Learning to Specify

… soundly

Suresh Jagannathan

Joint work with He Zhu, Stephen Magill, and Gustavo Petri

PURDUE UNIVERSITY

Goal

Program + Specifications → Verification Conditions

Types
Assertions
Contracts
Pre/Post
Loop Invariants
. . .

Manual

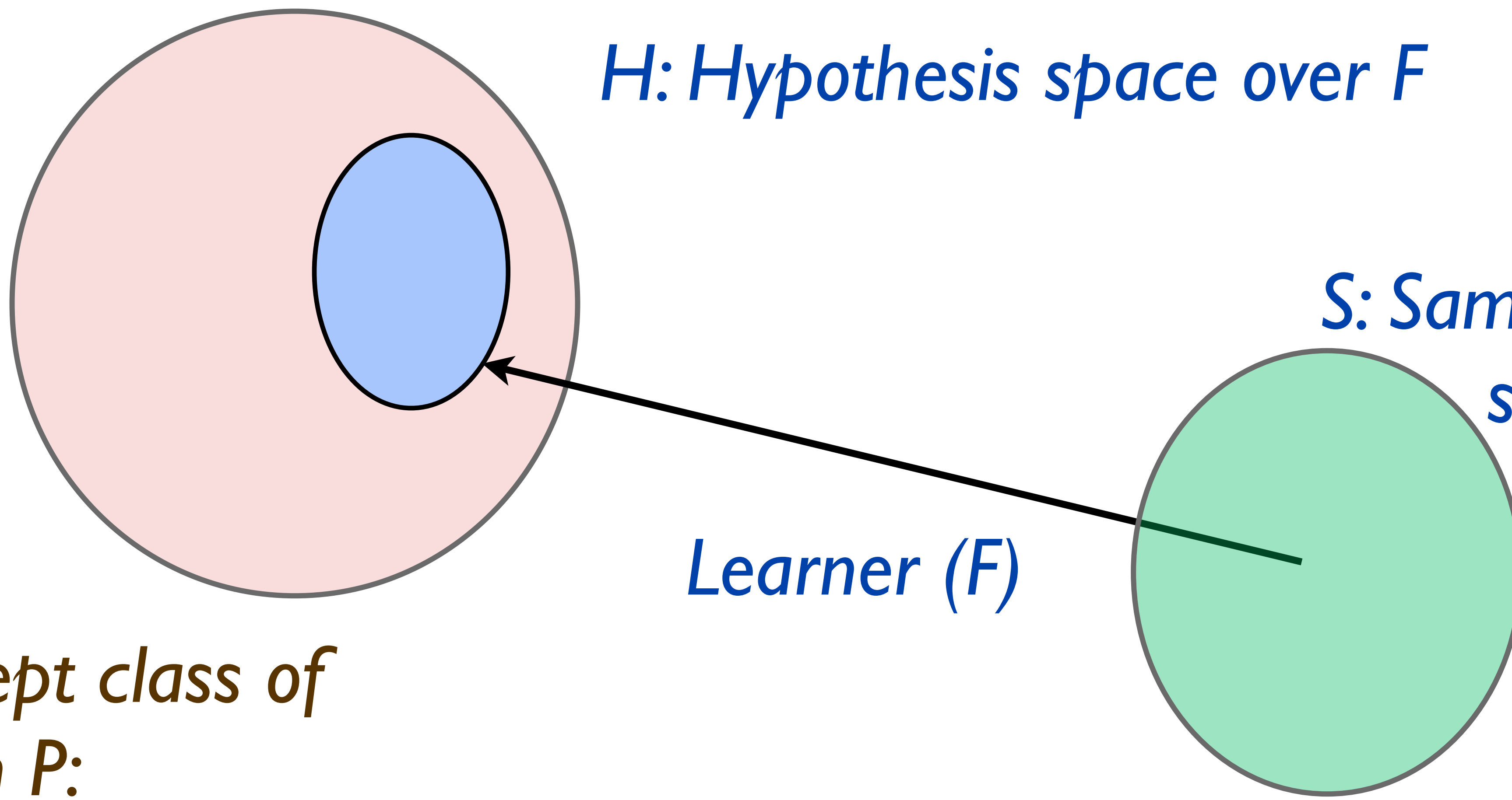
Automated

How do we automatically discover useful specifications to facilitate verification?

Learning …

Feature extraction: *P* ➔ *F*

Set of features: *F*



C: Concept class of program P:

Data structures,
Numeric domains,
…

Context and Challenges

- What is the language in which specifications are expressed?
  - ★ Decidability
- How do we generate samples?
  - ★ Coverage
- How do we generalize from samples?
  - ★ Turn postulated invariants to true invariants
  - ★ Soundness
- How do we infer inductive invariants?
  - ★ Necessary for automated verification
- How do we guarantee progress?
  - ★ Relate number of observations to quality of inference
- How do we ensure convergence?
  - ★ Will we eventually learn a true invariant?
- Quality of specifications (simplicity, minimality, ….)

A Programmer's Day ...

Defining data structures ...

```
type 'a list =
  | Nil
  | Cons 'a *
         'a list
```

```
type 'a tree =
  | Leaf
  | Node 'a *
         'a tree *
         'a tree
```

Writing functions ...

```
// flat: 'a list -> 'a tree -> 'a list
let rec flat accu t =
 match t with
   | Leaf -> accu
   | Node (x, l, r) ->
     flat (x::(flat accu r)) l

// elements: 'a tree -> 'a list
let elements t = flat [] t
```

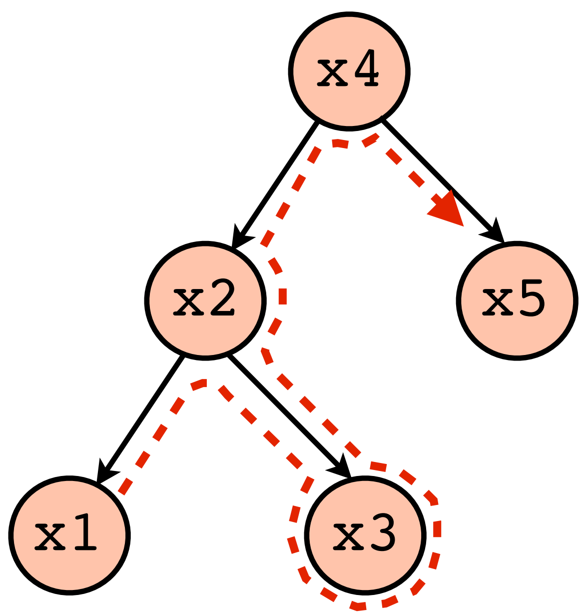
No assertions /
loop invariants
pre-conditions /
post-conditions!

A Programmer’s Day ...

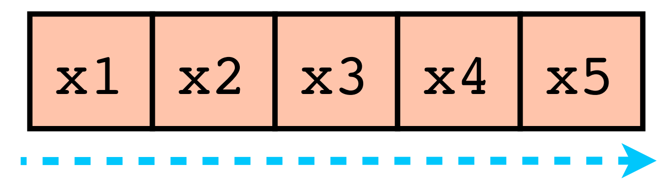
Testing code ...

```
// elements: ‘a tree -> ‘a list
let elements t = flat [] t
```

$$l = \text{elements } t$$



t



l

Implicitly discovers:

```
// specification:
// elements: ‘a tree -> ‘a list
// l = elements t
//                          ≡
// in-order(t)      forward-order(l)
```

Features of Data Structures ...

```
// elements: 'a tree -> 'a list
let elements t = flat [] t
```

$$l = \text{elements } t$$

Hypothesis Domain *over data structure features:*

$t \dashrightarrow u$ *Containment*

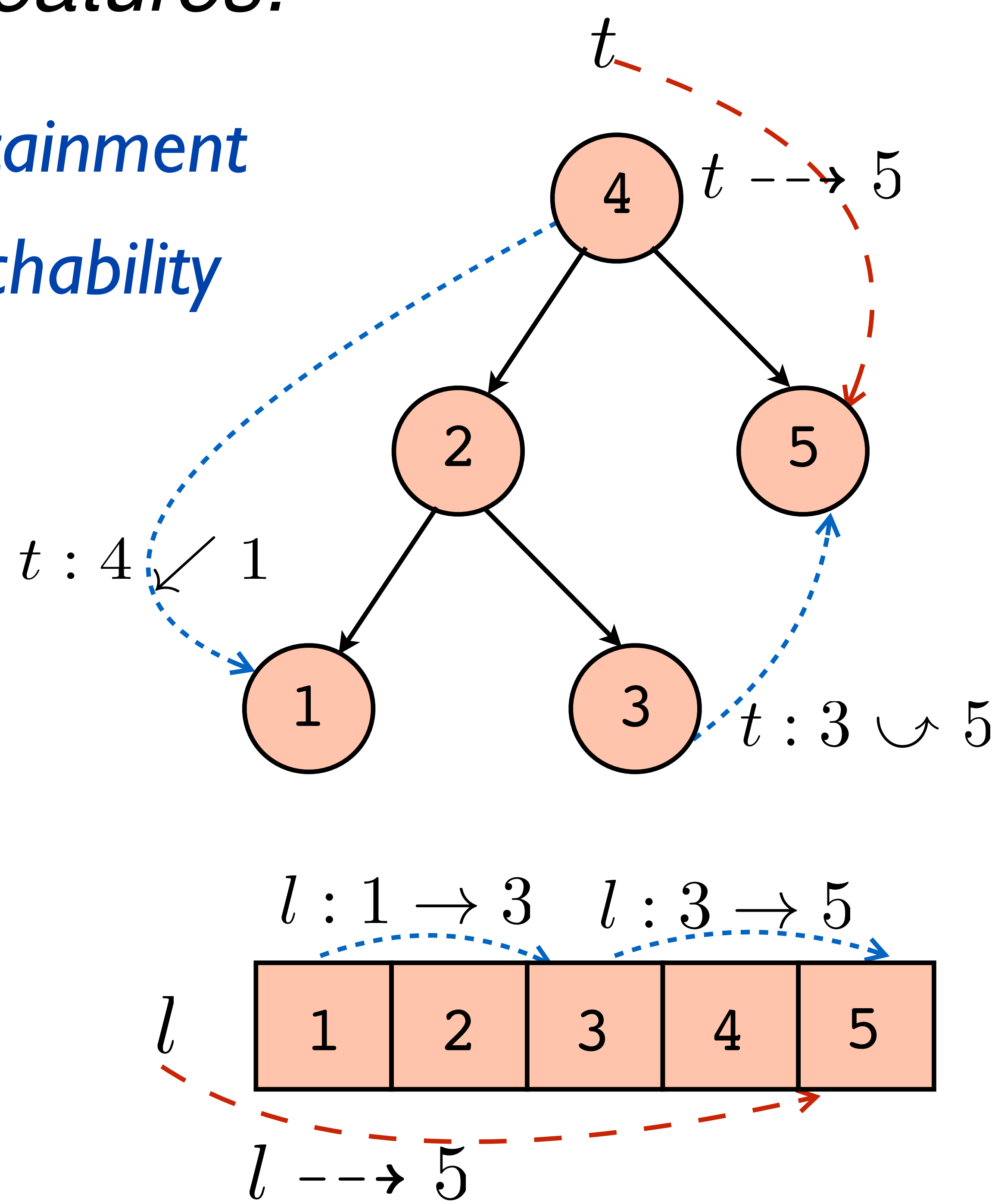
$t : u \searrow v$ *Reachability*

$t : u \swarrow v$

$t : u \curvearrowright v$

$l \dashrightarrow u$

$l : u \rightarrow v$

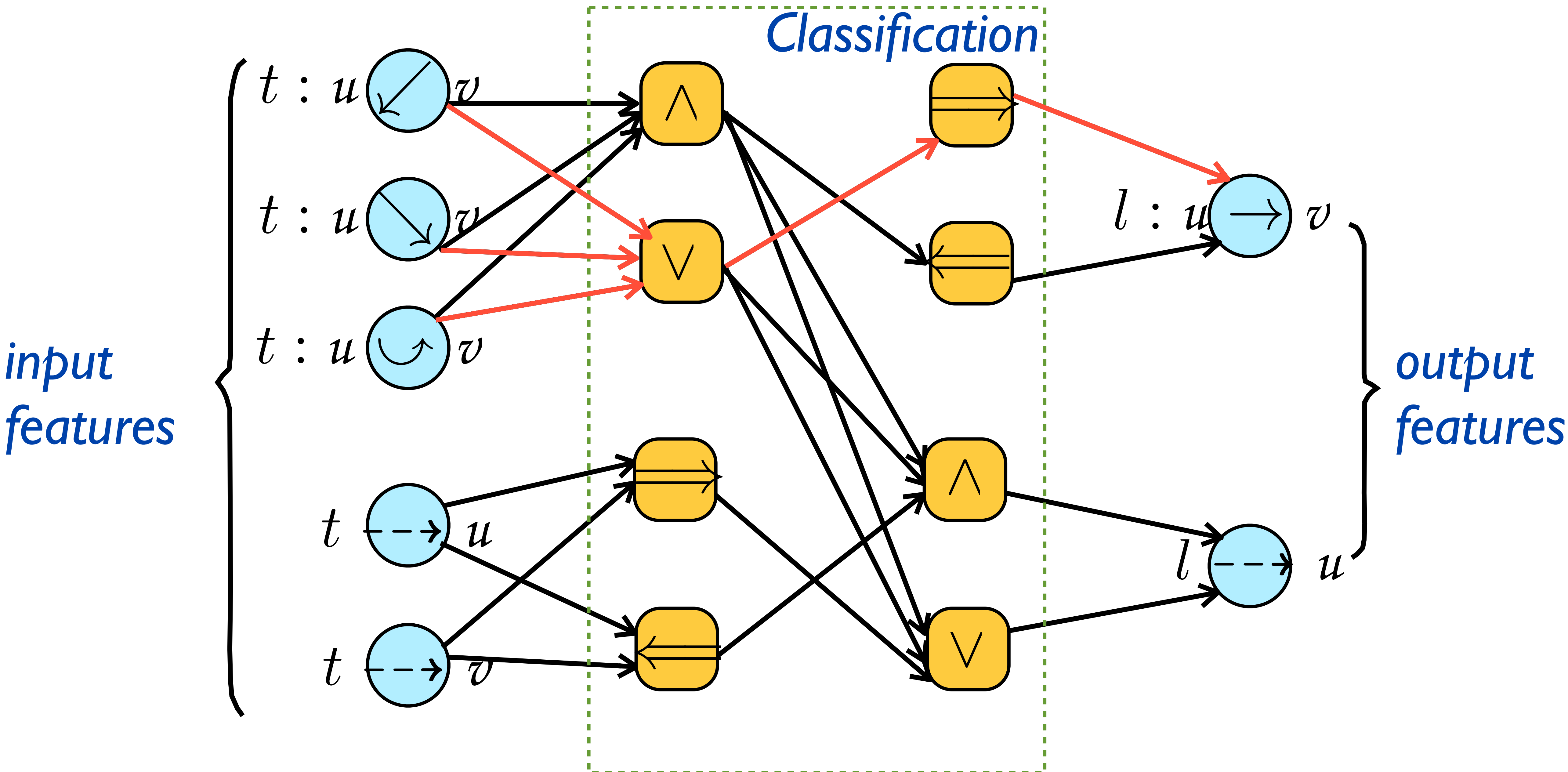


From features to specifications ...

```
// elements: 'a tree -> 'a list
let elements t = flat [] t
```

$$l = \text{elements } t$$

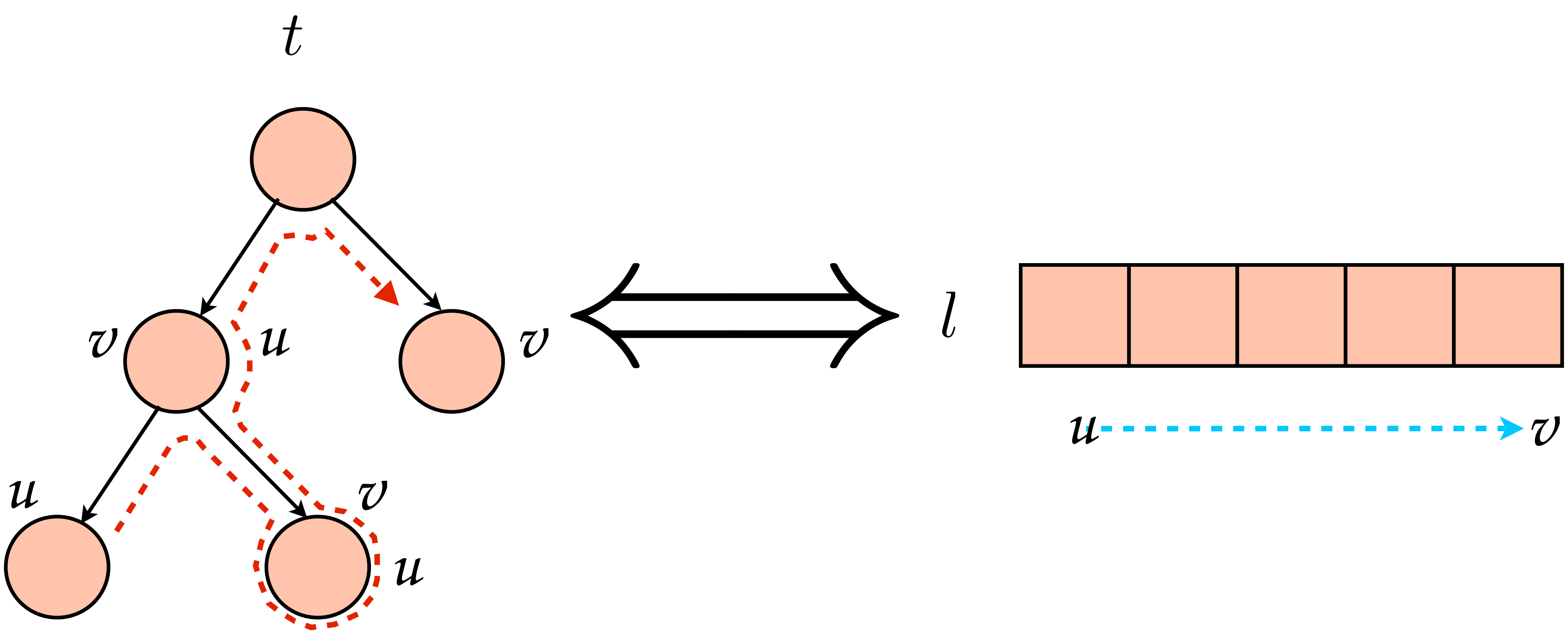
Predict truth of output features using a Boolean combination of input features ...



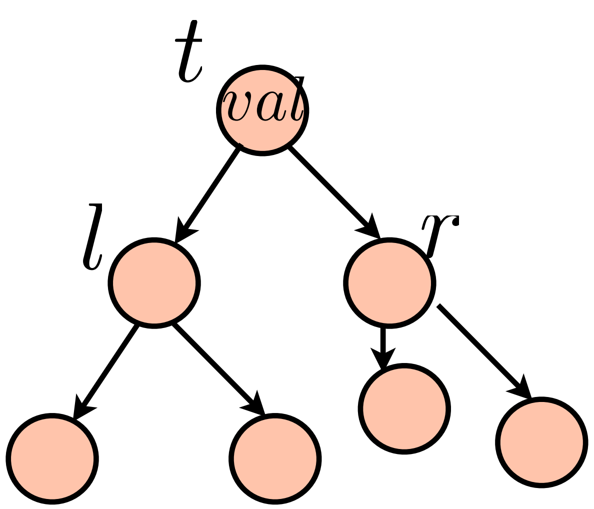
Specifications of Data Structures …

```
// specification:
// in-order of t ≡ forward-order of l
 l:list = elements (t:tree)
```

$$(\forall u\ v,\quad t : v \swarrow u \ \vee\ t : u \searrow v \ \vee\ t : u \circlearrowleft v \iff l : u \rightarrow v)$$



Feature Extraction ...

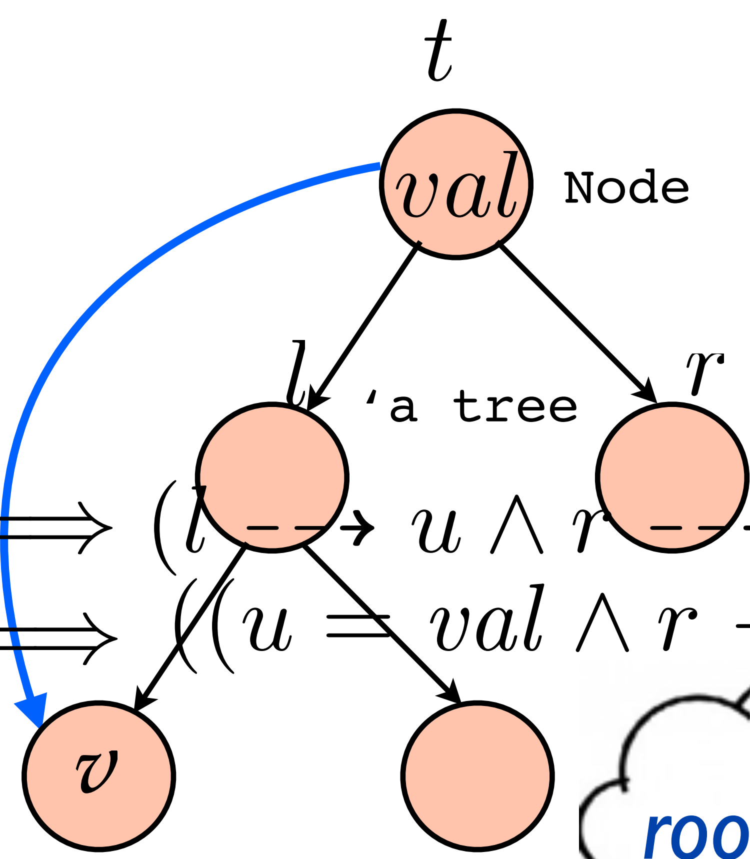



```
type 'a tree =
  | Leaf
  | Node 'a * 'a tree * 'a tree
```

$t : u \curvearrowright v$

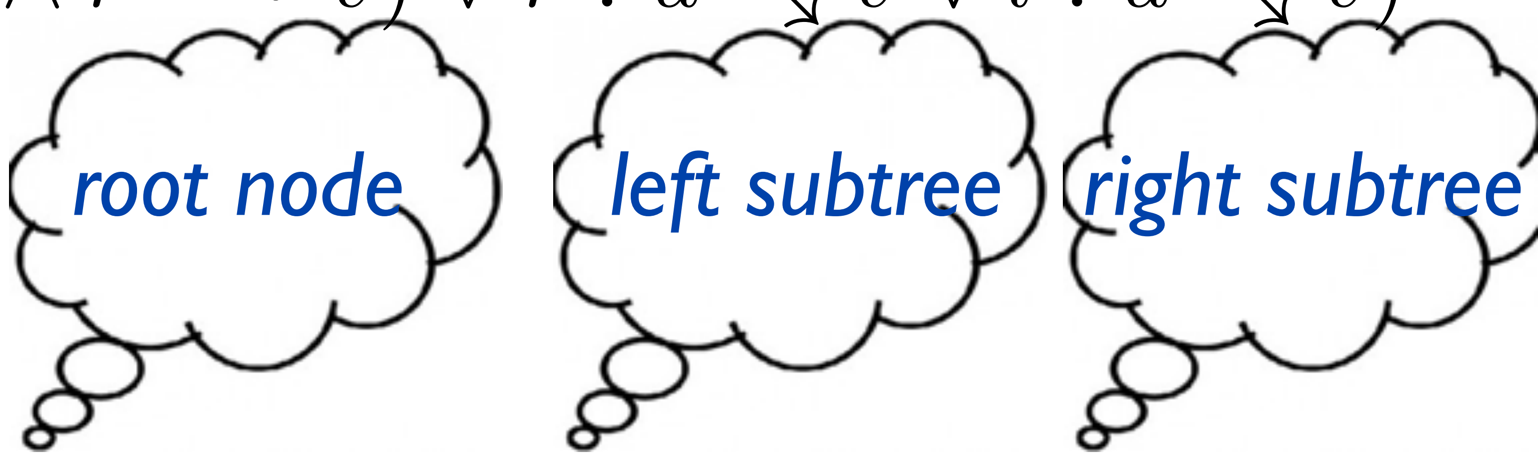
$t : u \swarrow v$

$t : u \searrow v$



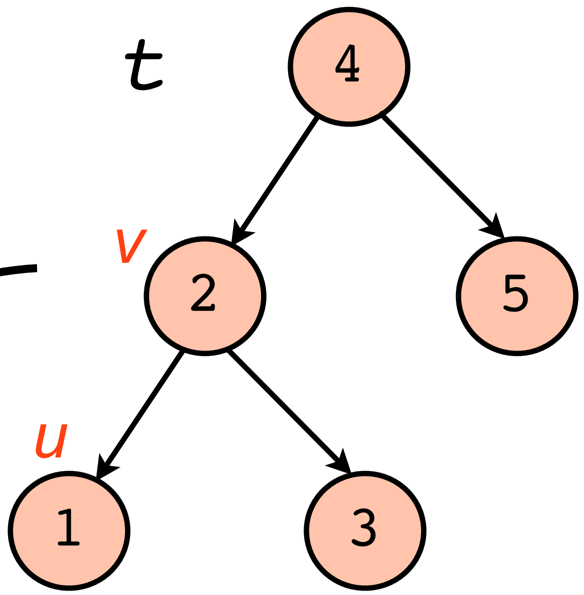
$$t : u \curvearrowright v \iff (l \dashrightarrow u \wedge r \dashrightarrow v) \vee l : u \curvearrowright v \vee r : u \curvearrowright v$$

$$t : u \searrow v \iff ((u = val \wedge r \dashrightarrow v) \vee r : u \searrow v \vee l : u \searrow v)$$



$$t : u \swarrow v \iff ((u = val \wedge l \dashrightarrow v) \vee l : u \swarrow v \vee r : u \swarrow v$$

Learner ...




```
// elements: 'a tree -> 'a list
let elements t = flat [] t
```

$$l = \text{elements } t$$

l

1	2	3	4	5

(u, v)

$\overbrace{\hspace{10em}}^{\textit{input features}}$ $\overbrace{\hspace{3em}}^{\textit{output features}}$

(u,v)	$t : u \swarrow v$	$t : u \searrow v$	$t : u \circlearrowleft v$	$t : v \swarrow u$	$t : v \searrow u$	$t : v \circlearrowleft u$	$t \dashrightarrow u$	$t \dashrightarrow v$		$l : u \rightarrow v$
(1,2)	0	0	0	1	0	0	1	1		1
(4,5)	0	1	0	0	0	0	1	1	pos	1
(2,5)	0	0	1	0	0	0	1	1		1
(3,1)	0	0	0	0	0	1	1	1		0
(3,2)	0	0	0	0	1	0	1	1	neg	0
(4,1)	1	0	0	0	0	0	1	1		0

⋮

Sample space

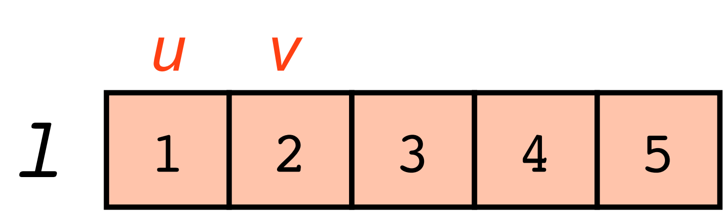
Learner ...

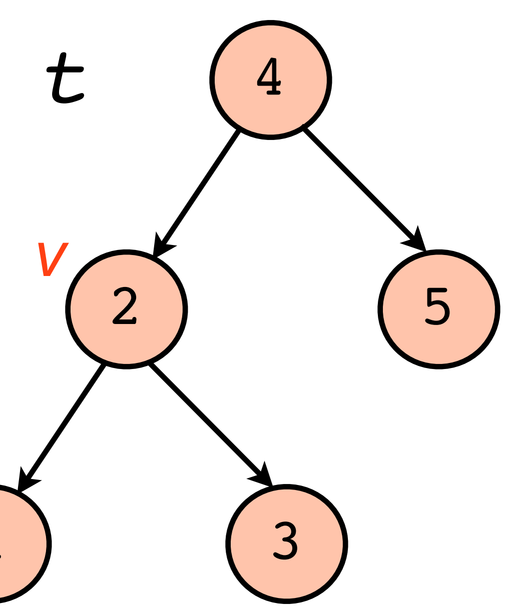
input features

	(u,v)	$t : u \swarrow v$	$t : u \searrow v$	$t : u \circlearrowleft v$	$t : v \swarrow u$	$t : v \searrow u$	$t : v \circlearrowleft u$	$t \dashrightarrow u$	$t \dashrightarrow v$	
pos samples	(1,2)	0	0	0	1	0	0	1	1	
$l : u \to v$	(4,5)	0	1	0	0	0	0	1	1	φ
	(2,5)	0	0	1	0	0	0	1	1	
neg samples	(3,1)	0	0	0	0	0	1	1	1	
$\neg\, l : u \to v$	(3,2)	0	0	0	0	1	0	1	1	$\neg\varphi$
	(4,1)	1	0	0	0	0	0	1	1	

SPECS

$$\varphi \iff l : u \to v$$





Learner ...

Truth Table

(u, v)	$t : u \swarrow v$	$t : u \searrow v$	$t : u \circlearrowleft v$	$t : v \swarrow u$	$t : v \searrow u$	$t : v \circlearrowleft u$	$t \dashrightarrow u$	$t \dashrightarrow v$		$l : u \rightarrow v$
(1,2)	0	0	0	1	0	0	1	1	pos	1
(4,5)	0	1	0	0	0	0	1	1	pos	1
(2,5)	0	0	1	0	0	0	1	1	pos	1
(3,1)	0	0	0	0	0	1	1	1	neg	0
(3,2)	0	0	0	0	1	0	1	1	neg	0
(4,1)	1	0	0	0	0	0	1	1	neg	0

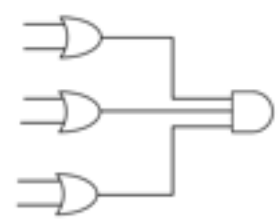
- Optimization task:
  - Constraint solvers

Learner ...

```
l:list = elements (t:tree)
```

Truth Table

$t : u \searrow v$	$t : u \circlearrowright v$	$t : v \swarrow u$	$l : u \to v$
0	0	1	1
1	0	0	1
0	1	0	1
0	0	0	0
0	0	0	0
0	0	0	0



$$(\forall u\ v,\ \left(\begin{array}{l} t : v \swarrow u \lor \\ t : u \circlearrowright v \lor \\ t : u \searrow v \end{array}\right) \iff l : u \to v)$$

Learner ...

$$\Longleftrightarrow$$

If and only if specifications are nice, but …

		input feature1	input feature2	input feature3	input feature4	input feature5	input feature6	input feature7	input feature8
pos samples	output feature	0	0	0	1	0	0	1	1
		0	1	0	0	0	0	1	1
		0	0	1	0	0	0	1	1
neg samples	¬output feature	0	0	0	1	0	0	1	1
		0	0	0	0	1	0	1	1
		1	0	0	0	0	0	1	1

No classifier!

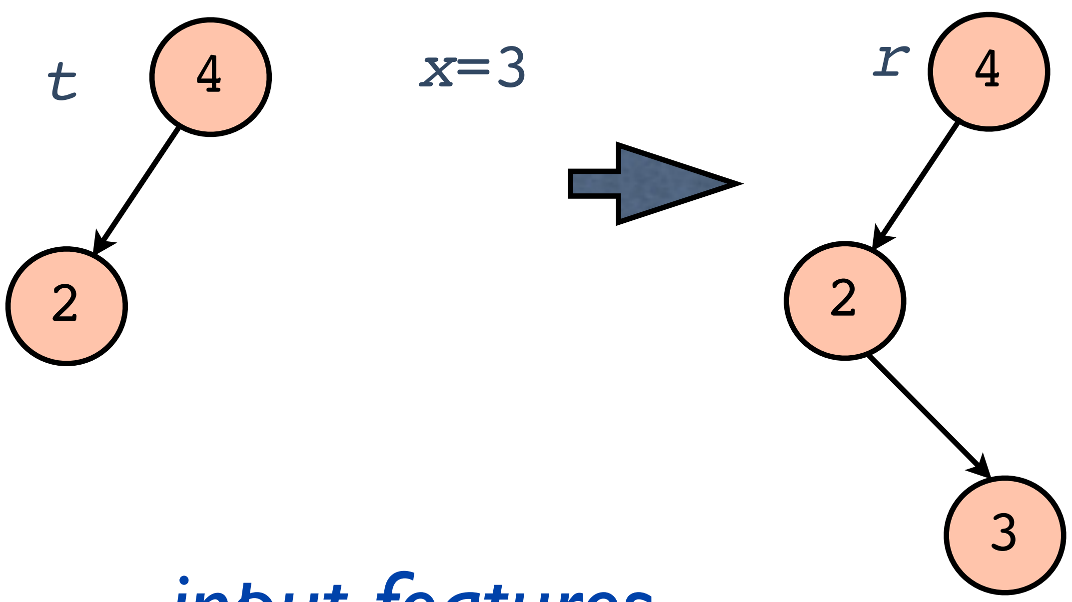
?

$$\Longleftarrow \quad \Longrightarrow$$

Binary Search Tree Insertion ...

```
let rec insert x t =
 match t with
  | Leaf -> Node (x, Leaf, Leaf)
  | Node (y, l, r) ->
     if x < y then
         Node (y, insert x l, r)
      else if y < x then
         Node (y, l, insert x r)
      else t
```

$r = \text{insert } 3\ t$



$\underbrace{\hspace{6cm}}_{}$ *input features* — *output features*

input features		output features
$\Pi_0\ t : u \swarrow v$	$\Pi_8\ u = x$	Π_{10}
$\Pi_1\ t : u \searrow v$	$\Pi_9\ v = x$	$r : u \swarrow v$
$\Pi_2\ t : u \circlearrowright v$		$\vdots$
$\Pi_3\ t : v \swarrow u$		
$\Pi_4\ t : v \searrow u$		
$\Pi_5\ t : v \circlearrowright u$		
$\Pi_6\ t \dashrightarrow u$		
$\Pi_7\ t \dashrightarrow v$		

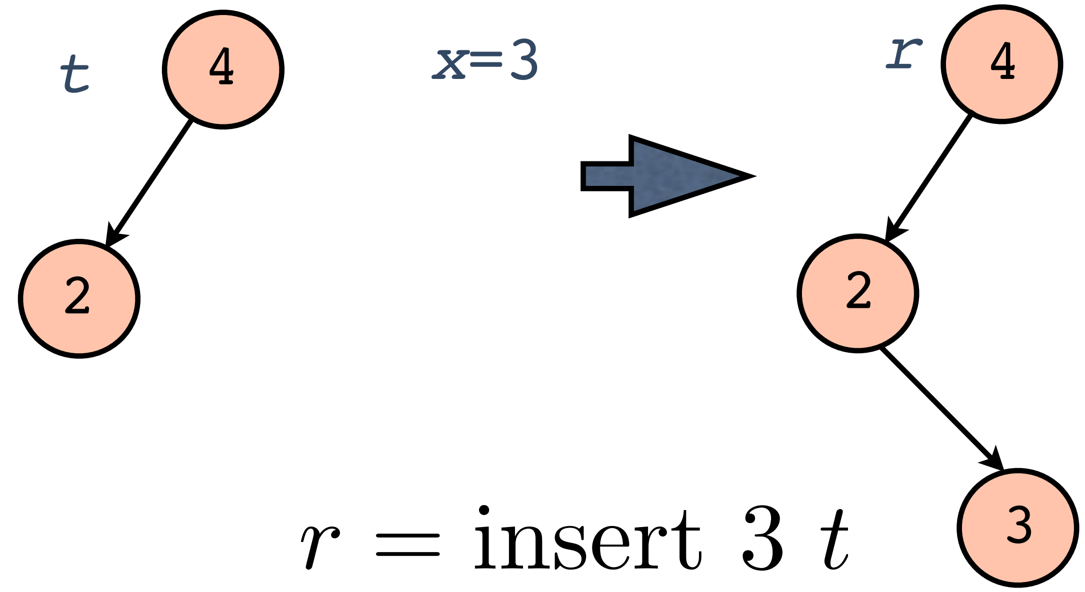
input features

(u, v)	Π_0	Π_1	Π_2	Π_3	Π_4	Π_5	Π_6	Π_7	Π_8	Π_9		Π_{10}
(4,3)	0	0	0	0	0	0	1	0	0	1	pos	1
(2,3)	0	0	0	0	0	0	1	0	0	1	neg	0

Problem:

Samples are not separable with existing features

Binary Search Tree Insertion ...



$$r = \text{insert } 3\ t$$

input features $\underbrace{\qquad}$ *output features*

input features		output features
$\Pi_0\ t : u \swarrow v$	$\Pi_8\ u = x$	Π_{10}
$\Pi_1\ t : u \searrow v$	$\Pi_9\ v = x$	$r : u \swarrow v$
$\Pi_2\ t : u \circlearrowleft v$		$\vdots$
$\Pi_3\ t : v \swarrow u$		
$\Pi_4\ t : v \searrow u$		
$\Pi_5\ t : v \circlearrowleft u$		
$\Pi_6\ t \dashrightarrow u$		
$\Pi_7\ t \dashrightarrow v$		

input features

	Π_0	Π_1	Π_2	Π_3	Π_4	Π_5	Π_6	Π_7	Π_8	Π_9	Π_{10}
~~(4,3)~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~1~~	~~0~~	~~0~~	~~1~~	~~1~~
(4,2)	1	0	0	0	0	0	1	1	0	0	1
~~(2,3)~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~1~~	~~0~~	~~0~~	~~1~~	~~0~~
(2,4)	0	0	1	0	0	0	1	1	0	0	0

$$\forall u\ v,\ t : u \swarrow v \Rightarrow r : u \swarrow v$$

$$\forall u\ v,\ r : u \swarrow v \Rightarrow \left(\begin{array}{c} (t \dashrightarrow u \wedge v = x) \vee \\ t : u \swarrow v \end{array} \right)$$

Verification

Encode candidate specifications as refinements in a refinement type system (LiquidTypes)

$$
\begin{array}{rcl}
\texttt{spec}(B,\ \psi) & = & \{\nu : B \mid \psi\} \\
\texttt{spec}(D,\ \psi) & = & \{\nu : D \mid \psi\} \\
\texttt{spec}(\{x : \tau_1 \to \tau_2\},\ \psi) & = & \{x : \tau_1 \to \texttt{spec}(\tau_2, \psi)\} \\
\texttt{specType}(\Gamma_f, f, \psi) & = & \texttt{spec}(\texttt{HM}(\Gamma_f, f), \psi)
\end{array}
$$

$$
\Gamma_f \vdash \texttt{fix}\,(\,\texttt{fun}\ f \to \lambda x.\, e) : \texttt{specType}(\Gamma_f, f, \psi)
$$

Unfold predicate definitions based on context

LIST MATCH

$$
\frac{\Gamma \vdash v : \texttt{'a list} \qquad \left[\ \Gamma;\ (\forall u\ v,\ v : u \to v \iff \texttt{false} \wedge \forall u,\ v \dashrightarrow u \iff \texttt{false})\ \right] \vdash e_1 : P \qquad \left[\begin{array}{l} \Gamma;\ x : \texttt{'a};\ xs : \texttt{'a list};\ (\forall u,\ v \dashrightarrow u \iff (u = x \vee xs \dashrightarrow u) \\ \quad \wedge\ \forall u\ v,\ v : u \to v \iff ((u = x \wedge xs \dashrightarrow v) \vee xs : u \to v)) \end{array}\right] \vdash e_2 : P}{\Gamma \vdash (\ \texttt{match}\ v\ \texttt{with} \mid \texttt{Nil} \to e_1 \mid \texttt{Cons}\,(x, xs) \to e_2\ ) : P}
$$

FUNCTION

$$
\frac{\Gamma;\ f : \{x : P_x \to P\};\ x : P_x \vdash e : P_e \qquad \Gamma;\ x : P_x \vdash P_e <: P}{\Gamma \vdash \texttt{fix}\,(\texttt{fun}\ f \to \lambda x.\, e) : \{x : P_x \to P\}}
$$

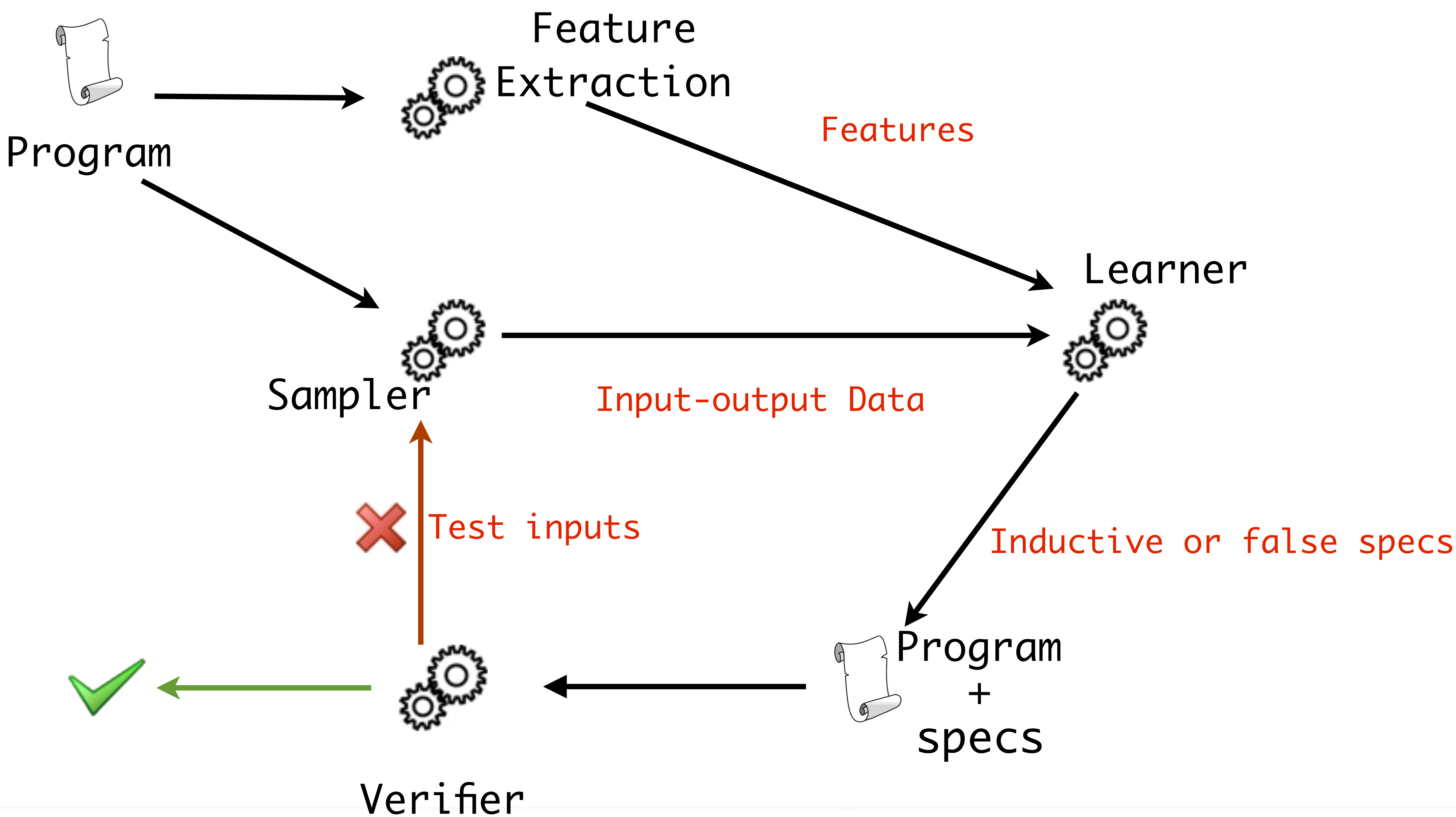
SUBTYPE DTYPE

$$
\frac{\texttt{Valid}(\langle\Gamma\rangle \wedge \langle\psi_1\rangle \Rightarrow \langle\psi_2\rangle)}{\Gamma \vdash \{D \mid \psi_1\} <: \{D \mid \psi_2\}}
$$

Propagate type constraints from function's pre-condition to its post-condition

Encoding yields (decidable) EPR formulae; completeness is ensured by axiomatizing transitive closure for supported data types

Verification and Convergence ...



Theorem: The learning algorithm eventually converges to the strongest inductive specification in the hypothesis space.

Experimental Results …

- DOrder -- implemented within the OCaml tool chain.
- Programmers write code as usual (with no annotation burden) while the tool reports program specifications.
- Fast verification (< 2 minutes), small # samples (~ 20 samples avg.)

Benchmark Programs	Specifications
● Okasaki's funcional Stack, Queue ● Lists: mem, concat, reverse, filter, insertionsort, quicksort, mergesort ● Set: list-based and tree-based implementations ● Heap: Leftist, Skew, Splay, Pairing, Binomial, Heapsort ● Tree: Treap, AVL, Braun, Splay, Redblack, Random-access-list, Proposition-lib and OCaml-Set-lib	● List reversal: input-forward is output-backward ● Balanced tree insertion preserves in-order relation ● Heap removal preserves parent-children relations of extant nodes ● Shape-data: Sorting, BST, Heap-ordered ● Numeric: Tree balance

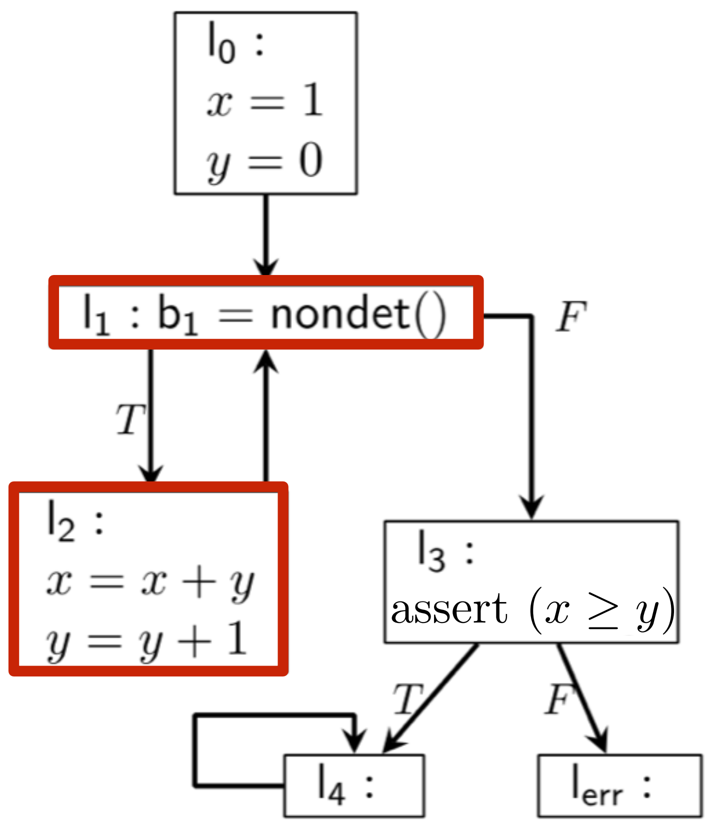
Loop (Numeric) Invariants

Program

```
main() {
  int x = 1;
  int y = 0;
  while (*) {
    x = x + y;
    y = y + 1;
  }
  assert (x >= y)
}
```

$p(x,y)$

CFG



VCs

Induction

$$x = 1 \wedge y = 0 \to p(x,y)$$

$$\boxed{p(x,y) \wedge x' = x + y \wedge y' = y + 1 \to p(x',y')}$$

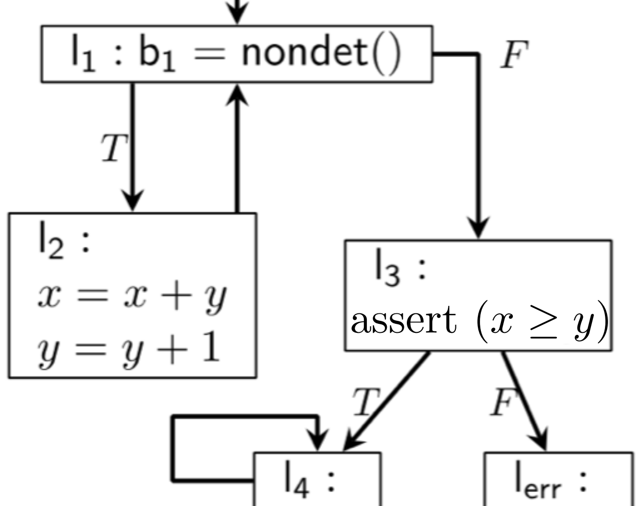
$$p(x,y) \wedge x' = x + y \wedge y' = y + 1 \to x' >= y'$$

$$x = 1 \wedge y = 0 \to x >= y$$

Spacer fails in this particular case

Data-Driven Invariant Inference

Sampling p(x, y)



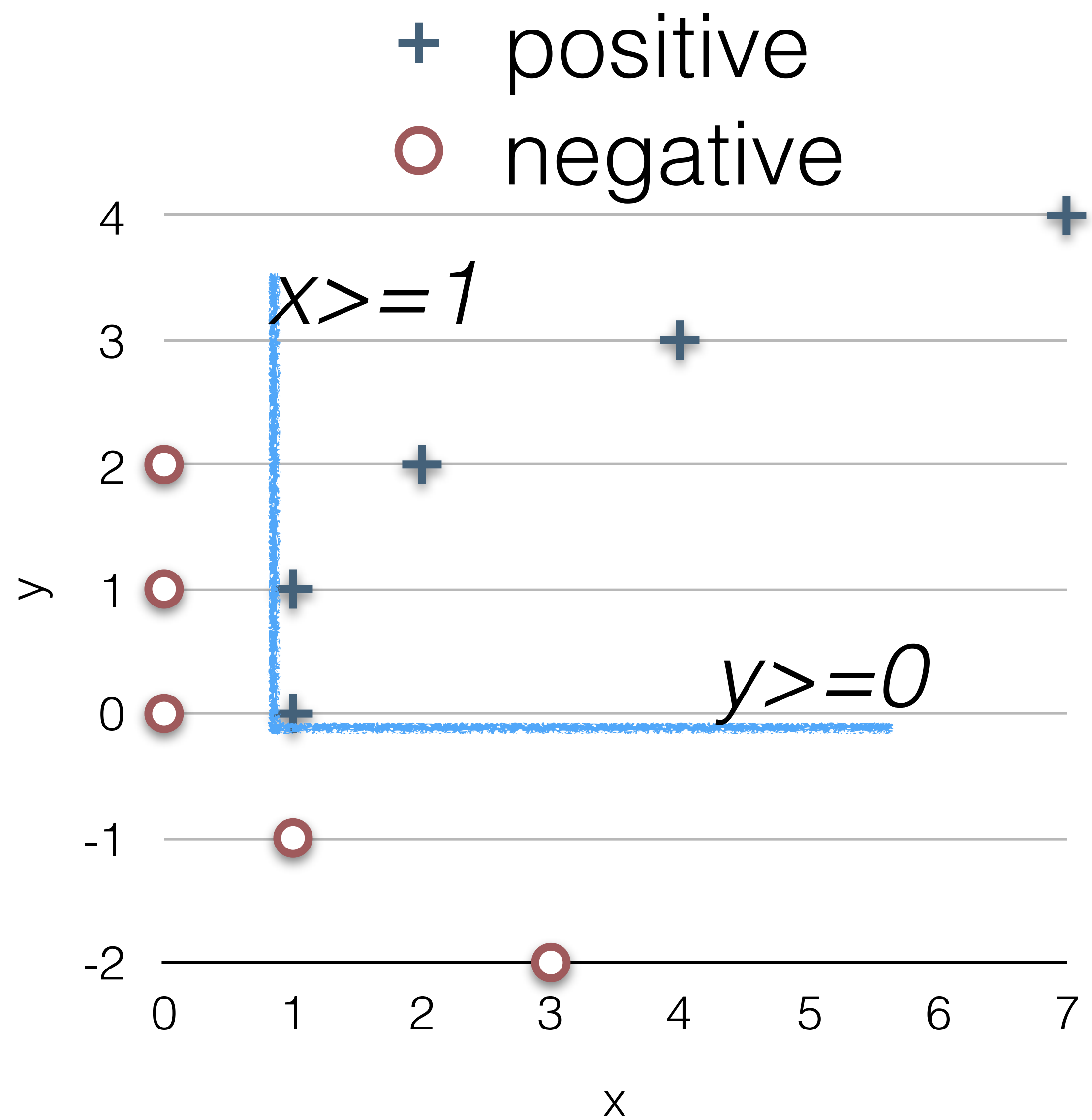
p(1,0), p(1,1), …

p(0,1) p(0,2), …

Ask Z3 *positive* *negative*

classification

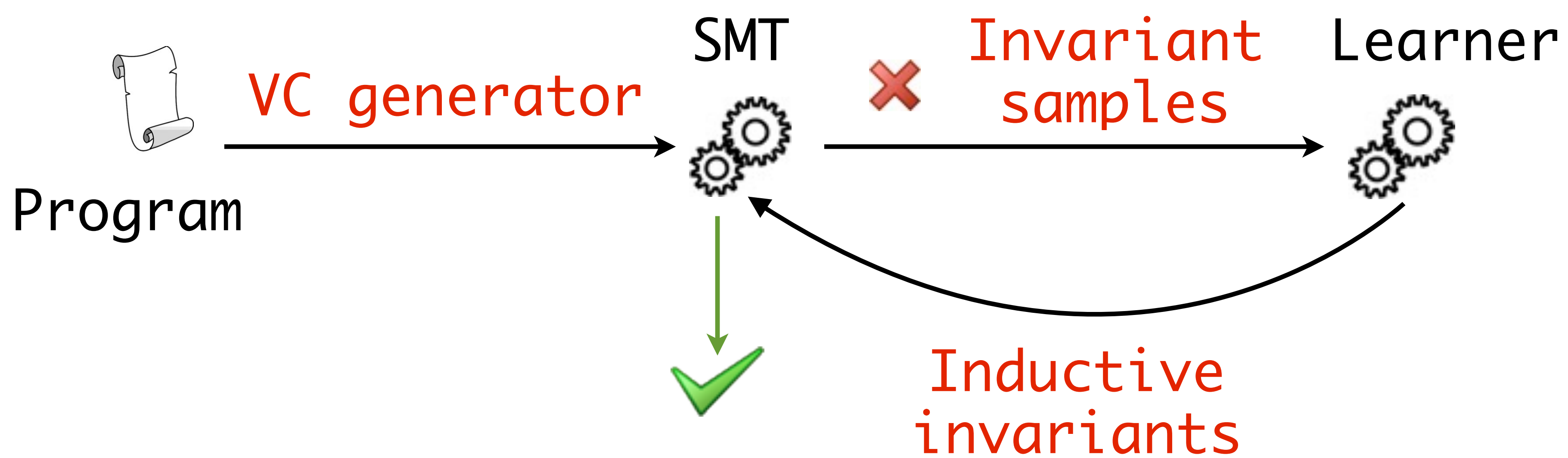
$$p(x, y) \equiv \{x >= 1 \wedge y >= 0\}$$



Data-Driven Invariant Inference for Recursive CHC systems

Vision:
An inductive invariant can be discovered from data

SynthHorn work flow:



Goal: Design a learner to learn *inductive* invariants from data

Hypothesis Domain

A *Machine Learning* Technique for invariants of *arbitrary Boolean* combination of *arbitrary linear arithmetic* *predicates.*

$$\bigvee_i \bigwedge_j \mathbf{w}_{ij}^T \cdot \mathbf{x}_{ij} + b_{ij}$$

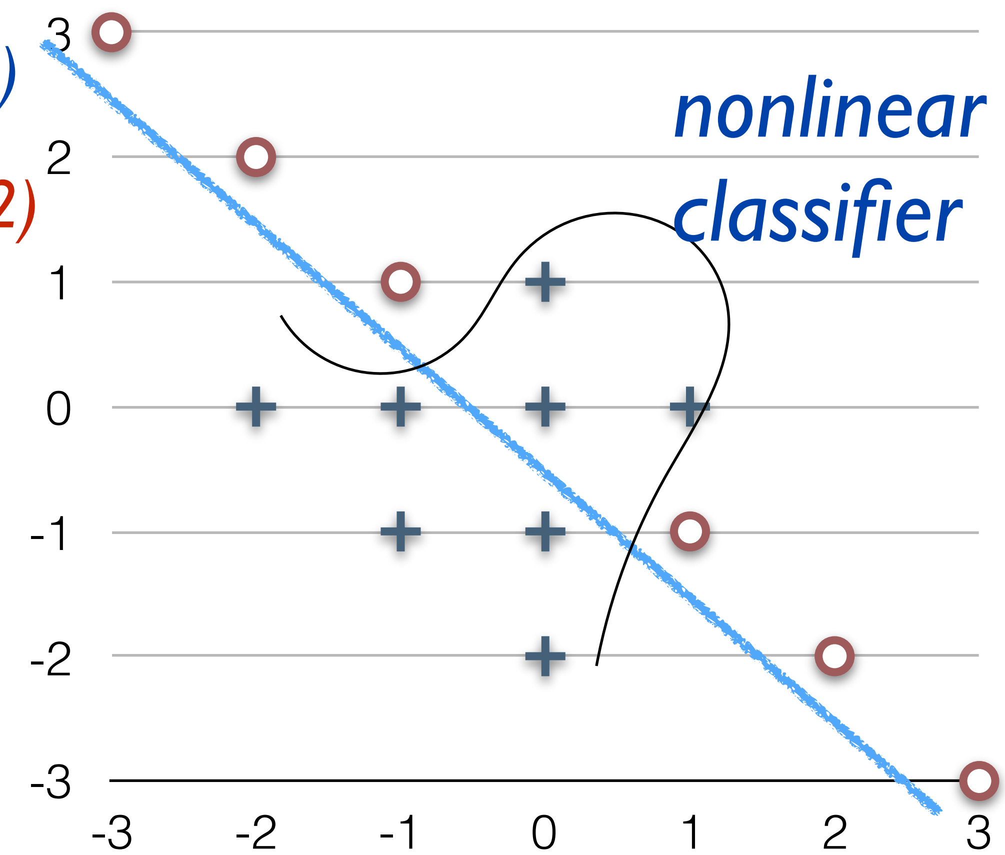
Linear Classification

Sampling p(x, y)

```
main() {
  int x,  y;
  x = 0;  y = *;
  while (y != 0) { // p(x,y)
    if (y < 0)  {x--;  y++;}
    else  {x++;  y—;}
    assert  (x != 0);
  }
}
```

p(1,0) p(1,1) p(2, 2) p(4,3) p(7,4)

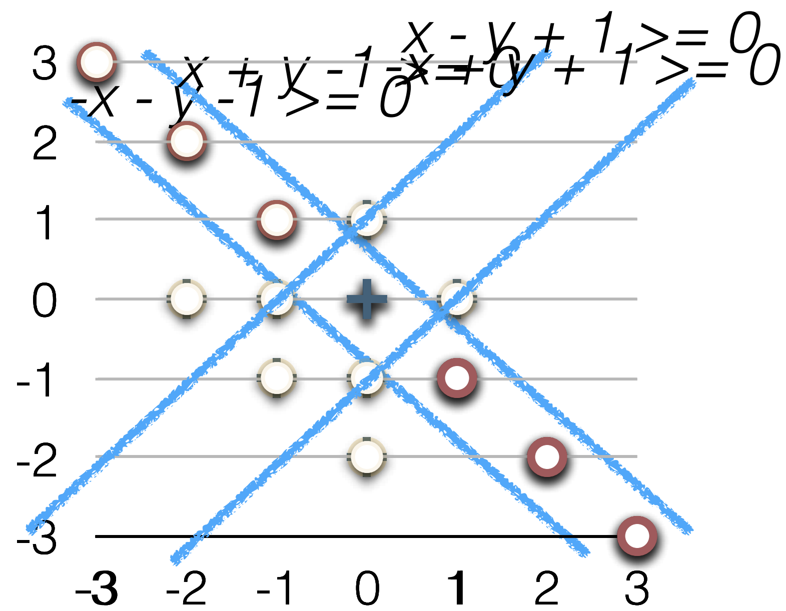
p(3,-2) p(1,-1) p(0,0) p(0,1) p(0,2)



- First take: use linear classification (SVM, Perceptron, Logistic Regression).
- But, there is a tension between Machine Learning and Verification: Generality vs. Safety.

Learning Arbitrarily Shaped Numeric Invariants ...

Given the data,



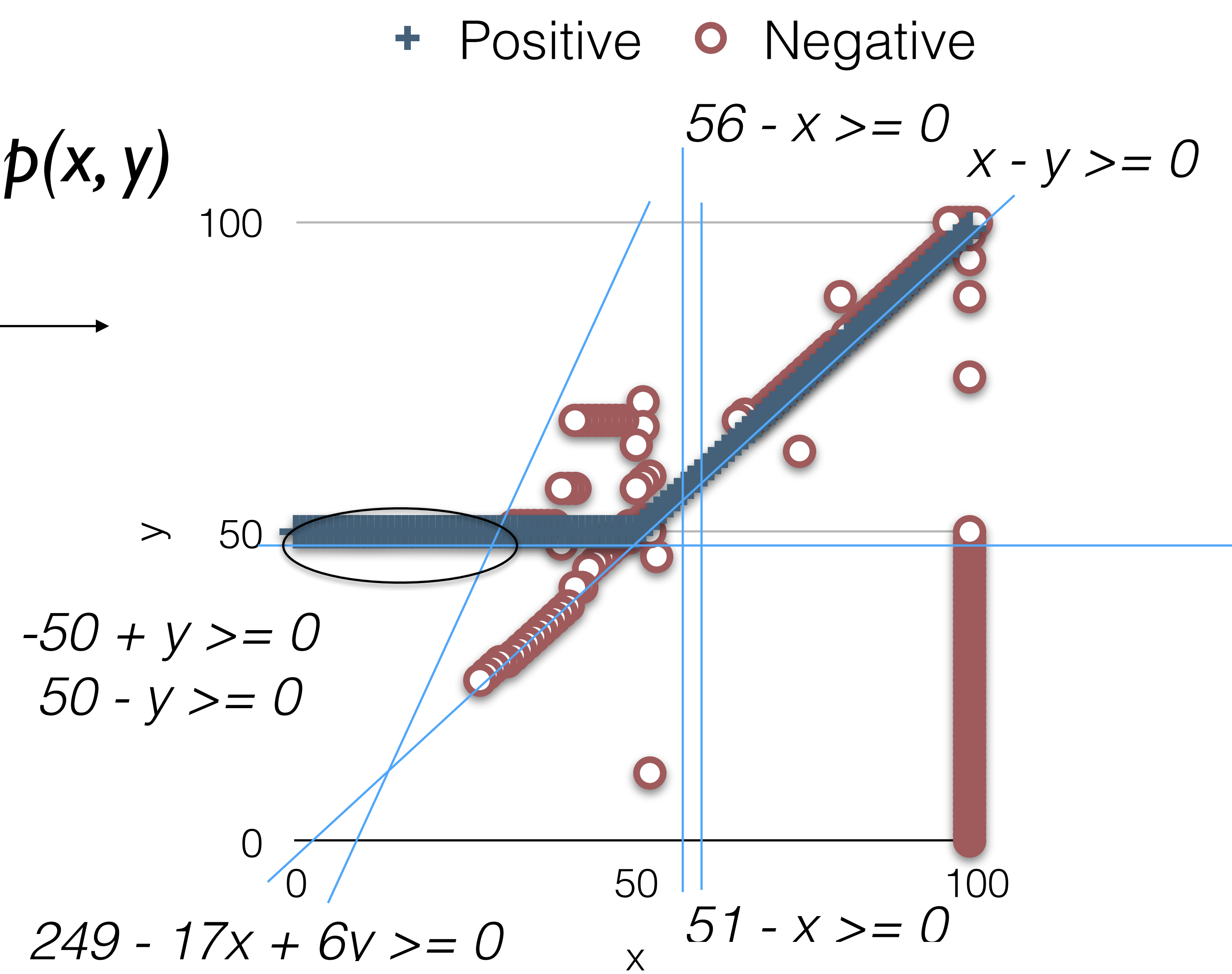
$-x - y - 1 \geq 0 \vee x + y - 1 \geq 0$

- Generality: Call linear classification by leveraging its ability to infer high quality classifiers even from data
- Safety: Call linear classification recursively until all samples are correctly separated.
- SynthHorn: Combine Generality and Safety together!

Combating Over- and Under-fitting

```
main() {
  int x,  y;
  x = 0;  y = 50;
  while (x < 100) { // p(x,y)
    x = x + 1;
    if (x > 50)  {y = y + 1;}
  }
  assert  (y == 100);
}
```

Sampling p(x, y)



$$\boxed{56 - x \geq 0} \wedge (\boxed{249 - 17x + 6y \geq 0} \vee -50 + y \geq 0 \wedge 50 - y \geq 0 \wedge \boxed{51 - x \geq 0} \vee x - y \geq 0 \wedge -x + y \geq 0) \vee x - y \geq 0 \wedge -x + y \geq 0$$

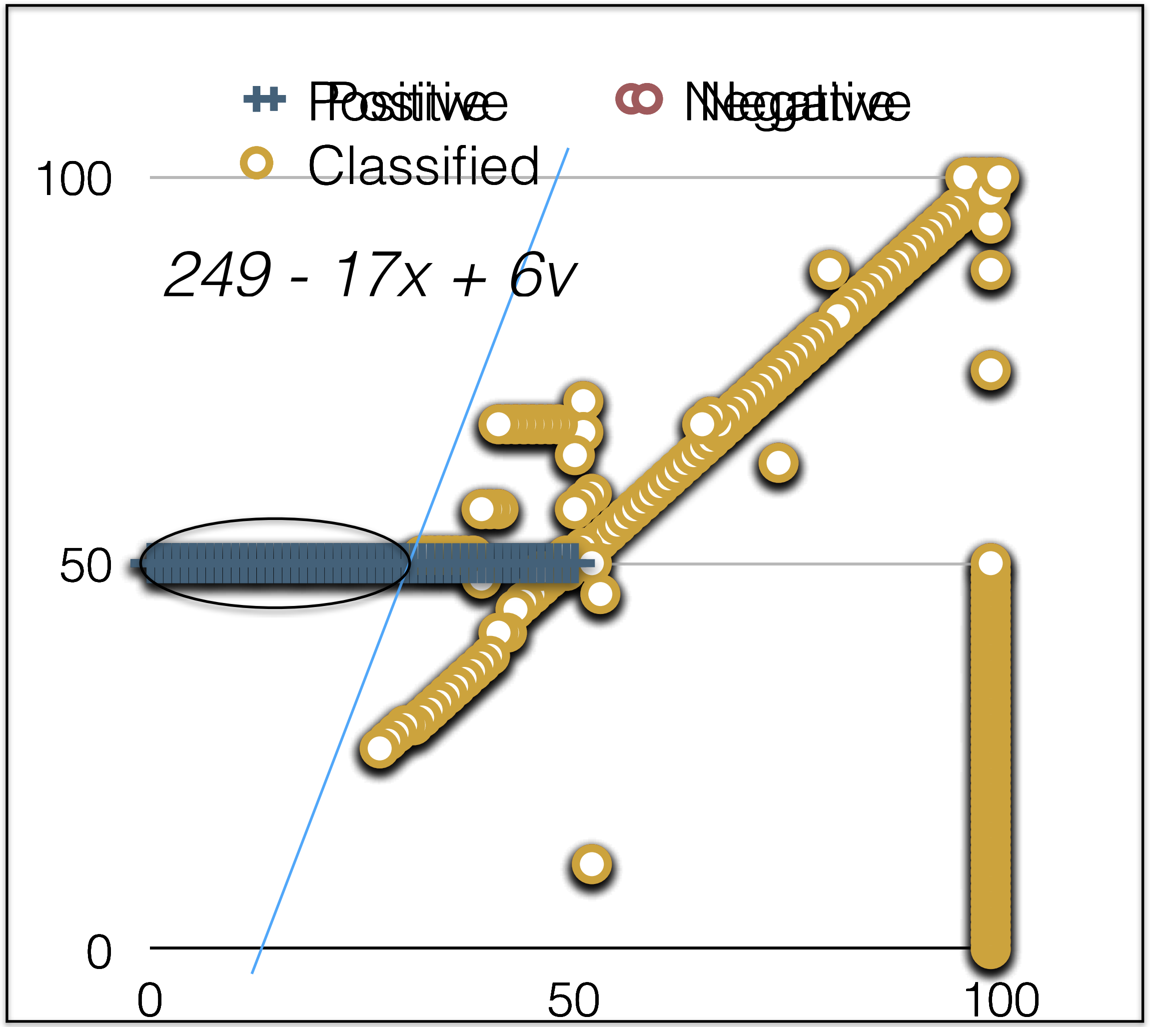
Z3

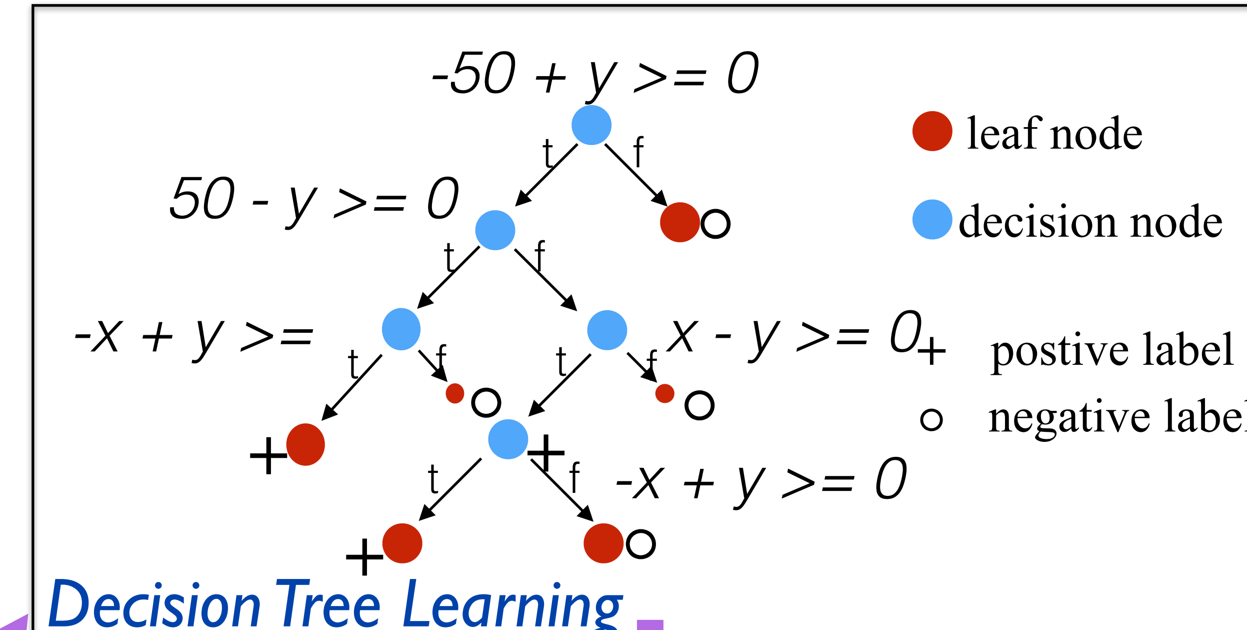
Combating Over- and Under-fitting

- Can we generalize the learned invariant solely using the data from which the linear classifiers are produced?

A <u>simple</u> invariant is more likely to generalize.

Goal: Design a learner to learn *<u>simple</u>* invariants





Data

-50 + y >= 0
50 - y >= 0
-x + y >= 0
x - y >= 0
~~56 - x >= 0~~
~~51 - x >= 0~~
~~249 - 17x + 6y >= 0~~

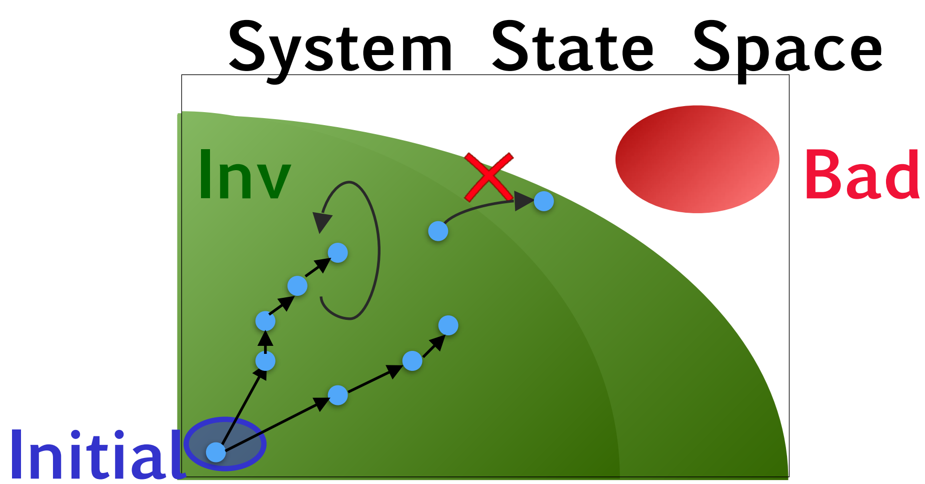
Learned classifiers from linear classification

$$p(x,y) \equiv -50 + y \geq 0 \wedge 50 - y \geq 0 \wedge -x + y \geq 0 \vee$$
$$-50 + y \geq 0 \wedge \neg(50 - y \geq 0) \wedge x - y \geq 0 \wedge -x + y \geq 0$$

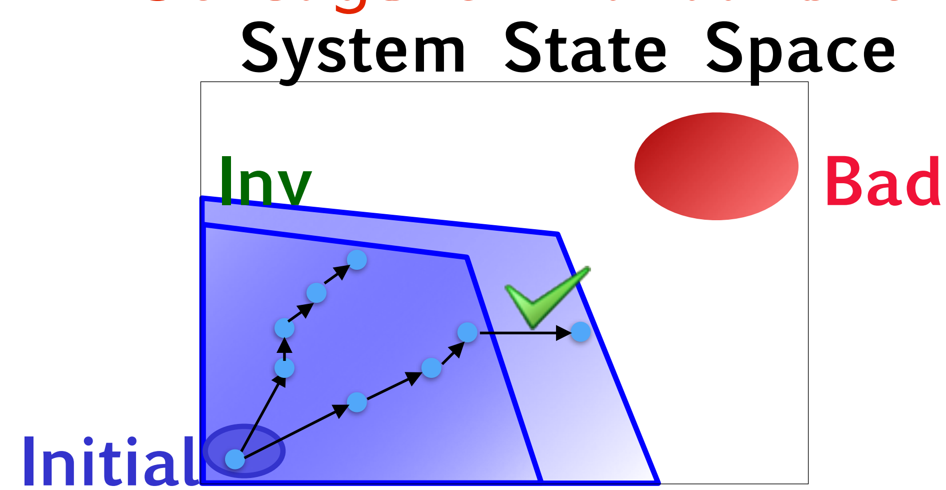
Z3

Counterexample guided sampling by Z3

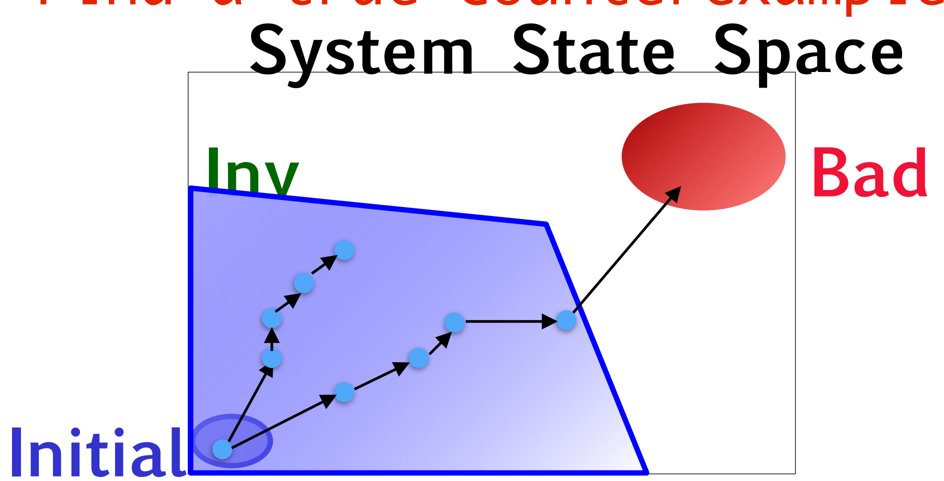
$$Tr(X, X') \wedge Inv[X] \rightarrow Inv[X']$$



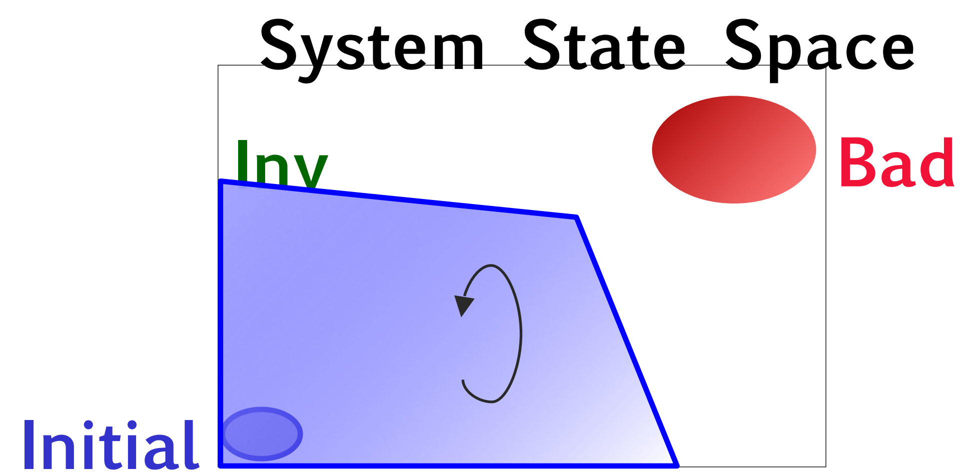
Strengthen Invariant
Weaken Invariant



Find a true counterexample

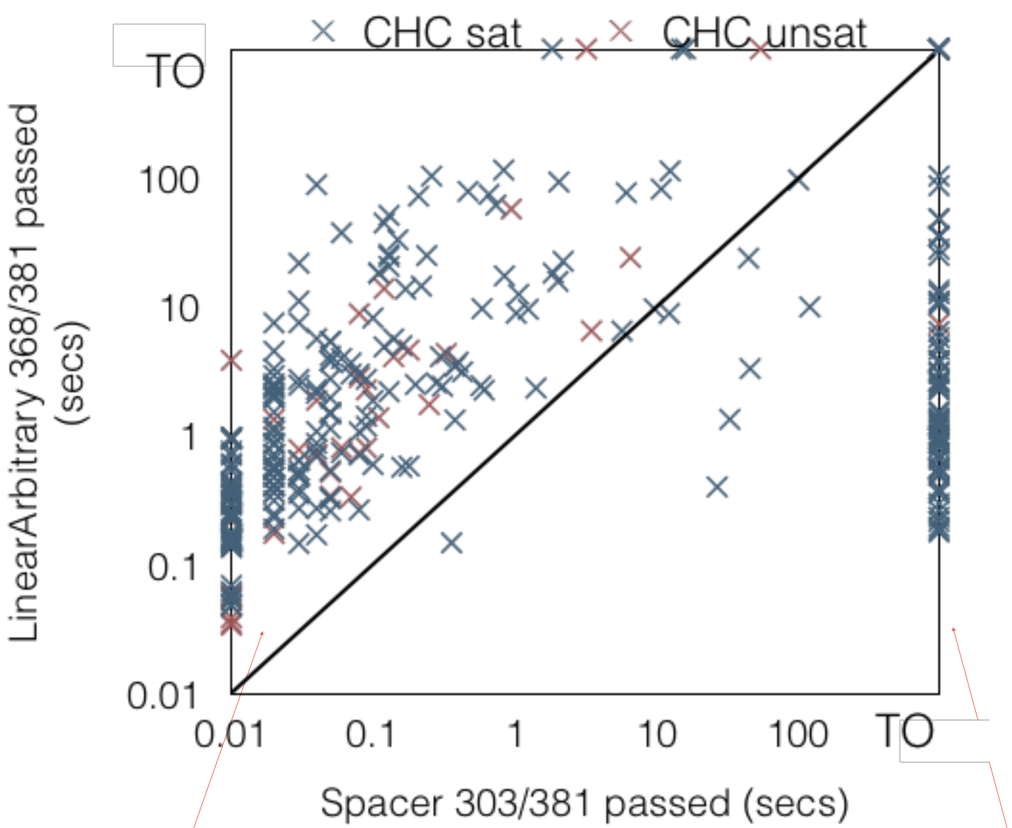


Find an inductive invariant



Experimental Results

- Collected 381 loop and recursive programs with intricate invariants



Comparison with GPDR, Spacer, Duality

Total	381
Z3-GPDR	300
Z3-Spacer	303
Z3-Duality	309
SynthHorn	368

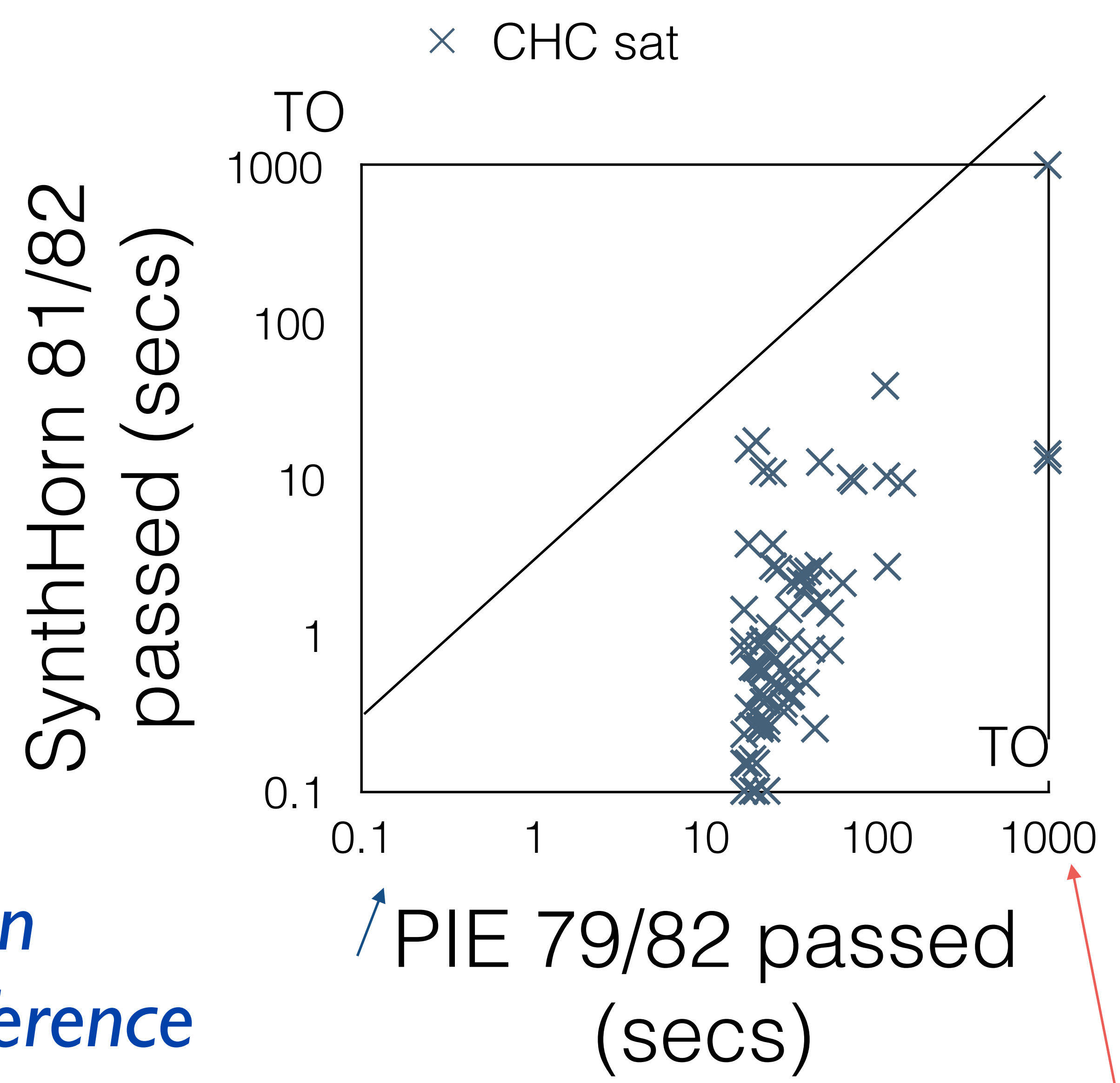
Spacer is faster

SynthHorn can verify more programs

Verified 644 programs (out of 679 considered from SV-COMP benchmarks)
Programs in excess of 10KLOC verified < 13 sec

Experimental Results

Comparison with PIE



A data-driven invariant inference tool using enumeration-based search (PLDI'16)

Machine learning leads to order-of-magnitude faster performance than enumeration

Summary

★ *Learning mechanisms provide a powerful framework for verifiable invariant inference over both data structure and numeric programs*

- Automation.
  - Leverage off-the-shelf solvers and classifiers for invariant discovery

- Guarantees.
  - The *strongest* specification (up to a hypothesis domain).

  - Ensure there *always* exists a test to refine an unverifiable specification (if hypothesis space is sufficient).

- Demonstrated applicability to real-world programs.

- Full verification pipeline.

See PLDI'18, PLDI'16, ICFP'15, VMCAI'15 for more details

★ *Extend ideas to*

- Specification inference of heap-manipulating programs (separation logic)

- Distributed protocols (inductive invariant inference on infinite-state systems)

- Program synthesis, generally