A Generic Instantiation Tool and a Case Study: A Generic Multiset Theory.*

F.J. Martín-Mateos, J.A. Alonso, M.J. Hidalgo and J.L. Ruiz-Reina
http://www.cs.us.es/{~fmartin,~jalonso,~mjoseh,~jruiz}
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

February 27, 2002

Abstract

In some cases, when we develop a formal theory in ACL2, it would be desirable that the definitions and theorems of the theory be as independent of a concrete implementation as possible. This is particularly interesting when we design theories about basic data types, making those developments more general, reusable and extensible. At the same time, it would also be desirable to be able to instantiate (in a convenient way) the definitions and theorems of the theory, for a concrete implementation. In this paper we present the development of a particular generic theory, and a tool to instantiate its events. As a case study we have chosen to describe a generic theory about finite multisets. It is also shown how this generic theory can be instantiated (using several macros we have defined) to build a theory about two different implementations of multisets. Finally we propose some directions for further research in this topic.

1 Motivation of this work

In [7] we proved the well-foundedness of an extension to finite multisets of a well-founded ACL2 relation, and showed how this result can be used to prove the termination of non-trivial recursive equations. In that work, multisets were represented using lists, and some of the theorems proved there depend on this particular representation. Now suppose that we have another ACL2 book developing a theory about finite multisets, based on a different implementation (for example, using association lists to represent characteristic functions). If we had wanted to use our well-foundedness theorem with this alternative representation of multisets, we would have had two possibilities:

- Change the well foundedness theorem to use the new multiset implementation.
- Rebuild the finite multiset book, changing the implementation.

This is illustrative of the following question: do the theorems proved in ACL2 about some data types reflect properties of these data types or of a concrete implementation of them? And also this example raises a more practical question: how do we develop an ACL2 book to be reusable? Some of the desirable properties for a book to be reusable are:

1. Generality: the more generality has a book, the more reusable it will be. Of course, this can complicate the proofs.

*This work has been supported by MCyT: Project TIC2000-1368-CO3-02

2. Tools: A book should provide tools to provide a convenient way to use the results proved in it.

3. Extensibility: The book should be capable of being extended.

4. Documentation: If the book has a good documentation, it will be easy to reuse its results.

We now concentrate on the discussion about how to improve the first and the second properties: how to make a book as general as possible, and how to implement a tool to instantiate the results of a book containing generic theorems. This is particularly interesting when we design theories about basic data types, making those developments more general and reusable.

One of the reasons why a book can lose generality is that sometimes the results in this book may rely on a particular representation of the objects we are reasoning about. For example, as we said above, to develop a book about finite multisets, we have at least two possibilities of implementation, using lists containing their elements or using association lists to represent their characteristic functions. It would be more interesting a book in which the properties are proved independently of their representation.

Another way of generality is obtained from some functions not directly related with the matter of the book, but with the properties needed in it. For example, in the multiset theory we could use any equivalence relation to compare the elements of multisets. Of course, the most common choice would be `equal`, but other equivalence relations are possible. It will be more useful a book about finite multisets of objects comparable with a generic equivalence relation `equiv`.

This kind of generic development if feasible in ACL2 with the `encapsulate` mechanism, which allows the user to introduce function symbols by axioms constraining them to have certain properties. To ensure consistency, witness functions having the same properties have to be exhibited. Thus, it is possible to develop a generic theory about a data type if one states the fundamental abstract properties of the data type by means of an `encapsulate`.

To reuse the results of a book developed in this way, the derived rule of inference of *functional instantiation* can be used: theorems about partially defined functions can be instantiated for new functions if they are known to have the same properties. However, the number of events to instantiate can be high, and moreover the potential user has to know the event names of the generic book. It would be better if the generic book provides some tools to instantiate its events in a more convenient way.

In this paper we discuss the development of a generic theory, and we describe some tools we have developed to make easier the instantiation process. As a case study, the generic theory we develop is about finite multisets. There is no special reason for this choice (except, as we said before, we had already proved some results about them). In fact, we think the same development could be done for any other basic data type.

In the sequel, when we talk about an ACL2 theory, we mean a sequence of events admitted by the system, as they appear in a book. For the moment we will only consider `defun` and `defthm` events. An ACL2 generic theory is a theory that can be derived from a set of functions with some properties (established with `encapsulate`).

The paper is structured as follows. In section 2, we describe our generic theory about multisets. Once described the assumed abstract properties, we show how we can define different operations and prove their properties, only from the assumed axioms. In section 3, we show how this generic book also provides the tools needed to instantiate the generic theory it describes. In section 4, we show how to instantiate the generic theory described in section 2, for two different implementations of multisets. Finally the last section discusses some conclusions and further work.

2 A finite multiset generic model

Intuitively, a multiset (also called a *bag*) is a set "with repeated elements". We are going to define our finite multiset generic theory in a style similar to an abstract data type, defining constructor and accessor functions, and also a predicate to identify empty multisets. Recursive functions on multisets are usually defined with the empty multiset involved in the basis case and the accessor functions in the recursive cases. For that reason, we need a well-founded measure to justify termination of these recursive functions. And finally, we need an equivalence relation, used to compare the elements of a multiset.

These functions are defined via `encapsulate`. The following are a list of the functions defined and the properties assumed about them:

- `(select *)` : To select an element from a multiset.
- `(reduct *)` : To reduce a multiset eliminating the element selected by `select`.
- `(include * *)` : To add an element to a multiset.
- `(empty *)` : To check if a multiset has no elements.
- `(measure *)` : The measure associated to a multiset. We assume the following properties about this function:
  1. The value returned is an ordinal:

     ```
     (defthm e0-ordinalp-measure
             (e0-ordinalp (measure m)))
     ```

  2. The measure of the result of reducing a non-empty multiset with the `reduce` function is smaller than the measure of the original multiset:

     ```
     (defthm reduct-measure
       (implies (not (empty m))
                (e0-ord-< (measure (reduct m)) (measure m))))
     ```

- `(equiv * *)` : An equivalence relation used to compare the multiset elements. We assume the properties proved by `(defequiv equiv)`.

To finish our generic model we have to assume some properties establishing the relation between the accessor functions (`select` and `reduct`) and the constructor function (`include`). For example, we could assume `(equal (include (select m) (reduct m)) m)`. But this assumption would be implementation dependent, because in some implementations, equality of multisets cannot be checked using `equal`. Moreover, we have not yet defined equality between multisets.

Therefore, we have to try a different approach to establish the relation between the accessor functions and the constructor function. Note that the fundamental concept in a multiset is not the membership relation, but *the number of occurrences of an element in a multiset*. And this value can be calculated by a recursive function using the accessor functions. We have named this function `count-bag-equiv` (it will be also referenced as μ in the sequel):

```
(defun count-bag-equiv (e m)
   (declare (xargs :measure (measure m)))
   (cond ((empty m) 0)
         ((equiv e (select m))
          (1+ (count-bag-equiv e (reduct m))))
         (t (count-bag-equiv e (reduct m)))))
```

Now we can state the following fundamental property, expressing the relation between `count--bag-equiv` and `include`:

```
(defthm count-include
  (equal (count-bag-equiv e1 (include e2 m))
         (if (equiv e1 e2)
             (1+ (count-bag-equiv e1 m))
             (count-bag-equiv e1 m))))
```

All the definitions and properties above are stated inside an `encapsulate`. At this point we have a basic multiset specification. A concrete implementation of finite multisets will be given by concrete definitions of these functions, verifying the assumed properties. From these properties, our intention is to develop a basic theory about finite multisets, defining new functions and proving properties about them. Is important to remark that the properties below are proved using only the above assumed axioms.

The first thing we have to do is to define the multiset equality[1] and prove some of its properties. We start defining the multiset membership relation, next the sub-multiset relation and finally the multiset equality. After that, we define some basic operations on multisets (union, intersection, minimal union and difference) and prove their main properties. All these definitions and theorems constitutes our generic theory about finite multisets.

2.1 The multiset membership relation

The elements of a multiset are obtained by means of the `select` function. The function implementing this relation is `member-bag-equiv`.

```
(defun member-bag-equiv (e m)
  (declare (xargs :measure (measure m)))
  (cond ((empty m) nil)
        ((equiv e (select m)) t)
        (t (member-bag-equiv e (reduct m)))))
```

We have proved that this function is congruent with respect to the equivalence relation `equiv` on the multiset elements. The following theorem characterizes the multiset membership relation with respect to the number of occurrences. This relation will be necessary to build a useful strategy to check the sub-multiset relation and multiset equality (see the appendix).

```
(defthm member-bag-equiv-count->=-1
  (iff (member-bag-equiv e m)
       (>= (count-bag-equiv e m) 1))
  :rule-classes :definition)
```

2.2 Removing one element from a multiset

To implement the sub-multiset relation, it is not enough to check that every element from the first multiset occurs in the second multiset. We have to take into account the number of occurrences. To achieve this, we have defined a function that removes one occurrence of an element from a multiset.

```
(defun remove-one-bag-equiv (e m)
  (declare (xargs :measure (measure m)))
```

[1] Unlike the theory of finite sets developed in [4], our theory is restricted to multisets "with one level".

```
  (cond ((empty m) m)
        ((equiv e (select m)) (reduct m))
        (t (include (select m)
                    (remove-one-bag-equiv e (reduct m))))))
```

We have proved that this function is congruent with respect to the equivalence relation `equiv` on the multiset elements. We have also characterized this function with respect to the number of occurrences:

```
(defthm remove-one-count
  (equal (count-bag-equiv e1 (remove-one-bag-equiv e2 m))
         (if (and (member-bag-equiv e2 m)
                  (equiv e1 e2))
             (1- (count-bag-equiv e1 m))
             (count-bag-equiv e1 m))))
```

2.3 The sub-multiset relation

We define the sub-multiset relation between two multisets `m1` and `m2` checking that the every selected element from `m1` occurs in `m2` and `(reduct m1)` is a sub-multiset of the resulting multiset obtained removing one occurrence of `(select m1)` from `m2`:

```
(defun sub-bag-equiv (m1 m2)
  (declare (xargs :measure (measure m1)))
  (cond ((empty m1) t)
        ((member-bag-equiv (select m1) m2)
         (sub-bag-equiv (reduct m1) (remove-one-bag-equiv (select m1) m2)))
        (t nil)))
```

An interesting property of `sub-bag-equiv` is the following:

```
(defthm sub-bag-equiv-count
  (implies (sub-bag-equiv m1 m2)
           (<= (count-bag-equiv e m1) (count-bag-equiv e m2))))
```

That is, if $M_1 \subseteq M_2$, then $\mu(e, M_1) \leq \mu(e, M_2)$, for all e. The converse of this theorem is also true. In fact, we can use this converse property to implement a strategy in order to prove theorems about the sub-multiset relation in a convenient way. This strategy is inspired on the work about finite sets in [4]. Let us describe this in more detail.

The converse of the above theorem can be expressed, in its more general form, as the following:

$$Hyp \rightarrow \forall e\ (\mu(e, M_1) \leq \mu(e, M_2)) \Longrightarrow Hyp \rightarrow M_1 \subseteq M_2$$

That is, if under some hypothesis Hyp, we can prove that $\forall e\ (\mu(e, M_1) \leq \mu(e, M_2))$, then under the same hypothesis we conclude that $M_1 \subseteq M_2$. Usually, reasoning about the number of occurrences is easier than reasoning about the sub-multiset relation. The above theorem can be stated in ACL2 in a general form, using `encapsulate`. Then it can be used by functional instantiation, to carry out a proof about the sub-multiset relation by reasoning about the number of occurrences of the elements in a multiset.

We have also implemented a function to compute an appropriate hint to prove of theorems about `sub-bag-equiv`. This hint is built by choosing an adequate functional instantiation of the theorem above. The function that computes the hint is called `defstrategy-sub-bag-hint`. In the appendix of this paper we explain in detail the formalization of the above theorem and the definition of the function that computes the hint.

Here is an example of how this computed hint can be used. In this case, the transitive property:

```
(defthm sub-bag-equiv-transitive
  (implies (and (sub-bag-equiv m1 m2)
                (sub-bag-equiv m2 m3))
           (sub-bag-equiv m1 m3))
  :hints (defstrategy-sub-bag-hint))
```

With this strategy, the theorem is proved automatically by the system, carrying out a proof based on the number of occurrences of elements in multisets. Since $M_1 \subseteq M_2$, then $\forall e\ (\mu(e, M_1) \leq \mu(e, M_2))$ and analogously $\forall e\ (\mu(e, M_2) \leq \mu(e, M_3))$. Therefore $\forall e\ (\mu(e, M_1) \leq \mu(e, M_3))$ and by the above theorem we have $M_1 \subseteq M_3$. This strategy hint is extensively used to prove properties of the `sub-bag-equiv` relation.

2.4 The multiset equality relation

We have defined the multiset equality relation based on the sub-multiset relation in a natural way:

```
(defun equal-bag-equiv (m1 m2)
  (and (sub-bag-equiv m1 m2)
       (sub-bag-equiv m2 m1)))
```

With the properties proved about the sub-multiset relation (reflexivity and transitivity) and the symmetry of this definition, we easily prove that multiset equality is an equivalence relation.

```
(defequiv equal-bag-equiv)
```

Finally, we prove the congruences of `member-bag-equiv`, `count-bag-equiv` and `sub-bag--equiv` with respect to the equality relation.

```
(defcong equal-bag-equiv iff (member-bag-equiv e m) 2)

(defcong equal-bag-equiv equal (count-bag-equiv e m) 2)

(defcong equal-bag-equiv iff (sub-bag-equiv m1 m2) 1)

(defcong equal-bag-equiv iff (sub-bag-equiv m1 m2) 2)
```

Similarly to the strategy given by `defstrategy-sub-bag-hint`, we have also defined an analogous strategy `defstrategy-equal-bag-hint`, to prove the equality of two multisets attending to the number of occurrences of their elements, based on the following theorem (see the appendix):

$$Hyp \rightarrow \forall e\ (\mu(e, M_1) = \mu(e, M_2)) \Longrightarrow Hyp \rightarrow M_1 = M_2$$

Again, this strategy hint is extensively used to prove properties about `equal-bag-equiv`.

2.5 Some operations on multisets

Once defined the (generic) equality between multisets, we have defined some operations on multisets: union, intersection, minimal union and difference, and we have proved some properties about them. Let us remark again that these properties are obtained only from the initial assumed properties.

For example, we will describe in more detail the union of two multisets. This operation is defined as the result of adding every element in the first multiset to the second one:

```
(defun union-bag-equiv (m1 m2)
  (declare (xargs :measure (measure m1)))
  (cond ((empty m1) m2)
        (t (include (select m1)
                    (union-bag-equiv (reduct m1) m2)))))
```

The number of occurrences of any element in the union is equal to the addition of the number of its occurrences in the original multisets. This property is fundamental to successfully use the strategies associated with the sub-multiset and equality relations:

```
(defthm count-union-bag-equiv
  (equal (count-bag-equiv e (union-bag-equiv m1 m2))
         (+ (count-bag-equiv e m1)
            (count-bag-equiv e m2))))
```

Here are a list of some basic properties we proved about union of multisets:

1. Congruences:

   ```
   (defcong equal-bag-equiv equal-bag-equiv (union-bag-equiv m1 m2) 1
      :hints (defstrategy-equal-bag-hint))

   (defcong equal-bag-equiv equal-bag-equiv (union-bag-equiv m1 m2) 2
      :hints (defstrategy-equal-bag-hint))
   ```

2. Commutativity:

   ```
   (defthm union-bag-equiv-conmutative
     (equal-bag-equiv (union-bag-equiv m2 m1) (union-bag-equiv m1 m2))
     :hints (defstrategy-equal-bag-hint))
   ```

3. Associativity:

   ```
   (defthm union-bag-equiv-associative
     (equal-bag-equiv (union-bag-equiv m1 (union-bag-equiv m2 m3))
                      (union-bag-equiv (union-bag-equiv m1 m2) m3))
     :hints (defstrategy-equal-bag-hint))
   ```

We have also defined (and proved some properties about them) the multiset intersection (`inter--bag-equiv`), difference (`diff-bag-equiv`) and minimal union (`unimin-bag-equiv`)[2]. See the supporting materials for details.

All these properties are proved in the book `generic-multiset.lisp`, which develops what we have named our "finite multiset generic theory", a sequence of definitions and theorems about finite multisets deduced from generic assumptions about them. But this book also contains a tool to make easy the instantiation of their properties for a concrete implementation, as we will explain in the next section.

[2]The union operation is not idempotent and has not the following property: the union is the smaller multiset that contains the original multisets. For example, if $M_1 = \{\{a, b\}\}$ then $M_1 \cup M_1 = \{\{a, b, a, b\}\} \neq M_1$ and if $M_2 = \{\{b, c\}\}$ and $M_3 = \{\{a, b, c\}\}$ then $M_1 \subseteq M_3$, $M_2 \subseteq M_3$ but $M_1 \cup M_2 \not\subseteq M_3$. There is another operation with these properties, called *minimal union*.

3 Generic theory instantiation tools

Suppose that somebody has developed a book with a generic theory about a data type (as the one described in the previous section). Now suppose a potential user of the book has developed a concrete implementation of this data type. If this implementation holds the assumed properties in the generic book, all the generic definitions and theorems can be translated to the concrete implementation, by means of functional instantiation. Our goal is to develop a tool to make easy this instantiation.

For that purpose, the author of the generic theory must define (as part of the generic book) a constant $*theory*$ whose value is the list of all instantiable events (definitions and theorems)[3]:

`(defconst` $*theory*$ $\langle list\ of\ instantiable\ events \rangle$`)`

Our intention is to provide assistance to the potential user of the generic book, making more convenient the instantiation process of the events stored in $*theory*$. For that purpose, we have defined a macro named `make-generic-theory`[4]. The author of the generic theory must include at the end of the generic book a call to this macro:

`(make-generic-theory` $*theory*$`)`

When the generic book is included by an user, the effect of this macro call is *the automatic definition* of a new macro named `definstance-`$*theory*$. For example, if (as part of the generic multiset book described in the previous section) we define the constant `*multiset*` and include a call to `(make-generic-theory *multiset*)`, a macro named `definstance-*multiset*` is automatically defined when this book is included. In a sense, we can see this macro as the tool provided by the author of the generic book to a potential user, in order to make easy instantiations of the generic events stored in the constant $*theory*$.

So the main idea is that the author of the generic book defines the instantiable events, and when an user includes it in her own book developing a particular implementation of the generic functions, she can easily instantiate the events of the generic book with only a simple macro call of `definstance-`$*theory*$. This macro has two arguments: the first one is an association list that relates the names of the concrete functions with the names of the generic functions and the second one is provided to give new names to the new events obtained by instantiation. This second argument is a string that will be used as a suffix to append to the generic names.

It should be clear that the macro we have defined is `make-generic-theory`. When this macro is called with a particular $*theory*$ constant containing a list of instantiable events of a generic book, the macro `definstance-`$*theory*$ is automatically defined, and it can be used to instantiate the generic events.

We will see two examples of the use of this instantiation tool in the next section. But previously we briefly describe two technical questions. First, this is the definition of the macro `make-generic-theory`:

```
(defmacro make-generic-theory (theory)
  (let ((definstance-theory
          (packn-string-in-pkg
           (string-append "DEFINSTANCE-" (string theory)) theory)))
    `(defmacro ,definstance-theory (subst suffix)
       (list* 'encapsulate
              'nil
              (functional-instance-lst ,theory subst (string-upcase suffix))))))
```

[3] At the end of this section we will explain how this list of events can be generated in a convenient way

[4] The tools defined in this section are included in the file `generic-theory` of the supporting materials. This file has to be included in any generic book.

The definition of the macro `definstance-`$*theory*$* is created by means of the function `functional-instance-lst`, that builds the list of instantiated events from the event list stored in $*theory*$. To instantiate a `defun` event is enough to replace in its body every occurrence of the generic function names with the concrete function names. The instances of `defthm` events are built in the same way, adding an adequate functional instantiation hint.

To build the functional instantiation hint associated with a `defthm` event, we have to know which functions are involved in it. If (when the event list was created) the elements were in the same order that they have in the generic book and this book has been certified, then we can assume that every `defthm` event depends only on the previous functions in the event list. Therefore, the functional instance hint associated with a `defthm` event can be built with a functional substitution binding every previously instantiated function and the association list used as the first argument of the `definstance-`$*theory*$* macro.

```
(defun remove-old-hints (event)
  (cond ((endp event) nil)
        ((eq ':hints (car event))
         (cddr event))
        (t (cons (car event) (remove-old-hints (cdr event))))))

(defun functional-instance (event old-name subst)
  (append (remove-old-hints event)
          `(:hints (("Goal"
                     :by (:functional-instance ,old-name ,@subst))))))

(defun functional-instance-lst (event-lst subst suffix)
  (declare (xargs :mode :program))
  (cond ((endp event-lst) nil)
        ((equal (car (car event-lst)) 'defun)
         (let ((new-subst
                (add-name-subst (cadr (car event-lst)) subst suffix)))
           (cons (subst-equal-lst new-subst (car event-lst))
                 (functional-instance-lst (cdr event-lst) new-subst suffix))))
        ((equal (car (car event-lst)) 'defthm)
         (let ((new-subst
                (add-name-subst (cadr (car event-lst)) subst suffix)))
           (cons (functional-instance
                   (subst-equal-lst new-subst (car event-lst))
                   (cadr (car event-lst))
                   subst)
                 (functional-instance-lst (cdr event-lst) subst suffix))))
        (t (functional-instance-lst (cdr event-lst) subst suffix))))
```

A second technical question is about the definition of the constant $*theory*$. Note that this constant has to store the instantiable events, not only their names. To make easy the creation of this event list, we have defined a macro that obtains it from the list of event names, by accessing the state of ACL2:

```
(defun get-event-lst-fn (event-name-lst wrld)
  (declare (xargs :mode :program))
  (cond ((endp event-name-lst) nil)
        (t (cons (get-event (car event-name-lst) wrld)
                 (get-event-lst-fn (cdr event-name-lst) wrld)))))
```

```
(defmacro get-event-lst (event-name-lst)
  `(get-event-lst-fn ,event-name-lst (w state)))
```

For example, in the case of our generic multiset theory, we define a constant `*multiset*` to store all the instantiable events of the book. The list of events can be obtained evaluating `get-event-lst` on the list of event names:

```
(get-event-lst
 '(member-bag-equiv
   equiv-implies-iff-member-bag-equiv-1
   remove-one-bag-equiv
   equiv-implies-equal-remove-one-bag-equiv-1
   ...))
```

Unfortunately, we do not know how to directly define a constant storing the result of the above macro, so it is necessary that the author of the generic book explicitly assigns the adequate value (which, at least, was obtained as the result of the above call to `get-event-list`). For example, our generic book about multiset contains the following constant definition:

```
(defconst *multiset*
  '((DEFUN MEMBER-BAG-EQUIV (E M)
      (DECLARE (XARGS :MEASURE (MEASURE M)))
      (COND ((EMPTY M) NIL)
            ((EQUIV E (SELECT M)) T)
            (T (MEMBER-BAG-EQUIV E (REDUCT M)))))
    (DEFTHM EQUIV-IMPLIES-IFF-MEMBER-BAG-EQUIV-1
      (IMPLIES (EQUIV E E-EQUIV)
               (IFF (MEMBER-BAG-EQUIV E M)
                    (MEMBER-BAG-EQUIV E-EQUIV M)))
      :RULE-CLASSES (:CONGRUENCE))
    (DEFUN REMOVE-ONE-BAG-EQUIV (E M)
      (DECLARE (XARGS :MEASURE (MEASURE M)))
      (COND ((EMPTY M) M)
            ((EQUIV E (SELECT M)) (REDUCT M))
            (T (INCLUDE (SELECT M)
                        (REMOVE-ONE-BAG-EQUIV E (REDUCT M))))))
    (DEFTHM EQUIV-IMPLIES-EQUAL-REMOVE-ONE-BAG-EQUIV-1
      (IMPLIES (EQUIV E E-EQUIV)
               (EQUAL (REMOVE-ONE-BAG-EQUIV E M)
                      (REMOVE-ONE-BAG-EQUIV E-EQUIV M)))
      :RULE-CLASSES (:CONGRUENCE))
      ...
  )
```

4 Two instantiation examples

We show in this section two examples of concrete implementations of finite multisets. In each case, the macro `definstance-*multiset*` is used to instantiate the events of the generic theory automatically generating the corresponding definitions and theorems for those concrete implementations.

4.1 Multisets represented as lists

Suppose we want to develop a book `multiset-list.lisp` about multisets represented as lists (containing their elements with the corresponding repetitions). And suppose we want to use, for this implementation, the functions and properties of the generic theory. For that purpose, we include the generic book:

```
(include-book "generic-multiset")
```

Note that this inclusion has automatically generated the definition of the macro `definstance--*multiset*`, that will be used to instantiate the events from the generic book. But before, we define the concrete counterparts of the generic functions `select`, `reduct`, `include`, `empty`, `measure`, `equiv` and `count-bag-equiv`. In this case, we use the primitive functions `car`, `cdr`, `cons`, `atom`, `acl2-count` and `equal`. Then, we only must to define the function `count-bag-equal-list` (the concrete counterpart of the generic `count-bag-equiv`) and prove a theorem analogue to `count-include` given in section 2.

```
(defthm count-include-list
  (equal (count-bag-equal-list e1 (cons e2 m))
         (if (equal e1 e2)
             (1+ (count-bag-equal-list e1 m))
             (count-bag-equal-list e1 m))))
```

Now, we instantiate all the events stored in the constant `*multiset*` simply by this macro call:

```
(definstance-*multiset*
  ((select car)
   (reduct cdr)
   (include cons)
   (empty atom)
   (measure acl2-count)
   (equiv equal)
   (count-bag-equiv count-bag-equal-list))
  "-list")
```

At this moment, new instantiated definitions and theorems are available in the ACL2 logical world. We show some examples using these new defined operations with this representation of multisets:

```
ACL2 !>(defconst *ma-list* '(a a b b c))
ACL2 !>(defconst *mb-list* '(b c c b c))
ACL2 !>(union-bag-equiv-list *ma-list* *mb-list*)
(A A B B C B C C B C)
ACL2 !>(inter-bag-equiv-list *ma-list* *mb-list*)
(B B C)
ACL2 !>(unimin-bag-equiv-list *ma-list* *mb-list*)
(A A B B C C C)
ACL2 !>(diff-bag-equiv-list *ma-list* *mb-list*)
(A A)
```

We also can check that the logical world includes instantiated theorems. For example, the following is a theorem about associativity of the union operation, for this concrete implementation:

```
ACL2 !>:pe union-bag-equiv-associative-list
          2:x(DEFINSTANCE-*MULTISET* (# # # ...)
                                     "-list")
             \
>             (DEFTHM UNION-BAG-EQUIV-ASSOCIATIVE-LIST
              (EQUAL-BAG-EQUIV-LIST
                   (UNION-BAG-EQUIV-LIST M1 (UNION-BAG-EQUIV-LIST M2 M3))
                   (UNION-BAG-EQUIV-LIST (UNION-BAG-EQUIV-LIST M1 M2) M3))
              :HINTS (("Goal" :BY ....)))
```

This instantiated definitions and theorems can now be used in this new book to obtain new definitions and theorems for this concrete implementation of multisets.

4.2 Multiset represented as association lists

An alternative representation of finite multisets is by means of association lists. Now, every multiset is an association list in which the components are pairs of the form `(`e `.` n`)`, where e is an element and n is a non negative integer indicating the number of occurrences of e in the multiset. We admit a multiset with many components associated with the same element, with an accumulative effect. Any other component will be ignored.

Thus, if we want to develop a book `multiset-assoc.lisp` about multisets represented in this way, and we also want to use the properties proved in the generic book, we proceed as in the previous subsection:

```
(include-book "generic-multiset")
```

Now we define the counterparts of of the generic functions `select`, `reduct`, `include`, `empty`, `measure`, `equiv` and `count-bag-equiv`. In this case the only primitive function used is `equal`, the concrete counterpart of `equiv`. In the following, we briefly describe the the rest of these new definitions.

To select an element from a multiset, we search the first pair `(`e `.` n`)`, with n a positive integer, and return e. To reduce a multiset we search the first pair `(`e `.` n`)`, with n a positive integer, and if n is greater than 1, we replace it with `(`e `.` $n-1$`)`, or we delete it. The function implementing these processes are named `select-assoc` and `reduct-assoc`, respectively. To include an element e in a multiset, we search the first pair `(`e `.` n`)`, with n a positive integer, and if this pair exists, we replace it with `(`e `.` $n+1$`)`. Otherwise, we add the pair `(`e `. 1)` at the end of the multiset. This process is implemented by the function `include-assoc`. An empty multiset is any atom or any list without pairs of the form `(`e `.` n`)`, with n a positive integer (tested by `empty-assoc`).

The measure associated with multisets is the number of its elements. This function always returns an ordinal and the measure of `(reduct-assoc m)` is less than the measure of `m`, whenever `m` is not empty. This measure is implemented by the function `measure-assoc`. Finally, the number of occurrences of an element e in a multiset `m` is the sum of every positive integer `n` such as the pair `(`e `.` n`)` occurs in `m`. This function, `count-bag-equal-assoc`, is proved to hold the corresponding theorem about its relation with the constructor function.

Now, we can instantiate the multiset generic theory, with this macro call to `definstance--*multiset*`:

```
(definstance-*multiset*
  ((select select-assoc)
   (reduct reduct-assoc)
   (include include-assoc)
   (empty empty-assoc)
```

```
   (measure measure-assoc)
   (equiv equal)
   (count-bag-equiv count-bag-equal-assoc))
  "-assoc")
```

The following are some examples using the new operations automatically defined:

```
> (defconst *ma-assoc* '((c . 1) (b . 2) (a . 2)))
> (defconst *mb-assoc* '((c . 3) (b . 2)))
> (union-bag-equiv-assoc *ma-assoc* *mb-assoc*)
((C . 4) (B . 4) (A . 2))
> (inter-bag-equiv-assoc *ma-assoc* *mb-assoc*)
((B . 2) (C . 1))
> (unimin-bag-equiv-assoc *ma-assoc* *mb-assoc*)
((C . 3) (A . 2) (B . 2))
> (diff-bag-equiv-assoc *ma-assoc* *mb-assoc*)
((A . 2))
```

And, as with the previous list implementation, we can check that the logical world includes instantiated theorems, for this implementation based on association list. For example:

```
ACL2 !>:pe union-bag-equiv-associative-assoc
         11:x(DEFINSTANCE-*MULTISET* (# # # ...)
                                     "-assoc")
             \
>             (DEFTHM UNION-BAG-EQUIV-ASSOCIATIVE-ASSOC
                (EQUAL-BAG-EQUIV-ASSOC
                  (UNION-BAG-EQUIV-ASSOC M1 (UNION-BAG-EQUIV-ASSOC M2 M3))
                  (UNION-BAG-EQUIV-ASSOC (UNION-BAG-EQUIV-ASSOC M1 M2) M3))
                :HINTS (("Goal" :BY ....)))
```

5 Conclusions and further work

We have shown the development of a tool to instantiate generic theories. As a case study, we have developed a generic theory about multisets and we have showed how our tool provides a convenient way to instantiate its generic events. This tool can certainly be improved, but we think that basically our approach can be used to instantiate generic events from other basic data types.

Here are some improvements that can be interesting:

- To extend the kind of events the instantiation tool can deal with (only `defun` and `defthm` events are considered at this moment).
- A better tool to build the list of instantiable events and store this list in a constant. At this moment, the author of a generic book has to evaluate `get-event-lst` and copy the result to assign this value to a constant.
- The functional instantiation hint given to the instantiated theorems has a functional substitution that probably binds more function symbols than needed. It would be necessary to know the dependencies among the events to optimize the construction of the instance hint.

As for the generic multiset theory presented, our development is not exhaustive, and it can be extended in several directions. For example, it is possible to incorporate new operations on multisets, proving their properties, or extend the set of properties of the defined functions, in the

line of some available multiset libraries for other reasoning systems: PVS [1], Isabelle [5], Coq [2] or Nuprl [3]. Another interesting work would be to prove the theorem shown in [7] in a generic frame.

Finally, it is also our intention to analyze the possible applications to the formalization of membrane computing (P–systems [6]) due to the relation existing between this field and multiset processing (originally the "Workshop on membrane computing" was called "Workshop on multiset processing").

References

[1] Rick Butler and David Griffioen. Theory of bags and finite bags. `http://shemesh.larc.nasa.gov/fm/ftp/larc/pvs-library/bags.dmp.gz`, 1996.

[2] G. Huet. Multiset. `http://pauillac.inria.fr/coq/library/sets/multiset.html`, 1995.

[3] Paul Jackson. Mset Theory. `http://www.cs.cornell.edu/info/people/jackson/nuprl/theories/mset.html`, 1995.

[4] J Strother Moore. Finite Set Theory in ACL2, 2001. Theorem Proving for Higher Order Logics – TPHOLs '01, R. J. Boulton and P. B. Jackson (eds.), Springer-Verlag LNCS 2152, pp 313-328, Sep 2001.

[5] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. The Supplemental Isabelle/HOL Library: Multisets, pp 44-59, 2001.

[6] Gheorghe Paun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.

[7] J.L. Ruiz-Reina, J.A. Alonso, M.J. Hidalgo, and F.J. Martín. Multiset relations: a tool for proving termination, 2000. In ACL2 Workshop 2000.

Appendix: proof strategies for multisets

In this appendix we describe how we formalize the following theorem in ACL2:

$$Hyp \rightarrow \forall e\ (\mu(e, M_1) \leq \mu(e, M_2)) \Longrightarrow Hyp \rightarrow M_1 \subseteq M_2$$

and how we define a function to compute an appropriate hint to use this theorem by functional instantiation. We have described in subsection 2.3 how this computed hint can be used to implement a proof strategy to deal with the sub-multiset relation. As we said in subsection 2.4, a similar proof strategy is codified for dealing with equality of multisets.

Due to the lack of quantification, we use `encapsulate` to state the conditions of the above theorem. We assume in an `encapsulate` the existence of two multisets such that, when some hypotheses are verified, the number of occurrences of any element in the first one is less or equal than the number of its occurrences in the second one:

```
(encapsulate
 (((sub-bag-strategy-m1) => *)
  ((sub-bag-strategy-m2) => *)
  ((sub-bag-strategy-hyp) => *))

 (local (defun sub-bag-strategy-m1 () nil))

 (local (defun sub-bag-strategy-m2 () nil))

 (local (defun sub-bag-strategy-hyp () t))

 (defthm sub-bag-equiv-strategy-constraint
   (implies (sub-bag-strategy-hyp)
            (<= (count-bag-equiv strategy-e (sub-bag-strategy-m1))
                (count-bag-equiv strategy-e (sub-bag-strategy-m2))))))
```

Note that both multisets and the hypothesis can be represented as generic constant functions. From the properties assumed in this `encapsulate`, we prove the sub-multiset relation between these multisets, whenever the hypotheses are verified:

```
(defthm sub-bag-equiv-strategy
  (implies (sub-bag-strategy-hyp)
    (sub-bag-equiv (sub-bag-strategy-m1) (sub-bag-strategy-m2))))
```

See the details of the proof of this theorem in the book `generic-multiset.lisp`.

Note that given any hypotheses Hyp and multisets M_1 and M_2, we could prove $Hyp \rightarrow M_1 \subseteq M_2$, by functional instantiation of the above theorem. Replacing `(sub-bag-strategy-hyp)`, `(sub--bag-strategy-m1)` and `(sub-bag-strategy-m2)`, by Hyp, M_1 and M_2, respectively, will bring the prover to proof attempt of $Hyp \rightarrow \forall e\ (\mu(e, M_1) \leq \mu(e, M_2))$.

To make convenient use of this strategy, we have defined a computed hint that, by the analysis of a theorem, builds an adequate functional instantiation. This hint is computed by the function `defstrategy-sub-bag-hint`, described in the following.

To obtain the adequate values of Hyp, M_1 and M_2 from the theorem we want to prove, we use the macro `case-match` to check that the theorem is about the sub-multiset relation and to extract those three values. This is done by the following function (note that this function is also used to implement the analogue strategy for equality):

```
(defun components-equal-sub-bag (form)
  (declare (xargs :mode :program))
  (case-match form
              ((’IMPLIES form-hyp
                         (’EQUAL-BAG-EQUIV form-m1 form-m2))
               (mv form-hyp form-m1 form-m2))
              ((’EQUAL-BAG-EQUIV form-m1 form-m2)
               (mv t form-m1 form-m2))
              ((’IMPLIES form-hyp
                         (’SUB-BAG-EQUIV form-m1 form-m2))
               (mv form-hyp form-m1 form-m2))
              ((’SUB-BAG-EQUIV form-m1 form-m2)
               (mv t form-m1 form-m2))
              (& (mv nil nil nil))))
```

Finally, the following is the definition the function `defstrategy-sub-bag-hint`:

```
(defun defstrategy-sub-bag-hint (id clause world)
  (declare (xargs :mode :program)
           (ignore id world))
  (mv-let (form-hyp form-m1 form-m2)
          (components-equal-sub-bag (first clause))
          (if form-hyp
              `(:use (:functional-instance
                      sub-bag-equiv-strategy
                      (sub-bag-strategy-m1 (lambda () ,form-m1))
                      (sub-bag-strategy-m2 (lambda () ,form-m2))
                      (sub-bag-strategy-hyp (lambda () ,form-hyp))))
              nil)))
```

This codified strategy can be improved in several ways. For example, instead of considering only the hint applied to the top goal, we could consider to extend the hint to other subgoals of the proof attempt. In this case, the function `components-equal-sub-bag` would have to deal with the formulas in clausal form.