (@ LOTS-OF-NUMBERS)))
took 680 milliseconds (0.680 seconds) to run.
0
```

7. A BDD IMPLEMENTATION

As a further example of the use of the HONS system, we present an implementation of the BDD if-then-else (ITE) operator of Bryant [3]. To implement BDDs requires three things: unique data representation, memoization of the ITE operator, and fast equality checking. By using HONS elements to represent BDDs we get canonical data representation and fast equality checking. By memoizing our BDD Q-ITE operator, from which all other Boolean connectives can be built, we have ITE function memoization.

We represent BDDs as HONS trees. We recognize a well-formed BDD with the NORMP predicate. Note, that the BDD does not contain the variables themselves, thus our internal representation of a BDD node is two machine pointers.

```
(defun normp (x)
  (declare (xargs :guard t))
  (if (atom x)
      (booleanp x)
    (and (normp (car x))
         (normp (cdr x))
         (if (atom (car x))
             (not (equal (car x) (cdr x)))
           t))))
```

To represent a variable, we use the function VAR-TO-TREE, which given variable order VARS builds the BDD representing variable VAR. Notice the first case is a non-sensible request as it corresponds to a request to build a BDD for a variable not in the BDD variable order.

```
(defun var-to-tree (var vars)
  (declare (xargs :guard (and (symbolp var)
                              (symbol-listp vars))))
  (cond ((atom vars) nil)
        ((eq var (car vars))
         (hons t nil))
        (t (hons (var-to-tree var (cdr vars))
                 (var-to-tree var (cdr vars))))))
```

In our BDD implementation, we use a BDD representation where all levels of the BDD exist until a branch can be specified by a terminal, T or NIL. The BDD algorithm requires several other simplifications to keep the representation of the trees canonical; we list the simplifications forms below:

```
(Q-ITE X X Y)   ==>  (Q-ITE X t Y)}
(Q-ITE X Y X)   ==>  (Q-ITE X Y nil}
(Q-ITE X Y Y)   ==>  Y
(Q-ITE X t nil) ==>  X
```

Shown below is the entire definition of the BDD ITE operator using our HONS implementation. Once the BDD ITE function Q-ITE is memozied, this BDD implementation operates similarly to other BDD packages without the use of dynamic variable re-ordering. Using a BDD reordering function we have defined (not shown) and verified, and using the function NUMBER-SUBTREES (see Section 6) to measure the size of BDDs, one could augment our BDD implementation to include dynamic variable re-ordering.

```
(defmacro qcar (x) `(cond ((atom ,x) ,x) (t (car ,x))))

(defmacro qcdr (x) `(cond ((atom ,x) ,x) (t (cdr ,x))))

(defmacro qcons (x y)
  `(cond ((or (and (eq ,x t)   (eq ,y t))
              (and (eq ,x nil) (eq ,y nil)))
          ,x)
         (t (hons ,x ,y))))

(defun q-ite (x y z)
  (declare (xargs :measure (acl2-count x) :guard t))
  (cond
   ((null x) z)
   ((atom x) y)
   (t (let ((y (if (hqual x y) t y))
            (z (if (hqual x z) nil z)))
        (cond
         ((hqual y z) y)
         ((and (eq y t) (eq z nil)) x)
         (t (let ((a (q-ite (car x) (qcar y) (qcar z)))
                  (d (q-ite (cdr x) (qcdr y) (qcdr z))))
              (qcons a d))))))))
```

By defining a meaning function, EVAL-BDD, for our BDD representation, we can prove that Q-ITE operates just like the ACL2 IF function.[1] Although not shown, we have also proved that when the Q-ITE function is given three arguments each recognized by NORMP, that Q-ITE returns an output recognized by NORMP.

```
(defun eval-bdd (x values)
  (declare (xargs :guard (boolean-listp values)))
  (if (atom x)
      x
    (if (car values)
        (eval-bdd (car x) (cdr values))
      (eval-bdd (cdr x) (cdr values)))))

(defthm q-ite-correct
  (implies (and (normp x)
                (normp y)
                (normp z))
           (equal (eval-bdd (q-ite x y z) vals)
                  (if (eval-bdd x vals)
                      (eval-bdd y vals)
                    (eval-bdd z vals)))))
```

We regularly use our BDD implementation for a variety of tasks where efficient Boolean function manipulations are required. For instance, we have used BDDs to represent sets, and BDD operations to perform set operations, such as unions and intersections. We have defined co-factoring and a general BDD re-ordering algorithm that we have proved correct. We have defined Boolean quantifier functions. Finally, we have developed a finite-state machine (FSM) language, and we have defined the reachability and other operations on FSMs represented in our language using BDD operations.

[1]We note our thanks to Qiang Zhang for getting this proof through the ACL2 system.

8. CONCLUSION

The introduction of function memoization into ACL2 makes the implementation of dynamic programming problems easier as it eliminates the need to store and retrieve previously computed values. The canonical representation of ACL2 data enables fast association list operations and often reduces memory requirements. Using these features we have defined the BDD ITE operator with just 10 lines of ACL2 code. We have also verified its implementation.

We believe the system-wide benefits of unique object representation will take some time to realize. We continue to discover new ways of implementing even basic functions with improved performance using the `HONS` system. We have used these techniques to implement a consensus algorithm for phylogenetic trees that is much faster than the best available implementations [4]. We have used the `HONS` system to implement a rewriter that have impressive performance, albeit on a small set of examples. The `HONS` system essentially makes hash tables available to the ACL2 user with a simple semantics.

9. REFERENCES

[1] Henry Baker. The Boyer Benchmark at Warp Speed. In *ACM Lisp Pointers*, Volume 3, July–September, 1992, pages 13–14.

[2] Henry Baker. A Tachy 'TAK'. ACM Lisp Pointers Volume 3, July–September, 1992, pages 22–23.

[3] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. In *IEEE Transactions on Computers*, Volume C-35, Number 8, pages 677-691, August, 1986.

[4] Robert S. Boyer, Warren A. Hunt Jr., and Serita M. Nelesen. A Compressed Format for Collections of Phylogenetic Trees and Improved Consensus Performance. In *Algorithms in Bioinformatics: 5th International Workshop, WABI 2005*, Lecture Notes in Computer Science 3692, pages 353-364, Springer-Verlag, 2005.

[5] A. .P. Ershov. On Programming of Arithmetic Operations. In the *Communications of the ACM*, Volume 118, Number 3, August, 1958, pages 427–430.

[6] Eiichi Goto. Monocopy and Associative Algorithms in Extended Lisp. University of Toyko, Technical Report TR-74-03, 1974.

[7] Richard P. Gabrial. Performance and Evaluation of Lisp Systems. MIT Press, 1985.

[8] M. Kaufmann and J S. Moore. ACL2: An Industrial Strength Version of NQTHM. Proceedings of the *Eleventh Annual Conference on Computer Assurance (COMPASS-96)*, pages 23-34, IEEE Computer Society Press, June 1996.

[9] Matt Kaufmann, Panagiotis Manolios and J Strother Moore. Computer-Aided Reasoning: An Approach. Kluwer Academic Press, Boston, Massachusetts, 2000.

[10] Donald Michie. Memo functions: a Language Feature with Rote Learning Properties. Technical Report MIP-R-29, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1967.

[11] Donald Michie. Memo Functions and Machine Learning. In *Nature*, Volumne 218, 1968, pages 19–22.

[12] J Strother Moore. Introduction to the OBDD Algorithm for the ATP Community. In *Journal of Automated Reasoning*, Volume 12, Number 1, February 1994, pages 33–45.

10. ACKNOWLEDGMENTS

We would like to acknowledge our many conversations with Bill Legato; our interactions certainly improved this work. We would like to thank the early users: Serita Nelesen and Sol Swords. We want to thank Qiang Zhang for his proof of our BDD ITE operator. This material is partially based upon work supported by DARPA and the National Science Foundation under Grant No. CNS-0429591.