Finite Set Theory based on Fully Ordered Lists

Jared Davis
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
jared@cs.utexas.edu

Abstract

We present a new finite set theory implementation for ACL2 wherein sets are implemented as fully ordered lists. This order unifies the notions of set equality and element equality by creating a unique representation for each set, which in turn enables nested sets to be trivially supported and eliminates the need for congruence rules.

We demonstrate that ordered sets can be reasoned about in the traditional style of membership arguments. Using this technique, we prove the classic properties of set operations in a natural and effortless manner. We then use the exciting new MBE feature of ACL2 to provide linear-time implementations of all basic set operations. These optimizations are made "behind the scenes" and do not adversely impact reasoning ability.

We finally develop a framework for reasoning about quantification over set elements. We also begin to provide common higher-order patterns from functional programming. The net result is an efficient library that is easy to use and reason about.

1 Introduction

Why reimplement set theory in ACL2? After all, the standard implementation [7] is already well distributed, documented, and quite good. Its congruence-oriented reasoning is extensible to a user's functions, and its `defx` macro provides access to more advanced proof strategies.

Execution efficiency is one consideration. The standard implementation represents sets as unordered lists. While some operations can be implemented efficiently this way, important functions such as equality and subset testing are quadratic. In contrast, all basic set operations can be implemented in linear time using ordered lists. This suggests an ordered implementation may work well for some large problems.

A more basic "objection" to unordered lists is that a single set may have many representations. For example, the set {1, 2} could be represented either by the list `(1 2)` or by `(2 1)`. Because of this, an explicit notion of set equality is needed, as are conventions for when set equality should be applied rather than `equal`. Further complicating the situation, set membership, subset, and set equality are now mutually recursive. In contrast, ordered lists provide a unique representation for each set, so only the standard definition of equality is needed. This advantage also holds over representations such as trees.

There are drawbacks. Unordered lists can ignore duplication to provide constant-time insertion, a fundamental operation. Full ordering also brings new challenges to reasoning. Moore remarks:

> "I found this approach to complicate set construction to a degree out of proportion to its merits. In particular, functions like `union` and `intersection`, which are quite easy to reason about in the list world (where order and duplication matter but are simply ignored), become quite difficult to reason about in the set world, where most of the attention is paid to the sorting of the output with respect to the total ordering." [7]

Beneath these words lies a challenging problem: how can the realities of an ordered implementation be abstracted away into the traditional view of sets as unordered collections? Success here is crucial: reasoning about union and intersection should not be based on the underlying implementation, but rather through an abstract, membership-based approach. Much of our initial effort is focused on achieving this abstraction.

We begin by introducing the core set operations and the strategies used to reason about them (Section 2). We partition our work into three "levels" which we name the *primitive*, *membership*, and *top* levels for easy reference. Later levels build off the work accomplished in the previous levels. The primitive level defines the set recognizer and focuses on wrapping the basic list operations (`car`, `cdr`, ...) in new "set primitives" which behave more predictably when applied to non-set objects. The membership level then introduces set membership and subset, and works towards abstracting away the set order. In its place, membership-based methods are developed for working with sets, including "pick-a-point" proofs of subset and double-containment proofs of equality. The top level builds from this by introducing the remaining set operations (e.g., union, intersection, difference) and developing reasoning strategies for these operations using the membership-based approach.

Our attention then turns to execution efficiency (Section 3). We introduce MBE, a new feature of ACL2, and show how it can be used to provide efficient versions of the set operations while preserving the reasoning strategies developed thus far. We briefly compare the performance of our implementation with the existing sets library, and add a sort so that we may quickly create sets from lists.

We then take another look at reasoning, interested now in how the library can be made more easily extensible to new problem domains (Section 4). We create instantiable "templates" for quantifying predicates over sets, e.g., $\forall a \in X, P(a)$ or $\exists a \in X, P(a)$. We discuss the relationship between these arguments and the pick-a-point strategy created in the membership level for reasoning about subsets, and discover that quantification is a generalization of this strategy. We then begin work on providing some common higher-order functional programming paradigms, such as *map* and *filter*, and find our work with quantification to be immediately useful in developing these strategies.

Finally, we conclude by looking at future directions for the library (Section 5). We consider our success with quantification-based reasoning and the benefit derived from automating functional instantiation. We explore the consequences of changing the set order, and take a cursory look into applying our techniques to other data structures.

2 The Basic Set Operations

We now turn our attention to defining the basic set operations. Our first task is to define the representation of sets. A total order on ACL2 objects, `<<`, was recently introduced and is available as a standard ACL2 book, `misc/total-order`. [8] We adopt this order verbatim, but the particular order is unimportant, and we will later consider the possibility of using alternate orders (Section 5).

We initially implemented the set recognizer using two functions: `unique` ensured that the list contained no duplicates, and `ordered` ensured that all of the elements were in order. Sets were then those lists which satisfied both uniqueness and order. This definition can be simplified, as the asymmetry of `<<` means that order implies uniqueness. In the end, `setp` is this:

```
(defun setp (X)
  (if (atom X)
      (null X)
    (or (null (cdr X))
        (and (consp (cdr X))
             (<< (car X) (cadr X))
             (setp (cdr X))))))
```

2.1 The Primitive Level

ACL2 functions are total, so set operations must be defined not only for sets, but also for non-set objects. As in [7], we adopt the *non-set convention*: if a function is passed a non-set object where a set is expected, we treat the object as the empty set. As a result of this decision, many theorems need not have extra hypotheses. For example, `(subset X X)` is now a global truth irrespective of the type of `X`. This has several useful consequences: rewrite rules become more widely applicable, and apply quickly because fewer hypotheses must be relieved. [6]

It is tempting to implement the remainder of the sets package directly using the usual list primitives: `car`, `cdr`, `cons`, and `endp`. Indeed, this was our initial approach. Using these functions, proofs about "simple" operations (element insertion/deletion, subset testing) were possible, but proofs about "complicated" operations (union, intersection, difference) became unmanageable. We came to believe these problems were not entirely artifacts of the set order, but at least partially because the list primitives do not respect the non-set convention. For example, `(1 1)` contains duplicate elements and hence is not a set, yet the list primitives do not treat it as empty:

- `(car '(1 1)) = 1`, should be `nil`.
- `(cdr '(1 1)) = (1)`, should be `nil`.
- `(endp '(1 1)) = nil`, should be `t`.
- `(cons 1 '(1 1)) = (1 1 1)`, should be `(1)`.

The list primitives were designed to operate on regular lists — not ordered sets. As a result, they are poor candidates on which to directly base a set library. Instead, we implement analogous *set primitives* that respect to the non-set convention:

- `sfix` – identity, but `nil` on non-sets.
- `head` – `car`, but `nil` on non-sets.
- `tail` – `cdr`, but `nil` on non-sets.
- `empty` – `endp`, but `t` on non-sets.
- `insert` – `cons`, but preserves the set order and treats non-sets as `nil`.

The set primitives form a primitive level of abstraction, one step removed from the list primitives. Definitions for these functions are provided in Figure 1.

```
(defun empty (X)
  (or (null X)
      (not (setp X))))

(defun sfix (X)
  (if (empty X) nil X))

(defun head (X)
  (car (sfix X)))

(defun tail (X)
  (cdr (sfix X)))

(defun insert (a X)
  (cond ((empty X) (list a))
        ((equal (head X) a) X)
        ((<< a (head X)) (cons a X))
        (t (cons (head X)
                 (insert a (tail X))))))
```

Figure 1: Primitive Set Functions

The non-set convention has interesting consequences. Earlier we asserted ordering gives a unique representation to every set, and this is true in that if `(setp X)` ∧ `(setp Y)`, then either `X = Y` or there is some element in one but not the other. Yet, since the set functions treat any non-set as empty, in a sense there are multiple representations for the empty set. (Of course, only one of these representations satisfies `setp`). To deal with these "improper" empty sets, we show that emptiness implies certain equalities. In particular, if `(empty X)` ∧ `(empty Y)`, then:

```
(head X) = (head Y)
(tail X) = (tail Y)
(insert a X) = (insert a Y)
(sfix X) = (sfix Y)
```

Eventually, we disable the definitions of `setp`, `sfix`, `empty`, `head`, `tail`, and `insert` to prevent the list primitives from occurring in higher level proofs about sets.

2.2 The Membership Level

The primitive level supports the non-sets convention well, but is a poor platform for set reasoning: the only tools it provides are induction over `insert`'s definition and some rules about the set order. In contrast, traditional set theory proofs are largely based on membership and view sets as unordered collections. The membership level builds from the primitive level, supplanting order-based reasoning with a membership-based approach. To begin, set membership and subset are introduced:

```
(defun in (a X)
  (and (not (empty X))
       (or (equal a (head X))
           (in a (tail X)))))

(defun subset (X Y)
  (or (empty X)
      (and (in (head X) Y)
           (subset (tail X) Y))))
```

We now have three goals: replacing order-based reasoning with membership-based reasoning, developing the pick-a-point strategy for proving subset relations, and showing that double containment is the same as equality.

Eliminating Order-Based Reasoning. We would like to reason about sets through membership instead of using the set order. Towards this end, we first prove the following theorem, which provides us with the understanding that a set's elements are unique. Importantly, this statement is made entirely in terms of membership:

```
(defthm head-unique
  (not (in (head X) (tail X))))
```

A second key step is to provide a new induction scheme for `insert`. `Insert` is the fundamental operation through which we construct sets, and inducting over insert is a common necessity. However, `insert` is defined using the set order, so in the course of these inductions the set order will be introduced into our proofs. We avoid this by defining a new induction scheme which uses membership for its cases.

The new scheme is natural. We make no inductive assumptions when `(empty X)` or `(in a X)`: in either of these cases, `(insert a X) = X`, so the property must only be shown to hold for `X`. Otherwise, `a` will be added somewhere in the list. As a third base case, if `a` will be placed at the front of the list, then we know `(head (insert a X)) = a` and `(tail (insert a X)) = X`. Finally, if `a` is not placed in the front of the list, we inductively assume the property holds for `(insert a (tail X))`.

No longer needing the set order for inductive proofs about `insert`, we are one step closer to reasoning exclusively through membership.

Pick-a-Point Proofs. In traditional mathematics, the pick-a-point method is typically used to prove subset relationships. The idea is to show that $\forall a : a \in X \Rightarrow a \in Y$ to conclude $X \subseteq Y$. So, to prove a subset relationship, simply pick an arbitrary point and show this membership relationship holds.

Because of the universal quantifier, this idea cannot be directly stated as an ACL2 rewrite rule. However, we can set up an "encapsulate" event and use it to accomplishes a similar reduction. Suppose `hyps`, `sub`, and `super` are some functions which happen to satisfy the following constraint:

```
(defthm membership-constraint
  (implies (and (hyps)
                (in a (sub)))
           (in a (super))))
```

Then the following is a theorem:

```
(defthm subset-by-membership
  (implies (hyps)
           (subset (sub) (super))))
```

The proof involves the use of a witness function. We create a new function, `(subset-witness X Y)`, that searches for an element which satisfies `(in a X)` and `(not (in a Y))`. An easy lemma is that if `subset-witness` fails to find such an element, then `X` is a subset of `Y`.

Now, suppose towards contradiction that `(subset-witness (sub) (super))` finds an element, `a`. Then, by the definition of `subset-witness`, `(in a (sub))` and `(not (in a (super)))`. But this directly contradicts the membership constraint. Therefore, we know that `(subset-witness (sub) (super))` cannot find a satisfactory element, and by our lemma we are done.

Concrete subset relationships can be proven through the functional instantiation of the functions `sub`, `super`, and `hyps`. In other words, we can now conclude `(implies (hyps) (subset (sub) (super)))` merely by proving that `membership-constraint` holds for our choices of `hyps`, `sub`, and `super`.

The theorem prover will not automatically try to use functional instantiation, but can be instructed to do so through hints. Here is an example theorem of such a hint:

```
(defthm subset-union-Y
  (subset Y (union X Y))
  :hints(("Goal" :use (:functional-instance
     subset-by-membership
       (sub   (lambda () Y))
       (super (lambda () (union X Y)))
       (hyps  (lambda () t))))))
```

This hint instructs the theorem prover to functionally instantiate `subset-by-membership`, selecting `t` for `hyps`, `Y` for `sub`, and `(union X Y)` for `super`. In other words, this hint will allow us to conclude:

```
(implies t
         (subset Y (union X Y)))
```

If we can show the corresponding membership constraint, namely:

```
(implies (and t (in a Y))
         (in a (union X Y)))
```

Fortunately, this is an easy proof for ACL2, and hence the conclusion of `(implies t (subset Y (union X Y)))` is accepted. A simple reduction allows us to conclude `(subset Y (union X Y))`, finishing the proof.

Automating Pick-a-Point Proofs. Ideally, a sets library should be capable of deep reasoning about sets without user interaction. Although it would not be difficult for a user to explicitly invoke a pick-a-point proof of a subset relation, we would prefer a more automatic solution. Towards this end, we have developed computed hints[1] to automatically apply this strategy when it seems applicable.

Rather than derail our discussion of set reasoning, we relegate the details of this process to Appendix B. The general idea is if (a) our goal is to prove a subset relationship, and (b) all other attempts at simplifying the conjecture have been exhausted, then an appropriate functional instantiation hint is suggested. The substitutions to make are extracted from the conjecture itself, i.e., if the conjecture is `(H` $\Rightarrow$ `(subset X Y))`, then we instantiate `hyps` with `H`, `sub` with `X`,

[1] Computed hints [3] allow hints to be automatically generated and suggested, rather than having a user write them explicitly.

and super with Y. As with subset-union-Y above, this reduces the subset argument to a membership argument.

Double Containment Proofs. Attention now turns to proving double containment is equality, and subset-by-membership is immediately useful. Suppose two sets are subsets of one another, i.e., $X \subseteq Y$ and $Y \subseteq X$. First, we show (head X) = (head Y). Next we show (in a (tail X)) implies (in a (tail Y)). Subset-by-membership is then called upon twice to show (tail X) is a subset of (tail Y), and vice versa. We induct on a “double-tail” scheme, so the inductive hypothesis asserts if the tails are mutual subsets they are equal. Now (head X) = (head Y) and (tail X) = (tail Y), so we conclude X = Y. The resulting rewrite rule is:

```
(defthm double-containment
  (implies (and (setp X)
                (setp Y))
           (equal (equal X Y)
                  (and (subset X Y)
                       (subset Y X)))))
```

Because this theorem is a rewrite rule, equalities between sets will be automatically rewritten into subset arguments. As an aside, we had been able to prove that double containment was equality even before implementing the pick-a-point strategy. Yet, the proof required first introducing the delete function and several theorems pertaining to it, then inducting by deleting (head X) from both sides of the proposed equality. In contrast, the above saves a significant amount of work as the theorems about delete (which had themselves required induction arguments) can now be proven automatically by appealing to double-containment.

The following rewrite strategy has been created: set equalities are reduced to containment arguments, and containment arguments are reduced to membership arguments. The beauty of this strategy is that relationships about in (which are typically easy to prove) are now sufficient to conclude subset and equality relations between complicated expressions, which might otherwise require hard inductions. This method bears a close resemblance to traditional set theory proofs and is a natural way to work with the set functions.

Finally, all remaining theorems mentioning the set order are disabled. Membership alone will now be used to prove theorems.

2.3 The Top Level

While the membership level provides a solid basis for set reasoning, the library is far from complete: we have only insertion, membership, subset testing, and the set primitives at this point. Using the membership level as a foundation, the top level introduces the remaining set theory functions: **delete**, **union**, **intersect**, **difference**, and **cardinality**. These functions are presented in Figure 2.

Using membership to prove subsets and equalities is a powerful approach. Theorems about these functions, so hard to prove in our early attempts, are now automatic. At this level, “The Method” is generally:

- Introduce a function, prove it produces sets and prove its basic membership properties
- Call upon the pick-a-point method to prove interesting equalities and subset relationships.

```
(defun delete (a X)
  (cond ((empty X) nil)
        ((equal a (head X)) (tail X))
        (t (insert (head X)
                   (delete a (tail X))))))

(defun union (X Y)
  (if (empty X)
      (sfix Y)
    (insert (head X) (union (tail X) Y))))

(defun intersect (X Y)
  (cond ((empty X) (sfix X))
        ((in (head X) Y)
         (insert (head X)
                 (intersect (tail X) Y)))
        (t (intersect (tail X) Y))))

(defun difference (X Y)
  (if (empty X)
      (sfix X)
    (if (in (head X) Y)
        (difference (tail X) Y)
      (insert (head X)
              (difference (tail X) Y)))))

 (defun cardinality (X)
   (if (empty X)
       0
     (1+ (cardinality (tail X)))))
```

Figure 2: Top Level Definitions

Many theorems are proven this way. Included among them are the associativity of union and intersection, the symmetry of union and intersection, distributivity of unions over intersections, DeMorgan laws for distributing differences, and so forth. Other theorems are not based entirely on the pick-a-point method, but are still carried out without mentioning the set order. (In particular, cardinality properties are demonstrated using our membership-based induction over insert.) Many selected theorems are listed in Appendix A.

Users of the library would typically base their work on the top level, using this same style of reasoning.

3 Execution Efficiency

When sets are implemented as ordered lists, all basic set operations can be implemented with linear complexity. However, the functions presented in Section 2 do little to realize this possibility. Here efficiency and reasoning conflict: we would like to take advantage of the set order to implement these functions more efficiently, yet the given definitions of `subset`, `union`, `intersection`, and `difference` are nice for reasoning precisely because they are described in terms of membership and not the set order. Fortunately, there is a nice solution to this problem using guards and MBE.

Guards and MBE. Although ACL2 functions are total, guards allow us to state an "intended domain" for functions. [5] [1] [4] Guards are often presented as a tool for ensuring the compatibility of ACL2 code with Common Lisp, but they can also be used as runtime assertions when *guard checking* is enabled, or as static checks through the process of *guard verification* (using the theorem prover to show whenever a function is called, its arguments satisfy its domain). We add guards to our basic set operations as listed in Figure 3.

Function	Guard
`(setp X)`	`t`
`(empty X)`	`(setp X)`
`(sfix X)`	`(setp X)`
`(head X)`	`(setp X)`
`(tail X)`	`(setp X)`
`(insert a X)`	`(setp X)`
`(in a X)`	`(setp X)`
`(subset X Y)`	`(setp X)` ∧ `(setp Y)`
`(delete a X)`	`(setp X)`
`(union X Y)`	`(setp X)` ∧ `(setp Y)`
`(intersect X Y)`	`(setp X)` ∧ `(setp Y)`
`(difference X Y)`	`(setp X)` ∧ `(setp Y)`
`(cardinality X)`	`(setp X)`

Figure 3: Guards for Set Functions

Introduced in ACL2 2.8, the MBE macro allows two separate definitions — one *logical*, and one *executable* — to be provided for a single function. When reasoning about the function, the logical definition is used. However, when executing the function on arguments that satisfy the function's guards, the executable definition is used instead. Note that for this substitution to be sound, both definitions must be proven to produce the same answer for any inputs satisfying the guards (hence MBE stands for "must be equal").

Achieving Efficiency. As a first step, the combination of MBE and guards can be used to provide faster versions of the set primitives. In Section 2, all primitives called `setp` (sometimes via `sfix`) to ensure their argument was a set. This is a terrible waste, since `setp` must examine the entire set. With guards to ensure the primitives operate only on sets, these `setp` calls are no longer necessary. We therefore make the following MBE substitutions: `(empty X)` is replaced by `(null X)`, `(sfix X)` by `X`, `(head X)` by `(car X)`, and `(tail X)` by `(cdr X)`. `Insert` is not changed; it is already linear now that the other primitives have been made efficient.

Our attention then turns to the other operations. `In` is unchanged since it is already linear. An alternative `in` could use the set order to stop early, for example `(in 1 '(2 3 4))` could terminate immediately because `(<< 1 2)`. But there are also cases when this would be slower, as each iteration would incur the overhead of a call to `<<`. Because of this, the given definition is retained.

Linear versions of `subset`, `union`, `intersect`, and `difference`, are shown in Figure 4. These are visibly more complicated than their original counterparts. Except for subset, the proofs of equivalence can be carried out by merely showing these functions produce sets and have the characteristic membership property, then appealing to double containment to finish the proof. Importantly, this approach obviates the need to directly induct against `union`, etc.

Even so, these are not trivial proofs and turn out to involve many cases: `cons` only produces sets under certain conditions, so theorems about it are complicated and weak. These proofs must be argued from "first principles" using the set order and induction, but this is not a violation of our goal of reasoning through membership: these functions are dependent

```
(defun fast-subset (X Y)
  (cond ((empty X) t)
        ((empty Y) nil)
        ((<< (head X) (head Y)) nil)
        ((equal (head X) (head Y))
         (fast-subset (tail X) (tail Y)))
        (t (fast-subset X (tail Y)))))

(defun fast-union (X Y)
  (cond ((empty X) Y)
        ((empty Y) X)
        ((equal (head X) (head Y))
         (cons (head X)
           (fast-union (tail X) (tail Y))))
        ((<< (head X) (head Y))
         (cons (head X)
           (fast-union (tail X) Y)))
        (t (cons (head Y)
             (fast-union X (tail Y))))))

(defun fast-intersect (X Y)
  (cond ((empty X) nil)
        ((empty Y) nil)
        ((equal (head X) (head Y))
         (cons (head X)
           (fast-intersect (tail X)
                           (tail Y))))
        ((<< (head X) (head Y))
         (fast-intersect (tail X) Y))
        (t (fast-intersect X (tail Y)))))

(defun fast-difference (X Y)
  (cond ((empty X) nil)
        ((empty Y) X)
        ((equal (head X) (head Y))
         (fast-difference (tail X)
                          (tail Y)))
        ((<< (head X) (head Y))
         (cons (head X)
           (fast-difference (tail X) Y)))
        (t (fast-difference X (tail Y)))))
```

Figure 4: Linear Time Implementations

on the implementation, and the set order is the only reason they work.

No Compromises. This solution sacrifices neither execution efficiency nor reasoning ability. All set operations are now constant or linear time, yet their logical definitions are simply the Lisp reflections of their mathematical meanings, seemingly unconcerned with implementation details.

This approach is particularly appealing for the freedom it provides the author. When designing the library throughout Section 2, efficiency was not considered (our most primitive functions had to examine the entire set), and we focused solely on developing simple, straightforward models and creating proof strategies.

Only then, after the theory was complete, did efficiency become a consideration. Efficient but complicated models of these functions were developed. No theory was developed about the complicated models beyond showing that they faithfully implemented the simple models. The power of the established theory was preserved, while the efficiency of the complicated models was gained.

Without MBE, even a convenience like the non-set convention would inflict a large efficiency penalty. If efficiency is sacrificed, the library may be too slow to be practically useful. If efficiency is not sacrificed, theorems will have extra hypotheses, proofs will be larger, and more effort will be required on the part of the library's designer and its users.

Efficiency Analysis. Set theory is so widely applicable and the library is sufficiently general that benchmarking its performance is difficult. There is no concept of representative input data. There are also at least five Lisp implementations on which ACL2 can be run, each supporting various operating systems and hardware, each with different performance characteristics.

Nevertheless, it would be nice to have some rudimentary test data to demonstrate the efficiency of these operations. Towards this end, we have put together a small set of test programs where sets are created from either random integers or random strings taken from a dictionary file. We timed the test programs using GCL on Linux machines, but make no claim that our tests were thorough or representative of the results which may occur in other contexts. Out of curiosity, analogous functions from the standard sets books were also timed for comparison purposes.[2]

The results were not surprising. The standard sets library significantly outperformed ordered sets for insertion, but ordered sets were significantly faster for intersections and differences. The performance difference in each case can be made arbitrarily large by choosing large enough sets. This is exactly what should be expected: the standard library can ignore

[2]The standard sets books do not have verified guards by default, so in each case we artificially verified their guards using `skip-proofs`. This should ensure comparisons are not being made between compiled and interpreted performance.

duplication and just use `cons` for a constant-time insert, whereas the ordered library must make a linear scan of the set. In the cases of set difference and intersection, the ordered sets library can use a linear pass where the standard library uses a quadratic algorithm.

We were not able to demonstrate that either library was consistently faster than the other at performing unions.[3] We did not test subset or set equality, but would expect the ordered library to significantly outperform the standard library in "true" cases on these functions.

Adding a Sort. Unordered sets are significantly faster at repeated element insertions than ordered sets. Unordered inserts take constant time, so inserting n elements is a linear operation; ordered inserts are each linear, so inserting n elements is quadratic.

Generating sets is a fundamental operation, so a more efficient method for building large sets is desirable. Towards this purpose, a simple merge sort was implemented, reducing the time needed for n inserts from n^2 to $n\ log_2\ n$. This is still not as good as the linear performance of unordered sets, and is the inescapable price paid for full ordering.

The sort itself is easy to write using the already efficient `union` operation to perform the merge. As with the other basic set operations, we use MBE to combine easy reasoning with efficiency in execution: `mergesort` is logically viewed simply as repeated insertions.

4 Instantiable Extensions

At this point we have covered the core of the sets library. Though efficient and relatively straightforward to reason about, this core is limited in its capabilities. Having more functionality available ahead of time may make modeling new problems easier, and although it is certainly impossible to foresee and cater to every need, we suspect we can provide at least a few widely applicable extensions.

Two extensions seem to be particularly good candidates. Having recently seen the benefits derived from our `subset-by-membership` strategy, it seems desirable to provide some support for quantification over set elements. Furthermore, the time honored patterns of functional programming are probably also good candidates with which to extend the library.

4.1 Quantification

We have found quantification over set elements to be quite useful, and would like to be able to support statements of the form "$\forall a \in X, P(a)$" and "$\exists a \in X, P(a)$" for some arbitrary predicate P.

To support this generality, we introduce an optional extension to the library. Here, we create a fully instantiable generic theory (as in [2]) and provide macros to create concrete instances of this theory. Our macros are complex enough to support predicates with any number of arguments, and also support the use of custom guards on those arguments. As some examples, simple predicates such as `integerp` or `stringp` are useful for defining typed sets. More complicated predicates allow us to express notions such as "sets of integers less than b". Note that `(subset X Y)` itself is nothing more than $\forall a \in X, P(a)$ where $P(a) =$ `(in a Y)`.

In the end, given a predicate, `(P a ...)`, where `...` is understood to represent 0 or more extra arguments, the user can invoke a single macro to create the following functions and an associated rewriting strategy:

- `(all<P> X ...)`, returns `t` if $\forall a \in X$, `(P a ...)`, or `nil` otherwise.
- `(exists<P> X ...)`, returns `t` if $\exists a \in X$, `(P a ...)`, or `nil` otherwise.
- `(find<P> X ...)`, returns an element $a \in X$ such that `(P a ...)`, or `nil` if no such element exists.

We also create quantifying functions for "not P" (named `all<not-P>`, `exists<not-P>`, ...), as well as "list" versions of each function (named `all-list<P>`, `all-list<not-P>`, ...) which are useful for reasoning through any calls to `mergesort` we may encounter. Furthermore, the macro also sets up an initial rewriting strategy.

Most of this strategy is relatively straightforward and is summarized in Appendix A, but we also set the stage for deeper reasoning about these functions. As we noticed before, `all<P>` is quite like `subset`, and in fact `(subset X Y)` is exactly `(all<in> X Y)`. Since the pick-a-point strategy was quite successful at proving theorems about subset, we suspect analogous strategies for handling `all-P` might be similarly successful here.

We generalize the pick a point strategy by doing away with `(super)` and simply using `(predicate`

[3]The results are dependent on the inputs. Both algorithms are linear, but the standard sets library simply `cons`es the elements from the first list onto the second, requiring n `cons`es and recursion where n is the length of the first list. In contrast, the ordered sets library must compare set orders, walking down both lists simultaneously.

`(sub))` instead of `(subset (sub) (super))` as the conclusion for our membership constraint. Where we would have previously substituted `X` for `(sub)` and `Y` for `(super)`, we now continue to substitute `X` for `(sub)`, but instead substitute `(lambda (x) (in x Y))` for `(predicate x)`. Hence, we can still perform pick a point proofs of subsets, but can also consider other substitutions for `predicate`.

As before, we set up computed hints to automate these strategies for each predicate that the user quantifies, and more details about how these hints are constructed are provided in Appendix B. Prior to ACL2 2.9, users were required to explicitly manage which computed hints would be active during proof attempts, but as of version 2.9, these hints can be automatically installed by the macro each time a new `all<P>` function is introduced, without requiring any action or knowledge from the user. Because of these changes, computed hints can now be a tightly integrated part of a library's overall reasoning strategy.

4.2 Higher Order Functions

We further extend the library with two common higher order patterns from functional programming: `filter` and `map`. We find that our theory of quantification helps us greatly here.

Filter. We extend the quantification macro to additionally provide a `filter<P>` function and its basic theorems when we create the quantification theory for a predicate. As an example, observe that `(filter<in> X Y)` is exactly equal to `(intersect X Y)`. We might also consider simple filtering, e.g. `filter<integerp>`. This addition is a quite painless extension of the existing macro developed to instantiate the quantification theory.

Map. We create a second, optional extension of the library which allows the user to introduce `map<F>` given some transforming function `F`. In order to provide as much generality as possible, we `F` can take any number of extra arguments. The function introduced takes the following form:

```
(defun map<F> (X ...)
  (if (empty X)
      nil
    (insert (F (head X) ...)
            (map<F> (tail X) ...))))
```

This is inefficient; we are essentially performing an insert sort. To remedy this situation, we use MBE to provide an executable definition which instead puts all of the mappings into a list, which can be sorted efficiently afterwards.

```
(defun map-list<F> (X ...)
  (if (endp X)
      nil
    (cons (F (car X) ...)
          (map-list<F> (cdr X) ...))))
```

For brevity, assume `F` takes no extra arguments. We guard `map<F>` with `(setp X)`, then work towards proving `(map<F> X) = (mergesort (map-list<F> X))`, which will be our MBE substitution. This equality is an easy proof by double containment.

To do much set reasoning about mappings, the most important property is that of membership. What can be said about `(in a (map X))`. Naturally, we would like to speak in terms of inverses. Conveniently, the quantification theory we have developed allows us to do this. In particular, we can introduce the following predicate, which returns true when `(F a) = b`, or in other words, when `a` is an inverse of `b`.

```
(defun inversep<F> (a b)
  (equal (F a) b))
```

We can then use the macro developed in our quantification extension to introduce the notion of `(exists<inversep<F>> X e)`. This allows us to ask if there is any inverse of `e` in `X`. In short, the following is a theorem:

```
(equal (in a (map<F> X))
       (exists<inversep<F>> X a))
```

We can now rapidly develop theorems about `map` using the pick a point method. Whenever we need to reason about membership in a mapped set, we simply consider the existence of inverses in the original set. Combining this with the pick a point method and all of the techniques we already have for proving membership properties, this becomes a powerful strategy that is sufficient to prove the obvious theorems relating subset, union, intersection, and difference to mapping. Many of these theorems are enumerated in Appendix A.

We conclude with an example of this strategy at work. The following proof method is discovered by ACL2 automatically with no user intervention, yet has a very natural feel.

```
(equal (map<F> (union X Y))
       (union (map<F> X) (map<F> Y)))
```

ACL2 first notices that `map` and `union` both produce sets, so a proof by double containment is employed. The first subgoal is to show that `(map<F> (union X Y))` is a subset of `(union (map<F> X) (map<F>`

`Y))`. Using a pick a point proof, we choose some element `a` in `(map<F> (union X Y))`. By our membership property, we see that `(exists<inversep<F>> (union X Y) a)`. Simple rewrites show that this is the same as `(exists<inversep<F>> X)` or `(exists<inversep<F>> Y)`. But this leads us to conclude that `(in a (map X))` or `(in a (map Y))`, and hence that the subset relationship holds, by the membership property of `union`. The second subgoal is similarly straightforward.

5 Conclusions

Set theory has been an excellent domain for learning to work with ACL2, and perhaps it would be a similarly rich pedagogical tool for interacting with other theorem provers. The functions are simple and familiar, yet we discovered many subtle challenges in trying to develop a sensible proof strategy.

In a domain as rich and general as set theory, there are few limits on how we might hope to extend the library. For the near term, we consider how we might improve the automation of functional instantiation, whether or not changing the set order might be useful, discuss the limits of instantiable theories, and leave open the question of applying our techniques to other container structures.

Lessons Learned. Hiding complexity behind levels of abstraction seems to be a crucial step in successfully implementing an ACL2 library. This is the entire idea behind the primitive and membership levels. We do not believe that proofs of the many theorems in the top level would have been nearly as easy using induction and properties of the set order, but they are simple once a membership strategy is available.

MBE has also shown itself to be an extremely useful tool. Without MBE, either efficiency would have to be sacrificed for reasoning ability (even the primitive operations would be linear in complexity to support the non-sets convention), or reasoning would be impaired by attempting to directly reason about the "fast" versions of the functions.

Functional Instantiation. A cornerstone of the library's reasoning ability is the reduction of subset problems to membership problems. Using encapsulation and functional instantiation in this way is a fairly standard trick; the new idea here is the use of computed hints to automate the process. This automation is nothing more than a very narrow instance of second-order pattern matching, where only `subset` (or other suitable triggers) are considered for instantiation, and only under certain conditions. Even this very limited match seems to find broad application in the set theory domain.

An interesting question is whether or not a more general form of second-order pattern matching could be implemented to automatically apply these types of strategies. There are many difficulties in implementing this, particularly the large number of matches and how to decide which one(s) to attempt to use. Still, it seems this could eventually become a powerful extension of functional instantiation.

Custom Set Orders. Another question, more specifically pertaining to ordered sets, is if it would be worthwhile to parameterize the set order. On one hand, this seems like a fairly easy thing to implement. The library is quite indifferent to the internal workings of the order: only the properties of irreflexivity, asymmetry, transitivity, and trichotomy are ever used in set reasoning, and the definition of `<<` is never used.

It seems like creating "custom" set orders is quite easy. For example, here is an adaptation of the existing order which places the integers first in "greatest to least" order:

```
(defun my-order (a b)
  (cond ((integerp a)
         (if (integerp b)
             (> a b)
           t))
        ((integerp b) nil)
        (t (<< a b))))
```

`My-order` still satisfies all the properties of a total order, and if given the guard `(and (integerp a) (integerp b))`, it would be easy to prove MBE equivalence to `>`. But complications arise from such a scheme: to verify the guards of the set functions, we might need to add additional guards or change our definitions. For example, we might modify `setp` to require that every element satisfy `integerp`, but this would destroy theorems such as `(setp (insert a X))` unless we modified the definition of `insert` to `ifix` its arguments, and so forth.

For now, we have elected to hold off on customizing set orders. In the meantime, it would certainly be possible (and easy) to "hard-wire" in a different order with a guard of `t`, but certainly this is not a general solution.

Instantiation's Limits. Using macros to automate functional instantiation is a difficult and poor way to emulate higher order programming.

One problem is the sheer number of events introduced with each instantiation. For example, the theory of mapping involves introducing a quantification theory for a new inversep function as well as several theorems. In total, there are well over 100 definitions and theorems introduced just to introduce a new map. At present, these events are all grouped into convenient theories which can be enabled and disabled at will, but it seems clear that in the long term, more sophisticated methods will be needed to handle this kind of load, or to reduce the number of theorems actually provided.

Another issue is the sheer complexity of writing the macros. To facilitate the size of these instantiable theories, we ended up providing a simple rewriter and using it to do most of the work of setting up functional instantiation hints and very restricted theories to prove the concrete versions of each theorem. Nevertheless, the system is still awkward and bulky. Techniques to more easily define and use generic theories (where these theories are complicated, and involve definitions, guards, theorems, computed hints, and so forth) would be welcome.

Other Containers. Mainstream programming languages often offer a myriad of "container classes." As users of ACL2, we tend to shun such complexity in favor of simple lists or association lists. Can MBE offer the same benefits to such structures in terms of interface/implementation separation, and can the automation of quantification-based arguments lend the same reasoning ability to other containers as they have to set theory? If so, perhaps these containers could become more practical for use in those ACL2 models where reasoning and efficiency are both important.

Availability. The library is released under the GNU General Public License. Full source code (and other materials) are available through the following web site:

http://www.cs.utexas.edu/users/jared/osets/

Acknowledgments. I am thankful to Sandip Ray for introducing me to witness functions, to Robert Krug for his help with computed hints and the strategy automation, to J Moore for his constructive ideas and support, to Matt Kaufmann for his improvements with handling of default hints, and to Eric Smith for his many practical suggestions regarding the library. Thanks also to Hanbing Liu, Omar El-Domeiri, and Serita Nelesen for their feedback about using the library, and their suggestions which have helped to "fill in" missing pieces in the rewriting strategy.

A Selected Theorems

The following is a list of many theorems provided by the ordered sets library. It is not comprehensive, but should give a good flavor of the rewriting strategy. For brevity, instantiable functions such as `all` are written without extra arguments/predicates, and traditional notation is freely mixed with ACL2 terms.

Set Creation

(setp (sfix X))
(setp (tail X))
(setp (insert a X))
(setp (delete a X))
(setp (union X Y))
(setp (intersect X Y))
(setp (difference X Y))
(setp (mergesort x))
(setp (filter X))
(setp (map X))

Membership

(in a (insert b X)) = (in a X) ∨ (equal a b)
(in a (delete b X)) = (in a X) ∧ ¬(equal a b)
(in a (union X Y)) = (in a X) ∨ (in a Y)
(in a (intersect X Y)) = (in a Y) ∧ (in a X)
(in a (difference X Y)) = (in a X) ∧ ¬(in a Y)
(in a (mergesort x)) ⇔ (in-list a x)
(in a (filter X)) = (P a) ∧ (in a X)
(subset X Y) ∧ (in a X) ⇒ (in a Y)
(subset X Y) ∧ ¬(in a Y) ⇒ ¬(in a X)
¬(in a a)

Non-Set Convention

(empty (sfix X)) = (empty X)
(head (sfix X)) = (head X)
(tail (sfix X)) = (tail X)
(in a (sfix X)) = (in a X)
(insert a (sfix X)) = (insert a X)
(delete a (sfix X)) = (delete a X)
(subset (sfix X) Y) = (subset X Y)
(subset X (sfix Y)) = (subset X Y)
(union (sfix X) Y) = (union X Y)
(union X (sfix Y)) = (union X Y)
(intersect (sfix X) Y) = (intersect X Y)
(intersect X (sfix Y)) = (intersect X Y)
(difference (sfix X) Y) = (difference X Y)
(difference X (sfix Y)) = (difference X Y)
(cardinality (sfix X)) = (cardinality X)
(all (sfix X)) = (all X)
(find (sfix X)) = (find X)
(filter (sfix X)) = (filter X)
(map (sfix X)) = (map X)

Insertion, Deletion

¬(empty (insert a X))
(in a X) ⇒ (equal (insert a X) (sfix X))
(insert a (insert b X)) = (insert b (insert a X))
(insert a (insert a X)) = (insert a X)
(insert a (delete a X)) = (insert a X)
¬(in a X) ⇒ (equal (delete a X) (sfix X))
(delete a (delete b X)) = (delete b (delete a X)))
(delete a (delete a X)) = (delete a X)
(delete a (insert a X)) = (delete a X)
(subset X (insert a X))
(subset (delete a X) X)

Union

(empty X) ⇒ (equal (union X Y) (sfix Y))
(empty Y) ⇒ (equal (union X Y) (sfix X))
(empty (union X Y)) = (empty X) ∧ (empty Y)
(subset X (union X Y))
(subset Y (union X Y))
(union X X) = (sfix X)
(union X Y) = (union Y X)
(union (union X Y) Z) = (union X (union Y Z))
(union X (union Y Z)) = (union Y (union X Z))
(union X (union X Z)) = (union X Z)
(union (insert a X) Y) = (insert a (union X Y))
(union X (insert a Y)) = (insert a (union X Y))

Intersect

(empty X) ⇒ (empty (intersect X Y))
(empty Y) ⇒ (empty (intersect X Y))
(subset (intersect X Y) X)
(subset (intersect X Y) Y)
(intersect X X) = (sfix X)
(intersect X Y) = (intersect Y X)
(∩ (∩ X Y) Z) = (∩ X (∩ Y Z))
(∩ X (∩ Y Z)) = (∩ Y (∩ X Z))
(intersect X (intersect X Z)) = (intersect X Z)
¬(in a Y) ⇒ (∩ (insert a X) Y) = (∩ X Y)
¬(in a X) ⇒ (∩ X (insert a Y)) = (∩ X Y)

Difference

(empty X) ⇒ (empty (difference X Y))
(empty Y) ⇒ (equal (difference X Y) (sfix X))))
(empty (difference X Y)) = (subset X Y)
(subset (difference X Y) X)

Cardinality

(integerp $|X|$)
$0 \le |X|$
($|X| = 0$) = (empty X)
$|X \cap Y| \le |X|$
$|X \cap Y| \le |Y|$
$|X \cup Y| = |X| + |Y| - |X \cap Y|$
$|X - Y| = |X| - |X \cap Y|$
$X \subseteq Y \Rightarrow |X| \le |Y|$
$X \nsubseteq Y \Rightarrow |X \cap Y| < |X|$

$$|\text{(insert a X)}| = \begin{cases} |X| & \text{if (in a X)} \\ |X|+1 & \text{otherwise} \end{cases}$$

$$|\text{(delete a X)}| = \begin{cases} |X|-1 & \text{if (in a X)} \\ |X| & \text{otherwise} \end{cases}$$

$|X| = |$(filter X)$| + |$(filter-not X)$|$
$|$(map X)$| \le |X|$

Miscellaneous

(union X (∩ Y Z)) = (∩ (union X Y) (union X Z))
(∩ X (union Y Z)) = (union (∩ X Y) (∩ X Z))
(diff X (union Y Z)) = (∩ (diff X Y) (diff X Z))
(diff X (∩ Y Z)) = (union (diff X Y) (diff X Z))

Quantification

(empty X) ⇒ (all X)
(all (sfix X)) = (all X)
(all X) ⇒ (all (tail X))
(all X) ⇒ (all (delete a X))
(all X) ∧ (in a X) ⇒ (P a)
(all X) ∧ ¬(P a) ⇒ ¬(in a X)
(all (insert a X)) = (P a) ∧ (all X)
(all (∪ X Y)) = (all X) ∧ (all Y)
(all X) ⇒ (all (intersect X Y))
(all Y) ⇒ (all (intersect X Y))
(all X) ⇒ (all (difference X Y))
(exists X) = (not (all-not X))

Filtering

(all (filter X))
(all X) ⇒ (filter X) = (sfix X)
(subset (filter X) X)

Mapping

(in a (map X)) = (∃-inversep X a)
(subset X Y) ⇒ (subset (map X) (map Y))
(map (insert a X)) = (insert (F a) (map X))
(map (delete a X)) ⊇ (delete (F a) (map X))
(map (∪ X Y)) = (∪ (map X) (map Y))
(map (∩ X Y)) ⊆ (∩ (map X) (map Y)))
(map X) − (map Y) ⊆ (map (X − Y))

B Details of Computed Hints

Much of our reasoning success lies in automatically suggesting functional instantiation hints. The details of constructing these hints are now presented. Throughout this appendix, we talk about proving “subset” using computed hints, but this process is the same for `all<P>` as well.

When to Suggest Hints. At any given point in a proof attempt, ACL2 is trying to show that some conclusion follows from some hypothesis. Our strategy is to suggest a hint only if all other attempts at simplification have failed, and only for conclusions of exactly `(subset X Y)` for some expressions `X` and `Y`.

If other attempts at simplification have not yet been exhausted, there may be rewrite rules that can prove the conjecture without falling back on a membership argument. For example, perhaps if we do not interfere, some rewrite rule will transform the conclusion to `(subset Y (union X Y))`, which can then be rewritten to `t` immediately by the rule `subset-union-Y`. In this case, suggesting a membership hint might still permit ACL2 to complete the proof, but may be less efficient and less natural.

Suggesting hints only for conclusions is more subtle. If we do encounter a conclusion of `(subset X Y)`, it certainly makes sense to suggest a hint: doing so will reduce the proof to a membership argument that may be easier to prove. A second, important aspect is that we never suggest hints to reduce a hypothesis: `(subset X Y)` is a strong hypothesis that we do not wish to weaken, and given `(subset X Y)` we can already conclude `(in a X)` $\Rightarrow$ `(in a Y)` using simple rewrite rules.

Note that we suggest hints only for conclusions which are *exactly* `(subset X Y)`. In other words, we would not suggest a membership argument for the following:

```
(implies (...)
         (foo a b (subset X Y) c))
```

Is this too restrictive? In this case, we might not even want to show that `(subset X Y)` holds. ACL2 may eventually produce a new subgoal for which we need to show `(subset X Y)`, and at that point our membership strategy can be applied. However, the truth of `(subset X Y)` might also be irrelevant to the truth of the entire conjecture. As it is not clear that a hint would be useful, we choose conservatively not to make a suggestion.

Creating Hints. Using a standard rewrite rule (with `syntaxp` hypotheses to enforce the above conditions), we identify the `subset` terms that we would like to suggest hints for. We "tag" these instances by rewriting them from `(subset X Y)` to `(subset-trigger X Y)`, a new function which is simply a synonym for `subset`.

Our hint generation function is allowed to examine the current *clause* ACL2 is working on. Clauses are disjunctions of terms. For example, the clause `(not a)` $\vee$ `(not b)` $\vee$ `c` represents the implication `(implies (and a b) c)`. We search the clause for an instance of `subset-trigger`, creating a hint if we find one.

The `subs` and `super` can be extracted easily from the `subset-trigger` term. To create `hyps`, we remove the `subset-trigger` term from the clause and then combine the remaining disjuncts by `and`'ing their negations. For example, given the clause `(not (empty (difference X Y)))` $\vee$ `(subset-trigger X Y)`, we first remove `(subset-trigger X Y)`, then negate the remaining disjunct to produce the hypothesis `(empty (difference X Y))`.

Finally, we build a `:functional-instance` hint, instantiating the theorem `subset-by-membership` with our newly computed choices of `sub`, `super`, and `hyps`.

User Notification. When hints are suggested, we output a brief message to the user. We tell the user our heuristics suggest using a pick-a-point style argument, and that we will therefore suggest a functional instantiation hint. The message includes instructions for disabling the strategy, in case it is not what the user has in mind.

References

[1] Bishop Brock, Matt Kaufmann, and J S. Moore. ACL2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri, editors, *First international conference on formal methods in computer-aided design*, volume 1166, pages 275–293, Palo Alto, CA, USA, 1996. Springer Verlag.

[2] F.J. Martín-Mateos, J.A. Alonso, M.J. Hidalgo, and J.L. Ruiz-Reina. A generic instantiation tool and a case study: A generic multiset theory. Third International Workshop on the ACL2 Theorem Prover and its Applications, April 2002.

[3] Jun Sawada. ACL2 computed hints: Extension and practice. ACL2 Workshop, October 2000.

[4] Matt Kaufmann and J S. Moore. Design goals of ACL2. Technical Report 101, Computational Logic, Inc., August 1994.

[5] Matt Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on common lisp. *Software Engineering*, 23(4):203–213, 1997.

[6] Matt Kaufmann and Rob Sumners. Efficient rewriting of operations on finite structures in ACL2. Third International Workshop on the

ACL2 Theorem Prover and Its Applications, April 2002.

[7] J S. Moore. Finite set theory in ACL2. 14th International Conference on Theorem Proving in Higher Order Logics, May 2000.

[8] Panagiotis Manolios and Matt Kaufmann. Adding a total order to ACL2. Third International Workshop on the ACL2 Theorem Prover and its Applications, April 2002.