Finite Set Theory in ACL2

J Strother Moore

Department of Computer Sciences
University of Texas at Austin
Office: TAY 4.140A
Email: moore@cs.utexas.edu

http://www.cs.utexas.edu/users/moore

Abstract of this Talk

- *Prelude*: The ACL2 background
- *Sets in Lisp*, e.g., to use to state code properties, invariants, etc.
- *Some of the Issues* in adding sets to ACL2.
- *Some of the Theorems* in our finite set theory package.
- *An Application*: Disk Paxos

An Example of Our Finite Set Theory

```
(implies
 (and (mem (apply phase p) (brace 1 2))
      (= disk'
         (except disk d p (apply block p)))
      (  = diskswritten'
         (except diskswritten p
                 (union (apply diskswritten p)
                        (brace d))))
      (invariant phase p disk ...))
 (invariant phase p disk' ...))
```

Lisp's "Set" Functions

- `member` and `subsetp` ignore order and duplications.
- `(member e x)` checks that `e` is `equal` to some element of `x`.
- `(subsetp x y)` checks that every element of `x` is a `member` of `y`.
- `equal` is primitive.

The Inadequacy of Lisp's "Set" Functions

`(member '(1 2) '((2 1)))` $\Rightarrow$ *false*

`(subsetp '((1 2)) '((2 1)))` $\Rightarrow$ *false*

But

$$\{1,\ 2\}\ \in\ \{\{2,\ 1\}\}$$

$$\{1,\ 2\}\}\ \ \subseteq\ \{\{2,\ 1\}\}$$

We want

- `(set-member e x)` to check that e is `set-equal` to some element of `x`.
- `(set-subsetp x y)` to check that every element of `x` is a `set-member` of `y`.
- `(set-equal x y)` to mean that `x` is a `set-subsetp` of y and *vice versa*.

Some Issues

Mutual Recursion

- (`set-member e x`) checks that e is `set-equal` to some element of `x`.
- (`set-subsetp x y`) checks that every element of `x` is a `set-member` of `y`.
- (`set-equal x y`) means that `x` is a `set-subsetp` of y and *vice versa*.

Discussion

ACL2 can handle mutual recursion, but offers little $^{\text{auto}}$matic support.

Moreover, if `f` and `g` are mutually-recursive, properties of `f` must typically be strengthened to include properties of `g` in order to prove them by mathematical induction (whether using ACL2 or not).

Our Approach:

We explored several approaches and eventually decided to avoid mutual recursion.

We defined `(canonicalize x)` to canonicalize a list representing a set.

```
(defun set-equal (x y)
  (equal (canonicalize x)
         (canonicalize y)))
```

`set-member` and `set-subsetp` are defined by simple recursion.

Issue: Equality v. Set-equality

The ACL2 prover is based on rewriting, e.g., replacement of `equals` by `equals`.

How do we make it replace `set-equals` by `set-equals`?

Our Approach:

Suppose ACL2 has proved that `set-equal` is an *equivalence relation*.

Suppose ACL2 has proved

Theorem

```
(set-equal (set-union a b)
           (set-union b a))
```

Remember: `(set-union a b)` may not be `equal` to `(set-union b a)`.

Theorem

```
(set-equal (set-union a b)
           (set-union b a))
```

How can we get ACL2 to rewrite

(set-member e (set-union α β))

to

(set-member e (set-union β α))

e.g., to use the Theorem above as a rewrite rule even though it does not express an equality?

Answer:

Prove the *congruence* lemma:

```
(implies (set-equal x y)
         (iff (set-member e x)
              (set-member e y)))
```

ACL2 supports *congruenced-based rewriting*.

When the ACL2 rewriter is invoked, it is told to *maintain* a given sense of equivalence.

The *congruence* lemma:

```
(implies (set-equal x y)
         (iff (set-member e x)
              (set-member e y)))
```

gives rise to this congruence table

```
(set-member u v)
                    equal   iff
------------------------------------
 u                  equal   equal
 v                  equal   set-equal
```

Theorem

```
(set-equal (set-union a b)
           (set-union b a))
```

Rewrite

```
(set-member e (set-union α β))
↑
[iff]
              ↑
              [set-equal]

              (set-union a b) ⇉ (set-union b a)
```

Issue: The Ur-Elements

Is everything a set?

E.g., $2 = \{0, 1\}$

or do we wish to allow ACL2 objects in sets,

e.g., $\{1,$ `"Hello world"`, `ILOAD`, `(1 2)`$\}$?

Answer:

We allow arbitrary ACL2 objects to be in sets.

To “embed” our arithmetic into sets greatly diminished the power of the system.

The system spent most of its time converting from the set representation of numbers to the ACL2 representation.

Many numbers, e.g., 2147483648, were impossible to represent.

In our representation, lists are treated as sets by the set theory functions.

`{1, "Hello world", ILOAD, (1 2)}`

is representedby the list

```
(1 "Hello world" ILOAD (:UR-CONS (1 2)))
```

That constant is `set-equal` to

```
("Hello world" 1 ILOAD ILOAD (:UR-CONS (1 2)))
```

Some Other Issues

- names of the set-theory functions (packages)
- set builder notation (macros)
- nondeterministic choice (constrained functions)

See the paper.

Some Theorems (in the "S" package)

```
(= (union a b)(union b a))

(= (union (union a b) c)(union a (union b c)))

 (union a nil) a)

= (card a) (card (union a b)))

ff (mem (choose a) a) (not (emptyp a)))

 (domain (union f g))
 (union (domain f) (domain g)))
```

Application

Carlos Pacheco (recently graduated UT undergraduate) has implemented a translator from (the non-temporal logic fragment of) TLA to ACL2.

He used ACL2 to prove two of the six invariants in the Lamport-Gafni proof of the correctness of the Disk Paxos algorithm.

See his Honors Thesis http://www.cs.utexas.-edu/ftp/pub/techreports/tr01-16.ps.Z.