Collection Types

Sequences, Arrays, Sets, and Bags

Jeffrey Maddalon[1]

j.m.maddalon@nasa.gov

NASA

PVS Class, 2012

[1] heavily based on a previous talk by Rick Butler

Outline

1. Membership Collections
   - Sets
   - Proving with Sets
   - Sets in Type Theory
   - Choose
   - Finite Sets
   - Bags
2. Object Collections
   - Sequence
   - Bounded Array
   - Array Operations
   - Finite Sequences

Membership Collections

These "membership" collections are available in PVS

- Sets `[T -> bool]`
- Finite Sets `[(is_finite) -> bool]`
- Bags (aka multisets) `[T -> nat]`
- Finite Bags `[(is_finite) -> nat]`

Sets in PVS

- A set is just a predicate (i.e., a function into `bool`):
```
letters:  TYPE = {a,b,c,d,e,f}
S: set[letters] = ...
```
- For example, if `S` represents:
```
S(a) --> TRUE      S(b) --> TRUE
S(c) --> FALSE     S(d) --> TRUE
S(e) --> TRUE      S(f) --> FALSE
```
  Then, it can be specified in PVS as:
```
S: set[letters] = (LAMBDA (x:  letters):
                     (x=a) OR (x=b) OR (x=d) OR (x=e))
```
  Alternatively, one could write:
```
S: set[letters] = { x:  letters | (x=a) OR (x=b) OR
                                  (x=d) OR (x=e) }
```
- But, there is no PVS set constructor:
```
S:set[letters] = { a, b, d, e }
```
  - However, this form can be used for *type* construction (see above)

The Sets Theory in Prelude

The `sets[T: TYPE]` theory is defined in the prelude:

```
sets [T: TYPE]: THEORY
BEGIN
  set: TYPE = [T -> bool]

  x, y: VAR T
  a, b, c: VAR set
  p: VAR PRED[T]

 member(x, a): bool = a(x)

 empty?(a): bool = (FORALL x: NOT member(x, a))
 emptyset: set = {x | false}
 nonempty?(a): bool = NOT empty?(a)
 fullset: set = {x | true}

 subset?(a, b): bool = (FORALL x: member(x, a) => member(x, b))
 strict_subset?(a, b): bool = subset?(a, b) & a /= b
...
```


The Sets Theory in Prelude (cont'd)

PVS Name	meaning
`union(a,b)`	everything in *a* or *b*
`intersection(a,b)`	anything in both *a* and *b*
`disjoint?(a,b)`	do *a* and *b* share any elements
`difference(a,b)`	all members of *a* that are not in *b*
`singleton(x)`	constructs set with element *x*
`add(x,a)`	add element *x* to *a*
`remove(x,a)`	remove element *x* from *a*
`choose(a)`	choose an arbitrary element of *a*
`rest(a)`	the set *a* without `choose(a)`

Some important lemmas about sets

Useful lemmas about sets in the `sets_lemmas` theory in the prelude

```
emptyset_is_empty?: LEMMA empty?(a) IFF a = emptyset
subset_transitive : LEMMA subset?(a, b) AND subset?(b, c)
                             IMPLIES subset?(a, c)
subset_emptyset   : LEMMA subset?(emptyset, a)
union_empty       : LEMMA union(a, emptyset) = a
union_subset1     : LEMMA subset?(a, union(a, b))

intersection_empty: LEMMA intersection(a, emptyset) = emptyset
distribute_intersection_union: LEMMA intersection(a, union(b, c))
                  = union(intersection(a, b), intersection(a, c))
distribute_union_intersection: LEMMA union(a, intersection(b, c))
                         = intersection(union(a, b), union(a, c))

member_add      : LEMMA member(x, a) IMPLIES add(x, a) = a
choose_member   : LEMMA NOT empty?(a) IMPLIES member(choose(a), a)
choose_singleton: LEMMA choose(singleton(x)) = x
```


Using Set Lemmas

- Using the lemma:

  `union_commutative:  LEMMA union(a, b) = union(b, a)`

- Usually, one must include the parent type:

  `(lemma "union_commutative[nat])"`

- Sometimes you can get away with

  `(rewrite "union_commutative)"`

  but not always!

Set Union and Intersection

$x \in B \cup C \equiv$ `union(B, C)(x) = B(x) OR C(x)`

$x \in B \cap C \equiv$ `intersection(B, C)(x) = B(x) AND C(x)`

Thus operations on sets can be reduced to propositional formulas by set membership, i.e.,

- `union(B, C)` is a function
- `union(B, C)(x)` is a propositional formula

Proving with subset?

```
  |-------
{1}    subset?(B, C)

Rule? (expand "subset?")

  |-------
{1}    (FORALL (x: int): member(x, B) => member(x, C))

Rule? (skosimp*)

  |-------
{1}    member(x!1, B) => member(x!1, C)

Rule? (expand "member")

  |-------
{1}    (B(x!1) => C(x!1))
```

This can get a little tedious, is there another way?

Interlude: Auto Rewriting

```
  |-------
{1}   factorial(5) > 100

Rule? (rewrite "factorial")
nn gets 5, Rewriting using factorial, matching in *,

  |-------
{1}   5 * factorial(4) > 100

Rule? (auto-rewrite "factorial")

  |-------
[1]   5 * factorial(4) > 100

Rule? (assert)
factorial rewrites factorial(1) to 1
factorial rewrites factorial(2) to 2
factorial rewrites factorial(3) to 6
factorial rewrites factorial(4) to 24
Simplifying, rewriting, and recording with decision procedures,
Q.E.D.
```


Set Auto-rewriting

An automatic reduction of set operations can be facilitated through use of

`(install-rewrites :defs t)`

which installs all the definitions used directly or indirectly in the original statement as auto-rewrite rules

Another form is

`(auto-rewrite-theory "sets[T]")`

which installs an entire theory as auto-rewrites.

- Be careful with this one. If the theory contains a commutativity result, this will cause an endless loop.

install-rewrites

```
{-1}  subset?(A!1, C!1)
  |-------
{1}   subset?(union(A!1, B!1), union(C!1, B!1))

Rule? (install-rewrites :defs t)
Rewriting relative to the theory: sets[real],
this simplifies to:
set_rewrite2 :

[-1]  subset?(A!1, C!1)
  |-------
[1]   subset?(union(A!1, B!1), union(C!1, B!1))
```


install-rewrites (cont'd)

```
Rule? (assert)
member rewrites member(x, A!1) to A!1(x)
member rewrites member(x, C!1) to C!1(x)
subset? rewrites subset?(A!1, C!1) to FORALL (x: real): A!1(x) => C!1(x)
member rewrites member(x, A!1) to A!1(x)
member rewrites member(x, B!1) to B!1(x)
union rewrites union(A!1, B!1)(x) to A!1(x) OR B!1(x)
member rewrites member(x, union(A!1, B!1)) to A!1(x) OR B!1(x)
member rewrites member(x, C!1) to C!1(x)
union rewrites union(C!1, B!1)(x) to C!1(x) OR B!1(x)
member rewrites member(x, union(C!1, B!1)) to C!1(x) OR B!1(x)
subset? rewrites subset?(union(A!1, B!1), union(C!1, B!1))
  to FORALL (x: real): A!1(x) OR B!1(x) => C!1(x) OR B!1(x)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
set_rewrite2 :

{-1}  FORALL (x: real): A!1(x) => C!1(x)
  |-------
{1}   FORALL (x: real): A!1(x) OR B!1(x) => C!1(x) OR B!1(x)
```

an easily proved formula.

How?

Set Equality

- To prove that two sets are equal we must use function extensionality:

$$f = g \;\; IFF \;\; \forall x : f(x) = g(x)$$

because sets are just functions into bools (i.e., predicates)

- `(decompose-equality)` will do the trick
- `(apply-extensionality)` is a less powerful version

Set Equality: Example

```
A: set[real] = { x: real | (x=1) OR (x=2) OR (x=3) }

equality: LEMMA A = add(1,add(2,singleton(3)))
```

```
ill_ext :

  |-------
{1}    A = add(1, add(2, singleton(3)))

Rule? (decompose-equality)

  |-------
{1}    A(x!1) = add(1, add(2, singleton(3)))(x!1)

Rule? (install-rewrites :defs t)

  |-------
[1]    A(x!1) = add(1, add(2, singleton(3)))(x!1)
```

Set Equality: Example (cont'd)

```
Rule? (assert)
A rewrites AA(x!1)
  to (x!1 = 1) OR (x!1 = 2) OR (x!1 = 3)
singleton rewrites singleton(3)(x!1)
  to  x!1 = 3
member rewrites member(x!1, singleton(3))
 to  x!1 = 3
add rewrites add(2, singleton(3))(x!1)
  to  2 = x!1 OR x!1 = 3
member rewrites member(x!1, add(2, singleton(3)))
  to  2 = x!1 OR x!1 = 3
add rewrites add(1, add(2, singleton(3)))(x!1)
  to  1 = x!1 OR 2 = x!1 OR x!1 = 3
Simplifying, rewriting, and recording with decision procedures,

  |-------
{1}    (((x!1 = 1) OR (x!1 = 2) OR (x!1 = 3)) =
                 (1 = x!1 OR 2 = x!1 OR x!1 = 3))

Rule? (ground)
No change on: (ground)
```

What happened here? Any suggestions?

Set Equality: Example (cont'd)

We need to convert the equality of two formulas into a propositional formula.

```
Rule? (iff)
Converting top level boolean equality into IFF form,
Converting equality to IFF,
this simplifies to:
ill_ext :

  |-------
{1}    (x!1 = 1) OR (x!1 = 2) OR (x!1 = 3) IFF
                  1 = x!1 OR 2 = x!1 OR x!1 = 3

Rule? (ground)
Applying propositional simplification and decision procedures,
Q.E.D.
```

Big Warning

Given

```
T_100: TYPE = { n: nat | 0 <= n AND n <= 100 }
T_125: TYPE = { n: nat | 25 <= n AND n <= 125 }
```

Then

{ t: T_100 | t = 50 } $\neq$ { t: T_125 | t = 50 }

Why?

Big Warning (cont'd)

Given

```
T_100: TYPE = { n: nat | 0 <= n AND n <= 100 }
T_125: TYPE = { n: nat | 25 <= n AND n <= 125 }
```

When we ask are these two sets equal

```
{ t:T_100 | t = 50 }     { t: T_125 | t = 50 }
```

We are really asking are these two functions equal?

```
(LAMBDA (t:T_100): t = 50)     (LAMBDA (t: T_125): t = 50)
```

THE DOMAINS ARE NOT EQUAL!

- The `decompose-equality` strategy requires the domains to be the same
- Even though in set theory semantics they represent the same set

Thoughts About Sets in Type Theory

Type theory offers several advantages over set theory

- Avoids the classic paradoxes in an intuitive way.
- Type checking uncovers errors
- More "natural" for people used to (most) programming languages

However, there are some disadvantages:

- Sets with the same elements but different domains are different.
  - The emptyset is not unique
    (i.e., `emptyset[T1]` and `emptyset[T2]` are not identical)
- There are different set operations for each basic element type. In other words, `card[T1]` is not the same function as `card[T2]`.

Back to "Big Warning"

If you give PVS

```
T_100: TYPE = { n: nat | 0 <= n AND n <= 100 }
l1: LEMMA {t:T_100 | t = 50} = {t: nat | n = 50}
```

it will recognize the domain mismatch and interpret this as

```
       |-------
{1}     {t: T_100 | t = 50} = restrict({n: nat | n = 50})
```

where `restrict` is defined in the prelude as:

```
restrict [T: TYPE, S: TYPE FROM T, R: TYPE]: THEORY
  BEGIN
    f: VAR [T -> R]
    s: VAR S

    restrict(f)(s): R = f(s)
    CONVERSION restrict
  END restrict
```

This CONVERSION helps here, but there are plenty of cases it doesn't.

The Moral Of the Story

MORAL: Define sets over the PARENT TYPE unless there is a very good reason not to.

USE

```
{ n: nat | P(n) AND n <= 100 }
```

RATHER THAN

```
T_100: TYPE = { n: nat | n <= 100 }
{ t:T_100 | P(t) }
```

This will keep all the domains the same.

Choose Function

- The `choose` function returns an arbitrary element of a nonempty set:
  `choose(p: (nonempty?)): (p) = epsilon(p)`
- An empty set will cause an unprovable TCC.
- If the set is potentially empty, one should use `epsilon` directly.
- `epsilon` produces an element in the set if one exists, and otherwise produces an arbitrary element of the type.
  - The parent type of the set must be nonempty.
- The function `epsilon` is defined as follows:

```
epsilons [T: NONEMPTY_TYPE]: THEORY
BEGIN
  p: VAR pred[T]
  x: VAR T

  epsilon(p): T

  epsilon_ax: AXIOM (EXISTS x: p(x)) => p(epsilon(p))
```

Choose Function: Additional Thoughts

- `choose` returns an arbitrary element, not a random element, thus if `x = choose(a)` and `y = choose(a)`, then `x` always equals `y`
- It would have been nice if `choose` had been defined without a body:
  `choose(p: (nonempty?)): (p)`
  since all of the properties needed are implicit in the return type.
  - If the body were not present, `choose` would not expand when using `(grind)` or `(auto-rewrite-theory "sets[nat]")`
  - Recommendation:

```
(auto-rewrite-theory "sets[nat]" :exclude "choose")
(grind :exclude "choose")
(install-rewrites :DEFS T :EXCLUDE "choose")
```


Motivation For Finite Sets

We would like to have to following functions defined over sets:

1. Cardinality function
2. Minimum and maximum over a set
3. Summation over a set

and the ability to perform set induction.

Basic Definitions

Let's define a predicate that indicates when a set is finite:

```
is_finite(S): bool = (EXISTS N, (f:  [(S)->below[N]]): injective?(f))
```

- So a set is finite if there is a one-to-one function between the members of the set and a finite set of natural numbers.
- The user is free to pick any `N` that is convenient and not necessarily the smallest.
- `injective?` is defined in the PVS prelude as:

```
functions [D, R: TYPE]: THEORY
  f, g: VAR [D -> R]
  x, x1, x2: VAR D
  y: VAR R

  injective?(f): bool = (FORALL x1, x2: (f(x1) = f(x2) => (x1 = x2)))

  surjective?(f): bool = (FORALL y: (EXISTS x: f(x) = y))

  bijective?(f): bool = injective?(f) & surjective?(f)
```


The type finite_set

```
finite_set:  TYPE = (is_finite) CONTAINING emptyset[T]
```

A nonempty finite set is defined as follows:

```
non_empty_finite_set:  TYPE = {s:  finite_set | NOT empty?(s)}
```

The declaration of a finite set variable:

```
IMPORTING finite_sets
S: VAR finite_set[T]
```

REMINDER:

`(is_finite)` is an abbreviation for the type

```
{t:  setof[T] | is_finite(t)}
```

Finite Set Operations

- The standard set operations are defined in the prelude theory, `sets`
- Because `finite_set` is a subtype of `set`, all of the operations on the `set` type are inherited by the `finite_set` type.

The set operations preserve finiteness:

```
A,B: VAR finite_sets

finite_union:        LEMMA is_finite(union(A,B))
finite_intersection: LEMMA is_finite(intersection(A,B))
finite_difference:   LEMMA is_finite(difference(A,B))

finite_add:          LEMMA is_finite(add(x,A))
finite_remove:       LEMMA is_finite(remove(x,A))

finite_subset:       LEMMA subset?(S,A) IMPLIES is_finite(S)

finite_singleton:    LEMMA is_finite(singleton(x))
finite_empty:        LEMMA is_finite(emptyset[T])
finite_rest:         LEMMA is_finite(rest(A))
```


Judgements for Finite Sets - for Reference

```
finite_singleton: JUDGEMENT singleton(x) HAS_TYPE finite_set

finite_union    : JUDGEMENT union(A, B) HAS_TYPE finite_set
finite_intersec1: JUDGEMENT intersection(s, A) HAS_TYPE finite_set
finite_intersec2: JUDGEMENT intersection(A, s) HAS_TYPE finite_set

nonempty_finite_is_nonempty: JUDGEMENT
      non_empty_finite_set SUBTYPE_OF (nonempty?[T])

nonemp_fin_un1: JUDGEMENT union(NA, B) HAS_TYPE non_empty_finite_set
```

- The inclusion of these judgements in the library will minimize the number of TCCs that are generated.
- Without the `JUDGEMENT` statements, every use of the basic set operations on a finite set (e.g. `add(x,S)`) in a context that requires a finite set, would result in the generation of a TCC.
- What's the different between these judgements and the lemmas on the previous page?

Structure Of The Finite Sets Library

The library contains the following theories

`finite_sets`	part of the prelude, not library (provides basic type and cardinality)
`finite_sets_sum`	summation over a set
`finite_sets_minmax`	min and max over a set
`finite_sets_inductions`	induction schemes
`finite_sets_sum_real`	additional properties for summations over real-valued functions
`finite_sets_int`	special results of integer sets
`finite_sets_nat`	special results of natural num sets

The library also contains theories `card_def`, `finite_sets_def`, and `card_lt` which are not meant to be directly imported.

Cardinality of a Finite Set - for Reference

```
S: VAR finite_set[T]

inj_set(S): (nonempty?[nat]) =
               {n | EXISTS (f: [(S)->below[n]]) : injective?(f) }

Card(S): nat = min(inj_set(S))

card(S): {n: nat| n = Card(S)}        % inhibit expansion
```

- Cardinality is defined to be the smallest `n` for which an injection exists.
- To inhibit expansion, the `card` function is defined using a return type that is a singleton.
- The definition can be retrieved using a `typepred` command (e.g. `typepred "card(S!1)"`) or the `card_bij` theorem:

```
card_bij:  THEOREM card(S) = N IFF
           (EXISTS (f:  [(S) -> below[N]]): bijective?(f))
```

Lemmas of card Over the Set Operations

card_union	$\|A \cup B\| = \|A\| + \|B\| - \|A \cap B\|$
card_add	add one if element is not in set
card_remove	remove one if element is in set
card_subset	$A \subseteq B$ implies $\|A\| \leq \|B\|$
card_emptyset	equals zero
card_singleton	equals one

Most users of the library will only need to use these lemmas and not the more fundamental definition of `card`.

Minimum and Maximum of a Set

The library[2] provides functions that return the minimum and maximum elements of a set

```
SS: VAR non_empty_finite_set[T]

min(SS): {a:T | SS(a) AND (FORALL (x:T): SS(x) IMPLIES a <= x)}
max(SS): {a:T | SS(a) AND (FORALL (x:T): SS(x) IMPLIES x <= a)}
```

- These functions are not constructively defined, but are merely constrained to return a value from a specified set.

The following useful properties of `min` and `max` over the set `union` operator are also provided:

```
A,B:   VAR non_empty_finite_set

min_union: LEMMA min(A) = x AND min(B) = y IMPLIES
                    min(union(A,B)) = min(x,y)

max_union: LEMMA max(A) = x AND max(B) = y IMPLIES
                    max(union(A,B)) = max(x,y)
```

[2] nasalib/finite_sets/finite_sets_minmax.pvs

Summation Over a Set

The library[3] provides summation

```
sum(S,f) : RECURSIVE R =
    IF (empty?(S)) THEN zero
    ELSE f(choose(S)) + sum(rest(S),f)
    ENDIF MEASURE (LAMBDA S,f: card(S))
```

Many useful properties of `sum` are available, including:

```
x : VAR T
S,A,B: VAR finite_set

sum_empty: THEOREM sum(emptyset[T],f) = zero

sum_singleton: THEOREM sum(singleton(x),f) = f(x) + zero

sum_add: THEOREM sum(add(x,S),f)
            = sum(S,f) + IF member(x,S) THEN zero ELSE f(x) ENDIF

sum_remove: THEOREM sum(remove(x,S),f)
              + IF member(x,S) THEN f(x) ELSE zero ENDIF = sum(S,f)
```

[3] nasalib/finite_sets/finite_sets_sum.pvs

Induction Schemes

The library[4] provides several induction schemes over sets:

`cardinal_induction`	inducts over cardinality of the set
`finite_set_induction`	`p(emptyset)` and `p(S) => p(add(e,S))`
`finite_set_ind_modified`	`p(emptyset)`, not `S(e)`, and `p(S) => p(add(e,S))`
`finite_set_induction_rest`	`p(emptyset)` and `rest(S) => p(S)`
`finite_set_induction_union`	`p(emptyset)` and `p(S1) AND p(S2) => union(S1,S2)`
`finite_set_induction_gen`	`(FORALL S2: |S2| < |S| => p(S2)) => p(S)`
`nonempty_card_induction`	inducts over cardinality of the set
`nonempty_finite_set_induct`	not `S(e)`, and `p(S) => p(add(e,S))`

Use these by, e.g., `(induct :name "finite_set_induction")`

[4] nasalib/finite_sets/finite_sets_inductions.pvs

Bags (aka Multisets)[5]

- Sets capture information about membership
- Bags capture information about quantity

  `bag:  TYPE = [T -> nat]`
- Located in the `structures` directory of the library
- Convert a bag to a set: `bag_to_set`

Some operations on bags:

```
emptybag          : bag = (LAMBDA t: 0)

insert(x,b)       : bag = (LAMBDA t: IF x = t THEN b(t) + 1 ELSE b(t) ENDIF)
purge(x,b)        : bag = (LAMBDA t: IF x = t THEN 0 ELSE b(t) ENDIF)
extract(x,b)      : bag = (LAMBDA t: IF x = t THEN b(t) ELSE 0 ENDIF)

plus(a,b)         : bag = (LAMBDA t: a(t) + b(t))
union(a,b)        : bag = (LAMBDA t: max(a(t),b(t)))
intersection(a,b): bag = (LAMBDA t: min(a(t),b(t)))
```

[5] Defined in NASA's `structures` library

Object Collections: Four Ways in PVS

- sequence `[nat -> T]`
- bounded array `[below(N) -> T]`
- finite sequence

  `[# length:  nat, seq:  [below[length] -> T] #]`
- list datatype

```
    list [T: TYPE]: DATATYPE
      BEGIN
        null: null?
        cons (car: T, cdr:list):cons?
      END list
```

lists will be covered in the abstract data type lecture

Sequence

PVS provides a sequence (i.e., unbounded array) as follows:

```
T: TYPE

A1: FUNCTION [nat -> T]
A2: ARRAY [nat -> T]
A3: [nat -> T]
A4: sequence[T]
```

all of which are the same.

Prelude sequences Theory

function	meaning
`nth(seq, n)`	n^{th} element of the sequence
`suffix(seq, n)`	sequence starting after the n^{th} element
`first(seq)`	first element
`rest(seq)`	sequence excluding the first element
`add(x, seq)`	add element x to the sequence
`delete(n, seq)`	delete the n^{th} element
`insert(x, n, seq)`	insert x into *seq* at n

In addition to these definitions are certain results such as:

- `add_first_rest: LEMMA add(first(seq), rest(seq)) = seq`

Bounded Array[6]

Sometimes it is useful to have an array that is indexed by integer subrange as in a programming language:

```
below_arrays[N: nat, T: TYPE]: THEORY
BEGIN
   below_array: TYPE = [below(N) -> T]

   A: VAR below_array
   x: VAR T
   ii: VAR below(N)

   in?(x,A): bool = (EXISTS ii: x = A(ii))
END below_arrays
```

Note that `below` is defined in PVS prelude

```
below(i: nat): TYPE = {s: nat | s < i}
```

[6]Defined in NASA's `structures` library

Definition of Array Maximum - for Reference

`imax_rec`[7] returns the index of the maximum value

```
imax_rec(A,ii,jj): RECURSIVE below(N) =
        IF jj <= ii THEN ii
        ELSE
            LET IX = imax_rec(A,jj-1) IN
            IF A(IX) <= A(jj) THEN jj ELSE IX ENDIF
        ENDIF MEASURE (LAMBDA A,ii,jj: jj)
```

This generates the following TCCs:

```
imax_rec_TCC1: OBLIGATION (FORALL (jj): jj = 0 IMPLIES 0 < N);
imax_rec_TCC2: OBLIGATION (FORALL (jj): NOT jj = 0
                              IMPLIES jj - 1 >= 0 AND jj - 1 < N);
imax_rec_TCC3: OBLIGATION (FORALL (A, jj): NOT jj = 0
                              IMPLIES jj - 1 < jj);
```

all of which are discharged with `M-x tcp`.

[7]`nasalib/finite_sets/finite_sets_inductions.pvs`

Properties of imax_rec - for Reference

```
imax_rec_lem:  LEMMA j <= jj IMPLIES A(j) <= A(imax_rec(A,jj))
```

Proof:

```
(""
 (induct "jj" 1)
 (("1" (flatten) (skosimp*) (expand "imax_rec") (assert))
  ("2" (skosimp*) (expand "imax_rec" +) (inst?) (lift-if) (ground))))
```

```
imax_rec_rng:  LEMMA 0 <= imax_rec(A,jj) AND imax_rec(A,jj) <= jj
```

Proof:

```
(""
 (induct "jj" 1)
 (("1" (flatten) (skosimp*) (expand "imax_rec") (propax))
  ("2" (skosimp*) (expand "imax_rec" +) (inst?) (lift-if) (ground))))
```


Definition of max(A) and Properties

```
imax(A): below(N) = imax_rec(A,N-1)

max(A): real = A(imax(A))


max_lem : LEMMA A(i) <= max(A)

imax_lem: LEMMA A(imax(A)) = max(A)

max_def : LEMMA A(i) <= max(A) AND in?(max(A),A)
```

Array Concatenation [8]

```
concat_arrays [n:nat, m:nat, T: TYPE]: THEORY
BEGIN
  IMPORTING below_arrays

  a_n: VAR below_array[n,T]
  a_m: VAR below_array[m,T]
  nm : VAR below(n+m)

  o(a_n, a_m): below_array[n+m,T]
                  = (LAMBDA nm: IF nm < n THEN a_n(nm)
                                ELSE a_m(nm - n)
                                ENDIF)
```

- The function o overloads a function already defined in the prelude.
- The return type of o depends upon the theory parameters n and m.
- o is an operator
  - Either o(A,B) or A o B are valid

[8] nasalib/structures/concat_arrays.pvs

Array Concatenation Properties

```
  a_n: VAR below_array[n,T]
  a_m: VAR below_array[m,T]
  nm : VAR below(n+m)

  concat_array_bot0: THEOREM m = 0 IMPLIES a_n o a_m = a_n
  concat_array_top0: THEOREM n = 0 IMPLIES a_n o a_m = a_m


  i: VAR below(n)
  j: VAR {i: int | i >= n AND i < n+m}

  concat_array_bot : THEOREM (a_n o a_m)(i) = a_n(i)
  concat_array_top : THEOREM (a_n o a_m)(j) = a_m(j-n)
```

Array Extraction

Given an array `A = [a_0, a_1, a_2, a_3, ..., a_(N-1)]`, we want the elements `A^(m,n) = [a_m, ..., a_n]`

```
caret_arrays [N:nat, T: TYPE]: THEORY
BEGIN
  IMPORTING below_arrays, empty_array_def

  A: VAR below_array[N,T]
  m, n: VAR nat
  p: VAR [nat, below[N]]

  empty_array: below_array[0,T]

  ^(A, p): below_array[LET (m, n) = p IN
                          IF m > n THEN 0
                          ELSE n - m + 1 ENDIF,T] =
      LET (m, n) = p IN
        IF m <= n THEN (LAMBDA (x: below[n-m+1]): A(x + m))
        ELSE empty_array
        ENDIF
```


Properties of Array Extraction

```
caret_all : LEMMA N > 0 IMPLIES A^(0,N-1) = A

caret_ii_0: LEMMA FORALL (i: below(N)): (A^(i,i))(0) = A(i)

caret_elim: LEMMA
     FORALL (j: below(N), i: upto(j), k: below(j-i+1)):
                                    (A ^ (i, j))(k) = A(i+k)
```

- `(A^(i,i))` extracts an array with a single element
- `(A^(i,i))(0)` returns the single element

Prelude Theory Finite Sequences

```
finite_sequences [T: TYPE]: THEORY
 BEGIN
  finite_sequence: TYPE = [# length:nat, seq:[below[length] -> T] #]
  finseq: TYPE = finite_sequence

  fs, fs1, fs2, fs3: VAR finseq
  m, n: VAR nat

  empty_seq: finseq =
    (# length := 0,
       seq := (LAMBDA (x: below[0]): epsilon! (t:T): true) #)

  finseq_appl(fs): [below[length(fs)] -> T] = fs`seq;
```


Finite Sequences Operations

Similar to bounded arrays, concatenation and extraction are defined

Concatenation operator:

```
o(fs1, fs2): finseq =
   LET lsum = fs1`length + fs2`length
    IN (# length := lsum,
          seq := (LAMBDA (n:below[lsum]):
                   IF n < fs1`length
                      THEN fs1`seq(n)
                      ELSE fs2`seq(n-fs1`length)
                   ENDIF) #);
```

Extraction operator:

```
p: VAR [nat, nat]

^(fs, p): finseq =
  LET (m, n) = p
   IN IF m > n OR m >= fs`length
     THEN empty_seq
     ELSE LET len = min(n - m + 1, fs`length - m)
          IN (# length := len,
                seq := (LAMBDA (x: below[len]): fs`seq(x + m)) #)
     ENDIF
```