Trapezoidal Generalization Over Linear Constraints

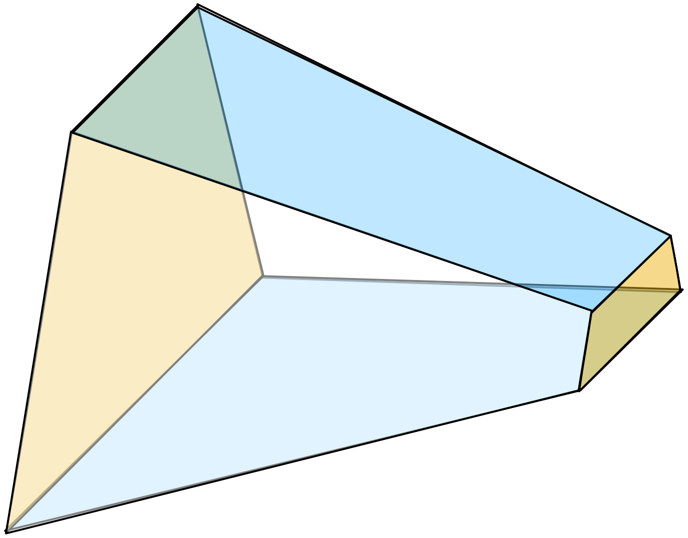
David Greve
Andrew Gacek

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) under Contract FA8750-16-C-0218.

Rockwell Collins
Building trust every day

Trapezoidal Generalization Talk Overview

- Motivation
  - Model-Based Fuzzing
- Previous Work
  - High-Level Spec
- Proof
  - Overview and Proof Pearls
- Future Efforts
  - Sampling



Model-Based Fuzzing

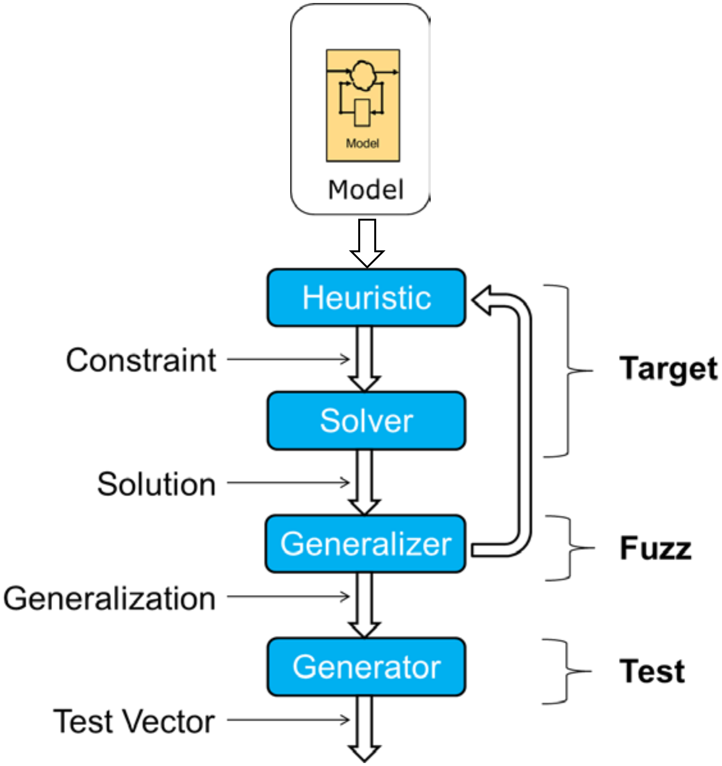
The use of Behavioral Models to perform Directed Fuzzing
in search of Cyber Vulnerabilities in Embedded System Targets

- Limited Knowledge of System Under Test
  - Requirements Specifications (Grey Box)
- Limited Visibility of System Behavior
  - Anomalous Behavior must Manifest at "System Level"
- Leverages Synergy Between Fuzzer and Solver Technologies
  - Solver Targets Known Behavior
  - Fuzzer Searches Unknown Behavior

Target What We Know
Fuzz What We Don't

Model-Based Fuzzing Pipeline

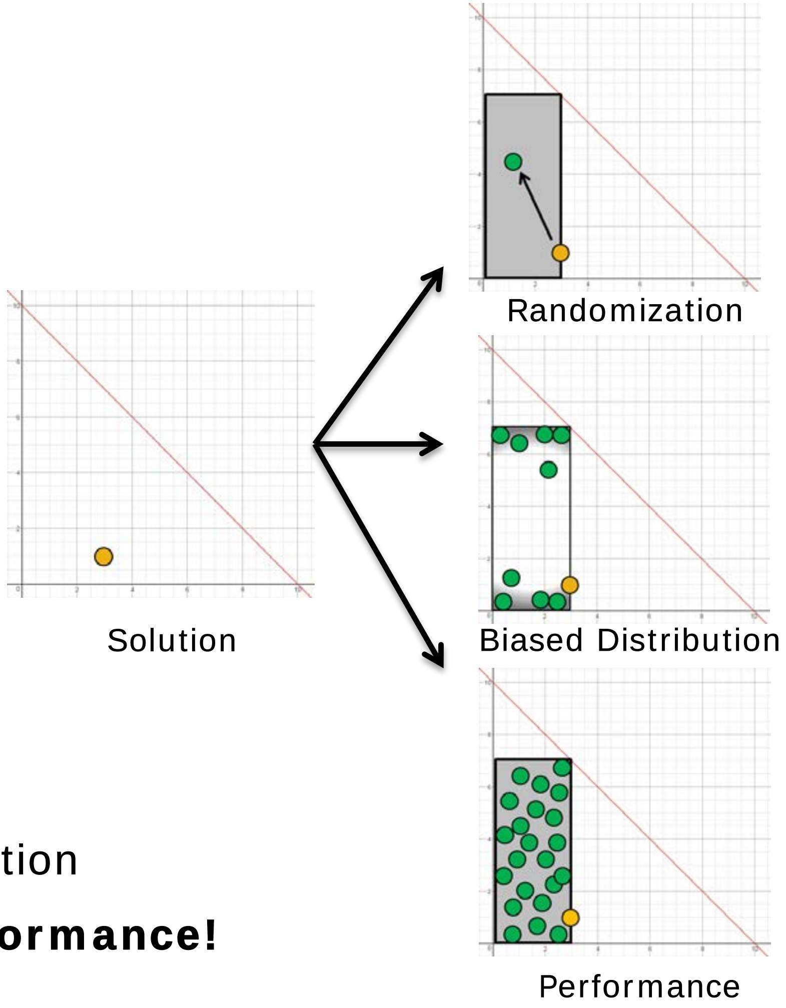
- *Model* Describes Fuzzing Target
  - Functional Behaviors, Stateful Protocols
- *Heuristics* are used to generate constraints
  - Driven by Testing Criteria/Metric
- *Constraint Solver* Generates Solutions
  - Solutions **Target** Interesting Model Behaviors
- *Generalizer* Randomizes (Fuzzes) Solution
  - **Explores** Behavioral Boundaries
- *Generator* Samples Generalization to produce Test Vectors
  - Much Faster than Solver



Generalization in Model-Based Fuzzing

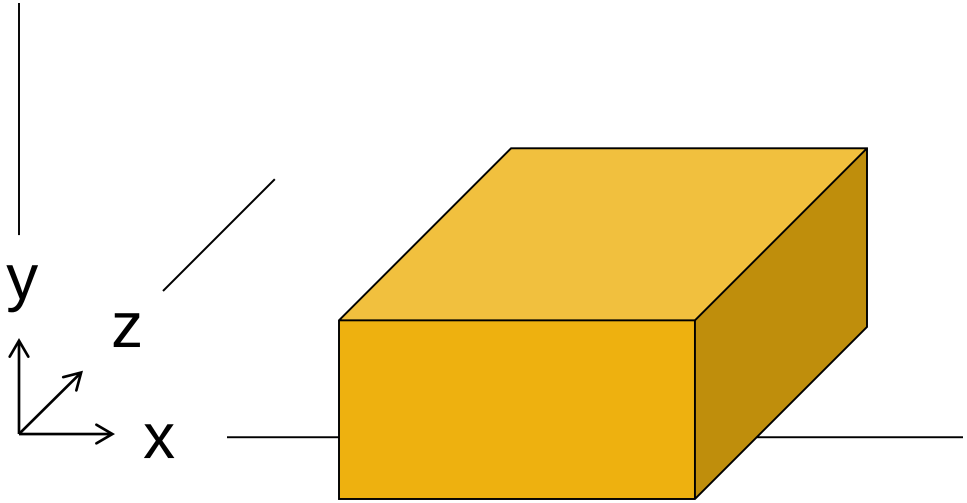
- Generalization
  - Transforms a Concrete Solution
    - Into a Set of Solutions
  - Produces a symbolic expression
    - In terms of system inputs
    - That Satisfies Constraint

- We use Generalization to
  - Randomize Solver Solutions
  - Influence Test Distributions
    - Boundary Value Testing
  - Decouple Solver from Test Generation
    - **Boost Test Generation Performance!**



Rectilinear Generalization

Lower Bound	Variable	Upper Bound
100 <	x	< 200
0 <	y	< 100
-50 <	z	< 50



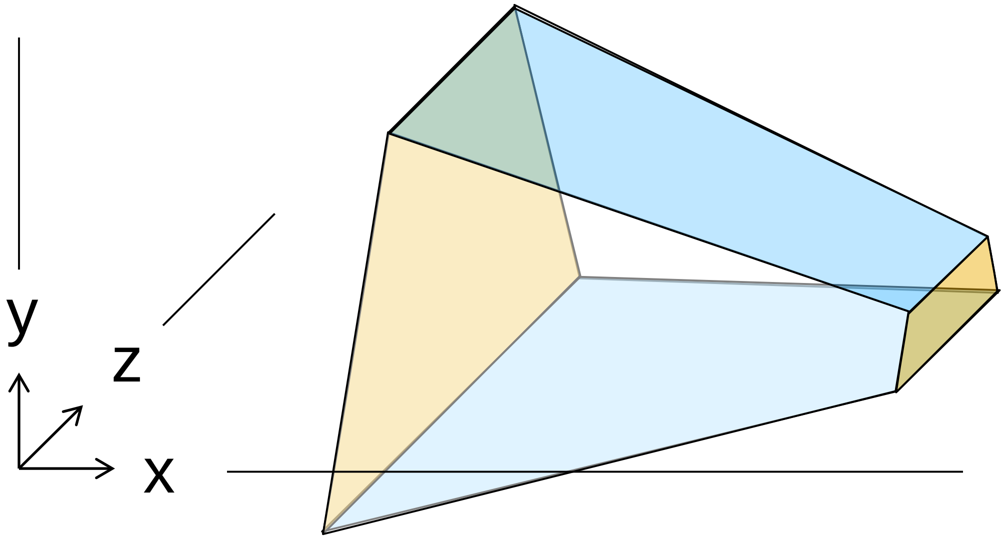
Trapezoidal Generalization

Lower Bound	Variable	Upper Bound
$100 <$	x	< 200
$3x - 290 <$	y	$< -3x + 970$
$y + x – 250 <$	z	$< -y + 7$

- List of Variable Bounds sorted by arbitrary variable ordering
  - Interpreted as a conjunction
- One Bound (Upper and Lower) Per Variable
- Bounds are rational 1st order multivariate polynomials
  - Expressed in terms of "smaller" variables

Trapezoidal Generalization

Lower Bound	Variable	Upper Bound
100 <	x	< 200
3x - 290 <	y	< -3x + 970
y + x – 250 <	z	< -y + 7



Trapezoidal Generalization (vs. Intervals)

- Reduced Dependency on Original Solution
- Better Approximation of Linear Features (Boundaries)
  - Enhanced Boundary Value Fuzzing
- Larger Generalization Regions
  - Each Counterexample yields more test vectors
- Bounded Representation Size
  - Worst Case Quadratic in #Inputs
- Efficient Computation
  - Worst Case Cubic Intersection
  - Worst Case Quartic for Integer Restriction
- Supports Efficient Sampling (Vector Generation)
  - Nearly As Efficient As Intervals

Sampling

Lower Bound	Variable	Upper Bound
$100 <$	x	< 200
$3x - 290 <$	y	$< -3x + 970$
$y + x – 250 <$	z	$< -y + 7$

Test Values
$x = 110$
y
z

Sampling

Lower Bound	Variable	Upper Bound
100 <	x	< 200
3x - 290 = 40 <	y	< 640 = -3x + 970
y + x – 250 <	z	< -y + 7

Test Values
x = 110
y = 50
z

Sampling

Lower Bound	Variable	Upper Bound
$100 <$	x	< 200
$3x - 290 = 40 <$	y	$< 640 = -3x + 970$
$y + x – 250 = -90 <$	z	$< -43 = -y + 7$

Test Values
$x = 110$
$y = 50$
$z = -50$

Trapezoidal Intersection Example 1

Lower	Variable	Upper
2 <=	x	
	y	< -x + 6

&

Lower	Variable	Upper
	x	< 5
3x + 2 <	y	

Lower	Variable	Upper
2 <=	x	< 5
3x + 2	y	< -x + 6

Trapezoidal Intersection Example 2

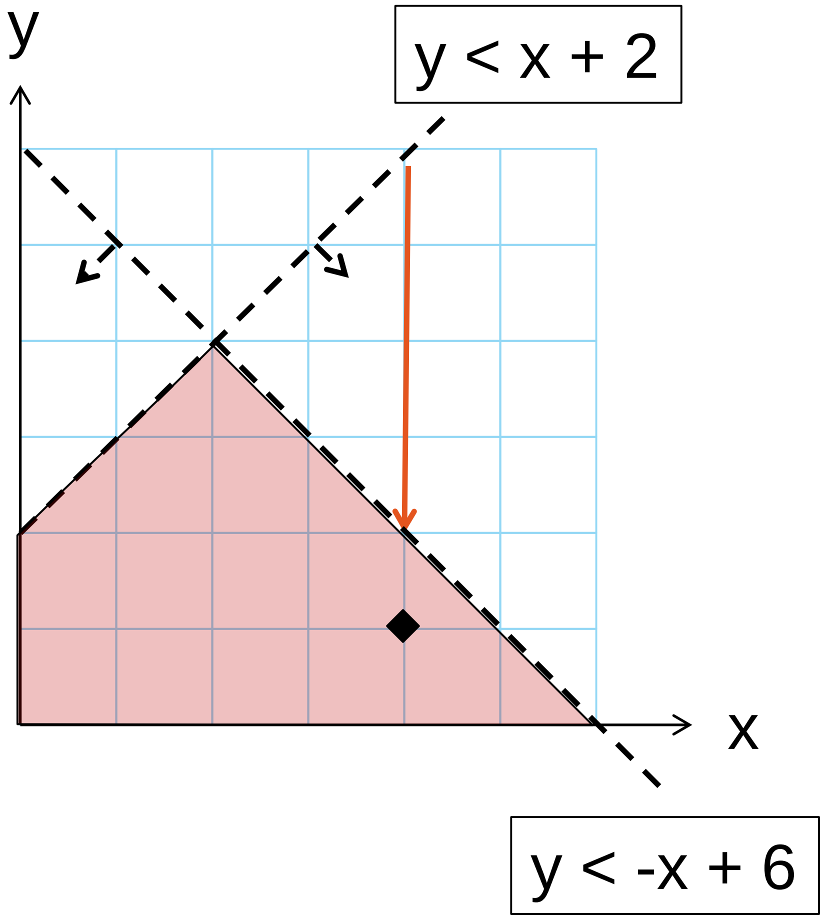
X = 4
Y = 1

Lower	Variable	Upper
	y	< x + 2

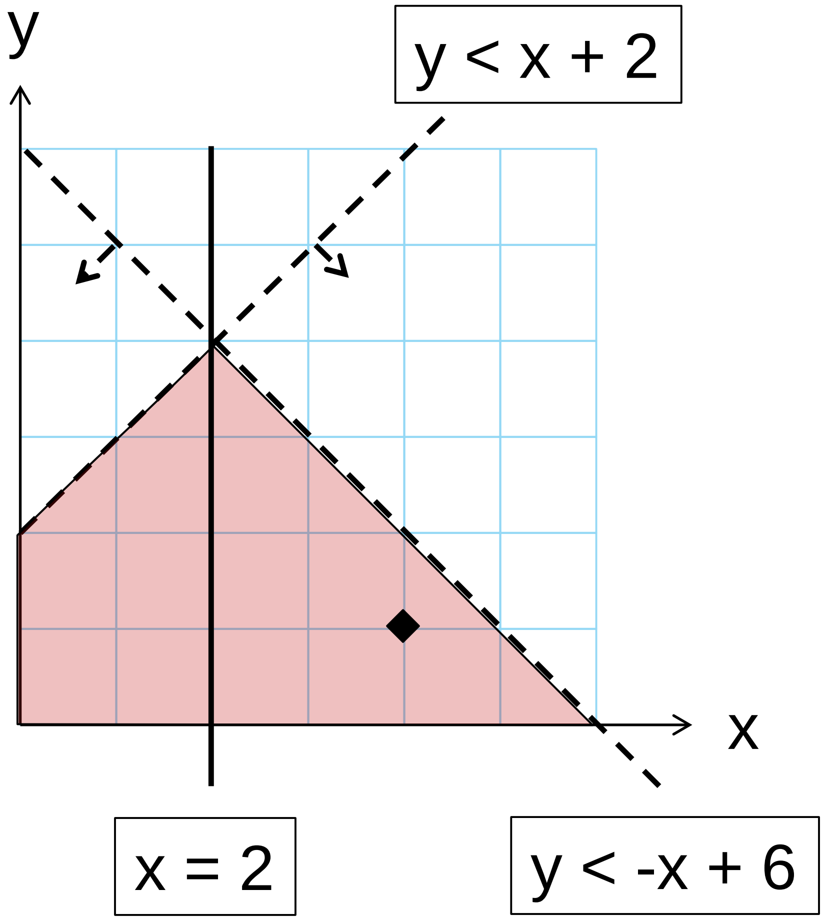
&

Lower	Variable	Upper
	y	< -x + 6

Domain Restriction

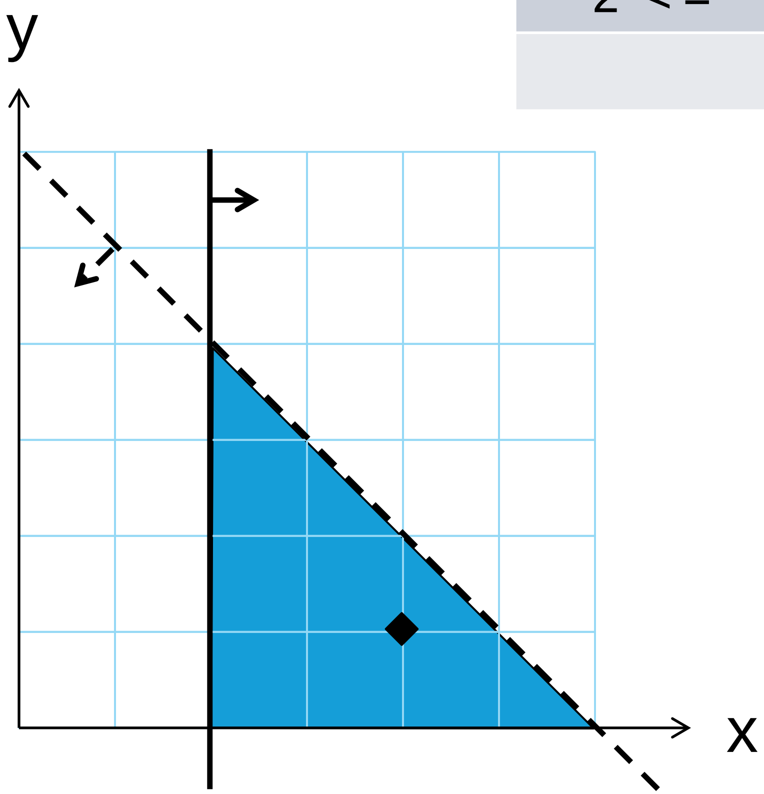


Domain Restriction



Domain Restriction

Lower	Variable	Upper
2 <=	x	
	y	< -x + 6



x >= 2

y < -x + 6

Trapezoidal Intersection Example 2

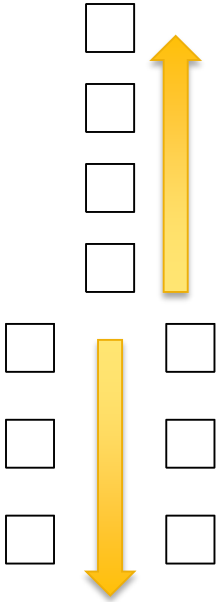
Lower	Variable	Upper
	y	< -x + 6

&

Lower	Variable	Upper
	y	< x + 2

Lower	Variable	Upper
2 <=	x	
	y	< -x + 6

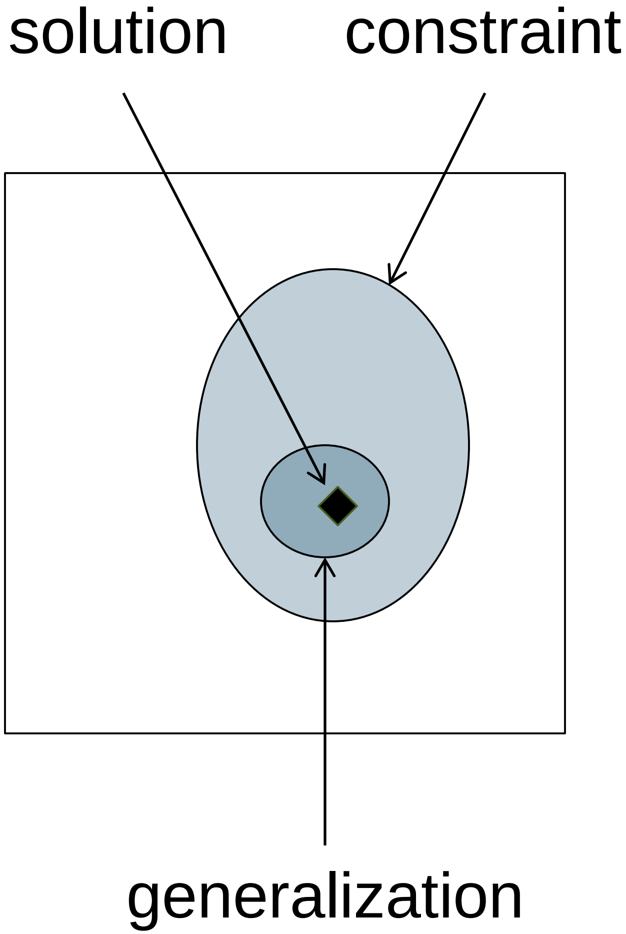
Trapezoid Intersection



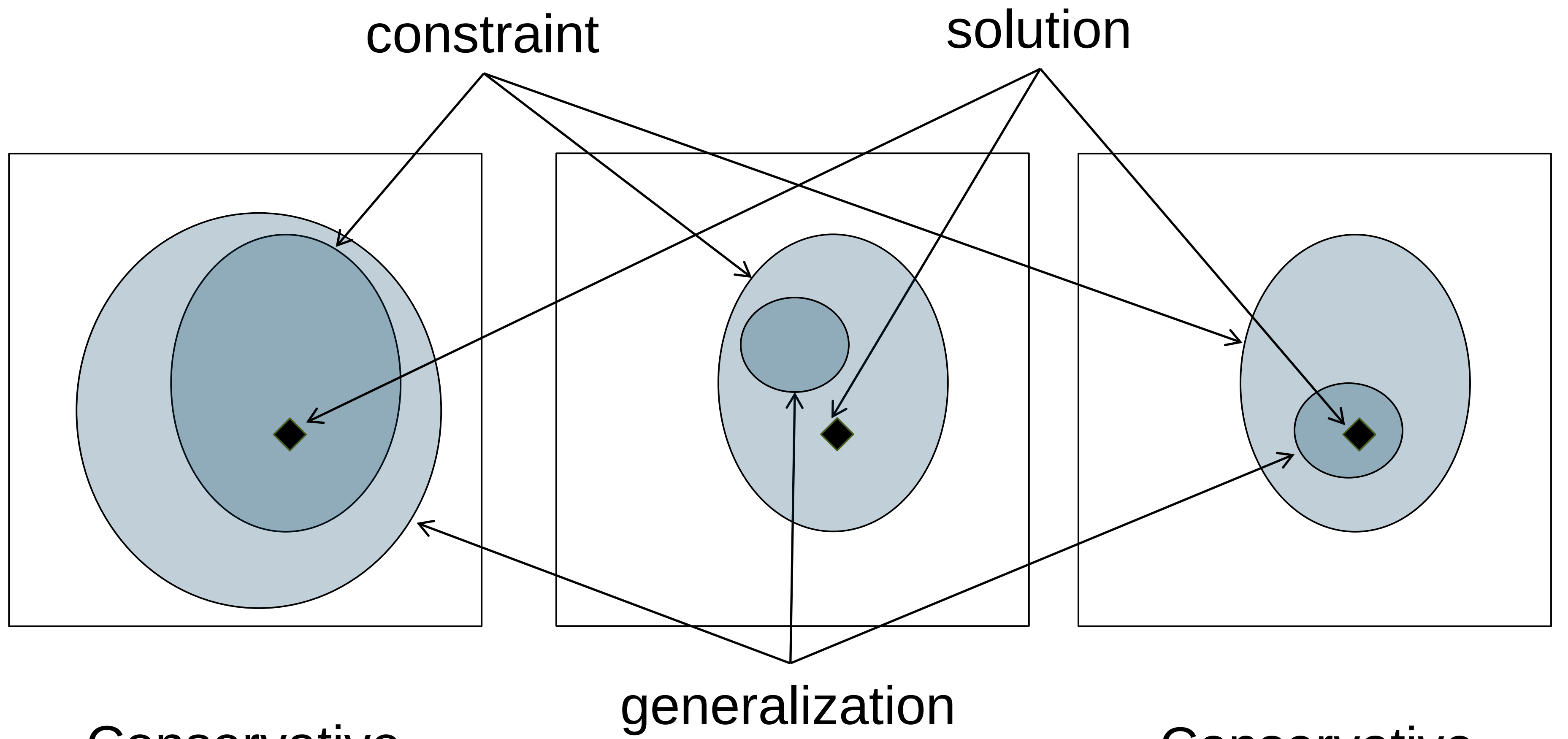
- If we intersect two trapezoids from smallest to largest
  - Domain Restrictions will be applied from largest to smallest
- Intersection of two variable constraints
  - May result in a Domain Restriction
- Domain Restrictions
  - Expressed in terms of Smaller Variables
- Intersection with a Domain Restriction
  - May result in 1 more (even smaller) restriction
    - Etc.
- Computational Complexity
  - Order N operations to intersect two trapezoids
  - Order N^2 operations to apply domain restrictions
  - Interval Intersection is Order N
  - Total Complexity Order N^3

Generalization Problem Statement

- Given
  - System Model
  - Constraint
  - Solution provided by Constraint Solver
- Generate a Generalization
  - Convert a single solution into a set of solutions
  - Express Result Concisely
    - Usually Generalization != Constraint
    - Result is Inexact



Possible Generalizations


constraint

solution

generalization

Conservative
Over-Approximation

Conservative
Under-Approximation

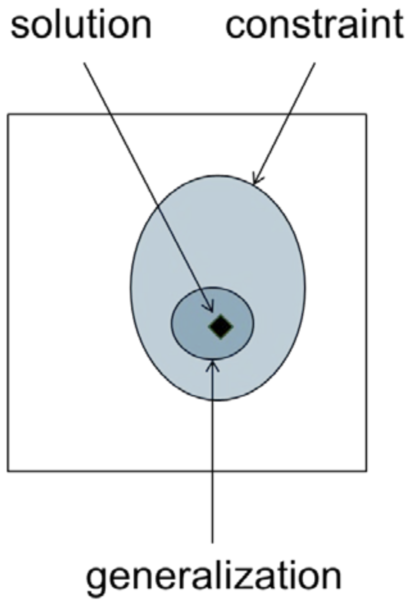
Previous Work (2017 Rump Session)

- Identified Conservative Under-Approximation
  - As Appropriate for our Application
- Formalized this Concept in ACL2
  - Expressed Correctness using 2 Invariants
- Refined a Set of Generalization Rules
  - We initially assumed that “Doing Nothing” was conservative
    - If you don’t change the expression, it trivially satisfies correctness
  - We were wrong !
  - It is easy to make these kinds of mistakes
    - ACL2 can help during algorithmic development
- Motivated continued Formalism
  - Verify Concrete Implementation

Generalization Correctness Statements

- Top Level Correctness Statement
  - Generalization Contains Original Solution
  - Generalization is a Subset of Original Constraint

- Invariants
  - Can be enforced incrementally
    - During Symbolic Simulation
  - Reduce to Correctness when applied to top level constraint

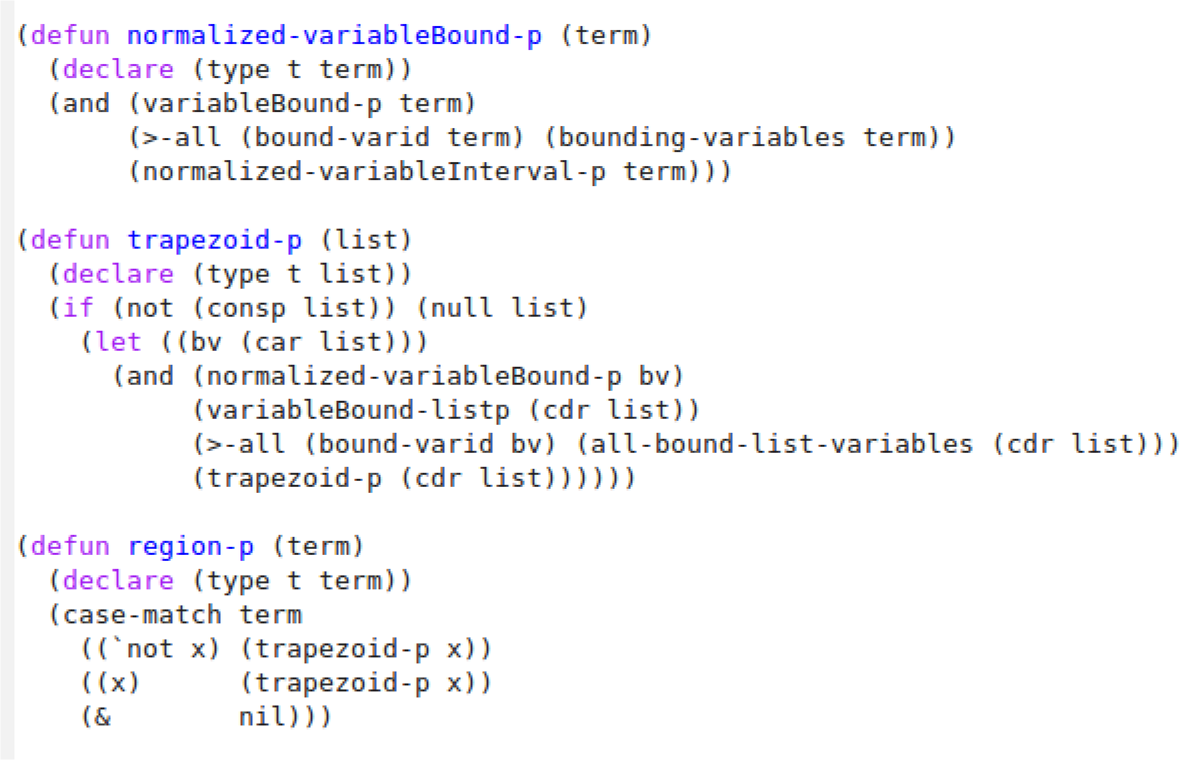


- Correctness Invariants
  - 1. Evaluating Solution on Generalization must be the same as Evaluating Solution on original expression
  - 2. Any input whose evaluation differs from that of the solution on the original expression must also differ on the Generalization

Trapezoidal Generalization: ACL2 Formalism

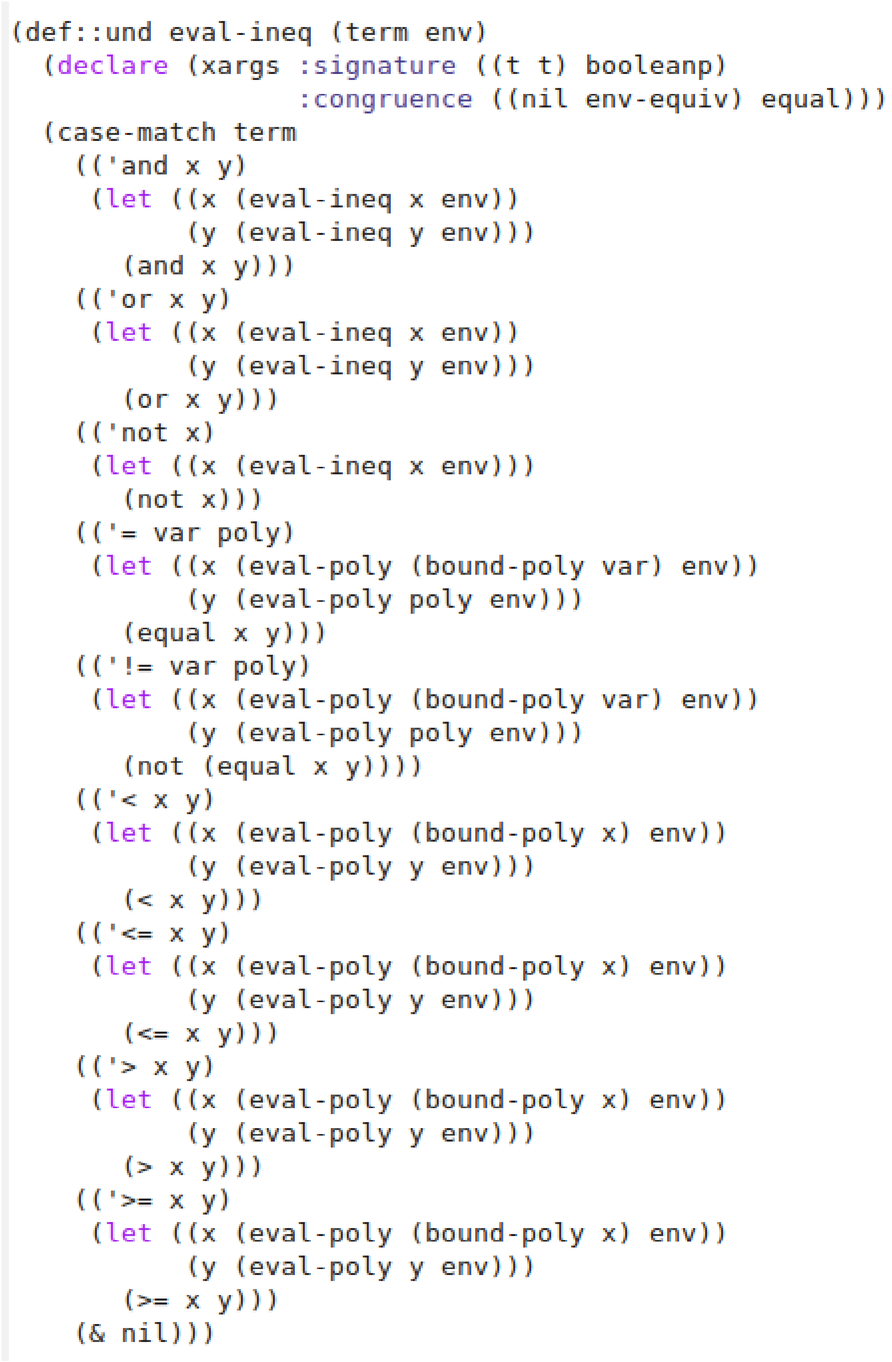
- Linear Rational Multi-Variate Polynomial Library
  - Formalization of Solving Equality/Inequality for one variable
- Interval Bounds
  - Bounds single variable w/to polynomials
  - Upper and/or Lower Inequalities or a single Equality
- Trapezoidal Data Structure, **Regions**
  - Ordered List of Interval Bounds
- Operational Building Blocks
  - Model Derived From Implementation Source Code
- Generalization Procedure
  - Generalizes a Solution Vector and produces a Region
  - Relative to arbitrary Boolean combinations of Linear Constraints
- Proof of Generalization Correctness
  - w/to 2 Correctness Invariants

Trapezoidal Data Types



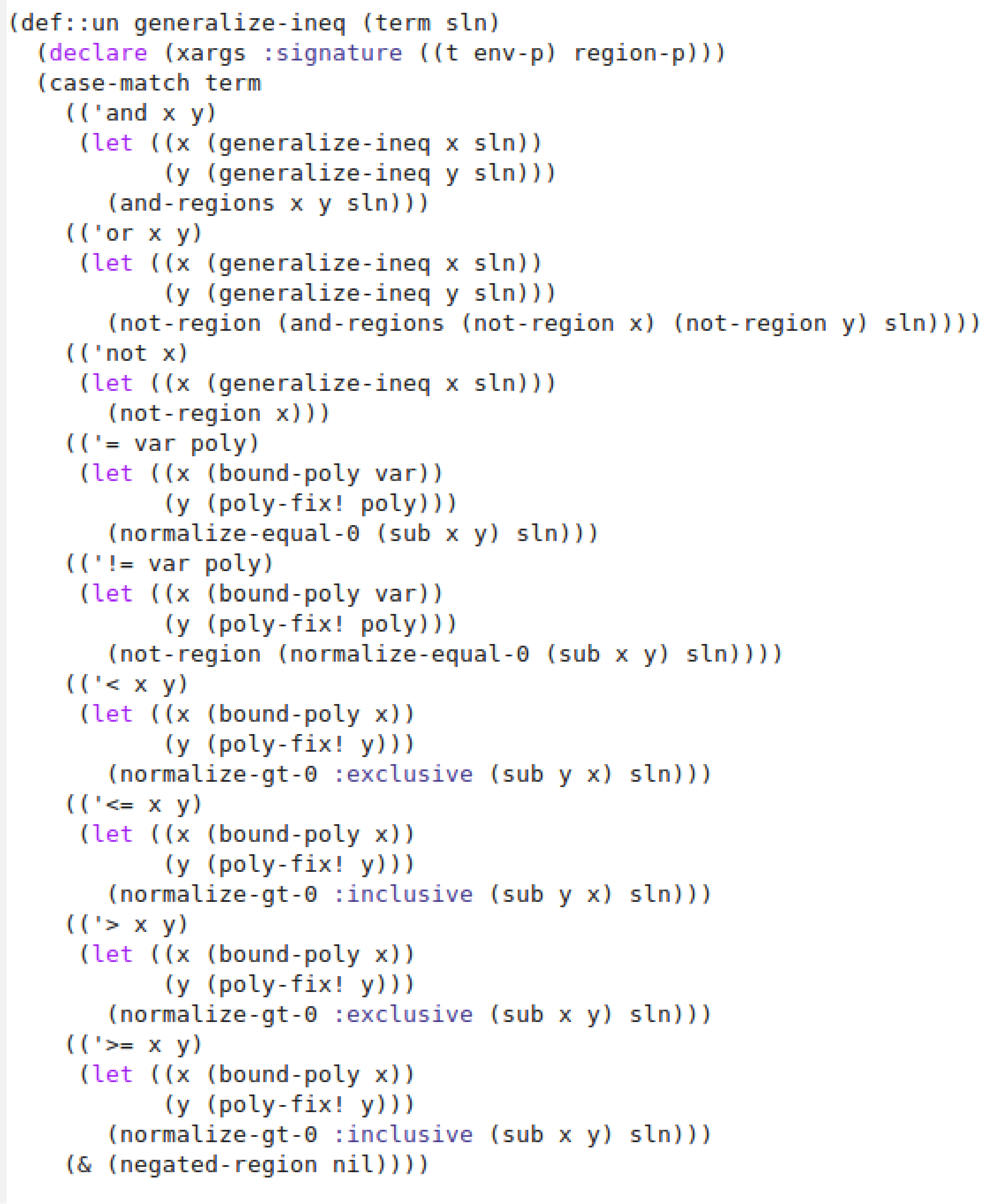
```
(defun normalized-variableBound-p (term)
  (declare (type t term))
  (and (variableBound-p term)
       (>-all (bound-varid term) (bounding-variables term))
       (normalized-variableInterval-p term)))

(defun trapezoid-p (list)
  (declare (type t list))
  (if (not (consp list)) (null list)
    (let ((bv (car list)))
      (and (normalized-variableBound-p bv)
           (variableBound-listp (cdr list))
           (>-all (bound-varid bv) (all-bound-list-variables (cdr list)))
           (trapezoid-p (cdr list))))))

(defun region-p (term)
  (declare (type t term))
  (case-match term
    ((`not x) (trapezoid-p x))
    ((x)      (trapezoid-p x))
    (&        nil)))
```

Evaluator



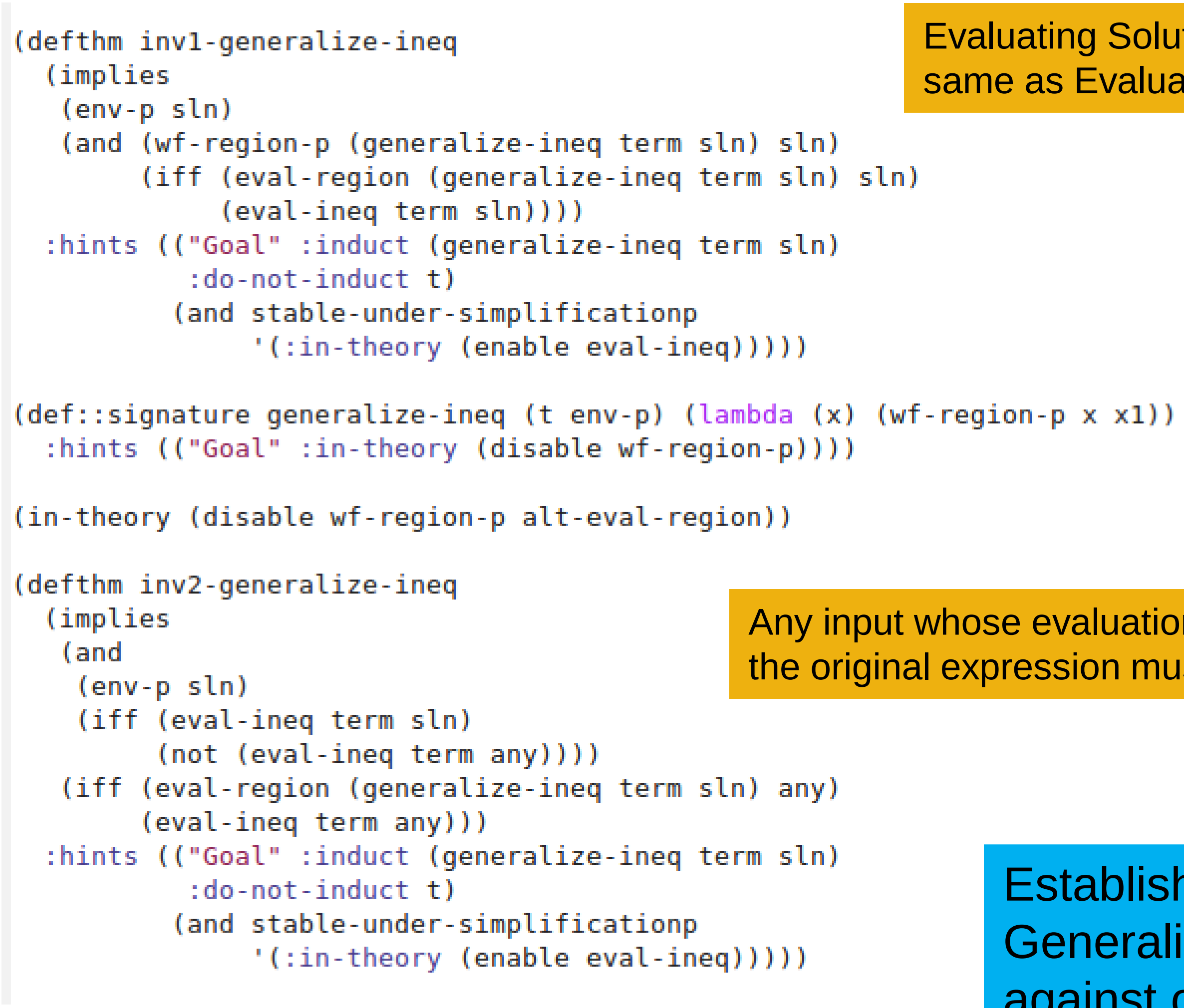
```
(def::und eval-ineq (term env)
  (declare (xargs :signature ((t t) booleanp)
                  :congruence ((nil env-equiv) equal)))
  (case-match term
    (('and x y)
     (let ((x (eval-ineq x env))
           (y (eval-ineq y env)))
       (and x y)))
    (('or x y)
     (let ((x (eval-ineq x env))
           (y (eval-ineq y env)))
       (or x y)))
    (('not x)
     (let ((x (eval-ineq x env)))
       (not x)))
    (('= var poly)
     (let ((x (eval-poly (bound-poly var) env))
           (y (eval-poly poly env)))
       (equal x y)))
    (('!= var poly)
     (let ((x (eval-poly (bound-poly var) env))
           (y (eval-poly poly env)))
       (not (equal x y))))
    (('< x y)
     (let ((x (eval-poly (bound-poly x) env))
           (y (eval-poly y env)))
       (< x y)))
    (('<= x y)
     (let ((x (eval-poly (bound-poly x) env))
           (y (eval-poly y env)))
       (<= x y)))
    (('> x y)
     (let ((x (eval-poly (bound-poly x) env))
           (y (eval-poly y env)))
       (> x y)))
    (('>= x y)
     (let ((x (eval-poly (bound-poly x) env))
           (y (eval-poly y env)))
       (>= x y)))
    (& nil)))
```

Generalizer



```
(def::un generalize-ineq (term sln)
  (declare (xargs :signature ((t env-p) region-p)))
  (case-match term
    (('and x y)
     (let ((x (generalize-ineq x sln))
           (y (generalize-ineq y sln)))
       (and-regions x y sln)))
    (('or x y)
     (let ((x (generalize-ineq x sln))
           (y (generalize-ineq y sln)))
       (not-region (and-regions (not-region x) (not-region y) sln))))
    (('not x)
     (let ((x (generalize-ineq x sln)))
       (not-region x)))
    (('= var poly)
     (let ((x (bound-poly var))
           (y (poly-fix! poly)))
       (normalize-equal-0 (sub x y) sln)))
    (('!= var poly)
     (let ((x (bound-poly var))
           (y (poly-fix! poly)))
       (not-region (normalize-equal-0 (sub x y) sln))))
    (('< x y)
     (let ((x (bound-poly x))
           (y (poly-fix! y)))
       (normalize-gt-0 :exclusive (sub y x) sln)))
    (('<= x y)
     (let ((x (bound-poly x))
           (y (poly-fix! y)))
       (normalize-gt-0 :inclusive (sub y x) sln)))
    (('> x y)
     (let ((x (bound-poly x))
           (y (poly-fix! y)))
       (normalize-gt-0 :exclusive (sub x y) sln)))
    (('>= x y)
     (let ((x (bound-poly x))
           (y (poly-fix! y)))
       (normalize-gt-0 :inclusive (sub x y) sln)))
    (& (negated-region nil))))
```

Generalization Correctness



```
(defthm inv1-generalize-ineq
  (implies
   (env-p sln)
   (and (wf-region-p (generalize-ineq term sln) sln)
        (iff (eval-region (generalize-ineq term sln) sln)
             (eval-ineq term sln))))
  :hints (("Goal" :induct (generalize-ineq term sln)
           :do-not-induct t)
          (and stable-under-simplificationp
               '(:in-theory (enable eval-ineq)))))

(def::signature generalize-ineq (t env-p) (lambda (x) (wf-region-p x x1))
  :hints (("Goal" :in-theory (disable wf-region-p))))

(in-theory (disable wf-region-p alt-eval-region))

(defthm inv2-generalize-ineq
  (implies
   (and
    (env-p sln)
    (iff (eval-ineq term sln)
         (not (eval-ineq term any))))
   (iff (eval-region (generalize-ineq term sln) any)
        (eval-ineq term any)))
  :hints (("Goal" :induct (generalize-ineq term sln)
           :do-not-induct t)
          (and stable-under-simplificationp
               '(:in-theory (enable eval-ineq)))))
```

Evaluating Solution on Generalization must be the same as Evaluating Solution on original expression

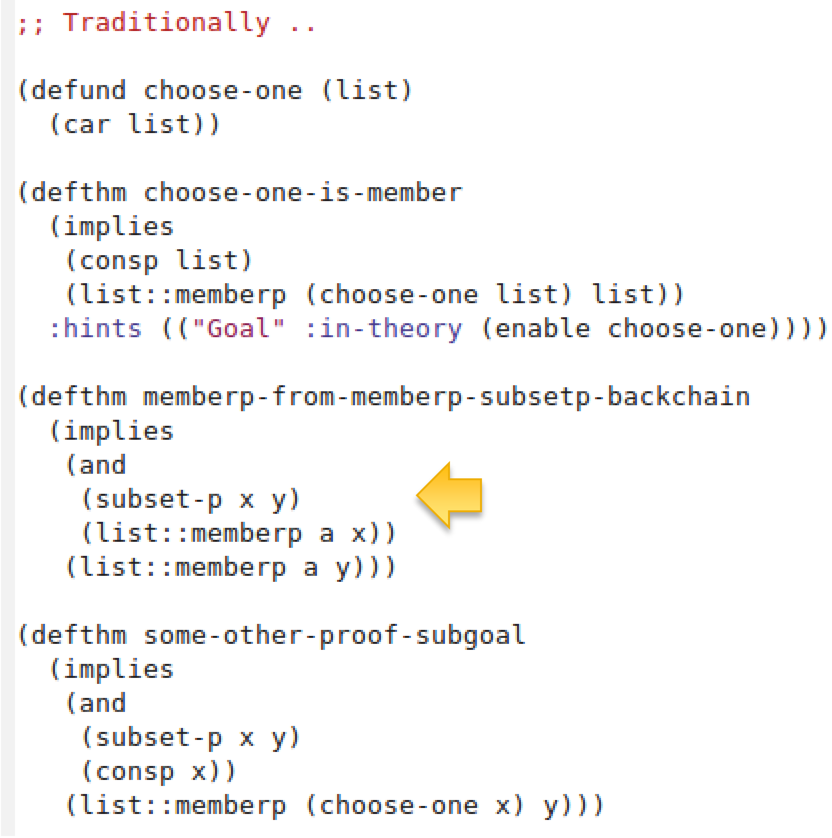
Any input whose evaluation differs from that of the solution on the original expression must also differ on the Generalization

Establishes Correctness of detailed Generalization Procedure Model against our Formal specification

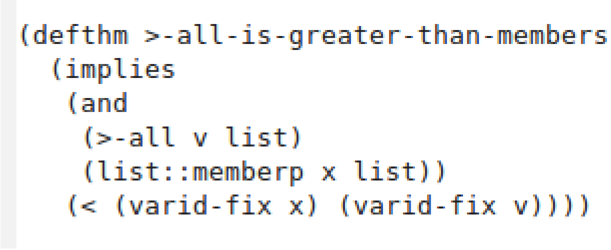
Proof Pearls (Weird Things Dave Does in ACL2)

- Non-Traditional Congruences (nary)
  - Used to verify variable ordering invariants
- Delayed/Partial Termination (def::ung)
  - Used to admit/reason about awkward functions
- Question about ACL2 Linear Capabilities

Traditional backchaining (member/subset)

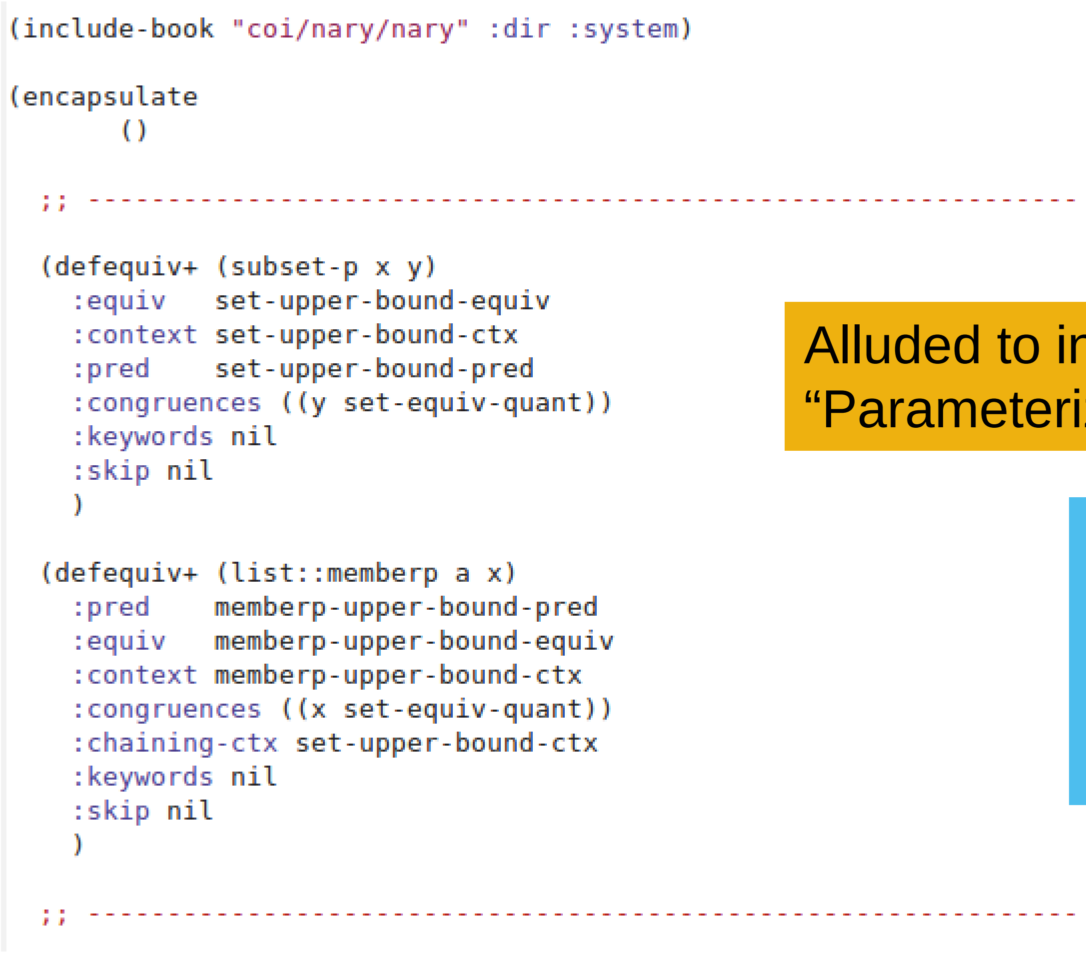


```
;; Traditionally ..

(defund choose-one (list)
  (car list))

(defthm choose-one-is-member
  (implies
   (consp list)
   (list::memberp (choose-one list) list))
  :hints (("Goal" :in-theory (enable choose-one))))

(defthm memberp-from-memberp-subsetp-backchain
  (implies
   (and
    (subset-p x y)
    (list::memberp a x))
   (list::memberp a y)))

(defthm some-other-proof-subgoal
  (implies
   (and
    (subset-p x y)
    (consp x))
   (list::memberp (choose-one x) y)))
```



```
(defthm >-all-is-greater-than-members
  (implies
   (and
    (>-all v list)
    (list::memberp x list))
   (< (varid-fix x) (varid-fix v))))
```

Non-Traditional (one-way) “equivalences”

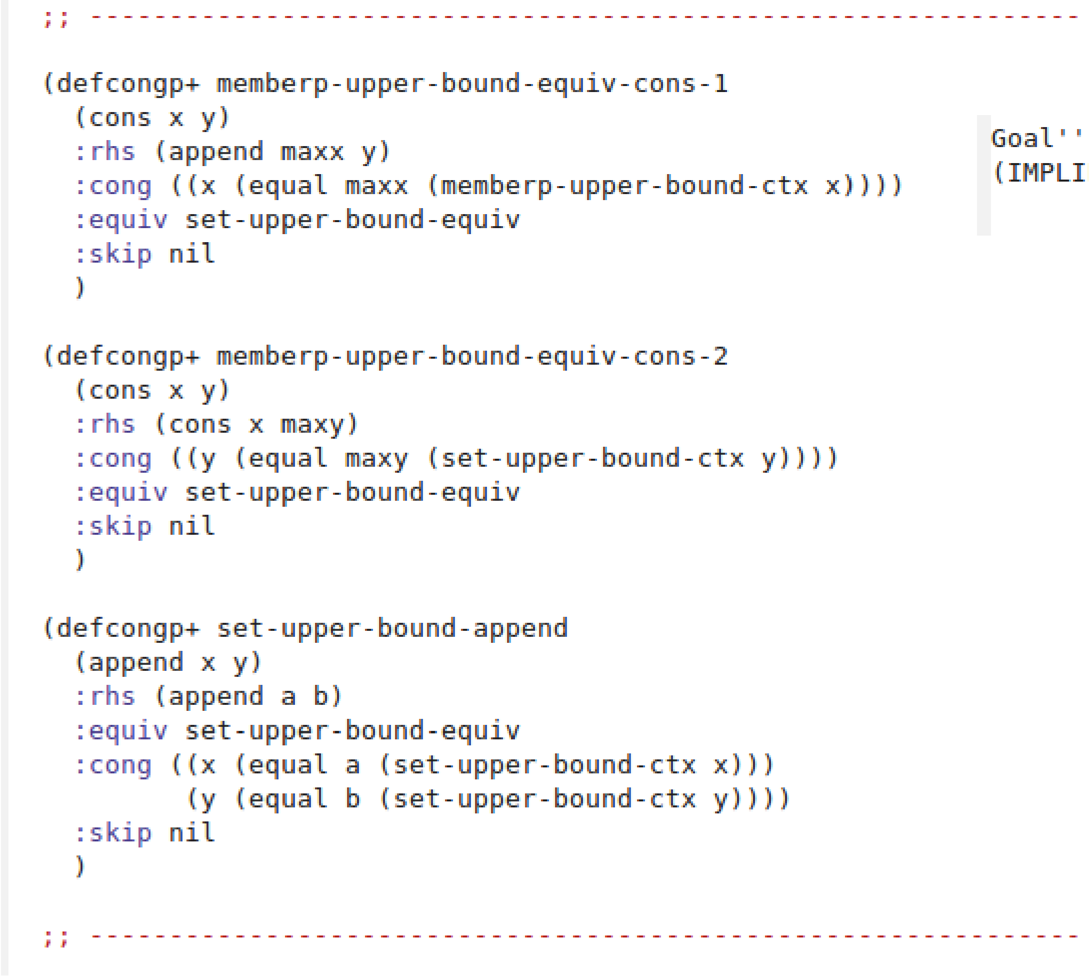


```
(include-book "coi/nary/nary" :dir :system)

(encapsulate
       ()

  ;; --------------------------------------------------------------

  (defequiv+ (subset-p x y)
    :equiv   set-upper-bound-equiv
    :context set-upper-bound-ctx
    :pred    set-upper-bound-pred
    :congruences ((y set-equiv-quant))
    :keywords nil
    :skip nil
    )

  (defequiv+ (list::memberp a x)
    :pred    memberp-upper-bound-pred
    :equiv   memberp-upper-bound-equiv
    :context memberp-upper-bound-ctx
    :congruences ((x set-equiv-quant))
    :chaining-ctx set-upper-bound-ctx
    :keywords nil
    :skip nil
    )

  ;; --------------------------------------------------------------
```

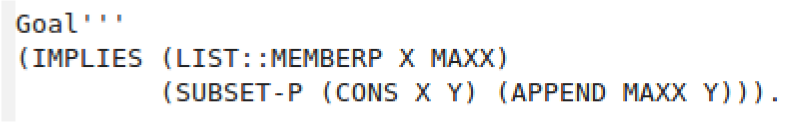
Alluded to in 2006 Workshop:
“Parameterized Congruences in ACL2”

```
(defthm generalized-cong-rule
  (implies
    (< x a)
    (equal (foo x)
           (foo a))))
```

Non-Traditional Congruences

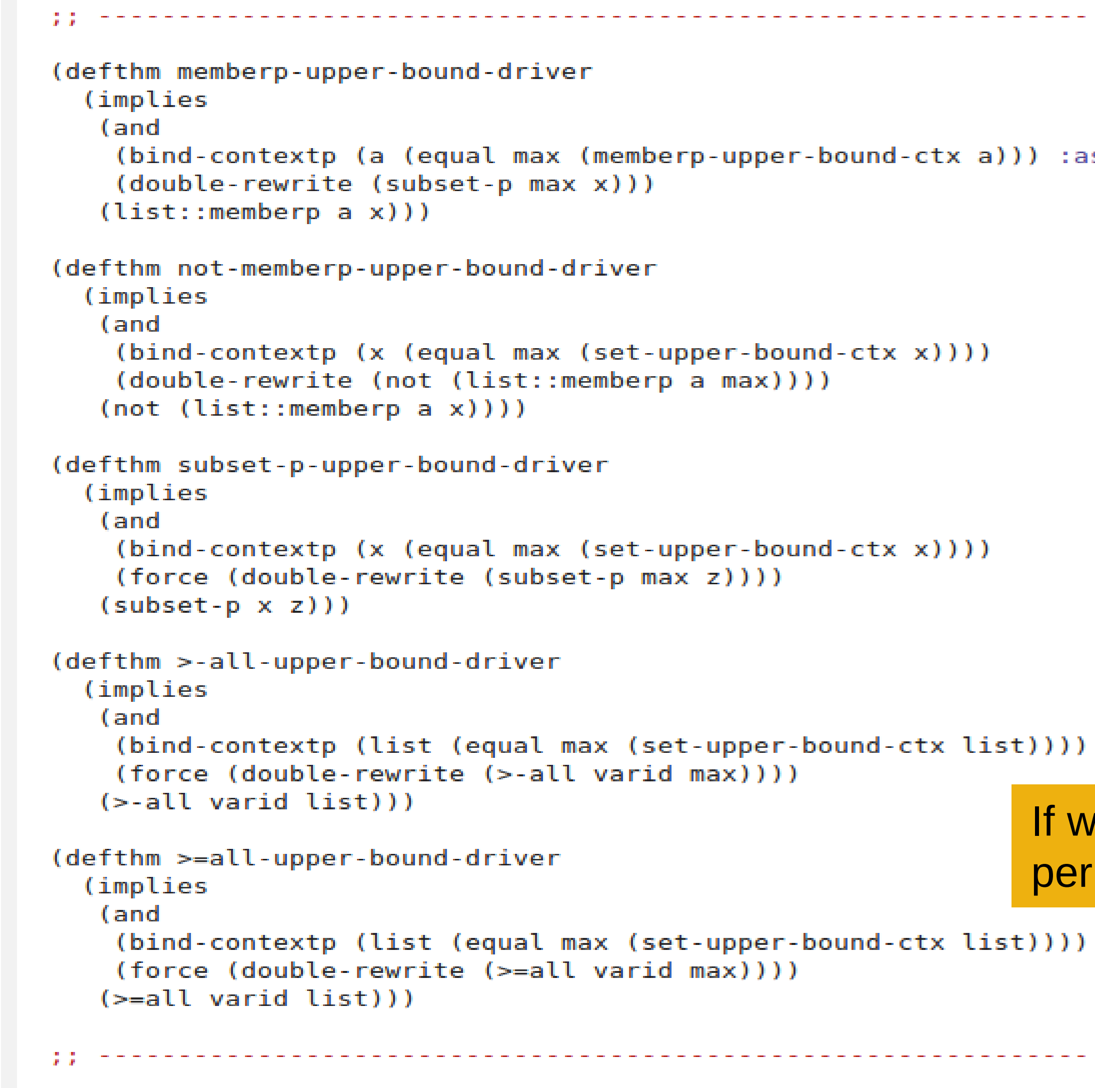


```
;; ------------------------------------------------------------

(defcongp+ memberp-upper-bound-equiv-cons-1
  (cons x y)
  :rhs (append maxx y)
  :cong ((x (equal maxx (memberp-upper-bound-ctx x))))
  :equiv set-upper-bound-equiv
  :skip nil
  )

(defcongp+ memberp-upper-bound-equiv-cons-2
  (cons x y)
  :rhs (cons x maxy)
  :cong ((y (equal maxy (set-upper-bound-ctx y))))
  :equiv set-upper-bound-equiv
  :skip nil
  )

(defcongp+ set-upper-bound-append
  (append x y)
  :rhs (append a b)
  :equiv set-upper-bound-equiv
  :cong ((x (equal a (set-upper-bound-ctx x)))
         (y (equal b (set-upper-bound-ctx y))))
  :skip nil
  )

;; ------------------------------------------------------------
```

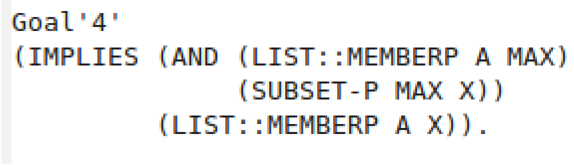


```
Goal'''
(IMPLIES (LIST::MEMBERP X MAXX)
         (SUBSET-P (CONS X Y) (APPEND MAXX Y))).
```

Non-Traditional “Driver” Rules



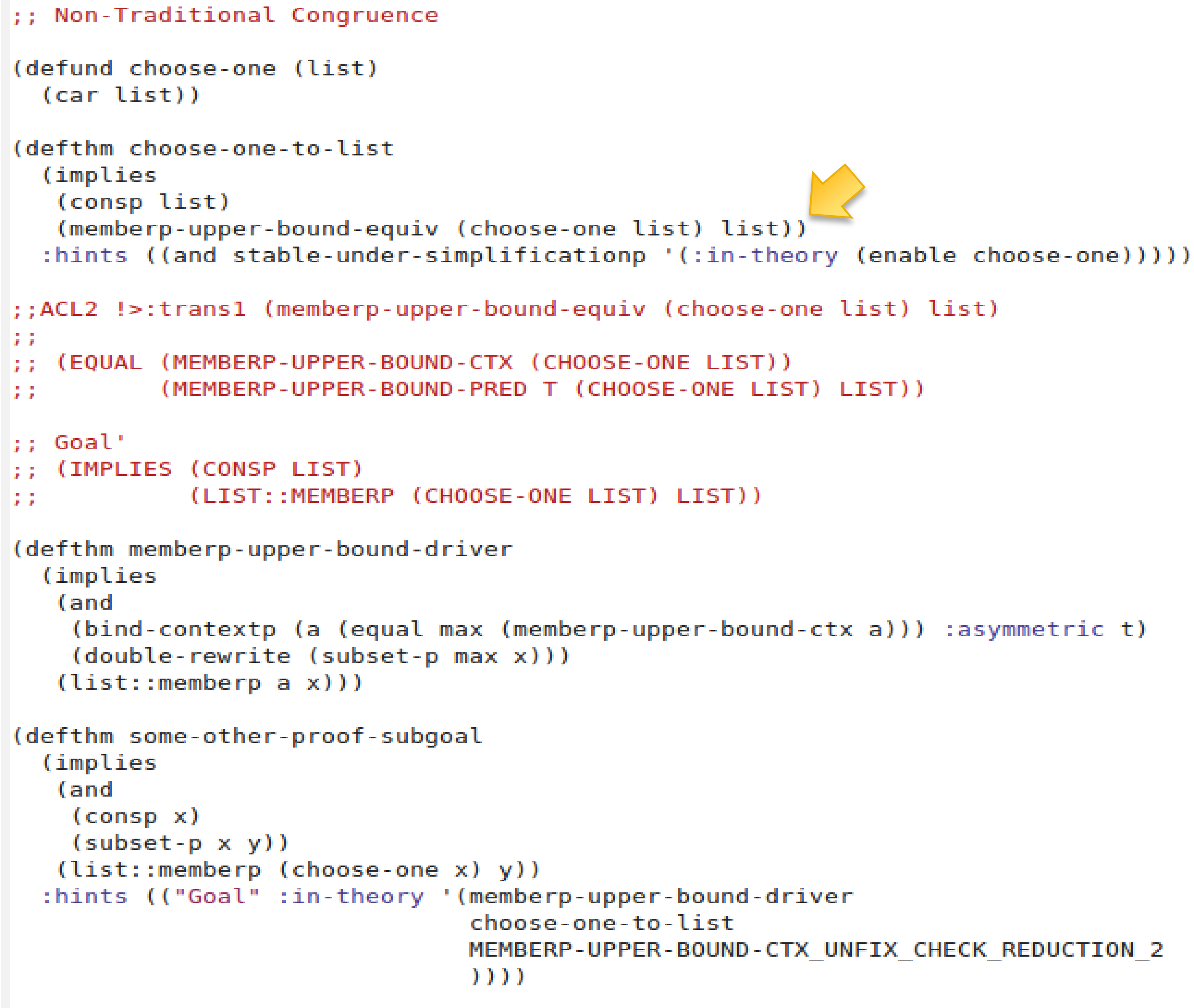
```
;; -----------------------------------------------------------------

(defthm memberp-upper-bound-driver
  (implies
   (and
    (bind-contextp (a (equal max (memberp-upper-bound-ctx a))) :asymmetric t)
    (double-rewrite (subset-p max x)))
   (list::memberp a x)))

(defthm not-memberp-upper-bound-driver
  (implies
   (and
    (bind-contextp (x (equal max (set-upper-bound-ctx x))))
    (double-rewrite (not (list::memberp a max))))
   (not (list::memberp a x))))

(defthm subset-p-upper-bound-driver
  (implies
   (and
    (bind-contextp (x (equal max (set-upper-bound-ctx x))))
    (force (double-rewrite (subset-p max z))))
   (subset-p x z)))

(defthm >-all-upper-bound-driver
  (implies
   (and
    (bind-contextp (list (equal max (set-upper-bound-ctx list))))
    (force (double-rewrite (>-all varid max))))
   (>-all varid list)))

(defthm >=all-upper-bound-driver
  (implies
   (and
    (bind-contextp (list (equal max (set-upper-bound-ctx list))))
    (force (double-rewrite (>=all varid max))))
   (>=all varid list)))

;; -----------------------------------------------------------------
```



```
Goal'4'
(IMPLIES (AND (LIST::MEMBERP A MAX)
              (SUBSET-P MAX X))
         (LIST::MEMBERP A X)).
```

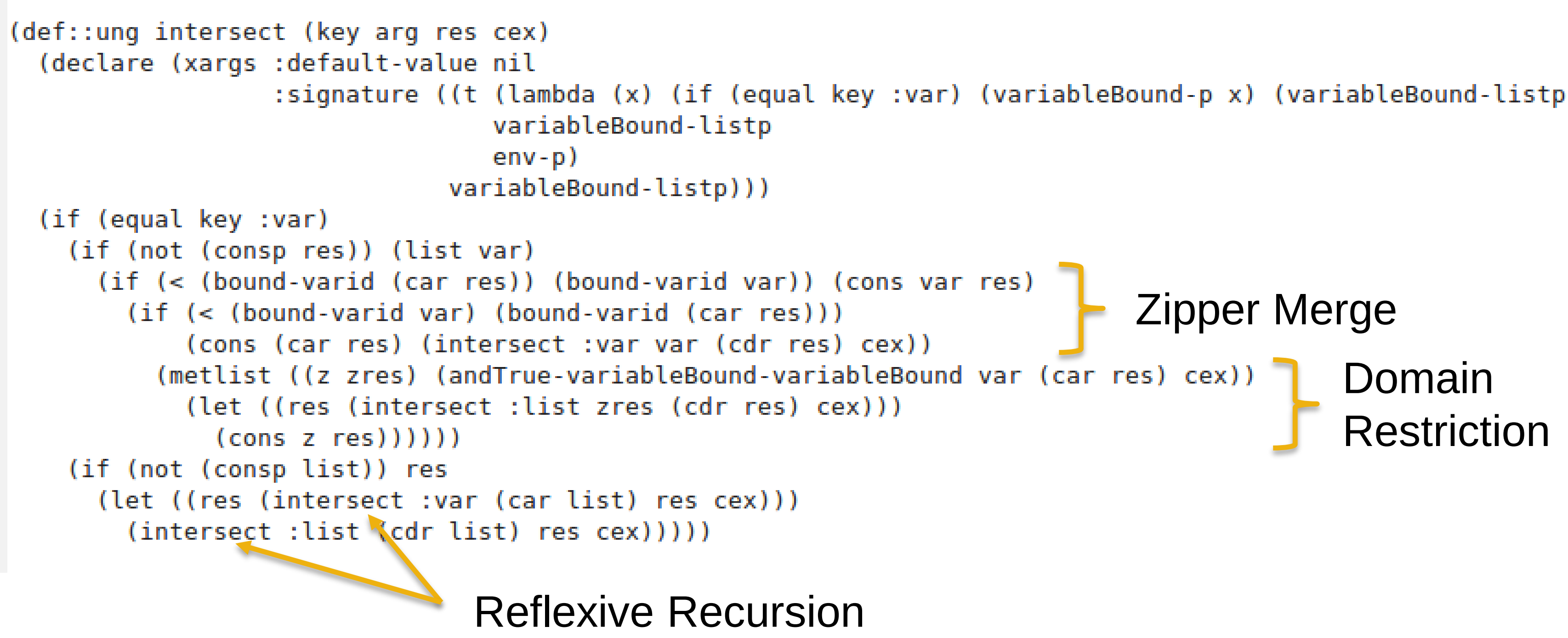
If we modified the ancestors check (?) perhaps these could be rewrite rules ..

Proof Using Non-Traditional Congruences



```
;; Non-Traditional Congruence

(defund choose-one (list)
  (car list))

(defthm choose-one-to-list
  (implies
   (consp list)
   (memberp-upper-bound-equiv (choose-one list) list))
  :hints ((and stable-under-simplificationp '(:in-theory (enable choose-one)))))

;;ACL2 !>:trans1 (memberp-upper-bound-equiv (choose-one list) list)
;;
;; (EQUAL (MEMBERP-UPPER-BOUND-CTX (CHOOSE-ONE LIST))
;;        (MEMBERP-UPPER-BOUND-PRED T (CHOOSE-ONE LIST) LIST))

;; Goal'
;; (IMPLIES (CONSP LIST)
;;          (LIST::MEMBERP (CHOOSE-ONE LIST) LIST))

(defthm memberp-upper-bound-driver
  (implies
   (and
    (bind-contextp (a (equal max (memberp-upper-bound-ctx a))) :asymmetric t)
    (double-rewrite (subset-p max x)))
   (list::memberp a x)))

(defthm some-other-proof-subgoal
  (implies
   (and
    (consp x)
    (subset-p x y))
   (list::memberp (choose-one x) y))
  :hints (("Goal" :in-theory '(memberp-upper-bound-driver
                               choose-one-to-list
                               MEMBERP-UPPER-BOUND-CTX_UNFIX_CHECK_REDUCTION_2
                               ))))
```

Admitting Awkward Functions (def::ung)



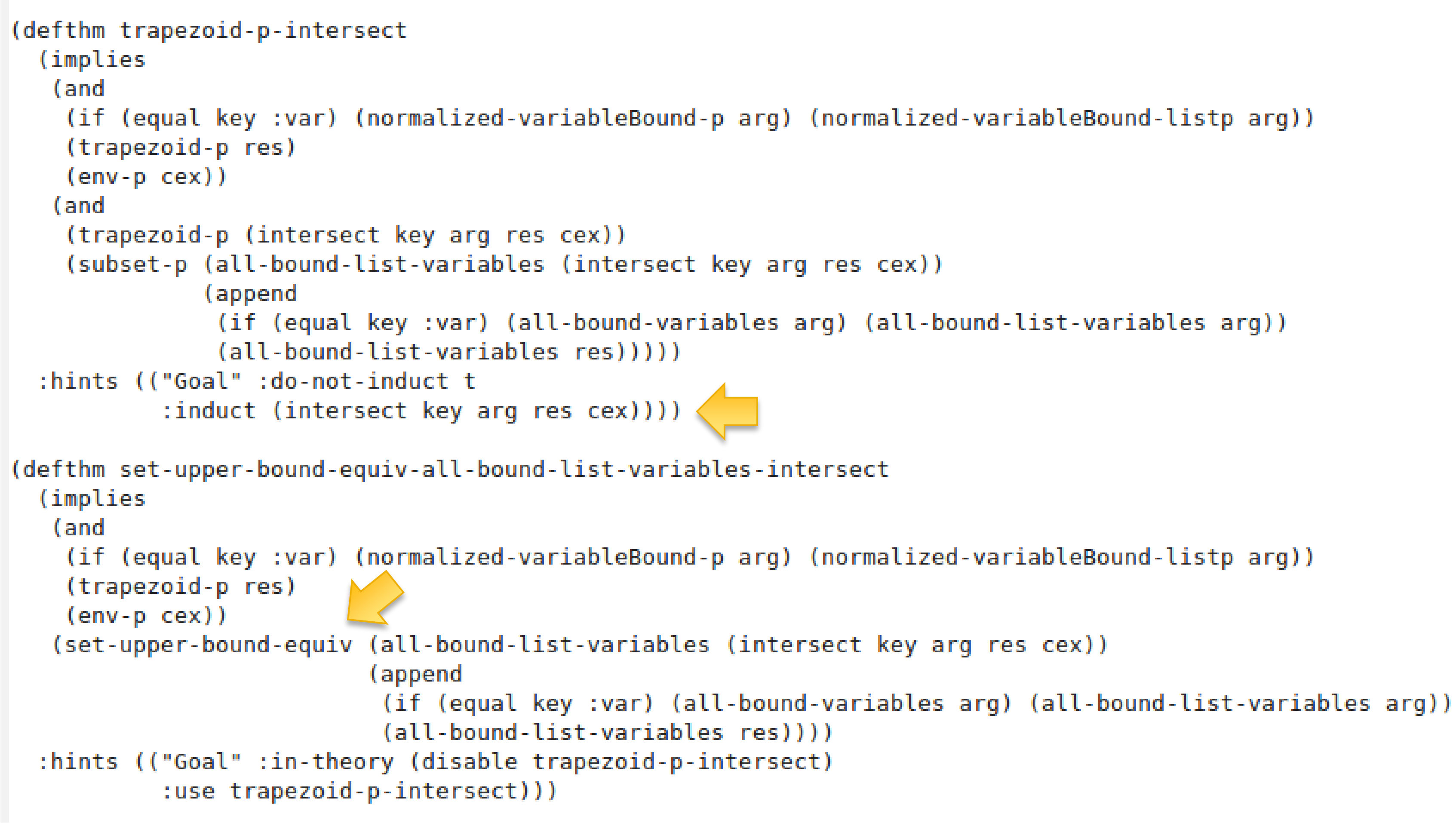
```
(def::ung intersect (key arg res cex)
  (declare (xargs :default-value nil
                  :signature ((t (lambda (x) (if (equal key :var) (variableBound-p x) (variableBound-listp x)))
                                 variableBound-listp
                                 env-p)
                              variableBound-listp)))
  (if (equal key :var)
    (if (not (consp res)) (list var)
      (if (< (bound-varid (car res)) (bound-varid var)) (cons var res)
        (if (< (bound-varid var) (bound-varid (car res)))
            (cons (car res) (intersect :var var (cdr res) cex))
          (metlist ((z zres) (andTrue-variableBound-variableBound var (car res) cex))
            (let ((res (intersect :list zres (cdr res) cex)))
              (cons z res))))))
    (if (not (consp list)) res
      (let ((res (intersect :var (car list) res cex)))
        (intersect :list (cdr list) res cex)))))
```

Zipper Merge

Domain Restriction

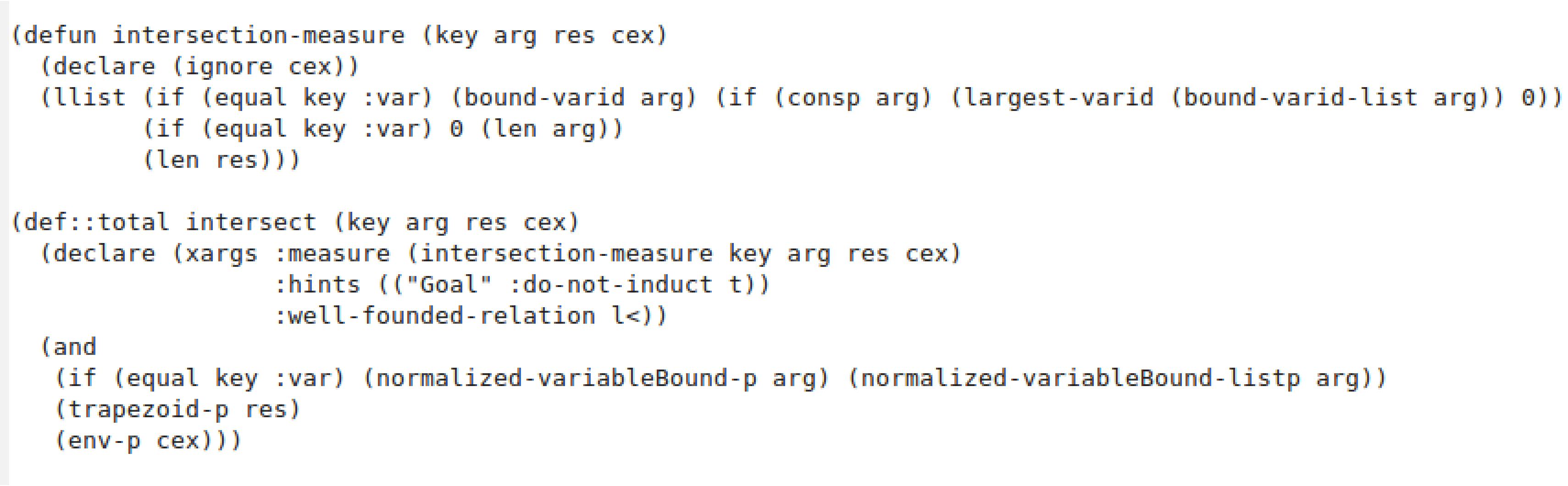
Reflexive Recursion

intersect type theorems



```
(defthm trapezoid-p-intersect
  (implies
   (and
    (if (equal key :var) (normalized-variableBound-p arg) (normalized-variableBound-listp arg))
    (trapezoid-p res)
    (env-p cex))
   (and
    (trapezoid-p (intersect key arg res cex))
    (subset-p (all-bound-list-variables (intersect key arg res cex))
              (append
               (if (equal key :var) (all-bound-variables arg) (all-bound-list-variables arg))
               (all-bound-list-variables res))))))
  :hints (("Goal" :do-not-induct t
           :induct (intersect key arg res cex))))

(defthm set-upper-bound-equiv-all-bound-list-variables-intersect
  (implies
   (and
    (if (equal key :var) (normalized-variableBound-p arg) (normalized-variableBound-listp arg))
    (trapezoid-p res)
    (env-p cex))
   (set-upper-bound-equiv (all-bound-list-variables (intersect key arg res cex))
                          (append
                           (if (equal key :var) (all-bound-variables arg) (all-bound-list-variables arg))
                           (all-bound-list-variables res))))
  :hints (("Goal" :in-theory (disable trapezoid-p-intersect)
           :use trapezoid-p-intersect)))
```

intersect measure and (conditional) termination



```
(defun intersection-measure (key arg res cex)
  (declare (ignore cex))
  (llist (if (equal key :var) (bound-varid arg) (if (consp arg) (largest-varid (bound-varid-list arg)) 0))
         (if (equal key :var) 0 (len arg))
         (len res)))

(def::total intersect (key arg res cex)
  (declare (xargs :measure (intersection-measure key arg res cex)
                  :hints (("Goal" :do-not-induct t))
                  :well-founded-relation l<))
  (and
   (if (equal key :var) (normalized-variableBound-p arg) (normalized-variableBound-listp arg))
   (trapezoid-p res)
   (env-p cex)))
```

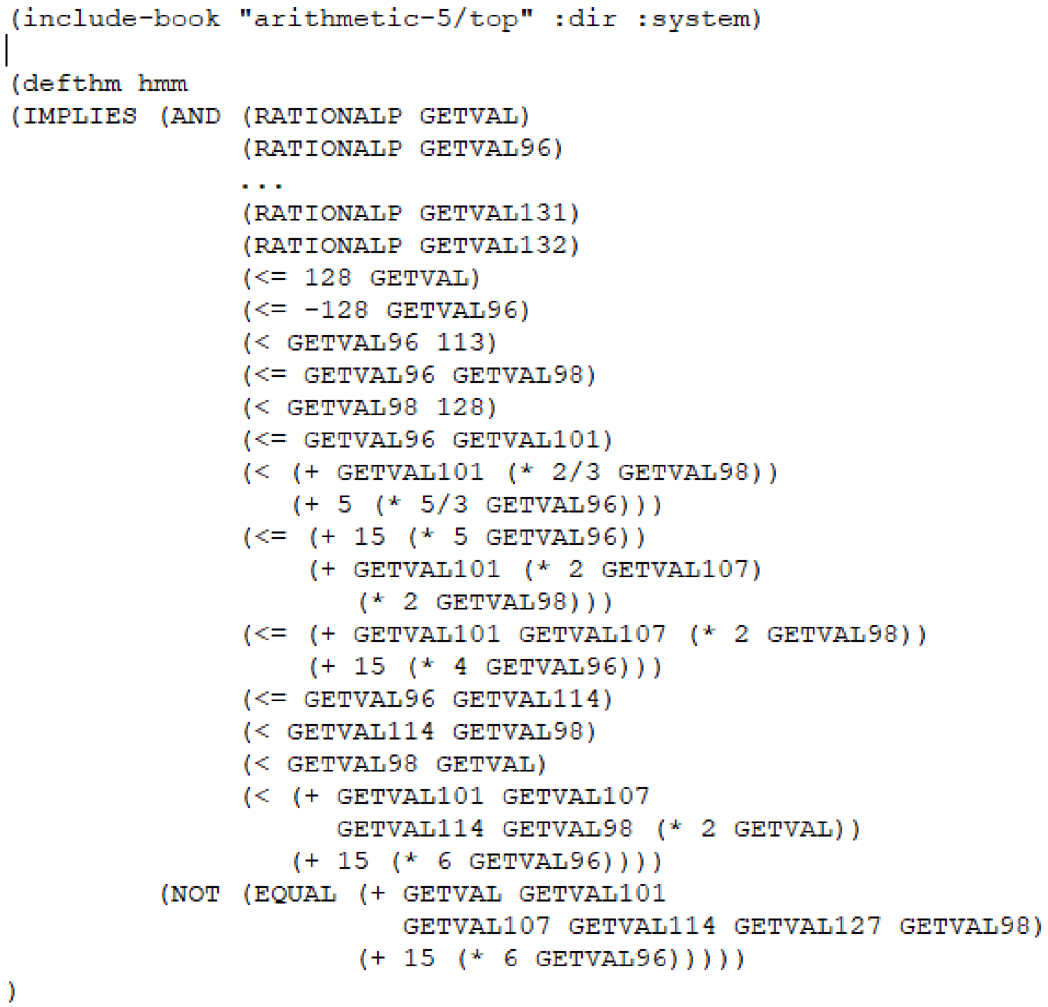
What are ACL2’s Linear Reasoning Capabilities?

- Doublecheck
  - Framework can emit ACL2 theorems during generalization
  - Instances of invariants 1 & 2
    - Trapezoids : Conjunctions of linear constraints

- Original Theorems Failed/Took Forever
  - Function Applications rather than Variables

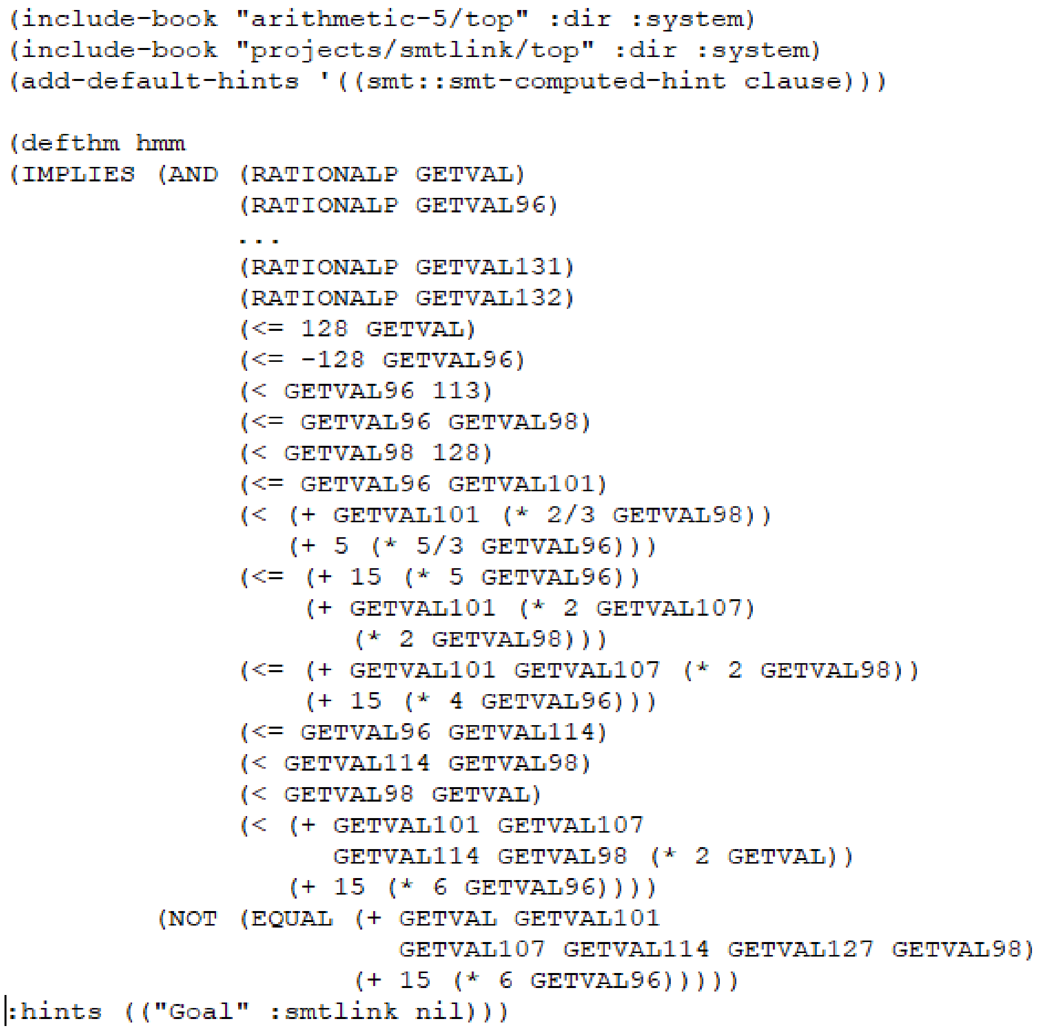
- Generalized Theorems Don’t Prove Consistently

What are ACL2’s Linear Reasoning Capabilities?



```
(include-book "arithmetic-5/top" :dir :system)

(defthm hmm
(IMPLIES (AND (RATIONALP GETVAL)
              (RATIONALP GETVAL96)
              ...
              (RATIONALP GETVAL131)
              (RATIONALP GETVAL132)
              (<= 128 GETVAL)
              (<= -128 GETVAL96)
              (< GETVAL96 113)
              (<= GETVAL96 GETVAL98)
              (< GETVAL98 128)
              (<= GETVAL96 GETVAL101)
              (< (+ GETVAL101 (* 2/3 GETVAL98))
                 (+ 5 (* 5/3 GETVAL96)))
              (<= (+ 15 (* 5 GETVAL96))
                  (+ GETVAL101 (* 2 GETVAL107)
                     (* 2 GETVAL98)))
              (<= (+ GETVAL101 GETVAL107 (* 2 GETVAL98))
                  (+ 15 (* 4 GETVAL96)))
              (<= GETVAL96 GETVAL114)
              (< GETVAL114 GETVAL98)
              (< GETVAL98 GETVAL)
              (< (+ GETVAL101 GETVAL107
                    GETVAL114 GETVAL98 (* 2 GETVAL))
                 (+ 15 (* 6 GETVAL96))))
         (NOT (EQUAL (+ GETVAL GETVAL101
                        GETVAL107 GETVAL114 GETVAL127 GETVAL98)
                     (+ 15 (* 6 GETVAL96))))))
)
```

What are ACL2’s Linear Reasoning Capabilities?



```
(include-book "arithmetic-5/top" :dir :system)
(include-book "projects/smtlink/top" :dir :system)
(add-default-hints '((smt::smt-computed-hint clause)))

(defthm hmm
(IMPLIES (AND (RATIONALP GETVAL)
              (RATIONALP GETVAL96)
              ...
              (RATIONALP GETVAL131)
              (RATIONALP GETVAL132)
              (<= 128 GETVAL)
              (<= -128 GETVAL96)
              (< GETVAL96 113)
              (<= GETVAL96 GETVAL98)
              (< GETVAL98 128)
              (<= GETVAL96 GETVAL101)
              (< (+ GETVAL101 (* 2/3 GETVAL98))
                 (+ 5 (* 5/3 GETVAL96)))
              (<= (+ 15 (* 5 GETVAL96))
                  (+ GETVAL101 (* 2 GETVAL107)
                     (* 2 GETVAL98)))
              (<= (+ GETVAL101 GETVAL107 (* 2 GETVAL98))
                  (+ 15 (* 4 GETVAL96)))
              (<= GETVAL96 GETVAL114)
              (< GETVAL114 GETVAL98)
              (< GETVAL98 GETVAL)
              (< (+ GETVAL101 GETVAL107
                    GETVAL114 GETVAL98 (* 2 GETVAL))
                 (+ 15 (* 6 GETVAL96))))
         (NOT (EQUAL (+ GETVAL GETVAL101
                        GETVAL107 GETVAL114 GETVAL127 GETVAL98)
                     (+ 15 (* 6 GETVAL96)))))
:hints (("Goal" :smtlink nil)))
```

What are ACL2's Linear Reasoning Capabilities?

- Doublecheck
  - Framework can emit ACL2 theorems during generalization
  - Instances of invariants 1 & 2
    - Trapezoids : Conjunctions of linear constraints

- Original Theorems Failed/Took Forever
  - Function Applications rather than Variables

- Generalized Theorems Don't Prove Consistently

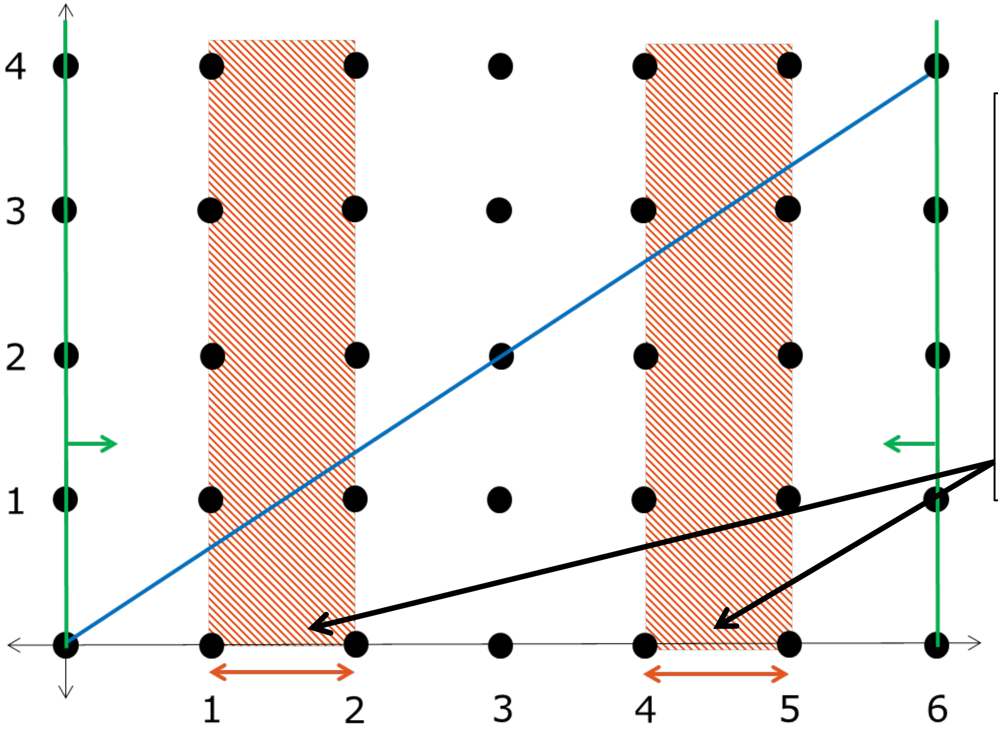
How does Linear Reasoning differ from LP?

Sampling (Oops ..)

Lower Bound	Variable	Upper Bound
100 <	x	< 200
3x - 290 = 40 <	y	< 640 = -3x + 970
y + x – 250 = 490 <	z	< -623 = -y + 7

Test Values
x = 110
y = 630
z = ??

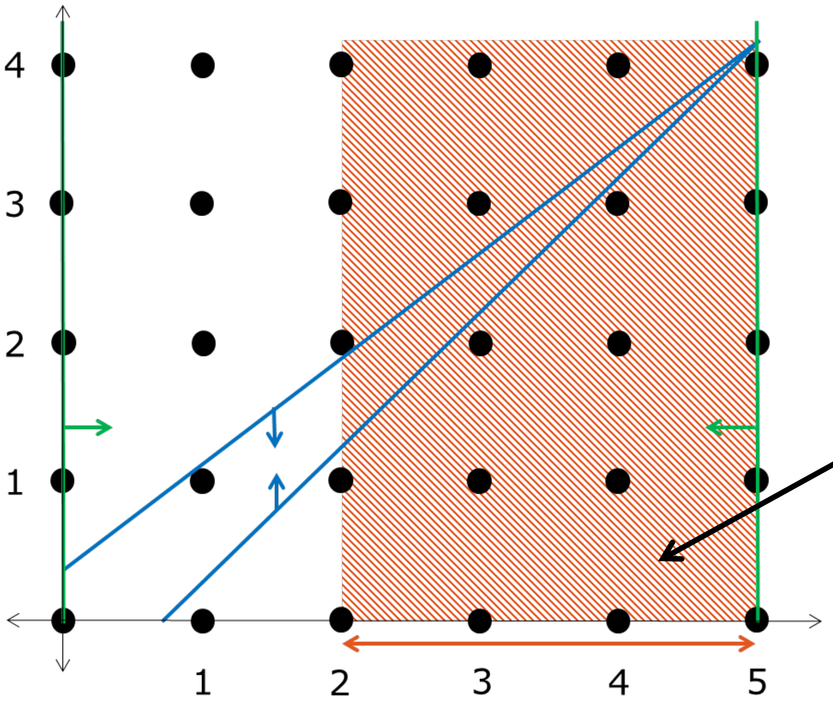
Integer Equality



If we choose a value of x in these regions, there is no integer value for y satisfying our constraints

Lower	Variable	Upper
0 <=	x	<= 6
(2/3)x =	y	= (2/3)x

Integer Intervals



If we choose a value of x in this region, there is no integer value for y satisfying our constraints

Lower	Variable	Upper
0 <=	x	<= 5
.9x – .4 <=	y	<= .74x + .4

Future Work

- We have defined a technique for restricting trapezoids
  - Restricted Trapezoids can be sampled
    - Without Inconsistencies
    - Without Backtracking
  - Even for Integer Valued Variables

- Remaining Challenge:
  - Prove that Restriction Works