Memory Taggings and Dynamic Data Structures

J Strother Moore

Department of Computer Sciences
University of Texas at Austin
Taylor Hall 2.124
Austin, Texas 78712
moore@cs.utexas.edu

Abstract. The aim of this paper is to help formal methods practitioners deal with the formal proofs of elementary properties of data structures represented in a RAM-like memory. This problem commonly arises when dealing with proofs of microcode, machine code, assembly code and compiler correctness. We formalize a version of the problem in ACL2 and discuss how to prove several important properties. In particular, our data structures are typed but the type information is not recorded in the representation. Algorithms for manipulating the representation implicitly respect the declared types. We support arbitrarily many record types within the RAM. Each field of each record is declared to contain either a data item or a pointer to a record of some fixed type. It is possible for records to be overlaid on other records and for the pointer structures to be circular. We define an elementary generic algorithm that recursively "chases" pointers and writes "arbitrary" values to the fields declared to contain data. The formal expression of the algorithm is complicated by both mutual recursion and reflexivity, which present technical challenges. We then consider certain "obvious" properties, especially the property that the algorithm does not modify addresses that are "unreachable" from the initial pointer. Because records may be overlaid in the RAM — causing a given address to be treated sometimes as data and other times as a pointer — these obvious properties are true under only certain restrictions. If the restrictions are stated in terms of reachability the proofs are complicated, if not blocked, by the technical issues above. To prove the properties we generalize them by introducing "memory taggings" and formalizing the idea that a given run of the algorithm treats addresses in a way consistent with some typing scheme. We then show how to relate the original reachability hypotheses to the existence of consistent taggings.

1 Background

It is common in programming practice to represent data structures in a RAM-like memory. Indeed, this is done so often that virtually all programmers are aware of certain obvious properties, e.g., that if the addresses reachable from one data structure are disjoint from those reachable from another, operations on the first data structure are independent of those on the other. Of course, care must be taken to make such "folklore" both precise and true. Doing so often makes it non-trivial to prove.

With this phenomenon in mind, Greve and Wilding [10] describe a challenge for the ACL2 community. They state that

> Arrays, structures, and stacks are common data structures that can be found in some form in nearly every useful software system. The utility that such structures provide in implementing complex systems, however, hinges on the often overlooked assumption of independence. What the programmer often takes for granted - that modifying structure X will not change the value of structure Y - becomes a significant burden when the formal correctness of the final implementation is considered in the context of a linear address space.

> Several verification projects with which we are familiar have faced this challenge, including Moore's Piton compiler [17], Bevier's KIT proof [2, 3], and Yu's 68020 code proofs [4] [citations added by current author]. In each project the disjointness of data structures was described and verified as part of the larger verification effort. However, the proof architecture in each of these projects involves blocks of data that resided within distinct ranges rather than arbitrary, dynamic structures. This approach is not feasible for the kinds of complex, pointer-rich data structures we encounter in proofs about microprocessor microarchitectures.

Greve and Wilding go on to describe the implementation of two specific record structures and specific traversal algorithms that modify certain fields in fixed ways. Structures of their type `A` contain data in the first and fourth fields and pointers to structures of type `A` in the second and third fields. Structures of type `B` contain data in the third field and pointers to structures of type `B` in the first and second. The algorithm for marking `A` structures sums the first and third fields into the first and recursively marks the `A` structure indicated by the third field. The algorithm for marking `B` structures clears the third field and recursively marks the `B` structure indicated by the first field. Such specific structures and algorithms are typically encountered in applications.

They then challenge the reader to prove that these algorithms do not interfere with one another under certain assumptions about the disjointness of the sets of pointers.[1]

One might hope to solve the problem once and for all, by characterizing a general data representation scheme of which most implementations are merely instances. The folkloric properties of those implementations could then be obtained by instantiation of the general scheme, saving the practitioner from developing yet another specific proof "from scratch." But generality imposes its own burdens. A scheme general enough to accommodate the variety of data layouts in RAM-like memories might have so many obscure special features that the practitioner may prefer to develop "from scratch" the proof of the particular algorithm in question. One might then hope merely that the lessons learned in developing the more general scheme might save the practitioner some effort.

This paper presents a fairly general scheme that captures many data layouts. Our generic traversal algorithm may write to any data field encountered in the exploration of the data structure but it does not write to any field declared to be a pointer. It is actually possible to cause the algorithm to modify a pointer field by overlaying the containing record with a record declaring the corresponding address to be data. Our theorems contain hypotheses precluding such usage. But the disparity — between what the algorithm can do versus how we intend to use it — is a source of difficulty in the analysis. Some practitioners will be able to use our results by instantiation. The method by which we proved the properties of the general scheme are also discussed, to help the practitioner who finds our scheme too limited.

Formal specification and verification of pointer manipulation algorithms has been of concern since the 1970s. In 1972, Burstall proposed techniques for verifying the destructive modification of data structures by modeling the heap explicitly [5]. One of the first mechanized proof systems designed to address some of the problems here was the interactive program verifier presented by Deutsch in his 1973 U.C. Berkeley PhD dissertation [7]. The Stanford Pascal verifier also included support for data structure manipulation [15], as did the State Delta Verification System (SDVS) [14]. For example, SDVS had mechanisms for dealing with the frame problem, so that one could infer that parts of memory were unchanged if they were in memory graph partitions disjoint from where writes were occurring.

---

[1] The challenge was issued to the ACL2 research group in Austin and formed the basis of the December, 2002, "ACL2 Fest" in Austin, where various researchers, including Greve and Wilding, presented solutions. This paper presents my solution.

Built-in techniques to detect aliasing and overlapping allocation are seen today in lightweight symbolic analysis tools such as SLAM [1] and Bandera [8]. But our interest is in proving theorems about pointer manipulation programs, where these techniques must be either derived as theorems or proof rules or integrated at the meta-level.

The problem is frequently confronted in code verification settings. Greve and Wilding [10] mention such work as [17, 2–4] where the primary focus is the proof of correctness of machine code. Proofs of garbage collection algorithms have also grappled with many of the problems noted here [23, 22, 9, 11].

The problems have also arisen when formalizing the semantics of programming languages providing pointer manipulation [6, 20, 16]. The work cited above generally grapples with the problem of data structures allocated in RAM-like memory and the correctness of implemented algorithms. Norrish [20] uses HOL to verify some proof rules for code fragments that manipulate pointers. Mehta and Nipkow [16] formalize a simple imperative programming language that contains pointers, they derive general proof rules as theorems in higher-order logic, and they use their rules to mechanically prove the correctness of a version of the Schorr-Waite graph marking algorithm. Their work is mechanically checked with Isabelle/HOL. In their correctness proof (but not their general proof rules) they deal with a sophisticated algorithm manipulating one particular data structure, while we deal with an elementary algorithm that interprets an declaration of many data types. More significantly, overlapping allocations are explicitly ruled out in [16].

Some of the work above, especially that dealing with garbage collection, grapples with the problem of de-referencing pointers modified during the course of the algorithm. Our algorithm does not provide this feature (though overlapping allocations can cause similar effects) and the theorems we prove about the algorithm preclude it.

To our knowledge, the most direct attack on the problems addressed here is the introduction of Separation Logic [21] of Reynolds (inspired in part by the work of Burstall). Reynolds proposes a logic designed to ease the task of reasoning about the separation of storage into disjoint components.

Our goals are much more limited than those of Separation Logic. Rather than develop a logic specifically to address storage concerns, we explore how to use one particular logic, ACL2, to reason in that area. We do this because ACL2 is already used in microarchitectural and machine-level code proofs, so the problems we face and solve here have immediate applicability to the ACL2 community. More broadly, the present paper identifies some of the obstacles encountered when reasoning about dynamically allocated pointer-based data structures and explains how and why our formal notion of "memory taggings" solves some of these problems.

2 Discussion

We support an arbitrary number of data types, each with an arbitrary but fixed number of fields. Each allocated instance of a data type is laid out in a contiguous block of addresses in a RAM. The type declaration declares each field to contain either "mutable" data or an "immutable" pointer to a record of some fixed type. In principle, any address in a RAM may be the target of a write, and any such write would modify the contents. Our algorithm explicitly writes only to addresses declared to be data fields by the type declaration of the containing record. We do not constrain the allocation scheme. In principle, we allow circular structures and records that are overlaid upon the same memory locations.

We define an elementary generic recursive traversal algorithm that modifies all the mutable data fields reachable (in a given recursion depth) from a given pointer.[2] We call this *marking* the data structure, though it may carry out a more general computation at each node than just setting a flag.

The algorithm is generic in the sense that the value written to each field is specified by a function that is constrained to be insensitive to changes in fields outside the current record. Note that this allows the "new value" for a given field to be the old value and hence the function can be functionally instantiated to produce algorithms that modify only some of the reachable fields. Typically, new field values might be functions of some fixed data and the current values of other fields within the current record. The "fixed data" could, in principle, be a table describing the pre-computed new value to be written to each slot.

Our algorithm is an elementary recursive traversal algorithm. It is meant to resemble the many such traversal algorithms in everyday systems. By instantiating the constrained function used to compute the new values written to each slot, one can obtain a broad class of mundane modification algorithms. Its elementary and generic nature is what makes it interesting. By analyzing its properties and discussing the issues we hope to illuminate the obstacles to proving the correctness of many similar algorithms.

We prove three main theorems, each being a generalized version of a challenge theorem in [10]. Roughly speaking these three theorems state that (i) the marking algorithm does not change the contents of an address not reachable from the target pointer, (ii) that composed markings on several targets enjoy property (i) as well, and (iii) the markings of disjoint data structures may be commuted. In this paper we focus on (i) and (iii), since (ii) follows straightforwardly from (i).

Each of these main theorems involve hypotheses that check that each reachable address is reachable by a unique path. This is formalized in [10] by taking the list concatenation of the sequences of reachable addresses and then asserting that the list contains no duplicate elements. This is easily shown to be equivalent to checking that no reachable address is reached twice and that different data structures contain disjoint reachable addresses. Regardless of how it is stated, this restriction involves a static determination of the addresses reachable from a given target address. The focus on the uniqueness of reachable addresses across several marked data structures neatly rules out overlaid records within a single data structure, circular data structures, and shared components among the data structures to be marked. But we found this neat hypothesis too strong to permit some inductive proofs.

A simple analogy may be found in the theorem $a = b \rightarrow a \cup b = b$. (In this example, $a \cup b$ denotes list union as defined in ACL2 via `(union-equal a b)` and $a \subseteq b$, below, denotes `(subsetp-equal a b)`.) The hypothesis, $a = b$, is too strong to permit this to be proved by induction. Weakening the hypothesis to $a \subseteq b$ requires the introduction of a new concept but produces a theorem that can be proved by induction on a and from which the original theorem can be proved. The "neat hypothesis" on the uniqueness of reachable addresses must be weakened in an analogous way.

We encountered three main obstacles while trying to prove formally theorems about our generic algorithm.

Obstacle 1. The problem requires the definition of mutually recursive functions. For example, to mark the data structure indicated by a pointer one must call it recursively on each pointer in each record. The "iteration" over the arbitrary number of fields in a record is formally a mutual recursion.

[2] Following the convention used in [10], all the functions defined here take a recursion depth as a parameter and "chase" pointers only to that depth. All of our theorems have variable symbols in the depth positions. Since no data structure can have more depth than the finite size of of the RAM without being circular, this depth limitation is not an *a priori* restriction. In our informal discussions of our algorithms and theorems we often omit mention of the depth parameter.

Obstacle 2. Marking can, in principle, modify pointers and then chase the modified pointers, sending it into regions of the RAM one would think unreachable by the analysis of the initial unmarked RAM. How can this happen if the marking algorithm does not explicitly write to pointer fields? The same RAM address might be used both as a mutable data item (in one record) and as an immutable pointer (in another record), with both records reachable from the target. This may happen if records are overlaid in the RAM. A consequence of such overlaying is that the addresses actually reached during marking might be different from those "reachable" initially. Our hypotheses will rule this out, but we will have to prove that they do. The key lemma is that under certain hypotheses the statically reachable addresses in a marked RAM are those of the initial RAM.

Obstacle 3 . Marking is "reflexive" [18]: the RAM produced by one recursive call of the algorithm (e.g., to chase the first pointer) is given as input to a subsequent call (e.g., chasing the second pointer). Reflexivity does not occur in [10] because only one field is recursively chased, but it would have arisen had multiple, specific pointers been chased. In our setting the reflexivity is manifested through mutual recursion, since there may be an arbitrary number of pointers in a record and each is chased in the context of the marking done for the previous ones.

To understand why these seemingly technical points represent obstacles one must remember that formal proofs, by definition, involve formulas and the manipulation of formulas by syntactic rules. Furthermore these syntactic rules are by no means arbitrary; violations permit the deduction of falsehoods. Such issues as whether a theorem has hypotheses or how the variables are shared among the parts of a formula heavily influence the operation of these rules. Foremost among the rules when recursively defined functions are present is mathematical induction. To prove $p(n)$ by induction one typically proves a base case, e.g., $p(0)$, and an induction step, $n \neq 0 \wedge p(n-1) \rightarrow p(n)$. Of special note is that the induction hypothesis, $p(n-1)$, is an instance of the formula being proved. Frequently, in informal proofs, we violate this rule and take as inductive hypotheses formulas that are not exactly instances of the theorem we announced as the goal. In some cases such informal proofs can be turned into formal proofs but they require the explicit strengthening of the theorem so that it admits the necessary inductive instances. This is extremely common in inductive proofs: we must prove stronger theorems than we initially wanted. If proofs are to be formally checked, such inductive generalization often becomes the main focus of the effort. Such is the case here.

Now let us reconsider the three obstacles. The first, mutual recursion, complicates inductive proofs. If f and g (e.g., "mark this pointer" and "find each pointer in the record and mark it") are defined mutually recursively, then to prove a theorem about f by induction one generally needs lemmas about g and vice versa. Hence, when mutual recursion is involved, one often has to strengthen the desired theorem into a conjunction of theorems about all the function symbols in the mutually recursive clique. One way to attack this is to define a singly recursive function that takes an additional argument and mimics the definition of each function in the clique as that argument varies. For example, one might define $h(i, x)$ so that when $i = 0$ the definition mimics that of f and when $i = 1$ the definition mimics that of g. That is, if the definition of $f(x)$ in some case involves both f and g, then the definition of h in that case involves calls of h on both values of i. It is then easy to show that $f(x)$ is $h(0, x)$ and $g(x)$ is $h(1, x)$. Furthermore, one can state and prove general theorems about $h(i, x)$ that can be instantiated to be theorems "about" f or g. During the proofs of those theorems by induction, one can obtain induction hypotheses about both f and g by the choices for i assumed in induction. We call h the *flagged version* of f and g. This is the standard ACL2 approach to handling mutual recursion (see the documentation topic `MUTUAL-RECURSION-PROOF-EXAMPLE` in ACL2's online documentation [12]) and is the method we chose here.

The second obstacle, that certain overlays can cause marking to overwrite pointers and hence reach "unreachable" addresses, means we must add hypotheses to our theorems to restrict our attention to RAMs in which some overlays are prohibited. But proving an implication by induction means we must pay attention to what we call the "detachment problem." When an implication, $p \rightarrow q$, is to be proved by induction, the inductive hypothesis is some instance, $(p' \rightarrow q')$. We often act as though the hypothesis were simply q' and use it, and p, to prove q. But in fact the hypothesis is an implication and we must detach q' from that implication by proving p', typically using p. This is just simple propositional calculus: $((p' \rightarrow q') \rightarrow (p \rightarrow q))$ is equivalent to $(((\neg p' \wedge p) \rightarrow q) \wedge ((p \wedge q') \rightarrow q))$. In informal proofs, the first conjunct is often overlooked.

Suppose *collect* is the function that collects the list (not the set) of addresses reachable from a given pointer and *mark* is the function that returns a new RAM after marking from a pointer. Consider the key lemma mentioned in Obstacle 2, "under certain hypotheses the statically reachable addresses in a marked RAM are those of the initial RAM." This formalizes as an implication. The conclusion is of the form $\mathit{collect}(\ldots, \mathit{ptr}, \ldots, \mathit{mark}(\ldots, \mathit{ram}, \ldots), \ldots) = \mathit{collect}(\ldots, \mathit{ptr}, \ldots, \mathit{ram}, \ldots)$. The hypothesis must ensure that marking does not write and then chase the values of new pointers. The "natural" way to express this is that the list of addresses collected from *ptr* includes no duplicates. This is a hypothesis about $\mathit{collect}(\ldots, \mathit{ptr}, \ldots, \mathit{ram}, \ldots)$.

Attempts to prove this key lemma are thwarted by Obstacle 3, *mark* is reflexive. Any inductive argument about marking a RAM will require an inductive hypothesis about marking a (partially) marked RAM. That is, one of our induction hypotheses will be formed by replacing all occurrences of the variable *ram* by $\mathit{mark}(\ldots, \mathit{ram}, \ldots)$. Because one such occurrence of *ram* is in the hypothesis of the conjecture we are trying to prove by induction, we must deal with the detachment problem. We must show that there are no duplicates among the collection of reachable addresses in a partially marked RAM. But this leads us back to the key lemma itself.

The challenge theorems that caused us to identify this key lemma suffer this same blockage.

The way around these obstacles is a radical strengthening of the theorems we are trying to prove, achieved by weakening the hypothesis on the duplication of reachable addresses. We will replace some uses of the notion of reachability with the notion of a "tagging." A tagging allows us to say, formally, that each address is used consistently as data or pointer, but not both. Each strengthened theorem assumes the existence of a consistent tagging for the markings used in the theorem. We then prove that the reachability hypotheses of the original theorem imply the existence of a suitable tagging; for example, if each address is visited only once, then it is easy to see how to tag each address with a single type of use. We also show that two consistent taggings can be combined to produce a third, under suitable hypotheses about disjointness. This result then allows us to compose applications of the marking algorithm.

But recall our remark above "The focus on the uniqueness of reachable addresses across several data structures [10] neatly rules out overlaid records within a single data structure, circular data structures, and shared components among the data structures to be marked." The very neatness of the hypothesis made it difficult to find the strengthening. An irrational urge to preserve that neatness led us at first simply to replace the reachability hypothesis with the "consistent use" hypothesis. But the reachability hypothesis served many purposes; some of them, e.g., absence of circularity, had to be replaced by consistent use, while others, e.g., absence of shared components among two marked data structures, had to be teased out and stated explicitly.

To be more precise we must present the formulas. We will present our formulas in the formal syntax of ACL2, which is based on Common Lisp. Thus, instead of writing $f(x)$ and $h(0, x)$ we will write `(f x)` and `(h 0 x)`. We write in this formal notation to remind the reader that we are truly focused on the formalization of the concepts and proofs, not just on the intuitive ideas.

3 Outline

The rest of this paper proceeds as follows. In Section 4 we describe our formalization of the problem, including how we represent RAMs, how data structure layouts are declared, the marking algorithm, the notion of reachable addresses, and the top-level theorems analogous to those in [10]. In Section 5 we describe our use of taggings to capture the notion of consistent use and we strengthen the main theorems using that notion. In Section 6 we describe how to construct consistent taggings from suitable disjointness conditions on the reachable addresses, allowing us to prove the top-level theorems from our strengthened ones. In Section 7 we hint at a few omitted details. In Section 8 we recapitulate our strategy with reference to the problems discussed. The ACL2 script for the definitions and proofs described here is available at http://www.cs.utexas.edu/pub/moore/publications/memory-taggings/index.html.

4 Basic Concepts

To represent RAMs we use the Kaufmann-Sumners "records" book [13].[3] This book implements a notion of finite function or mapping. The domain and range of these finite mappings are the ACL2 universe of objects. In this paper, we refer to such mappings as "RAMs," the elements in the domain are called "addresses" and value associated with a given address in a RAM is called the "contents" of the address. Sometimes the contents of an address is called a "word" but it may be an arbitrary ACL2 object in our model. The records book defines two functions, `g` (for "get") and `s` (for "set") for manipulating RAMs. If `a` is an address and `ram` is a RAM, then `(g a ram)` – the application of the function symbol `g` to `a` and `ram` – denotes the contents of `a` in `ram`. Similarly, `(s a v ram)` denotes the ram produced from `ram` by (re-)associating address `a` with contents `v` and changing no other associations. The key theorem about these two functions is

```
(equal (g a (s b c ram))
       (if (equal a b)
           c
           (g a ram)))
```

The records book is very convenient precisely because it imposes no restrictions on the any of the variables above. However, in our use, addresses are integers. Sometimes the contents of an address is treated as another address (a pointer). To *dereference* or *chase* a pointer is to fetch its contents. The integer `0` is used as the *null pointer*. It is never dereferenced.

We allow the user to specify the layout of each type of data structure to be represented in the RAM. Each data structure has a fixed number of fields which are always allocated in contiguous blocks of addresses in the RAM. Each field is declared to be either data or pointer; in the latter case, the declaration describes the single type of data structure to which the pointer points. For example, a data structure of type `A` might have four successive fields, allocated to four successive addresses, which are declared respectively to be data, a pointer to a structure of type `A`, a pointer to a structure of type `B`, and data. The null pointer may be used as the contents of any pointer field, regardless of declared type.

Each data structure to be represented is described by a pair. The first component of the pair must be a symbol naming the data type, e.g., `A`, `B`, or `RB-TREE`, etc. The second component is a list

[3] In ACL2, a collection of definitions and theorems packaged for convenient re-use is called a *book*. The book in question is `books/misc/records.lisp` in the ACL2 distribution. Kaufmann and Sumners' use of the term "records" is different than ours and we avoid it here.

of length n, $(x_1 \ \ldots \ x_n)$ specifying the use of each of the n successive fields. If x_i is a symbol, the i^{th} field is a pointer to a record of type x_i (or the null pointer). Otherwise, the i^{th} field contains data. The list $(x_1 \ \ldots \ x_n)$ is called the *descriptor* for the type with which it is paired. The collection of all types and their descriptors is called the data type *declaration*.

```
((A . ("int" A B "int"))
 (B . (B "float" B)))
```

corresponds to a description of `A`-type structures as illustrated above and `B`-type structures as consisting of three words, the first and last of which are pointers to `B` structures and the middle being data. Recall that in our declarations, symbols such as `A` and `B` indicate that the fields are pointers to records of the named type, and non-symbols indicate data fields. The particular non-symbols used above, `"int"` and `"float"`, are string constants in ACL2. These strings are suggestive of restrictions on the contents but our model enforces no restrictions on data fields; our marking algorithm might write any object into such a field.

Here is our marking algorithm. Here, `typ` is some symbol indicating the type of data structure being marked, `ptr` is an address pointing to the first word in the data structure, `n` is a depth limit on how far we chase pointers to substructures and insures termination even if we have circular data structures, `ram` is the RAM, and `dcl` is the declarations describing the layout of all data structures.

```
(mutual-recursion
(defun mark (typ ptr n ram dcl)
  (let ((descriptor (cdr (assoc typ dcl))))    ; Get descriptor of typ.
    (if (zp n)                                 ; Stop if depth exhausted
        ram
      (if (zp ptr)                             ;  or if ptr is null
          ram
        (if (atom descriptor)                  ;  or if descriptor illegal.
            ram
          (let ((ram (s* typ ptr 0 ram dcl)))  ; Update all data fields of
                                               ;  this record.
            (mark-lst typ                      ; Then chase successive
                      ptr                      ;  pointers in this record
                      0                        ;  starting at the 0th.
                      (- n 1)
                      ram
                      dcl)))))))
(defun mark-lst (typ ptr i n ram dcl)
  (let* ((descriptor (cdr (assoc typ dcl)))    ; Get descriptor of typ,
         (slot-typ (nth i descriptor))         ;  type of this slot, and
         (i (nfix i)))                         ;  offset of slot in record.
    (cond ((zp ptr) ram)                       ; Stop if ptr null or i
          ((<= (len descriptor) i) ram)        ;  outside of block.
          ((symbolp slot-typ)                  ; If slot holds a pointer,
           (let ((ram (mark slot-typ           ;  chase and mark it, and
                            (g (+ ptr i) ram)
                            n ram dcl)))
             (mark-lst typ                     ;  then mark the rest of the
                       ptr                     ;  slots.
                       (+ 1 i)
```

```
                 n ram dcl)))            ; (See Note below.)
        (t (mark-lst typ                 ; If slot holds data,
                     ptr                 ;  then mark the rest of the
                     (+ 1 i)             ;  slots.
                     n ram dcl)))))
)
```

ACL2 must prove the termination of every function admitted to the logic. This is done by identifying an ordinal-valued measure that decreases in each call. ACL2's heuristics do not guess an appropriate measure above and the user must actually supply it. We have omitted the measure and refer the reader to the script.

Note: The recursive call of `mark-lst` described by the comment "*then mark the rest ...*" above uses as the value of `ram` the result of a recursive call of `mark` (in the `let` binding). This is *reflection*: recursion on the output of another recursive call.

Here is the function that `mark` uses to update all the data fields in a record. The arguments are the type, `typ`, of the pointer, the pointer `ptr`, the offset of the next slot `i` (initially `0`), the RAM `ram`, and the declaration `dcl` of all data structures.

```
(defun s* (typ ptr i ram dcl)
  (let* ((descriptor (cdr (assoc typ dcl)))    ; Get descriptor of typ,
         (i (nfix i))                          ;  next slot, and
         (slot-typ (nth i descriptor)))        ;  type of slot.
    (cond
     ((zp ptr) ram)                            ; Stop if ptr is null.
     ((< i (len descriptor))                   ; If more slots to go
      (cond
       ((symbolp slot-typ)                     ;  but this slot is a pointer,
        (s* typ ptr (+ 1 i) ram dcl))          ;  update the others.
       (t (let ((ram (s (+ ptr i)              ; else, write to this slot,
                        (new-field-value typ ptr i ram dcl)
                        ram)))
            (s* typ ptr (+ 1 i) ram dcl)))))   ;  and update the others.
     (t ram))))                                ; Else, no slots left, stop.
```

The definition of `s*` uses the record assignment function, `s`, to set each data field to the value provided by the expression `(new-field-value typ ptr i ram dcl)`. The function `new-field-value` may be thought of as calculating the new value to be written to slot `i` of the data structure of type `typ` at location `ptr` in `ram`. That function is introduced into the logic with the following notation.

```
(encapsulate
 (((new-field-value * * * * *) => *))
 (local (defun new-field-value (typ ptr i ram dcl)
          (declare (ignore typ ptr i ram dcl))
          0))
 (defthm new-field-value-s-commutes
   (implies (not (member addr (seq-int ptr (len (cdr (assoc typ dcl))))))
            (equal (new-field-value typ ptr i (s addr val ram) dcl)
                   (new-field-value typ ptr i ram             dcl)))))
```

This declares `new-field-value` to be a function that takes five arguments and returns one result, satisfying the theorem named `new-field-value-s-commutes`, above. The `local` definition of `new-field-value` serves merely as a witness: there is at least one function satisfying that constraint. In the constraint, `(seq-int ptr (len (cdr (assoc typ dcl))))` is the list of the k successive integers from `ptr` upward, where k is the number of fields in the declared type of `ptr`. These integers are the RAM addresses allocated to the record at `ptr`. The constraint says that the value returned by `new-field-value` on a RAM produced by writing to an address `addr` is the same as the value returned on the original RAM, if `addr` is not one of the fields in the record at location `ptr`.

We define `collect` in a way analogous to `mark`. It returns all of the addresses allocated to the data structure at `ptr`, to a depth of `n`. To save space we do not exhibit the definition, but it is mutually recursive with `collect-lst` in the same way `mark` is mutually recursive with `mark-lst`.

Challenge Theorem 1 from [10] in our setting,

```
(defthm challenge-theorem-1
  (implies (and (not (member addr (collect typ ptr n ram dcl)))
                (unique (collect typ ptr n ram dcl)))
           (equal (g addr (mark typ ptr n ram dcl))
                  (g addr ram)))),
```

states that the contents of address `addr` is unaffected by marking the data structure at `ptr`, provided `addr` is not among the addresses allocated to the data structure and those addresses are unique. The function `unique` checks that no member of its argument occurs twice.

Consider proving `challenge-theorem-1` by induction. First, since `mark` and `mark-lst` are mutually recursive we must simultaneously prove analogous theorems about both. This is Obstacle 1. Second, one of the inductive hypotheses we will want, when dealing with the `mark-lst` theorem will be one in which the `ram` above is replaced by a marked RAM, namely the output of `mark` on a certain substructure, since `mark-lst` is reflexive. This is Obstacle 3. But then we will have to solve the detachment problem. We will be trying to establish that the collection across a marked RAM is unique and we will be doing this before we know how the primitive accessor for RAMs, `g`, behaves with respect to `mark`. This is Obstacle 2. We address these later.

Challenge Theorem 3,

```
(defthm challenge-theorem-3
  (implies (unique (append (collect typ1 ptr1 n1 ram dcl)
                           (collect typ2 ptr2 n2 ram dcl)))
           (equal (mark typ1 ptr1 n1 (mark typ2 ptr2 n2 ram dcl)
                        dcl)
                  (mark typ2 ptr2 n2 (mark typ1 ptr1 n1 ram dcl)
                        dcl)))),
```

states that the order in which the data structures at `ptr1` and `ptr2` are marked can be interchanged if the addresses in the concatenation of the allocated addresses are unique. The more basic Challenge Theorem 1 is needed in the proof of Challenge Theorem 3.

5 Consistent Use

To circumvent the mutual recursion in `mark` and `collect` we introduce singly-recursive flagged versions of those functions, called `mark-fn` and `collect-fn`. We state all but our top-level theorems in terms of those. For example, `(mark-fn fn typ ptr i n ram dcl)` is equal to `(mark typ ptr n ram dcl)` if `fn` is `:ONE` and `(mark-lst typ ptr i n ram dcl)` if `fn` is `:ALL`.[4] Thus, when proving

[4] In the explanation of how we eliminated mutual recursion we used the constants 0 and 1 to distinguish the two behaviors of h. But the ACL2 community tends to use symbolic constants rather than numeric constants. In Common Lisp, a symbol whose name begins with a colon is a keyword constant.

theorems about mark-fn one can obtain inductive hypotheses about both mark and mark-lst, depending on how one instantiates fn. We use mark-fn and collect-fn extensively below but the reader may think of them as mark and collect (by choosing fn to be :ONE).

But the main obstacle is that collect may not visit every address visited by mark, in general, because mark may update and then chase addresses. As noted, this may happen if mark visits the same address multiple times, sometimes treating it as data (possibly overwriting it) and other times as a pointer (and chasing the newly written value). While the hypotheses of the two challenge theorems prevent this, they do it in such a heavy-handed way that the theorems cannot be proved by induction since mark and mark-lst are reflexive.

To side-step this problem we introduce the idea of a *tagging*. A tagging is a RAM that maps each address in it to one of two values, :DATA or :PTR. We say a tagging is "consistent" (or "ok") for a pointer iff every time an address reachable from the pointer is visited it is treated as specified in the tagging. We defined the flagged version of this notion.[5]

```
(defun tags-ok-fn (fn typ ptr i n ram dcl tags)
  (if (equal fn :ALL)
; If we are to check the tagging of every pointer in the block
; from ptr+i through the end of the block:
      (let* ((descriptor (cdr (assoc typ dcl)))
             (slot-typ (nth i descriptor))
             (i (nfix i)))
        (cond
         ((zp ptr) t)                    ; Stop if ptr illegal
         ((<= (len descriptor) i) t)   ;   or block exhausted.
; If next slot is declared to hold a pointer, check that it is so
; tagged and the chase and check that pointer and the rest.
         ((symbolp slot-typ)
          (cond
           ((equal (g (+ ptr i) tags) :PTR)
            (and (tags-ok-fn :ONE slot-typ (g (+ ptr i) ram) i n ram dcl tags)
                 (tags-ok-fn :ALL typ ptr (+ 1 i) n ram dcl tags)))
           (t nil)))
; Otherwise, the slot is declared to hold data.  Check that it is so
; tagged and check the rest.
         ((equal (g (+ ptr i) tags) :DATA)
          (tags-ok-fn :ALL typ ptr (+ 1 i) n ram dcl tags))
         (t nil)))
; If we are to check the tagging of a single pointer:
    (let ((descriptor (cdr (assoc typ dcl))))
      (if (zp n)
          t
        (if (zp ptr)
            t
          (if (atom descriptor)
```

[5] The concepts which might be named "tags-ok" and "tags-ok-lst" are mutually recursive but we never need them in our top-level theorems and so do not actually define them in our script; for simplicity in this paper we use (tags-ok typ ptr n ram dcl tags) as an abbreviation for (tags-ok-fn :ONE typ ptr 0 n ram dcl tags).

```
              t
; Check the tags of all the addresses in the block, and then
; recur on the pointers in the block (see note below).
             (and (s*-tags-ok typ ptr 0 dcl tags)
                  (tags-ok-fn :ALL typ ptr 0 (- n 1) ram dcl tags))))))))
```

Note: The call of `s*-tags-ok` above is redundant. Removing it does not change the value of the function since each address' tag is checked as we scan looking for the pointers in the block. But checking all the addresses "at once" makes it slightly more convenient to state certain technical lemmas.

Using the notion of a tagging we can restate Challenge Theorem 1.

```
(defthm g-mark-fn-2
  (implies (and (not (member addr (collect-fn fn typ ptr i n ram dcl)))
                (tags-ok-fn fn typ ptr i n ram dcl tags))
           (equal (g addr (mark-fn fn typ ptr i n ram dcl))
                  (g addr ram))))
```

The first hypothesis is the same as in the challenge theorem and says that `addr` is not reachable from `ptr` in `ram`. The second hypothesis says that `tags` is a consistent tagging for `ptr` in `ram`. The conclusion is that marking `ptr` in `ram` does not change the contents of `addr`. Observe that we no longer insist that the reachable addresses be unique, only that `tags` is a consistent tagging. Observe also that the tagging is mentioned in the hypotheses of the theorem but not in the conclusion. We can read the second hypothesis merely to say *there exists* a consistent tagging. The fact that `tags` is a "free variable" in these theorems allows us to instantiate it creatively later.

This theorem is proved by induction on the `mark-fn` terms. But how do we deal with Obstacles 2 and 3? That is, how do we solve the detachment problem when `ram` is replaced by a reflexive marking? Two lemmas are key. The first is used to relieve the tag consistency hypothesis.

```
(defthm tags-ok-fn-mark-fn
  (implies (tags-ok-fn fn2 typ2 ptr2 i2 n2 ram dcl tags)
           (equal (tags-ok-fn fn1 typ1 ptr1 i1 n1
                              (mark-fn fn2 typ2 ptr2 i2 n2 ram dcl)
                              dcl tags)
                  (tags-ok-fn fn1 typ1 ptr1 i1 n1 ram dcl tags))))
```

What does this lemma tell us? Suppose `tags` is a consistent tagging for `ptr2` in some RAM `ram`. Consider the RAM `ram'` obtained by marking `ram` from `ptr2`. Then `tags` is a consistent tagging for `ram'` and any pointer `ptr1` precisely if `tags` is consistent for `ram` and `ptr1`. We seem to suffer Obstacles 2 and 3 in this theorem also but they can be overcome with a suitable inductive instance. (See the induction hint provided in the script.)

What makes `tags-ok-fn-mark-fn` inductively provable is that `tags` is not changed and the `tags-ok-fn` in the hypothesis tells us that we will treat every address we encounter exactly the way specified in `tags`. We do not know how to state a similarly strong theorem with `collect-fn` instead of `tags-ok-fn` because when `tags` is eliminated so is the assurance that the same address encountered in two different RAMs will be treated identically.

The second key lemma, needed to relieve the `collect-fn` hypothesis in the induction hypothesis of `g-mark-fn-2`, is essentially the theorem we were looking for earlier (referred to as the key lemma in our discussion of Obstacle 2 on page 5)

```
(defthm collect-fn-mark-fn
```

```
  (implies (and (tags-ok-fn fn1 typ1 ptr1 i1 n1 ram dcl tags)
                (tags-ok-fn fn2 typ2 ptr2 i2 n2 ram dcl tags))
           (equal (collect-fn fn1 typ1 ptr1 i1 n1
                              (mark-fn fn2 typ2 ptr2 i2 n2 ram dcl)
                              dcl)
                  (collect-fn fn1 typ1 ptr1 i1 n1 ram dcl))))
```

a relation between `collect-fn` and `mark-fn`. But now — with the hypothesis about `tags` tying together the way we treat addresses and `tags-ok-fn-mark-fn` to solve Obstacles 2 and 3 in this inductive proof — the proof goes through by the induction suggested by `mark-fn`.

We have been discussing the strengthened version of Challenge Theorem 1. What of Challenge Theorem 3? Here is its strengthened form.

```
(defthm mark-fn-mark-fn
  (implies (and (tags-ok-fn fn1 typ1 ptr1 i1 n1 ram dcl tags)
                (tags-ok-fn fn2 typ2 ptr2 i2 n2 ram dcl tags)
                (not (intersectp (collect-fn fn1 typ1 ptr1 i1 n1 ram dcl)
                                 (collect-fn fn2 typ2 ptr2 i2 n2 ram dcl))))
           (equal (mark-fn fn1 typ1 ptr1 i1 n1
                           (mark-fn fn2 typ2 ptr2 i2 n2 ram dcl)
                           dcl)
                  (mark-fn fn2 typ2 ptr2 i2 n2
                           (mark-fn fn1 typ1 ptr1 i1 n1 ram dcl)
                           dcl))))
```

It says that the order of two markings (on `ptr1` and `ptr2`) can be interchanged provided we have a single tagging that is consistent with both pointers and the reachable addresses from both pointers are disjoint. No mention is made of uniqueness. This theorem is proved by induction (as suggested by `mark-fn` on `ptr2`) and appeals to the two key lemmas previously discussed to skirt Obstacles 2 and 3.

6 Compositions

Having proved the strengthened versions of the challenge theorems it remains to prove the original versions. We deal with this step rather briefly in this paper.

Instantiating the strengthened version of Challenge Theorem 1 with `fn` = `:ONE`, `i` = `0` and using the theorems relating `collect-fn`, `mark-fn`, and `tags-ok-fn` to `collect`, `mark` and `tags-ok`, respectively, gives us:

```
(implies (and (not (member addr (collect typ ptr n ram dcl)))      ; (1)
              (tags-ok typ ptr n ram dcl tags))
         (equal (g addr (mark typ ptr n ram dcl))
                (g addr ram))).
```

We wish to prove `challenge-theorem-1`,

```
(implies (and (not (member addr (collect typ ptr n ram dcl)))      ; (2)
              (unique (collect typ ptr n ram dcl)))
         (equal (g addr (mark typ ptr n ram dcl))
                (g addr ram))),
```

Thus, all we must show is that the uniqueness of (collect typ ptr n ram dcl) implies the existence of a tagging, tags, satisfying (tags-ok typ ptr n ram dcl tags).

We state this by defining a function, named tags-witness[6] that constructs a suitable tagging, given ptr, ram, etc. The key idea is to explore ram from ptr just as collect does and accumulate into an initially empty tagging the use (:DATA or :PTR) to which each address is put.

The tagging constructed by tags-witness is consistent if the collected addresses are unique, since each address is used only once and in the way specified. Thus, we can prove (2) from (1).

We now turn to the original version of Challenge Theorem 3. We have proved its strengthened form and can, again, eliminate the flagged functions by instantiation to obtain the theorem:

```
(implies (and (tags-ok typ1 ptr1 n1 ram dcl tags)                ; (3)
              (tags-ok typ2 ptr2 n2 ram dcl tags)
              (not (intersectp (collect typ1 ptr1 n1 ram dcl)
                               (collect typ2 ptr2 n2 ram dcl))))
         (equal (mark typ1 ptr1 n1 (mark typ2 ptr2 n2 ram dcl) dcl)
                (mark typ2 ptr2 n2 (mark typ1 ptr1 n1 ram dcl) dcl))).
```

We wish to prove challenge-theorem-3,

```
(implies (unique (append (collect typ1 ptr1 n1 ram dcl)          ; (4)
                         (collect typ2 ptr2 n2 ram dcl)))
         (equal (mark typ1 ptr1 n1 (mark typ2 ptr2 n2 ram dcl) dcl)
                (mark typ2 ptr2 n2 (mark typ1 ptr1 n1 ram dcl) dcl))),
```

Note that the conclusions are the same so we must show that the hypothesis of (4) implies the hypotheses of (3). From the hypothesis of (4) we know that the collections from ptr1 and ptr2 are unique and disjoint. The disjointness result gives us the third hypothesis of (3). The two uniqueness results, together with the previously proved property of tags-witness, allows us to build two taggings, say tags1 and tags2, that are consistent with ptr1 and ptr2, respectively. But to establish the first two hypotheses of (3) we must show the existence of a single tagging, tags, that is consistent for both ptr1 and ptr2. Such a tagging can be constructed by "unioning" tags1 and tags2. The addresses mapped by each are disjoint. Thus, the tag assigned by the union for an address reachable from ptr1 is the tag in tags1; an analogous remark holds for addresses reachable from ptr2 and tags2. Thus, the union is a tagging consistent for both ptr1 and ptr2 and we have thus proved (4) from (3).

We see that the "neatly packaged" single hypothesis of (4) is actually used several ways. The disjointness it implies is required explicitly in (3) simply to guarantee that the two mark terms do not write to the same addresses. The uniqueness it implies is over-strong and must be replaced in (3) by the guarantee that each address is treated in a consistent way. Addresses may be visited multiple times as long as they are used consistently on each visit. We use both aspects of the hypothesis of (4) to establish the existence of a tagging that is consistent for both pointers.

7 Hints at Some Omitted Details

A total of 91 theorems are proved in the script described here. Approximately 30 of the theorems required hints and we have not indicated in the displays above that a hint might have been required.

[6] Actually, we define the flagged function tags-witness-fn because "tags-witness" is mutually recursive with "tags-witness-lst." We put those names in quotations because we never actually introduce them formally in our script but content ourselves to using tags-witness-fn with the appropriate value of fn.

About a third of the hints specify inductions not chosen automatically. Another common class of hints specialize `(+ i ptr)` to `ptr` by instantiating `i` to `0` or specify free variables creatively.

We have skipped over many of the details of our construction of taggings. One interesting detail is that we "union" two taggings with the function `merge-tags` and use `merge-tags` in the definition of `tags-witness`. Thus, contrary to the order in which we discussed the ideas, the theorem that `merge-tags` preserves consistency of both taggings is proved first and used in the proof that that `tags-witness` constructs a consistent tagging.

An annoying aspect of our proof is that the records book does not provide a way to determine all the addresses mapped by a tagging. Thus, `merge-tags` (which essentially just unions two finite functions with disjoint domains) actually takes two taggings and a list of addresses and extends one of the taggings by adding the assignments from the other for all the addresses listed. We then must use `collect` to determine the relevant addresses. This in turn forces us to prove, essentially, that the only addresses mapped by the tagging constructed by `tags-witness` are those returned by `collect`. This would have been unnecessary, we believe, had we used the notion of finite functions provided by `books/finite-set-theory/set-theory` [19] instead of RAMs for taggings.

Another interesting detail is a subtlety in how one defines the reachable addresses in the general theorems about taggings. When collecting from the head of a block (i.e., from the address of the block), we collect all the addresses in the block and then add the ones reachable from the pointers. The addresses reachable from the pointers do not include the addresses holding the pointers, since they are always included when the block was first visited. But to state inductively provable theorems about collecting and tagging it is necessary to think about visiting a block from the middle onward. These theorems require a more extensive sense of collection (which is called "kollection" in our script) which includes the addresses from the middle of the block onward through the block. This notion collapses to collection when starting at the beginning.

8 Conclusion

The aim of this paper is to help formal methods practitioners deal with the formal proofs of properties of data structures represented in a RAM-like memory. This problem commonly arises when dealing with proofs of microcode, machine code, assembly code and compiler correctness. We recapitulate the main ideas to help the reader put them in context.

- We provide a fairly general treatment of the representation of data structures in RAM. We permit the user to declare an arbitrary number of types, each containing a fixed but arbitrary number of fields and each to be represented by a contiguous block of memory representing the successive fields of the record. Each field is declared to contain a data item or a pointer to a record of some fixed but arbitrary type.
- We define an elementary generic recursive traversal algorithm that takes as input a pointer of a given type, a RAM and the declaration of all types. The algorithm explores the entire data structure down to an arbitrary depth and updates each data field. The updated value is constrained to be some function of the other fields in the record or else determined externally from the RAM. We do not specify how records are allocated and allow for overlapping records with conflicting assignments of types to addresses. We allow circular structures. But the theorems we prove restrict what one can conclude about the result of marking a RAM containing such structures.
- We define the notion of the addresses reachable from a given pointer in a RAM.
- We define the notion of a tagging of the addresses reachable from a pointer. This notion allows us to make precise the idea that every address visited in a traversal from the pointer is always interpreted consistently as either data or pointer, but never as both.

- We prove that if a pointer has a consistent tagging in a RAM and an address is not reachable from the pointer in the initial RAM then exploring and marking the RAM from the pointer does not write to that address. This is non-obvious because of the possibility that overlapping allocations may cause the marking algorithm to change a pointer and reach parts of the RAM not seen in the initial reachability analysis.
- We prove that if there is a single tagging that is consistent for two pointers and the addresses reachable from those pointers are disjoint, then one can explore and mark the two data structures in either order with identical effects on the RAM.
- We show that if the addresses reachable from a pointer are reached only once in an exploration then there exists a consistent tagging for the pointer.
- We show that consistent taggings for each of two pointers can be merged to create a consistent tagging for both pointers, provided the reachable addresses are disjoint.
- We combine the two results about the existence of consistent taggings with the two earlier theorems to prove two weaker (but still interesting) theorems suggested by [10]. The first is that if an address is not among the reachable addresses of a pointer and the reachable addresses are unique, then the address is not changed by traversing and marking the pointer. The second is that if the addresses in the concatenation of the reachable addresses of two pointers are unique, then one can traverse and mark the two pointers in either order with identical effect on the RAM. These results most immediately apply to RAMs representing acyclic, non-overlapping data structures. But our general results apply to a wider class of RAMs.

Most importantly, we describe the obstacles encountered in formalizing and proving these results formally in a quantifier-free first order logic. The formal introduction of memory taggings was motivated entirely by technical considerations in the proofs. We believe it is a generally useful formal concept when dealing with dynamically allocated pointer-based data structures represented in a RAM-like memory.

9 Acknowledgements

I wish to thank Dave Greve and Matt Wilding for their challenge problem.

References

1. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. *SIGPLAN Notices: Conference Record of POPL 2002*, 37(1):1–3, January 2002.
2. W. R. Bevier. A verified operating system kernel. Ph.d. dissertation, University of Texas at Austin, 1987.
3. W.R. Bevier, W.A. Hunt, J S. Moore, and W.D. Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4):409–530, 1989.
4. R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.
5. R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. In *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, 1972.
6. J. Cook and S. Subramanian. A formal semantics for C in Nqthm. Technical Report Technical Report 517D, Trusted Information Systems, October 1994.
7. L. P. Deutsch. An interactive program verifier. Technical Report CSL-73-1, Xerox Palo Alto Research Center, May 1973.

8. M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 177–187, Los Alamitos, CA., May 2001. IEEE Computer Society Press.
9. G. Gonthier. Verifying the safety of a practical concurrent garbage collector. In *Proceedings of CAV '96*. Springer Verlag LNCS 1102, 1996.
10. D. Greve and M. Wilding. Dynamic datastructures in ACL2: A challenge, November 2002.
11. Paul B. Jackson. Verifying a garbage collection algorithm. In Jim Grundy and Malcolm Newey, editors, *11th International Conference on Theorem Proving in Higher-Order Logics: TPHOLs'98*, volume 1479 of *Lecture Notes in Computer Science*, pages 225–244. Springer-Verlag, September 1998.
12. M. Kaufmann and J S. Moore. *The ACL2 User's Manual*. `http://www.cs.utexas.edu/users/moore/-acl2/acl2-doc.html`, 1999.
13. M. Kaufmann and R. Sumners. Efficient rewriting of data structures in ACL2. In *ACL2 Workshop 2002*, Grenoble, April 2002. http://www.cs.utexas.edu/users/moore/acl2/workshop-2002/.
14. B. Levy, I. Filippenko, L. Marcus, and T. Menas. Using the State Delta Verification Systems (SDVS) for hardware verification. In *Theorem Provers in Circuit Design, Proceedings of the IFIP WG10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice, and Experience*, pages 337–360. North-Holland, Amsterdam, 1992.
15. D. Luckham and N. Suzuki. Verification of array, record, and pointer operations in Pascal. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):226 – 244, October 1979.
16. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. Technical Report http://www.in.tum.de/~{mehta,nipkow}/, Institut für Informatik, Technische Universität München, 2003.
17. J S. Moore. *Piton: A Mechanically Verified Assembly-Level Language*. Automated Reasoning Series, Kluwer Academic Publishers, 1996.
18. J S. Moore. An exercise in graph theory. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 41–74, Boston, MA., 2000. Kluwer Academic Press.
19. J S. Moore. Finite set theory in ACL2. In *Proceedings of TPHOLs 2001*, volume 2152, pages 313–328, Heidelberg, 2001.
20. M. Norrish. Deterministic expressions in C. In *Proceedings of ESOP/ETAPS'99*, pages 147–163. Springer-Verlag LNCS 1576, 1999.
21. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in CS*. IEEE, 2002.
22. D. Russinoff. A mechanically verified incremental garbage collector. *Formal Aspects of Computing*, 6:359–390, 1994.
23. R. W. Topor. The correctness of the Schorr-Waite list marking algorithm. *Acta Informatica*, 11(1):211–221, 1979.