

Fully Ordered Finite Sets for ACL2

ACL2 Meeting, February 11
Jared Davis

Outline

Motivation for ordered sets

Definition of sets and the order

The non-set convention

New primitives, the primitive level

Reasoning through membership

Automation through computed hints

Reasoning about equality

The outer level

Execution efficiency through MBE

Motivation for Ordered Sets

An interesting project.

“Nice” features:

- Unique representation for sets

- Set equality is equal, no congruence reasoning

Practical consideration: execution efficiency

- Ordered sets: all operations are linear

- Unordered: some quadratic (subset, equality, intersection),
some constant time (insertion)

Motivation for Ordered Sets (2)

Full order rejected for the standard sets library:

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

Defining Sets and their Order

$(\ll a b)$ is a total order, from `misc/total-order`

Irreflexive, transitive, asymmetric, trichotomy

The particular order is not important

Sets: fully ordered true lists.

```
(defun setp (X)
  (declare (xargs :guard t))
  (if (atom X)
      (null X)
      (or (null (cdr X))
          (and (consp (cdr X))
                (<< (car X) (cadr X))
                (setp (cdr X))))))
```

The Non-Set Convention

Set functions will treat non-sets as the empty set.

This Eliminates setp hypotheses from rewrite rules:

$$(\text{in } a \text{ (union } X \ Y)) = (\text{in } a \ X) \vee (\text{in } a \ Y)$$

$$(\text{subset } X \ X)$$

$$a \neq b \rightarrow (\text{delete } a \text{ (insert } b \ X)) = (\text{insert } b \text{ (delete } a \ X))$$

Advantages:

Rewrite rules are more general, apply more often.

Fewer hypotheses to relieve = rewriting is faster.

Rules are “nice” looking, simple.

The Non-Sets Convention (2)

Example: an early version of intersect

```
(defun intersect (X Y)
  (declare (xargs :guard (and (setp X) (setp Y))))
  (if (setp X)
      (cond ((endp X) nil)
            ((in (car X) Y)
             (insert (car X) (intersect (cdr X) Y)))
            (t (intersect (cdr X) Y)))
      nil))
```

Explicit `setp` check avoids keeping elements from a non-set `X`, which would violate the non-sets convention.

The Non-Sets Convention (3)

The primitive list functions do not respect the convention:

$(\text{car } '(1\ 1)) = 1$

$(\text{car } \text{nil}) = \text{nil}$

$(\text{cdr } '(1\ 1)) = (1)$

$(\text{cdr } \text{nil}) = \text{nil}$

$(\text{endp } '(1\ 1)) = \text{nil}$

$(\text{endp } \text{nil}) = \text{t}$

$(\text{len } '(1\ 1)) = 2$

$(\text{len } \text{nil}) = 0$

$(\text{cons } 1\ '(1\ 1)) = (1\ 1\ 1)$

$(\text{cons } 1\ \text{nil}) = (1)$

Result: extra cases into definitions, proofs.

New Primitives

Perhaps reasoning difficulty is due to these extra cases?

Solution: new primitives that behave correctly

`(head X)` – `car`, but `nil` for non-sets.

`(tail X)` – `cdr`, but `nil` for non-sets.

`(empty X)` – `endp`, but `t` for non-sets.

`(cardinality X)` – `len`, but `0` for non-sets.

`(insert a X)` – ordered insert, treats non-set X's as the empty set.

New Primitives (2)

```
(defun empty (X)
  (declare (xargs :guard (setp X)))
  (or (null X)
      (not (setp X))))
```

```
(defun sfix (X)
  (declare (xargs :guard (setp X)))
  (if (empty X)
      nil
      X))
```

```
(defun head (X)
  (declare (xargs :guard (and (setp X) (not (empty X)))))
  (car (sfix X)))
```

```
(defun tail (X)
  (declare (xargs :guard (and (setp X) (not (empty X)))))
  (cdr (sfix X)))
```

New Primitives (3)

```
(defun cardinality (X)
  (declare (xargs :guard (setp X)))
  (if (empty X)
      0
      (1+ (cardinality (tail X)))))
```

```
(defun insert (a X)
  (declare (xargs :guard (setp X)))
  (cond ((empty X) (list a))
        ((equal (head X) a) X)
        ((<< a (head X)) (cons a X))
        (t (cons (head X)
                  (insert a (tail X))))))
```

New Primitives (4)

New version of intersect:

```
(defun intersect (X Y)
  (declare (xargs :guard (and (setp X) (setp Y))))
  (cond ((empty X) (sfix X))
        ((in (head X) Y)
         (insert (head X) (intersect (tail X) Y)))
        (t (intersect (tail X) Y)))
```

The Primitive Level

Primitive level: reasoning about the primitives

New primitives are total replacements for car, cdr, etc.

Definitions must be disabled, so we won't see the list functions.

Boring theorems:

Head and tail decrease acl2-count (for termination proofs)

Sfix, tail, and insert create sets (for guard verification)

Reconstruction of X from its head and tail

Insert non-empty, tail of empty is empty

Sfix cancels from head, tail, insert, empty

Card empty = 0, card (insert a empty) = 1.

The Primitive Level (2)

Emptiness theorems: if (empty X) \wedge (empty Y) then:

Heads, tails, sfixes, and inserts are equal.

Reasoning about set order

Note: nil is greater than everything (<< is changed.)

```
(head (insert a X)) = a           if (<< a (head X))
                    (head x)      otherwise.
```

```
(tail (insert a X)) = (sfix X)      if (<< a (head X))
                    (tail X)        if a = (head X)
                    (insert a (tail X)) otherwise.
```

```
(<< a (head X))  $\Rightarrow$  (<< a (head (tail X)))
```

... other various properties ...

Reasoning through Membership

```
(defun in (a X)
  (declare (xargs :guard (setp X)))
  (and (not (empty X))
       (or (equal a (head X))
           (in a (tail X)))))
```

```
(defun subset (X Y)
  (declare (xargs :guard (and (setp X) (setp Y))))
  (or (empty X)
      (and (in (head X) Y)
           (subset (tail X) Y))))
```

Reasoning through Membership (2)

Given:

```
(defun union (X Y) ...)  
(defthm in-union (in a (union X Y)) = (or (in a X) (in a Y)))
```

Show:

```
(defthm union-subset-X (subset X (union X Y)))
```

Traditional argument:

Choose $a \in X$, arbitrary, fixed. Now $a \in (X \cup Y)$ by in-union.

Since a was chosen arbitrarily, $\forall e : e \in X \Rightarrow e \in (X \cup Y)$.

Recall: $A \subseteq B \Leftrightarrow (\forall e : e \in A \Rightarrow e \in B)$.

So $X \subseteq X \cup Y$, by the definition of subset. ■

Reasoning through Membership (3)

ACL2 argument:

Perhaps we can prove *1 by induction... When applied to the goal at hand the above induction scheme produces the following three nontautological subgoals.

```
Subgoal *1/3 (IMPLIES (AND (NOT (EMPTY X))
                           (NOT (IN (HEAD X) (UNION X Y))))
              (SUBSET X (UNION X Y)))
```

=> T immediately

```
Subgoal *1/2 (IMPLIES (AND (NOT (EMPTY X))
                           (IN (HEAD X) (UNION X Y))
                           (SUBSET (TAIL X) (UNION (TAIL X) Y))))
              (SUBSET X (UNION X Y)))
```

=> T after *1/2'6' and a second induction

```
Subgoal *1/1 (IMPLIES (EMPTY X)
                      (SUBSET X (UNION X Y)))
```

=> T immediately

Reasoning through Membership (4)

How can we write “ $A \subseteq B \Leftrightarrow (\forall e : e \in A \Rightarrow e \in B)$ ” in ACL2?

“ \Rightarrow ” is easy: `(defthm subset-in $x \subseteq y \wedge a \in x \Rightarrow a \in y$)`

“ \Leftarrow ” is not as easy. We basically want to write:

$$[\forall e : e \in A \Rightarrow e \in B] \Rightarrow A \subseteq B$$

No way to write this directly, so consider the contrapositive:

$$\neg [\forall e : e \in A \Rightarrow e \in B] \Rightarrow \neg [A \subseteq B]$$

$$[\exists e : e \in A \wedge e \notin B] \Rightarrow \neg [A \subseteq B]$$

We can express this in ACL2, by introducing a new function.

Reasoning through Membership (5)

Subset-witness answers “ $\exists e : e \in A \wedge e \notin B$ ” ?

It returns the pair: (t e) or (nil nil).

```
(defun subset-witness (X Y)
  (declare (xargs :guard (and (setp X) (setp Y))))
  (cond ((empty X) (list nil nil))
        ((in (head X) Y) (subset-witness (tail X) Y))
        (t (list t (head X)))))
```

Theorems:

1. `(car (subset-witness X Y)) = (not (subset X Y))`

2. If `(car (subset-witness X Y)) ≠ nil`, then:

`(in (cadr (subset-witness X Y)) X)`, and

`(not (in (cadr (subset-witness X Y)) Y))`

Reasoning through Membership (6)

Goal: be able to argue $[\forall e : e \in A \Rightarrow e \in B] \Rightarrow A \subseteq B$.
Subset-witness is used to introduce a generic theorem to do this.

```
(encapsulate
  ((sub) => *)           ; the alleged subset, e.g. "A" above
  ((super) => *)        ; the alleged superset, e.g. "B" above
  ((hyps) => *))        ; hypotheses under which  $[ \forall e : e \in A \Rightarrow e \in B ]$  holds

  (local (defun sub () nil))
  (local (defun super () nil))
  (local (defun hyps () t))

  (defthm membership-constraint
    (implies (hyps)
      (implies (in e (sub)) (in e (super))))))

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

Reasoning through Membership (7)

Theorem: $(\text{hyps}) \Rightarrow (\text{subset } (\text{sub}) (\text{super}))$

Proof: Assume the hypothesis, letting $(\text{hyps}) = \text{t}$.

By the membership constraint, $(\text{in } e (\text{sub})) \Rightarrow (\text{in } e (\text{super}))$ *

Now let $(\text{status}, \text{element}) = (\text{subset-witness } (\text{sub}) (\text{super}))$.

Case 1: $\text{status} = \text{nil}$

Then by (1), $(\text{subset } (\text{sub}) (\text{super}))$, and we are done.

Case 2: $\text{status} \neq \text{nil}$

Then by (2), $(\text{in element } (\text{sub})) \wedge (\text{not } (\text{in element } (\text{super})))$.

But this contradicts *, so this case cannot occur. ■

Reasoning Through Membership (8)

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

We now augment the goal above by adding the hypothesis indicated by the `:USE` hint. The hypothesis can be derived from `SUBSET-BY-MEMBERSHIP` via functional instantiation, provided we can establish the constraint generated. Thus, we now have the two subgoals shown below.

```
Subgoal 2 (IMPLIES (IMPLIES T (SUBSET X (UNION X Y)))
                  (SUBSET X (UNION X Y))).
```

But we reduce the conjecture to `T`, by case analysis.

```
Subgoal 1 (IMPLIES (IN DO-NOT-REUSE-THIS-NAME-1 X)
                  (IN DO-NOT-REUSE-THIS-NAME-1 (UNION X Y))).
```

But simplification reduces this to `T`, using the `:rewrite` rule `UNION-IN` and the `:type-prescription` rule `IN`.

Q.E.D.

Automation Through Computed Hints

Likely candidates are tagged to trigger the strategy:

```
(defthm pick-a-point-subset-strategy
  (implies (and (syntaxp (rewriting-goal-lit x mfc state))
                (syntaxp (rewriting-conc-lit 'subset x y mfc state)))
            (equal (subset X Y) (pick-a-point-trigger X Y))))
```

Syntaxp here: (1) do not apply if backchaining,
(2) do not rewrite hypotheses.

The Tag:

```
(defun pick-a-point-trigger (X Y) (subset X Y))
```

The new generic theorem:

```
(defthm subset-by-membership
  (implies (sbm-hyps)
            (pick-a-point-trigger (sbm-sub) (sbm-super))))
```

Automation Through Computed Hints (2)

Computing the hint: (simplified)

... several functions omitted ...

```
(defun pick-a-point-subset-hint (id clause world stable)
  (declare (xargs :mode :program))
  (if (not stable)
      nil
      (let ((harvest (harvest-function clause 'pick-a-point-trigger))))
        (if (not harvest)
            nil
            (let ((hints `(:use ,(build-subset-hints ...))))
              (prog2$
                (cw *pick-a-point-docs* ... hints)
                hints))))))
```

Automation Through Computed Hints (3)

```
(DEFTHM UNION-SUBSET-X (SUBSET X (UNION X Y))) ; Actual theorem, no hints
```

This simplifies, using the `:definition` `SYNP` and the `:rewrite` rule `PICK-A-POINT-SUBSET-STRATEGY`, to

```
Goal '  
(PICK-A-POINT-TRIGGER X (UNION X Y)).
```

Note: Pick-a-Point Proof of Subset

It looks like a pick-a-point strategy might be a good way of proving this conjecture. So, we introduce the following hint:

```
("Goal'" :USE  
  (:FUNCTIONAL-INSTANCE SUBSET-BY-MEMBERSHIP  
    (SBM-HYPS (LAMBDA NIL T))  
    (SBM-SUB (LAMBDA NIL X))  
    (SBM-SUPER (LAMBDA NIL (UNION X Y))))))
```

If this fails, you should check that for each instance of `subset-by-membership` above, you can prove the following:

```
(sbm-hyps) => [(in a (sbm-sub)) => (in a (sbm-super))]
```

The pick-a-point method is often a good way to prove subsets between sets. However, if this is not the strategy you want to use, you can disable it by explicitly giving the following hint: `(in-theory (disable pick-a-point-subset-strategy))`

Automation Through Computed Hints (4)

[Note: A hint was supplied for our processing of the goal below. Thanks!]

Goal'

```
(PICK-A-POINT-TRIGGER X (UNION X Y)).
```

We now augment the goal above by adding the hypothesis indicated by the :USE hint. The hypothesis can be derived from SUBSET-BY-MEMBERSHIP via functional instantiation, provided we can establish the constraint generated. Thus, we now have the two subgoals shown below.

Subgoal 2

```
(IMPLIES (IMPLIES T (PICK-A-POINT-TRIGGER X (UNION X Y)))
          (PICK-A-POINT-TRIGGER X (UNION X Y))).
```

But we reduce the conjecture to T, by case analysis.

Subgoal 1

```
(IMPLIES (IN DO-NOT-REUSE-THIS-NAME-1 X)
          (IN DO-NOT-REUSE-THIS-NAME-1 (UNION X Y))).
```

But simplification reduces this to T, using the :rewrite rule UNION-IN and the :type-prescription rule IN.

Q.E.D.

Reasoning About Equality

We also want to be able to do proofs of equality via double containment. The following is a theorem:

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

We will do the same thing we did with subset. We need a new trigger, a new encapsulate, and a new computed hint.

```
(defun double-containment-trigger (X Y)
  (equal X Y))
```

```
(defthm double-containment-strategy
  (implies (and (setp X)
                (setp Y)
                (syntaxp (rewriting-goal-lit x mfc state))
                (syntaxp (rewriting-conc-lit 'equal x y mfc state)))
           (equal (equal X Y) (double-containment-trigger X Y))))
```

Reasoning About Equality (2)

```
(encapsulate
  (( (ebm-lhs) => *)
    ( (ebm-rhs) => *)
    ( (ebm-hyps) => *) )

  (local (defun ebm-lhs () nil))
  (local (defun ebm-rhs () nil))
  (local (defun ebm-hyps () t))

  (defthm X-set-constraint (implies (ebm-hyps) (setp (ebm-lhs))))
  (defthm Y-set-constraint (implies (ebm-hyps) (setp (ebm-rhs))))

  (defthm membership-constraint-equal
    (implies (ebm-hyps)
      (equal (in do-not-reuse-this-name-2 (ebm-lhs))
              (in do-not-reuse-this-name-2 (ebm-rhs))))))

  (defthm equal-by-membership
    (implies (ebm-hyps)
      (double-containment-trigger (ebm-lhs) (ebm-rhs))))
```


Reasoning About Equality (4), example

```
(defthm intersect-insert-X
  (implies (not (in a Y))
    (equal (intersect (insert a X) Y)
      (intersect X Y))))
```

This simplifies, using the `:rewrite` rule `INTERSECT-SYMMETRIC`, to

```
Goal' (IMPLIES (NOT (IN A Y))
  (EQUAL (INTERSECT Y (INSERT A X))
    (INTERSECT X Y))).
```

This simplifies, using the `:definition` `SYNP` and the `:rewrite` rules `DOUBLE-CONTAINMENT-STRATEGY` and `INTERSECT-SET`, to

```
Goal'' (IMPLIES (NOT (IN A Y))
  (DOUBLE-CONTAINMENT-TRIGGER (INTERSECT Y (INSERT A X))
    (INTERSECT X Y))).
```

Note: Double Containment Proof

It looks like double containment might be a good strategy for proving this conjecture. So, we introduce the following hint:

```
("Goal''" :USE
  ( (:FUNCTIONAL-INSTANCE EQUAL-BY-MEMBERSHIP
    (EBM-HYPS (LAMBDA NIL (NOT (IN A Y))))
    (EBM-LHS (LAMBDA NIL (INTERSECT Y (INSERT A X))))
    (EBM-RHS (LAMBDA NIL (INTERSECT X Y))))))
```

Reasoning About Equality (5), example

If this fails, you should check that for each instance of equal-by-membership above, you can prove the following:

```
(ebm-hyps) => (setp (ebm-lhs))
(ebm-hyps) => (setp (ebm-rhs))
(ebm-hyps) => (in a (ebm-lhs)) = (in a (ebm-rhs))
```

Double containment is almost always the best way to prove equalities between sets. However, if this is not the strategy you want to use, you can disable it by explicitly giving the following hint: `(in-theory (disable double-containment-strategy))`

[Note: A hint was supplied for our processing of the goal below. Thanks!]

```
Goal'' (IMPLIES (NOT (IN A Y))
              (DOUBLE-CONTAINMENT-TRIGGER (INTERSECT Y (INSERT A X))
                                           (INTERSECT X Y))).
```

We now augment the goal above by adding the hypothesis indicated by the `:USE` hint. The hypothesis can be derived from `EQUAL-BY-MEMBERSHIP` via functional instantiation, provided we can establish the three constraints generated. Thus, we now have the two subgoals shown below.

Reasoning About Equality (6), example

Subgoal 2

```
(IMPLIES (AND (IMPLIES (NOT (IN A Y))
                    (DOUBLE-CONTAINMENT-TRIGGER (INTERSECT Y (INSERT A X))
                                                  (INTERSECT X Y)))
          (NOT (IN A Y)))
 (DOUBLE-CONTAINMENT-TRIGGER (INTERSECT Y (INSERT A X))
                              (INTERSECT X Y))).
```

But we reduce the conjecture to T, by case analysis.

Subgoal 1

```
(IMPLIES (NOT (IN A Y))
 (EQUAL (IN DO-NOT-REUSE-THIS-NAME-2 (INTERSECT Y (INSERT A X)))
        (IN DO-NOT-REUSE-THIS-NAME-2 (INTERSECT X Y))))
```

[... 5-way case split, all prove automatically in one simplification ...]

But simplification reduces this to T, using primitive type reasoning.

Q.E.D.

Reasoning about Insert (without using order)

```
(defun insert (a X) ; recall insert's definition
  (declare (xargs :guard (setp X)))
  (cond ((empty X) (list a))
        ((equal (head X) a) X)
        ((<< a (head X)) (cons a X))
        (t (cons (head X) (insert a (tail X))))))

(defun weak-insert-induction (a X) ; order-less induction scheme
  (declare (xargs :guard (setp X)))
  (cond ((empty X) nil)
        ((in a X) nil)
        ((equal (head (insert a X)) a) nil)
        (t (list (weak-insert-induction a (tail X))))))

(in-theory (disable (:induction insert))) ; use new scheme by default
(defthm use-weak-insert-induction t
  :rule-classes ((:induction
                  :pattern (insert a X)
                  :scheme (weak-insert-induction a X))))

(implies (and (not (in a X)) ; support lemmas
              (not (equal (head (insert a X)) a)))
          (and (equal (head (insert a X)) (head X)))
              (equal (tail (insert a X)) (insert a (tail X)))))

(implies (and (not (in a X))
              (equal (head (insert a X)) a))
          (equal (tail (insert a X)) (sfix X)))
```

The Outer Level

Primitive Level
Respect Non-Set Convention

Define primitives (setp, empty, head, tail, ...)
Provide theorems about them, set order.
Disable all primitives.

Membership Level
Do Away with Set Order

Define in and subset.
Develop pick-a-point and double-containment strategies.
Provide non-order based insert induction.
Disable all order-based theorems.

Outer Level
“User Space”

Define delete, union, intersect, difference, [...]
Develop theorems about them using the membership-level's reasoning system.

The Outer Level (2)

```
(defun delete (a X)
  (declare (xargs :guard (setp X)))
  (cond ((empty X) nil)
        ((equal a (head X)) (tail X))
        (t (insert (head X) (delete a (tail X))))))
```

```
(defun union (X Y)
  (declare (xargs :guard (and (setp X) (setp Y))))
  (if (empty X)
      (sfix Y)
      (insert (head X) (union (tail X) Y))))
```

```
(defun intersect (X Y)
  (declare (xargs :guard (and (setp X) (setp Y))))
  (cond ((empty X) (sfix X))
        ((in (head X) Y)
         (insert (head X) (intersect (tail X) Y)))
        (t (intersect (tail X) Y))))
```

```
(defun difference (X Y)
  (declare (xargs :guard (and (setp X) (setp Y))))
  (cond ((empty X) (sfix X))
        ((in (head X) Y) (difference (tail X) Y))
        (t (insert (head X) (difference (tail X) Y)))))
```

The Outer Level (3)

“The New Method”

1. Introduce your set function using the primitives
2. Show it produces a set
3. Show its membership property
4. Automatically prove everything else with double containment.

60+ theorems in the file, 7 hints total. 3 from a bug and 2 apiece for the following:

```
(implies (subset X Y)
         (<= (cardinality X) (cardinality Y)))
```

```
(implies (and (setp X)
              (setp Y)
              (subset X Y)
              (equal (cardinality X) (cardinality Y)))
         (equal (equal X Y) t))
```

Execution Efficiency with MBE

Functions as presented are nowhere near linear. Even the primitives are horrible:

```
(defun empty (X)
  (declare (xargs :guard (setp X)))
  (or (null X) (not (setp X))))
```

```
(defun sfix (X)
  (declare (xargs :guard (setp X)))
  (if (empty X) nil X))
```

```
(defun head (X)
  (declare (xargs :guard (and (setp X) (not (empty X)))))
  (car (sfix X)))
```

```
(defun tail (X)
  (declare (xargs :guard (and (setp X) (not (empty X)))))
  (cdr (sfix X)))
```

Execution Efficiency with MBE (2)

When guards are satisfied, primitives can be implemented efficiently:

```
(empty X) = (null X)
(sfix X) = X
(head X) = (car X)
(tail X) = (cdr X)
(cardinality X) = (len X)
```

MBE: use efficient version for execution, nice version for reasoning. Syntax example:

```
(defun empty (X)
  (declare (xargs :guard (setp X)))
  (mbe :logic (or (null X) (not (setp X)))
       :exec  (null X)))
```

Benefit: non-set convention respected without losing efficiency.

Execution Efficiency with MBE (3)

```
(defun fast-subset (X Y) ; linear in len(Y)
  (declare (xargs :guard (and (setp X) (setp Y))))
  (cond ((empty X) t)
        ((empty Y) nil)
        ((<< (head X) (head Y)) nil)
        ((equal (head X) (head Y)) (fast-subset (tail X) (tail Y)))
        (t (fast-subset X (tail Y)))))

(defun fast-union (X Y) ; linear in len(X) + len(Y)
  (declare (xargs :guard (and (setp X) (setp Y))))
  (cond ((empty X) Y)
        ((empty Y) X)
        ((equal (head X) (head Y))
         (cons (head X) (fast-union (tail X) (tail Y))))
        ((<< (head X) (head Y))
         (cons (head X) (fast-union (tail X) Y)))
        (t (cons (head Y) (fast-union X (tail Y))))))

(defun fast-intersect (X Y) ; linear in len(X) + len(Y)
  (declare (xargs :guard (and (setp X) (setp Y))))
  (cond ((empty X) nil)
        ((empty Y) nil)
        ((equal (head X) (head Y))
         (cons (head X)
               (fast-intersect (tail X) (tail Y))))
        ((<< (head X) (head Y))
         (fast-intersect (tail X) Y))
        (t (fast-intersect X (tail Y)))))
```

Execution Efficiency with MBE (4)

```
(defun fast-difference (X Y) ; linear in len(X) + len(Y)
  (declare (xargs :guard (and (setp X) (setp Y))))
  (cond ((empty X) nil)
        ((empty Y) X)
        ((equal (head X) (head Y))
         (fast-difference (tail X) (tail Y)))
        ((<< (head X) (head Y))
         (cons (head X) (fast-difference (tail X) Y)))
        (t (fast-difference X (tail Y)))))
```

Difficult equivalence proofs.

Strategy: apply “the new method” to show that the fast versions create sets and have the expected membership properties.

Even this is difficult: lots of cases, 12-way induction splits!

More hints than theorems in the file.

Certification time: fast.cert = 24s, entire build = 44s.

Execution Efficiency with MBE (5)

Timing: Insert 100,000 random integers [0, 10000): *drawbridge*

Set Size	Baseline	SETS::Insert	Scons (art. verified guards)
10,	8,	11,	9
25,	8,	14,	8
50,	8,	20,	9
100,	8,	32,	8
250,	8,	67,	9
500,	9,	128,	8
1000,	8,	247,	8
2500,	8,	627,	9
5000,	8,	1112,	8

Timing: 10,000 Unions of two random sets of integers [0, 10000): *stanchion*

Set Size	Baseline	SETS::Union	S::union (art. verified guards)
10,	10,	20,	14
25,	8,	38,	24
50,	7,	68,	41
100,	8,	122,	72
250,	8,	302,	176
500,	8,	330,	329
1000,	8,	246,	294
2500,	8,	318,	248
5000,	9,	339,	238

Execution Efficiency with MBE (6)

Timing: 10,000 intersects of two random sets of integers [0, 1000): *brunehilda*

Set Size	Baseline	SETS::intersect	S::intersection
10,	18,	28,	202
25,	14,	49,	1172
50,	14,	86,	4565
100,	14,	192,	[condor-police, >10k]
250,	14,	362,	[condor-police, >10k]
500,	14,	537,	[condor-police, >10k]
1000,	14,	771,	[condor-police, >10k]

Timing: 10,000 differences of two random sets of integers, [0, 1000): *brunehilda*

Set Size	Baseline	SETS::difference	S::diff
10,	18,	22,	75,
25,	14,	32,	369,
50,	14,	49,	1492,
100,	14,	81,	5505,
250,	14,	163,	[condor-police, >10k]
500,	14,	247,	[condor-police, >10k]
1000,	14,	337,	[condor-police, >10k]

Fair comparison of equality / subset?

Extensions and Future Ideas

Typed Sets: sets whose elements all satisfy some predicate

```
(SETS::introduce-typed-set integerp integer-setp)
```

Just a macro that expands into some functions and theorems. Proofs are done by explicit functional instantiation of generic theorems, for example:

```
(defthm typed-set-insert
  (equal (typed-setp (insert a X))
    (and (recognizer a)
      (or (typed-setp X)
        (empty X)))))
```

```
(defthm typed-set-intersect
  (implies (or (typed-setp X)
    (typed-setp Y))
    (typed-setp (intersect X Y))))
```

Idea: could we use MBE to somehow provide an equivalent but faster order for typed sets? E.g., just use “<” for integer sets?

Extensions and Future Ideas (2)

Robert – extend computed hints to be able to invoke the theorem prover.

Idea is:

1. Spot double containment / pick a point targets as normal.
2. Test if the theorem prover will succeed using the strategies.
3. If successful, give the hint and let the proof succeed.
Else, tell the user you tried it (and how to disable it to avoid the efficiency hit), then go on with the proof attempt without the hint.

A sort of “user-specified heuristic” search capability, where you can try out a few strategies in hopes of finding one that works.