Unique ACL2 Object Representation


February 25, 2009

Computer Sciences Department
University of Texas
1 University Way, M/S C0500
Austin, TX 78712-0233

E-mail:
{boyer,hunt}@cs.utexas.edu
TEL: +1 512 471 9748
FAX: +1 512 471 8885

Centaur Technology, Inc.
7600-C N. Capital of Texas Hwy
Suite 300
Austin, Texas 78731

E-mail:
{boyer,hunt}@centtech.com
TEL: +1 512 418 5797
FAX: +1 512 794 0717
The introduction of unique object representation to the ACL2 system allows ACL2 users to sometimes write much more efficient algorithms.

- The logical story
  - No changes to the ACL2 logic.
  - HONS defined to be CONS.
  - Association lists provided with constant-time lookup.
  - Function memoization mechanism provided.

- The implementation story
  - Internal data structures are used to identify unique CONS pairs.
  - Hash tables are used to support fast association list access.
  - Memoized function results are stored in hash tables.
  - Real-time performance monitoring provided with function memoization.

- To use HONS effectively, the HONS frontier must be understood.
Presentation Outline

1 Introduction

1 Definition of HONS and HONS-EQUAL

1 The HONS Frontier

1 Fast Association Lists

1 Function Memoization

1 Real-Time Performance Measurement
HONS and HONS-EQUAL are introduced as normal ACL2 functions.

```
(defmacro defn (f a &rest r)
  `(defun ,f ,a (declare (xargs :guard t)) ,@r))
```

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

```
(defun hons-equal (x y) (equal x y))
```

- **HONS**
  - is exactly defined to be CONS, and
  - runs approximately 20 times slower (with CCL) than CONS.

- **HONS-EQUAL**
  - is exactly defined to be EQUAL, and
  - performs *short-circuit* equality checks.
Although, nowhere made available, one must always keep the HONS frontier in mind; all objects within the frontier have a unique internal representation.

We recognize a unique object with the internal HONSP predicate.

- All constants
- CONS objects created with HONS
How Do We Maintain the HONS Frontier?

When the \((\text{HONS } x \ y)\) is evaluated, a two-level lookup is performed.
HONS-COPY duplicates objects as necessary to extend the HONS frontier.

```
(defn hons-copy (x) x) ;; Has internal implementation
```

For example, if HONS-COPY is called with a reference to the left-most pointer to the upper-left CONS node, then the graph is transformed.
Using unique ACL2 objects as association-list keys, we have developed a faster lookup mechanism that obeys this semantics.

```lisp
(defun hons-assoc-equal (x y)
  (cond ((atom y) nil)
    ((and (consp (car y))
       (hons-equal x (car (car y))))
      (car y))
    (t (hons-assoc-equal x (cdr y))))
)

(defun hons-get-fn-do-hopy (x l)
  ;; Has an "under-the-hood" implementation.
  (hons-assoc-equal x l))

(defmacro hons-get (x l)
  (list 'hons-get-fn-do-hopy x l))
```
We define two ACL2 functions to aid with the construction of fast association lists.

\[
\text{(defn hons-acons \ (key value l)}
\text{\ (cons (cons (hons-copy key) value) l))}
\]

\[
\text{(defn hons-acons! \ (key value l)}
\text{\ (hons (hons (hons-copy key) value) l))}
\]

Notice that \text{HONS-ACONS!} creates an association list which is itself is a unique object.

- Such an association list may assist function memoization; however
- Such an association list may be \text{stolen} – more later.
When using HONS-ACONS, the HONS frontier is only with the association list keys – the spine is composed of CONS objects.
When using **HONS-ACONS!**, everything is within the HONS frontier.
How is the Association-List Hashtable Found?

When \texttt{HONS-GET} is called, we use the \textit{top-most} \texttt{CONS} as a key into a table of fast association lists; however, it might be stolen!
Function Memoization

For functions that are repeatably called on highly structure-shared data objects (e.g., BDDs), function memoization can reduce evaluation costs.

- Common-Lisp compliant functions may be memoized.
- An associated hash table is created when the function is memoized.

Computing the value of a function requires several steps.

- A condition is computed to see if memoization should be attempted.
- When a memoized function is called, its args are combined into a key.
- Using this key, a lookup is done in the memoization hash table.
- If the lookup is successful, the corresponding previously computed value is returned.
- Otherwise, the original function is called, and its result is computed.
- This newly computed value is then installed in this function memoization table with the key just computed.
- Finally, the answer is returned.
Functions with a single argument are memoized with a single hashtable.

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

(memoize 'fib :condition '(< 40 x)) ;; *** D E M O ***
Memoizing \((F \ x \ y \ z)\) requires two PONS objects — a function-specific collection of HONS-like objects with supporting hashtables.
Unique objection and function memoization allow a very small, but
competitive BDD package to be created.

(defabbrev qcar (x) (if (consp x) (car x) x))
(defabbrev qcdr (x) (if (consp x) (cdr x) x))

(defabbrev qcons (x y)
  (if (or (and (eq x t) (eq y t))
          (and (eq x nil) (eq y nil)))
    x
    (hons x y))))

(defn q-not (x)
  (if (atom x)
    (if x nil t)
    (hons (q-not (car x))
      (q-not (cdr x))))))
This Q-ITE function includes optimizations necessary to keep BDD objects normalized. This implementation is in everyday, industrial use.

\[
\text{(defn q-ite (x y z)}
\]

\[
\begin{align*}
& \quad \text{(cond)} \\
& \quad \quad \text{((null x) z)} \\
& \quad \quad \text{((atom x) y)} \\
& \quad \quad \text{(t (let ((y (if (hqual x y) t y)) ; Simp Left branch}
& \quad \quad \quad \text{\quad (z (if (hqual x z) nil z))) ; Simp Right branch}
& \quad \quad \text{\begin{align*}
& \quad \quad \quad \text{(cond}} \\
& \quad \quad \quad \quad \text{((hqual y z) y))} \quad ; \text{(if x y y) } \Rightarrow \text{ y} \\
& \quad \quad \quad \quad \text{((and (eq y t) (eq z nil)) x)} \quad ; \text{(if x T NIL) } \Rightarrow \text{ x} \\
& \quad \quad \quad \quad \text{((and (eq y nil) (eq z t)) (q-not x))} \quad ; \text{For speed} \\
& \quad \quad \quad \quad \text{(t (let ((a (q-ite (car x) (qcar y) (qcar z)))}
& \quad \quad \quad \quad \quad \quad \text{(d (q-ite (cdr x) (qcdr y) (qcdr z))))}
& \quad \quad \quad \quad \quad \quad \text{(qcons a d)))))))))
\end{align*}
\]
Real-Time Performance Measurement

In real time, we track the number of CONS objects identified as HONS objects – this information can be used as a real-time performance monitor.

After loading examples.lsp, this HONS information is externally available.

? (hsum)
(defun hons-summary

Hons hits/calls 2.8E+5 / 4.7E+5 = 0.58
*HONS-CDR-HT* count/size 1.46E+5 / 2.01E+5 = 0.73
*HONS-CDR-HT-EQL* count/size 3.9E+3 / 5.2E+3 = 0.74
*NIL-HT* count/size 2.6E+4 / 2.6E+4 = 0.99
*HONS-STR-HT* count/size 5.3E+3 / 7.8E+3 = 0.67
Number of sub tables 16
Sum of sub table sizes 9.2E+3
Number of honeses 2.24E+5

223556
Example Summary of Q-NOT Measurements

(defun Q-NOT hits/calls 3.4E+4 / 5.4E+4 = 0.63
Time of all outermost calls 0.42
Time per call 7.7E-6
Heap bytes allocated 3.2E+6
Heap bytes allocated per call 59.61
Hons calls 2.1E+4
Time per missed call 2.07E-5
From Q-NOT 4.1E+4 calls
From T-FIX 9.9E+3 calls took 0.25
From outside 1.69E+3 calls took 0.15
From F-NOT 942 calls took 7.7E-3
From Q-BINARY-XOR 488 calls took 4.9E-3
From Q-BINARY-IFF 94 calls took 1.51E-4
From Q-ITE-FN 18 calls took 1.63E-4
From NQV 6 calls took 4.0E-4
Memoize table count/size 2.0E+4 / 2.6E+4 = 0.76)
Presently, in `books/misc` the files `qi.lisp` and `qi-correct.lisp` provide additional HONS-based functionality.

- `qi.lisp` – definitions of BDD package
- `qi-correct.lisp` – verification of BDD functions

Jared Davis and Sol Swords have written a new book that extends the books mentioned above.

- Library can rewrite all BDD functions to Q-ITE functions
- Library provides other rewriting strategies.
- Provides “pick-a-point” proof support for BDD-related proofs.