On enumeration of monadic predicates and n-ary relations

Harsh Raju Chamarthi
Northeastern University

November 30, 2012

Problem

Goal

Find counterexamples to a given formula in ACL2.

$(\text{and } hyp_1 \cdots hyp_n) \longrightarrow concl$

Problem

Goal

Find counterexamples to a given formula in ACL2.

$(\textsf{and}\ hyp_1 \cdots hyp_n) \longrightarrow concl$

Given a set of hyp_i enumerate all satisfying assignments

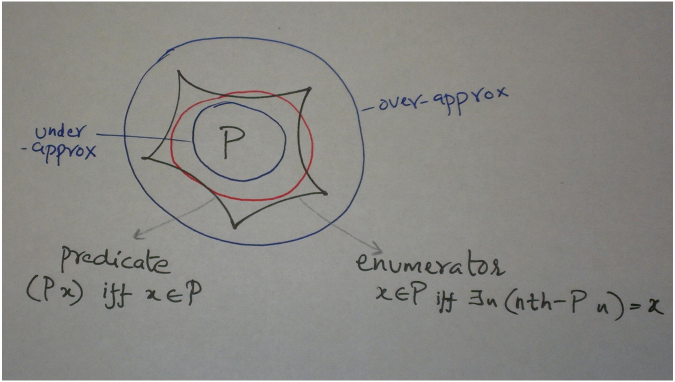
Problem

Goal

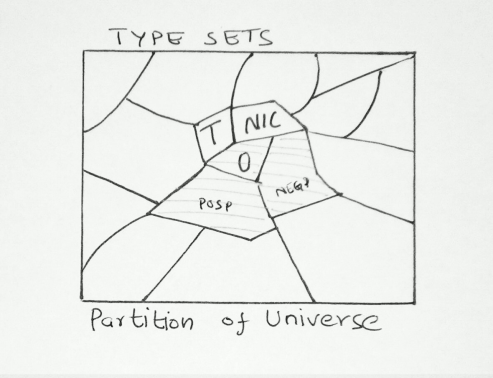
Find counterexamples to a given formula in ACL2.

(and $hyp_1 \cdots hyp_n) \longrightarrow concl$

Given a set of hyp_i enumerate all satisfying assignments



Background - Type sets



- 14 Primitive types
- Boolean combinations
- Represented as Bit strings
- Limited Expressibility

Background - Defdata framework

Defdata adds

- Product types
  - Constructors: cons, /, complex
- Inductive types

Background - Defdata framework

Defdata adds

- Product types
  - Constructors: cons, /, complex
- Inductive types

foo is a *defdata type* iff

1. predicate `foop` is defined and
2. either enumerator `nth-foo` or `*foo-values*` is defined

Examples

Product type -- `(defdata bar (cons (/ 1 pos) nat-list))`
Inductive type -- `(defdata loi (oneof nil (cons integer loi)))`

Background - Defdata framework

Defdata adds

- Product types
  - Constructors: cons, /, complex
- Inductive types

- *NOT*, *AND* combinations not supported
- Better, but still limited expressibility

Definition

enum expression gives an enumerating characterization of a variable. *enum set* is a disjunction of enumerator expressions, whose meaning is the union of the respective type domains characterized by the enum expressions.

Generative/Inductive Types (Different representation)

As much as possible express each type as an Inductive type with base elements and a finite set of generators.

$$
\begin{aligned}
posp : Base &= \{1\} \quad Gen = \{S\} \\
evenp : Base &= \{0\} \quad Gen = \{S \circ S\} \\
/3p : Base &= \{0\} \quad Gen = \{S \circ S \circ S\} \\
string\text{-}listp : Base &= \{nil\} \quad Gen = \{\lambda x.(cons\ a\ x) \mid_{(stringp\ a)}\}
\end{aligned}
$$

- A type $P : [Base : a, Gen : f]$ can be enumerated by listing members in a manner reminiscent of Herbrand Universe i.e.

$$\{a, fa, ffa, fffa, ffffa \ldots\}$$

- Clearly an enumerator for P can be easily derived:

```
(nth-P n) = if (zp n) a (f (nth-P(1-n)))
```

Generative Types (continued ...)

The $[Base, Gen]$ representation helps in deriving *AND* combinations.

$$evenp \wedge /3p \equiv Base = \{0\} \quad Gen = S \circ S \circ S \circ S \circ S \circ S$$

Heuristic - Take intersection of bases and the LCM of the generators.

NOT still not so amenable.
Source predicate - push the negation all the way inside i.e.

```
(~str-listp x) =
if (endp x)
   (not (equal x nil))
 (or (not (strp (car x)))
     (~str-listp (cdr x)))
```

$$Base = ATOM - \{nil\} \cup (cons\, a\, L)\ |_{(\ strp\ a)}$$

$$Gen = (cons\ a)\ |_{(strp\ a)}$$

Monadic Recursive Predicates

- Foregoing language for "types" still not expressive enough.

  e.g. `orderedp`, `no-duplicatesp`

```
(no-duplicatesp X) =
   if (endp X)
      T
    (and (not (in (car X) (cdr X)))
         (no-duplicatesp (cdr X)))
```

- Dependent Recursion

$$\textit{no-duplicatesp} : \textit{Base} = \{\textit{nil}\}, \textit{Gen} = \lambda x.(\textit{cons}\, a\, x) \mid_{a \notin x}$$

To characterize `no-duplicatesp`, need to know a enumerating characterization of n-ary relations!!

Binary Relations

`(in a X)`

- Find all $< a, X >$ pairs that satisfy `(in a X)` $\neq$ `nil`
- Given X, find all a that satisfy `(in a X)` $\neq$ `nil`
- Given a, find all X that satisfy `(in a X)` $\neq$ `nil`

Binary Relations

```
(in a X)
```

$a \mid_{a \in X}$

Natively supported.

$X \mid_{a \in X}$

Use a FIXing Rule to obtain an enum expression!

```
X = (insert a X')
```

Binary Relations

```
(in a X)
```

$a \mid_{a \in X}$

Natively supported.

$X \mid_{a \in X}$

Use a FIXing Rule to obtain an enum expression!

```
X = (insert a X')
```

$x \mid_{x<y}$

$y \mid_{x<y}$

Binary Relations

```
(in a X)
```

$a \mid_{a \in X}$

Natively supported.

$X \mid_{a \in X}$

Use a FIXing Rule to obtain an enum expression!

```
X = (insert a X')
```

$x \mid_{x<y}$

Use $x = (y - z) \mid_{z>0}$

$y \mid_{x<y}$

Use $y = (x + z) \mid_{z>0}$

Binary Relations

```
(in a X)
```

$a \mid_{a \in X}$

Natively supported.

$X \mid_{a \in X}$

Use a FIXing Rule to obtain an enum expression!

`X = (insert a X')`

$x \mid_{x<y}$

Use $x = (y - z) \mid_{z>0}$

$y \mid_{x<y}$

Use $y = (x + z) \mid_{z>0}$

Fix Rules

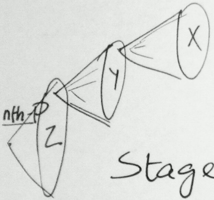
Like *Elim* rules. `(defthm in-fix2 (in a (insert a X)))` Eliminate a relation in favor of enum expressions

e.g. $= f(\ldots)$ or $\inf(\ldots)$.

Enumerating Relations

$$\langle X,Y,Z\rangle \left|\begin{array}{l}{}^{\langle\Omega,\Omega,\Omega\rangle} \\ \hline R(X,Y,Z)\end{array}\right. \equiv X \left|\begin{array}{l}{}^{\text{es}:\Omega} \\ \hline X\in R^{-1}(Y,Z) \ \text{ or } \ X=R^{-1}(Y,Z,n) \\ Q(Y,Z)\end{array}\right.$$

$$\equiv X \left|\begin{array}{l}{}^{=R^{-1}(Y,Z,n)} \\ \hline T\end{array}\right. \bullet \ \langle Y,Z\rangle \left|\begin{array}{l}{}^{\langle\Omega,\Omega\rangle} \\ \hline Q(Y,Z)\end{array}\right.$$



$$\equiv \quad '' \quad \bullet \ Y \left|\begin{array}{l}{}^{\in g(Z)} \\ \hline P(Z)\end{array}\right.$$

$$\equiv \quad '' \quad \bullet \quad '' \quad \bullet \ Z \left|\begin{array}{l}{}^{\text{es}:\Omega} \\ \hline P(Z)\end{array}\right.$$

Staged Enumeration!

Monadic

On deriving R^{-1}

```
(subsetp X Y) = (if (endp X)
                    T
                  (and (in (car X) Y)
                       (subsetp (cdr X) Y)))
```

```
(subsetp⁻² n X) = (if (endp X)
                      (nth-all n)
                    (insert (car X)
                            (subsetp⁻² n (cdr X)))
```

```
(subsetp⁻¹ n Y) = (if (zp n)
                      nil
                    (cons (nth* n1 Y)
                          (subsetp⁻¹ p n2 Y))
```

Probably doable, but more elegant to let user specify *ELIM* rules

$$\textit{elim for } X: \quad X = Y - Z$$

$$\textit{elim for } Y: \quad Y = X \cup Z$$

Monadic Predicates (continued...)

X	tlp
	(orderedp X)

orderedp X = if (or (endp x)
                    (endp (cdr X)))
                 T
                 (and (≤ (car x) (cadr x))
                      (orderedp (cdr x

{def} ≡

X	tlp	blp
	(or (endp x) (endp (cdr x)))	consp X consp (cdr X) ≤ (car X) (cadr X) orderedp (cdr X)

{OR} ≡

X	~~tlp~~ nil	tlp	tlp
	~~(endp x~~ T	consp endp (cdr X)	"

{...} ≡

X	= nil	= (cons a nil)	(cons a (cons b X3))
	T	T	a ≤ b orderedp X3

← ACL2 both   orderedp (cons b X3)

Orderedp : Base = {nil, [a]}   Gen = $\lambda y.(\text{cons } a\ (\text{cons } b\ y))\big|_{a \le b}$

Monadic Predicates (continued...)

X	
	tlp
	(orderedp X)

orderedp X = if (or (endp x) (endp (cdr X)))

T

(and (≤ (car x) (cadr x)) (orderedp (cdr X)))

{def} ≡ X

tlp	tlp
(or (endp x) (endp (cdr x)))	consp X consp (cdr X) ≤ (car X) (cadr X) orderedp (cdr X)

{OR} ≡ X

tlp nil	tlp	tlp
~~(endp x~~ T	consp endp (cdr X)	"

{...} ≡ X

=nil	=(cons a nil)	(cons a (cons b X3))
T	T	a ≤ b orderedp X3

← ACL2 both — orderedp (cons b X3)

FIXing Rule

(orderep (sort X))

Orderedp : Base = {nil, [a]}   Gen = $\lambda y.$(cons a (cons b y))$|_{a \leq b}$

Monadic Pred (AND)

$X \left| \dfrac{\ }{\begin{array}{l} \text{str-listp } X \\ \text{no-dup } X \\ \text{orderedp } X \end{array}} \right. \equiv X \left| \dfrac{\text{nth-str-list}}{\begin{array}{l} \text{no-dup } X \\ \text{orderedp } X \end{array}} \right.$

Apply FIX rules

i.e $X \left| \dfrac{\text{(sort (rem-dup ...}}{T} \right.$

or, Thread the 3 functions i.e take LCM

$\equiv X$	= nil	= cons a nil	= cons a (cons b X3)
	T	(strp a)	$a \le b$, $a \notin$ (cons b X3), $b \notin X3$ → $a < b$, $b \notin X3$; strp a, b; str-listp ∧ no-dup ∧ orderedp X3
	base	base	recursive

ad2-count less

stlp∧no-dup∧ordp: Base = { nil , (cons a nil) | (strp a) }

conjunction predicate

Gen = λy. (cons a (cons b y)) | strp a, b ; $a < b$; $b \notin y$

Monadic Predicates (more complex)

Ques: Can all monadic predicates be represented in be represented in [*Base*, *Gen*] form?

Consider `squarep` and `primep`

```
(squarep x) = (sq1 x x)
(sq1 b x) = if (zp b)
                 nil
               if b*b = x
                  T
                (sq1 b-1 x)
```

```
(primep x) = (nd X) = 2
           = (Pr1 x x-1)

(Pr1 x y) = if y = 1
                 T
               (and (not (div x y))
                    (pr! x y-1))
```

Base = ? Gen = ??

```
(nth-square n) = (* n n)
```

Fix Rule

```
(posp x) => (squarep (* x x))
```

```
(nth-prime n) = ...
```

Fix Rule ??

Monadic Predicates (more complex)

Ques: Can all monadic predicates be represented in be represented in $[Base, Gen]$ form?

Consider `squarep` and `primep`

```
(squarep x) = (sq1 x x)
(sq1 b x) = if (zp b)
               nil
             if b*b = x
                T
              (sq1 b-1 x)

(primep x) = (nd X) = 2
           = (Pr1 x x-1)

(Pr1 x y) = if y = 1
               T
             (and (not (div x y))
                  (pr! x y-1))
```

Base = ? Gen = ??

`(nth-square n) = (* n n)`

Fix Rule

`(posp x) => (squarep (* x x))`

`(nth-prime n) = ...`

Fix Rule ??

$R(x, f(x), g(x))$ is a problem to enumerate ...

Equations and Inverses

From

$$X \,\Big|_{g(x)=y}$$

we would like to derive the `es:` $= g^{-}(y)$

From `(append X Y) = Z` we would like to derive `es: (difference Z Y)`$\Big|_{Y \subset Z}$

Mechanizable?

```
L = (zip l1 l2) = (if (or (endp l1)
                          (end p l2))
                      nil
                    (cons (cons (car l1) (car l2))
                          (zip cdr l1) (cdr l2)))

(l1, l2) = (unzip L) = (if (endp L)
                           (mv nil nil)
                         (mv-let (l1 l2)
                                 (unzip (cdr L))
                          (b* ((cons a b)  (car L))
                            (mv (cons a1 l1) (cons b l2)))))
```

Mechanizable?

```
L = (zip l1 l2) = (if (or (endp l1)
                          (end p l2))
                      nil
                    (cons (cons (car l1) (car l2))
                          (zip cdr l1) (cdr l2)))

(l1, l2) = (unzip L) = (if (endp L)
                           (mv nil nil)
                         (mv-let (l1 l2)
                                 (unzip (cdr L))
                          (b* ((cons a b)  (car L))
                            (mv (cons a1 l1) (cons b l2))))))
```

Inverse/Elim Rule for `zip`

```
(zip (strip-cars L) (strip-cdrs L)) = L
```

A ternary relation

```
(shufflep x y z) =
(if (endpz)
    x = y = z = nil
(if (endp x)
    y = z
(if (endp y)
    x = z
(or
 (and (car x) = (car z)
      (shufflep x' y z')
 (and ((car y) = (car z)
      (shufflep x y' z')))))
```

```
z = (shuffle x y) =
if (and (endp x) (endp y))
   nil
if (endp x)
   y
if (endp y)
   x
 (choose
  (cons (car x)
        (shuffle x' y))
  (cons (car y)
        (shuffle x y')))
```

Ques: Under what circumstances can this derivation be mechanized?

Interesting example...

```
(adj-listp G) =
(and (symbol-alistp G)
     (adj-list1p G (strip-cars G))
```

```
(adj-list1p G dom) =
(if (end G)
    T
  (and (consp (car G))
       (subsetp (cdar G) dom)
      (adj-list) P (cdr G) dom)
```

Method 1: Thread and derive

$$Base = \{nil\} \quad Gen = \lambda g.(c\,(c\,a\,b)\,g)\,|_{b \subset dom}$$

Method 2: Staged enumeration. Apply Fix Rules...

Method 3: Rewrite `G = (zip dom edges-list)`. Derive dom is `symbol-listp`. Then derive:

```
(R el dom) = (if (endp el)
                 T
               (and (subsetp (car el) dom)
                    (R (cdr el) dom))
```

Negation and Conjunction of Monadic Predicates

`(no-duplicatesp X) => (orderep X)`

Counterexamples: Enumerate `(and (no-dup X) (not (ordered X)))`

```
(no-dup X) =
(if (endp X)
    T
  (if (endp (cdr X))
      T
    (if (> (car X) (cadr X))
        (and (not (in (car X) X'))
             (no-dup X'))
      (and (not (in (car X) X'))
           (no-dup X')))))
```

```
Negate!
~(orderedp X) =
(if (endp X)
    nil
  (if (endp (cdr X))
      nil
    (if (car X) > (cadr X)
        T
      (~orderedp X'))))
```

Match the IF structure and merge the two predicates to get:

Negation and Conjunction of Monadic Predicates

```
(no-duplicatesp X) => (orderep X)
```

Counterexamples: Enumerate `(and (no-dup X) (not (ordered X)))`

```
(no-dup X) =
(if (endp X)
    T
  (if (endp (cdr X))
      T
    (if (> (car X) (cadr X))
        (and (not (in (car X) X'))
             (no-dup X'))
      (and (not (in (car X) X'))
           (no-dup X')))))
```

```
Negate!
~(orderedp X) =
(if (endp X)
    nil
  (if (endp (cdr X))
      nil
    (if (car X) > (cadr X)
        T
      (~orderedp X')))))
```

Match the IF structure and merge the two predicates to get:

```
(|no-dup & ~orderedp| X) =
(if (endp X)
    (and T nil)
  (if (endp (cdr X))
      (and T nil)
    (if (> (car X) (cadr X))
        (and (not (in (car X) (cdr X)))
             (no-dup (cdr X)))
      (and (not (in (car X) (cdr X)))
           (|no-dup & ~orderedp| (cdr X))))))
```

Recap

- Generative types
  - Base, Gen representation
  - AND
  - NOT
- Richer Types
  - Monadic Predicates build on n-ary relations
  - Instances of relations $R(x, x)$ are hard...
- Need Elim/Fix/Inverse Rules from the user to program the Cgen capability
- Staged Enumeration (Dependency graph dictated by Rules above)

Finally...

- Find a corresponding "The Method" for CGen framework.
- Interactive Non-Theorem Disproving
  same philosophy as ACL2, more integration with ACL2.
- Fundamental Questions.

Applications

- Lemma generation
- Internal Heuristics (Generalize, Induction)
- Counterexample generation

Finally...

- Find a corresponding "The Method" for CGen framework.
- Interactive Non-Theorem Disproving
  same philosophy as ACL2, more integration with ACL2.
- Fundamental Questions.

Applications

- Lemma generation
- Internal Heuristics (Generalize, Induction)
- Counterexample generation

Thank You!