Finite Set Theory on Ordered Lists in ACL2

Jared Davis
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712, USA
jared@cs.utexas.edu
http://www.cs.utexas.edu/users/jared/

Abstract. We present a finite set theory implementation for ACL2. Our library represents sets as fully ordered lists, and provides efficient implementations of the typical set theory operations such as insertion, deletion, union, intersection, difference, cardinality, and sorting lists to create sets. It also includes facilities for quantifying predicates over sets, filtering sets by some criteria, and taking images of sets.
We demonstrate that despite our insistence on full order, it is possible to mirror traditional set theoretic proof techniques and reason through membership. At the same time, we are able to benefit from having a unique representation for each set, which unifies the notions of set and element equality and allows us to handle nested sets trivially.

1 Introduction

ACL2 is a functional programming language based on a subset of Common Lisp. Its users can introduce new functions, test them on concrete inputs, and run them as programs. Each new function defined in ACL2 also corresponds to an axiom in an extensible, first-order logic of total recursive functions: the ACL2 logic. To reason in this logic, ACL2 includes a general purpose theorem prover which can be guided either indirectly with lemmas or directly with hints.

In this paper we present a new finite set theory implementation for ACL2. ACL2 is often used to model other systems such as microprocessors [2] or virtual machines [4]. Our goal is to provide a useful library of set theory operations and reasoning that can be used during the development of these models and other ACL2 software. The execution speed of ACL2 models is relevant for their use as simulators, so our set operations should ideally be as efficient as possible.

The logic supports five kinds of objects: numbers, symbols, characters, strings, and pairs (“conses”). Conses are constructed with the function cons\((a, b)\). Given a cons \(x\), \(\text{car}(x)\) returns its first element, and \(\text{cdr}(x)\) its second. Lists are defined recursively as follows: the symbol \(\text{nil}\) represents the empty list, and \(\text{cons}(a, b)\) is a list whenever \(b\) is a list. We will generally abbreviate the list \(\text{cons}(x_1, \text{cons}(x_2, \ldots, \text{cons}(x_n, \text{nil}) \ldots))\) as \(\langle x_1 \ x_2 \ldots \ x_n \rangle\).

ACL2 is dynamically typed, so conses are permitted to contain elements of mixed types. Types can be “recognized” with various functions, e.g., consp\((x)\)
returns true when \( x \) is a pair, integerp\((x)\) returns true when \( x \) is an integer, and so forth. Functions in the ACL2 logic are total and must operate on any type of object, so for example the function equal\((x, y)\) can compare any two objects for equality and thus corresponds exactly to the meta-theoretical notion of equality between terms.

There are extensive lemma libraries and various decision procedures in place for reasoning in this logic, and we would like to work within these existing frameworks. This immediately requires that our representation of sets be based on conses, the only ACL2 data type which can “contain” other objects. Still, conses are flexible enough that we can imagine representing sets in many different ways (e.g., lists or trees, ordered or unordered, with duplicates permitted or not).

Moore [5] has implemented finite set theory in ACL2 using unordered lists that permit duplicates. In his library a set may have many representations, e.g., the set \( \{1, 2\} \) can be represented either by any of the lists \((1 2)\), \((2 1)\), \((1 1 2)\), etc., so a new function is needed to test if sets are equivalent. Supporting nested sets is also complicated: the straightforward definitions of equality, subset testing, and set membership would be mutually recursive (which would complicate reasoning about them), so canonicalization is used to break the loop at set equality. Still, there are complications: suppose we want to know if \( e \in X \), and \( e \) is a cons. If \( e \) is a set we need to use our set equality function to test for a nested set, but if \( e \) is some non-set cons we will need to use equal instead. The library introduces a convention to denote non-set conses, but this adds some complication for the user. Finally, there are efficiency issues to consider: operations such as subset testing and cardinality have quadratic time complexity (although ignoring duplicates allows insertion to be constant time).

In contrast, our work bases sets on ordered lists with no duplicates. This provides a unique representation for any set, an advantage which holds over many tree-based representations as well. As a result, only the standard definition of equality is needed and nested sets pose no challenge. Also, all of our functions are implemented as linear time operations by taking advantage of the set order. These characteristics make our ordered sets library attractive, but required some careful thought. Moore had also explored this approach, but concluded that it “complicated set construction to a degree out of proportion to its merits.” In this paper, we will show how this complication can be hidden away beneath an abstract, membership-based view of sets.

Set theory has been studied extensively in the theorem proving community. Paulson [7, 8] has axiomatized ZF set theory in Isabelle and derived set-based formalizations of pairing, functions, relations, natural numbers, and other recursive data types. The Mizar community (www.mizar.org) has used Tarski-Grothendieck set theory [1] as a basis for work which today spans some 2,000 definitions of mathematical concepts and 30,000 related theorems. Both of these efforts include proofs of many “heavy” theorems from classic mathematics.

Our work is significantly different in character from these efforts. To be usable from within ACL2 models, our set theory library cannot be based on an axiomatization of set theory and must rather exist within the framework of the
ACL2 logic. Since ACL2 users can already write efficiently executable functions and use many types of objects (e.g., pairs, lists, numbers), we would add little value by reformulating these ideas with set theory. Finally, our interest is in providing set theory as a utility for use in the development of other ACL2 programs, rather than to study and prove the classical properties of set theory itself.

We begin by introducing the core set operations and the strategies used to reason about them (Section 2). We show how ordered lists can be hidden and how our sets can be reasoned about using only membership. We then show how MBE, a new feature of ACL2, can be used to provide efficient versions of the set operations while preserving the reasoning strategies we have already developed (Section 3). We then work towards making the library more easily extensible to new problem domains through instantiable “templates” for common usage patterns (Section 4). Finally, we conclude by looking at future directions for the library (Section 5) and ACL2 itself.

2 The Basic Set Operations

We would like to represent sets with those lists which are fully ordered. To do this, we first need a notion of a total order on ACL2 objects. The details [13] are not relevant, so simply assume that the function ≪ is a total order, i.e., it is irreflexive, asymmetric, transitive, and can compare any two ACL2 objects.

Like Common Lisp, ACL2 is dynamically typed. Typically, the ACL2 user does not define a new data type, but rather introduces a predicate which recognizes those objects which are considered to be of the correct type. So, instead of introducing a new data type for sets, we simply write a function, setp, that decides whether or not an object is a set (i.e., whether it is a list whose elements are all in order). Using an ML-like pattern-matching notation, we might write this function as follows:

\[
\begin{align*}
\text{setp}(\text{nil}) &= \text{true} \\
\text{setp}(x :: \text{nil}) &= \text{true} \\
\text{setp}(x :: y :: xs) &= (x \ll y) \text{ and setp}(y :: xs)
\end{align*}
\]

We mentioned earlier that ACL2 functions must produce a value for any input. For our definition of setp to be admissible, we must add a final case to handle non-list objects, such as \text{setp}(\text{false}) = \text{false}. This would not make sense in a statically typed language, but in Lisp and ACL2 it is perfectly legitimate to ask if, e.g., 3 or “foo” is a set. In the concrete syntax of ACL2, setp is:

\[
\begin{align*}
\text{(defun setp} (X) & \text{)} \\
\text{(if} \text{ (endp} X) & \text{)} ; \text{if } X \text{ is not a list} \\
\text{(equal} X \text{ nil}) & \text{)} ; \text{true iff } X \text{ is nil} \\
\text{(or} \text{ (equal} (\text{cdr} X) \text{ nil}) & \text{)} ; \text{else true if } X \text{ is } x::\text{nil} \\
\text{(and} \text{ (consp} (\text{cdr} X)) & \text{)} ; \text{else true if } X \text{ is } x::y::xs \\
\text{(\ll (car} X) (\text{cadr} X)) & \text{)} ; \text{and } x \ll y \\
\text{\text{setp} (\text{cdr} X)))))) & \text{) ; and } y::xs \text{ is a set}
\end{align*}
\]
2.1 The Primitive Level

Like `setp`, our other set operations (e.g., `union`) must return some value not only when they are applied to sets, but also when they are given non-sets as inputs. As in [5], we adopt a sweeping non-set convention: if one of our functions is passed a non-set object where a set is expected, we treat the object as the empty set, `nil`.

This is a fairly standard trick among ACL2 users, and essentially it amounts to mapping the universe of ACL2 objects into the universe of sets such that all of the sets map to themselves, and anything which is not a set is mapped to some sensible default. We happen to choose the empty set, but we could have chosen any other set, so long as we are consistent in our choice. Through this mapping, we are able to make many theorems “hypothesis-free.” For example, `subset(X, X)` is now a global truth irrespective of the type of `X`. This has several useful consequences [12]: our theorems become more widely applicable, and can be applied more quickly because fewer hypotheses must be relieved.

It is tempting to implement the remainder of the sets package directly using the usual Lisp primitives such as `car`, `cdr`, `cons`, and `endp`\(^1\). Indeed, this was our initial approach. Using these functions, proofs about “simple” operations (e.g., `insert` or `subset`) were possible, but as Moore discovered, proofs about “complicated” operations (union, intersect, ...) 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, the list `(1 1)` contains duplicate elements, so it is not a set. Yet, the list primitives do not treat it as the empty set:

- `car((1 1)) = 1`, but we would prefer `nil`.
- `cadr((1 1)) = (1)`, but we would prefer `nil`.
- `endp((1 1)) = nil`, but we would prefer `true`.
- `cons(1, (1 1)) = (1 1 1)`, but we would prefer `(1)`.

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

- `sfix(X) = X` on sets, `nil` on non-sets (our mapping function).
- `head(X) = car(X)` on sets, but `nil` on non-sets.
- `tail(X) = cdr(X)` on sets, but `nil` on non-sets.
- `empty(X) = endp(X)` on sets, but `true` on non-sets.
- `insert(a, X)` — ordered insert on sets, but treats non-set `X`’s as `nil`.

The set primitives form a primitive level of abstraction, one step removed from the list primitives. This abstraction becomes useful when, after proving some basic theorems about these functions, we instruct the theorem prover to “disable” their definitions: from that point on, the prover is not allowed to use the definitions of `head`, `tail`, and so forth, unless we explicitly permit it to do so. As a result, cases for handling the non-sets vs. sets are merged together and do not impact our proofs.

\(^1\) `endp(x)` is simply `not(consp(x))` and is often used to check for the empty list.
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 \textit{insert}'s definition and some rules about \lll. 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. These are the simple definitions you would find in any functional or logic programming setting:

\begin{align*}
in(a, X) &= \text{not}(\text{empty}(X)) \text{ and } [\text{equal}(a, \text{head}(X)) \text{ or in}(a, \text{tail}(X))] \\
\text{subset}(X, Y) &= \text{empty}(X) \text{ or } [\text{in}(\text{head}(X), Y) \text{ and subset}(\text{tail}(X), Y)]
\end{align*}

We then prove that \textit{head}(X) is not a member of \textit{tail}(X). This theorem gives us the understanding that set elements are unique, without making any mention of \lll. We also provide a new induction scheme for \textit{insert}. \textit{Insert} is the fundamental operation through which we construct sets, and inducting over its definition is a common necessity at this point. Since \textit{insert} is defined using the set order, in the course of these inductions \lll will be introduced into our proofs. We can avoid this by defining a new, “weak” induction scheme which uses membership for its cases rather than \lll.

The new scheme for \textit{insert}(a, X) is straightforward. One base case is that X is empty, in which case we know \textit{insert}(a, X) = \{a\}, and another base case is that \(a \in X\), in which case we know that \textit{insert}(a, X) = X. Otherwise, a will be added somewhere in the list. As a third base case, if \textit{head}(\textit{insert}(a, X)) = a, then \(a\) has been added to the front of the list and we need no inductive hypothesis. Our final and only inductive case is when \textit{head}(\textit{insert}(a, X)) \neq a, in which case it must be \textit{head}(X), and we will inductively assume that our property holds for \textit{insert}(a, \textit{tail}(X)). We effectively get the power of induction over \textit{insert} without mentioning \lll.

\textbf{Pick a Point Proof Strategy} In traditional mathematics, membership arguments are the standard way of proving subset relations. In other words, to show that \(X \subseteq Y\), we typically appeal to the classical definition of subset, i.e., we try to show \(\forall a, a \in X \implies a \in Y\). These proofs are sometimes called pick a point proofs, since proving this universally quantified is equivalent to picking some arbitrary “point” \(a \in X\) and showing that \(a \in Y\).

Unfortunately, we cannot express the key reduction from \(X \subseteq Y\) to \(\forall a, a \in X \implies a \in Y\) in ACL2 as a standard rewrite rule, because all variables we can use in such rules are implicitly universally quantified. However, we can use ACL2’s encapsulation mechanism to accomplish a similar reduction. Encapsulation [11] is somewhat similar to parameterized theories and theory morphism mechanisms used in other theorem provers. It allows us to prove properties about undefined function symbols, constrained to satisfy certain properties, and then reuse these proofs for real functions that also satisfy those constraints.
The basic idea is the following: suppose \( \text{hyps}() \), \( \text{sub}() \), and \( \text{super}() \) are some functions of no arguments which happen to satisfy the following constraint:

\[
\text{hyps}() \land \text{in}(a, \text{sub}()) \Rightarrow \text{in}(a, \text{super}())
\]

By induction using of a witness function which explicitly searches for a counterexample, we can prove the following theorem, called \textit{subset-by-membership}.

\[
\text{hyps}() \Rightarrow \text{subset}(\text{sub}(), \text{super}())
\]

Now, to prove a concrete subset relationship, we can simply instantiate the functions \( \text{sub}, \text{super}, \) and \( \text{hyps} \) with the appropriate \( \lambda \)-expressions. This allows us to conclude that one expression is a subset of another simply by showing that our constraint holds for our choices of \( \text{hyps}, \text{sub}, \) and \( \text{super} \). In other words, it allows us to reduce \( X \subseteq Y \) to \( \forall a, a \in X \Rightarrow a \in Y \).

ACL2 will not automatically attempt functional instantiation, but can be instructed to do so through explicit hints. Still, we would prefer a more automatic solution. Towards this end, we have developed computed hints [6] which automatically apply this strategy when it seems applicable. Such “default” hints essentially extend the theorem prover with a new heuristic for the explicit purpose of proving subset relations.

When does the hint apply? The general idea is if our goal is to prove a subset relationship, and all other attempts at simplifying the conjecture have been exhausted, then a functional instantiation hint is automatically suggested. The \( \lambda \)-substitutions to make are extracted from the conjecture itself, i.e., if the conjecture is \( H \Rightarrow X \subseteq Y \), then we instantiate \textit{subset-by-membership} with \{\( \text{hyps}/(\lambda().H), \text{sub}/(\lambda().X), \text{super}/(\lambda().Y) \}\}. Then, we only have to show that our constraint holds, i.e., \( H \land a \in X \Rightarrow a \in Y \).

**Double Containment Proof Strategy** Our attention now turns to proving double containment is equality, and \textit{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 \( \text{head}(X) = \text{head}(Y) \). Next we show \( \text{in}(a, (\text{tail}X)) \Rightarrow \text{in}(a, \text{tail}(Y)) \)

\textit{Subset-by-membership} is then called upon twice to show \( \text{tail}(X) \) is a subset of \( \text{tail}(Y) \), and vice versa. We induct on a “double-tail” scheme, so that the inductive hypothesis asserts if the tails are mutual subsets they are equal. Now \( \text{head}(X) = \text{head}(Y) \) and \( \text{tail}(X) = \text{tail}(Y) \), so we conclude \( X = Y \).

The resulting rewrite rule, which we call \textit{double-containment}, will replace instances of \( X = Y \) with \( X \subseteq Y \land Y \subseteq X \) whenever \( X \) and \( Y \) are known to be sets. ACL2 will automatically try to use this rule in future proofs, to reduce equalities between sets 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 \textit{delete} function and several theorems pertaining to it, then inducting by deleting \( \text{head}(X) \) from both sides of the proposed equality. In contrast, the above is much cleaner as the theorems about \textit{delete} (which had themselves required induction arguments) can now be proven automatically by double containment.
In summary, set equalities are reduced to containment arguments, and containment arguments are reduced to membership arguments. The beauty of this strategy is that properties of membership, which are typically easy to prove, are now sufficient to conclude subset and equality relations between complicated expressions that might otherwise require hard inductions. Finally, all remaining theorems mentioning the set order are disabled, prohibiting ACL2 from using them in future reasoning. Membership alone will now be used to prove theorems about the set theory functions.

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 (e.g., head and tail) at this point. The top level introduces the remaining set theory functions: delete, union, intersect, difference, and cardinality. These definitions are trivial and inefficient, and we omit them.

The membership level is a powerful basis for our work. Theorems about these “complicated” functions, so hard to prove in both our early attempts and in Moore’s work, are now automatic. We simply introduce the new functions and prove (through simple inductions) that they (1) produce sets and (2) have the right membership characteristics. From then on, our pick a point and double containment strategies can automatically discover many interesting theorems, such as the associativity of union and intersect, the symmetry of union and intersect, the distributivity of union over intersect, the DeMorgan laws for distributing difference, and so forth. Some other theorems, such as cardinality properties, are not based entirely on the pick a point method but are still carried out without mentioning the set order (e.g., through membership based induction over insert). Many selected theorems are listed in Appendix A.

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

3 Execution Efficiency

When sets are represented 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 simpler membership-based 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.

3.1 Guards and MBE

Although ACL2 functions are total, guards [9, 10] allow us to state an “intended domain” for functions. Guards are often presented as a tool for ensuring the
compatibility of ACL2 code with Common Lisp, but they can also be used as run-time 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 guards). We add guards to our basic set operations as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>Guard</th>
</tr>
</thead>
<tbody>
<tr>
<td>setp(X)</td>
<td>none</td>
</tr>
<tr>
<td>empty(X)</td>
<td>setp(X)</td>
</tr>
<tr>
<td>sfix(X)</td>
<td>setp(X)</td>
</tr>
<tr>
<td>head(X)</td>
<td>setp(X)</td>
</tr>
<tr>
<td>tail(X)</td>
<td>setp(X)</td>
</tr>
<tr>
<td>insert(a, X)</td>
<td>setp(X)</td>
</tr>
<tr>
<td>in(a, X)</td>
<td>setp(X)</td>
</tr>
<tr>
<td>subset(X, Y)</td>
<td>setp(X) \land setp(Y)</td>
</tr>
<tr>
<td>delete(a, X)</td>
<td>setp(X)</td>
</tr>
<tr>
<td>union(X, Y)</td>
<td>setp(X) \land setp(Y)</td>
</tr>
<tr>
<td>intersect(X, Y)</td>
<td>setp(X) \land setp(Y)</td>
</tr>
<tr>
<td>difference(X, Y)</td>
<td>setp(X) \land setp(Y)</td>
</tr>
<tr>
<td>cardinality(X)</td>
<td>setp(X)</td>
</tr>
</tbody>
</table>

Introduced recently, MBE allows two separate definitions — one logical, and one executable — to be provided for a single function. When reasoning about the function in the ACL2 logic, 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 accepted, both definitions must be proven to produce the same answer for any inputs satisfying the guards (MBE stands for “must be equal”).

### 3.2 Efficiency Set Operations

We can use MBE to achieve linear implementations of all our set functions. First we provide faster versions of the set primitives. To support the non-set convention, our set primitives use setp to decide if their argument is a set, but this is wasteful since setp examines the entire set. With guards in place to ensure the primitives are only called on sets, these setp calls are no longer necessary, and we can use MBE to replace our set primitives with the equivalent list primitives: sfix(X) is replaced by X, empty by endp, head by car, and tail by cdr. Insert is not changed; it becomes linear once the primitives have been made efficient.

Our attention then turns to the other operations. In is already linear, but we could alternately use the set order to stop early, e.g., in(1,(2 3 4)) could immediately return false because 1 ≪ 2. However, it is not clear that doing this would be an efficiency win, as in would need to call ≪ at each step, so we arbitrarily choose not to make this substitution.

Linear versions of subset, union, intersect, and difference, are then introduced. These functions simultaneously walk through the elements of both of their
arguments, and are significantly more complicated than the simple definitions we use for reasoning. Except for \textit{subset}, the equivalence proofs 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 \textit{union}, etc. Even so, these are not trivial proofs and turn out to involve many cases: \textit{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 implementations are dependent on our underlying representation.

3.3 Adding a Sort

We would also like an efficient method for building large sets. Towards this purpose we provide a simple merge sort, 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 set construction in Moore’s unordered set library, and is the inescapable price paid for full ordering. The sort itself is easy to write using our already efficient \textit{union} operation to perform the merge. As with the other basic set operations, we use MBE to combine easy reasoning with efficiency in execution: \textit{mergesort} is logically viewed simply as repeated insertions.

3.4 Comments on this Approach

This approach frees the author to focus on one concern at a time. Execution efficiency was ignored completely as we designed our logical definitions and reasoning strategy, and even our most primitive functions had to examine the entire set. After our theory was complete, efficient implementations of our functions were developed, but these definitions were never reasoned about beyond showing that they faithfully implemented the simple models.

Without the separation that MBE enables, we are caught in a dilemma: reasoning conveniences like the non-set convention and our simple mathematical definitions would inflict enormous efficiency penalties, perhaps rendering our library too slow to be practically useful, yet without such conveniences our theorems would be burdened with extra hypotheses, our proofs would be larger, and more effort would be required to design and to use the library. But with this separation we sacrifice neither execution efficiency nor reasoning ability: all of our set operations are now constant or linear time, yet their logical definitions are simply the Lisp reflections of their mathematical meanings, free from the complications of their implementations.

4 Instantiable Extensions

We have now covered the core of the sets library. Though efficient and straightforward to reason about, this core has limited capabilities. Although it is certainly
impossible to foresee and cater to every future need, we suspect that having extended functionality may make modeling new problems more easily.

Two extensions seem to be particularly good candidates. Having recently seen the benefits derived from our subset-by-membership strategy, which is essentially the quantification of membership over sets, it seems desirable to provide a more general framework for quantifying other predicates over sets. Furthermore, the time honored patterns of functional programming are probably also good candidates with which to extend the library.

4.1 Quantification

We would like to be able to support quantifying predicates over sets. In other words, for some set $X$, we would like to be able to ask “$\forall a \in X, P(a)$,” or “$\exists a \in X, P(a)$.” These ideas make natural second order functions, but since ACL2 is a first order system, we instead create a fully instantiable generic theory [3] and provide a macro to create concrete, first order instances of this theory.

In general, given a predicate $P(a, \ldots)$, where $\ldots$ is understood to represent 0 or more extra arguments, the user can invoke a single macro to create the following functions:

- $\text{all} \langle P \rangle (X, \ldots)$, returns true iff $\forall a \in X, P(a, \ldots)$
- $\text{exists} \langle P \rangle (X, \ldots)$, returns true iff $\exists a \in X, P(a, \ldots)$
- $\text{find} \langle P \rangle (X, \ldots)$, returns an element $a \in X$ such that $P(a, \ldots)$, or $\text{nil}$ if no such element exists.

A strategy for reasoning about these functions is also created. Some selected theorems from this strategy are listed in Appendix A, but we also notice that $\text{subset} (X, Y)$ is exactly $\text{all} \langle \text{in} \rangle (X, Y)$. We generalize the pick a point strategy developed for $\text{subset}$ so that it can also be used to reduce a proof of $\text{all} \langle P \rangle (X, \ldots)$ to $\forall a, a \in X \Rightarrow P(a, \ldots)$, allowing us to perform pick a point style proofs for any predicate instead of just membership. As before, we set up computed hints to automatically attempt these strategies for each instance of our theory.

Instantiating this theory with simple predicates such as $\text{integerp}$ allows us to define typed sets. More complex predicates allow us to express notions such as “sets of integers less than $k$.” We have found this theory to be useful, e.g., for quickly introducing graph theory in terms of edge sets and vertex sets.

4.2 Extensions from Functional Programming

We now extend the quantification macro so that the user can filter sets, keeping only those elements that satisfy some predicate. In addition to creating $\text{filter} \langle P \rangle (X, \ldots)$, a few theorems are also created. Filtering is straightforward and we will not discuss it in depth, but one brief observation is that $\text{intersect} (X, Y)$ is nothing more than $\text{filter} \langle \text{in} \rangle (X, Y)$.

We then create a new macro that allows the user to introduce $\text{map} \langle F \rangle (X, \ldots)$, which takes the image of set $X$ under some function $F(a, \ldots)$. Its definition is:
map(F)(X, ...) = \{ nil \text{ if empty}(X) \\
insert(F(head(X), ...), map(F)(tail(X), ...)) \text{ otherwise} \}

But this is inefficient (we are essentially performing an insert sort). To remedy this, we also introduce the following function:

maplist(F)(X, ...) = \{ nil \text{ if endp}(X) \\
cons(F(car(X), ...), maplist(F)(cdr(X), ...)) \text{ otherwise} \}

Through double containment, we show that when setp(X), map(F)(X, ...) = mergesort(maplist(F)(X, ...)). We guard map with setp(X), then use MBE to make this substitution for better execution efficiency, while retaining our simple definition for reasoning.

To reason about map, we first show that map always produces sets. We would then like to discuss map’s relationship with membership. Towards this end, we introduce the following predicate, which returns true a is an inverse of b.

inversep(F)(a, b, ...) = equal(F(a, ...), b)

We then use our previous macro to quantify inversep(F) over sets, and prove the following is a theorem.

\[ \text{in}(a, \text{map}(F)(X, ...)) = \text{exists}(\text{inversep}(F))(X, a, ...). \]

We give this as a rewrite rule so that any instances of in(a, map(F)(X, ...)) will be rewritten away into the existence of inverses. Since we already know how to reason about exists, we can now reason well about membership in map.

### 4.3 A Concluding Example

Now that we can reason about membership in mappings, map fits very nicely into our overall membership-based reasoning strategy. We conclude with an example of this strategy at work. This proof is discovered by ACL2 with no user intervention. Other similar theorems are also listed in Appendix A. For brevity, we will omit the ... and assume that F has only a single argument.

map(F)(\text{union}(X, Y)) = \text{union}(map(F)(X), map(F)(Y))

Since map and union both produce sets, ACL2 attempts a proof by double containment.

The first goal is to show that map(F)(\text{union}(X, Y)) is a subset of union(map(F)(X), map(F)(Y)). To prove this, ACL2 attempts a pick a point proof, and supposes that there is some element a ∈ map(F)(\text{union}(X, Y)). ACL2 now uses our theorem about map membership to conclude exists(inversep(F))(\text{union}(X, Y), a). A simple rewrite about existence in unions allows ACL2 to conclude: exists(inversep(F))(X, a) or exists(inversep(F))(Y, a).
Now, ACL2 wants to show that $a \in \text{union}(\text{map}(F)(X), \text{map}(F)(Y))$. By our membership property of union, it sees that this is the same as showing $\text{in}(a, \text{map}(F)(X))$ or $\text{in}(a, \text{map}(F)(Y))$. Applying our map membership theorem twice, it sees that this is the same as showing that either $\exists \text{inversep}(F)(X, a)$ or $\exists \text{inversep}(F)(Y, a)$. But this is exactly what it concluded was true in the previous paragraph, so it is done with this goal. The second goal is similarly straightforward.

5 Conclusions

When designing functions to be reasoned about, hiding complexity behind layers of abstraction seems to be crucial. This is the entire idea behind the primitive level, where our set primitives such as $\text{head}$ and $\text{tail}$ serve the useful purpose of containing the cases needed to support the non-set convention. It is also the motivation behind using simple logical definitions for our set theory functions such as $\text{subset}$, $\text{union}$, and so forth, which are based on membership rather than the set order.

Developing a library for ACL2 involves not only writing the code for the library’s functions, but also developing useful strategies for reasoning about those functions. These goals can conflict with one another because efficient code is often more complicated to reason about. MBE has shown itself to be an extremely useful tool. It allows us to soundly use efficient implementations of our functions for execution while still using simple definitions for reasoning. This allows our library to simultaneously provide the best of both worlds to our users.

For the set theory library in particular, a cornerstone of our reasoning ability is our 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 $\text{subset}$ (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.

Mainstream programming languages often offer a myriad of container classes. ACL2 users 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 is as or more important than efficiency.
Acknowledgements I am thankful to Sandip Ray for his help with 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.

References

A Selected Theorems

The following is a list of many of the theorems provided by our ordered sets library. It is not comprehensive, but should give a good flavor of the rewriting strategy. For brevity, the instantiable functions (e.g., \textit{all}), are written without predicates or extra arguments, and we freely mix traditional and ACL2 syntax.

\textbf{Set Creation}

\begin{align*}
\text{setp (sfix X)} & \quad \text{setp (intersect X Y)} \\
\text{setp (tail X)} & \quad \text{setp (difference X Y)} \\
\text{setp (insert a X)} & \quad \text{setp (mergesort x)} \\
\text{setp (delete a X)} & \quad \text{setp (filter X)} \\
\text{setp (union X Y)} & \quad \text{setp (map X)}
\end{align*}

\textbf{Membership}

\begin{align*}
\text{in a (insert b X)} & = \text{in a X} \lor \text{equal a b} \\
\text{in a (delete b X)} & = \text{in a X} \land \neg \text{equal a b} \\
\text{in a (union X Y)} & = \text{in a X} \lor \text{in a Y} \\
\text{in a (intersect X Y)} & = \text{in a Y} \land \text{in a X} \\
\text{in a (difference X Y)} & = \text{in a X} \land \neg \text{in a Y} \\
\text{in a (mergesort x)} & \leftrightarrow \text{in-list a x} \\
\text{in a (filter X)} & = \text{P a} \land \text{in a X} \\
\neg \text{in a a}
\end{align*}

\textbf{Non-Set Convention}

\begin{align*}
\text{empty (sfix X)} & = \text{empty X} \quad \text{intersect (sfix X Y)} = \text{intersect X Y} \\
\text{head (sfix X)} & = \text{head X} \quad \text{intersect X (sfix Y)} = \text{intersect X Y} \\
\text{tail (sfix X)} & = \text{tail X} \quad \text{difference (sfix X Y)} = \text{difference X Y} \\
\text{in a (sfix X)} & = \text{in a X} \quad \text{difference X (sfix Y)} = \text{difference X Y} \\
\text{insert a (sfix X)} & = \text{insert a X} \quad \text{cardinality (sfix X)} = \text{cardinality X} \\
\text{delete a (sfix X)} & = \text{delete a X} \quad \text{all (sfix X)} = \text{all X} \\
\text{subset (sfix X Y)} & = \text{subset X Y} \quad \text{find (sfix X)} = \text{find X} \\
\text{subset X (sfix Y)} & = \text{subset X Y} \quad \text{filter (sfix X)} = \text{filter X} \\
\text{union (sfix X Y)} & = \text{union X Y} \quad \text{map (sfix X)} = \text{map X} \\
\text{union X (sfix Y)} & = \text{union X Y}
\end{align*}

\textbf{Insertion, Deletion}

\begin{align*}
\neg \text{empty (insert a X)} \\
\text{in a X} \Rightarrow \text{equal (insert a X) (sfix X)} \\
\text{insert a (insert b X)} & = \text{insert b (insert a X)} \\
\text{insert a (insert a X)} & = \text{insert a X} \\
\text{insert a (delete a X)} & = \text{insert a X} \\
\neg \text{in a X} \Rightarrow \text{equal (delete a X) (sfix X)} \\
\text{delete a (delete b X)} & = \text{delete b (delete a X)} \\
\text{delete a (delete a X)} & = \text{delete a X} \\
\neg \text{in a X} \Rightarrow \text{equal (insert a X) (sfix X)} \\
\text{delete a (delete b X)} & = \text{delete b (delete a X)} \\
\text{delete a (delete a X)} & = \text{delete a X}
\end{align*}
(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))
\(\emptyset (\text{union} X Y)\) = (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 Y 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)
\(\cap (\cap X Y) Z = (\cap X (\cap Y Z))\)
\(\cap X (\cap Y Z) = (\cap Y (\cap X Z))\)
(intersect X (intersect X Z)) = (intersect X Z)
¬(in a Y) ⇒ (\(\cap (\text{insert a} X) Y = (\cap X Y)\))
¬(in a X) ⇒ (\(\cap X (\text{insert a} Y) = (\cap X Y)\))

**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 ≤ |X|
(|X| = 0) = (empty X)
\(|X \cap Y| \leq |X|\)
\(|X \cap Y| \leq |Y|\)
\(|X \cup Y| = |X| + |Y| - |X \cap Y|\)
\(|X - Y| = |X| - |X \cap Y|\)
\(X \subseteq Y \Rightarrow |X| \leq |Y|\)
\(X \nsubseteq Y \Rightarrow |X \cap Y| < |X|\)
\[
\begin{align*}
|\text{(insert a } X)\rangle &= \begin{cases} |X| & \text{if (in a } X) \\ |X| + 1 & \text{otherwise} \end{cases} \\
|\text{(delete a } X)\rangle &= \begin{cases} |X| - 1 & \text{if (in a } X) \\ |X| & \text{otherwise} \end{cases} \\
|\text{(map } X)\rangle &\leq |X|
\end{align*}
\]

**Miscellaneous**

\begin{align*}
\text{(union } X \text{ (intersect } Y Z)\rangle &= \text{(intersect (union } X Y) \text{ (union } X Z)\rangle) \\
\text{(intersect } X \text{ (union } Y Z)\rangle &= \text{(union (intersect } X Y) \text{ (intersect } X Z)\rangle) \\
\text{(difference } X \text{ (union } Y Z)\rangle &= \text{(intersect (difference } X Y) \text{ (difference } X Z)\rangle) \\
\text{(difference } X \text{ (intersect } Y Z)\rangle &= \text{(union (difference } X Y) \text{ (difference } X Z)\rangle)
\end{align*}

**Quantification**

\begin{align*}
\text{(empty } X\rangle \Rightarrow (\text{all } X) &\quad (\text{all (insert a } X)\rangle = (P a) \land (\text{all } X) \\
(\text{all (sfix } X)\rangle = (\text{all } X) &\quad (\text{all } (\cup X Y)\rangle = (\text{all } X) \land (\text{all } Y) \\
(\text{all } X) \Rightarrow (\text{all (tail } X)\rangle &\quad (\text{all } X) \Rightarrow (\text{all (intersect } X Y)\rangle) \\
(\text{all } X) \Rightarrow (\text{all (delete a } X)\rangle &\quad (\text{all } Y) \Rightarrow (\text{all (intersect } X Y)\rangle) \\
(\text{all } X) \land (\text{in a } X) \Rightarrow (P a) &\quad (\text{all } X) \Rightarrow (\text{all (difference } X Y)\rangle) \\
(\text{all } X) \land \neg(P a) \Rightarrow \neg(\text{in a } X) &\quad (\text{exists } X) = (\text{not (all-not } X)\rangle
\end{align*}

**Filtering**

\begin{align*}
(\text{all (filter } X)\rangle &\quad (\text{all } X) \Rightarrow (\text{filter } X) = (\text{sfix } X) \\
(\text{subset (filter } X) \text{ } X) &
\end{align*}

**Mapping**

\begin{align*}
(\text{in a (map } X)\rangle &= (\exists\text{-inversep } X a) \\
(\text{subset } X Y) \Rightarrow (\text{subset (map } X) \text{ (map } Y)\rangle &\quad (\text{map (insert a } X)\rangle = (\text{insert (F a) (map } X)\rangle) \\
(\text{map (delete a } X)\rangle &\supseteq (\text{delete (F a) (map } X)\rangle) \\
(\text{map (union } X Y)\rangle &= (\text{(union (map } X) \text{ (map } Y)\rangle)) \\
(\text{map (intersect } X Y)\rangle &\subseteq (\text{(inter:(map } X) \text{ (map } Y)\rangle)) \\
(\text{map } X) - (\text{map } Y) &\subseteq (\text{map } (X - Y))
\end{align*}