The Thesis Committee for Hormoz Zarnani

certifies that this is the approved version of the following thesis:

An Incremental Approach to Verifying the Intentional Naming System

Committee:

J Strother Moore, Supervisor

Robter S. Boyer

Sarfraz Khurshid

An Incremental Approach to Verifying the Intentional Naming System

by

Hormoz Zarnani

Undergraduate Honors Thesis

Presented to the Faculty of

The University of Texas at Austin

in Partial Fulfillment of the Requirements

for the Degree of

Bachelore of Sciences

The University of Texas at Austin

December 2005

An Incremental Approach to Verifying the Intentional Naming System

Approved by
Supervising Committee:

_____

_____

To my parents

Acknowledgments

This work would not have been possible without the help and support of many individuals. I am especially thankful to my advisor, Professor J Strother Moore, for his wisdom, encouragement, and patience. I am also grateful to my committee members Professors Robert Boyer and Sarfraz Khurshid. Most especially, I would like to thank my family for their love an support.

Hormoz Zarnani

The University of Texas at Austin

December 2005

An Incremental Approach to Verifying the Intentional Naming System

Hormoz Zarnani, B.S.
The University of Texas at Austin, 2005

Supervisor: J Strother Moore

Lookup-Name is the name resolution algorithm for the Intentional Naming System (INS). We formalize this algorithm in the ACL2 logic and develop a specification for its correctness. We discuss certain difficulties that arise in our formalization of this algorithm which restrict our ability to prove its correctness. We describe a simpler version of the algorithm, specify it, and prove it correct. We discuss how the proof of the simple algorithm can be used as a guide to prove the original algorithm correct.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

The Intentional Naming System (INS) is a scheme for resource discovery in dynamic networked environments. In INS, services are referred to by *intentional names*, not by their low-level network locations. This naming system has a number of algorithms that perform various operations on intentional names.

The original design of INS was modeled in Alloy and was analyzed using the Alloy Analyzer (AA)[1] as described in [4]. Alloy exposed several flaws in both the algorithms of INS and its underlying naming semantics. Those flaws were fixed and the corrected version of INS was analyzed using Alloy. The new design of INS was found to have no more errors.

AA's analysis is sound; however, since it performs bounded checking, its analysis is incomplete. Being a sound analyzer, AA found "real" flaws in INS (*i.e.* they were not false positives). But because AA is incomplete, it may not necessarily have found *all* of the flaws in INS. Besides, it is possible that the corrections made to INS introduced new errors, and Alloy missed them when it was used to analyze

[1] Alloy is a declarative, structural modeling language based on first-order logic and is used for expressing complex structural constraints and behavior. The Alloy Analyzer is a SAT-based constraint solver that provides fully automatic simulation and checking.

the re-designed system.

Lookup-Name, the name resolution algorithm of INS responsible for resolving queries to records in databases, is one of the most important algorithms of this naming system. AA revealed the largest number of bugs in Lookup-Name than in any other algorithm of INS.

In this paper, we present our attempt to prove the correctness of Lookup-Name. We formalize this algorithm and develop a correctness specification for it. Our formalization of the algorithm led to a complex model that was awkward to reason about formally.[2] Our failed attempts at verifying Lookup-Name suggested a need to simplify the problem. Hence, we abstracted out the details of the complex model that were causing us to stagnate and reduced the problem to one that is easier to reason about.

We recognized that the operation of Lookup-Name can be viewed as two separate tasks: decide *tree containment* and collect records. We observed that one, namely deciding tree containment, can be addressed separately. We described a simple algorithm to decide tree containment, specified it, and proved it correct. We believe that the theorems we proved to verify this simple algorithm will be suggestive of those we will need to prove for the actual problem. That is to say, we anticipate that the proof of the correctness of the Lookup-Name algorithm will involve many lemmas that are closely related to those we proved for the tree containment problem.

The remainder of this paper is organized as follows. In Chapter 3 we present our ACL2 model of Lookup-Name and the two data structures that it uses, namely queries and databases, as well as our specification of its correctness. In Section 3.2.3, we discuss the complexities of our formalization of the algorithm in more detail. In

[2]We stress here that ACL2 has been successfully used to verify much more complex designs than ours. We use the word "complex" above relative to our available time for this project.

Chapter 4, we describe our simplification of the problem – we introduce a simplified algorithm to only decide tree containment; develop a specification of correctness similar to that we developed for Lookup-Name; and verify the simplified algorithm. Finally, in Chapter 5 we summarize the paper and discuss our future work plans.

Chapter 2

Background

2.1 The Intentional Naming System

The Intentional Naming System (INS) is a scheme for resource discovery and service location in dynamic networks of devices and computers [1]. In INS, services are described and referred to by *intentional names*. These names describe a set of properties that a service provides. This naming scheme allows client applications to refer to *what* service they are looking for as opposed to *where* the desired service is located. In other words, clients refer to the service they *intend* to obtain by describing it (*e.g.* "the nearest color printer for transparencies"), not by specifying its low-level network location (*e.g.* a host name such as "18.13.0.44"). INS also allows applications (*i.e.* clients and service providers) to communicate seamlessly despite changes in the mapping from service names to service locations while a session is in progress.

An intentional name is an arrangement of alternating levels of *attributes* and *values* in a tree structure which represent the service property. The mapping between service descriptions and their locations is maintained by *name resolvers*.

The database in a name resolver provides three key operations: resolving queries to records, adding new services, and finding the name used by a given service in its advertisement. Queries are resolved to appropriate service(s) using the *Lookup-Name* operation.[1] Adding new services and finding services are performed using the *Add-Name* and *Get-Name* operations, respectively.

2.2 ACL2

We use ACL2 in our study of formalizing and reasoning about the Lookup-Name algorithm. For that reason, we give a brief introduction to ACL2 in this section.[2]

"ACL2" stands for A Computational Logic for Applicative Common Lisp, which is the name of a functional programming language, a first-order mathematical logic, and a mechanical theorem prover. It is a theorem prover in the Boyer-Moore tradition that uses rewriting, decision procedures, mathematical induction and many other proof techniques to prove theorems in a first-order mathematical theory of recursively defined functions and inductively constructed objects.

ACL2 has a Definitional Principle, which is one of the means by which the theory can be extended. When a user submits a function definition, she must prove that the function terminates. Non-recursive functions always terminate. However, for recursive functions, she must prove "measure conjectures" establishing that some measure of the arguments is decreasing in a well-founded way under the tests governing the recursion. Only after she has proved these conjectures is the definitional equation "admitted" as a new axiom. We show an example of proving measure conjectures for a (mutually) recursive function on page 27.

ACL2 also has an Induction Principle. It is one of the inference rules that

[1] This algorithm will be the focus of this paper.

[2] The block quotations in this section have been taken from [3] with permission of the authors.

enables us to prove theorems in ACL2. It is entirely based on

One of the rules of inference in ACL2 is the Induction Principle. It is based on well-founded induction up to ϵ_0. We show an example of an inductive proof in Section B.1.

For more detailed information about the ACL2 logic, we refer the reader to [2, 3].

Chapter 3

Formalization of The Lookup-Name Algorithm

In this chapter, we present our formal model of the Lookup-Name algorithm. We then develop a correctness specification for this algorithm. Finally, we discuss the complexities of the model.

3.1 The ACL2 Model

3.1.1 INS Data Structures

Lookup-Name operates on two types of data structures – queries (*a.k.a.* intentional name specifiers) and databases (*a.k.a.* intentional name resolver trees). Our model of these data structures is based on the INS description in [1].

AV-Trees

Queries and databases are represented as a special form of trees called *attribute-value trees* (or *av-trees*), which are trees with alternating levels of attribute nodes and value nodes. We model these trees as lists in ACL2. In general, a tree is modeled as a list of the form `(data . children)`, where `data` is the information contained in the root node of the tree and `children` is the list of the subtrees. We do not use a tag field for specifying the type of the node because it can be determined by the position of the node relative to the root of the tree. In an av-tree, every attribute node has at least one (value) child. Another constraint on av-trees is that the children of node are mutually unique. In other words, no two siblings are the same.

Queries

Queries (*a.k.a. name specifiers*) are special kinds of av-trees. In queries, the only information that a node contains is either an attribute or a value. We call the attribute (or the value) of an attribute (or value) node the "label" of that node. Queries have a structural constraint: that attribute nodes have "exactly" one child (as opposed to "at least" one child). The mutually recursive functions[1] `q-avp`, `q-vap`, and `q-av-listp`, as shown in Figure 3.1.1 recognize an attribute-rooted query, a value-rooted query, and a list of value-rooted queries, respectively.

[1] A note about mutual recursion: in general, when we define functions to operate on trees with arbitrary branching (*i.e.* each node can have an arbitrary number of children), we need two mutually recursive functions, one to operate on a single tree and another to operate on a list of trees. For that reason, most of the functions we define in this paper will be mutually recursive.

```
(mutual-recursion
 (defun q-avp (x)
   (and (consp x)
        (q-vap (car (children x)))
        (null  (cdr (children x)))))
 (defun q-vap (x)
   (and (consp x)
        (consp (data x))
        (q-av-listp (children x))))
 (defun q-av-listp (x)
   (if (endp x)
       (null x)
     (and (q-avp      (car x))
          (q-av-listp (cdr x))))))
```

Figure 3.1: Model of a query

Databases

Databases (*a.k.a. name trees*) are another special kind of av-trees. There are two differences between the structure of queries and databases. First, there is no constraint on the number of children for a node in a database other than those on av-trees. Second, a value node in a database contains "pointers" to records. So, for a database value node, `data` is another list of the form `(value . records)`. So, a value rooted database tree is a list of the form `((value . records) . children)`, where `children` is a list of attribute rooted database trees. The mutually recursive functions in Figure 3.1.1 recognize a well-formed database.

Reading the recognizer for a database tree (Figure 3.1.1) clause-by-clause: `(1.1)` and `(2.1)` ensure that the node is a `consp` (recall that trees are modeled as `consp`'s; a singleton tree is one whose `cdr` is an `endp`). `(2.2)` and `(2.3)` ensure that a value node's `data` is composed of two parts, namely a (value) label and a record set. `(1.3)` and `(2.4)` check that the children of a node are mutually unique,

```
(mutual-recursion
 (defun db-avp (x)                    ; (1)
   (and (consp x)                     ; (1.1)
        (consp (children x))          ; (1.2)
        (no-dup-vals-p (children x)) ; (1.3)
        (db-va-listp (children x)))) ; (1.4)
 (defun db-vap (x)                    ; (2)
   (and (consp x)                     ; (2.1)
        (consp (data x))              ; (2.2)
        (true-listp (rec x))          ; (2.3)
        (no-dup-atts-p (children x)) ; (2.4)
        (db-av-listp (children x)))) ; (2.5)
 (defun db-av-listp (x)               ; (3)
   (if (endp x)                       ; (3.1)
       (null x)                       ; (3.2)
     (and (db-avp      (car x))      ; (3.3)
          (db-av-listp (cdr x)))))   ; (3.4)
 (defun db-va-listp (x)               ; (4)
   (if (endp x)                       ; (4.1)
       (null x)                       ; (4.2)
     (and (db-vap      (car x))      ; (4.3)
          (db-va-listp (cdr x))))))  ; (4.4)
```

Figure 3.2: Model of a database

ensuring that the database tree is "unambiguous." `(1.2)` ensures that an attribute node is always followed by at least one value node. `(1.4)` ensures that the children of an attribute node are value nodes (*i.e.* they are well-formed value-rooted database trees). Similarly, `(2.5)` ensures that the children of a value node are attribute nodes. We define `NIL` to be the empty list of trees, and hence lines `(3.2)` and `(4.2)`.

3.1.2 Lookup-Name

As mentioned before, Lookup-Name is the name resolution algorithm of INS. Given a query and a database, the Lookup-Name algorithm retrieves the set of all records in the database that match the query. It does so by first initializing the result set to all possible records in the database, and then reducing the result set through a series of recursive calls. It is a depth-first algorithm that recurs on both of its inputs.

The corrected version of Lookup-Name published in [4] is shown in Algorithm 1. `Lookup-name`,[2] shown in Figure 3.3, is the ACL2 model of the corrected Lookup-Name algorithm.

In the definition of `lookup-name`, `all-recs` is a function that collects all of the records in a database. `Att-assoc` is a function that reruns the first (attribute-rooted) tree in its second argument that has a root matching the tree in its first argument. `Val-assoc` works in a similar manner and is used for . (These two functions are analogous to the `assoc` function.) `Int` defines the set intersection in our `sets` book, which is a light-weight set theory.[3] `Compute-s-prime` is a function that performs the computation corresponding to the lines 8 - 12 of Algorithm 1.

Note that, again, we are using mutual recursion. This is due to the fact that ACL2 does not have any looping constructs. As a consequence, the for loop

[2] ACL2 is case-insensitive. Therefore `Lookup-name` and `lookup-name` are the same symbols.

[3] We developed a light-weight set theory book in order to reason about sets of records. The interested reader is referred to Appendix A.1.2 for the details of this set theory.

Algorithm 1 Lookup-Name

```
    Lookup-Name(T, n)
 1: S ← the set of all possible records
 2: for each av-pair p := (n_a, n_v) in n do
 3:    T_a ← the child of T such that T_a's attribute = n_a's attribute
 4:    if T_a = null then
 5:       return ∅
 6:    end if
 7:    if n_v = * then {wild card matching}
 8:       S' ← ∅
 9:       for each T_v which is a child of T_a do
10:          S' ← S' ∪ all the records in subtree rooted at T_v and its subtrees
11:       end for
12:       S ← S ∩ S'
13:    else {normal matching}
14:       T_v ← the child of T_a such that T_v's value = n_v's value
15:       if T_v = null then
16:          return ∅
17:       end if
18:       if p is a leaf node then
19:          S ← S ∩ all the records in subtree rooted at T_v and its subtrees
20:       else
21:          S ← S ∩ Lookup-Name(T_v, p)
22:       end if
23:    end if
24: end for
25: return S
```

```
(mutual-recursion
 (defun lookup-name (db q)
   (declare (xargs :measure (1+ (acl2-count q))))
   (let ((db-children (children db))
         (q-children  (children q))
         (s           (all-recs db)))
     (lookup-name-for db-children q-children s)))
 (defun lookup-name-for (db-av-list q-av-list s)
   (let* ((qa  (car q-av-list))
          (qv  (car (children qa)))
          (dba (att-assoc qa db-av-list))
          (dbv (val-assoc qv (children dba))))
     (cond
      ((endp q-av-list) s)
      ((endp dba) nil)
      ((eq (r-val qv) '*)                        ;; Wild-card matching
       (compute-s-prime (children dba)))
      ((endp dbv) nil)
      ((endp (children qv))
       (lookup-name-for
        db-av-list (cdr q-av-list) (int s (all-recs dbv))))
      (t
       (lookup-name-for
        db-av-list (cdr q-av-list) (int s (lookup-name dbv qv)))))))))
```

Figure 3.3: Model of the Lookup-Name algorithm

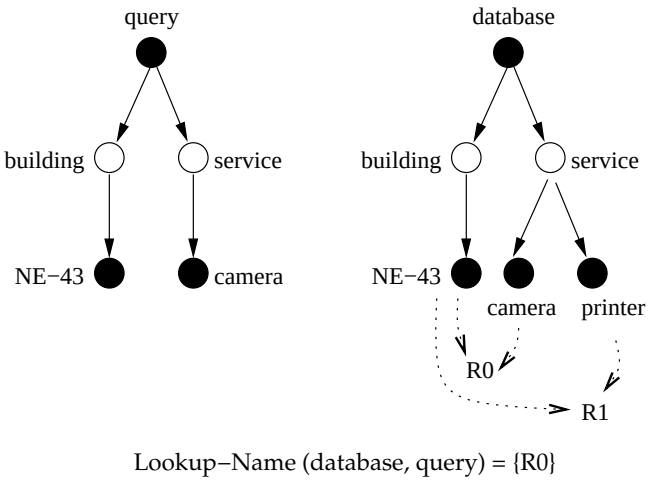

Figure 3.4: An Example of the Lookup-Name Operation

corresponding to the lines 2 - 24 must be modeled as a recursive function. Since there is a recursive call to Lookup-Name inside the for loop, the two functions `lookup-name` and `lookup-name-for` have to be defined mutually-recursively. (The for loop corresponding to the computation of S' also has to be modeled as a recursive function, and hence `compute-s-prime`; but notice that it is not part of the mutual recursion because it does not contain a call to any of the functions in the mutual-recursion.)

The task of `lookup-name` is to initialize S (line 1 of Algorithm 1), and the task of `lookup-name-for` to the rest of the computation. Almost the entire actual computation is done in `lookup-name-for`. An example of the Lookup-Name operation is shown in Figure 3.4.

3.2 Correctness Specification

The correctness of `lookup-name` may be informally stated as follows: a record `r` is in the result set of `lookup-name` on the query `q` and the database `db` if and only if `r` *satisfies* `q` in `db`.

3.2.1 Notion of Satisfiability

To express the notion of *satisfies*, we have to define several concepts.

Path Set

Define a path p in a tree T to be a sequence of nodes such that the first element of p is a node in T and every $i + 1^{st}$ node in p is a child of the i^{th} node in T.

Now define the path set of T to be the set of all paths in T such that the first element of the path is the root of T and the last element of the path is a tip of T. (As a consequence, the cardinality of the path set of a tree is equal to the number of the tips of that tree.) Let *flattening* a tree mean to enumerate the elements of the path set of that tree. Below we define `flatten1` to perform this operation.

```
(mutual-recursion
 (defun flatten1 (x)
   (if (tipp x)
       (list (list (data x)))
     (fold-cons (data x) (flatten1-list (children x)))))
 (defun flatten1-list (xs)
   (if (endp xs)
       nil
     (append (flatten1      (car xs))
             (flatten1-list (cdr xs))))))
```

Reachability

We say that *a record r is reachable in path p_2 through path p_1* if the following two conditions hold: first, p_1 is a prefix of p_2; second, r is in the suffix of p_2 resulting from chopping off p_1.

In order to formalize the notion of reachability described above, we first have to formalize the concept of a record being in a path. `In-path` defines this concept.

```
(defun in-path (flg r path)
  (cond
   ((endp path) nil)
   ((eq flg 'va)
    (or (member r (rec (car path)))
        (in-path 'av r (cdr path))))
   (t (in-path 'va r (cdr path)))))  ;; No records in an att node.  Skip.
```

Since in an av-tree the type of the nodes alternate from one level to the next, a path in an av-tree will be a sequence of nodes with alternating types. We will call this an *av-path*. The argument `flg` in the definition of `in-path` is a flag that specifies the type of the first node in `path`. If the flag is `va`, then the first element on the path is a value node (it is an attribte node otherwise). The type of the first node on a path determines the type of the rest of nodes on the path, just as with av-trees where the type of the root node determines the types of the rest of the nodes.

Having defined `in-path`, we can define `reachable` which determines if a record `r` is in a path `p2` through another path `p1`.

```
(defun reachable (flg r p1 p2)
  (if (endp p1)
      nil
    (if (eq flg 'va)
        (cond
```

```
        ((eq '* (val (car p1)))
         (in-path 'va r p2))
        ((equal (val (car p1))
                (val (car p2)))
         (if (endp (cddr p1))
             (in-path 'va r p2)
           (reachable 'av r (cdr p1) (cdr p2))))
        (t nil))
      (cond
       ((equal (att (car p1))
               (att (car p2)))
        (reachable 'va r (cdr p1) (cdr p2)))
       (t nil)))))
```

With `reachable` we can only express the reachability of a record in one path through another path. However, recall that trees are made of a set of paths, and therefore in order to express the notion of a record satisfying a query tree in a database tree, we must be able to express the reachability of a record through a set of paths in another set of paths. Below we define `member-reachable`, to express the reachability of a record in a set of paths through a single path, and `subset-reachable`, to express the reachability of a record in a set of paths through another set of paths.

```
(defun member-reachable (flg r qp dbp-list)
  (if (endp dbp-list)
      nil
    (or (reachable        flg r qp (car dbp-list))
        (member-reachable flg r qp (cdr dbp-list)))))


(defun subset-reachable (flg r qp-list dbp-list)
  (if (endp qp-list)
      t
    (and (member-reachable flg r (car qp-list) dbp-list)
         (subset-reachable flg r (cdr qp-list) dbp-list))))
```

Satisfiability

Having defined the notions of path set and reachability, we can now define the notion of "satisfies" discussed earlier.

```
(defun satp (r q db)
  (subset-reachable 'va r (flatten1 q) (flatten1 db)))
```

Another, yet more formal way to think about "satisfies" is the following: `r` satisfies `q` in `db` if and only if the path set of `q` is a `subset-reachable` of the path set of `db` with respect to `r`.

3.2.2 Statement of the Correctness Theorem

Having defined the notion of satisfiability, we can state the correctness theorem for `lookup-name` as follows. (`Databasep` and `queryp` are aliases for `db-vap` and `q-vap`, which are recognizers for a value-rooted database tree and a value-rooted query tree, respectively.)

```
(defthm lookup-name-correct
   (implies (and (databasep db)
                 (queryp q))
            (iff (member r (lookup-name db q))
                 (satp r q db))))
```

3.2.3 Complexity of the Formalization

The first and perhaps the most difficult aspect of our formalization of the problem is that there is a lot of mutual recursion involved. In general, mutually recursive functions are somewhat awkward to reason about formally. In our case, we have four-way mutually recursive functions, two of which are due to the data structures being trees (with unbounded branching) and two are due to the alternation of the

node types. We have tried to eliminate some of the mutual recursion, but we never succeeded.

Another problem arises from the fact that Lookup-Name examines two consecutive nodes at a time, and compares the nodes in a rather out-of-order manner. Looking at the algorithm, we see that Lookup-Name always assumes that the root of the tree has already been matched and starts by looking at the immediate children of the root. Of course, it always matches the roots of the two trees before the next recursive call. This "look-ahead matching" also adds to the complexity of the challenge.

We have, however, developed a strategy to attack the problem. We observe that the operation of Lookup-Name can be viewed as a two-step operation: given a pair of query and database, it first determines if the query is "contained" in the database; then, if the result of the first problem is positive, it determines if a record appears in every subtree to which the tips of the first subtree point. The next chapter will address the first of these two problems.

Chapter 4

The Simplified Problem

In this chapter, we describe a simplified version of the Lookup-Name algorithm, which we call Tree-Containment. We then formalize this algorithm, specify it, and prove it correct.

In order to eliminate the complexity of the problem, we make three simplifying assumptions:

- We abstract out the notion of a record, thereby changing the reachability problem into a *prefixness* problem. We define the prefixness problem as follows: given two paths, determine if one is a prefix of the other. This is the most important simplification we make.
- We assume that the nodes of a tree are homogeneous. This reduces the complexity of our mutual recursions by a factor of two.[1]
- Eliminate the notion of a wild card.

[1] By the complexity of a mutual recursion we mean the number of functions in it.

4.1 The Tree Containment Problem

Recall that we can think of Lookup-Name as an algorithm that performs two tasks: given two trees T_1 and T_2, it determines if T_1 matches (or is "contained" in) T_2; if so, it then returns the intersection of the records of the subtrees of T_2 to which the tips of T_1 point. In other words, Lookup-Name solves two problems, namely the "tree containment" problem and the "data collection" problem. We believe that we can first focus on the (more difficult) tree containment problem and address it separately, and then focus on the data collection problem.

We can informally state the tree containment problem as follows: given a pair of trees, decide whether the first is contained in the second. We say that a tree is "contained" in another if and only if at each level, all of the nodes of the first tree have a match in the second tree at the same level, and the parent of the every node in the first tree is the same as that of its match in the other tree. We assume no restriction on the structure of the trees. Also, no (wild card) symbol is to be interpreted in a special way. Moreover, both input trees are of the same type.

4.2 The Model

In this section, we formalize the Tree-Containment algorithm. We present our ACL2 model of the data structure used and then model the algorithm.

4.2.1 Data Structure

As a consequence of our abstracting out the notion of a record, the nodes in the new tree will not be composite as they were with Lookup-Name (*i.e.* a label and a record set contained in the value nodes). For the sake of simplicity, we assume

```
(mutual-recursion
 (defun atreep (x)
   (if (atom x)                                          ; (1)
       t
     (and (atom  (car x))                                ; (2.1)
          (consp (cdr x))                                ; (2.2)
          (atree-listp (cdr x)))))                       ; (2.3)
 (defun atree-listp (xs)
   (if (endp xs)
       (null xs)                                         ; (3)
     (and (atreep (car xs))                              ; (4.1)
          (not (member (root        (car xs))            ; (4.2)
                       (strip-roots (cdr xs))))
          (atree-listp (cdr xs))))))                      ; (4.3)
```

Figure 4.1: Model of the simplified tree

that the nodes in our trees are `atoms`. Unlike av-trees, there is no restriction on the depth of the tip nodes.[2] We will model a singleton tree as an `atom`, and a forest as a list of the form `(root . children)`, where every element of `children` is another well-formed tree (*i.e.* a single-noded tree or another forest).

The formal definition of a well-formed tree is given by `atreep` and its mutually recursive counterpart `atree-listp` as shown in Figure 4.2.1.

We now briefly explain the tree recognizer clause-by-clause. `(1)` recognizes a singleton tree. `(2)` recognizes a forest; `(2.1)` requires that the root of the forest be an atom, and `(2.2)` ensures that the node has at least one child (since it is a forest; otherwise it would have been recognized by clause `(1)`). `(2.3)` ensures that the children of `x` are a list of well-formed trees.

`Atree-listp` recognizes a list of "sibling" trees. We require that all siblings have unique roots. The reason we put this restrictions on sibling trees is that we

[2] In an av-tree, all tips have to appear at even levels. This is because only value nodes can appear as tips, and a value node can only be at an even level.

are trying to model trees to resemble those of INS, namely av-trees.[3] Unambiguity also has a significant impact on the way we define the notion of correctness. We define `NIL` to be an empty list of trees, hence `(3)`. `(4)` recognizes a non-empty list of sibling trees.

Note that if `atree-listp` were to recognize an arbitrary list of trees, then we would use clauses `(4.1)` and `(4.3)` only. But we add clause `(4.2)` to check for the unambiguity constraint. Equivalently, we could have performed this check in `atreep` with `(no-duplicatesp (strip-roots (cdr x))`. Obviously, this would be much more efficient than the current implementation because it would require only one call to `strip-roots`. Also, one might argue that the latter way of defining the recognizer is preferable because that way `atree-listp` would be a recognizer for an arbitrary list of trees, whereas this definition of the recognizer is for a list of trees that have mutually unique roots. That is, it recognizes a list of "sibling" trees.[4] But we claim that our choice of defining the recognizer this way is a reasonable one because we will always be reasoning about a list of siblings; we will never have to reason about an arbitrary list of trees. Since we always will be only concerned with "proving theorems" that involve these functions, and our definition of the tree recognizer will make it much easier to reason about them than if they were defined in the other way, we insist that our definitions of `atreep` and `atree-listp` are most suitable for our purposes.

[3] In INS trees, a node has unique children, which is essential when searching a list of siblings for a match.

[4] As a matter of fact, this is how we checked for unambiguity in our INS tree recognizers an in our original definition of the recognizer. But when proving the completeness theorem for the simplified problem, we realized that this way of defining trees is best.

```
(mutual-recursion
 (defun contdp (t1 t2)
   (declare (xargs :measure (+ (acl2-count t1)
                               (acl2-count t2))))
   (cond
    ((not (requal t1 t2)) nil)
    ((tipp t1) t)
    ((tipp t2) nil)
    (t (contd-listp (chdn t1) (chdn t2)))))
 (defun contd-listp (t1-list t2-list)
   (declare (xargs :measure (+ (acl2-count t1-list)
                               (acl2-count t2-list))))
   (if (endp t1-list)
       t
     (and (contdp-aux  (car t1-list) t2-list)
          (contd-listp (cdr t1-list) t2-list))))
 (defun contdp-aux (t1 t2-list)
   (declare (xargs :measure (+ (acl2-count t1)
                               (acl2-count t2-list))))
   (if (endp t2-list)
       nil
     (or (contdp     t1 (car t2-list))
         (contdp-aux t1 (cdr t2-list))))))
```

Figure 4.2: Model of the Tree-Containment algorithm

4.2.2 Model of the Tree Containment Algorithm

Since our objective behind working on this simplified problem is to develop the intuition of how to solve the actual problem (of verifying Lookup-Name), we will define an algorithm to solve the tree containment problem that is similar to Lookup-Name.

Define a depth-first algorithm to decide tree containment as follows: if the roots of the trees are different, then the answer is negative. Otherwise, determine if every child of the first tree is contained in one of the children of the second tree.

Figure 4.2.2 shows the model of `contdp` in ACL2.

`Contdp` decides if tree `t1` is contained in `t2`. `Contd-listp` decides if a list of trees `t1-list` are contained in another list of trees `t2-list`, *i.e.* every tree in `t1-list` is contained in at one of the trees in `t2-list`. Finally, `contdp-aux` decides if a tree `t1` is contained in one of the trees in `t2-list`.

Reading these three mutually recursive functions clause-by-clause, `(1)` if the roots of the trees do not match, then `t1` is not contained in `t2`. `(2)` if the roots of the trees match and the `t1` tree is a leaf node, then the `t1` tree is contained in `t2`. `(3)` if `t2` is a leaf node but `t1` is not, then `t1` is not contained in `t2`. `(4)` otherwise, *i.e.* if the roots of the trees match and none of them is a leaf, determine if every tree in the list of the children of `t1` is contained in one of the trees in the list of the children of `t2`. `(5)` if we have exhausted the list of the children of `t1`, *i.e.* if all of the children of `t1` were contained in one of the children of `t2`, then `t1`, is contained in `t2`. `(6.1)` otherwise, *i.e.* if there are some children of `t2` left to check, then the first child in the list of the children of `t1` must be contained in at least one the trees in the list of the children of `t2`, and `(6.2)` the every other child of `t1` must be contained in a child of `t2`. The auxiliary function `contdp-aux` determines if a single tree is a contained in a list of trees. Basically, `(7)` says that if the list of the children of `t2` has been exhausted, then the search for a match has failed, so `t1` is not contained in `t2`. `(8.1)` if `t1` (a subtree of `t1` in `contdp`) contained in the first element of `t2-list` (a subtree of `t2` in `contdp`) then a match was found. `(8.2)`, otherwise, continue searching the rest of list of the children of `t2` in `contdp`.

The reader might notice that the two functions `contd-listp` and `contd-aux` are very similar in structure to the set theory function `subsetp` and `member`.[5] This is not surprising. In fact, one way to think about what the tree containment algorithm does is that it checks if two trees have the same roots, and then checks if the children

[5]These functions are define in ACL2, in our set theory book.

of one are a "subset" (up to tree-containment) of the children of the other. The main difference between these functions is that `subsetp` and `memberp` use `equal` to compare two objects for equality, whereas `contd-listp` and `contd-aux` use `contdp` to check for tree-containment. This observation will help us through out the proof process.

Below is an example of some well-formed and ill-formed trees.

```
(defthm Example0
  (let ((t1 'A)
        (t2 '(A))
        (t3 '(A B))
        (t4 '(A B C))
        (t5 '(A (B C) D))
        (t6 '(A (B (C D))))
        (t7 '(A B B))
        (t8 '(A (B D) (C D)))
        (t9 '((A D) B C)))
    (and
     (atreep t1)
     (not (atreep t2))
     (atreep t3)
     (atreep t4)
     (atreep t5)
     (atreep t6)
     (not (atreep t7))
     (atreep t8)
     (not (atreep t9))))
  :rule-classes nil)
```

The above conjuncts inform us that `t1` is a well-formed (singleton) tree. `t2` is not a tree because it is not a singleton tree (since it is a `consp`), but it does not have any children, which is impossible. `t3` is well-formed because it is a tree with two nodes `A` and `B`, where `A` is the root and `B` is the child of `A`. `t4`, `t5`, and `t6` are well-formed. `t7` is ill-formed because the root of the tree, `A`, has two identical

children, and thus it is ambigous. `t9` is ill-formed because the root of a tree must be an atom, *i.e.* a single node, whereas this tree has another tree as its root.

4.2.3 Termination Argument

In order to get ACL2 to admit a function, we must prove that it always terminates on any input. For non-recursive functions this task is trivial. On the other hand, proving termination for recursive functions requires finding some measure that decreases in every recursive call, which can be difficult at times. It can be even more difficult for mutually recursive functions, functions with reflexive definitions [2], and functions that recur on an argument before performing certain checks.

Let's consider `contdp`. Why does this function terminate? If we remove the specified measures in the three mutually recursive functions defined in Section 4.2.2 and submit the functions to ACL2, the theorem prover's heuristics will not find an appropriate measure and the mutually recursive functions will not be admitted.

As mentioned before, to prove termination for recursive functions, find a measure that decreases in every recursive call. This task is relatively easy for singly recursive functions because there is only one measure to be dealt with. However, for mutually recursive functions, we must do more work – we have to prove that a measure of the arguments of every function in the mutually recursive clique is smaller than the measure of every call to a mutually recursive function. We found the measures for the three mutually recursive functions in the model of Tree-Containment as follows: let `m1`, `m2`, and `m3` be the measures for `contdp`, `contdp-list`, and `contdp-aux` respectively. We must find `m1`, `m2`, and `m3` such that:

- `(m2 (chdn t1) (chdn t2)) < (m1 t1 t2)`
- `(m3 (car t1-list) t2-list) < (m2 t1-list t2-list)`

- `(m2 (cdr t1-list) t2-list) < (m2 t1-list t2-list)`
- `(m1 t1 (car t2-list)) < (m3 t1 t2-list)`
- `(m3 t1 (cdr t2-list)) < (m3 t1 t2-list)`

We observe that the sum of the `acl2-count`'s of the arguments of each function satisfies the above conditions.

Notice that clauses `(2)` and `(3)` in the definition of `contdp` are not necessary; that is, we could have removed those two clauses and still define an equivalent function. To see why, let's consider what would happen if we removed those two checks: for non-`tipp` `t1` and `t2`, the behavior of the function would be unchanged. But suppose `t1` were a `tipp` (*i.e.* `(chdn t1)` is an `endp`), then the flow of the execusion would reach `(contd-listp (chdn t1) (chdn t2))`, which would return `t` since `t1-list` is bound to an `endp` (because `(chdn t1)` is an `endp`). Similarly, if `t2` were a `tipp` (*i.e.* `(chdn t2)` is an `endp`), then the control flow would reach `(contd-listp (chdn t1) (chdn t2))`, which would then call `(contdp-aux (car t1-list) t2-list)` in `contd-listp` (assuming that `t1` is not a `tipp`), which would be `NIL` because `(contdp-aux anything endp)` would be `NIL`. So if we omit caluses `(1)` and `(2)` in `contdp`, those cases would be handled by the base cases of `contd-listp` and `contdp-aux`. But we chose to test for and handle these two cases in `contdp` because this way, the measures for our mutually recursive functions would be straight forward, and therefore their termination proof would be much easier. More importantly, this choice would make it much easier to reason about these mutually recursive functions.

4.3 Verification

In this section, we develop a notion of correctness for the Tree-Containment algorithm and then use it to prove `contdp` correct.[6]

4.3.1 Specification

There may be many different ways to formalize the notion of correctness for the Tree-Containment algorithm. Since, again, the purpose of solving the tree containment problem is to learn how to solve the more complex problem of verifying the INS Lookup-Name, we will define a notion of correctness similar to the one we defined in Section 3.2.

We desire `contdp` to have the following property.

```
(defthm contdp-correct
  (implies (and (atreep x)
                (atreep y))
           (iff (contdp x y)
                (satp x y))))
```

That is to say, given a pair of well-formed trees `x` and `y`, `x` is contained in `y` if and only if `x` *satisfies* `y`. The notion of satisfiability in the tree containment problem is similar to that we defined for Lookup-Name. However, we will neeed to replace the notion of reachability by a weaker one.

Path Set

We use the same definition of path set and `flatten` for enumerating the members of the path set for a tree. The following example demonstrates the function of `flatten`.

[6] We do not explain the proof in this draft. But the ACL2 proof scripts are in appendix A.3.

```
(defthm Example2
  (let ((t6 '(A (B E F) (C G (H I)) D)))
    (equal
     (flatten t6)
     '((A B E)
       (A B F)
       (A C G)
       (A C H I)
       (A D))))
  :rule-classes nil)
```

Prefixness

We now define the most fundamental function in our definition of the notion of correctness for `contdp`. We defined the prefixness problem earlier. We define the function `prefixp` below which decides prefixness. Given a pair of paths `p1` and `p2`, `prefixp` determines if `p1` is a prefix of `p2`. The reader might notice the structural similarity between this function and `reachable` which we defined for the notion of correctness for Lookup-Name. The difference between the two functions is that `reachable` answers two questions: (1) is `p1` a prefix of `p2` and (2) does a record `r` appear along the suffix of `p2` after chopping off `p1`. But `prefixp` only answers the first question.

```
(defun prefixp (p1 p2)
  (cond
   ((endp p1) t)
   ((endp p2) nil)
   (t (and (equal   (car p1) (car p2))
           (prefixp (cdr p1) (cdr p2))))))
```

`Are-prefixes-in`, defined below, checks that every path in its first argument is a prefix of another in its second argument.

```
(defun are-prefixes-in (p-list1 p-list2)
  (if (endp p-list1)
      t
    (and (is-a-prefix-in  (car p-list1) p-list2)
         (are-prefixes-in (cdr p-list1) p-list2))))
```

`Is-a-prefix-in`, defined below, determines if a path `p` is a prefix of at least one path in a set of paths `p-list`.

```
(defun is-a-prefix-in (p p-list)
  (if (endp p-list)
      nil
    (or (prefixp        p (car p-list))
        (is-a-prefix-in p (cdr p-list)))))
```

Note that `are-prefixes-in` and `is-a-prefix-in` are structurally very similar to `subset-reachable` and `member-reachable` in the definition of the notion correctness for Lookup-Name in Section 3.2.1. Also, note that the structure of the two set-theory functions `subsetp` and `memberp` are very similar to that of `are-prefixes-in` and `is-a-prefix-in` the definition of notion correctness for `contdp`. The only difference between these functions and the set-theory functions is that `are-prefixes-in` and `is-a-prefix-in` are only concerned with a weaker condition than equality, namely with prefixness, and therefore we use `prefixp` for comparison. The set theory functions, on the other hand, are concerned with equality and therefore use `equal` for comparison. (Of course, one fundamental difference is that `equal` is an equivalence relation but `prefixp` is not; but that is perfectly fine because `contdp` is not an equivalence relation either.)

Having said that, we can think of a tree satisfying another tree as follows: a tree satisfies another if and only if its paths set is a subset, up to prefixness, of the paths set of the other tree.

```
(defun satp (t1 t2)
  (are-prefixes-in (flatten t1) (flatten t2)))
```

4.3.2 Proof

In this section, we give a high-level overview of the correctness proof for Tree-Containment. The complete proof script is in Section A.3.3.

We split the correctness theorem into a soundness and a completeness theorem. Informally stated, assuming that `x` and `y` are well-formed trees, the soundness theorem is "if tree `x` is recognized to be contained in `y` by `contdp`, then `x` satisfies `y`;" the correctness theorem is "if tree `x` satisfies `y`, then `contdp` recognizes `x` to be contained in `y`."

Below are the the formulas for the soundness and completeness theorems for `contdp`.

```
(defthm contdp-sound
  (implies (and (atreep x)
                (atreep y)
                (contdp x y))
           (satp x y)))


(defthm contdp-complete
  (implies (and (atreep x)
                (atreep y)
                (satp x y))
           (contdp x y)))
```

We would like to prove these two theorems by induction. Note that `atreep` and `contdp` are mutually recursive. When we prove theorems about mutually recursive functions by induction, we have to state the theorem about all of the functions in the mutually recursive clique. To see why, consider the following example. Suppose $f(x)$ and $g(x)$ are mutually recursive functions, *i.e.* f makes a call to g and

vice versa. Now imagine a theorem that states "f has a property p" as follows: *hypotheses* $\Rightarrow p(f(x))$. Denote this formula by $a(x)$. To prove this theorem by induction, we must assume some hypothesis of the form $a(x')$, where x' is some smaller piece of x that is passed to g. But $a(x)$ does not involve g. Hence we cannot assume $a(x')$, and therefore $a(x)$ cannot be proved by induction. Similarly, the soundess and completeness theorems cannot be inductively proved as they are stated.

We could make the above theorems to be a conjunction of implications involving appropriate combinations of the functions `contdp`, `contdp-list`, `contdp-aux`, `atreep`, and `atree-listp`. This method would work fine; however, it often requires the user to supply induction hints for every theorem that involves mutually recursive functions. We need to prove many such theorems, so we seek an alternative method to ease the theorem proving process. Our approach is to convert every mutually recursion to a flagged singly recursive form; establish the equivalence of the singly recursive functions to their corresponding mutual recursion; and then prove all of our theorems about the singly recursive functions. In our case, we define `atreep-fn` with a two-way flag – one corresponding to `atreep` and `atree-listp` – and `contdp-fn` with a three-way flag.[7] This method has the advantage that by instantiating the flag we can capture all appropriate combinations of the mutually recursive functions. To do so, all we have to do is to use an induction scheme that involves the flag to one of the mutually recursive functions.

Soundness

The following formula is the statement of the theorem for the soundness of the singly recursive form of `contdp`.

[7] See the proof script in Section A.3.3 for the details.

```
(defthm contdp-sound-general-case
  (implies (and (contdp-fn flg x y)
                (equal flg1 (flg1 flg))
                (equal flg2 (flg2 flg))
                (atreep-fn flg1 x)
                (atreep-fn flg2 y))
           (are-prefixes-in
            (flatten-fn flg1 x)
            (flatten-fn flg2 y))))
```

In the above formula, the functions `flg1` and `flg2` are used to determine the appropriate flag for `atreep-fn`. More precisely, they are used to determine the structure of the trees `x` and `y`. For instance, when `flg` is `'aux`, *i.e.* when `contdp-fn` corresponds to `contdp-aux`, we want the first argument of `contdp-fn` to be a single tree and the second argument to be a list of trees. So `(flg1 'aux)` will be `'tree` and `(flg2 'aux)` will be `'list`, and therefore the `atreep-fn` function will ensure that `x` is a single tree and `y` is a list of trees.

In order to prove the above theorem we have to prove a number of lemmas relating `are-prefixes-in` and the functions that `flatten-fn` is made of, such as `fold-cons` and `append`. Some of these theorems include, `are-prefixes-in` distributes over `append` in its first formal; also, `append` is commutative in the second formal of `are-prefixes-in`. Many times to prove a lemma about `are-prefixes-in` we have to prove an analogue lemma about `is-a-prefix-in`.

Completeness

The following is the completeness theorem for `contdp`.

```
(defthm contdp-complete-general-case
  (implies (and
            (atreep-fn (flg1 flg) x)
            (atreep-fn (flg2 flg) y)
            (are-prefixes-in
             (flatten-fn (flg1 flg) x)
             (flatten-fn (flg2 flg) y)))
           (contdp-fn flg x y)))
```

Most of the lemmas proved in the soundness proof relating `are-prifexes-in` and `is-a-prefix-in` to functions like `append` are used in the completeness proof. Additionally, we must prove lemmas relating `are-prifexes-in` and `is-a-prefix-in` to `prefixp`. Many of the lemmas for the completeness proof make extensive use of the assumption that the children of a node are unique. One such theorem, `are-prefixes-in-append`, allows us to distribute `are-prefixes-in` over `append` in the second argument of `are-prefixes-in`. Four corollaries of this lemma were used in the proof of the completeness theorem.

In total, we proved 26 definitions and 56 lemmas to prove the correctness of `contdp`. 20 out of these 26 definitions were established to obtain the correctness theorem and the remaining were introduced in the proof (*e.g.* conversion of the mutual recursions to singly recursive form, new functions to reason about the type of certain functions, etc.). Our the 56 lemmas, 15 were list-processing facts; 16 were used in the soundess proof. The remaining lemmas, toghether with many of the lemmas we proved in the soundness proof helped us establish the completeness of `contdp`.

4.4 Incrementally Extending Simple to Actual Model

In this section, we breifly describe how the simplified problem of Tree-Containment can be incrementally changed back to the original problem of Lookup-Name.[8]

Recall that the most significant simplification we made in the tree containment problem was to disregard the notion of reachability. Our first step to extend the tree containment problem is to change the `contd` function so that it returns the set of all nodes in the subtree(s) of the second tree to which the tip(s) of the first tree point. We will call this new function `lookup-node`.

In order to formalize this new function, we have to define another function, an iterator, that collects all of the nodes in a tree. Below we define `all-nodes` to collect the nodes of a tree.

```
(mutual-recursion
 (defun all-nodes (x)
   (if (tipp x)
       (list (root x))
     (cons (root x)
           (all-nodes-list (chdn x)))))
 (defun all-nodes-list (xs)
   (if (endp xs)
       nil
     (append (all-nodes      (car xs))
             (all-nodes-list (cdr xs))))))
```

Having defined `all-nodes`, we modify `contdp` and its mutually recursive counter-parts as follows: replace all occurrences of `t` by an appropriate call to `all-nodes-list`; then change all occurrences of `and` and `or` by `int` and `unn`, respectively.[9] The occurrences of `nil` will not be changed. But notice that `nil` meant

[8] We have not yet experimented with this approach, and so we do not have any results to support it. But doing so is part of our future work plans (see Section 5.2).

[9] `Int` and `unn`, defined in the `sets` book, compute the intersection and the union of sets, respectively. See Appendix A.1.2 for more details.

false in the context of `contdp`, but now it denotes *the empty set*. Below is the definition of `lookup-node`.

```
(mutual-recursion
 (defun lookup-node (t1 t2)
   (declare (xargs :measure (+ (acl2-count t1)
                               (acl2-count t2))))
   (cond
    ((not (requal t1 t2)) nil)
    ((tipp t1) (all-nodes-list (chdn t2)))
    ((tipp t2) nil)
    (t (lookup-node-listp (chdn t1) (chdn t2)))))
 (defun lookup-node-listp (t1-list t2-list)
   (declare (xargs :measure (+ (acl2-count t1-list)
                               (acl2-count t2-list))))
   (if (endp t1-list)
       (all-nodes-list t2-list)
     (int (lookup-node-aux   (car t1-list) t2-list)
          (lookup-node-listp (cdr t1-list) t2-list))))
 (defun lookup-node-aux (t1 t2-list)
   (declare (xargs :measure (+ (acl2-count t1)
                               (acl2-count t2-list))))
   (if (endp t2-list)
       nil
     (unn (lookup-node     t1 (car t2-list))
          (lookup-node-aux t1 (cdr t2-list))))))
```

Note that our use of the function `unn` in `lookup-node-aux` is justified because we assume that the arguments to `contdp` are well-formed trees, which means that no two siblings are the same. This implies that if there is a match for `t1` in `t2-list`, then it is unique.

The following example demonstrates the function of `lookup-node`.

```
(defthm Example3
  (let ((t1 'A)
        (t2 '(A B C))
        (t3 '(A (B F (H I D)) (C (E F G) D))))
```

```
  (and
   (set-equal (lookup-node t1 t2)
              '(B C))
   (set-equal (lookup-node t1 t3)
              '(B C D E F G H I))
   (set-equal (lookup-node t2 t3)
              '(F D))))
:rule-classes nil)
```

We should be able to use a very similar notion of correctness as the one we developed for `lookup-name` to specify `lookup-node`. We also should be able to prove the correctness of `lookup-node` with similar lemmas to those in the proof of the correctness of `contdp`. Note that the main differences between `lookup-node` and `lookup-name` are that the former assumes that the nodes are *atomic*, and it uses the entire node for comparison and also returns whole nodes in its result set; however, the former assumes that there are two parts to the nodes of a tree (*i.e.* the nodes are *composite*) – a label and a record set – and it uses the label for comparison and returns records as its result set. In addition, `lookup-nodes` does not know the notion of attributes and values, and therefore does not discriminate the nodes when constructing its result set. On the other hand, `lookup-name` only uses the even-leveled nodes (*i.e.* the value nodes) to construct its result set.

The next step is to introduce the notion of attributes and values. This will require two changes: modifying data structures to ensure the appropriate structure (as with `db-vap` and `q-vap`); modifying the algorithm to visit two consecuitive nodes at a time. We will call this algorithm `lookup-val-nodes`.

The previous two steps are very important in our extending `contdp` to `lookup-name`. We anticipiate two (or possibly more) other steps in the process of transforming the simple model to the complex model. Of these include introducing the concept of composite nodes for value nodes and introducing the notion of a wild-

Algorithm	Data Structure(s)	New notion / change
`contdp`	`atom` Trees	–
`lookup-nodes`	`atom` Trees	Prefixness to Reachability
`lookup-val-nodes`	`atom` AV-Trees	Heterogeneous nodes
`lookup-recds`	composite AV-Trees	Composite nodes
`lookup-name`	composite AV-Trees	Wild card

Table 4.1: Incrementally Extending the Tree-Containment

card. We call the former `lookup-recds`. After introducing the notion of records, we can add the notion of a wild-card, thereby obtaining the actual algorithm, Lookup-Name. Table 4.4 summarizes the steps to change `contdp` to `lookup-name`.

Chapter 5

Conclusion

5.1 Summary

We presented our formalization of Lookup-Name, INS's name resolution algorithm, and our attempt to verify the algorithm. We also presented an approach to solving this problem by reducing the problem to a simpler one which, we beleive, can guide us in verifying the complex model. We show a possible way of incrementally extending the simple model to the full-blown model of the Lookup-Name algorithm.

5.2 Future Work

We have only briefly described how we expect to extend the simple model and obtain the actual model. We plan to experiment with this approach and determine how it will lead us to the proof of the correctness of the original algorithm, Lookup-Name.

Bibliography

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 186–201. ACM Press, 1999. Available from: `http://wind.lcs.mit.edu/papers/`.

[2] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.

[3] Matt Kaufmann and J Strother Moore. How to prove theorems formally. 2005. Available from: `http://www.cs.utexas.edu/~moore/publications/how-to-prove-thms/index.html`.

[4] Sarfraz Khurshid and Daniel Jackson. Correcting a naming architecture using lightweight constraint analysis. 2001. Available from: `http://sdg.lcs.mit.edu/publications.html`.

Appendix A

ACL2 Event Files

A.1 Helpers

A.1.1 List Processing

```
(in-package "ACL2")

(defun rev (x)
   (if (endp x)
       nil
     (append (rev (cdr x)) (list (car x)))))

(defun fold-cons (a list)
  (if (endp list)
      nil
    (cons (cons a (car list))
          (fold-cons a (cdr list)))))

(defun fold-append (list list-list)
  (if (endp list-list)
      nil ;; may be i'll need to replace this by list-list;  if I do this, then
    ;; I can remove the (true-listp list) hypothesis in my endp-fold-append
    (cons (append list (car list-list))
          (fold-append list (cdr list-list)))))
```

```
(defun fold-list (list)
  (if (endp list)
      nil
    (cons (list (car list))
          (fold-list (cdr list)))))

(defthm append-is-assoc
  (equal (append (append a b) c)
         (append a (append b c))))

(defthm append-nil
 (implies (true-listp x)
          (equal (append x nil)
                 x)))

(defthm true-listp-append
  (implies (true-listp y)
           (true-listp (append x y))))

(defthm distrib-of-fold-cons-over-append
 (equal (fold-cons a (append list1 list2))
(append (fold-cons a list1)
(fold-cons a list2))))

(defthm distrib-of-fold-append-over-append
  (equal (fold-append x (append a b))
         (append (fold-append x a) (fold-append x b))))

(defthm endp-fold-append
  (implies (and (endp a)
                (true-listp list))
           (equal (fold-append a list)
                  list)))

; ----------------------------------------------------------------------

(defthm consp-fold-cons
  (equal (consp (fold-cons x y))
         (consp y)))
```

```
(defthm consp-append
  (equal (consp (append x y))
         (or (consp x)
             (consp y))))

(defthm car-fold-cons
  (equal (car (fold-cons x y))
         (if (endp y)
             nil
           (cons x (car y)))))

(defthm cdr-fold-cons
  (equal (cdr (fold-cons x y))
         (if (endp y)
             nil
           (fold-cons x (cdr y)))))
```

A.1.2 Set Theory

```
;; sets.lisp

(in-package "ACL2")

(include-book "list-helpers")

;; Ultra light-weight set-theory

(defun set-equal (x y)
  "Double containment"
  (and (subsetp x y)
       (subsetp y x)))

(defthm subsetp-cons
  (implies (subsetp x y)
           (subsetp x (cons a y))))

(defthm set-equal-reflexive
  (set-equal x x))

(defthm set-equal-symmetric
```

```
  (implies (set-equal x y)
           (set-equal y x)))

(defthm subsetp-reflexive
  (subsetp x x))

(defthm subsetp-transitive
  (implies (and (subsetp x y)
                (subsetp y z))
           (subsetp x z)))

(defthm set-equal-transitive
  (implies (and (set-equal x y)
                (set-equal y z))
           (set-equal x z)))

(defequiv set-equal)

;;-----------------------------------------------------------------------------

(defun empty-setp (x)
  (endp x))

(defun binary-unn (x y)
  "Computes the union of two sets"
  (cond
   ((empty-setp x) y)
   (t (cons (car x) (binary-unn (cdr x) y)))))

(defmacro unn (x y &rest rst)
  (xxxjoin 'binary-unn
           (cons x (cons y rst))))


;; Computes the union of a list of sets

(defun big-unn (sets)
  (if (endp sets)
      nil
    (unn (car sets)
         (big-unn (cdr sets)))))
```

```
(defun binary-int (x y)
  "Computes the intersection of two sets"
  (cond
   ((endp x) nil)
   ((member (car x) y) (cons (car x) (binary-int (cdr x) y)))
   (t (binary-int (cdr x) y))))


;; Computes the intersection of a list of sets

(defmacro int (x y &rest rst)
  (xxxjoin 'binary-int
           (cons x (cons y rst))))

;;---------------------------------------------------------------------------

;; Alternative definition for binary union.

;; The difference between binary-unn2 and binary-unn (above) is that
;; binary-unn2 always returns a true-listp.

(defun binary-unn2 (x y)
  (declare (xargs :measure (+ (acl2-count x) (acl2-count y))))
  (cond
   ((and (endp x) (endp y)) nil)
   ((consp x) (cons (car x) (binary-unn2 (cdr x) y)))
   (t (cons (car y) (binary-unn2 x (cdr y))))))

(defthm bin-unn-endp-true-list
  (implies (and (endp x) (true-listp y))
           (equal (binary-unn2 x y) y)))

(defthm binary-unn2=binary-unn
 (implies (true-listp y)
          (equal (binary-unn2 x y)
                 (binary-unn x y))))

(in-theory (disable binary-unn2 bin-unn-endp-true-list binary-unn2=binary-unn))

;;---------------------------------------------------------------------------
```

```
;; Theorems about unn

(defthm unn-is-assoc
  (equal (unn (unn x y) z)
         (unn x (unn y z))))

(defthm member-unn
  (iff (member a (unn x y))
       (or (member a x)
           (member a y))))

(defthm unn-subsetp1
  (subsetp x (unn x y)))

(defthm unn-subsetp2
  (subsetp y (unn x y)))

(defthm two-subsets-unn
  (implies (and (subsetp x z)
                (subsetp y z))
           (subsetp (unn x y) z)))

(defthm unn-is-comm
  (set-equal (unn x y)
             (unn y x)))

(defthm binary-unn-nil
  (implies (true-listp x)
           (equal (binary-unn x nil) x)))

(defthm binary-unn-true-listp
  (implies (true-listp y)
           (true-listp (binary-unn x y))))

;;-----------------------------------------------------------------------------

;; Theorems about int

(defthm member-int
  (iff (member a (int x y))
```

```
       (and (member a x)
            (member a y))))


;; Not only are (int (int x y) z) and (int x (int y z)) set-equal, but also
;; they are equal

(defthm int-is-assoc
  (equal (int (int x y) z)
         (int x (int y z))))

(defthm int-subsetp
  (and (subsetp (int x y) x)
       (subsetp (int x y) y)))

(defthm subsetp-int
  (implies (and (subsetp a x)
                (subsetp a y))
           (subsetp a (int x y))))

(defthm int-is-comm-set-equal
  (set-equal (int x y)
             (int y x)))

;;-----------------------------------------------------------------------------

(in-theory (disable set-equal))
```

A.2 Lookup-Name

A.2.1 Model of Lookup-Name

```
;; ins-defs.lisp

(include-book "sets")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Generic tree:
;;
```

```
;;   (data . children)

;; Accessors & predicates for a generic tree:

(defun data     (x) (car x))
(defun children (x) (cdr x))
(defun tipp     (x) (endp (children x)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Accessors


;; Accessors for the attribute, value, and records of the data of a node.

(defun att (x) x)
(defun val (x) (car x))
(defun rec (x) (cdr x))


;; Accessors for the attribute, value, and records of the root of a tree

(defun r-att (x) (att (data x)))
(defun r-val (x) (val (data x)))
(defun r-rec (x) (rec (data x)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun strip-root-atts (av-list)
  (if (endp av-list)
      nil
    (cons (r-att           (car av-list))
          (strip-root-atts (cdr av-list)))))

(defun strip-root-vals (va-list)
  (if (endp va-list)
      nil
    (cons (r-val           (car va-list))
          (strip-root-vals (cdr va-list)))))


;; The following functions check if a list of value-rooted trees (or a list of
```

```
;; attribute-rooted trees) has two trees with the same root label.

(defun no-dup-vals-p (va-list)
  (no-duplicatesp (strip-root-vals va-list)))

(defun no-dup-atts-p (av-list)
  (no-duplicatesp (strip-root-atts av-list)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Database nodes:
;;
;; - db-av is a list of the form (att . non-empty-db-va-list).
;;   I.e. it is a list of the form (att db-va . db-va-list)
;;
;; - db-va is a list of the form ((val . recs) . av-list)
;;
;; We put no restriction on att or val.  Recs, however, has to be a true-listp.

(mutual-recursion
 (defun db-avp (x)
   (and (consp x)
        (consp (children x))            ;; At least one child
        (no-dup-vals-p (children x))    ;; No duplicate value nodes
        (db-va-listp (children x))))
 (defun db-vap (x)
   (and (consp x)
        (consp (data x))
        (true-listp (rec x))
        (no-dup-atts-p (children x))    ;; No duplicate attribute nodes
        (db-av-listp (children x))))
 (defun db-av-listp (x)
   (if (endp x)
       (null x)
     (and (db-avp      (car x))
          (db-av-listp (cdr x)))))
 (defun db-va-listp (x)
   (if (endp x)
       (null x)
     (and (db-vap      (car x))
          (db-va-listp (cdr x)))))
)
```

```
;; Singly recursive version of db-avp, etc.

(defun dbp (flg x)
  (cond
   ((eq flg 'av)
    (and (consp x)
         (consp (children x))           ;; At least one child
         (no-dup-vals-p (children x))   ;; No duplicate value nodes
         (dbp 'va-list (children x))))
   ((eq flg 'va)
    (and (consp x)
         (consp (data x))
         (true-listp (rec x))
         (no-dup-atts-p (children x))   ;; No duplicate attribute nodes
         (dbp 'av-list (children x))))
   ((eq flg 'av-list)
    (if (endp x)
        (null x)
      (and (dbp 'av      (car x))
           (dbp 'av-list (cdr x)))))
   (t
    (if (endp x)
        (null x)
      (and (dbp 'va      (car x))
           (dbp 'va-list (cdr x)))))))

(defthm dbp-mutrec-to-singrec-lemma
  (equal (dbp flg x)
         (cond ((equal flg 'va) (db-vap x))
               ((equal flg 'av) (db-avp x))
               ((equal flg 'av-list) (db-av-listp x))
               (t (db-va-listp x)))))

(defthm dbp-mutrec-to-singrec
  (and (equal (db-vap  x)
              (dbp 'va x))
       (equal (db-avp  x)
              (dbp 'av x))
       (equal (db-va-listp  x)
```

```
              (dbp 'va-list x))
       (equal (db-av-listp  x)
              (dbp 'av-list x))))

(in-theory (disable dbp-mutrec-to-singrec-lemma))

(defthm dbp-true-listp
  (implies (dbp flg x)
           (true-listp x)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Query nodes:
;;
;; - q-av is a list of the form (att q-va . nil)
;; - q-va is a list of the form ((val . ignore-cdr) . q-av-list)

(mutual-recursion
 (defun q-avp (x)
   (and (consp x)
        (q-vap (car (children x)))
        (null  (cdr (children x)))))
 (defun q-vap (x)
   (and (consp x)
        (consp (data x))
        (q-av-listp (children x))))
 (defun q-av-listp (x)
   (if (endp x)
       (null x)
     (and (q-avp     (car x))
          (q-av-listp (cdr x)))))
)


;; Singly recursive version of q-avp, etc.

(defun qp (flg x)
  (cond
   ((eq flg 'av)
    (and (consp x)
         (qp 'va (car (children x)))
         (null  (cdr (children x)))))
```

```
   ((eq flg 'va)
    (and (consp x)
         (consp (data x))
         (qp 'av-list (children x))))
   (t
    (if (endp x)
        (null x)
      (and (qp 'av      (car x))
           (qp 'av-list (cdr x)))))))

(defthm qp-mutrec-to-singrec-lemma
  (equal (qp flg x)
         (cond ((equal flg 'va) (q-vap x))
               ((equal flg 'av) (q-avp x))
               (t (q-av-listp x)))))

(defthm qp-mutrec-to-singrec
  (and (equal (q-vap  x)
              (qp 'va x))
       (equal (q-avp  x)
              (qp 'av x))
       (equal (q-av-listp  x)
              (qp 'av-list x))))

(in-theory (disable qp-mutrec-to-singrec-lemma))

(defthm qp-true-listp
  (implies (qp flg x)
           (true-listp x)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun databasep (x) (db-vap x))
(defun queryp    (x) (q-vap  x))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Unions the records of a database

(mutual-recursion
 (defun all-recs (db-va)  ;; All records in a value-rooted tree
   (if (tipp db-va)
```

```
       (r-rec db-va)
     (unn (r-rec db-va)
          (all-recs-av-list (children db-va)))))
 (defun all-recs-av (db-av)
   (if (tipp db-av)
       nil
     (all-recs-va-list (children db-av))))
 (defun all-recs-va-list (db-va-list)
   (if (endp db-va-list)
       nil
     (unn (all-recs         (car db-va-list))
          (all-recs-va-list (cdr db-va-list)))))
 (defun all-recs-av-list (db-av-list)
   (if (endp db-av-list)
       nil
     (unn (all-recs-av      (car db-av-list))
          (all-recs-av-list (cdr db-av-list)))))
)


;; Singly-recursive version of all-recs

(defun all-recs-fn (flg db)
  (cond
   ((eq flg 'va)
    (if (tipp db)
        (r-rec db)
      (unn (r-rec db)
           (all-recs-fn 'av-list (children db)))))
   ((eq flg 'av)
    (if (tipp db)
        nil
      (all-recs-fn 'va-list (children db))))
   ((eq flg 'av-list)
    (if (endp db)
        nil
      (unn (all-recs-fn 'av      (car db))
           (all-recs-fn 'av-list (cdr db)))))
   (t
    (if (endp db)
        nil
```

```
      (unn (all-recs-fn 'va      (car db))
           (all-recs-fn 'va-list (cdr db)))))))

(defthm all-recs-to-all-recs-fn
  (and (equal (all-recs x)
              (all-recs-fn 'va x))
       (equal (all-recs-av x)
              (all-recs-fn 'av x))
       (equal (all-recs-av-list x)
              (all-recs-fn 'av-list x))
       (equal (all-recs-va-list x)
              (all-recs-fn 'va-list x))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun compute-s-prime (db-va-list)
  (if (endp db-va-list)
      nil
    (unn (all-recs        (car db-va-list))
         (compute-s-prime (cdr db-va-list)))))

(defthm compute-s-prime=all-recs-va-list
  (equal (compute-s-prime db-va-list)
         (all-recs-va-list db-va-list)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Node matching functions
;;
;; The following functions are similar to the native assoc function.  They
;; lookup a node in a list of nodes.  They are different from the native assoc
;; in that they use a particular field of an object for matching.

(defun att-assoc (q-av db-av-list)
  (cond
   ((endp db-av-list) nil)
   ((equal (r-att q-av)
           (r-att (car db-av-list)))
    (car db-av-list))
   (t
    (att-assoc q-av (cdr db-av-list)))))
```

```
(defun val-assoc (q-va db-va-list)
  (cond
   ((endp db-va-list) nil)
   ((equal (r-val q-va) (r-val (car db-va-list)))
    (car db-va-list))
   (t
    (val-assoc q-va (cdr db-va-list)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Model of Lookup-Name

;; The measure for lookup-name-for is acl2-count of q-av-list.

(mutual-recursion
 (defun lookup-name (db q)
   (declare (xargs :measure (1+ (acl2-count q))))
   (let ((db-children (children db))
         (q-children  (children q))
         (s           (all-recs db)))
     (lookup-name-for db-children q-children s)))
 (defun lookup-name-for (db-av-list q-av-list s)
   (let* ((qa  (car q-av-list))
          (qv  (car (children qa)))
          (dba (att-assoc qa db-av-list))
          (dbv (val-assoc qv (children dba))))
     (cond
      ((endp q-av-list) s)
      ((endp dba) nil)
      ((eq (r-val qv) '*)                       ;; Wildcard matching
       (compute-s-prime (children dba)))
      ((endp dbv) nil)
      ((endp (children qv))
       (lookup-name-for
        db-av-list (cdr q-av-list) (int s (all-recs dbv))))
      (t
       (lookup-name-for
        db-av-list (cdr q-av-list) (int s (lookup-name dbv qv)))))))
)

;;---------------------------------------------------------------------------
```

```
;; singly-recursive version of lookup-name

(defun m-lookup (flg x)
  (if (eq flg 'for)
      (acl2-count x)
    (1+ (acl2-count x))))

(defun lookup-name-fn (flg db q s)
  (declare (xargs :measure (m-lookup flg q)))
  (if (eq flg 'for)
      (let* ((qa  (car q))
             (qv  (car (children qa)))
             (dba (att-assoc qa db))
             (dbv (val-assoc qv (children dba))))
        (cond
         ((endp q) s)
         ((endp dba) nil)
         ((eq (r-val qv) '*)
          (compute-s-prime (children dba)))
         ((endp dbv) nil)
         ((endp (children qv))
          (lookup-name-fn 'for
           db (cdr q) (int s (all-recs dbv))))
         (t
          (lookup-name-fn 'for
           db (cdr q) (int s (lookup-name-fn 'lkp dbv qv s))))))
    (let ((db-av-list (children db))
          (q-av-list  (children q))
          (s          (all-recs db)))
      (lookup-name-fn 'for db-av-list q-av-list s))))

(defthm lookup-name-fn-to-lookup-name
  (equal (lookup-name-fn flg db q s)
         (if (eq flg 'for)
             (lookup-name-for db q s)
           (lookup-name db q))))

(in-theory (disable lookup-name-fn-to-lookup-name))

(defthm lookup-name-to-lookup-name-fn
   (and (equal (lookup-name db q)
```

```
              (lookup-name-fn 'lkp db q s))
        (equal (lookup-name-for db-av-list q-av-list s)
               (lookup-name-fn 'for db-av-list q-av-list s)))
   :hints (("Goal"
            :in-theory (e/d (lookup-name-fn-to-lookup-name)
                            (lookup-name-fn lookup-name lookup-name-for)))))
```

A.2.2 Specification of Lookup-Name

```
;; ins-spec.lisp

(ld "ins-defs.lisp")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Notion of correctness

(mutual-recursion
 (defun flatten1 (x)
   (if (tipp x)
       (list (list (data x)))
     (fold-cons (data x) (flatten1-list (children x)))))
 (defun flatten1-list (xs)
   (if (endp xs)
       nil
     (append (flatten1      (car xs))
             (flatten1-list (cdr xs)))))
)

(defun flatten1-fn (flg x)
  (if (eq flg 'flatten)
      (if (tipp x)
          (list (list (data x)))
        (fold-cons (data x) (flatten1-fn 'flatten-list (children x))))
    (if (endp x)
        nil
      (append (flatten1-fn 'flatten      (car x))
              (flatten1-fn 'flatten-list (cdr x))))))

(defthm true-listp-flatten1-fn
  (true-listp (flatten1-fn flg x)))
```

```
(defthm flatten=flatten1
  (equal (flatten-fn flg x stk)
         (fold-append (rev stk)
                      (flatten1-fn flg x))))

(in-theory (disable flatten=flatten1))

(defthm flatten-to-flatten-fn-lemma
  (equal (flatten-fn flg x stack)
         (if (equal flg 'flatten)
             (flatten x stack)
           (flatten-list x stack)))
  :rule-classes nil)

(defthm flatten-to-flatten-fn
  (and (equal (flatten x stack) (flatten-fn 'flatten x stack))
       (equal (flatten-list x stack) (flatten-fn 'flatten-list x stack)))
  :hints (("Goal" :use ((:instance flatten-to-flatten-fn-lemma (flg 'flatten))
                        (:instance flatten-to-flatten-fn-lemma (flg 'flatten-list))))))

;;-----------------------------------------------------------------------------

;; Thm: flatten always returns a non-empty list
;;
;; and
;;
;; if (rev stack) is a true-listp, flatten always returns true-listp.

(defthm Example1
  (let ((db0 '((root)))
        (db1 '((root r0)))
        (db2 '((root
                (serv ((printer r0 r1))
                      ((scanner r2)))
                (bldg ((main r1))
                      ((aces r0 r2))))))
        (q0 '((root)))
        (q1 '((root)
              (serv ((printer)))))
        (q2 '((root)
```

```
            (serv ((printer)))
            (bldg ((main)))))
       (q3 '((root)
            (serv ((printer))
                  ((scanner))))))
    (and
     (databasep db0)
     (databasep db1)
     (databasep db2)
     (queryp q0)
     (queryp q1)
     (not (queryp q3))
     (set-equal (lookup-name db2 q2)
                '(r1))
     (equal (flatten1 db2)
            '(((root) serv (printer r0 r1))
              ((root) serv (scanner r2))
              ((root) bldg (main r1))
              ((root) bldg (aces r0 r2))))))
  :rule-classes nil)


;; We define a well-formed value-attribute path to be a non-empty true-listp of
;; node data in which
;;
;;  1) every even element is a well-formed value node data, i.e. it is a cons
;;     with its cdr being a well-formed record set (i.e. a true-listp)
;;
;;  2) we put no restriction on the odd nodes because they correspond to
;;     attributes
;;
;;  3) The length of the path is odd because there must be a value node data
;;     followed by an av-path (i.e. zero or more pairs of attribute-values,
;;     i.e. an attribute followed by a va-path).
;;
;;  Remark: the definition of va-pathp and av-pathp are a lot like those of
;;  db-vap (or q-vap) and db-avp (or q-avp), respectively.
;;
;;  Remark: the length of a va-path is always odd, and the length of an av-path
;;  is always even.
```

```
(mutual-recursion
 (defun va-pathp (p)
   (and (consp (car p))
        (true-listp (rec (car p)))
        (av-pathp (cdr p))))
 (defun av-pathp (p)
   (if (endp p)
       (null p)
     (va-pathp (cdr p))))
)


;; Singly-recursive version of va-pathp and av-pathp

(defun pathp-fn (flg p)
  (if (eq flg 'va)
      (and (consp (car p))
           (true-listp (rec (car p)))
           (pathp-fn 'av (cdr p)))
    (if (endp p)
        (null p)
      (pathp-fn 'va (cdr p)))))

(defthm pathp-to-pathp-fn
  (and (equal (va-pathp p)
              (pathp-fn 'va p))
       (equal (av-pathp p)
              (pathp-fn 'av p))))

;; Some potentially useful theorems are below:
;;
;; 1) Any path is a true-listp
;;
;; 2) Reversing an av-path and appending it onto the left of a va-path results
;;    in a va-path.
;;
;; 3)
;;
;; (thm (implies (or (db-vap x)
;;                   (q-vap x))
;;               (va-path-listp (flatten1 x))))
```

```
(defthm true-listp-pathp-fn
  (implies (pathp-fn flg x)
           (true-listp x)))

;; In-path determines if r can be found in the record set of a value
;; node along the path path.
;;
;; If flg is 'va, then we're looking at a path that starts with a value.

(defun in-path (flg r path)
  (cond
   ((endp path) nil)
   ((eq flg 'va)
    (or (member r (rec (car path)))
        (in-path 'av r (cdr path))))
   (t (in-path 'va r (cdr path)))))     ;; A record can never be in an
                                        ;; attribute, so search the
                                        ;; rest of the path.


;; Determines if record r is reachable in the database path dbp through the
;; query path qp.

(defun reachable (flg r p1 p2)
  (if (endp p1)
      nil
    (if (eq flg 'va)
        (cond
         ((eq '* (val (car p1)))
          (in-path 'va r p2))
         ((equal (val (car p1))
                 (val (car p2)))
          (if (endp (cddr p1))
              (in-path 'va r p2)
            (reachable 'av r (cdr p1) (cdr p2))))
         (t nil))
      (cond
       ((equal (att (car p1))
               (att (car p2)))
        (reachable 'va r (cdr p1) (cdr p2)))
```

```
       (t nil)))))


;; Determines if record r is reachable in at least one database path in
;; dbp-list from the query path qp.

(defun member-reachable (flg r qp dbp-list)
  (if (endp dbp-list)
      nil
    (or (reachable  flg r qp (car dbp-list))
        (member-reachable flg r qp (cdr dbp-list)))))


;; Determines if every query path in qp-list leads to record r through at least
;; one of the database paths in dbp-list.

(defun subset-reachable (flg r qp-list dbp-list)
  (if (endp qp-list)
      t
    (and (member-reachable flg r (car qp-list) dbp-list)
         (subset-reachable flg r (cdr qp-list) dbp-list))))


;; Satp checks if r satisfies query q in database db.

(defun satp (r q db)
  (subset-reachable 'va r (flatten1 q) (flatten1 db)))

;; (defun satp (r q db)
;;   (subset-reachable r (flatten1 q) (flatten1 db)))

(defconst *q2* '((root)
                 (serv ((printer)))
                 (bldg ((main)))))

(defconst *q1* '((root)
                 (serv ((printer)))))

(defconst *q0* '((root)))

(defconst *db2* '((root)
```

```
                  (serv ((printer r0 r1))
                        ((scanner r2)))
                  (bldg ((main r1))
                        ((aces r0 r2)))))

(defthm Example2
  (let ((db0 '((root)))
        (db1 '((root r0)))
        (db2 '((root)
               (serv ((printer r0 r1))
                     ((scanner r2)))
               (bldg ((main r1))
                     ((aces r0 r2)))))
        (q0 '((root)))
        (q1 '((root)
              (serv ((printer)))))
        (q2 '((root)
              (serv ((printer)))
              (bldg ((main)))))
        (q3 '((root)
              (serv ((printer))
                    ((scanner))))))
    (and (flatten1 db0)
         (flatten1 db1)
         (flatten1 db2)
         (flatten1 q0 )
         (flatten1 q1 )
         (flatten1 q2 )
         (flatten1 q3 )
         (satp 'r0 q1 db2)
         (not (satp 'r0 q2 db2))))
  :rule-classes nil)

;;-----------------------------------------------------------------------------

(defthm consp-flatten1-fn
  (equal (consp (flatten1-fn flg x))
         (if (equal flg 'flatten)
             t
           (consp x))))
```

```
;; This is suggested by failed proof of member-reachable-cons-fold-cons
(defthm not-in-path-not-reachable
  (implies (not (in-path flg r p2))
           (not (reachable flg r p1 p2))))

(defthm member-reachable-nil-path
  (not (member-reachable flg r nil p)))

(in-theory (disable member-reachable-nil-path))


(i-am-here)

(defthm lookup-name-sound-general-case
  (implies (and (dbp (if (equal flg 'for) 'av-list 'va) db)
                (qp  (if (equal flg 'for) 'av-list 'va) q)
                (member r (lookup-name-fn flg db q s))
                (if (equal flg 'for)
                    t
                  (equal (caar q) (caar db))))
           (subset-reachable (if (equal flg 'for) 'av 'va)
                             r
                             (flatten1-fn (if (equal flg 'for)
                                              'flatten-list
                                            'flatten)
                                          q)
                             (flatten1-fn (if (equal flg 'for)
                                              'flatten-list
                                            'flatten)
                                          db)))
  :otf-flg t
  :hints (("Goal"
           :induct (lookup-name-fn flg db q s)
           :do-not '(eliminate-destructors generalize fertilize)
           :do-not-induct t))
  :rule-classes nil)


(skip-proofs
(defthm reachable-fold-append-flatten1-fn
  (iff (subset-reachable r
```

```
                  (fold-append  qp  q-path-list)
                  (fold-append dbp db-path-list))
       (if (eq 'failure (chop qp dbp))
           nil
         (subset-reachable r
                     q-path-list
                     (fold-append (chop qp dbp) db-path-list)))))
)

(skip-proofs
(defthm lookup-name-sound
  (implies (and (databasep db)
                (queryp q)
                (member r (lookup-name db q)))
           (satp r q db))
  :hints (("Goal" :in-theory (enable flatten=flatten1)))
  :rule-classes nil)
)

(skip-proofs
(defthm lookup-name-complete
  (implies (and (databasep db)
                (queryp q)
                (satp r q db))
           (member r (lookup-name db q)))
  :rule-classes nil)
)

(defthm lookup-name-correct
   (implies (and (databasep db)
                 (queryp q))
            (iff (member r (lookup-name db q))
                 (satp r q db)))
   :hints (("Goal"
            :use ((:instance lookup-name-sound)
                  (:instance lookup-name-complete))
            :in-theory (disable member lookup-name satp
                                lookup-name-to-lookup-name-fn)))
   :rule-classes nil)
```

A.3 Tree-Containment

A.3.1 Model of Tree-Containment

```
(in-package "ACL2")

(include-book "list-helpers")

;; Accessors for an integer tree

(defun root (x)
  (if (atom x) x (car x)))

(defun chdn (x)
  (cdr x))

(defun tipp (x)
  (atom x))

(defun strip-roots (atree-list)
  (if (endp atree-list)
      nil
    (cons (root        (car atree-list))
          (strip-roots (cdr atree-list)))))


;; (root . children); root

;; Recognizer for a tree of atoms

(mutual-recursion
 (defun atreep (x)
   (if (atom x)
       t
     (and (atom (car x))
          (consp (cdr x))  ;; There must be at least one child
          (atree-listp (cdr x)))))
 (defun atree-listp (xs)
   (if (endp xs)
       (null xs)
```

```
     (and (atreep (car xs))
          (not (member (root (car xs)) (strip-roots (cdr xs))))
          (atree-listp (cdr xs)))))
)


;; Determines if two trees have the same root

(defun requal (t1 t2)
  (equal (root t1)
         (root t2)))

;; Determines if t1 is contained in t2 from the root

(mutual-recursion
 (defun contdp (t1 t2)
   (declare (xargs :measure (+ (acl2-count t1)
                               (acl2-count t2))))
   (cond
    ((not (requal t1 t2)) nil)
    ((tipp t1) t)
    ((tipp t2) nil)
    (t (contd-listp (chdn t1) (chdn t2)))))
 (defun contd-listp (t1-list t2-list)
   (declare (xargs :measure (+ (acl2-count t1-list)
                               (acl2-count t2-list))))
   (if (endp t1-list)
       t
     (and (contdp-aux  (car t1-list) t2-list)
          (contd-listp (cdr t1-list) t2-list))))
 (defun contdp-aux (t1 t2-list)
   (declare (xargs :measure (+ (acl2-count t1)
                               (acl2-count t2-list))))
   (if (endp t2-list)
       nil
     (or (contdp     t1 (car t2-list))
         (contdp-aux t1 (cdr t2-list)))))
)


;; Graphical representations of t1, t2, t3, t4, and t5, respectively:
```

```
;; t1:
;;
;; 1

;; t2:
;;
;;   1
;;  / \
;; 2   3

;; t3:
;;
;;   1
;;  / \
;; 2   3
;;     |
;;     6

;; t4:
;;
;;   1
;;  / \
;; 2   4

;; t5:
;;
;;       1
;;    /  |  \
;;   2   3   4
;;  /\  /\   |
;; 5 8 6 7   9

(defthm Example1
  (let ((t1 1)
        (t2 '(1 2 3))
        (t3 '(1 2 (3 6)))
        (t4 '(1 2 4))
        (t5 '(1 (2 5 8) (3 6 7) (4 9))))
    (and
     (contdp t1 t1)
```

```
    (contdp t1 t2)
    (contdp t1 t3)
    (contdp t1 t4)
    (contdp t1 t5)
    (contdp t2 t2)
    (contdp t2 t3)
    (contdp t2 t5)
    (contdp t3 t5)
    (not (contdp t4 t3))
    (contdp t4 t5)
    ))
  :rule-classes nil)
```

A.3.2 Specification of Tree-Containment

```
(in-package "ACL2")

(include-book "contdp-defs")

;; Notion of correctness

(mutual-recursion
 (defun flatten (x)
   (if (tipp x)
       (list (list x))
     (fold-cons (root x) (flatten-list (chdn x)))))
 (defun flatten-list (xs)
   (if (endp xs)
       nil
     (append (flatten (car xs))
             (flatten-list (cdr xs)))))
)

;; t6:
;;
;;       1
;;    /  |  \
;;   2   3   4
;;  /\  /\
;; 5 8 6 7
```

```
;;       |
;;       9

(defthm Example2
  (let ((t6 '(1 (2 5 8) (3 6 (7 9)) 4)))
    (equal
     (flatten t6)
     '((1 2 5)
       (1 2 8)
       (1 3 6)
       (1 3 7 9)
       (1 4))))
  :rule-classes nil)

;; Determines if p1 is a prefix of p2.  This is analogous to "reachable" in
;; INS.

(defun prefixp (p1 p2)
  (cond
   ((endp p1) t)
   ((endp p2) nil)
   (t (and (equal (car p1) (car p2))
           (prefixp (cdr p1) (cdr p2))))))


;; The following determines if there exists a path in p-list (a list of paths) such
;; that p-list is a suffix of it.
;;
;; This is analogous to reachable1 in INS.

(defun is-a-prefix-in (p p-list)
  (if (endp p-list)
      nil
    (or (prefixp p (car p-list))
        (is-a-prefix-in p (cdr p-list)))))

;; The following function determines if "every" path in p-list1 is a prefix of a
;; path in p-list2.
;;
;; This is analogous to reachable2 in INS.
```

```
(defun are-prefixes-in (p-list1 p-list2)
  (if (endp p-list1)
      t
    (and (is-a-prefix-in  (car p-list1) p-list2)
         (are-prefixes-in (cdr p-list1) p-list2))))

(defun satp (x y)
  (are-prefixes-in (flatten x) (flatten y)))
```

A.3.3 Proof of Tree-Containment

```
(in-package "ACL2")

(include-book "contdp-spec")

;; First convert all mutually recursive functions to singly recursive form.

(defun atreep-fn (flg x)
  (if (eq flg 'tree)
      (if (atom x)
          t
        (and (atom (car x))
             (consp (cdr x))  ;; There must be at least one child
             (atreep-fn 'list (cdr x))))
    (if (endp x)
        (null x)
      (and (atreep-fn 'tree (car x))
           (not (member (root (car x)) (strip-roots (cdr x))))
           (atreep-fn 'list (cdr x))))))

(defthm atreep-to-atreep-fn-lemma
  (equal (atreep-fn flg x)
         (if (eq flg 'tree)
             (atreep x)
           (atree-listp x))))

(defthm atreep-to-atreep-fn
  (and (equal (atreep x)
              (atreep-fn 'tree x))
       (equal (atree-listp x)
```

```
              (atreep-fn 'list x))))

(in-theory (disable atreep-to-atreep-fn-lemma))


(defun contdp-fn (flg t1 t2)
  (declare (xargs :measure (+ (acl2-count t1)
                              (acl2-count t2))))
  (cond
   ((eq flg 'tree)
    (cond
     ((not (requal t1 t2)) nil)
     ((tipp t1) t)
     ((tipp t2) nil)
     (t (contdp-fn 'list (chdn t1) (chdn t2)))))
   ((eq flg 'list)
    (if (endp t1)
        t
      (and (contdp-fn 'aux  (car t1) t2)
           (contdp-fn 'list (cdr t1) t2))))
   (t
    (if (endp t2)
        nil
      (or (contdp-fn 'tree t1 (car t2))
          (contdp-fn 'aux  t1 (cdr t2)))))))

(defthm contdp-to-contdp-fn-lemma
  (equal (contdp-fn flg t1 t2)
         (cond
          ((eq flg 'tree) (contdp      t1 t2))
          ((eq flg 'list) (contd-listp t1 t2))
          (t              (contdp-aux  t1 t2)))))

(defthm contdp-to-contdp-fn
  (and
   (equal (contdp      t1 t2) (contdp-fn 'tree t1 t2))
   (equal (contd-listp t1 t2) (contdp-fn 'list t1 t2))
   (equal (contdp-aux  t1 t2) (contdp-fn 'aux  t1 t2))))

(in-theory (disable contdp-to-contdp-fn-lemma))
```

```
(defun flatten-fn (flg x)
  (if (eq flg 'tree)
      (if (tipp x)
          (list (list x))
        (fold-cons (root x) (flatten-fn 'list (chdn x))))
    (if (endp x)
        nil
      (append (flatten-fn 'tree (car x))
              (flatten-fn 'list (cdr x))))))

(defthm flatten-to-flatten-fn-lemma
  (equal (flatten-fn flg x)
         (if (eq flg 'tree)
             (flatten x)
           (flatten-list x))))

(defthm flatten-to-flatten-fn
  (and (equal (flatten x)
              (flatten-fn 'tree x))
       (equal (flatten-list x)
              (flatten-fn 'list x))))

(in-theory (disable flatten-to-flatten-fn-lemma))

; ------------------------------------------------------------------------------

(defthm true-listp-flatten-fn
  (true-listp (flatten-fn flg x)))

(defthm consp-flatten-fn
  (implies (atreep-fn flg x)
           (equal (consp (flatten-fn flg x))
                  (if (eq flg 'tree)
                      t
                    (consp x)))))

(defthm is-a-prefix-in-append
  (equal (is-a-prefix-in p (append p-list1 p-list2))
         (or (is-a-prefix-in p p-list1)
             (is-a-prefix-in p p-list2))))
```

```
(defthm are-prefixes-append
  (equal (are-prefixes-in (append list1 list2) list3)
         (and (are-prefixes-in list1 list3)
              (are-prefixes-in list2 list3))))

(defthm are-prefixes-append1
 (implies (are-prefixes-in list1 list2)
          (are-prefixes-in list1 (append list2 list3))))

(defthm are-prefixes-in-append-comm
  (equal (are-prefixes-in list1 (append list2 list3))
         (are-prefixes-in list1 (append list3 list2))))

(defthm not-equal-cons-not-is-a-prefix-in
 (implies (not (equal x1 x2))
          (not (is-a-prefix-in (cons x1 p)
                               (fold-cons x2 p-list)))))

(defthm is-a-prefix-in-cons
  (equal (is-a-prefix-in (cons x1 p)
                         (fold-cons x2 p-list))
         (and (equal x1 x2)
              (is-a-prefix-in p p-list))))

;;-----------------------------------------------------------------------------

(defthm are-prefixes-fold-cons
  (equal (are-prefixes-in (fold-cons x1 p-list1)
                          (fold-cons x2 p-list2))
         (if (equal x1 x2)
             (are-prefixes-in p-list1 p-list2)
           (endp p-list1))))

(defthm are-prefixes-in-cons
  (implies (are-prefixes-in x y)
           (are-prefixes-in x (cons e y))))


;; The following two functions are used in the statement of the soundness and
;; completeness theorems for determining the type of the input trees (i.e. a
;; single tree vs. a list of trees).
```

```
(defun flg1 (flg)
  (cond
   ((eq flg 'tree) 'tree)
   ((eq flg 'list) 'list)
   (t              'tree) ;; This case corresponds to 'aux
   ))

(defun flg2 (flg)
  (cond
   ((eq flg 'tree) 'tree)
   ((eq flg 'list) 'list)
   (t              'list) ;; This case corresponds to 'aux
   ))

;; Soundness

(defthm contdp-sound-general-case
  (implies (and (contdp-fn flg x y)
                (equal flg1 (flg1 flg))
                (equal flg2 (flg2 flg))
                (atreep-fn flg1 x)
                (atreep-fn flg2 y))
           (are-prefixes-in
            (flatten-fn flg1 x)
            (flatten-fn flg2 y))))

;;-----------------------------------------------------------------------------

(defthm prefixp-reflexive
  (prefixp x x))

(defthm prefixp-transitive
  (implies (and (prefixp x y)
                (prefixp y z))
           (prefixp x z)))

(defthm are-prefixes-in-reflexive
  (are-prefixes-in x x))

(defthm not-is-prefix-in
```

```
  (implies (and (not (is-a-prefix-in x z))
                (is-a-prefix-in y z))
           (not (prefixp x y))))

(defthm are-prefixes-in-transitive
  (implies (and (are-prefixes-in x y)
                (are-prefixes-in y z))
           (are-prefixes-in x z)))

(defthm or--are-prefixes-in--append
 (implies (or (are-prefixes-in x y)
              (are-prefixes-in x z))
          (are-prefixes-in x (append y z))))

;;-----------------------------------------------------------------------------

(defthm is-a-prefix-in-not-member
 (implies (and (consp p)
               (not (member (car p) (strip-cars y))))
          (not (is-a-prefix-in p y))))


;; Recognizes a list of conses (equivalent to alistp, except that alistp
;; requires the list to be a true-listp as well)

(defun consp-listp (list)
  (if (endp list)
      (null list)
    (and (consp       (car list))
         (consp-listp (cdr list)))))

(defthm true-listp-consp-listp
  (implies (consp-listp x)
           (true-listp x)))

(defthm consp-listp-append
  (implies (and (consp-listp x)
                (consp-listp y))
           (consp-listp (append x y))))

(defthm consp-listp-fold-cons
```

```
  (consp-listp (fold-cons x list)))

(defthm consp-listp-flatten-fn
  (consp-listp (flatten-fn flg x)))

(defthm non-empty-consp-listp-are-prefixes-in
 (implies (and (consp p-list1)
               (consp-listp p-list1)
               (not (subsetp (strip-cars p-list1)
                             (strip-cars p-list2))))
          (not (are-prefixes-in p-list1 p-list2))))


;; Very important theorem

(defthm are-prefixes-in-append
  (implies
   (not (member b (strip-cars list)))
   (equal (are-prefixes-in (fold-cons a x)
                           (append (fold-cons b y)
                                   list))
          (or (are-prefixes-in (fold-cons a x)
                               (fold-cons b y))
              (are-prefixes-in (fold-cons a x)
                               list)))))

(defthm are-prefixes-in-append-cor
  (implies
   (not (member b (strip-cars list)))
   (equal (are-prefixes-in (fold-cons a x)
                           (append list
                                   (fold-cons b y)))
          (or (are-prefixes-in (fold-cons a x)
                               (fold-cons b y))
              (are-prefixes-in (fold-cons a x)
                               list)))))

(defthm member-strip-cars-fold-cons
 (iff (member a (strip-cars (fold-cons b x)))
      (if (consp x)
          (equal a b)
```

```
        nil)))

(defthm contdp-complete-general-case-subgoal-1-3
  (implies (and (consp x2)
                (atreep-fn 'list x2))
           (not (are-prefixes-in (fold-cons x1 (flatten-fn 'list x2))
                                 (list (list x1))))))

(defthm strip-cars-append
 (equal (strip-cars (append x y))
        (append (strip-cars x)
                (strip-cars y))))

(defthm member-append
 (iff (member e (append x y))
      (or (member e x)
          (member e y))))

(defthm strip-roots-strip-cars
  (implies
   (atreep-fn 'list tree-list)
   (iff (member x (strip-roots tree-list))
        (member x (strip-cars (flatten-fn 'list tree-list))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defthm are-prefixes-in-append-cor2
 (implies
  (not (member e (strip-cars y)))
  (equal
   (are-prefixes-in (flatten-fn 'tree x)
                    (append y (fold-cons e z)))
   (or (are-prefixes-in (flatten-fn 'tree x)
                        (fold-cons e z))
       (are-prefixes-in (flatten-fn 'tree x)
                        y))))
 :hints (("Goal"
          :expand (flatten-fn 'tree x))))

(defthm are-prefixes-in-append-cor3
 (implies
```

```
  (not (member e (strip-cars y)))
  (equal
   (are-prefixes-in (flatten-fn 'tree x)
                    (append (fold-cons e z) y))
   (or (are-prefixes-in (flatten-fn 'tree x)
                        (fold-cons e z))
       (are-prefixes-in (flatten-fn 'tree x)
                        y))))
 :hints (("Goal"
          :expand (flatten-fn 'tree x))))

(defthm are-prefixes-in-append-cor4
  (implies
   (not (member (car y) (strip-cars (flatten-fn 'list (cdr y)))))
   (equal
    (are-prefixes-in (flatten-fn 'tree x)
                     (cons (list (car y))
                           (flatten-fn 'list (cdr y))))
    (or (are-prefixes-in (flatten-fn 'tree x)
                         (fold-cons (car y) '(nil)))
        (are-prefixes-in (flatten-fn 'tree x)
                         (flatten-fn 'list (cdr y))))))
  :hints (("Goal"
           :in-theory (disable are-prefixes-in-append-cor3)
           :use (:instance are-prefixes-in-append-cor3
                           (e (car y))
                           (z '(nil))
                           (y (flatten-fn 'list (cdr y)))))))

(defthm contdp-complete-general-case
  (implies (and
            (atreep-fn (flg1 flg) x)
            (atreep-fn (flg2 flg) y)
            (are-prefixes-in
             (flatten-fn (flg1 flg) x)
             (flatten-fn (flg2 flg) y)))
           (contdp-fn flg x y)))

(defthm contdp-correct
  (implies (and (atreep x)
                (atreep y))
```

```
(iff (contdp x y)
     (satp x y))))
```

Appendix B

A Detailed Explanation of an Inductive Proof

B.1 Distributivity of Fold-cons over Append

```
ACL2 !>
(defthm distrib-of-fold-cons-over-append
 (equal (fold-cons a (append list1 list2))
        (append (fold-cons a list1)
                (fold-cons a list2))))

Name the formula above *1.

Perhaps we can prove *1 by induction.  Three induction schemes are
suggested by this conjecture.  These merge into two derived induction
schemes.  However, one of these is flawed and so we are left with one
viable candidate.

We will induct according to a scheme suggested by (FOLD-CONS A LIST1),
but modified to accommodate (APPEND LIST1 LIST2).  These suggestions
were produced using the :induction rules BINARY-APPEND and FOLD-CONS.
If we let  (:P A LIST1 LIST2) denote *1 above then the induction scheme
we'll use is
```

```
(AND (IMPLIES (AND (NOT (ENDP LIST1))
                   (:P A (CDR LIST1) LIST2))
              (:P A LIST1 LIST2))
     (IMPLIES (ENDP LIST1)
              (:P A LIST1 LIST2))).
```

This induction is justified by the same argument used to admit FOLD-CONS, namely, the measure (ACL2-COUNT LIST1) is decreasing according to the relation O< (which is known to be well-founded on the domain recognized by O-P). When applied to the goal at hand the above induction scheme produces the following two nontautological subgoals.

```
Subgoal *1/2
(IMPLIES (AND (NOT (ENDP LIST1))
              (EQUAL (FOLD-CONS A (APPEND (CDR LIST1) LIST2))
                     (APPEND (FOLD-CONS A (CDR LIST1))
                             (FOLD-CONS A LIST2))))
         (EQUAL (FOLD-CONS A (APPEND LIST1 LIST2))
                (APPEND (FOLD-CONS A LIST1)
                        (FOLD-CONS A LIST2)))).
```

By the simple :definition ENDP we reduce the conjecture to

```
Subgoal *1/2'
(IMPLIES (AND (CONSP LIST1)
              (EQUAL (FOLD-CONS A (APPEND (CDR LIST1) LIST2))
                     (APPEND (FOLD-CONS A (CDR LIST1))
                             (FOLD-CONS A LIST2))))
         (EQUAL (FOLD-CONS A (APPEND LIST1 LIST2))
                (APPEND (FOLD-CONS A LIST1)
                        (FOLD-CONS A LIST2)))).
```

But simplification reduces this to T, using the :definitions BINARY-APPEND and FOLD-CONS, primitive type reasoning, the :rewrite rules CAR-CONS and CDR-CONS and the :type-prescription rule FOLD-CONS.

```
Subgoal *1/1
(IMPLIES (ENDP LIST1)
         (EQUAL (FOLD-CONS A (APPEND LIST1 LIST2))
                (APPEND (FOLD-CONS A LIST1)
                        (FOLD-CONS A LIST2)))).
```

```
By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/1'
(IMPLIES (NOT (CONSP LIST1))
         (EQUAL (FOLD-CONS A (APPEND LIST1 LIST2))
                (APPEND (FOLD-CONS A LIST1)
                        (FOLD-CONS A LIST2)))).

But simplification reduces this to T, using the :definitions BINARY-
APPEND and FOLD-CONS, the :executable-counterpart of CONSP and primitive
type reasoning.

That completes the proof of *1.

Q.E.D.

Summary
Form:  ( DEFTHM DISTRIB-OF-FOLD-CONS-OVER-APPEND ...)
Rules: ((:DEFINITION BINARY-APPEND)
        (:DEFINITION ENDP)
        (:DEFINITION FOLD-CONS)
        (:DEFINITION NOT)
        (:EXECUTABLE-COUNTERPART CONSP)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:INDUCTION BINARY-APPEND)
        (:INDUCTION FOLD-CONS)
        (:REWRITE CAR-CONS)
        (:REWRITE CDR-CONS)
        (:TYPE-PRESCRIPTION FOLD-CONS))
Warnings:  None
Time:  0.01 seconds (prove: 0.01, print: 0.00, other: 0.00)
 DISTRIB-OF-FOLD-CONS-OVER-APPEND
ACL2 !>
```