Finite Set Theory in ACL2

J Strother Moore*

Department of Computer Sciences, University of Texas at Austin,
Taylor Hall 2.124, Austin, Texas 78712
moore@cs.utexas.edu telephone: 512 471 9568
WWW home page: http://www.cs.utexas.edu/users/moore

Abstract. ACL2 is a first-order, essentially quantifier free logic of computable recursive functions based on an applicative subset of Common Lisp. It supports lists as a primitive data structure. We describe how we have formalized a practical finite set theory for ACL2. Our finite set theory "book" includes set equality, set membership, the subset relation, set manipulation functions such as `union`, `intersection`, etc., a choice function, a representation of finite functions as sets of ordered pairs and a collection of useful functions for manipulating them (e.g., domain, range, apply) and others. The book provides many lemmas about these primitives, as well as macros for dealing with set comprehension and some other "higher order" features of set theory, and various strategies or tactics for proving theorems in set theory. The goal of this work is not to provide "heavy duty set theory" – a task more suited to other logics – but to allow the ACL2 user to use sets in a "light weight" fashion in specifications, while continuing to exploit ACL2's efficient executability, built in proof techniques for certain domains, and extensive lemma libraries.

1 Introduction

Doesn't ACL2 [4, 3] already provide sets? It contains such standard Lisp functions as `member`, `subsetp`, `union` and `intersection`. These operate on lists, ignoring duplication and order. For example, consider the two lists `(1 2)` and `(2 1)`. Both lists can be used to represent the set $\{1, 2\}$. `Member` can be used to determine that `1` and `2` are elements of both lists and that `3` is an element of neither. `Subsetp` can be used to determine that each list is a subset of the other and so we can say they are "set equal."

But these primitives are like their set theory counterparts only on sets of atoms. For example, `(2 1)` is not a `member` of `((1 2))` even though $\{2, 1\}$ is a member of $\{\{1, 2\}\}$. That is because `member` compares elements with Lisp's `equal`, not "set equality." We wish to define finite sets in such a way that sets can be elements of other sets.

Extensive set theory work has been done in both the Isabelle and the Mizar communities. For example, Paulson [9, 10] describes a formalization of Zermelo-

* This work was supported in part by Compaq Systems Research Center, Palo Alto, CA.

Fraenkel set theory and proves Cantor's Theorem and Ramsey's Theorem. Paulson and Grabczewski develop ZF to the point of proving equivalent twenty different formulations of the Axiom of Choice. The Mizar system is essentially based on Tarski Grothendieck set theory and virtually all of the extensive results proved by the Mizar community and published in their *Journal of Formalized Mathematics* (`http://www.mizar.org/JFM`) may be thought of as set theoretic in nature. See especially [1] and the other articles in Volume 1 of the Journal. There is a little set theory work in HOL [2]. If one's goal is to check the results of set theory, we would recommend Isabelle or Mizar.

But our goal is not "heavy duty set theory." Our goal is to provide sets to the ACL2 user. A typical use of ACL2 is to build an executable model of some system. Execution speed is essential. The ACL2 model of the microprocessor described by Greve and Wilding in [3] executes at 3 million simulated instructions per second (on a 733 MHz machine). Lisp's data structures are excellent for this. But now imagine that in formalizing some property, the ACL2 user wishes to speak of the set of all the states visited during some execution or the powerset of the set of all scheduled threads. A little "light weight" set theory is clearly useful here, whether efficiently executable or not.

We could, of course, adopt a system like Isabelle or Mizar and formalize all of ACL2's objects as sets. For example, we could adopt the von Neumann coding and represent the number 1 as the set containing the empty set, say. Then natural number addition has a straightforward set theoretic definition. But the elementary arithmetic functions are already defined in ACL2 and they do not treat $\{\{\}\}$ as the multiplicative identity. So, without some kind of reflection principle, we are forced by this embedding to abandon ACL2's native objects and the functions for manipulating them, in favor of a set theoretic treatment of everything. That also means abandoning ACL2's execution efficiency, built in decision procedures, and existing lemma libraries.

But we want to preserve those things. So we want sets of native ACL2 objects – integers, rationals, strings, symbols and lists representing all manner of other computational objects such as bit vectors, frames, stacks, class tables, superclass hierarchies, etc. We want to be able to load libraries defining various computational objects and functions for manipulating them and theorems relating those functions, and we want to collect those objects together into sets. (Our approach also allows ACL2 objects to contain (objects to be treated as) sets. Since ACL2 is untyped, the functions manipulating those objects must "know" when a component is to be treated as a set and use the set theory functions to manipulate it.)

Having established the need to do *something* to provide first-class finite sets in ACL2 while preserving ACL2's native objects, here is a quick sketch of what we will do.

We are going to develop what might be called the "hereditarily finite sets built from the ACL2 universe of objects." That is, the base elements, or "ur-elements," of our sets will be the ACL2 objects and the elements of all sets will be these ur-elements or other such sets.

ACL2 does not support the introduction of new data types. So we will represent the set theory individuals (e.g., ur-elements and all the sets we can build) as ACL2 objects. We define an equivalence relation, `=`, on these ACL2 objects so that two objects are equivalent precisely if they represent the same set theory individual.[1] We define "set theory functions and predicates" that treat objects as though they were the set theory individuals they represent. These functions and predicates enjoy the expected algebraic properties of their set theory counterparts. Throughout this paper we use "`=`" to mean this new equivalence relation, not the normal Leibniz identity, which we denote as "$=$". Once we get going, the distinction is not germane to this paper.

So that we do not redefine existing Lisp functions, we do all of our work in a new symbol package named `"S"` (for "sets"). Common Lisp provides packages so users can have disjoint name spaces. We "import" into our `"S"` package all of the standard ACL2 symbols except for `union`, `intersection`, `subsetp`, `add-to-set`, `functionp`, `=`, and `apply`, and use `"S"` as the selected (implicit) package in this paper. This means that when we use a familiar Lisp symbol, such as `car`, we are referring to its standard Lisp meaning. That is, `car` is just shorthand for the symbol whose full name is `lisp::car`. But when we use one of the imported symbols above we are referring to the symbol of that name in our `"S"` package, e.g., by `union` we mean `s::union`.

The primitive set theory predicates and functions, such as `member` and `union`, are defined so that `=` is a congruence relation for each argument position. A unary function f admits `=` as a congruence relation if $u \mathtt{=} v \rightarrow f(u) \mathtt{=} f(v)$. This notion is extended to functions and predicates of arbitrary arity and to different equivalence relations in the hypothesis and conclusion. Thus, $u \mathtt{=} v \rightarrow (\mathtt{p}(x,u) \leftrightarrow \mathtt{p}(x,v))$ is a congruence rule. It tells us that $\mathtt{p}(x,u)$ is equivalent (in "$\leftrightarrow$" sense of propositional equivalence) to $\mathtt{p}(x,v)$ when u is equivalent (in the "`=`" sense) to v.

The ACL2 theorem prover contains special provisions for dealing with user-defined equivalence relations and congruence rules. When ACL2 rewrites a term, it maintains a given sense of equivalence, specified by an equivalence relation. That goal equivalence relation and the known congruence rules determine the equivalences that may be used as rewrite rules. For example, suppose the system is rewriting $\mathtt{p}(\gamma, \mathtt{union}(\alpha, \beta))$, maintaining the goal equivalence "$\leftrightarrow$". Then the above congruence rule allows the system to shift from "$\leftrightarrow$" to "`=`" while rewriting the second argument of `p`. That, in turn, allows it to use theorems concluding with "`=`" as rewrite rules. For example, $\mathtt{union}(x,y) \mathtt{=} \mathtt{union}(y,x)$ is such a rule. Using it, the rewriter may replace the $\mathtt{union}(\alpha, \beta)$ here by $\mathtt{union}(\beta, \alpha)$, even though the two terms are not necessarily equal (in the "$=$" sense of Leibniz identity native to the logic).

[1] In ACL2, true and false are denoted by the objects `T` and `NIL`. "Relations" and "predicates" are defined as Boolean valued functions, i.e., functions that return either `T` or `NIL`. By proving that a relation is an equivalence relation the user may cause the ACL2 theorem prover to manipulate it with special techniques described below.

In ACL2, a collection of definitions and theorems is called a "book." Books may be included into an ACL2 session to configure the theorem prover. Our set theory book is available at `http://www.cs.utexas.edu/users/moore/-publications/finite-set-theory`. In this paper we present our formulas in an abstract syntax rather than ACL2's concrete syntax.

Why might this work be of interest outside the ACL2 user community? First, it highlights the importance of certain general theorem proving techniques supported by ACL2 (especially congruence based rewriting). Second, we explore a variety of issues that arise in any attempt to embed one mathematical formalism in another, e.g., identity versus equivalence, mutual recursion versus canonicalization, useful definitional schemes, transparency with respect to existing proof techniques, etc. Third, if one takes the view that all we have done is *implement* finite set theory in a programming language and prove that the implementation satisfies the standard algebraic laws, then the paper represents an interesting challenge to any software verification system. Fourth, if your theorem prover provides set theory, define the powerset function recursively and see if your system can do the proofs required in Section 9 with less guidance than ACL2 requires.

2 Basics

Definitions. The *ACL2 universe* consists of all ACL2 numbers (rationals and complex rationals), characters, strings, symbols, and conses of objects in the ACL2 universe. These notions are made clear in [4]; informally, these are the objects the ACL2 programmer can construct and manipulate. The elements of the ACL2 universe are called *ACL2 objects.*

Definition. An *ur-element* is an ACL2 object other than the keyword symbol `:UR-CONS`.

To motivate what follows, now imagine some construction of finite sets allowing us to distinguish non-empty sets from ACL2 objects and identifying the empty set with the ACL2 object `NIL`. We are interested in all the finite sets containing ur-elements and/or other such finite sets.

Definition. The *hereditarily finite ACL2 sets* is the smallest set S with the property that a set s is an element of S precisely if s is finite and every element of s is either an ur-element or is some element of S.

Definition. The *ACL2 set theory universe* consists of all the ur-elements and all the hereditarily finite ACL2 sets. An element of the ACL2 set theory universe is called an *ACL2 set theory individual.* Note that not every set theory individual is a set, e.g., some are numbers, strings, etc.

We are interested in representing the ACL2 set theory individuals. In particular, each such individual can be represented by an ACL2 object, often in multiple ways.

Definition. Let x be an ACL2 set theory individual. We define a *representative* of x recursively as follows. If x is an atomic ur-element (a number, character, string or symbol), then a (in fact, the) representative of x is x itself. If x is a cons, then a (the) representative of x is `(:UR-CONS` x`)`. If x is the empty set, a representative is `NIL`. Otherwise, x is a non-empty set containing some element e. In this case, a representative of x is the cons whose car is a representative of e and whose cdr is a representative of the set $x \setminus \{e\}$.

In our set theory book we define an equivalence relation `=` with the following property: two ACL2 objects are equivalent under `=` precisely if they represent the same ACL2 set theory individual. We do not show the definition here. Our equivalence relation is insensitive to the order in which elements of sets are presented and in fact allows duplications of elements. Our elimination of the symbol `:UR-CONS` as an ur-element makes our representation of sets non-ambiguous. The `=` relation can distinguish ur-elements from each other and from sets. Two sets are `=` precisely when they are subsets of one another.

Henceforth, we speak of ACL2 objects as though they were the ACL2 set theory individuals they represent. For example, here are three example sets.

- The set containing the symbolic names of the first three days of the week, which ordinarily might be written {`SUNDAY`, `MONDAY`, `TUESDAY`}, may be written `(SUNDAY MONDAY TUESDAY)`. Equivalently (in the "`=`" sense), it may be written `(MONDAY SUNDAY TUESDAY SUNDAY)`.
- The set of digits, as integers, may be written `(0 1 2 3 4 5 6 7 8 9)`.
- The set consisting of the set of even digits and the set of odd digits may be written `((0 2 4 6 8) (1 3 5 7 9))`.

When an ACL2 list, e.g., a machine state, thread, stack, etc., is used as an ur-element, it must be embedded in the `:UR-CONS` form so it is not treated as a set. Here is a set that contains the set of even digits, the set of odd digits, the list of even digits in ascending order and the list of odd digits in ascending order.

```
((0 2 4 6 8)
 (1 3 5 7 9)
 (:UR-CONS (0 2 4 6 8))
 (:UR-CONS (1 3 5 7 9)))
```

Because ACL2's is a syntactically untyped language it is possible to use ur-elements where sets are expected. We deal with this with a sweeping convention. **The Non-Set Convention**. If a non-`NIL` ur-element is used where a set is expected, all set theory functions shall behave as though the empty set had been used instead.

For example, our answer to the question "does `3` occur as an element in the set `5`?" is "no," because `5` is not a set and hence the question is treated as though it had been "does `3` occur as an element in the set $\{\}$?" This should not concern the reader since such "ill-formed" questions are never asked. We tend to ignore such issues in this paper.

Following the normal rules of Lisp, it is necessary to quote values when they are used as literal constants in terms. For example, `cardinality('(1 2 3))` denotes the application of the function `cardinality` to (a constant representing) the set $\{1, 2, 3\}$.

3 Set Theoretic Functions and Theorems Proved

In Figure 1 we name some of the functions and macros defined in the `"S"` package. We discuss below how other operations can be defined.

Since numbers in ACL2 set theory are just ACL2's numbers and all the ACL2 arithmetic functions and predicates (except `=`) are imported into the `"S"` package, the arithmetic functions of ACL2 set theory are exactly the arithmetic functions of ACL2.

In Figure 2 we list some of the theorems available in the set theory book. These theorems have some subtle attractive properties that one can appreciate only by considering alternative formulations of set theory in ACL2. One is that most are not burdened by hypotheses restricting them to sets or individuals. Another is the use of `=` as the only sense of equivalence. Any theorem concluding with an `=` could be used as a rewrite rule (under certain restrictions imposed by ACL2's rewriter). Still another is that the proofs of most of these theorems are extraordinarily simple. We omit many of our theorems (especially duals) for brevity.

4 The Choice Function

In order to allow us to define certain functions on sets, such as extracting the components of an ordered pair represented by $\{x, \{x, y\}\}$, we must be able to obtain an element of a non-empty set. We therefore defined `choose`.

Key properties of `choose`(s) are that it admits `=` as a congruence, i.e., the choice is independent of the representation or presentation of s, and `choose`(s) is a member of s if s is non-empty. Ideally, perhaps, nothing else need be said about `choose`. But it is in fact quite specifically defined. We simply opt seldom to expose its other properties. But logically speaking those properties are present and important.

`Choose` is computable. For example, `choose('(1 2 3 4))` is `4`. Our definition of `choose` chooses the largest element of the set, where the ordering is a certain (arbitrarily defined but unspecified here) total ordering on our objects.

Our definition of `choose` allows us to prove the following property.

Weak Choose-Scons Property:
`choose`(`scons`(e, a)) `=` $e \vee$ `choose`(`scons`(e, a)) `=` `choose`(a).

That is, `choose` on `scons`(e, a) is either e or the choice from a. Of course, it is possible to say exactly which of these cases obtains: `choose`(`scons`(e, a)) is e, if e dominates `choose`(a), and is `choose`(a) otherwise. We call this stronger statement the *Strong Choose-Scons Property.* The strong property requires mention of the total order while the weak property does not. Since this can complicate proofs, we avoid the use of the strong property when possible.

ACL2 allows the user to constrain undefined functions to satisfy arbitrary properties, provided some function (a "witness") can be shown to have those properties. Using this feature of ACL2, it is possible to introduce an undefined

`ur-elementp`(a)	`T` or `NIL` according to whether a is an ur-element.
`setp`(a)	`T` or `NIL` according to whether a is a set.
`scons`(e, a)	$\{e\} \cup a$.
`brace`$(\alpha_1, \ldots, \alpha_k)$	The set whose elements are given by the values of the k expressions; this is known as "roster notation".
a = b	If a and b are the same individual, then `T`, otherwise, `NIL`.
`mem`(e, a)	$e \in a$. Both arguments are treated as sets.
`subsetp`(a, b)	$a \subseteq b$.
`cardinality`(a)	$\|a\|$.
`nats`(n)	$\{i \mid i \in \mathbf{N} \wedge 0 \leq i \leq n\}$.
`union`(a, b)	$a \cup b$.
`intersection`(a, b)	$a \cap b$.
`diff`(a, b)	$a \setminus b$.
`choose`(a)	An element of the set a, if a is non-empty.
`pair`(x, y)	A set representing the ordered pair $\langle x, y\rangle$. We call such a set simply a *pair*. `Pair`(x, y) is defined to be `brace`$(x,$ `brace`$(x, y))$.
`pairp`(a)	`T` or `NIL` according to whether a is a pair.
`hd`(a)	If a is the pair $\langle x, y\rangle$, then x; otherwise, `NIL`.
`tl`(a)	If a is the pair $\langle x, y\rangle$, then y; otherwise, `NIL`.
`pairps`(s)	`T` or `NIL` according to whether s is a set of pairs.
`functionp`(f)	If f is a set of pairs and no two elements of f have the same `hd`, then `T`; otherwise, `NIL`. If a function f contains `pair`(e, v), then we say v is the *value* of f on e.
`domain`(f)	$\{e \mid \exists x(x \in f \wedge e = \texttt{hd}(x)\ )\}$.
`range`(f)	$\{e \mid \exists x(x \in f \wedge e = \texttt{tl}(x)\ )\}$.
`apply`(f, e)	If f is a function and e is in its domain, then the value of f on e.
`except`(f, e, v)	If f is a function then the function that is everywhere equal to f except on e where the value is v.
`restrict`(f, a)	The image of f on those elements of a in the domain of f.
`sequencep`(s)	`T` if s is a function whose domain is $\{1, \ldots, \|s\|\}$, otherwise `NIL`. Generally, ACL2 lists serve more directly but we formalized sequences as an application.
`shift`(i, j, f, d)	If i and j are integers, then the function obtained by mapping $k + d$ to `apply`(f, k), for every $i \leq k \leq j$. If i or j is not an integer, the result is `NIL`.
`concat`(r, s)	`union`$(r,$ `shift`$(1,$ `cardinality`$(s), s,$ `cardinality`$(r)))$.

Fig. 1. Some Functions of Our Set Theory

$\bullet$subsetp(x, x).
$\bullet$subsetp$(a, b) \wedge$ subsetp$(b, c) \rightarrow$ subsetp(a, c).
$\bullet$mem$(e, a) \wedge$ subsetp$(a, b) \rightarrow$ mem(e, b).
$\bullet$setp$(a) \wedge$ setp$(b) \rightarrow ((a = b) \leftrightarrow ($subsetp$(a, b) \wedge$ subsetp$(b, a)))$.
$\bullet\neg$ mem(e, e).
$\bullet$mem$(e,$ union$(a, b)) \leftrightarrow (mem(e, a) \vee$ mem$(e, b))$.
$\bullet$subsetp$(a,$ union$(a, b))$.
$\bullet$subsetp$(a_1, a_2) \rightarrow$ subsetp(union(a_1, b), union$(a_2, b))$.
$\bullet$union$(a, b) =$ union(b, a).
$\bullet$union(union$(a, b), c) =$ union$(a,$ union$(b, c))$.
$\bullet$cardinality$(a) \leq$ cardinality(union$(a, b))$.
$\bullet$cardinality$(a) = 0 \leftrightarrow$ ur-elementp(a).
$\bullet$intersection$(a, b) =$ NIL
$\rightarrow$ cardinality(union$(a, b)) =$ cardinality$(a) +$ cardinality(b).
$\bullet$mem(choose$(a), a) \leftrightarrow \neg$ ur-elementp(a).
$\bullet$choose(scons$(e, a)) = e \vee$ choose(scons$(e, a)) =$ choose(a).
$\bullet$choose(scons$(e,$ NIL$)) = e$.
$\bullet$cardinality$(x) = 1 \wedge$ mem$(e, x) \rightarrow$ scons$(e,$ NIL$) = x$.
$\bullet$mem$(e,$ diff$(a, b)) \leftrightarrow (mem(e, a) \wedge \neg$ mem$(e, b))$.
$\bullet$subsetp$(a_1, a_2) \rightarrow$ subsetp(diff(a_1, b), diff$(a_2, b))$.
$\bullet$subsetp$(a, b) \wedge$ subsetp$(b, c) \rightarrow$ union(diff(b, a), diff$(c, b)) =$ diff(c, a).
$\bullet$cardinality(diff$(a, b)) \leq$ cardinality(a).
$\bullet$intersection(diff$(b, c), a) =$ diff(intersection$(a, b), c)$.
$\bullet$hd(pair$(x, y)) = x$.
$\bullet$tl(pair$(x, y)) = y$.
$\bullet$pairp(pair(x, y)).
$\bullet$pairp$(a) \rightarrow$ pair(hd(a), tl$(a)) = a$.
$\bullet$pair$(x_1, y_1) =$ pair$(x_2, y_2) \leftrightarrow (x_1 = x_2 \wedge y_1 = y_2)$.
$\bullet$functionp$(f) \rightarrow$ functionp(except$(f, x, v))$.
$\bullet$apply(except$(f, x, v), y) =$ (**if** $x = y$ **then** v **else** apply$(f, y))$.
$\bullet$pairps$(f) \rightarrow$ domain(except$(f, x, v)) =$ scons$(x,$ domain$(f))$.
$\bullet$subsetp(range(except(f, x, v)), scons$(v,$ range$(f)))$.
$\bullet$domain(union$(f, g)) =$ union(domain(f), domain$(g))$.
$\bullet$range(union$(f, g)) =$ union(range(f), range$(g))$.
$\bullet$cardinality(domain$(f)) \leq$ cardinality(f).
$\bullet$cardinality(range$(f)) \leq$ cardinality(f).
$\bullet$functionp$(f) \wedge$ functionp$(g) \wedge$ intersection(domain(f), domain$(g)) =$ NIL
$\rightarrow$ functionp(union$(f, g))$.
$\bullet$domain(restrict$(f, s)) =$ intersection$(s,$ domain$(f))$.
$\bullet$functionp$(f) \wedge$ functionp$(g) \wedge$ intersection(domain(f), domain$(g)) =$ NIL
$\rightarrow$ apply(union$(f, g), x)$
$=$ (**if** mem$(x,$ domain$(f))$ **then** apply(f, x) **else** apply$(g, x))$.
$\bullet$sequencep$(a) \wedge$ sequencep$(b) \wedge$ sequencep(c)
$\rightarrow$ concat(concat$(a, b), c) =$ concat$(a,$ concat$(b, c))$.

Fig. 2. Some Theorems Proved

choice function, `ch`, which admits `=` as a congruence and which selects a member of a non-empty set, without otherwise constraining the choice made. Our `choose` can be used as a witness to introduce `ch`.

It should be noted that `ch` does not enjoy the analogue of the *Weak Choose-Scons Property.* That is, it is impossible to prove from the properties of `ch` above that $\texttt{ch}(\texttt{scons}(e, s))$ is either e or $\texttt{ch}(s)$. For example, `ch` might choose the largest element of the set if the cardinality of the set is odd and the smallest element if the cardinality is even. Such a `ch` would have the required properties and furthermore `ch(scons(1, '(2 3)))` would be `3`, which is neither `1` nor `ch('(2 3))`, which is `2`.

Therefore, even when using only the weak property, we assume more about `choose` than we could about an arbitrary choice function. Most importantly, our `choose` is executable, which means that functions defined in terms of it are also executable. Such functions include `pair`, `hd`, `tl` and `apply`.

5 Behind the Scenes

Two sets are equal if and only if they are subsets of one another. If we *define* set equality this way, then equality, membership and subset are mutually recursive: set equality is defined in terms of subset, membership is iterated set equality, and subset is iterated membership. ACL2 supports mutual recursion. But mutual recursion can be awkward when dealing with induction. To prove an inductive theorem about one function in a clique of mutually recursive functions, one must often prove a conjunction of related theorems about the other functions of the clique. While ACL2 can often manage the proofs, the user must state the conjunction of theorems in order for the conjecture to be strong enough to be provable. We found this often inconvenient, especially in the early going when nothing but the definitions of the functions are available.

We considered many ways around this obstacle. The eventual solution, which was supported by some work done concurrently by Pete Manolios, was to introduce the notion of canonical forms. Without engaging in mutual recursion it is possible to

- define a total order on our objects,
- canonicalize lists so that their elements are presented in this order and without duplicates,
- define set equality to compare sets in canonical form,
- define membership as iterated set equality,
- define subset as iterated membership, and
- prove that two sets are set equal iff they are subsets of one another.

This program actually has at least two interpretations and we explored both. The interpretation initially favored was to keep lists representing sets in canonical form all the time. That is, the basic set constructor, e.g., $\texttt{scons}(e, x)$, inserts e (if it is not already there) at the position in x at which it belongs. This has the powerful attraction that set equality, `=`, is Leibniz identity, $=$.

But we 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. In the end we abandoned this approach and adopted a second interpretation of the program above: lists representing sets are created and manipulated in non-canonical form and are canonicalized only for determining whether two sets are equal. This was quite effective. `Scons` is `cons` (with appropriate treatment of `:UR-CONS`), `union` is essentially `append`, etc. ACL2 is designed to prove theorems about these kinds of functions.

Another question that drew our attention was: what are the "ur-elements" of our set theory? The first attack was to formalize hereditarily finite sets: finite sets built entirely from the empty set. Initially we felt that the details of the set theory were irrelevant to the user, since the high level individuals with which the user would deal — numbers, sequences, functions, etc., — would be abstractly represented. According to this thinking, proofs about these high level individuals would be conducted more or less algebraically, using theorems provided by the set theory book.

However, we found the use of hereditarily finite sets to be too cumbersome.

- Concrete examples of sets representing naturals, pairs, etc., were practically impossible to read and comprehend. Here is 3 in the von Neumann representation of the naturals $\{\{\{\{\}\}\{\}\}\{\{\}\}\{\}\}$. It was hard to test definitions and conjectures.
- The need to embed everything as sets forced ACL2 to spend its resources unraveling the embedding rather than dealing with the gist of the user's problem. This was particularly evident when dealing with arithmetic.

In the final view of this project, we saw the objective as to produce a set theory that was "natural" to ACL2's mode of reasoning, so that its power would be spent at the core of the user's problem, not on getting down there. Arithmetic in our set theory is just ACL2's arithmetic. Arbitrary ACL2 objects can be collected into sets. The set primitives are simple and their definitions are usually easily manipulated to derive clean versions of the algebraic laws of set theory. Because of support for congruences, these laws can then be used in the normal way to manipulate set expressions without regard for how sets are actually represented, provided certain basic conventions are followed. The main conventions are that `=` be used to compare sets and that every time a new set generating function is introduced the appropriate congruence rules are proved establishing that the function admits set equality as a congruence relation.

6 Codified Proof Strategies

The set theory book includes several proof strategies particular to set theory. Such strategies are convenient because they overcome ACL2's lack of quantification.

To prove $\alpha = \beta$, where α and β are two set theory expressions that produce sets (as opposed to ur-elements), it is sometimes convenient to prove that $(e \in \alpha) \leftrightarrow (e \in \beta)$. That is, a set is entirely determined by its elements. This fact may be formalized in set theory as $(\forall e : e \in a \leftrightarrow e \in b) \rightarrow a = b$.

But ACL2 does not have the quantificational power to express this fact directly.[2] We have defined an ACL2 macro (named `defx`) that allows the user to direct the theorem prover to prove a theorem using a specified strategy. When `defx` is used to prove $\gamma \rightarrow \alpha = \beta$ with the "set equivalence" strategy it generates two subgoals, $\gamma \wedge \texttt{mem}(e, \alpha) \rightarrow \texttt{mem}(e, \beta)$ and its symmetric counterpart, and then proves the main theorem by "functional instantiation" [5] of a general theorem. (The general theorem can be described as follows. Suppose that `alpha` and `beta` are two 0-ary functions satisfying the constraint $\texttt{mem}(e, \texttt{alpha}()) \rightarrow \texttt{mem}(e, \texttt{beta}())$. Then $\texttt{subsetp}(\texttt{alpha}(), \texttt{beta}())$ is a theorem.)

Another special strategy, called "functional equivalence," is useful when α and β are functions: prove that applying them produces identical results. Four subgoals are produced, (a) $\texttt{functionp}(\alpha)$, (b) $\texttt{functionp}(\beta)$, (c) $\texttt{domain}(\alpha) = \texttt{domain}(\beta)$, and (d) $\texttt{mem}(e, \texttt{domain}(\alpha)) \rightarrow \texttt{apply}(\alpha, e) = \texttt{apply}(\beta, e)$. Proof obligation (d) could be simplified by dropping the hypothesis; we included it simply because it weakens the proof obligation. The previously mentioned theorem

- $\texttt{sequencep}(a) \wedge \texttt{sequencep}(b) \wedge \texttt{sequencep}(c)$
  $\rightarrow \texttt{concat}(\texttt{concat}(a, b), c) = \texttt{concat}(a, \texttt{concat}(b, c))$

is proved with the functional equivalence strategy. (Sequences in this set theory are functions. We would expect the ACL2 user to prefer to use the native lists, but we formalized sequences-as-functions to test the library.) The two `concat` expressions are equivalent because they yeild the same results when applied to arbitrary indices.

`Defx` is defined in a general way that allows the user to add new strategies. (Hint to ACL2 *cognoscenti*: define each strategy as a macro. `Defx` forms expand to calls of the strategy.) The `defx` form provides a uniform appearance in command files ("books") and allows the continued use of Emacs' tag feature for indexing names. This use of macros is novel to most ACL2 users.

7 Recursive Functions on Sets

The first order nature of ACL2, combined with the absence of quantification, prevents the formalization of set comprehension in its general form. That is, it is impossible in ACL2 to formalize with complete generality such notation as "$\{x \mid \phi(x)\}$." We have implemented some macros to mitigate the problem.

The first step is to consider only notation such as $\{x \mid x \in s \wedge \phi(x)\}$, where s is a (finite) set. It is then possible to map over s with a recursive function to identify the appropriate elements. But because ACL2 is first-order, we cannot

[2] Actually, ACL2 does provide full first-order quantification via `defun-sk`, but that is no more convenient that what we are about to describe.

define a function that takes ϕ as an argument.[3] So the second step is to define a recursive function, f, for any given ϕ, to compute the above set from s.

The most obvious disadvantages of this approach are (a) one must introduce a function symbol f to capture each use of set-builder notation, (b) one must prove "the same" theorems about each such f, and (c) one must prove theorems that relate any two such f's that become entwined in the same problem. Much of this can be handled by macros. We therefore deal with the most fundamental issues here.

The general scheme for defining a recursive function to extract the ϕ subset from a set s is:

Def $f(s)$ = **if** `ur-elementp`(s)
  then `NIL`
  else if $\phi($`scar`$(s))$
    then `scons`$($`scar`$(s), f($`scdr`$(s)))$
    else $f($`scdr`$(s))$.

The test on whether s is an `ur-elementp` is the standard way to enforce the Non-Set Convention. This test recognizes `NIL`, but also all other ur-elements, as the base case of the recursion. They are all treated equivalently.

Otherwise, s is a non-empty set and we define f in terms of `scar`(s) and `scdr`(s). We have not previously mentioned these functions because they are not pure "set theory" functions: they do not admit = as a congruence relation. It is best to think of `scar` and `scdr` as working on a presentation of a set. `Scar` returns the first element presented and `scdr` returns the set containing all the others. But a set may have multiple presentations. For example, `'(1 2) = '(2 1)` but `scar('(1 2))` is 1 while `scar('(2 1))` is 2.

In fact, `ur-elementp`, `scar`, `scdr`, and `scons` are exactly analogous to `atom`, `car`, `cdr`, and `cons`, except that conses marked with `:UR-CONS` are treated as atoms. Ignoring the issue raised by `:UR-CONS`, the definition of f above is

Def $f(s)$ = **if** `atom`(s)
  then `NIL`
  else if $\phi($`car`$(s))$
    then `cons`$($`car`$(s), f($`cdr`$(s)))$
    else $f($`cdr`$(s))$.

It thus computes a list, not a set.[4] It may seem counterintuitive to prefer recursive definitions of set theory functions in terms of functions that expose the underlying representation. But this is a deliberate choice and is perhaps the key discovery made in the project. (We discuss what we call "recursion by `choose`"

[3] Using `apply` we could, of course, define the ACL2 function that takes a set s and a finite predicate f represented as a set, and returns the subset of the former satisfying the latter.

[4] Of course it computes a list: sets are lists in ACL2. More precisely, it presents the set in an order determined by the presentation of its arguments.

in [7]. This recursive scheme is entirely set theoretic in nature but makes inductive proofs a little more awkward because of the issues surrounding `choose` and `scons`.)

The main appeal of using `scar` and `scdr` is that it usually makes it straightforward to prove inductively the fundamental theorems about newly defined recursive set theory functions. Such proofs are generally isomorphic to the proofs of analogous theorems about the analogous list processing functions. The latter kind of theorems are ACL2's "bread and butter." We illustrate a recursive definition of a set in Section 9.

8 The Defmap Macro

We have defined a macro to make it easy to define functions corresponding to two common set builder forms. Each use of the macro not only defines a function but also proves certain theorems about the new function.

Def $f(v_1, \ldots, v_k) =$ **for** x **in** v_i **such_that** ϕ
defines $f(v_1, \ldots, v_k)$ to be $\{x \mid x \in v_i \wedge \phi\}$.
Def $f(v_1, \ldots, v_k) =$ **for** x **in** v_i **map** ϕ
defines $f(v_1, \ldots, v_k)$ to be $\{e \mid \exists x(x \in v_i \wedge e = \phi)\}$.

For example, in the case of the first form above, the lemmas proved about f include that it produces a set, that x is a member of the answer iff x is in v_i and satisfies ϕ, that the answer is a subset of v_i, that the function admits `=` as a congruence, and that `union` and `intersection` (in the i^{th} argument) distribute over f. Analogous theorems are proved about the other form.

9 Example

Set theory is so rich that the book described here barely scratches the surface. A relevant question though is whether we can build on this foundation. In this section we show a complete development of a simple book that defines the powerset of a set and proves two facts about it: that its elements are precisely the subsets of the set and that our definition admits set equality as a congruence. The theorems in this section are proved automatically by ACL2 (given the hints below) after including the set theory book. The book is shown in the abstract syntax used throughout this paper, but every Lisp form in the book is represented somehow below.

`in-package("S")`

Here is the definition of powerset.

Def `scons-to-every`$(e, s) =$ **for** x **in** s **map** `scons`(e, x).
Def `powerset`$(s) =$
  if `ur-elementp`(s)
    then `brace(NIL)`

```
    else union(powerset(scdr(s)),
               scons-to-every(scar(s), powerset(scdr(s)))).
```

`Powerset` *builds a set.*

Lemma
`setp(powerset(s))`.

In fact, it builds a set of sets. But to say that we must define `set-of-setsp` *and prove that it admits* `=` *as a congruence.*

```
Def set-of-setsp(p) =
  if ur-elementp(p)
    then T
    else setp(scar(p)) ∧ set-of-setsp(scdr(p)).
```

We use the standard `defx` *strategy for proving congruence for a predicate.*

Defx ...
  $a = b \rightarrow$ `set-of-setsp`$(a) =$ `set-of-setsp`(b).

`Powerset` *builds a set of sets.*

Lemma
`set-of-setsp(powerset(s))`.

Here is the fundamental fact about membership in `scons-to-every`.

Lemma
```
setp(p) ∧ set-of-setsp(p) ∧ setp(s1)
 → (mem(s1,scons-to-every(e,p))
     ↔
     (mem(e,s1) ∧ (mem(s1,p) ∨ mem(diff(s1,brace(e)),p))))
```

The following function is used to tell ACL2 how to induct in the next theorem. It says: induct on b and assume two inductive hypotheses.

```
Def induction-hint(a,b) =
  if ur-elementp(b)
    then list(a,b)
    else list(induction-hint(a,scdr(b)),
              induction-hint(diff(a,brace(scar(b))),scdr(b))).
```

The `powerset` *contains precisely the subsets. This is our main theorem here.*

Theorem
`setp(e)` $\rightarrow$ `(mem(e,powerset(s))` $\leftrightarrow$ `subsetp(e,s))`.
Hint:: Induct according to `induction-hint(e, s)`.

The next lemma is needed for the final `defx` *command.*

Lemma
`set-of-setsp(s)` $\wedge$ `subsetp(s,powerset(b))` $\wedge$ `mem(e,b)`

$\rightarrow$ `subsetp(scons-to-every(`e, s`),powerset(`b`))`.
Hint: Induct according to `scons-to-every(`e, s`)`.

We use the standard `defx` *strategy for proving congruence for a set builder.*

Defx ...
$a = b \rightarrow$ `powerset(`a`) = powerset(`b`)`.

10 Conclusions

An early version of the finite set theory book is part of the general distribution of ACL2 but this is still a work in progress.

The main application of the set theory book is an ongoing experiment by Pacheco [8], of the Department of Computer Sciences, University of Texas at Austin, involving the translation of proof obligations from Leslie Lamport's TLA [6] into ACL2. TLA is based on set theory and thus requires more than "a little set theory" since the notation used in the TLA model frequently uses set constructor notation. To preserve ACL2's reasoning power, we represent TLA numbers, strings and certain other objects with their ACL2 counterparts rather than with Lamport's (unspecified) set representatives. Thus, the TLA experiment involves a mixture of sets and native ACL2 objects.

The TLA experiment has uncovered a few omitted lemmas about functions in our set theory book. More problematically, it has exposed a wealth of well-developed concepts in set theory that are definable but not defined in our book, such as powersets, cross products between sets, the functions between two sets, etc. Such concepts must not only be defined but the highly interconnected web of theorems linking them must be developed. Finally, the TLA experiment has highlighted the need for more support of set comprehension, a feature which gives set theory much of its expressive power. Pacheco has produced prototype macro definitions providing some additional support but much more remains. The whole issue of "faking" higher order expressions in ACL2's first order language will probably require additional low-level support in the ACL2 system itself, such as extensions to our macro feature and connections between macros and the output routines of ACL2.

Set theory is so expressive and so well developed that such problems are not surprising. But the TLA experiment, so far, has not led us to abandon or seek to change the basic representational decisions discussed here. Indeed, we are encouraged by the experiment's success.

11 Acknowledgments

I thank Pete Manolios for his help in exploring the issues concerning mutual recursion in the definitions of the set theory primitives. I am also grateful to Carlos Pacheco for his willingness to dive into the undocumented set theory

book and begin to use it. Finally, I am grateful to Yuan Yu and Leslie Lamport for helping clarify the goals of the set theory work.

References

1. Grzegorz Bancerek. A model of ZF set theory language. *Journal of Formalized Mathematics*, 1, 1989. `http://mizar.org/JFM/Vol1/zf_lang.html`.
2. M. J. C. Gordon. Higher order logic, set theory or both? In `http://www.cl.cam.ac.uk/~mjcg/papers/holst/index.html`. Invited talk, TPHOLs 96, Turku, Finland, August 1996.
3. M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Press, 2000.
4. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Press, 2000.
5. M. Kaufmann and J S. Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.
6. L. Lamport. The temporal logic of actions. *ACM Trans. on Programming Languages and Systems*, 16(3):872–923, May 1994.
7. J S. Moore. Recursion by choose. In `http://www.cs.utexas.edu/users/moore/-publications/finite-set-theory/recursion-by-choose.lisp`. Department of Computer Sciences, University of Texas at Austin, 2000.
8. Carlos Pacheco. Reasoning about TLA actions. Technical Report CS-TR-01-16, Computer Sciences, University of Texas at Austin, May 2001. `http://www.cs.-utexas.edu/ftp/pub/techreports/tr01-16.ps.Z`.
9. L. C. Paulson. Set theory for verification: I. from foundations to functions. *Journal of Automated Reasoning*, 11:353–389, 1993.
10. L. C. Paulson. Set theory for verification: Ii. induction and recursion. *Journal of Automated Reasoning*, 15:167–215, 1995.