Proof of Transitive Closure Property of Directed Acyclic Graphs

Fares Fraij
Department of Information Technology
Al-Hussein Bin Talal University
Ma'an, Jordan
P.O. Box 20
fares@ahu.edu.jo

Steve Roach
Department of Computer Science
The University of Texas at El Paso
El Paso, TX 79968
U.S.A.
sroach@utep.edu

ABSTRACT

This paper presents a formal correctness proof for some properties of *restricted* finite directed acyclic graphs (DAGs). A restricted graph has a single root and arbitrary siblings. The siblings are assigned integers, string values, or restricted DAGs. Leafs must be assigned string values. The main property is the *transitive closure*. Our restricted graphs and the properties are formalized in ACL2, and an ACL2 book has been prepared for reuse.

1. INTRODUCTION

Sandia National Laboratories developed the Sandia Secure Processor (SSP) as a computational infrastructure for high-consequence embedded real-time systems [1]. The SSP is a restricted form of the Java Virtual Machine. The job of the SSP class loader is to correctly translate Java class files into a form suitable for execution by the SSP, a form called a *ROM image*.

Our work is motivated by our desire to prove the correctness of a transformational approach to loading Java class files for the SSP, specifically, loading the constant pool (CP). For our purposes, assume that a constant pool entry consists of an integer identifier and a value. A value is an index, a string, or an entry. An index corresponds to the identifier of some other entry in the pool. Given this definition of a constant pool, we can represent a pool using a Directed Acyclic Graph (DAG) [1], as shown in Figure 1.

In this paper, the terms *constant pool* and *DAG* are used interchangeably. There are further restrictions on our graphs. In our constant pool, strings should appear just once, and every entry should be dereferencable, i.e., should lead to a set of strings.

A graph is a tuple (**N**, **E, R**) where **N** is the set of *nodes*; **E** is a set of directed *edges*, i.e., pairs (n_a, n_b) originating at n_a and terminating at n_b; and **R** is a distinguished set of nodes $\mathbf{R} \subseteq \mathbf{N}$ called *root nodes*. For each edge, the first node in the pair is the *head node,* and the second node in the pair is the *tail node*. A *leaf node* is a node n_t such that there are no edges in **E** with n_t as the head. A node n_k is *reachable* from a node n_1 if there exists a sequence of edges $e_1, e_2, \ldots, e_{k-1}$ such that:

- The head of e_1 is n_1.
- The tail of e_{k-1} is n_k.
- For every adjacent pair of edges in the sequence, e_i, e_{i+1}, the tail of e_i is the head of e_{i+1}.

A graph defines the reachable relation[1]. The *transitive closure* of the reachable relation is the graph such that for any pair of nodes (n_a, n_b) if n_b is reachable from n_a, then there is an edge (n_a, n_b) in **E**.

A DAG is a graph that does not contain cycles, that is, no node is reachable from itself. A DAG represents a partial order on the nodes of the graph. DAGs have many applications in mathematics and computer science. As a data structure, DAG can be used to organize elements of interest in a hierarchy to expedite searching for an element.

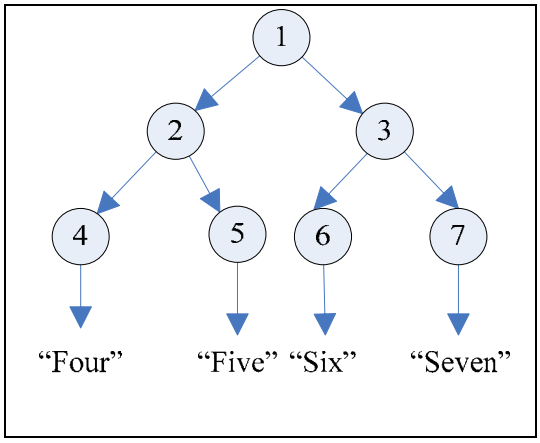

Figure 1: A DAG

In the context of DAG, a crucial notion to be formalized is the acyclicity of an index. An index is acyclic in a constant pool if there are no cycles in the DAG of reachable indexes.

```
c1 = ((1 . (2 . 3))
      (2 . (4 . 5))
      (3 . (6 . 7))
      (4 . "Four")
      (5 . "Five")
      (6 . "Six")
      (7 . "Seven")))
```

Figure 2: ACL2 Model of DAG

[1]This relation may be described by many different graphs.

One implementation of the SSP class loader utilizes a transformational approach. In this loader, a class file is refined by transformation rules in a stepwise fashion until a ROM image is obtained. When loading the CP, the CP is refined by a transformation rule that removes a single level of indirection. We want to show that the semantics of the CP is preserved by this transformation.

Graph-theory-related proofs have been previously investigated in the context of ACL2 [2, 3]. This paper demonstrates the correctness of the function **dref**, which takes an index and returns the "meaning" of that index, i.e., the terminal strings to which the index refers. The approach is to model the CP as a DAG and transform the DAG by substitution, then prove that that the function **dref** yields the same value when applied to a constant pool and to the same constant pool after applying the substitution rules to it.

This paper is organized as follows: Section 2 introduces the modeling of the DAG, the substitution process, and the semantic function. In Section 3, sketches the proof of correctness. Finally, the paper is concluded in Section 4. A book is constructed to enable ACL2 users to reuse the theorems and lemmas.

2. Modeling Phase.

An important concept is the meaning of a structure in the constant pool. This denotation is achieved through a *semantic* function **dref**. From an operational perspective, **dref** can be thought of as a function that, when given a data object model *p* (containing constant pool indexes) and a constant pool model *cp*, will directly try to resolve all indexes in *p* by repeatedly replacing indexes with the data found in the corresponding constant pool entry. For example, consider the constant pool, c_1, presented in Figures 1 and 2. The result of applying the function **dref** to the row indexed 1 in c_1 will result in the following data object ((“Four” . “Five”) . “Six” . “Seven”).

In our class loader, a constant pool goes through a set of substitutions that result in a *resolved* version of the original constant pool. The ACL2 function **apply-rule-to-entry** applies the substitution rule *r* to the i^{th} entry in a constant pool. A valid substitution must preserve the acyclicity of a constant pool. For example, consider again the constant pool, c1, presented in Figures 1 and 2. The result of applying the function **apply-rule-to-entry** using the substitution rule *2 → (4 . 5)* to the first row in c1, *(1 . (2 . 3))*, will result in a new constant pool similar to c1 except that the first row will be replaced by the following object: *(1 . (4 . 5) . 3)*.

2.1 Modeling the DAG and Substitution Rules.

A DAG is modeled by a list of *entries*. Each entry is modeled as an ACL2 *alist* where its **car** is a natural number, referred to as an *index*, and its **cdr** is a data object, which may be an index, a string, or a structure consisting of indexes, strings, or a combination of both. Formally, data objects are elements of the set $\sum$ defined by the regular expression: $\sum = (I + S)^+$ where *I* denotes the set of all possible constant pool indexes, and *S* denotes the set of all terminal strings (e.g., UTF8 strings allowable in Java). A DAG is *well-formed* if it satisfies these restrictions.

It is worth mentioning that the set of *valid* substitution rules are easily generated from a DAG. For instance, the substitution rules that correspond to the DAG shown in Figures 1 and 2 are introduced in Figure 3.

```
1 → (2 . 3)
2 → (4 . 5)
3 → (6 . 7)
4 → "Four"
5 → "Five"
6 → "Six"
7 → "Seven"
```

Figure 3: Valid Substitution Rules

2.2 Semantic function.

The semantic function, **dref**, represents the transitive closure of the value obtained by following an index in the constant pool. The ACL2 implementation of the function **dref** is as follows.

```
(defun dref (p dag)
  (declare (xargs :measure (acount p dag)))
  (cond ((acyclicp p dag)
         (cond ((atom p)
                (if (natp p)
                    (dref (valueOf p dag) dag)
                  p))
               (t (cons (dref (car p) dag)
                        (dref (cdr p) dag)))))
        (t nil)))
```

From an operational perspective, **dref** can be thought of as a function that, when given a data object model *p* and a DAG model *dag*, will directly try to resolve all indexes in *p* by repeatedly replacing indexes with the data found in the corresponding DAG entry.

Acyclicity is modeled with the ACL2 function **acyclicp**. Let *p* be an object, i.e., a list of indexes, and *dag* be a constant pool. **acyclicp** returns `true` if each of the indexes in *p* is acyclic in *dag*. Thus, we only care about acyclicity with respect to a data object. The key idea is that as we walk a branch of the DAG from root to leaf, the DAG is *cyclic* if we visit some node twice. Thus, for each branch, we keep track of the nodes that have already been visited. **acyclicp1** takes a data object, a DAG, and a list of indexes visited so far. **acyclicp** calls **acyclicp1** with the inputs *p*, *dag*, and the empty set **nil** as the set of indexes *visited* so far. If *p* is an atom, **acyclicp1** checks whether *p* is well-formed in *dag*. *p* is well-formed in *dag* if it is not a natural number or if it is a natural number, but it is not associated with any other entry in *dag* and has not been encountered in the set *visited*. The ACL2 representation of the function **acyclicp** is as follows.

```
(defun acyclicp (p dag)
  (acyclicp1 p dag nil))
```

The ACL2 representation of the function **acyclicp1** is as follows.

```
(defun acyclicp1 (p dag visited)
(declare (xargs :measure
    (llist (len (setdiff (all-indexes-in-cp dag) visited))
                (acl2-count p))
                :well-founded-relation l<))
      (cond ((atom p)
           (cond ((natp p)
                (cond ((not (assoc p dag)) nil); p not in dag
                     ((member-equal p visited) nil);p visited
                     (t  ; p is a bare index
                           (acyclicp1 (valueOf p dag)
                           dag
                           (cons p visited)))))
                (t t)))
          (t (and (acyclicp1 (car p) dag visited)
                (acyclicp1 (cdr p) dag visited))))))
```

2.3 Modeling the Substitution Process.

The index substitution process is modeled in ACL2 by the function **apply-rule-to-entry**, which applies the substitution rule *r* to the i^{th} entry in dag. For instance, consider the dag given in Figure 1 and its corresponding substitution rules given in Figure 3. The result of applying the substitution rule *2 → (4 . 5)* to the first row in c_1 will result in a new dag in which row 1 will be as follows: *'((1 . (4 . 5) . 3).*

The proof that the substitution function **apply-rule-to-entry** is correctness-preserving is based on the semantic function **dref,** The result of following a structure *p* in a DAG will lead to the same ultimate value obtained by following *p* in a DAG to which a substitution rule has been applied.

The function **apply-rule-to-entry** checks whether the condition **(ok-rulep r dag)** holds. For example, the rule *3 → (6 . 7)* is **ok-rulep** in c_1, however the rule *2→ (2 . 5)* is not. For a rule to be well-formed with respect to a DAG, it must satisfy several conditions. First, it must be a rule that applies to the DAG, i.e., the head of the rule must be one of the DAG's indexes. Second, the rule must be acyclic with respect to the DAG. The head and tail of the rule must lead to the same value in the DAG. Finally, application of the rule must result in a DAG that has less indirection than the original DAG, i.e., the length of at least one branch of the DAG must be reduced. The ACL2 model of the function **apply-rule-to-entry** is presented as follows.

```
(defun apply-rule-to-entry(r i dag)
 (if (ok-rulep r dag)
    (apply-r-entry1 r i dag)
  dag))
```

```
(defun apply-rule-to-entry1 (rule position cp)
 (let ((rhs (valueOf position cp)))
  (cond ((matchValueRulep rhs rule)
       (replaceValue position
          (applySubstitution rhs
                rule)
          cp))
       (t cp))))
```

3. Proof of Correctness.

Substitution rules are applied to a DAG to transform it to a resolved DAG. Our goal is to show that the transformations preserve meaning. Informally, the main theorem states that given a well-formed DAG dag, and an index p in dag, the results of following p to its ultimate value in the resolved dag and following the index p to its ultimate value in dag are identical.

3.1 Proof of the Main Theorem

The main theorem states that the application of the function apply-rule-to-entry to dag will preserve the semantics of dag, i.e., the substitution rule r when applied to the index i in dag will preserve meaning. The main correctness theorem can be stated in ACL2 as follows.

```
;; Main Theorem
(defthm dref-apply-rule-to-entry-1-new
 (implies (and (ok-rulep rule dag)
          (uniqueNodeIDp dag)
          (acyclicp i dag);;
          (acyclicp p dag))
       (equal (dref p (apply-rule-to-entry1 rule i dag))
          (dref p dag)))
 :hints (("Goal" :in-theory (enable ok-rulep))))
```

The function **apply-rule-to-entry1** is non-recursive and, therefore, the unfolded version of the theorem is

```
 ;; lemma 2
(defthm dref-put-ok-rulep
 (implies (and (ok-rulep rule dag)
          (uniqueNodeIDp dag)
          (acyclicp i dag);;
          (acyclicp p dag))
       (equal
        (dref p (replaceValue i
              (applySubstitution (valueOf i dag) rule)
                    dag))
        (dref p dag))))
```

To prove Lemma 2, we have first to prove that, given some hypotheses, the application of the term **(replaceValue i (applySubstitution (valueOf i dag) rule) dag)** preserves acyclicity. This is expressed in Lemma 3 below. Lemma 2 and Lemma 3 represents the crux of the proof and will be further investigated in Section 3.2.

```
;; lemma 3
(defthm acyclicp-i-put-i-replace-indexes-general-1
 (implies (and (ok-rulep r dag)
               (uniqueNodeIDp dag)
               (acyclicp i dag)
               (acyclicp p dag))
    (acyclicp p (replaceValue i
          (applySubstitution (valueOf i dag) r) dag))))
```

A useful lemma proves the conjecture that, given some hypotheses, the function **apply-rule-to-entry1** preserves acyclicity. Such a conjecture is represented in Lemma 4. Lemma 4 is proven automatically by ACL2 since its unfolded version, Lemma 3, is already proven.

```
;; lemma 4
(defthm acyclicp-apply-r-entry1-new
 (implies (and (ok-rulep r dag)
          (acyclicp p dag)
          (acyclicp i dag)
          (uniqueNodeIDp dag))
       (acyclicp p (apply-rule-to-entry1 r i dag))))
```

3.2 The Proof of the Crux Lemmas.

The proof of the crux lemmas, namely Lemma 2 and Lemma 3, is achieved by first proving Lemma 3. After this, Lemma 4 is proven by induction. Note that Lemma 3 uses the non-recursive predicate **acyclicp**, which will be expanded by ACL2 to the predicate **acyclicp1**. Thus the proof of Lemma 3 requires proving a more general lemma regarding **acyclicp1**. In any attempt to prove a theorem that includes the predicate **(acyclicp p dag)**, ACL2 will expand the predicate to **(acyclicp1 p dag nil)**. Thus, one has to prove a more general theorem about the predicate **(acyclicp1 p dag visited)**, for any *visited*. Then the predicate **(acyclicp1 p dag nil)** can be instantiated with *visited* set to **nil**, and therefore the theorem about the predicate **(acyclicp p dag)** is proven.

Lemma 5 represents the generalized version of the Lemma 3 after expanding the predicate **acyclicp** to **acyclip1**.

```
;; lemma 5
(defthm acyclicp-i-replaceValue-i-applySubstitution-general-1
 (implies (and (ok-rulep rule dag)
          (acyclicp1 p dag visited)
          (acyclicp1 I dag visited)
          (uniqueNodeIDp dag))
       (acyclicp1 p (replaceValue i
            (applySubstitution
                        (valueOf i dag) rule) dag))) visited)
```

To prove lemma 5, a similar lemma stating that the term **(applySubstitution p r)** preserves acyclicity must be proven.

```
;; lemma 6
(defthm acyclicp1-replace-indexes-1
 (implies
  (and (uniqueNodeIDp dag)
     (acyclicp1 p dag seen)
     (acyclicp1 (cdr r) dag visited))
 (acyclicp1 (applySubstitution p r) dag visited))
```

Having proven Lemma 6, the proof of Lemma 5.1 is achievable by splitting the proof of Lemma 6 into two cases: (1) *i* is not reachable from *p* and (2) *i* is reachable from *p*.

Then, Lemma 3 is provable via instantiating *visited* with **nil**. Having proven Lemma 3, Lemma 2 is proven automatically by ACL2.

3.3 A Theorem about `dref`.

An interesting theorem about the function **dref** is that given a *dag* in which all pointers are acyclic, represented by the predicate **(acyclic-constant-poolp dag),** and an acyclic index *p* in *dag***,** then the result of dereferencing *p* in *dag* contains *no* indexes. Note that the predicate **no-indexesp** is a predicate that takes as an input a structure (or an object) and returns **true** if the structure contains no indexes; **nil** otherwise.

```
(defthm no-pointersp-dref
 (implies (and (acyclic-constant-poolp dag)
          (acyclicp p dag))
       (no-indexesp (dref p dag))))
```

4. Conclusion.

This paper introduced a formal proof of correctness of properties for a restricted form of DAG that has practical importance. The representation of the DAG has its root to the constant pool in Java and the proof highlights the correctness preserving property of transitive closure as valid rules are applied to the DAG. The result of this work is an ACL2 book that can be reused by the ACL2 community when encountering proofs that include such DAG structure.

5. ACKNOWLEDGMENTS

The authors would like to thank J Moore and Hanbing Liu for their many contributions to this work.

6. REFERENCES

[1] Victor Winter, *et al*, A Transformational Perspective into the Core of an Abstract Class Loader for the SSP, ACM Transactions on Embedded Systems, Vol. No,. 2006.

[2] J Strother Moore. An exercise in graph theory. In Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers, 2000.

[3] J Strother Moore, Qiang Zhang: Proof Pearl: Dijkstra's Shortest Path Algorithm Verified with ACL2, TPHOLs 2005: 373-384.