A Solution to the Rockwell Challenge

Hanbing Liu

June 29, 2003

Abstract

The Rockwell challenge is concerned with how to reason effectively about updates to dynamic data structures [1]. The major difficulty comes from the fact that those dynamic structures are represented as memory cells in a linear address space, where the concept of objects being independent and well formed is not explicit. In this paper, I present my solution to this challenge problem. The key is to recover the concept of a complicated data structure being composed of independent objects and to reduce update *during* traversal of the data structure to simpler operations that apply a suitable sequences of updates *after* the traversal. Major proof steps are presented in this paper. The intuitions are also explained. A sketch of a generalized problem is presented together with its partial solution.

1 Background

When programmers reason about how their programs behave, they seldom think about data structures in their programs as entities laid down in a linear address space. Most likely, a program is considered to be implementing some algorithm that manipulates abstract semantic objects.

When people adopt this approach of treating data structures in memory as "abstract" objects, they implicitly take the assumption that each object is an independent entity, i.e. each object has an identity that is independent of the value that is stored in the object; for an object to be shared among other complex objects, the object must be shared in its entirety; fields "belong" to objects — updating a field of one object does not affect any fields in any other objects; a complex object has its internal structure (or "shape"). The "shape" of a object is determined by what objects it has as its components and the shape of those objects. The "shape" of an object can only be changed by replacing some of its components with a different object, i.e. updating some "link" field.

These assumptions of programmers are justified, because compilers and runtime systems of programming languages have been designed to provide the necessary abstraction to allow them to think this way. For example, in a typical C implementation, although each object is no more than a set of memory cells in the linear address space, the C compiler and the runtime system guarantee that:

(1) memory cells from different objects never intersect; (2) different fields of a C data structure are mapped to distinct memory locations; (3) an object can have components, by recording references (pointers) to other objects in appropriate memory cells.

However, in the context of Rockwell challenge, the programs that we need to reason about are low level programs. The assumptions about having correct high-level language compiler and runtime system are absent. In fact, solving the Rockwell challenge contributes to acquire a precise understanding about what is implicitly expected from a correct high-level language compiler and its runtime system.

If we look at the Rockwell challenge problem [1] from this perspective, the big picture of any solution is to devise an effective methodology for configuring ACL2[4] to "understand" such key notions as object identity, object sharing and the "shape" of a data structure. The ultimate goal is to allow programmers to focus on the internal logic of their programs, instead of spending time reasoning about how objects are represented.

In this paper, I present my solution to the Rockwell challenge as an attempt to recover such notions as "data cells", "link cells", and objects being "well-formed" and distinct entities, so that reasoning about dynamic data structures in ACL2 can be made easier.

2 The Original Challenge Problem

Greve and Wilding describe the challenge problem in [1]. They were reasoning about operations on complex and pointer-rich data structures from a certain microprocessor microarchitecture model. This challenge problem summarizes some essential aspects of the difficulty they were facing.

In this simplified "real world" problem, they introduce two kinds of recursive data structures. These data structures have a low level representation as a set of memory cells from a linear address space. Greve and Wilding define operations that update these recursive data structures during a traversal of the structure. The a-mark-objects is such an operation:

```
(defun a-mark-objects (addr n ram)
  (declare (xargs :measure (nfix n)))
  (if (zp n) ram
   (if (zp addr) ram
    (let ((ram (s addr (+ (g addr ram) (g (+ 2 addr) ram)) ram)))
         (a-mark-objects (g (+ addr 2) ram) (1- n) ram)))))
```

where the (g addr ram), (s addr ram) operations correspond to read and write operations to the memory cell at addr in ram.

We can see from the definition of a-mark-objects that the actual path followed during a traversal depends on the *current* value stored in the address (+ 2 addr) when the update-on-the-fly process encounters the node at addr. On the other hand, a-collect, as used in the challenge to specify properties, traverses the structure and collects a list of memory cells for representing a recursive structure — before any update.

The first challenge is to show that if the nodes in this linked recursive data structure do not overlap, i.e. each node occupies a distinct range of memory cells, operation a-mark-objects never changes values of any memory cells that are not used in representing the initial object.

```
(defthm rd-over-a-mark-objects
  (let ((list (a-collect ptr n ram)))
    (implies (and (not (member addr list))
                  (unique list))
             (equal (g addr (a-mark-objects ptr n ram))
                    (g addr ram)))))
```

The (unique (a-collect ptr n ram)) hypothesis is used to characterize the above assumption that nodes in the recursive data structure do not overlap. However, we will see that this characterization is only a "convenient" approximation. In fact, the uniqueness requirement of a-collect is quite restrictive. It prevents many kinds of *object sharing.* An object cannot be pointed to by more than one pointer. We can show that any object, which satisfies this requirement, can only be of "tree" shape. Operations for updating more general data structures such as graphs, including any circular linked list, cannot be studied under this uniqueness assumption. In a later part of this paper, I will explain my solution to allow reasoning about such data structures by replacing the concise "uniqueness" requirement with a more elaborate but less restrictive hypothesis. That will serve as an answer to the 4th question posed by the challenge problem in [1].

In establishing this first property, the major difficulty lies in how to best utilize the uniqueness assumption. Because of the update-on-the-fly nature of a-mark-objects, the recursion pattern of the update operation can be very different from that of a-collect.

The next property to establish is a property about compositions of different operations on independent objects. The property says that the composition of different update operations does not affect the value of a memory cell if the cell is not used in representing any of the objects, and objects are "well-formed" and "independent" (as characterized by the uniqueness hypothesis). In particular we need to show:

```
(defthm read-over-bab
 (let ((list (append (b-collect ptr1 n1 ram)
                     (a-collect ptr2 n2 ram)
                     (b-collect ptr3 n3 ram))))
  (implies (and (not (member addr list))
                (unique list))
         (equal (g addr (compose-bab ptr1 n1 ptr2 n2 ptr3 n3 ram))
                (g addr ram)))))
```

Here compose-bab represents a composition of three operations, a B-type followed by an A-type, which in turn followed by a B on three objects. The property that if the three objects satisfying the constraint, updating them does not has any affect on memory cells outside the objects.

A subsequent challenge here is to design a general approach that can prove similar properties for arbitrary compositions of many different operations. In this paper, I only present my first attack on the original problem in detail, and in section 4, I briefly explain my approach of generalizing my method to deal with different kinds of operations and different data structures.

The difficulty I encountered in proving this property was not quite related to the problem itself so much as it was related to configuring ACL2 to reason about sets: reasoning about subset relations, set union, and set intersection. I will comment more on this point later.

The third property is to show certain operations on distinct "well-formed" objects commute. The key lemma for establishing this property is to characterize the set of cells that are either read or written by the operation.

More details are in the next section, where I present my proofs of the above three properties and explain the intuitions behind the approach.

3 Solution

Before I go into the details of explaining how I prove the three properties in the challenge problem, two key observations are explained.

The first key of my solution is recognizing that when we update non-link cells in the representation of an object, the structure (or "shape") of the object does not change. When the "shape" of an object does not change, many important facts are available about the data structure, such as that the set of memory cells used to represent the object remains the same. I characterize the "shape" of an object, by introducing an equivalence relation on different memory configurations with respect to an object. Updates to non-link cells preserve the structural equivalence.

Another key aspect is that I adopted an approach for reducing the "dynamic" nature of updating during the traversal to a more "static" view, in which, update-on-the-fly is converted into updates to a sequence of locations. Under appropriate hypothesis, the sequence is determined by a read-only traversal of the original data structure. After all, from the view of the memory component, any operations and their arbitrary compositions are "equal" to a corresponding update sequence. I introduce operations that collect all addresses that would be modified into a sequence of updates. I also define a separate apply operation to actually update the data structure. This approach allows me to talk about the cells being read or written explicitly. It simplifies the reasoning about interactions/interferences between operations. Applying different composition operations becomes applying a certain permutation of the sequences of the updates.

In the rest of the section, I explain my solution to the challenge problem in more detail. The proof script is available online at [3].

3.1 Task I: Read over A-mark-objects

The first task is to show: "the read of an arbitrary address location outside of the data structure modified by a-mark-objects is not affected by a-mark-object" — under the "uniqueness" assumption [1].

We define:

```
(defun collect-A-updates-dynamic (rc)
  (let ((n (n rc))
        (addr (addr rc))
        (ram  (ram rc)))
    (if (zp n) nil
     (if (zp addr) nil
      (let*
       ((rc1 (rc-s addr (+ (g addr ram) (g (+ 2 addr) ram)) rc))
        (ram (ram rc1))
        (addr (addr rc1))
        (n    (n    rc1)))
       (append
         (list addr)
         (collect-A-updates-dynamic
          (make-ram-config (g (+ 2 addr) ram) (1- n) ram))))))))
```

This function is not very different from a-mark-objects, except that it takes an rc (*RAM configuration*) — a 3-tuple that records the object reference addr, the depth n, and the ram. Instead of returning a modified ram, it returns a list of A-type nodes to be updated.

We will explain more about the motivation for grouping addr, n and ram into a rc (*RAM configuration*) later. Briefly, it is suggested by the observation that properties of an object in a high-level language description can be naturally mapped to properties of its corresponding equivalent class in RAM configurations. That is properties of an object are really properties of the corresponding equivalent class (for most properties that we are interested in).

With the above definition of collect-A-updates-dynamic, it is easy to prove

```
(defthm a-mark-objects-alt-definition
  (equal (a-mark-objects addr n ram)
         (apply-a-updates (collect-a-updates-dynamic
                            (make-ram-config addr n ram))
                          ram))
  :rule-classes :definition)
```

Now the remaining task is to show if addr is not a member of memory cells that represent the initial object, it is not among the cells that will be updated.

```
(implies (not (member addr (a-collect-1 rc)))
         (not (member addr
                (a-updates-w (collect-a-update-dynamic rc)))))
```

where a-collect-1 is similar to a-collect, except it takes only one input rc, which encodes three parameters for the latter. The function a-updates-w converts a sequence of A nodes for updating into a sequence of memory cells that will actually be written to.

Unfortunately, the above property is not true without an additional hypothesis. To traverse an object, collect-A-update-dynamic needs to interpret the *current* value stored in certain fields as links to follow during the traversal. Because the procedure collect-A-update-dynamic updates fields along the way, if we do not have an suitable hypothesis, it is possible that the values stored in the link cells can be changed, and the traversal may take a different path, i.e. "shape" of the object has been changed. As a result, although the memory cell at addr is not used to represent the original object, it may still be possible for it to be among the cells that represents the new objects which are transformed from the initial object.

For the above to be true, we need to capture the essence of an object's structure ("shape") being unchanged. We also need to characterize under what condition, an update does not change the "shape".

For the former, we define the important notion of two RAM configurations being structurally equivalent. Intuitively, two RAM configurations are *structurally equivalent* with respect to an object, if and only if the representation of the objects in the two configurations only differ in their data fields. Many implications of the "shape" being the same are expressed as congruence rules for equivalent RAM configurations. For example, the set of cells used to represent an object in structurally equivalent configurations are the same; and the set of link cells of two structurally equivalent objects are the same.

For the latter, to characterize under what conditions, an update will preserve the structural equivalence of the RAM configuration, I prove the following theorem.

```
(defthm set-non-link-cells-collect-equal
  (implies (not (member x (a-collect-link-cells-static rc)))
           (struct-equiv-A-ram-config (rc-s x v rc) rc)))
```

where a-collect-link-cells-static returns all cells used to store links in the object. The key observation is that as long as an update does not change a link cell of an object, the object's "shape" will not change. Here (rc-s x v rc) is analogous to (s x v ram), which set the value at x to v in RAM configuration rc.

Using the above result and the observation that structure equivalent ram has same set of link/data cells, I proved the following by inducting on collect-A-update-dynamic:

```
(defthm not-overlap-implies-collect-a-update-dynamic-equal-static
  (implies (not (overlap (a-collect-data-cells-static rc)
                         (a-collect-link-cells-static rc)))
           (equal (collect-a-updates-dynamic rc)
                  (collect-a-updates-static rc))))
```

where collect-a-updates-static is used to traverse the original structure and collect encountered A-nodes for updating. Different from the the corresponding "dynamic" version, it does not update the structure on the fly. It is easy to show (a-update-w (collect-a-updates-static rc)) is a subset of (a-collect-1 rc), i.e. if addr is not a member of the latter, it will not be in (a-update-w (collect-a-updates-static rc)).

I also established the following:

```
(defthm unique-implies-no-overlap
  (implies (unique (a-collect-1 rc))
           (not (overlap (a-collect-data-cells-static rc)
                         (a-collect-link-cells-static rc)))))
```

After proving the above lemmas, we are in a position to prove the main theorem:

```
(defthm rd-read-over-a-mark-objects
 (let ((list (a-collect ptr n ram)))
   (implies (and (not (member addr list))
                 (unique list))
          (equal (g addr (a-mark-objects ptr n ram))
                 (g addr ram)))))
```

Using the (unique (a-collect addr n ram)) condition, we know that the data cells and link cells of the object at ptr in ram do not overlap. Thus a-mark-objects can be reduced to apply a sequence of updates which collect-A-static-updates returns. Furthermore, using the hypothesis that addr is not used to represent the original object, we conclude the cell at addr is not updated.

As mentioned earlier, the hypothesis (unique (a-collect ptr n ram)) used in the original challenge problem is quite restrictive. Although such a hypothesis succinctly captures a sufficient requirement for nodes being independent, it only allows us to reason about update operations on "tree" shaped data structures.

In my approach, I have used a more elaborate hypothesis that talks about data cells and link cells being non-overlap, which does not have the above limitation. Key properties of various update operations are proved with respect to this new hypothesis. In some sense, the non-overlapping property captures the essential (or at least a much weaker) condition ensuring that a class of update operations are shape preserving.

Under this non-overlapping hypothesis about data cells and link cells, dynamic update operations can be reduced to simple operations of applying a sequence of statically collected updates. By statically collected updates, I mean that the locations of memory cells for updating are "collected" by traversing the original representation of the object. There are no "dynamic" updates to the object during the traversal. This makes it easier to prove properties of these operations.

So far, my approach can only be used to reason about the properties of operations that only changes the data fields of a *well-formed* object, whose link cells and data cells do not overlap. Every operation preserves the "shape" of a well-formed object. It is conceivable that one can extend the approach to reason about operations that update the link cells to form new structures — by studying properties of link cells and data cells of resulting structures. However this extension has not been done yet.

There is a subtlety in my approach of reducing on-the-fly updates into a sequence of statically determined updates. It is concerned with value dependency between different write operation.

In general, the value being written by a latter write operation may depend on a value being written by an earlier operation. In my approach, because only addresses of the to be updated celled are collected, while values to be written are still computed on the fly, i.e. the value for a latter write operation can dependent on a previous write operation. My approach can accommodate any value dependency issue. One can imagine a different approach of reducing the on-the-fly updates, in which, value dependency could be a problem. In this slightly different approach, both the address and the value to be written are recorded together to be statically applied in batch. While this latter approach is still applicable to the challenge problem in its concrete form, this approach faces serious limitations. The reduction can not be conducted when there are value dependency between different write operations.

In fact, in my work to generalize the approach [2], there are some efforts to characterize value dependency between different write operations. The exported properties of the encapsulated function new-field-value in [2] is a first attempt.

3.2 Task II: Read over compose-bab

We need to show

```
(defthm read-over-bab
 (let ((list (append (b-collect ptr1 n1 ram)
                     (a-collect ptr2 n2 ram)
                     (b-collect ptr3 n3 ram))))
  (implies (and (not (member addr list))
                (unique list))
   (equal (g addr (compose-bab ptr1 n1 ptr2 n2 ptr3 n3 ram))
          (g addr ram)))))
```

This is to prove "the read of an arbitrary address location outside of the union of the three data structures modified by compose-bab is not effected by compose-bab" [1], where compose-bab is a composition of three a-mark-object style operations.

```
(defun compose-bab (ptr1 n1 ptr2 n2 ptr3 n3 ram)
  (let ((ram (b-mark-objects ptr1 n1 ram)))
    (let ((ram (a-mark-objects ptr2 n2 ram)))
      (let ((ram (b-mark-objects ptr3 n3 ram)))
      ram))))
```

The proof of the property is straightforward after one can successively reduce each a-mark-object style operation into operations such as (apply-A-updates (collect-A-updates-static rc)), under the "uniqueness" assumption.

I followed the same strategies in the first proof. First, I prove that compose-bab is equal to a simple operation that applies a particular sequence of updates.

```
(defthm equal-compose-bab-apply-bab
 (equal (compose-bab addr1 n1 addr2 n2 addr3 n3 ram)
        (apply-bab-updates (collect-bab-updates-dynamic
                             addr1 n1 addr2 n2 addr3 n3 ram) ram)))
```

where, collect-bab-updates-dynamic is defined as

```
(defun collect-bab-updates-dynamic (ptr1 n1 ptr2 n2 ptr3 n3 ram)
  (let* ((rc1 (make-ram-config ptr1 n1 ram))
         (rc2 (make-ram-config ptr2 n2
                         (apply-B-updates
                               (collect-B-updates-dynamic rc1)
                               (ram rc1))))
         (rc3 (make-ram-config addr3 n3
                         (apply-A-updates
                               (collect-A-updates-dynamic rc2)
                               (ram rc2)))))
  (list   (collect-B-updates-dynamic rc1)
          (collect-A-updates-dynamic rc2)
          (collect-B-updates-dynamic rc3))))
```

then I prove under the "uniqueness" condition,

```
(defthm unique-equal-collect-dynamic-to-static
  (implies
   (unique
    (append (b-collect-1 (make-ram-config ptr1 n1 ram))
            (a-collect-1 (make-ram-config ptr2 n2 ram))
            (b-collect-1 (make-ram-config ptr3 n3 ram))))
   (equal (collect-bab-updates-dynamic ptr1 n1 ptr2 n2 ptr3 n3 ram)
          (collect-bab-updates-static  ptr1 n1 ptr2 n2 ptr3 n3 ram)))
```

Using the subset relation between set of cells to be updated (as determined with collect-bab-updates-static) and the union of cells that represent the three initial data structures, we can conclude that addr is not among the cells to be updated and its value does not change.

With the supporting lemmas introduced in the first proof, the major challenge of this proof is more about how to configure ACL2 to reason about set operations than introducing new concepts that capture the properties of "mark" operations.

For example, it is hard to configure ACL2 to use known subset relations to reason about set intersections. To prove the above theorem unique-equal-collect--dynamic-to-static, I need to prove many lemmas about updated cells in one object not intersecting with the link cells of another object. Despite the fact that I proved many subset relations between different types of cells in different objects, I did not succeed in configuring ACL2 to discover the non-intersection properties I need. As a workaround, I proved the following theorem

```
(defthm overlap-subset
  (implies (and (overlap a c)
                (subsetp a b)
                (subsetp c d))
           (overlap b d)))
```

This theorem is powerful for using as a `:use` hint, however, ACL2 cannot use it effectively because it cannot find the proper bindings for free variables.

One particular aspect of the challenge problem makes this weakness of lacking/not using a good ACL2 "book" about set theory even worse. As we know, the challenge involves two concrete data structures and the corresponding concrete operations. This results in a quadratic number of theorems (of essentially the same form) needing to be proved. Each of them needs an explicit `:use` hint. One concrete version is

```
(defthm unique-implies-no-overlap-A-data-B-link
  (implies (unique (append (a-collect-1 rc1)
                           (b-collect-1 rc2)))
           (not (overlap
                  (a-updates-w (collect-A-updates-static rc1))
                  (B-collect-link-cells-static rc2))))
  :hints (("Goal" :in-theory (disable  overlap-subset)
           :use ((:instance overlap-subset
                            (A (a-updates-w
                                (collect-A-updates-static rc1)))
                            (b (a-collect-1 rc1))
                            (c (b-collect-link-cells-static rc2))
                            (d (b-collect-1 rc2))))))
  :rule-classes :forward-chaining)
```

If we have more data structures and more operations, this approach is not feasible.

One potential solution is to carefully structure an ACL2 "book" about sets, allowing these theorems to be proved without needing a manual hint. This approach may even eliminate the need to write down these lemmas explicitly. However this solution implies such repeated rewriting would be needed, which may have an impact on the total proving time.

The other solution for simplifying the proof is to generalize the problem so that only one theorem is needed for characterizing all cases. I will discuss my generalization in section 4, which is still a work-in-progress.

3.3 Task III: a-mark-over-b-mark

This part of the challenge is to prove under the "uniqueness" assumption, operations on independent objects commute.

```
(defthm a-mark-over-b-mark
  (implies
   (let ((list (append (a-collect ptr1 n1 ram)
                       (b-collect ptr2 n2 ram))))
     (unique list))
   (equal
    (a-mark-objects ptr1 n1 (b-mark-objects ptr2 n2 ram))
    (b-mark-objects ptr2 n2 (a-mark-objects ptr1 n1 ram)))))
```

The proof is fairly easy, after we proved the second part of the challenge. In the proof development of the previous property, we proved, under the "uniqueness" condition, the following lemma:

```
(equal (a-mark-objects ptr1 n1 (b-mark-objects ptr2 n2 ram))
       (apply-a-updates (collect-A-updates-static
                         (make-ram-config ptr1 n1 ram))
          (apply-b-updates (collect-B-updates-static
                            (make-ram-config ptr2 n2 ram) ram))))
```

which marks the B object first, and a similar lemma for marking the A object first.

After we could reduce the composition of two "marks" to the composition of two "apply updates", we need to prove under the "uniqueness" hypothesis

```
(equal (apply-B-updates l2 (apply-A-updates l1 ram))
       (apply-A-updates l1 (apply-B-updates l2 ram)))
```

The fundamental reason for why two operations can commute is that both operations never change any cell which is read or written by the other operation. I introduce the concept of the cells which are either read or written during a sequence of updates. I prove

```
(defthm apply-update-ram
  (implies (not (overlap (a-data-cell-w-r l1) (b-data-cell-w-r l2)))
           (equal (apply-B-updates l2 (apply-A-updates l1 ram))
                  (apply-A-updates l1 (apply-B-updates l2 ram)))))
```

Combining the above two results and using the subset relations between sets such as (a-data-cell-w-r (collect-a-updates-static rc)) and (a-collect-1 rc), I prove the main goal as listed in the beginning of this section.

4 Generalization

It is an observation that if we "map" the properties in this challenge back to properties of some high-level language routine for updating objects, these properties are really trivial. In fact they are in some sense, *inherent* and *assumed.*

Thus, the challenge of establishing the above three properties is only a tool of Greve and Wilding to illustrate the importance of implicit assumptions, which people often take for granted when they reason about high-level programming languages. These assumptions include the basic concept of object identity, as well as more complicated concepts like the independence between objects. Without recapturing these implicit assumptions properly, it is hard to prove those properties. So the real challenge problem is how to best capture these implicit assumptions.

Examining the data model that we have for high-level programming languages, we ask the "meta" question: why these properties are "trivially" true and why a proof may exist. One may realize the answers to the above questions are not dependent on the concrete structures nor the concrete operations. Instead, those answers rely on how we map the data model of the high-level language to a low abstraction level model.

This observation suggests there is a more general problem that we can formalize by focusing on how we map an abstract data model to a low level model, i.e. what "invariant" we need to preserve for the mapping, so that properties of high-level objects are preserved.

In the rest of section, I first present how I formalize the concept of set of objects in a linear address space. I also introduce what constraints are necessary for this definition to match the concept of objects in high-level languages. Then I briefly show how I model update operations and reason about them. I formulate some generalized properties that correspond to the original version of properties. I will also present my partial result towards establishing them. Many formulations are inspired by Moore's solution to a different generalization of the same challenge problem, which he presented in a local ACL2 seminar [5]. My partial result is also available online [2].

In the generalized problem, data structures are no longer concrete, a `map` is used to associate a type name to a description of the structures of values of that type. This concept of using "map" to hide away the concrete data structure is borrowed directly from J Moore's formalization in [5]

I introduce the concept of a generalized *RAM configuration*. A RAM configuration records a list of *generalized pointers* that correspond to objects in a high-level language description. Besides recording the address of an object, a generalized pointer also records the type of the pointer and the depth of the object (i.e. `n` in the original properties). I define a structural equivalence between RAM configurations. It is used to capture the concept of two sets of objects that have the "shape". For two RAM configurations to be structural equivalent, every object in the one RAM configuration is structure equivalent to a corresponding object in the other RAM configuration.

A sequence of updates to a set of objects is represented with a function that takes a RAM configuration and applies `mark`-style operations to each object described in the configuration. An update to each object is based on the type information of the object. The value written during the update is specified by an encapsulated function, which is slightly more general than J Moore's version in [5]. Details about the required properties of this function are in [2].

I introduce an extra constraint for a "well-formed" RAM configuration so that a RAM configuration can really be mapped back to a set of more abstract objects in the high-level language. The constraint requires that in a "well-formed" RAM configuration, link cells collected from the objects do not overlap with data cells from those objects.

I proved similar properties for the generalized version. For example, I proved that updating a non-link cell preserves the structural equivalence. I proved that a mark operation on one object from a "well formed" configuration can be reduced to an operation that applies a certain sequence of updates. The sequence can be determined by crawling through the initial structure without updating the structure.

I proved the property that

```
(defthm g-over-mark-one-objects
  (implies
    (and (not (overlap
                (collect-data-cells-node typ ptr n ram map)
                (collect-link-cells-node typ ptr n ram map)))
         (not (member addr
                (updates-2-ws (collect-updates-static-node
                                     typ ptr n ram map)))))
    (equal (g addr (mark-node typ ptr n ram map))
           (g addr ram))))
```

which corresponds to the first property in the original problem.

The second property in the original problem can be considered as instances of the following theorem

```
(defthm g-over-mark-objects
  (implies (and (not (overlap (collect-data-cells rc)
                              (collect-link-cells rc)))
                (not (member addr
                             (updates-2-ws
                               (collect-updates-static rc)))))
           (equal (g addr (mark-all rc))
                  (g addr (ram rc)))))
```

where collect-updates-static, collect-data-cells, collect-link-cells crawl the each objects pointed by the "generalized pointers" in the rc.

I intend to prove a property which says, under a slightly stronger hypothesis (similar to the hypothesis in my proof for the third property in section 3), two mark-all operations return the same state when the lists of objects mentioned in two well-formed RAM configurations are a permutation of each other.

More details about my generalization of the problem are described in [2]. As an important note, this generalization is still a work in progress. A recent bug was discovered. My formulation of the structurally equivalent RAM configuration does not handle the concept of NULL pointers correctly. Operations will treat NULL pointers like just any other pointers. Implication of this modeling error has not been investigated fully.

5 Conclusion

In this paper, I present my solution to the original Rockwell challenge problem. The focus is to recover the notion of objects being independent entities. The key of my approach is to reduce any dynamic update operation — update during traversal — to a simple operation that applies a sequence of statically collected updates. By statically collected updates, I mean that the locations of memory cells for updating are "collected" by traversing the original representation of the object. To achieve this reduction, I characterized the condition under which the "shape" of an object remains unchanged.

The big picture of this challenge problem is to devise effective ways to reason about dynamic data structures in a low-level representation. My approach of recovering the notions of link cells and data cells for each individual data structure has worked for the concrete challenge problem, however it has its limitations. It becomes tedious when we have many different data structures and even more types of operations. My work to generalize the approach is still in progress.

References

[1] D. Greve and M. Wilding. Dynamic datastructures in ACL2: A challenge. http://hokiepokie.org/docs/festival02.txt, November 2002.

[2] H. Liu. Work in progress of generalizing the Rockwell Challenge. `http://www.cs.utexas.edu/~hbl/pub/ws/sol2.lisp`.

[3] H. Liu. Solutions to rockwell challenge. `http://www.cs.utexas.edu/~hbl/pub/ws/sol1.lisp`, December 2002.

[4] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-aided Reasoning: An approach.* Kluwer Academic Publishers, 2000.

[5] J S. Moore. A generalization of the Rockwell Challenge. `ftp://ftp.cs.utexas.edu/pub/moore/general.lisp`, December 2002.