An ACL2 Library for Bags (Multisets)

Eric Smith*, Serita Nelesen*, David Greve, Matthew Wilding, and Raymond Richards

Rockwell Collins Advanced Technology Center
Cedar Rapids, IA 52498 US

***Eric and Serita are students at Stanford University and the University of Texas at Austin, respectively.**

Background

- The AAMP7 microcode has instructions that access memory.
- Rockwell Collins has a library, GACC, for reasoning about programs which use those instructions.
- GACC uses bags to represent collections of addresses.

Outline

- Why bags?
- Functions and predicates about bags
- Basic bag rules - can be too expensive!
- :Meta rules to the rescue!

Why bags?

- We often need to show that two memory operations don't interfere (i.e., that they affect different addresses).
- Two main ways to show that addresses `a` and `b` differ:

  (1) `a` and `b` belong to collections which are disjoint from each other.

  (2) `a` and `b` are separately included in a collection that contains no duplicates.

Why bags? (continued)

- We need to reason about collections of addresses.
- We must keep track of duplicates.
- The order of elements in our collections isn't meaningful.
- Multisets are collections in which elements can appear multiple times but in which the order of elements doesn't matter. Perfect!
- Multisets are also called "bags."

Implementation of Bags

- We currently implement bags as lists.
- Ex: `(4 1 1 5 1)
- Ex: nil

- We may change this representation later.

Operations On Bags

- (bag-insert a x) : Insert element a into bag x.
- (remove-1 a x) : Remove one occurrence of element a from bag x.
- (remove-all a x) : Remove all occurrences of element a from bag x.
- (bag-sum x y) : Combine the bags x and y.
- (bag-difference x y) : Remove the elements in bag y from bag x.

Predicates on Bags

- (memberp a x) : Does a appear in bag x?
- (subbagp x y) : Does each element appear in bag y at least as many times as it appears in bag x?
- (disjoint x y) : Do the bags x and y have no elements in common?
- (unique x) : Does no element appear in x more than once?
- (bagp x) : Is x is a bag?
- (empty-bagp x) : Is x is an empty bag?

More Operations on Bags

- (count a x) : Return the multiplicity of a in x.
- (perm x y) : Equivalence relation to test whether x and y represent the same bag (i.e., whether they agree on the count for each element). Allows congruence reasoning.

Rules About Bags

The bags library has two kinds of rules:

1. Basic rules for simplifying terms in the usual ACL2 style.
2. Fancy rules (mostly :meta rules) for cases in which the basic rules are too expensive.

Some Basic Bag Rules

```
(defthm unique-of-append
  (equal (unique (append x y))
         (and (unique x)
              (unique y)
              (disjoint x y))))

(defthm disjoint-of-append-one
  (equal (disjoint (append x y) z)
         (and (disjoint x z)
              (disjoint y z))))

(defthm disjoint-of-append-two
  (equal (disjoint x (append y z))
         (and (disjoint x y)
              (disjoint x z))))
```

Basic Rules Can Be Expensive!

```
(unique (append a b c d e f))      ➔       (and (unique a)
                                                (unique b)
                                                (unique c)
                                                (unique d)
                                                (unique e)
                                                (unique f)
                                                (disjoint e f)
                                                (disjoint d e)
                                                (disjoint d f)
                                                (disjoint c d)
                                                (disjoint c e)
                                                (disjoint c f)
                                                (disjoint b c)
                                                (disjoint b d)
                                                (disjoint b e)
                                                (disjoint b f)
                                                (disjoint a b)
                                                (disjoint a c)
                                                (disjoint a d)
                                                (disjoint a e)
                                                (disjoint a f))
```

This is a quadratic blowup! (We get one disjoint claim for each pair of arguments to append.)

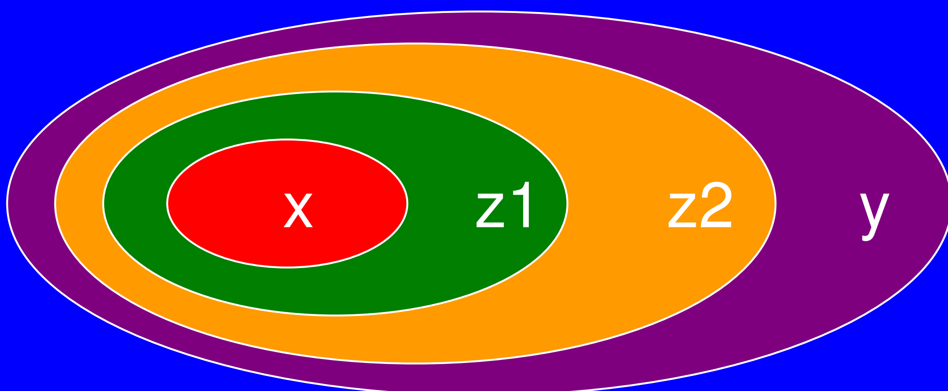
But sometimes we append dozens of things! Yikes!

:Meta Rules to the Rescue !

- We disable potentially expensive basic rules and use :meta rules for the cases we care about.
- We care most about establishing certain predicates (disjoint, unique, etc.).
- Our :meta rules search through the known facts (i.e., the type-alist) to try to find a line of reasoning showing that the predicate of interest is true.

Example: Subbag Chain

- Intuition: To show (subbagp x y), we use known facts to construct a "subbag chain" from x to y.
- We might know (subbagp x z1), (subbagp z1 z2), and (subbagp z2 y).
- We can conclude (subbagp x y).
- Think: x $\subseteq$ z1 $\subseteq$ z2 $\subseteq$ y.



"Syntactic" Subbags

- Sometimes we can tell just by looking at two terms that one is a subbag of the other.
- Ex: x is always subbag of (append x z).

- If we discover (subbagp (append x z) y), we can conclude (subbagp x y).
- Think: x $\subseteq$ (append x z) $\subseteq$ y.

The Rule for Subbagp

Ways to show (subbagp x y):

1. Notice that (syntax-subbagp x y).

or:

2. Discover (subbagp *blah1 blah2*), where:
   (syntax-subbagp x *blah1*), and then show: (subbagp *blah2* y).

Think: x $\subseteq$ *blah1* $\subseteq$ *blah2* $\subseteq$ y

Concrete Example

```
(defthm example
      (implies (and (subbagp x z)
                    (subbagp (append z v) w)
                    (subbagp w y))
               (subbagp x y)))
```

Think: x ⊆ z ⊆ (append z v) ⊆ w ⊆ y

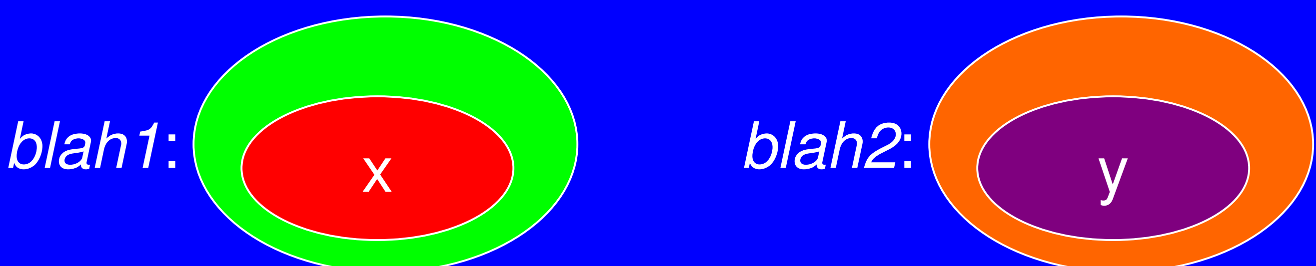
The Rule for Disjointness, part 1

To show (disjoint x y):

Discover (disjoint *blah1 blah2*), and then show (subbagp x *blah1*) and (subbagp y *blah2*).

or vice versa:

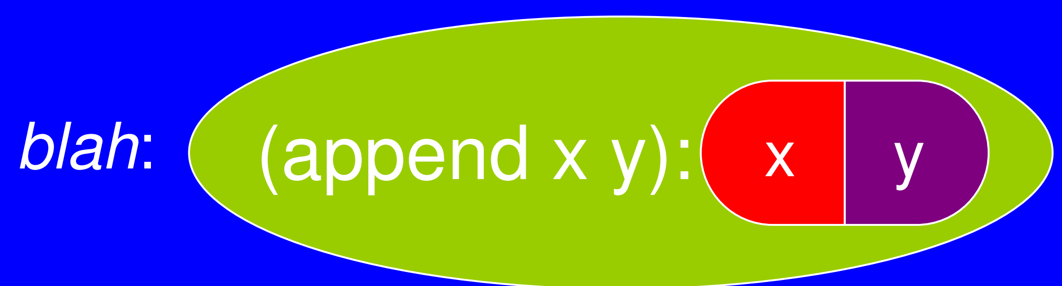
Discover (disjoint *blah1 blah2*), and then show (subbagp y *blah1*) and (subbagp x *blah2*).



The Rule for Disjointness, part 2

Or, to show (disjoint x y):

Discover (unique *blah*), and then show (subbagp (append x y) *blah*).



Other Predicates We Handle

- (unique x)
- (memberp a x)
- (not (memberp a x))
- (not (equal a b))
- (subbagp (append x y) bag) and similar predicates

Implementation

- Our :meta reasoning is of the “extended” sort. That is, we make use of the metafunction context (or mfc).
- We call mfc-type-alist to get the collection of currently known facts.
- But ACL2 has no axioms about mfc-type-alist!
- So our :meta rules must generate hypotheses.
- Before applying the rules, ACL2 must relieve the hypotheses.

Problem with ACL2

- The problem: Variables which are mentioned in the generated hypotheses -- but not in the rule's left-hand-side -- are treated as free. So ACL2 searches for free-variable matches. This isn't what we want at all!
- Ex: Show (subbagp x y) using (subbagp x z) and (subbagp z y).
- The terms mentioning z came from the type-alist.
- So don't try to match z with something else!

Change to ACL2

- Generated hypotheses can now contain, in essence, calls of bind-free.
- Now our code can bind the variables.
- Now we can write solid :meta rules that use the metafunction context.

:Meta Rules in Action

Our rules prove these theorems in about 0.01 seconds each:

```
(defthmd disjoint-test4
  (implies (and (subbagp x x0)
                (subbagp y y0)
                (subbagp (append x0 y0) z)
                (subbagp z z0)
                (subbagp z0 z1)
                (unique z1))
           (disjoint x y)))

(defthmd non-memberp-test1
  (implies (and (subbagp p q)
                (subbagp q (append r s))
                (subbagp (append r s) v)
                (memberp a j)
                (subbagp j (append k l))
                (subbagp (append k l) m)
                (disjoint m v)
                )
           (not (memberp a p))))
```

Future work

- Make the interface more abstract (e.g., use bag-sum instead of append).
- Add more bag functions to the library (e.g., bag-intersection).
- Consider sorting the elements of our bags.
- Investigate the few instances where we still have to enable the basic rules.
- Could we use something like a decision procedure for bags? (Keep a pot of bag facts analogous to the pot of linear arithmetic facts?)

Conclusion

- We've implemented a library about bags. It has been used at Rockwell, and we hope others will use it too.
- The library uses fancy :meta rules when the basic rules would cause quadratic blowups.
- The :meta rules are fairly nice. (To show *foo*, discover a term of the form *bar*, and then show *baz*.)
- The :meta rules access the mfc. Our work led to a change in ACL2 which will help others who want to use facts from the mfc.