Memories: Array-like Records for ACL2

Jared Davis
Department of Computer Sciences
The University of Texas at Austin
jared@cs.utexas.edu

ABSTRACT

We have written a new records library for modelling fixed-size arrays and linear memories. Our implementation provides fixnum-optimized $O(\log_2 n)$ reads and writes from addresses $0, 1, \ldots, n-1$. Space is not allocated until locations are used, so large address spaces can be represented. We do not use single-threaded objects or ACL2 arrays, which frees the user from syntactic restrictions and slow-array warnings. Finally, we can prove the same hypothesis-free rewrite rules found in `misc/records` for efficient rewriting during theorem proving.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*correctness proofs, formal methods*; E.1 [**Data**]: Data Structures—*arrays, records*; B.3.4 [**Memory Structures**]: Reliability, Testing, and Fault-Tolerance

General Terms

Verification, Performance

Keywords

ACL2, arrays, fixnum optimization, linear address spaces, mbe, records

1. BACKGROUND

A record is a collection of named values which can be accessed and updated. In ACL2, records might be represented in many ways. If there are a fixed set of keys, custom `cons` structures or position-based lists might be appropriate. For more general records, alists or the `misc/records` book [3] can be used.

The `misc/records` book introduces two functions:

- `(g a r)` retrieves the value of field `a` in record `r`, and
- `(s a v r)` copies record `r`, changing the value of field `a` to `v`.

ACL2 '06 Seattle, Washington USA

These functions are novel in that *every* ACL2 object can be interpreted as if it were a record, *any* object can be used as a key, and *any* object can be used as a value. In particular, the following are theorems:

1.
```
(equal (g a (s a v r))
       v)
```
2.
```
(implies (not (equal a1 a2))
         (equal (g a1 (s a2 v r))
                (g a1 r)))
```
3.
```
(equal (s a (g a r) r)
       r)
```
4.
```
(equal (s a v1 (s a v2 r))
       (s a v1 r))
```
5.
```
(implies (not (equal a1 a2))
         (equal (s a1 v1 (s a2 v2 r))
                (s a2 v2 (s a1 v1 r))))
```

These high quality rules make records attractive for reasoning about *linear address spaces*: records where the names are $0, 1, \ldots, n-1$. Other approaches, such as indexing into lists, or using alists, single threaded objects, and ACL2 arrays do not facilitate these hypothesis-free rules and can therefore be more cumbersome to reason about [3]. It should be no surprise that the *Rockwell Challenge* [2] suggests using records as a memory representation, and that records are used in the solutions proposed by Liu [4] and Moore [6].

Ease of reasoning is not enough. The ACL2 user would also like to efficiently simulate his or her model, particularly when studying a microprocessor [8]. To simulate long-running programs, the memory model must be implemented efficiently. Here `misc/records` may be inadequate, particularly if the memory to simulate is large: `g` and `s` require two and three linear passes through the memory, respectively. An enhanced book by Rob Sumners, `defexec/-chapter3/records`, uses MBE to eliminate these additional passes. But the number of addresses in a processor's memory might be enormously large and memory operations occur frequently in programs, so linear-time memory operations may be unacceptably slow.

2. OUR APPROACH

We have developed *memories*, a new records library which provides fast access to array-like addresses. To do this, we combine a `misc/record` with an efficient tree structure. The tree is used to hold the values of array addresses, while the

record is used to hold other values. This hybrid approach has several advantages, which we enumerate below.

The same hypothesis-free rewrite rules of `misc/records` are theorems, so reasoning about memories is as straightforward as reasoning about `misc/records`. The system is entirely applicative, so the user faces neither syntactic restrictions from single threaded objects nor slow array warnings.

Space efficiency is achieved, because the tree starts small and grows as new locations are written. This permits even huge arrays with 2^{64} addresses to be simulated on 32–bit machines, so long as only a reasonable number of those addresses are used during simulation.

Time efficiency is achieved for array addresses, with loads and stores taking $O(\log_2 n)$, where n is size of the array. These are single pass operations, optimized with MBE and fixnum arithmetic. Using a 2.8 GHz Pentium 4 test system with GCL 2.6.7 and ACL2 3.0, we have the following results reading from and writing to a pre-populated memory.

Memory Size	**Loads/sec**	**Stores/sec**
2^8	11,000,000	2,200,000
2^{16}	2,500,000	625,000
2^{32}	785,000	300,000
2^{64}	240,000	125,000

There is no meaningful way to compare memories to `misc/records`, where the performance even of the `defexec`-enhanced version is highly dependent upon the order of the storage operations and the size of the record being manipulated. For example, zeroing the first 2^{16} addresses of an empty `defexec` record took less than a second when the locations were written in descending order, but took 15 minutes in ascending order. When the record was pre-populated instead of being empty, these same operations took 9 and 15 minutes, respectively. On the other hand, memories are not significantly affected by the order of stores, and it took less than a second to do each of these operations with memories, pre-populated or not, even with a size of 2^{64}.

3. IMPLEMENTATION: MEMTREES

Our memories are mainly based on a custom tree structure we call **memtrees**. Every memtree has some fixed depth and can be used to store up to 2^d elements, addressed by $0, 1, \ldots, 2^d - 1$. These trees are not particularly complicated:

- Any ACL2 object, x, is a memtree of depth 0. Such trees have a single element, whose address is zero, and whose value is exactly x.
- `(`a `.` b`)` is a memtree of depth $d > 0$ whenever a, b are memtrees of depth $d - 1$. Since a, b each have 2^{d-1} elements, the resulting tree has 2^d elements.

We permit `nil` to represent a memtree of any depth whose elements are all `nil`. This allows us to collapse unused blocks of memory into a single `nil`.

We say memtrees m_1 and m_2 are equivalent if their every address has the same value. We would like equivalent memtrees to be `equal`, so that no new equivalence relation is needed. Unfortunately, we can now imagine equivalent trees that are not `equal`, for example, `nil` and `(nil . nil)` are both memtrees of depth 1 which are equivalent. To remedy this, we further require either a or b to be non-`nil` for `(`a `.` b`)` to be a memtree. This way, only the most collapsed version of each tree is acceptable.

```
(defun _memtree-p (mtree depth)
  (cond ((zp depth) t)
        ((atom mtree) (null mtree))
        (t (and (_memtree-p (car mtree) (1- depth))
                (_memtree-p (cdr mtree) (1- depth))
                (or (car mtree) (cdr mtree))))))
```

We think of array addresses in their base–2 expansions, i.e., as sequences of bits. These bit sequences describe paths through the tree. Whenever the "next bit" is 0, we follow the `car` of the tree, and otherwise we follow the `cdr`. Since the `car` and `cdr` of `nil` are `nil`, no special care needs to be taken to support loading from collapsed memtrees:

```
(defun _memtree-load (addr mtree depth)
  (if (zp depth)
      mtree
    (_memtree-load (floor addr 2)
                   (if (equal (mod addr 2) 0)
                       (car mtree)
                     (cdr mtree))
                   (1- depth))))
```

Storing is more complicated since we must leave the tree in canonical form. To handle this, we write two functions: one which stores `nil`, and one which stores non-`nil` values. Below, we only show the function to store non-`nil` values, since the other is so similar:

```
(defun _memtree-store (addr elem mtree depth)
  (if (zp depth)
      elem
    (let ((quotient (floor addr 2)))
      (if (equal (mod addr 2) 0)
          (cons (_memtree-store quotient elem
                                (car mtree)
                                (1- depth))
                (cdr mtree))
        (cons (car mtree)
              (_memtree-store quotient elem
                              (cdr mtree)
                              (1- depth)))))))
```

To make memtree loading and storing fast, each of these functions is guarded so that `depth` is a natural number, `mtree` is a memtree of the appropriate depth, and `addr` is an acceptable array address (i.e., a natural less than $2^{\texttt{depth}}$). We use MBE to perform "strength reduction", replacing calls of `(floor addr 2)` with `(ash addr -1)`, and calls of `(mod addr 2)` with `(logand addr 1)`.

We also take advantage of fixnum arithmetic. We begin by introducing "fixnum versions" of each function, by guarding them so that `addr` and `depth` must be fixnums. This allows for our bit operations and our decrementing of `depth` to be compiled by GCL into efficient C arithmetic operations, such as `&` and `>>`.

In practice, `depth` will always be a fixnum. For example, even an array of size 2^{64} only requires a depth of 64. However, `addr` does not enjoy this property, e.g., a 64–bit memory will result in addresses which are too large to be fixnums on 32–bit host machines.

To deal with this complication, we add "half-optimized" versions of each function, wherein we only assume that `depth` is a fixnum (i.e., we do not require that `addr` is also a fixnum). Here, we change the base case: instead of recurring until `depth` is zero, we recur only until `depth` is small enough that `addr` must be a fixnum. At that point, we call our "fully-optimized" version of the function, allowing us to take advantage of fixnum arithmetic for both `addr` and `depth` in the final part of the computation. In short, we use bignum arithmetic only until it is safe to switch to fixnum arithmetic.

All of this work is really just "peephole optimization" that does nothing to improve upon our algorithmic complexity. However, the practical benefits are quite significant and lead to the performance results mentioned earlier.

We wrap these many variants into single `load` and `store` operations which inspect their inputs and then call the appropriately optimized function. These decisions are based on the sizes of the inputs and, for stores, whether or not the value to store is `nil`. These conveniences add minimal overhead, and shield our users from needing to deal with fixnum concerns in their guards.

4. IMPLEMENTATION: MEMORIES

We would like to prove the basic "record theorems" about memtrees. To eliminate some hypotheses, we can redesign the `load` and `store` functions so that they "fix" their arguments as in [1, 5]. We treat bad depths and addresses as if they were zero, and ill-formed memtrees as if they were `nil`. These conveniences add no execution overhead thanks to guards and MBE.

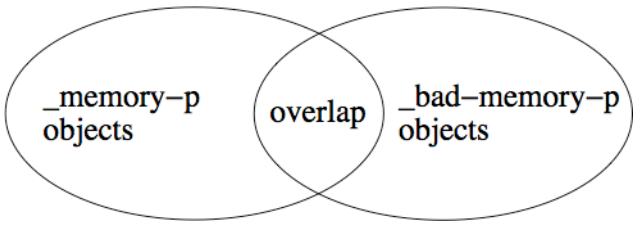
With these changes, we can almost prove the record theorems. However, we need hypotheses to ensure that the addresses are valid and, for the third theorem, that the memtree is well-formed. Also, the memtree's depth parameter unaesthetically occurs in each theorem. These are the same sort of issues that make records so appealing in the first place!

Most of these deficiencies can be addressed with a new abstraction. A **memory** groups together a memtree with some other data, including:

- The depth of the memtree (by bundling this alongside the memtree, we no longer need to consider and manage tree depths separately, so `load` and `store` need not take an extra `depth` parameter),
- An "intended size" of the memory (this can be any integer with $0 \leq \mathit{size} \leq 2^{\texttt{depth}}$, and allows us to act as if our memory sizes need not be powers of two), and
- A regular `misc/record` (this gives us somewhere to store values for non-array addresses, freeing us from address fixing and eliminating address-related hypotheses).

After these changes, we can remove all our hypotheses except for the well-formedness of our memory in the third theorem. This is a particularly difficult hypothesis to remove. Fortunately, the same issue was encountered and addressed in [3], and the solution described there can be successfully applied here as well.

The basic idea is as follows. We are going to partition the ACL2 universe into three infinite sets: the memories, the bad memories, and their overlap, as the following Venn diagram suggests:



The `_memory-p` objects are the aggregates we have described above, and include a memtree component, size, depth, and record. The `_bad-memory-p` objects are all other ACL2 objects, and also all of the proper `_memory-p` objects with size and depth 0, whose memtrees are recursively `_bad-memory-p` objects.

The infinite size of the overlap allows us to create an isomorphism between all of the `_bad-memory-p` objects and the overlap objects.

- `_to-mem` takes any `_bad-memory-p` object, x, and maps it to a new `_memory-p` object whose size/depth are zero, record is `nil`, and memtree is x.
- `_from-mem` takes any overlap object and simply returns its memtree.

When given inputs that are not `_bad-memory-p`s, `_to-mem` and `_from-mem` simply act as the identity. Furthermore, the following are theorems:

- `(_memory-p (_to-mem x))`
- `(equal (_from-mem (_to-mem mem)) mem)`
- ```
  (implies (_memory-p mem)
           (equal (_to-mem (_from-mem mem))
                  mem))
  ```

We now rename our "straightforward" implementations of `load` and `store` to `_load` and `_store`, and create the following new wrappers which also perform these various conversions:

- `(load addr mem) = (_load addr (_to-mem mem))`
- ```
  (store addr val mem) =
   (_from-mem (_store addr val (_to-mem mem)))
  ```

This is all quite similar to the work in [3]. In particular, `_to-mem` is like `acl2->rec`, `_from-mem` like `rec->acl2`, `_load` and `_store` are like `g-aux` and `s-aux` and `load` and `store` are like `g` and `s`.

With these definitions in place, we can prove all five record theorems without hypotheses. As an example, we will now sketch the proof of property 3. Begin with `(store addr (load addr mem) mem)`. By the definitions of store and load, this is the same as:

```
(_from-mem (_store addr
                   (_load addr (_to-mem mem))
                                 a _memory-p object
                   (_to-mem mem)))
                   a _memory-p object
```

This call of _store now has the form (_store a (_load a m) m), and we can see that m is a _memory-p object. Since m is a _memory-p object, it is well-formed from the perspective of _store, and the entire _store reduces to m. This results in the term (_from-mem (_to-mem mem)), which further reduces to mem, which is what we wanted to show. The other proofs are similar.

Our last remaining task is to make these operations efficient. We define the "good memories", recognized by memory-p, as those memories whose size and depth are nonzero. For any memory-p object, _to-mem and _from-mem are the identity. Then, by guarding load and store to take only valid memory-p objects as inputs, we can use MBE to remove these conversions entirely during execution.

5. CONCLUSIONS

Our library is entirely applicative, supports the same hypothesis-free rewrite rules as misc/records, can be used to represent large memories, and allows for fast simulation. It should be ideal for modelling large processor memories, where both simulation speed and reasoning ability are important.

We were somewhat fortunate that our memtrees have a canonical form. Other tree structures such as misc/symbol-btrees probably cannot be combined with records in this way without sacrificing equal-based reasoning.

The library is freely available under the terms of the GNU General Public License, and will be distributed with the next version of ACL2.

6. AFTERWORD

The virtues of hypothesis-free, equal-based rewrite rules have been promoted in previous workshop papers. To support such rules, a total order was added to ACL2 [7], leading to the misc/records library and also to my ordered sets library [1]. The memories presented here follow this tradition. But I am not entirely convinced that equal-based hypothesis-free rewriting is such a good idea.

Consider misc/records. A clever trick was required to develop definitions which would support these rewrite rules, and it is tricky to extend the library without understanding how this trick works. An alist-equiv based library using assoc and acons is far simpler to write, and can more easily be extended to support operations such as domain and range.

The main reason to favor hypothesis-free rewriting is that hypotheses can prevent rules from firing. This can slow down a proof attempt or, worse, require a user to investigate why a term is not being rewritten. I have found that the combination of force and backchain limits is usually sufficient to address these problems. And, compared with discovering a deep trick such as this invertible mapping, this approaches seems much simpler.

A downside of equivalence based rewriting is the need to prove congruence rules. In many of the most typical cases, the desired rules are easy to identify and are not difficult to prove. But, we will admit that it can be a burden to deal with aggregates when their components make use of different equivalence relations.

For example, suppose we use unordered lists to represent sets and alists to represent mappings. Then, it may take some effort to write down what we mean by equivalent "sets of maps," and the resulting operation may not be particularly efficient. On the other hand, using ordered sets and records, the same operation is just equal.

7. REFERENCES

[1] Jared Davis. Finite set theory based on fully ordered lists. In *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2004)*, November 2004.

[2] David Greve and Matt Wilding. Dynamic datastructures in ACL2: A challenge, November 2002.

[3] Matt Kaufmann and Rob Sumners. Efficient rewriting of operations on finite structures in ACL2. In *Third International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-2002)*, April 2002.

[4] Hanbing Liu. A solution to the Rockwell challenge. In *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2003)*, July 2003.

[5] J S. Moore. Finite set theory in ACL2. In *14th International Conference on Theorem Proving in Higher Order Logics*, September 2000.

[6] J S. Moore. Memory taggings and dynamic data structures. In *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2003)*, July 2003.

[7] Panagiotis Manolios and Matt Kaufmann. Adding a total order to ACL2. In *Third International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-2002)*, April 2002.

[8] Matthew Wilding, David Greve, and David Hardin. Efficient simulation of formal processor models. *Formal Methods in System Design*, 18(3), May 2001.