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 microprocessor has instructions that access memory.
- Rockwell has a library, GACC, for reasoning about such instructions.
- To say that two memory operations don't interfere is to say that they affect different addresses.
- GACC often deals with collections of addresses.
- This sort of reasoning seems common to many systems which access memory.

- Two main ways to show that A and B are different addresses:

  (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.

- For this reasoning we care about the presence of elements in the collections, but not their order.
- Situation (2) requires us to keep track of duplicates.
- So, we use multisets (or “bags”).

Operations That Return Bags:

- (bag-insert a x) : Insert element A into bag X. [currently implemented with cons]
- (bag-sum x y) : Combine the bags X and Y. [currently implemented with append]
- (remove-1 a x) : Remove the first occurrence of element A from bag X.
- (remove-all a x) : Remove all occurrences of element A from bag X.
- (bag-difference x y) : Remove the elements in bag Y from bag X.

Predicates on Bags:

- (bagp x) : Test whether X is a bag [currently: t]
- (empty-bagp x) : Test whether X is an empty bag. [currently: (not (consp x))]
- (memberp a x) : Test whether A appears in bag X. (always returns a boolean)
- (subbagp x y) : Test whether, for any element A, A appears in bag Y at least as many times as it appears in bag X.
- (disjoint x y) : Test whether X and Y have no elements in common.
- (unique x) : Test whether no element appears in X more than once.

Misc.

- (count a x) : Return the multiplicity of A in X.

- (perm x y) : Equivalences relation to test whether two bags are permutations of each other (i.e., whether they agree on the count for each element)

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 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))))
```

These 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 like unique-of-append and instead manage the reasoning ourselves.
- 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 an “argument” that the predicate of interest is true.
- Ex: To show (subbagp x y), try to find several subbagp facts which can be chained together using transitivity.

Situations our :meta rules can handle:

- (subbagp x y)
- (unique x)
- (disjoint x y)
- (memberp a x)
- (not (memberp a x))
- (unique-subbagps x y bag) : Both X and Y fit inside BAG.
- (unique-memberps a b bag) : Both A and B fit inside BAG.
- (unique-memberp-and-subbagp a x bag) : Both A and X fit inside BAG.
- (not (equal a b))

Example: Subbagp Chain

- Intuition: To show (subbagp x y), we find some known fact, F, of the form (subbagp x BLAH), and then show (subbagp BLAH y), recursively. When asked to show something is a subbagp of itself, we immediately return t.
- Subtlety: We have the notion of a "syntax subbag." To say (syntax-subbagp x y) means we can tell (subbagp x y) just by looking at X and Y.
- Ex: (syntax-subbagp '(cons a b) '(append b (cons a c)) is true for all a, b, and c.
- Fact F above need not have x itself as its first argument. It's enough to find (subbagp BLAH1 BLAH2) where x is a syntax subbag of BLAH1. We then try to show (subbagp BLAH2 y).

Example: Subbagp Chain, cont.

Ways to show (subbagp x y):

1. Discover that (syntax-subbagp x y).
2. Find (subbagp BLAH1 BLAH2), where (syntax-subbagp x BLAH1), and then show (subbagp BLAH2 y).

   Think: $x \subseteq BLAH1 \subseteq BLAH2 \subseteq y$

Note that this is a recursive procedure.

Concrete Example

```
(defthmd example
      (implies (and (subbagp x z)
                    (subbagp (append z aa) w)
                    (subbagp w y))
               (subbagp x y)))
```

Think: $x \subseteq z \subseteq (\text{append } z\ aa) \subseteq w \subseteq y$

Example: Disjointness

Ways to show (disjoint x y):

1. Find (disjoint BLAH1 BLAH2), and then show (subbagp x BLAH1) and (subbagp y BLAH2).
2. Find (disjoint BLAH1 BLAH2), and then show (subbagp y BLAH1) and (subbagp x BLAH2).
3. Find (unique BLAH), and then show (unique-subbagps x y BLAH).

Implementation Details

- Our :meta reasoning is of the “extended” sort. That is, we make use of the mfc (or metafunction context). In particular, we search the type-alist for known facts.
- Each extended :meta rule requires a generated hypotheses (roughly, the conjunction of all the facts we used to convince ourselves that the rewrite is okay).

Change to ACL2

- By default, variables which are mentioned in the generated hypotheses for a :meta rule -- but not in the rule's left-hand-side -- are treated as free. So ACL2 searches for matches. This isn't what we want at all!
- Ex: If we can show (subbagp x y) by finding a subbagp chain involving (subbagp x z), we know exactly what z should be bound to (namely, z itself!), since we found (subbagp x z) on the type-alist.
- It is now legal for hypothesis metafunctions to generate, in essence, calls of syntaxp and bind-free. So we can bind the variables ourselves.
- It is now legal to add a :backchain-limit-lst to :meta rules to ensure that no work is done when relieving the hypotheses. (Our hyps come straight from the type-alist, so backchaining should never be necessary.)
- Together, these changes let us write solid :meta rules that make use of information from the mfc.

:Meta Rules in Action

Our rules prove these 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

- Add more bag functions to the library (e.g., bag-intersection).
- Make the interface more abstract (e.g., use bag-insert instead of cons).
- Investigate the instances where we have to enable the basic rules.
- Is there an interesting decidable theory here?

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*, find a term of the form *blah*, and then show *bar*.)
- The :meta rules access the mfc. Our work led to a change to ACL2 which we think will help others who want to access the mfc.