An ACL2 Library for Bags (Multisets)

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

ewsmith@stanford.edu,
serita@cs.utexas.edu,
dagreve@rockwellcollins.com,
mmwildin@rockwellcollins.com,
rjricha1@rockwellcollins.com

Rockwell Collins Advanced Technology Center
Cedar Rapids, IA 52498 USA

November 19, 2004

Abstract

In support of our ongoing ACL2 work, Rockwell Collins has developed a library of definitions and lemmas for bags (or multisets). This paper describes that library. Our early work with bags had the limited purpose of supporting a proof about the AAMP7 microprocessor but contained many nice features, including fancy `bind-free` and `:meta` rules for use when more basic rules would be expensive. We have collected the bag definitions and rules into a library and have added many new rules to make the library more widely useful. Work on the library continues.

1 Motivation and History

The bags library arose from Rockwell Collins' proof of a separation property for the AAMP7 microprocessor [2]. In particular, bags support GACC, the "generalized accessor" library [1]. In the AAMP7 work, the elements of our bags were typically addresses in the processor's address space. Often, after collecting the bag of addresses occupied by a structure in memory, we needed to claim that it contained no duplicates or that it was disjoint from some other bag. Such independence claims were needed to establish non-interference between different operations, and we believe that reasoning of this sort is inherent in the analysis of real-world systems that manipulate memory.

The bags library contains two main types of rules: basic rules to rewrite bag claims in the typical ACL2 fashion, and efficient `:meta` and `bind-free` rules to handle common situations in which the basic rules cause quadratic blowups. The `:meta` and `bind-free` rules were developed before many of the basic rules.[1]

Having a separate library for bags has several advantages. First, we can reuse the bag reasoning in future projects at Rockwell Collins. Second, we can distribute the library to the ACL2 community. (We expect that the bags library will be distributed with the books from the 2004 ACL2 Workshop.) Third, with all of the bag rules collected in one place, we can more easily make experimental changes to the library to try out different approaches.

*Eric Smith is currently a Ph.D. student at Stanford University.

†Serita Nelesen is currently a Ph.D. student at the University of Texas at Austin.

[1]Serita Nelesen proved many `:meta` and `bind-free` rules in the summer of 2003, from a specification provided by David Greve. In the summer of 2004, Eric Smith extracted all of the bag rules into a separate library called `bags`, added a few `:meta` rules, replaced some `bind-free` rules with `:meta` rules, and added definitions and rules to flesh out the basic theory of bags.

The rest of this paper has the following structure: Section 2 introduces bags and argues that they are the most appropriate way to represent collections of AAMP7 addresses. Section 3 outlines the basic theory of bags, and Section 4 discusses the basic bag rewrite rules. Section 5 shows that the basic rules are insufficient for reasoning about real-world systems, and Section 6 discusses the fancy rules that we use in place of some of the basic rules. Section 7 discusses a change made to ACL2 after our work uncovered a subtle issue. Section 8 shows that our library easily proves some non-trivial bag theorems, and Section 9 discusses future work.

2 Introduction to Bags

Bags are collections in which the multiplicity of elements is significant but their order is not. One might think of bags as variants of sets which can meaningfully contain duplicates. Thus, bags are sometimes called multisets. One might also think of bags as variants of lists in which the order of elements is irrelevant.

Our AAMP7 work often required us to reason about the collections of addresses occupied by structures in memory. We believe that those collections are best represented as bags, not as sets or as simple lists.

A common claim made in the AAMP7 proofs is that the addresses occupied by a structure are all unique. To express this claim, we collect the structure's addresses into a bag and then use the predicate `unique` to claim that the bag contains no duplicates. This tells us that different parts of the structure do not overlap with each other, a fact that seems crucial if we are going to reason about destructive updates to it. We need to know that updating one part of the structure doesn't change other parts! Note that using a set instead of a bag in this way would not work. If we collected the structure's addresses into a set, knowing that that set was `unique` would tell us nothing. Since sets cannot contain duplicates, their uniqueness is trivially true. Since we use a bag, any overlap of the structure with itself would result in some address appearing more than once in the bag.

On the other hand, we don't want to use lists for our collections of addresses, because lists respect the order of their elements. Two lists are equal only if their elements appear in the same order, but most of the operations on our collections of addresses are not concerned with the order of elements. Reordering a collection has no effect when we test for membership of an element in the collection, when we test whether the collection contains another, when we test whether the collection contains no duplicates, or when we test whether the collection is disjoint with another collection.

Furthermore, some of the AAMP7 proofs require a way to express the equivalence of collections which differ in the order of their elements. For example, imagine that we first extract the addresses occupied by the pointer fields of a data structure and then extract the addresses occupied by the value fields. Combining the two resulting collections should yield a collection that is equivalent to the collection of *all* the addresses occupied by the structure. But the equivalence used in this case cannot be the standard `equal` test for lists, since extracting the pointer and value addresses separately and then combining them will put the addresses in an order different from that obtained by extracting all of the addresses at once. Our library has a notion of equivalence, called `perm`, which exactly captures what we want to express in this case. Using `perm` to test for equivalence indicates that you are probably not dealing with simple lists.

Bags, which model multiplicity but not order, fall between lists, which model both, and sets, which model neither. For the reasons discussed above, we believe bags are the best choice for representing the collections of addresses in our work.

3 The Theory of Bags

Here are the main functions that deal with bags:

Functions that return bags:

```
(bag-insert a x) ;insert element A into bag X [currently we use cons]
(bag-sum x y) ;combine the bags X and Y [currently we use append]
(remove-1 a x) ;remove the first occurrence of element A from bag X
(remove-all a x) ;remove all occurrences of A from bag X
(bag-difference x y) ;remove the elements in bag Y from bag X [currently we have remove-bag]
```

Predicates on bags:

```
(bagp x) ; test whether X is a bag [currently returns t]
(empty-bagp x) ;test whether X is an empty bag [currently we use (not (consp x))]
(memberp a x) ;test whether A appears in bag X (returns a boolean, unlike member)
(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
```

Other functions:

```
(count a x) ;return the multiplicity of A in X
```

Equivalence relation on bags:

```
(perm x y) ;test whether two bags are permutations of each other
           ;(i.e., whether they agree on the count for each element)
```

We currently implement bags as simple lists. Because we do not sort our bags, lists that are permutations of each other represent the same bag. For example, the bag containing `1` and `2` can be represented by `'(1 2)` or `'(2 1)`. (But perhaps we *should* sort our bags; see "Future Work" below.)

4 Basic Rules

The bags library contains many rules characterizing the basic theory of bags. Those rules are too numerous and varied to list here, so we refer the reader to the books in the library itself for the details.

Many of the basic bag rules are fairly straightforward and conform to typical ACL2 usage. We attempt to normalize bag terms, and we simplify applications of predicates to bag terms. Worth noting are the many `:congruence` rules for the equivalence relation `perm`. For example,

```
(defcong perm equal (memberp a x) 2)
```

says that permuting the second argument to a `memberp` call has no effect on what the `memberp` call returns.

Among our basic rules are some which we keep enabled, such as

```
(defthm remove-1-of-cons-same
  (equal (remove-1 a (cons a x))
         x))
```

However, some basic rules can be expensive, as we describe in the next section.

5 Why the Basic Rules Are Insufficient

In real-world reasoning, the basic bag rules can be too expensive. Consider the following 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))))
```

At first glance, these seem like perfectly reasonable rules, and they probably *are* reasonable for some applications. However, our proofs often deal with large terms. It is not unusual to see the function `unique` applied to a nest of `append`s which is dozens of levels deep. Consider what the rules above do to the terms they rewrite. For example, rewriting `(unique (append a b c d e f))` yields the following conjunction:

```
(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))
```

Note that this conjunction contains a disjointness claim for every pair of arguments to `append`. In general, rewriting `(unique (append` a_1 a_2 `...` a_n `))` will result in a conjunction of n `unique` claims and "n choose 2" $= (n^2 - n)/2$ `disjoint` claims. It turns out that the number of mentions of the a_i in the result is exactly i^2, since each of the $(n^2 - n)/2$ `disjoint` claims mentions two of the a_i and each of the n `unique` claims mentions one of the a_i. Rewriting a "`unique` of `append`" term causes a quadratic blowup in the number of mentions of the subterms. We believe that this quadratic blowup makes the basic bag rules too expensive to apply to the large terms in our most demanding proofs; instead we use `:meta` and `bind-free` rules to do much of our bag reasoning.

6 Our Fancy Rules

We have seen that rewriting a "`unique` of `append`" term results in a quadratic blowup. So instead of rewriting such a term, we leave it un-rewritten. We get away with this because, roughly speaking, our fancy `bind-free` and `:meta` rules can look into such un-rewritten terms and access the information therein.

The rest of this section focuses on our newest and most advanced rules, which are `:meta` rules used to establish various bag predicates. (These rules replace some of our earlier `bind-free` rules.) Most of these `:meta` rules are of the extended sort. That is, they use the metafunction context, or `mfc`. (See the documentation topic `EXTENDED-METAFUNCTIONS`.) In particular, our rules use `mfc-type-alist` to access all of the facts that are currently known when we rewrite a term. Roughly speaking, these facts include the hypotheses of the current goal and the negation of its conclusion, as well as other facts, such as those which arise from forward chaining. (But of course care is taken to avoid the circularity of using facts which arose from the current literal itself.)

Most typically, our bag reasoning involves the need to establish a predicate on bags (e.g., `subbagp` or `disjoint`). Often the facts needed to do so are explicitly present in the goal but are left un-rewritten (since rewriting them might be expensive, as just discussed). Such a situation is usually handled by a `:meta` rule. The

`:meta` rule searches through the type-alist (i.e., the known facts) for some line of reasoning which establishes that the predicate being rewritten is true. If it finds such a line of reasoning, it rewrites the predicate to `t`. Otherwise, it makes no change.

Our simplest `:meta` rule can establish a `subbagp` claim by finding a "subbag chain" and appealing to the transitivity of the subbag relation. For example, it can establish `(subbagp x y)` by finding a sequence of bags that begins with `x` and ends with `y` and that has the property that each bag in the chain is a subbag of the bag that follows it. So if the type-alist contains `(subbagp x z1)`, `(subbagp z1 z2)`, and `(subbagp z2 y)`, the rule can conclude `(subbagp x y)`. It uses the chain from `x` to `z1` to `z2` to `y`. (The bag `x` is a subbag of `z1`, which is a subbag of `z2`, which is a subbag of `y`. So `x` must be a subbag of `y`.)

Sometimes we can tell just by looking at two terms that one is a subbag of the other, and the `:meta` rule for `subbagp` uses such "syntactic" facts. For example, `x` must always be a subbag of `(append x z)` – regardless of the values of `x` and `z` – because `(append x z)` "syntactically" contains `x`. In this case, we say that `x` is a "syntactic subbag" of `(append x z)`, and we get one link of the chain "for free" (i.e., without having to discover any facts on the type-alist.) So if our `:meta` rule discovers `(subbagp (append x z) y)` on the type-alist, it can immediately conclude `(subbagp x y)`. The subbag chain has two links: `(subbagp x (append x z))`, which we get syntactically, and `(subbagp (append x z) y)` which we get from the type-alist.

Our `subbagp :meta` rule can prove the following theorem:

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

by constructing a subbag chain from `x` to `z` to `(append z v)` to `w` to `y`. (Here the link between `z` and `(append z v)` is of the syntactic sort, and the other three links come from the type-alist.)

To do most of its work, our `subbagp :meta` rule calls a "show function" called `show-subbagp-from-type-alist`. That function searches the type-alist for a fact which justifies the first step in a potentially useful line of reasoning, and it then calls itself recursively to do the rest of the work. For example, if it finds `(subbagp x z)` then it needs only to show `(subbagp z y)` in order to conclude `(subbagp x y)`. So it calls itself recursively on `(subbagp z y)`. If this recursive call succeeds, the function returns successfully. Otherwise, it continues searching the type-alist for potentially helpful facts.

We have `:meta` rules to establish predicates other than `subbagp`. Each rule has a "show function" which takes the first step in a line of reasoning and then just calls other "show functions" (perhaps including itself, recursively) to do the rest of the work. For example, `show-disjoint-from-type-alist` can establish `(disjoint x y)` in several ways:

1. It can discover `(disjoint blah1 blah2)` on the type-alist and then call `show-subbagp-from-type-alist` to show both `(subbagp x blah1)` and `(subbagp y blah2)`. The intuition here is that if `blah1` and `blah2` have no overlap, but `blah1` contains `x` and `blah2` contains `y`, then `x` and `y` cannot overlap.

2. It can follow the same reasoning as in 1. but with `x` contained in `blah2` and `y` contained in `blah1`.

3. It can discover `(unique blah)` and then show `(subbagp (append x y) blah)` by calling another "show function," `show-unique-subbagps-from-type-alist`, which is tuned to establish just that sort of "`append` is a `subbagp`" fact. The intuition here is that `x` and `y` cannot not overlap because, if they did, `(append x y)` would contain duplicates, and so would its superbag `blah`, which we know contains no duplicates.

Note that `show-disjoint-from-type-alist` merely finds the first step in a line of reasoning and then calls other "show functions" to do the rest of the work. Such a design is typical of our most advanced `:meta` rules.

One of the ways that our `:meta` rule for disjointness can work is by finding an appropriate call to `unique` on the type-alist. For example, it can establish `(disjoint x y)` from `(unique (append x z w y v))`. This works even though the "`unique` of `append`" term is not simplified. Recall that simplifying such terms can cause quadratic blowups, so we leave them unsimplified. In such situations, we need the `:meta` rule in order to establish the predicate we care about (in this case, `disjoint`).

Our library contains a fancy bag rule to establish each predicate we care about, even in the presence of unsimplified terms. Since the library supports GACC, the generalized accessor library, our fancy rules are currently

aimed at establishing the four predicates of most interest to GACC: `disjoint`, `subbagp`, `memberp` (or the negation of `memberp`), and `unique`. (We also have limited support for using bag facts to show that two elements of bags must be unequal.)

Unfortunately, the reasoning employed by our fancy rules is weak in the sense that it is only "syntactic." That is, it relies only on information explicitly present on the type-alist. For example, imagine that the type-alist contains some non-bag facts which together imply `(unique x)` but that `(unique x)` is not derivable solely from the bag facts on the type-alist. In this case, our fancy rules will not be able to use the `(unique x)` fact when they search for proofs. (Perhaps `:forward-chaining` rules could help in cases like this, but they would require the user to spend time proving them, and they might cause slower proofs. Or perhaps our fancy rules could call `mfc-rw` to check whether key facts missing from their lines of reasoning might turn out to be provable using deeper (non-bag) reasoning. But this seems potentially very expensive.) In any case, our rules seem strong enough to handle most of the cases that arise in the AAMP7 proof.

We believe that our fancy bag rules represent a good trade-off between performance and power. In fact, we think that our rules do pretty well, considering that they operate in a context where, for performance reasons, we refrain from rewriting certain terms.

7 A Change to ACL2

One complication of using extended metafunctions is that they are not allowed to trust the information contained in the `mfc`; no axioms are provided about the values returned by functions like `mfc-type-alist`. Thus each `:meta` rule generates a hypothesis which includes all the facts involved in the line of reasoning it has discovered. That hypothesis must be relieved before the `:meta` rule can fire. Each of our "show functions" thus has a corresponding "hyp-for" function, which, after the "show function" succeeds, goes back and collects all the facts used in the successful line of reasoning. (For more information, see the code in `eric-meta.lisp` in the bags library and see the documentation topic `EXTENDED-METAFUNCTIONS`.)

Our use of generated hypotheses in our metafunctions revealed what we believe to be a subtle weakness in versions of ACL2 up through 2.9. (The issue has been addressed in version 2.9.1.) With the old behavior, variables which appear in a `:meta` rule's generated hypothesis – but not in the rule's left-hand-side – are treated as free. So ACL2 searches for matches for those variables when relieving the hypothesis. This isn't what we want at all! For example, our `subbagp` rule can show `(subbagp x y)` by finding `(subbagp x z)` and `(subbagp z y)` on the type-alist. The generated hypothesis in this case is essentially the conjunction of `(subbagp x z)` and `(subbagp z y)`. But the variable z does not appear in the rule's left-hand-side: `(subbagp x y)`. So when relieving the generated hypothesis, ACL2 treats `z` as a free variable. This isn't what we want at all! The facts we discovered on the type-alist are about `z` itself, so we don't want ACL2 to try to find other bindings for z.

We found an example where this behavior prevented a `:meta` rule from firing. (Roughly speaking, ACL2 always splits the generated hypothesis into conjuncts. In our example, ACL2 found a free variable match which satisfied one of the conjuncts but which failed to satisfy a later conjunct. Since :meta rules behave as if `:match-free :once` has been declared, one spurious match was enough to cause the whole rule to fail.)

The problem described above has been addressed in ACL2 2.9.1. The crucial change is that generated hypotheses can now contain what are essentially calls to `bind-free` (actually, calls to `synp`, which is what the `bind-free` macro expands to). Such calls were previously illegal in generated hypotheses for `:meta` rules. Our generated hypotheses include such calls to instruct ACL2 to bind all "free variables" to themselves. Thus, we can prevent ACL2 from searching for other bindings of those variables.

This change to ACL2 (along with some other related changes) allows us to write solid extended `:meta` rules. We hope that these improvements will help others who also want to write `:meta` rules which use facts from the `mfc`.

8 Some Difficult Examples

As a demonstration of the power of our bags library, we exhibit the following lemmas:

```
(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))))
```

Our bags library can prove each of these lemmas in about 0.01 seconds on a fast machine. This kind of speed is important because many of our proofs are large.

9 Future Work

The bags library is a work in progress. It implicitly includes many design decisions, and we are not sure that we have made the best choice in each case. For example, some design decisions may have been made for historical reasons which are no longer relevant. But we can say this: The bags library is sufficient to support the AAMP7 separation proof. Each change made to the library is tested by replaying that proof after the change. If a proposed change requires more than minor tweaking of the proofs (e.g., the addition of many hints) or significantly slows down the proofs, we consider refraining from making the change.

The following list contains ideas for future versions of the bags library:

- We would like to continue to flesh out the library by adding more rules and ensuring that all relevant functions are present. Some typical bag functions, especially `bag-union` and `bag-intersection`, were not necessary for the AAMP7 work. We should add those functions and prove rules about them. (Note: We do handle `bag-sum`, currently implemented by `append`. This paper distinguishes `bag-sum`, in which the count of an element in the result is the *sum* of its counts in the arguments, from `bag-union`, in which the count of an element in the result is the *maximum* of its counts in the arguments.)

- We would like to make the interface to the bags world more abstract. For instance, instead of using `cons` and `append` to build and combine bags, we would prefer to use our own functions `bag-insert` and `bag-sum`. This will allow us to change the implementation of such functions easily in the future. Since bag insertion was previously done with `cons`, we will have to search for `cons` and replace some calls of it with `bag-insert`. Human intervention will be required to determine whether each call to `cons` is doing bag insertion or performing some other non-bag operation. Once we have transitioned to use `bag-insert` instead of `cons`, future changes to `bag-insert` (e.g., if we decide to sort our bags, as discussed below) will require much less human intervention.

- We could normalize our bags by sorting their elements. This would allow us to test for bag equality using `equal` instead of `perm`, since each bag would have a unique representation. Bag equality is central to the theory of bags, and, for a variety of reasons, we think it might be better to use plain old `equal` for bag equality instead of using `perm` and its associated `:congruence` rules. However, we would have to handle non-bags intelligently. What if the user types `'(2 1)` and uses that value as a bag? Perhaps we could address this issue by providing a function that converts lists into bags.

- Currently, any ACL2 object satisfies `bagp`. However, we might consider requiring bags to be true-lists. Currently, bag functions can return non-true-lists; for example, `(remove-1 'a 3)` yields 3. We believe that this behavior has advantages for syntactic reasoning (in particular, it is convenient for the writer of metafunctions), but we would like to investigate the matter further. (Would making a more restrictive `bagp` introduce hypotheses into our rules? If so, would they be hard to relieve? Could we drop those hypotheses by recasting our rules in terms of a `bag-fix` function?)

- Sometimes the proofs that employ the bags library enable the bag functions. Ideally, the user of our library would never have to look inside these functions. Perhaps we could prove more rules about the bags functions to spare the user from having to expose their internal details. Also, sometimes our proofs enable the potentially expensive basic bag rules. We would like to investigate such situations to determine whether we could handle them with additional `bind-free` or `:meta` rules or whether the proofs in question are inherently too deep for our syntactic rules to handle.

- We would like to provide better documentation for the library, especially documentation about the fancy rules.

- Finally, we wonder whether we could use an efficient decision procedure (or something like a decision procedure) to answer some or all bag questions. Is some useful theory involving bags efficiently decidable? Might we somehow integrate a decision procedure for bags (or for a subset of the theory of bags) into ACL2? Would doing so buy us anything, such as faster proofs? Could our work benefit from the creation of a "bag pot" (analogous to ACL2's linear pot) to store all currently known bag facts?

10 Works Cited

References

[1] Greve, D. Address Enumeration and Reasoning Over Linear Address Spaces. In *ACL2 Workshop 2004* (November 2004).

[2] Greve, D., Richards, R., and Wilding, M. A Proof-Free Summary of Intrinsic Partitioning Verification. In *ACL2 Workshop 2004* (November 2004).