Invariant Detection

One-Slide Summary

- An **invariant** is a logical formula that is always true at a particular program location.

- Given a program and a program location, **invariant detection** learns an invariant that is true whenever execution reaches that location. Dynamic invariant detect uses execution traces to learn invariants.

- The **Daikon** algorithm for dynamic invariant detection enumerates simple candidate invariants and retains only those that hold on all traces.

- The **Dig** algorithm for dynamic invariant detection generates nonlinear polynomial and linear array invariants via SMT, polyhedra, and equations.

Motivation

- Backward axiomatic semantics requires loop invariants for verification conditions.

- Forward axiomatic semantics (symbolic execution) requires loop invariants for all backward branch targets.

- Recursive functions may also require pre- and post-conditions in these frameworks.

- How many have you seen in real life?

Finding Invariants Manually

Proof of a Program: FIND

C. A. R. HOARE
Queen's University, * *Belfast, Ireland*

A proof is given of the correctness of the algorithm "Find." First, an informal description is given of the purpose of the program and the method used. A systematic technique is described for constucting the program proof during the process of coding it, in such a way as to prevent the intrusion of logical errors. The proof of termination is treated as a separate exercise. Finally, some conclusions relating to general programming methodology are drawn.

KEY WORDS AND PHRASES: proofs of programs, programming methodology, program documentation, program correctness, theory of programming
CR CATEGORIES: 4.0, 4.22, 5.21, 5.23, 5.24

1. Introduction

In a number of papers [1, 2, 3] the desirability of proving the correctness of programs has been suggested and this has been illustrated by proofs of simple example programs. In this paper the construction of the proof of a useful, efficient, and nontrivial program, using a method based on invariants, is shown. It is suggested that if a proof is constructed as part of the coding process for an algorithm, it is hardly more laborious than the traditional practice of program testing.

sort the whole array. If the array is small, this would be a good method; but if the array is large, the time taken to sort it will also be large. The Find program is designed to take advantage of the weaker requirements to save much of the time which would be involved in a full sort.

The usefulness of the Find program arises from its application to the problem of finding the median or other quantiles of a set of observations stored in a computer array. For example, if N is odd and f is set to $(N + 1)/2$, the effect of the Find program will be to place an observation with value equal to the median in $A[f]$. Similarly the first quartile may be found by setting f to $(N + 1)/4$, and so on.

The method used is based on the principle that the desired effect of Find is to move lower valued elements of the array to one end—the "left-hand" end—and higher valued elements of the array to the other end—the "right-hand" end. (See Table I(a)). This suggests that the array be scanned, starting at the left-hand end and moving rightward. Any element encountered which is small will remain where it is, but any element which is large should be moved up to the right-hand end of the array, in exchange for a small one. In order to find such a small element, a separate scan is made, starting at the right-hand end and moving leftward. In this scan, any large element encountered remains where it is; the first small element encountered is moved down to the left-hand end in exchange for the large element already encountered in the rightward scan. Then both scans can be resumed until the next exchange is necessary. The process is repeated until the scans meet somewhere in the middle of the array. It is then known that all elements to the left of this meeting point will be small, and all elements to the right will be

#4

Finding Invariants Manually

- Hoare's post-condition:

The required result is:

$$\forall p, q(1 \leq p \leq f \leq q \leq N \supset A[p] \leq A[f] \leq A[q])$$

[Found]

- Some intermediate invariants:

$$m \leq f \ \& \ \forall p, q(1 \leq p < m \leq q \leq N \supset A[p] \leq A[q])$$

[m-invariant]

Similarly, n is intended to point to the rightmost element of the middle part; it must never be less than f, and there will always be a split just to the right of it:

$$f \leq n \ \& \ \forall p, q(1 \leq p \leq n < q \leq N \supset A[p] \leq A[q])$$

[n-invariant]

Final Result

```
begin
  comment  This program operates on an array A[1:N], and a
    value of f(1 ≤ f ≤ N). Its effect is to rearrange the elements
    of A in such a way that:
      ∀p,q(1≤p≤f≤q≤N⊃A[p]≤A[f]≤A[q]);    ⟵
  integer m, n;   comment
      m ≤ f & ∀p,q(1≤p<m≤q≤N⊃A[p]≤A[q]),   ⟵
      f ≤ n & ∀p,q(1≤p≤n<q≤N⊃A[p]≤A[q]);   ⟵
    m := 1;   n := N;
  while m < n do
  begin integer r, i, j, w;
    comment
        m ≤ i & ∀p(1≤p<i⊃A[p]≤r),   ⟵
        j ≤ n & ∀q(j<q≤N⊃r≤A[q]);   ⟵
      r := A[f];   i := m;   j := n;
    while i ≤ j do
    begin while A[i] < r do i := i + 1;
      while r < A[j] do j := j − 1
      comment   A[j] ≤ r ≤ A[i];   ⟵
      if i ≤ j then
      begin w := A[i];   A[i] := A[j];   A[j] := w;
        comment   A[i] ≤ r ≤ A[j];   ⟵
        i := i + 1;   j := j − 1;
      end
    end increase i and decrease j;
    if f ≤ j then n := j
  else if i ≤ f then m := i
    else go to L
  end reduce middle part;
L:
end Find
```

"Reading Quiz"

- What is "FIND" all about?
- Why did it take seven pages?
- What program was Hoare actually proving?
  - Roman à clef
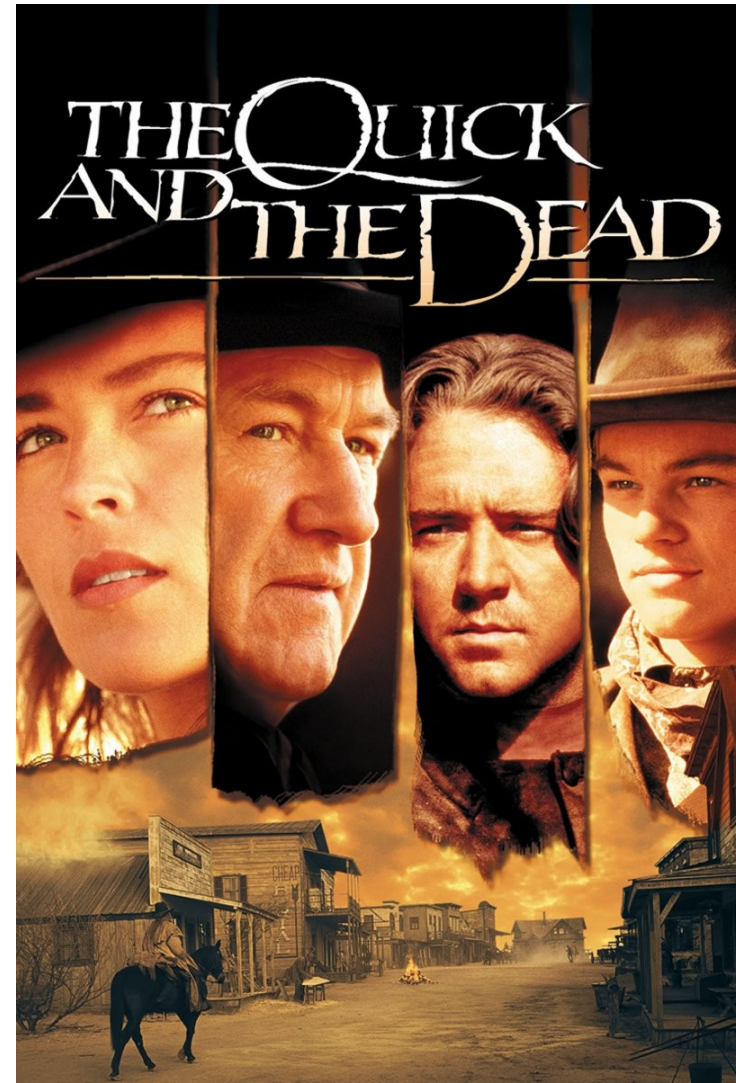
Cunning Plan

- Given a program location, if we could infer an invariant for that location, we could have …
  - Loop invariants (location = loop head)
  - Function pre-conditions (location = entry)
  - Function post-conditions (location = exit)
- Can we do this automatically?
- Two insights:
  - An invariant always holds on all executions
  - We can detect spurious false invariants

Dynamic Invariant Detection

- What if we require that the program come equipped with inputs?
  - An indicative workload
  - High-coverage test cases
- Since an invariant holds on every execution (by definition), any candidate invariant that fails even once can be tossed out!
- Plan: generate many candidate invariants, filter out the false ones!

I could while away the hours ...

- Given:
  - while b do c
- Instrument:
  - while b do (print Inv1; print Inv2; ... ; c)
  - Run on all tests, filter out on false
- How many candidate invariants are there?

verb
1. pass time in a leisurely manner.
   "a diversion to while away the long afternoons"
   synonyms: pass, spend, occupy, use up, fritter, kill
   "tennis helped to while away the time"

Invariant Templates

- Given program variables x, y, and z
  - x = c   constant
  - x != 0   non-zero
  - x >= c   bounds
  - y = ax + b   linear
  - x < y   ordering
  - (x + y) % b = a   math functions
  - z = ax + by + c   linear
- At most three variables at a time: finite!

Daikon

- The **Daikon** invariant detection algorithm
  - For every program location
    - For all triples of in-scope variables
      - Instantiate invariant templates to obtain candidate invariants
      - Instrument program
  - For every test case
    - Run instrumented program
    - Filter out any falsified candidate invariant
- Running time: cubic in in-scope variables, linear in test suite, linear in program

Daikon Weaknesses

- What could go wrong?

Daikon Weaknesses

- ## False Negatives

  - If your invariant does not fit a template, Daikon cannot find it

  - Example: l + u – 1 <= 2p <= l + u (bsearch pivot)

  - Example:

    The required result is:

    $$\forall p, q (1 \leq p \leq f \leq q \leq N \supset A[p] \leq A[f] \leq A[q])$$

    [Found]

- Nothing prevents Daikon from finding these
- But each increase in the language of candidate invariants bloats the runtime

Daikon Weaknesses

- False Positives from limited input

  - If you only test your sorting program on one input, [4;2;3], Daikon will learn output[0] = 2

  - But making high-coverage, high-adequacy tests is easy, no? That's why we're doing formal verification. Oh, right.

- False Positives from linguistic coincidence

  - Ex: ptr % 4 == 0

  - Ex: x <= MAX_INT

  - Not false, but not related to correctness.

Dynamic Invariant Detection

- Daikon is ill-suited for richer languages of invariants (e.g., non-linear relations, array relations, etc.) because all candidate invariants must be listed and considered.

- Idea:
  - Instead of listing invariants, list values, and induce invariants via constraint solving
  - Ex: instead of printing x>y, x<y, x>=y, etc., just print out x and y and figure out which is true later

Dig

- Apply three techniques we've already learned to learn rich invariants

- Nonlinear Equalities of Polynomials
  - via Equation Solving
- Nonlinear Inequalities of Polynomials
  - via Convex Polyhedra
- Linear Equalities of Arrays
  - via SMT Solving

Dig Example: Cohen's Division

```
1   def intdiv(x, y):
2       q = 0  // quotient
3       r = x  // remainder
4       while r ≥ y:
5           a = 1
6           b = y
7           while r ≥ 2b:
8               [L]  // loop invariant
9               a = 2a
10              b = 2b
11          r = r - b
12          q = q + a
13      return q
```

Cohen's Division on input (15,2)

```
1   def intdiv(x, y):
2       q = 0    // quotient
3       r = x    // remainder
4       while r ≥ y:
5           a = 1
6           b = y
7           while r ≥ 2b:
8               [L]   // loop invariant
9               a = 2a
10              b = 2b
11          r = r - b
12          q = q + a
13      return q
```

x	y	a	b	q	r
15	2	1	2	0	15
15	2	2	4	0	15
15	2	1	2	4	7

Cohen's Division on input (4,1)

```
1   def intdiv(x, y):
2       q = 0    // quotient
3       r = x    // remainder
4       while r ≥ y:
5           a = 1
6           b = y
7           while r ≥ 2b:
8               [L]  // loop invariant
9               a = 2a
10              b = 2b
11          r = r - b
12          q = q + a
13      return q
```

x	y	a	b	q	r
15	2	1	2	0	15
15	2	2	4	0	15
15	2	1	2	4	7
4	1	1	1	0	4
4	1	2	2	0	4

Cohen's Division Desires

```
1   def intdiv(x, y):
2       q = 0   // quotient
3       r = x   // remainder
4       while r ≥ y:
5           a = 1
6           b = y
7           while r ≥ 2b:
8               [L]   // loop invariant
9               a = 2a
10              b = 2b
11          r = r - b
12          q = q + a
13      return q
```

x	y	a	b	q	r
15	2	1	2	0	15
15	2	2	4	0	15
15	2	1	2	4	7
4	1	1	1	0	4
4	1	2	2	0	4

$$\{b = ya, x = qy + r, r \geq 2ya\}$$

Nonlinear Equalities

- Division example: x = qy + r   *(qy+r-x = 0)*
- GCD example: g = iA + jB   *(iA+jB-g = 0)*
- Goal:

Find equations of the form

$$c_0 + c_1 x + c_2 y + c_3 xy + \cdots + c_n x^d y^d = 0, \quad c_i \in \mathbb{R}$$

- Approach:
  - Pick a maximum Degree
  - Generate Terms
  - Equation Solving (ex: x + y = 2, x – y = 4)

Terms, Equations, Solutions

- ## Terms and Degrees

$$V = \{r, y, a\}; \quad \deg_{\max} = 2 \quad \Rightarrow \quad T = \{1, r, y, a, ry, ra, ya, r^2, y^2, a^2\}$$

- ## Equation Template

$$c_1 + c_2 r + c_3 y + c_4 a + c_5 ry + c_6 ra + c_7 ya + c_8 r^2 + c_9 y^2 + c_{10} a^2 = 0$$

- ## Systems of Equations

trace 1 : $\{r = 15, y = 2, a = 1\}$

eq 1 : $c_1 + 15c_2 + 2c_3 + c_4 + 30c_5 + 15c_6 + 2c_7 + 225c_8 + 4c_9 + c_{10} = 0$

- ## Solve for coefficients

$$V = \{x, y, a, b, q, r\}; \quad \deg_{\max} = 2 \quad \Rightarrow \quad \{b = ya, x = qy + r\}$$

How?

- Six variables (xyabqr) at deg 2 is 28 terms:

$$c_1 + c_2 y + c_3 q + c_4 x + c_5 b + c_6 a + c_7 r + c_8 y^2$$
$$+ c_9 qy + c_{10} xy + c_{11} by + c_{12} ay + c_{13} ry$$
$$+ c_{14} q^2 + c_{15} qx + c_{16} bq + c_{17} aq + c_{18} qr$$
$$+ c_{19} x^2 + c_{20} bx + c_{21} ax + c_{22} rx + c_{23} b^2$$
$$+ c_{24} ab + c_{25} br + c_{26} a^2 + c_{27} ar + c_{28} r^2 = 0.$$

- Use trace values for (xyabqr), find solutions
  - 28 terms + 5 traces = underconstrained!
  - c_4 = -V c_5 = T c_7 = c_9 = V
  - c_{12} = -T c_{16} = c_{27} = -U c_{21} = U c_{other} = 0

How Now?

- Solutions:
  - $c_4 = -V$ $c_5 = T$ $c_7 = c_9 = V$
  - $c_{12} = -T$ $c_{16} = c_{27} = -U$ $c_{21} = U$ $c_{other} = 0$
- Try T=0, U=0, V=1 (V alone → related vars)
- That gives $c_4 = -1$ $c_7 = c_9 = 1$

$$c_1 + c_2 y + c_3 q + \underline{c_4 x} + c_5 b + c_6 a + \underline{c_7 r} + c_8 y^2$$
$$+ \underline{c_9 qy} + c_{10} xy + c_{11} by + c_{12} ay + c_{13} ry$$
$$+ c_{14} q^2 + c_{15} qx + c_{16} bq + c_{17} aq + c_{18} qr$$
$$+ c_{19} x^2 + c_{20} bx + c_{21} ax + c_{22} rx + c_{23} b^2$$
$$+ c_{24} ab + c_{25} br + c_{26} a^2 + c_{27} ar + c_{28} r^2 = 0.$$

⟶ -x + r + qy = 0

aka

x = qy + r

#25

Q: Computer Science

- This American computer scientist graduated with a BA in Mathematics from the University of Virginia. He received the Turing award for "developing Model-Checking into a highly effective verification technology that is widely adopted in the hardware and software industries." He recognized that, unlike Hoare's annotation-heavy logical approach, Pnueli's temporal logics could be checked mechanically. With his student Ken McMillan he addressed the state space explosion problem via symbolic model checking (BDDs).

Nonlinear Inequalities

- Sqrt Example: x + err >= y*y >= x – err
- Goal:

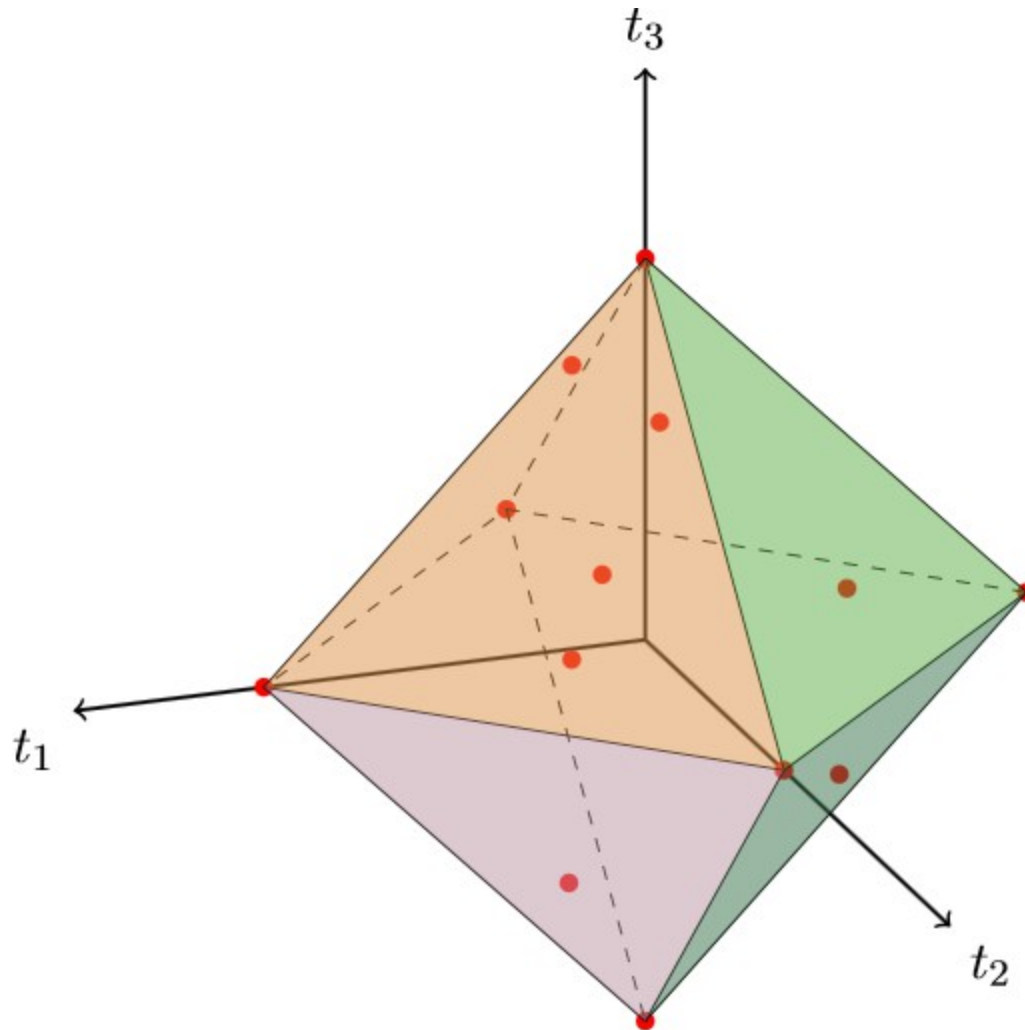Find inequalities of the form

$$c_0 + c_1 x + c_2 y + c_3 xy + \cdots + c_n x^d y^d \geq 0, \quad c_i \in \mathbb{R}$$

- Approach:
  - Pick maximum Degree
  - Represent trace values as polyhedra points
  - Build bounded convex polyhedron
  - Deduce new invariants from loop guard
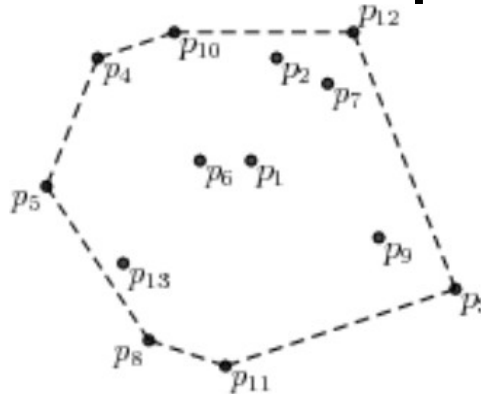
Polyhedra From Trace Points
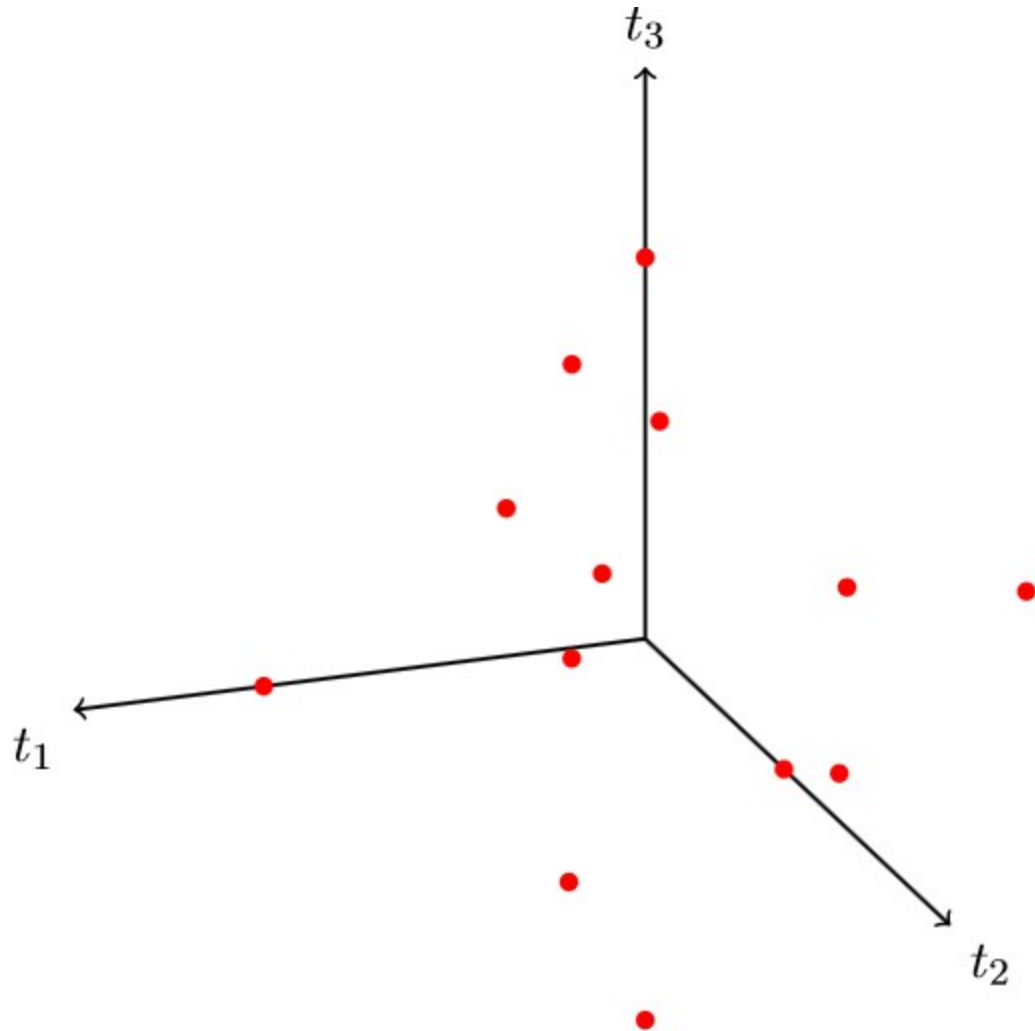
Polyhedra

Polyhedra Facets

- Facets of polyhedra correspond to inequalities

- Think Simplex!
- Given k points in n dimensions, can build the enclosing bounded convex polyhedron in $k^{n/2}$
- First trace from example before gives point: (1,15,2,1,30,15,2,225,4,1)
- One of the facets: r – 2ya >= 0

Comparison

- Simplex:
  - Given more than enough facets (edges, inequalities), find the convex space and enumerate the points

- Dig's Polyhedra Approach:
  - Given more than enough points, find the convex space and enumerate the facets (edges, inequalities)

Dig Deduction
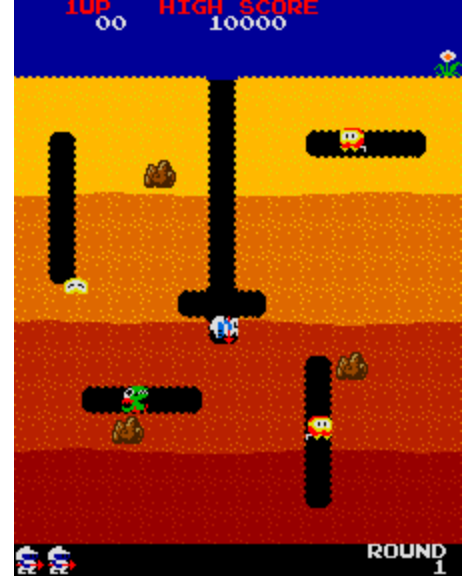
- Given a loop:
  - while (r >= 2b) { [L] ... }

- Obtain equality relations at location L:
  - b = ay   qy + r = x

- Add loop guard to each one and simplify:
  - r >= 2b && b = ay   → **r >= 2ay**
  - r >= 2b && qy + r = x   → **x - yq >= 2b**

Linear Array Relations

- AES Examples
  - block2State   r[i][j] = t[4i + j]
  - keySetupEnc8   r[i][j] = cypherKey[8i + j]
- Goal:

Find simple array relations of the form

$$A_1 + c_2 A_2 + \cdots + c_n A_n + c_0 = 0, \quad c_i \in \mathbb{R}$$

- Approach:
  - Flatten array elements into variables
  - Learn relations among elements, lift to indices

Array Encoding

- Represent array elements with new variables

trace 1   : $\{A = [-546, -641, 34], B = [-78, 3, -92, -34, 4]\}$

trace 2   : $\{A = [133, -333, -323], B = [19, 96, -48, -80, -47]\}$

trace 3   : $\ldots$

$$\Downarrow$$

	A_0	A_1	A_2	B_0	B_1	B_2	B_3	B_4
trace 1	-546	-641	34	-78	3	-92	-34	4
trace 2	-133	-333	-323	-19	96	-48	-80	-47
trace 3	$\ldots$							
$\vdots$								

- Find relations:

$$A_0 - 7B_0 = 0$$
$$A_1 - 7B_2 = 3$$
$$A_2 - 7B_4 = 6$$

Simple Array Relations

- Hypothesize: A[i] = l B[j] + k
- Find j:
  - Write relation between A[i] and B[j] as j=ip+q

$$A_0 - 7B_0 = 0 \quad \Rightarrow \quad 0 = 0p + q$$
$$A_1 - 7B_2 = 3 \quad \Rightarrow \quad 2 = 1p + q$$
$$A_2 - 7B_4 = 6 \quad \Rightarrow \quad 4 = 2p + q$$

  - Solve for p and q: $\{q = 0, p = 2\} \quad \Rightarrow \quad j = 2i$

$$\Rightarrow \quad A[i] = lB[2i] + k$$

- Find l and k: $A[i] = 7B[2i] + 3i$

Nested Array Relations

- AES Example: invSubBytes   R[i][j] = S[T[i][j]]
- Goal:

Find nested array relations of the grammar

$$A[i_1] \cdots [i_k] \;\mapsto\; e$$
$$e \;\mapsto\; B[e] \cdots [e]$$

E.g. $A[i][j] = B[C[j+3]][D[E[2i+j]]]$

- Approach:
  - Reachability finds possible nesting relations
  - Reduce to satisfiability problem, use SMT

Finding Nested Array Relations

- Example. Given (from traces):

  $A = [7, 1, -3], B = [1, -3, 5, 1, 0, 7, 1], C = [8, 5, 6, 6, 2, 1, 4]$

- Generate candidate nestings:

  $A[i] = B[\ldots], A[i] = C[\ldots], \ldots, C[i] = A[\ldots], A[i] = B[C[\ldots]], \ldots$

- Validate nestings:

  Discard $B[i] = C[\ldots]$ because $B[1] \notin C$
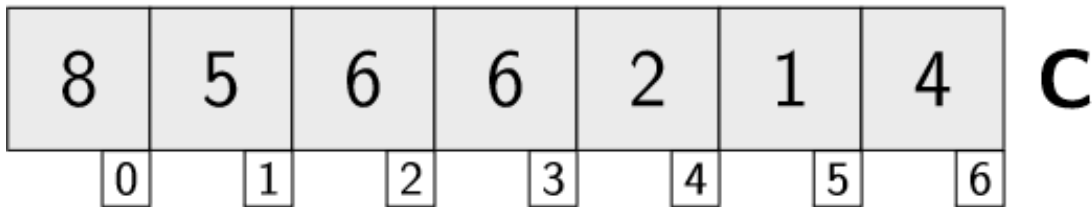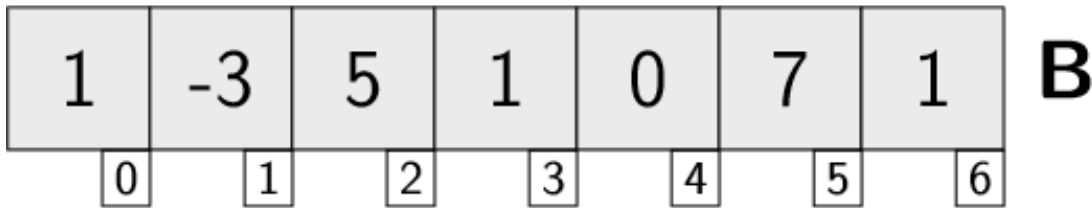
- Let's do A[i] = B[C[j]] as an example …

Reachability Analysis Example

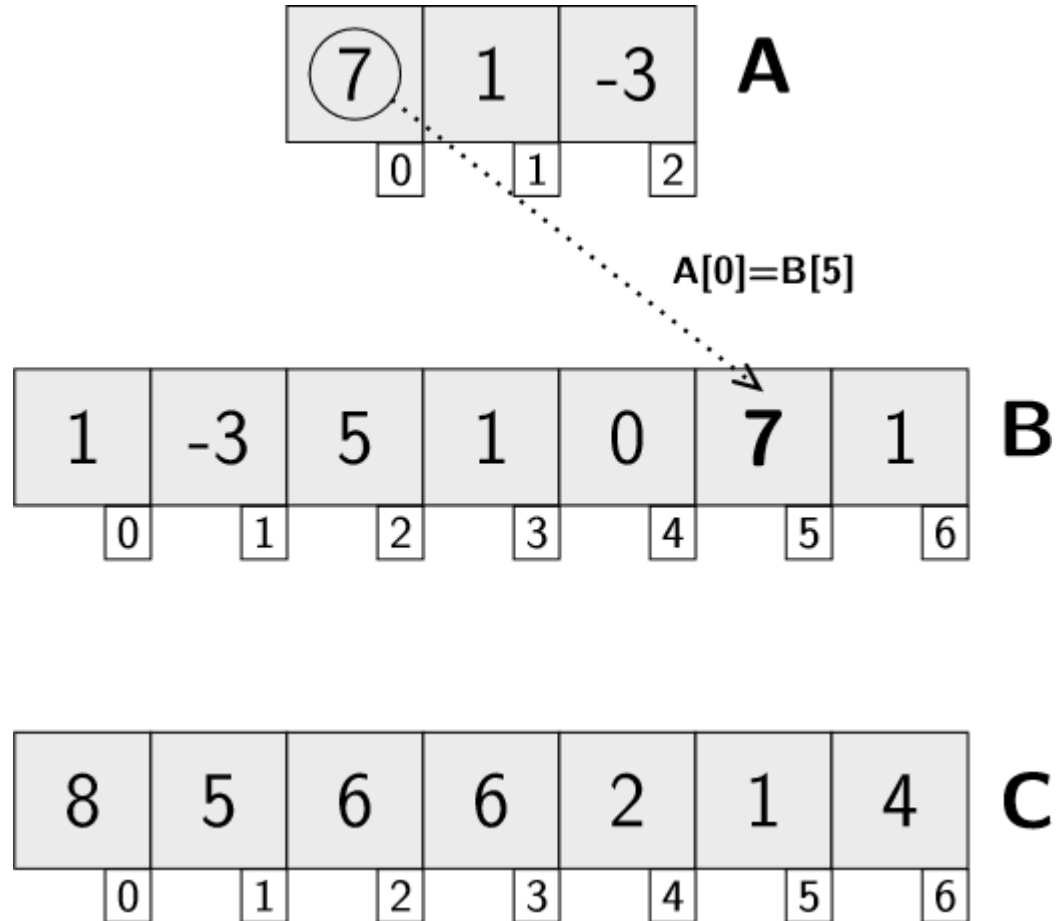| 7	1	-3	**A**
0	1	2	

$$A[0] \stackrel{?}{=} B[C[\ldots]]$$
$$A[1] \stackrel{?}{=} B[C[\ldots]]$$
$$A[2] \stackrel{?}{=} B[C[\ldots]]$$

1	-3	5	1	0	7	1	**B**
0	1	2	3	4	5	6	

8	5	6	6	2	1	4	**C**
0	1	2	3	4	5	6	
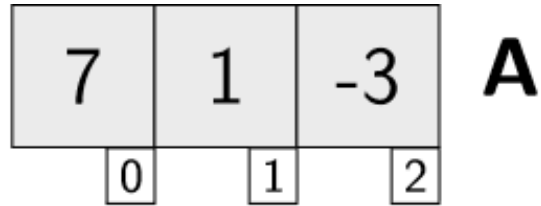
Reachability, A[0]=B[5]

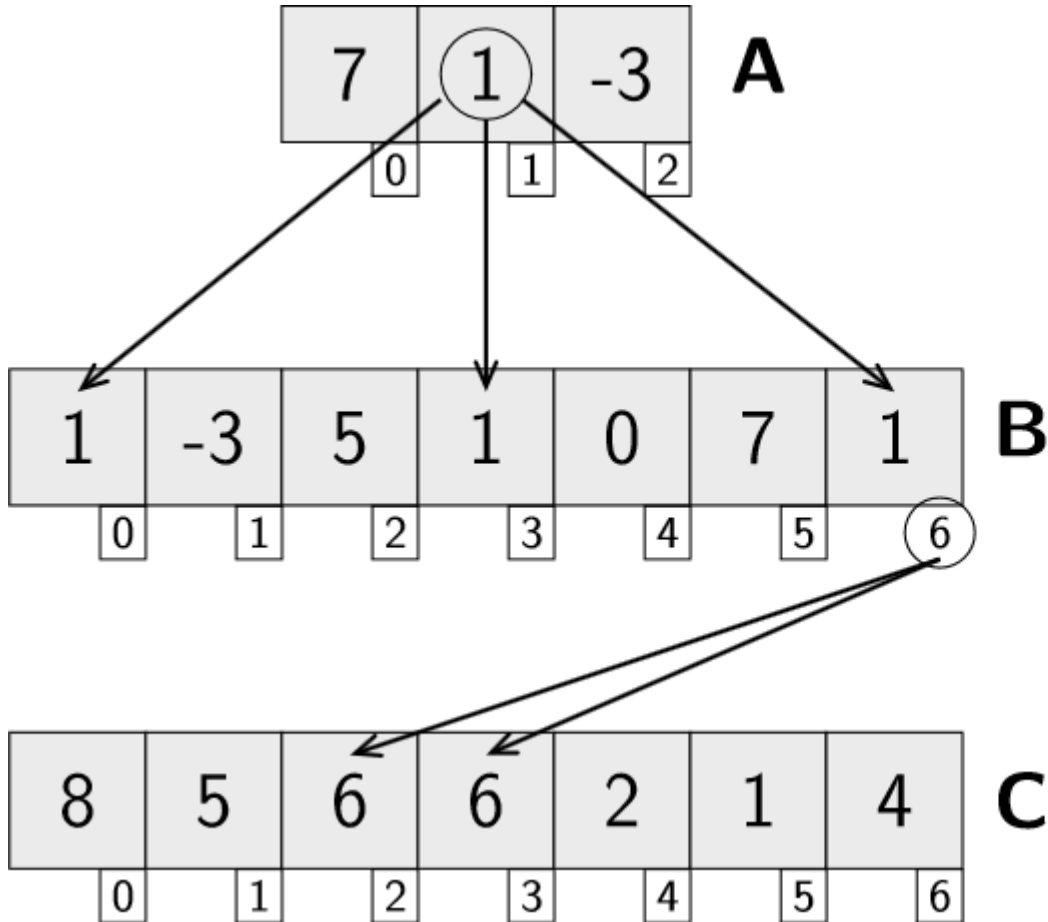# Reachability, A[0]=B[C[1]]

$$A[0] \quad = \quad B[C[1]]$$
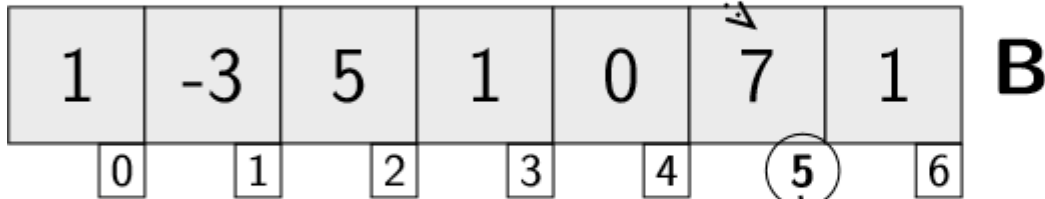
B[5]=B[C[1]]

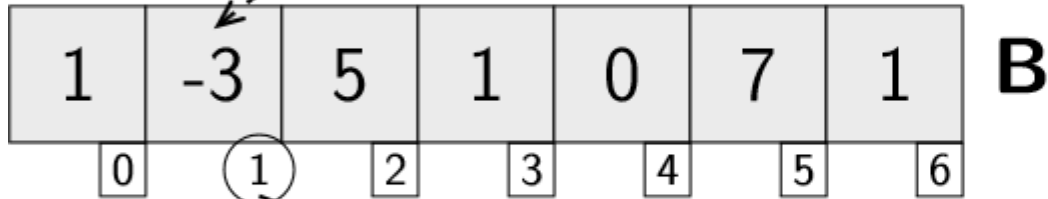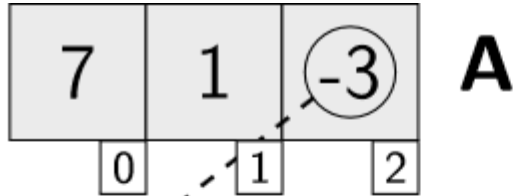# Reachability, A[1] = B[C[…]]
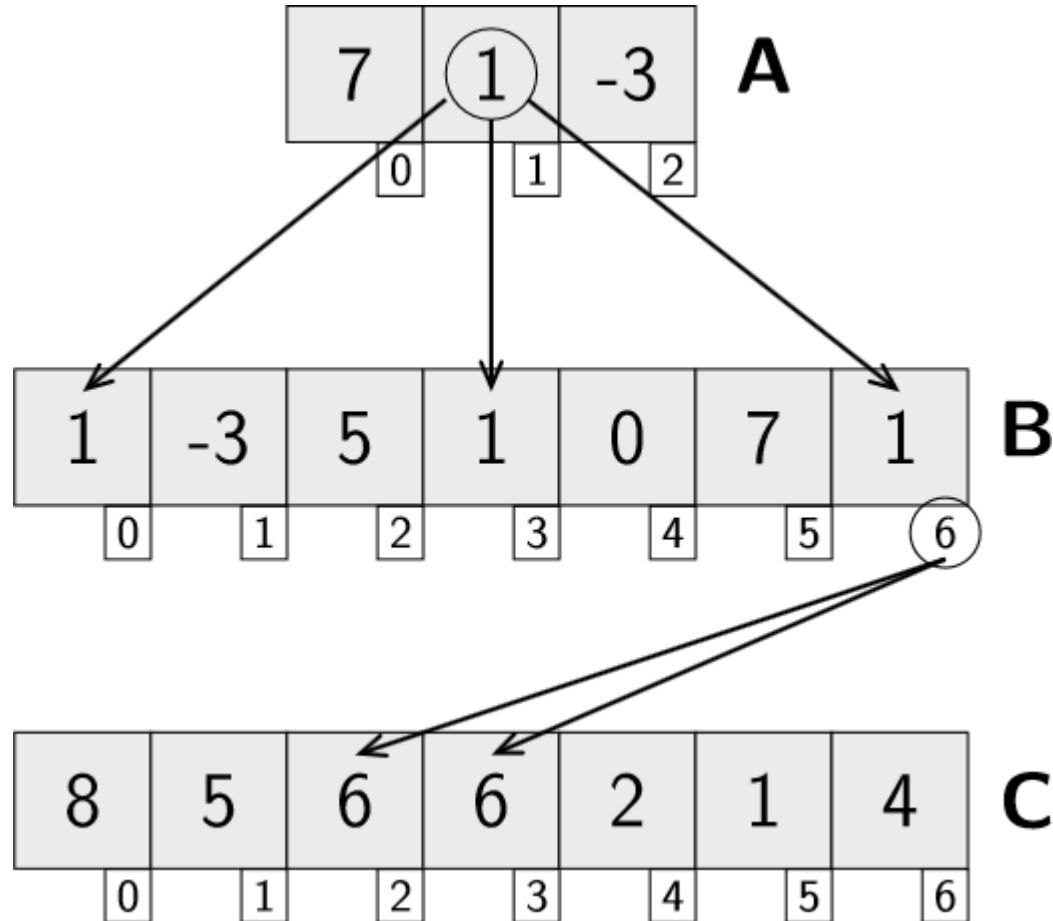
Reachability Example

$$A[0] = B[C[1]]$$
$$A[1] = B[C[2]] \lor B[C[3]]$$

Reachability, A[2] = B[C[5]]

$$A[0] = B[C[1]]$$
$$A[1] = B[C[2]] \vee B[C[3]]$$
$$A[2] = B[C[5]]$$

Equation Solving

$$A[0] \quad = \quad B[C[1]]$$
$$A[1] \quad = \quad B[C[2]] \lor B[C[3]]$$
$$A[2] \quad = \quad B[C[5]]$$

- Express relation A[i]=B[C[j]] via j = ip + q
- Handle all disjunctions
  - We'll do "left" first

Equation Solving

$$A[0] = B[C[1]]$$
$$A[1] = B[C[2]] \lor B[C[3]]$$
$$A[2] = B[C[5]]$$

- Express relation A[i]=B[C[j]] via j = ip + q

$$\{1 = 0p + q, 2 = 1p + q, 5 = 2p + q\}$$

- Solve for p, q:
  - No solution
- Backtrack, try "right" disjunction

Equation Solving

$$A[0] \quad = \quad B[C[1]]$$
$$A[1] \quad = \quad B[C[2]] \vee B[C[3]]$$
$$A[2] \quad = \quad B[C[5]]$$

- Express relation A[i]=B[C[j]] via j = ip + q

$$\{1 = 0p + q, 3 = 1p + q, 5 = 2p + q\}$$

- Solve for p, q:

$$\{q = 1, p = 2\} \quad \Rightarrow \quad j = 2i + 1$$
$$\Rightarrow \quad A[i] = B[C[2i + 1]]$$
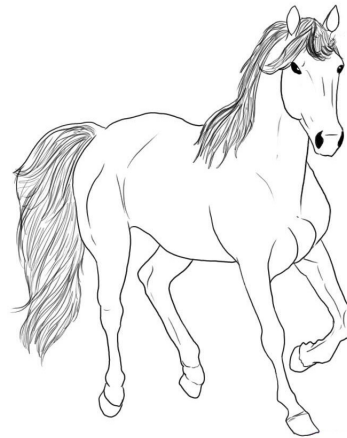
Ugh, Backtracking.

- Why should we have to do backtracking to deal with disjunctions?

- We have SAT and SMT solvers to do that!

- We'll just reduce solving the difficult problem of invariant generation to solving SAT

How to draw a horse

① Draw a unicorn. ② Delete the horn.

SMT Solving
(after reachability)

- Recall: A[i]=B[C[j]] via j = ip + q

$$A[0] = B[C[1]]$$
$$A[1] = B[C[2]] \vee B[C[3]]$$
$$A[2] = B[C[5]]$$
$$\Downarrow$$

$$(0p + q = 1) \wedge (1p + q = 2 \vee 1p + q = 3) \wedge (2p + q = 5)$$

- Just ask SMT to solve for p and q!

What about false positives?

- Given candidate invariants …
  - Just try to prove them using axiomatic semantics.
  - Recall: you can't be fooled by false invariants.

- See subsequent papers
  - "Using Dynamic Analysis to Generate Disjunctive Invariants", ICSE 2014

"When you have the golden claw, the solution is in the palm of your hands."

- Why cover invariant detection?
  - It is necessary for axiomatic semantics to be practical.

- Reinforces recurring theme:
  - The techniques you are learning are the backbone of modern PL research.

  - Simplex → Nonlinear Equalities, Inequalities
  - Model Checking, SMT Solving → Array Invariants
  - Invariants → Symbolic Execution

Homework

- HW4 Due
- Reading AI Papers for Monday