Learning to Verify the Heap

Siddharth Krishna

Marc Brockschmidt, Yuxin Chen, Byron Cook,
Pushmeet Kohli, Daniel Tarlow and He Zhu

Verifying heap-manipulating programs

```
procedure insert(lst: Node, elt: Node)
  returns (res: Node)
  requires elt ↦ null ∗ lseg(lst,null)
  ensures lseg(res,null)
{
  if (lst != null)
  {
    var curr := lst;
    while (nondet() && curr.next != null)
      invariant curr ≠ null : elt ↦ null ∗ lseg(lst,curr) ∗ lseg(curr,null)
    {
      curr := curr.next;
    }
    elt.next := curr.next;
    curr.next := elt;
    return lst;
  }
  else return elt;
}
```

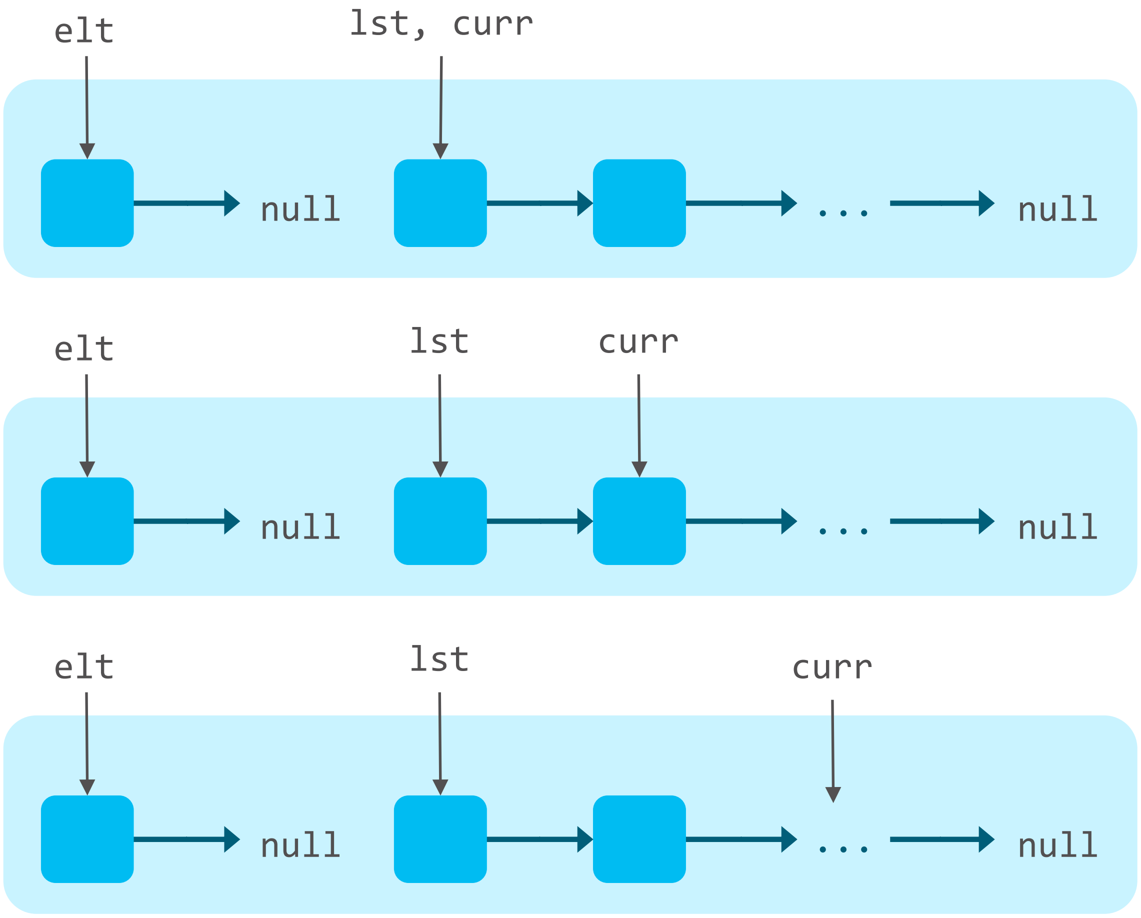
Verifying heap-manipulating programs

```
procedure insertion_sort(lst: Node)
  requires lseg(lst, null) ∗ lst ≠ null
  ensures slseg(lst, null)
{
  var prv := null, srt := lst;
  while (srt != null)
    invariant (prv = null ∗ srt = lst ∗ lseg(lst, null))
      || (lseg (lst, prv) ∗ pr ↦ srt ∗ lseg (srt, null))
  {
    var curr := srt.next;
    var min := srt;
    while (curr != null)
      invariant (prv = null ∗ lseg (lst, srt) ∗ lseg(srt, min) ∗ lseg(min, curr) ∗ lseg(curr, null))
        || (lseg(lst, prv) ∗ lseg (prv, srt) ∗ lseg(srt, min) ∗ lseg(min, curr) ∗ lseg(curr, null))
      invariant min ≠ null
    {
      if (curr.data < min.data) {
        min := curr;
      }
      curr := curr.next;
    }
    var tmp := min.data;
    min.data := srt.data;
    srt.data := tmp;
    prv := srt;
    srt := srt.next;
  }
}
```

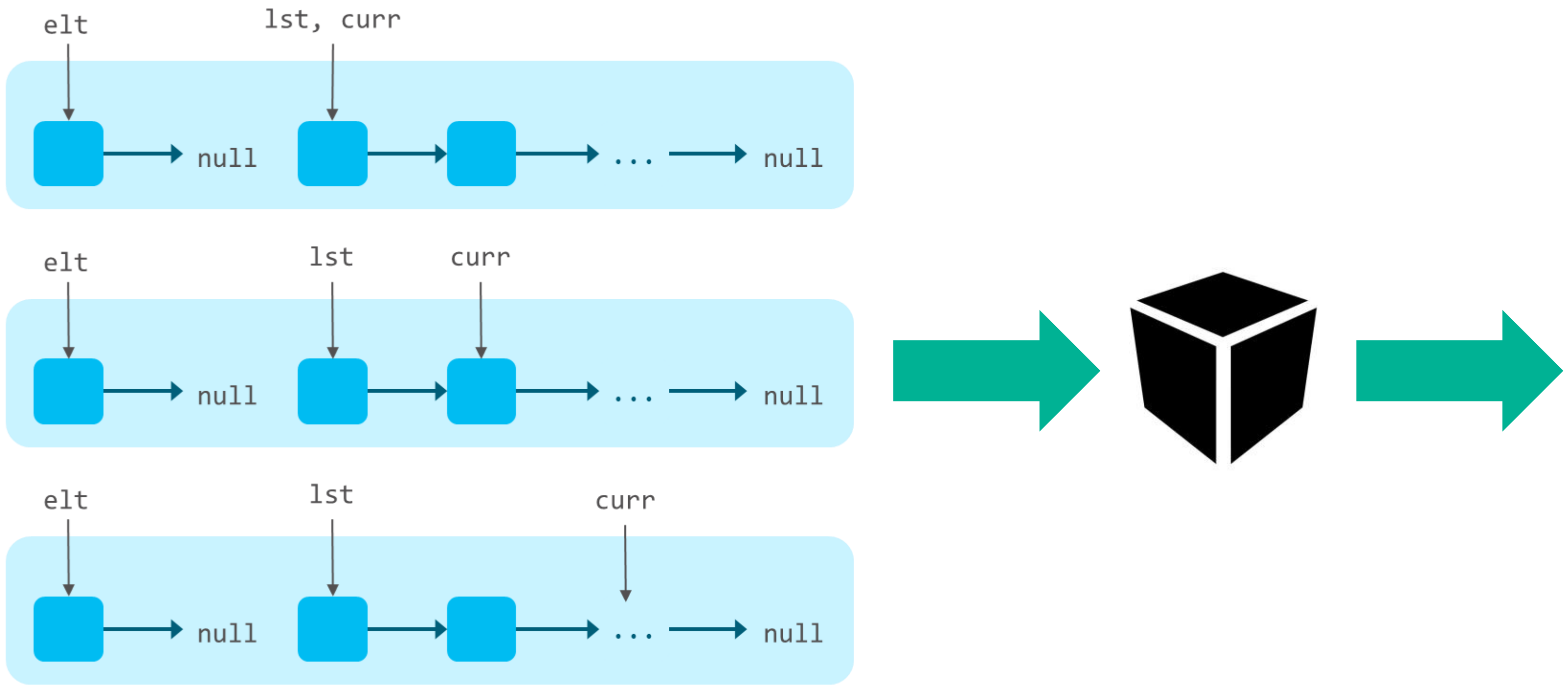
But how do we find these annotations?

Data

```
procedure insert(lst: Node, elt: Node)
  returns (res: Node)
  requires elt ↦ null ∗ lseg(lst,null)
  ensures lseg(res,null)
{
  if (lst != null)
  {
    var curr := lst;
    while (nondet() && curr.next != null)
    {
      invariant curr ≠ null : elt ↦ null ∗ lseg(lst,curr)
                                   ∗ lseg(curr,null)
      curr := curr.next;
    }
    elt.next := curr.next;
    curr.next := elt;
    return lst;
  }
  else return elt;
}
```



Machine Learning



$$curr \neq null : \\ elt \mapsto null \ * \ lseg(lst, curr) * \ lseg(curr, null)$$

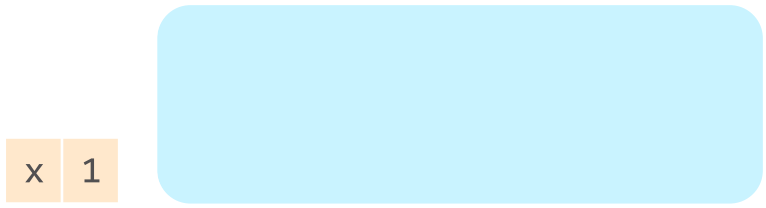
In this talk

- ML based formula prediction
  - Arbitrary (pre-defined) inductive predicates
  - Nested predicates
  - Disjunctions
  - Predictions are not training $\Rightarrow$ fast
- Refinement loop with program verifier
- Data invariants
  - Functional correctness
- Fully automatically verify programs
  - merge & quick sort

Separation Logic

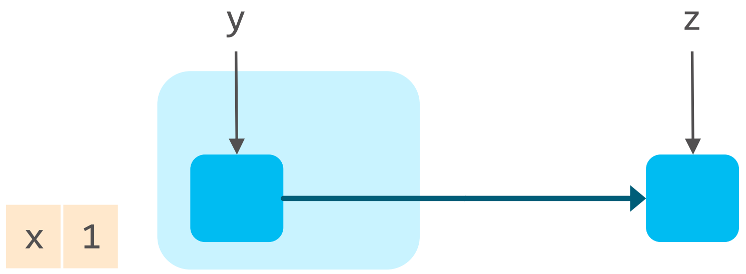
Separation Logic

emp



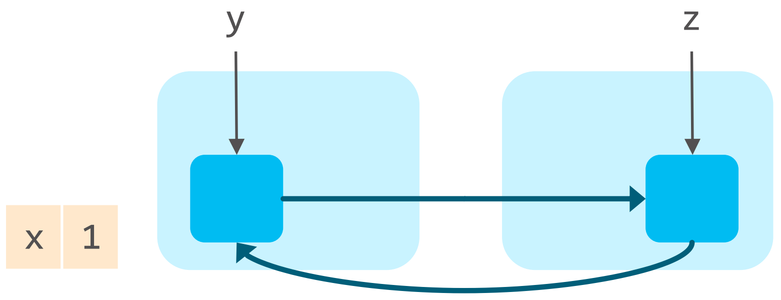
Separation Logic

$$y \mapsto z$$



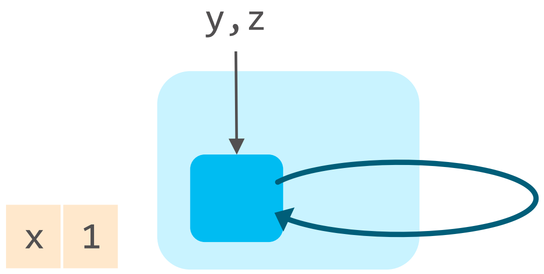
Separation Logic

$$y \mapsto z \; * z \mapsto y$$



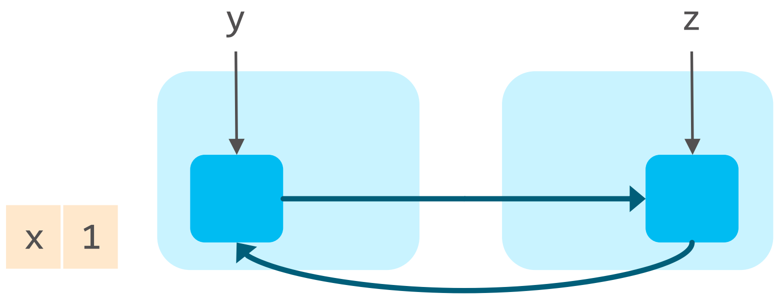
Separation Logic

$$y \mapsto z \wedge z \mapsto y$$



Separation Logic

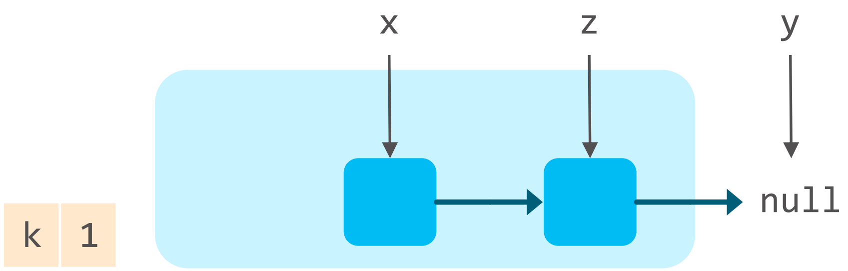
$$x = 1 : y \mapsto z * z \mapsto y$$



SL of list segments

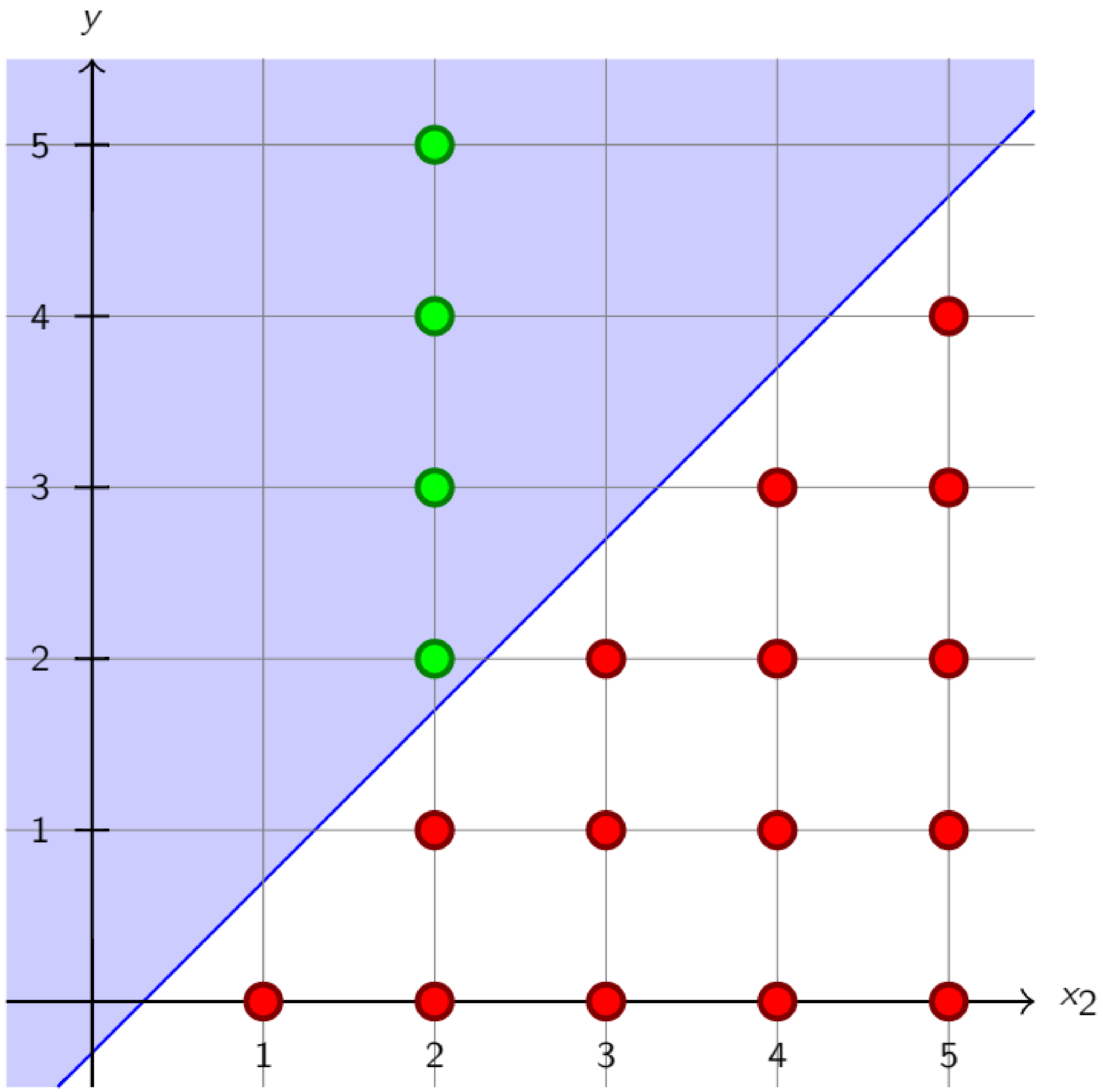
$$lseg(x, y) \coloneqq \exists\, z. (x = y : emp) \vee (x \neq y : x \mapsto z \ * lseg(z, y))$$

$lseg(x, null)$
$\equiv \exists z \,.\, x \neq null : x \mapsto z \ * lseg(z, null)$
$\equiv \ \ldots$
$\equiv x \mapsto z \ * z \mapsto null \ * emp$



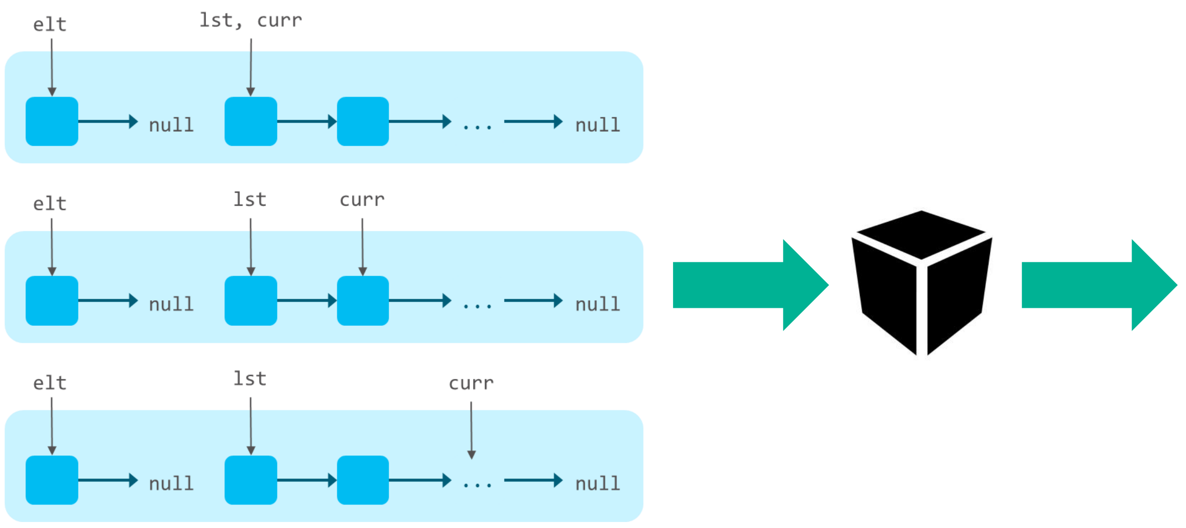
Learning

Previous ML: numeric



$(y \geq x)$

The problem



$$curr \neq null : \ elt \mapsto null \ * \\ lseg(lst, curr) * \ lseg(curr, null)$$

Graphs → Formulas

Solution

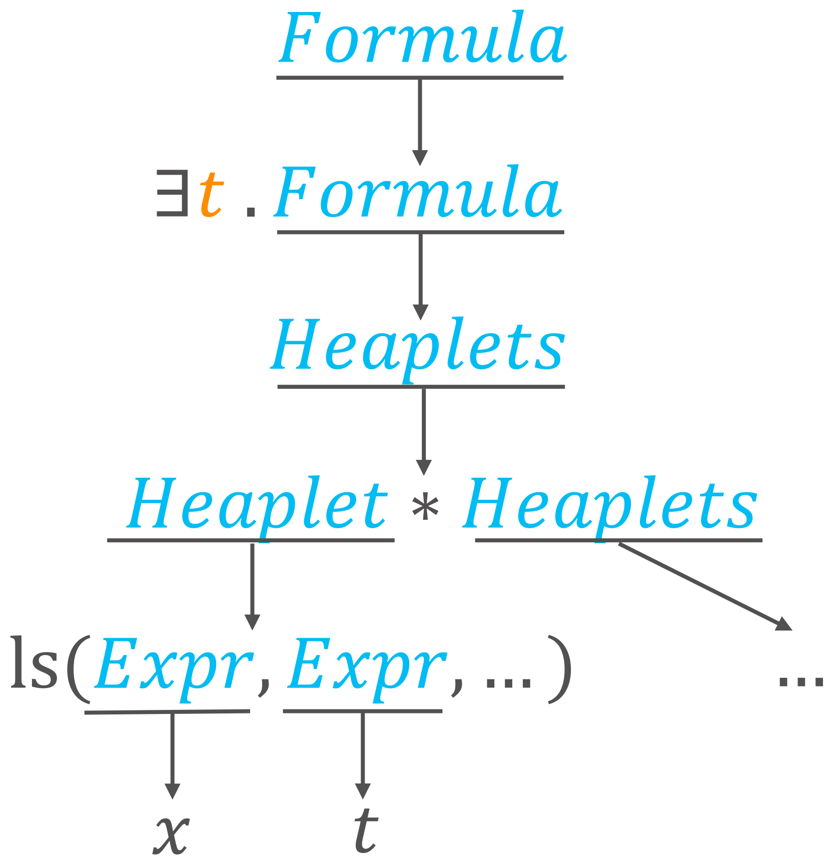
- Formulas as parse trees from grammar:

$$
\begin{aligned}
Formula &\rightarrow \exists\, Var\,.\, Formula \mid Heaplets \\
Heaplets &\rightarrow Heaplet * Heaplets \mid emp \\
Heaplet &\rightarrow \mathrm{ls}(Expr, Expr, \lambda Var, Var, Var, Var \rightarrow Formula) \\
&\quad \mid \mathrm{tree}(Expr, \lambda Var, Var, Var, Var \rightarrow Formula) \\
Expr &\rightarrow 0 \mid Var
\end{aligned}
$$

Solution

$$
\begin{aligned}
\mathit{Formula} &\to \exists\, \mathit{Var} \,.\, \mathit{Formula} \mid \mathit{Heaplets} \\
\mathit{Heaplets} &\to \mathit{Heaplet} * \mathit{Heaplets} \mid \mathit{emp} \\
\mathit{Heaplet} &\to \mathrm{ls}(\mathit{Expr}, \mathit{Expr}, \lambda \mathit{Var}, \mathit{Var}, \mathit{Var}, \mathit{Var} \to \mathit{Formula}) \\
&\quad \mid \mathrm{tree}(\mathit{Expr}, \lambda \mathit{Var}, \mathit{Var}, \mathit{Var}, \mathit{Var} \to \mathit{Formula}) \\
\mathit{Exp} &\to 0 \mid \mathit{Var}
\end{aligned}
$$

$$ls(x, t, \_) * ls(t, t, \_)$$



Key idea

- Each production step is a classification task

$$\begin{aligned} \mathit{Heaplet} \quad &\to \mathrm{ls}(\mathit{Expr}, \mathit{Expr}, \lambda \mathit{Var}, \mathit{Var}, \mathit{Var}, \mathit{Var} \to \mathit{Formula}) \\ &\mid \mathrm{tree}(\mathit{Expr}, \lambda \mathit{Var}, \mathit{Var}, \mathit{Var}, \mathit{Var} \to \mathit{Formula}) \end{aligned}$$

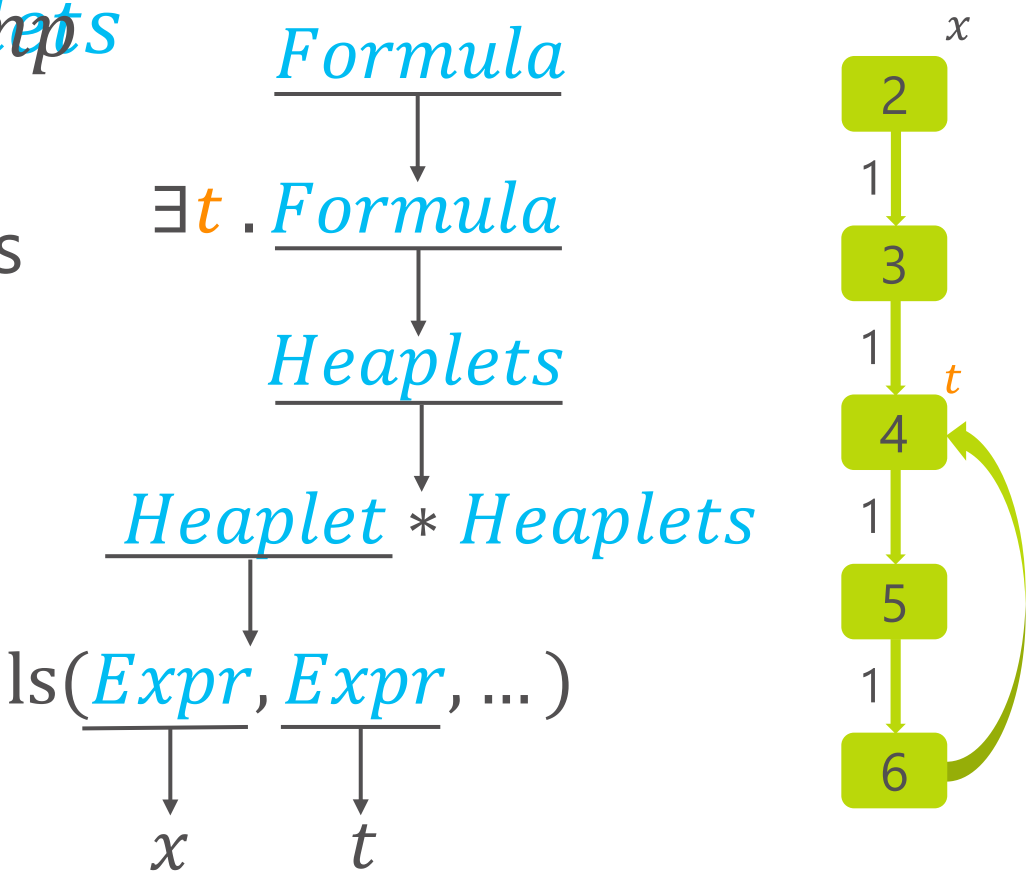
- Input graphs → $\phi \in \mathbb{R}^d$ feature vector
- Example:
  - Feature: ϕ = max degree
  - Classifier: $ITE(\phi < 3, ls, tree)$

Prediction

Sequence of production choices

$$Heaplet \to \mathrm{ls}(Expr, Expr, \ldots)$$
$$\mid \mathrm{tree}(Expr, \ldots)$$

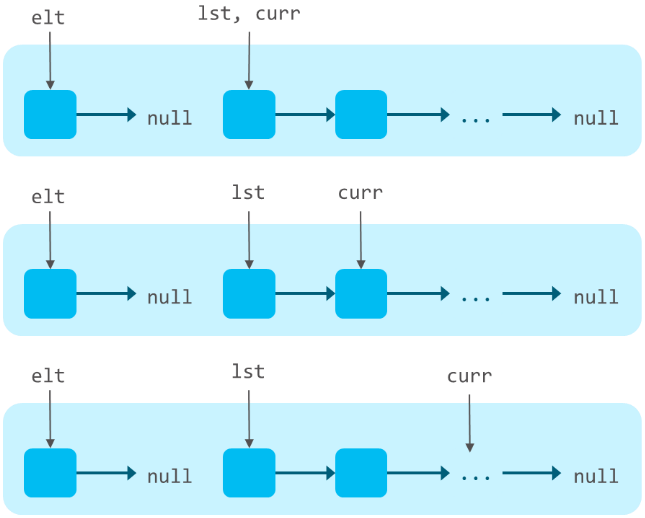
1. Existentials: Name “special” graph nodes
2. Number of heap parts
3. Type of data structure
4. Argument names
5. Repeat for nested structures



Training data

- Enumerate formulas & satisfying states
  - Fast and plenty!

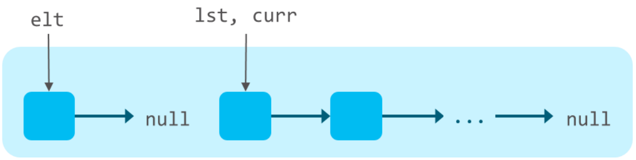
$$lseg(lst, curr)$$
$$lseg(curr, lst)$$
$$lseg(lst, curr) * lseg(curr, null)$$
$$\ldots$$
$$curr \neq null : elt \mapsto null * lseg(lst, curr) * lseg(curr, null)$$
$$\ldots$$

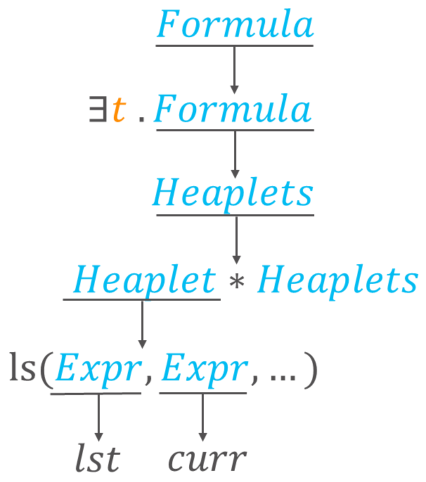


Training data

- For each (formula, state) pair:
  - Walk through parse tree and generate labelled training data for each predictor

$curr \neq null : elt \mapsto null * lseg(lst, curr) * lseg(curr, null)$





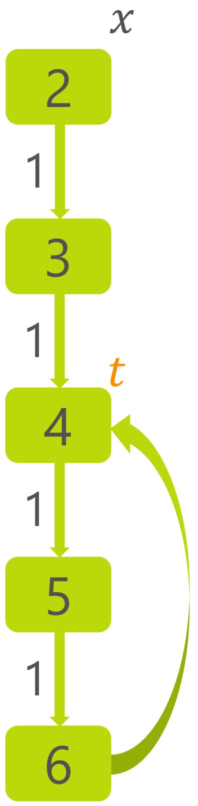
$Heaplet \rightarrow \mathrm{ls}(Expr, Expr, \_) \mid \mathrm{tree}(Expr, \_)$

$\phi = (0, 1, 2, 5, 1, 1, 0, \dots, 1)\ , P = \mathrm{ls}$

Features used

$$Formula \rightarrow \exists\, Var\,.\,Formula \mid Heaplets$$

$$\begin{array}{c} Formula \\ \downarrow \\ \exists t\,.\,Formula \end{array}$$



For every node:

- Part of / can reach an SCC
- In- and out-degrees
- Above/below average in/out degrees

ML Predictor: NN with maximum likelihood

Features used

$$Heaplets \rightarrow Heaplet * Heaplets \mid emp$$

For every node:

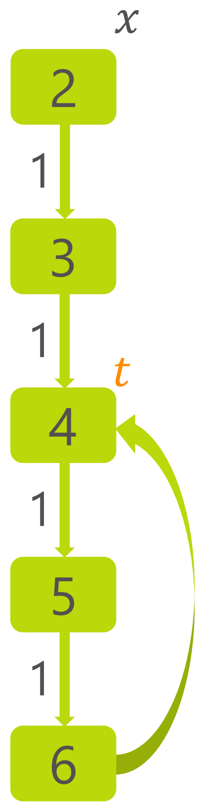
- 1-gram = (in-degree, out-degree)
- 1-gram depth

For the graph:

- Frequencies of 1-grams and 2-grams

ML predictor: logistic regression

$$\underline{Formula}$$
$$\downarrow$$
$$\exists t\, .\, \underline{Formula}$$
$$\downarrow$$
$$\underline{Heaplets}$$
$$\downarrow$$
$$Heaplet * Heaplets$$



Features used

$$
\begin{aligned}
Heaplet \;&\to \mathrm{ls}(Expr, Expr, \_) \\
&\mid \mathrm{tree}(Expr, \_)
\end{aligned}
$$

For every node not defined so far:

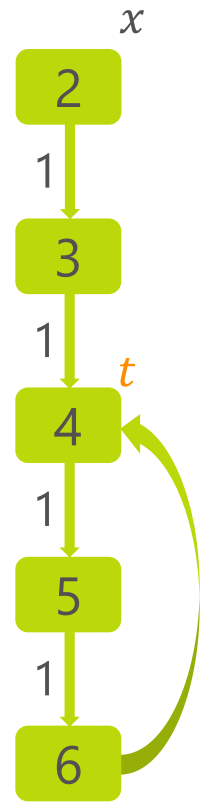
- In- and out-degree features from before

For the set of such nodes:

- Num of nodes of in/out degree $\leq k$

ML predictor: logistic regression

$$
\begin{array}{c}
\underline{Formula} \\
\downarrow \\
\exists t\,.\,\underline{Formula} \\
\downarrow \\
\underline{Heaplets} \\
\downarrow \\
\underline{Heaplet} * Heaplets \\
\downarrow \\
\mathrm{ls}(Expr, Expr, \ldots)
\end{array}
$$



Features used

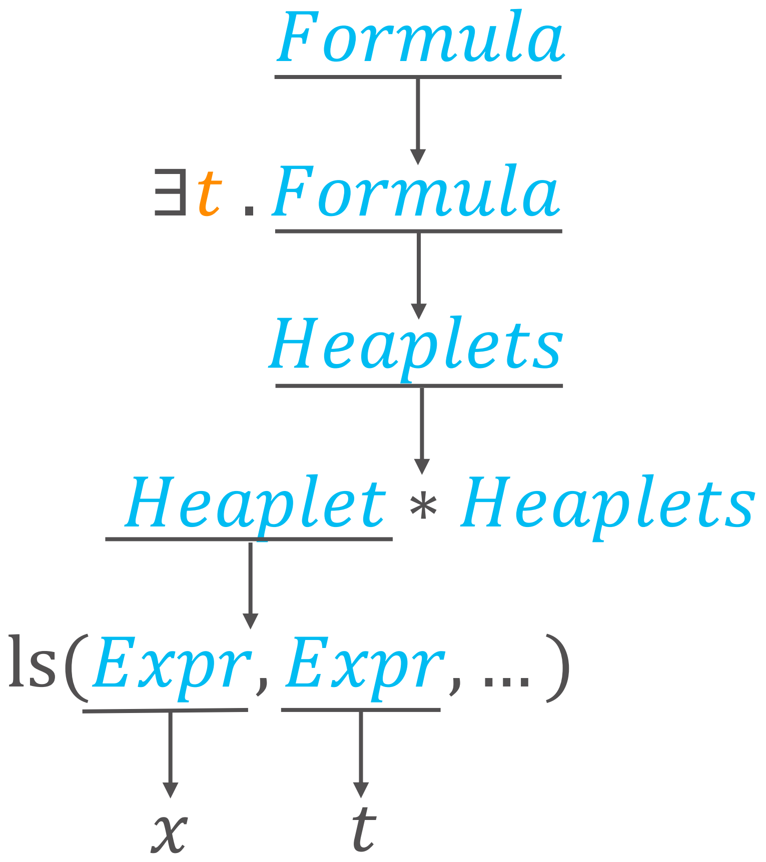
$Expr \rightarrow 0 \mid Var$

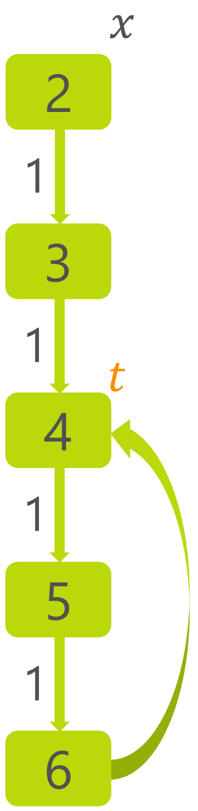
Consider *enclosed defining identifiers.*

For every identifier x and enclosed defining identifier e:

- x reaches/reached by e
- x directly reaches/reached by e
- x simply reaches/reached by e
- x is part of/reaches an SCC
- Frequency of 1 and 2-grams reachable by x

ML predictor: NN with maximum likelihood





Comparison to previous ML

Numeric Invariants:

- Training at verification time
- Training data: Observations
- Desired Invariant
  ~ Model

Our Heap Invariants:

- Training beforehand
- Training data: Independent
- Desired Invariant
  ~ Predicted Label

Platypus

Algorithm 1 Pseudocode for PlatypusCore

Input: Grammar G, input objects $\hat{\mathcal{H}}$, (partial) parse tree $\mathcal{T} = (\mathcal{A}, g, ch)$, nonterminal node a to expand

$N \leftarrow g(a)$ {nonterminal symbol of a in $\mathcal{T}$}
$\boldsymbol{\phi} \leftarrow \boldsymbol{\phi}^N(\hat{\mathcal{H}}, \mathcal{T})$ {compute features}
$P \leftarrow$ most likely production $N \rightarrow \mathcal{S}^+$ from G considering $\boldsymbol{\phi}$
$\mathcal{T} \leftarrow$ insert new nodes into $\mathcal{T}$ according to P
for all children $a' \in ch(a)$ labeled by nonterminal **do**
    $\mathcal{T} \leftarrow \mathsf{PlatypusCore}(G, \hat{\mathcal{H}}, \mathcal{T}, a')$
return $\mathcal{T}$

Aside: nested data structures

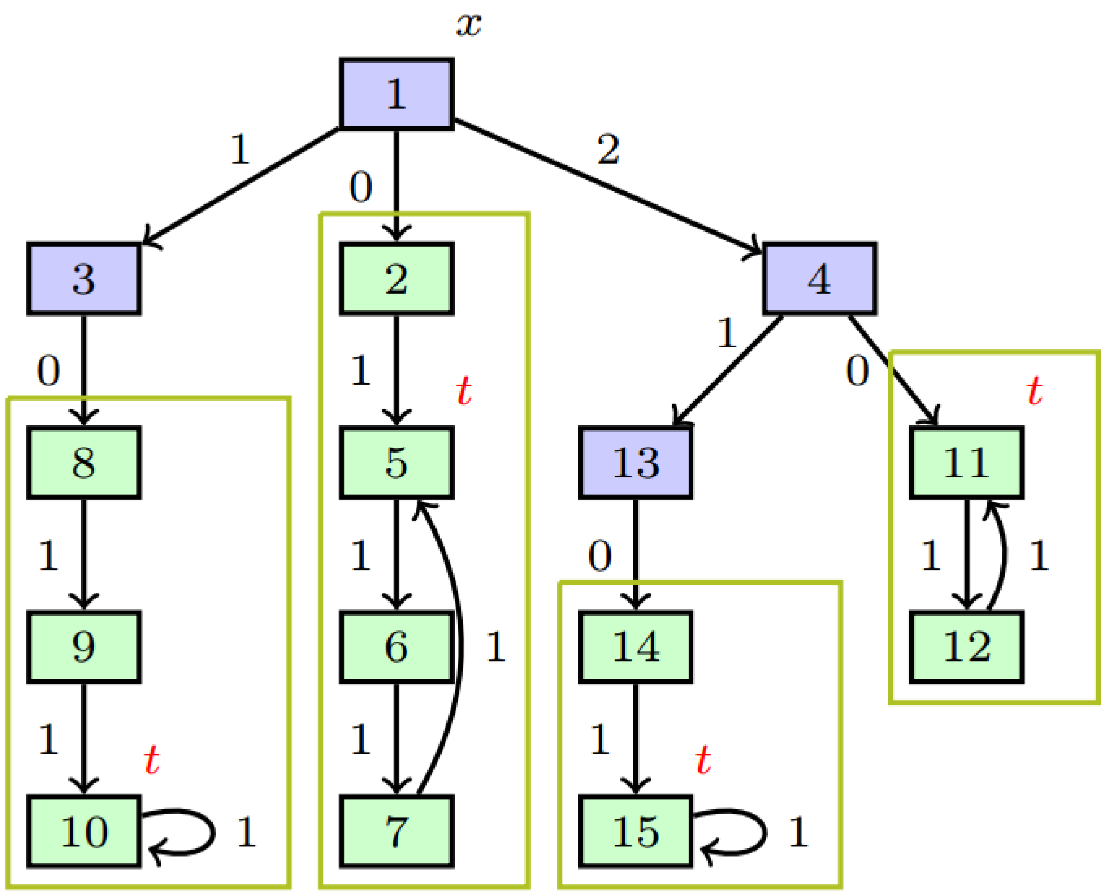
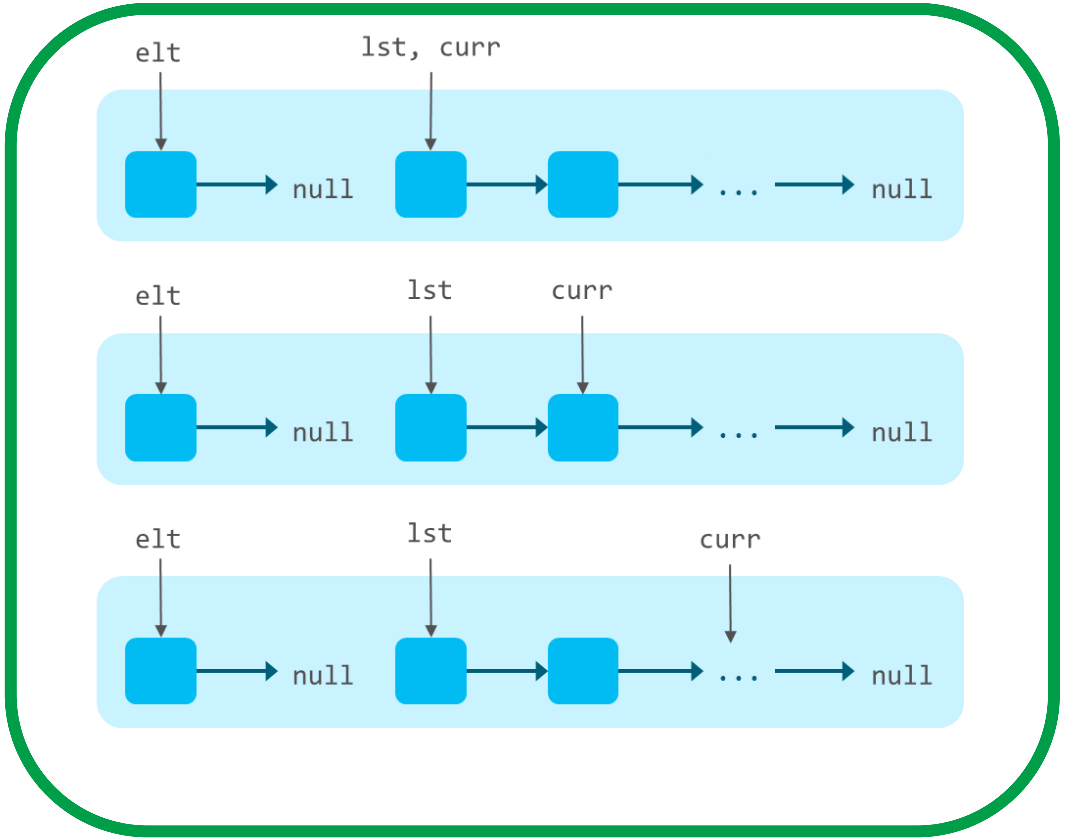


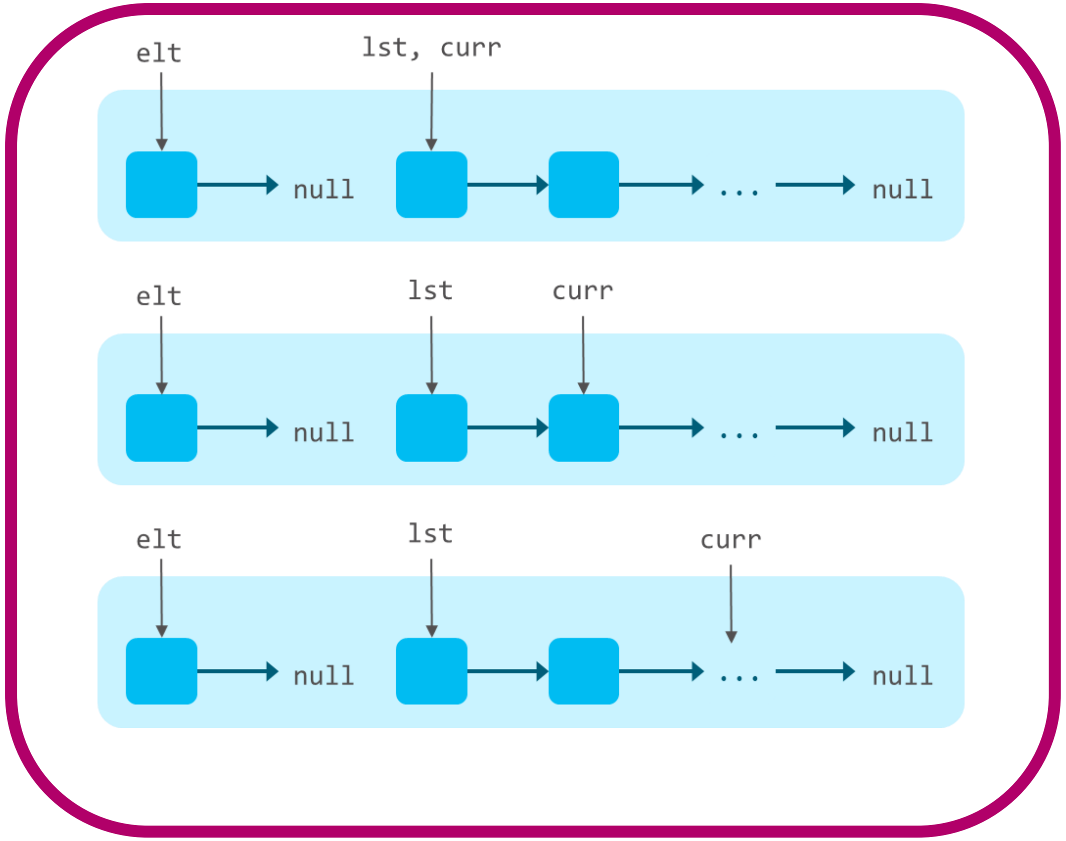
Fig. 4: Binary tree of panhandle lists described by the formula $\mathsf{tree}(x, \lambda i_1, i_2, i_3, i_4 \to \exists t.\mathsf{ls}(i_2, t, \top) * \mathsf{ls}(t, t, \top))$

Disjunctions

Disjunctions

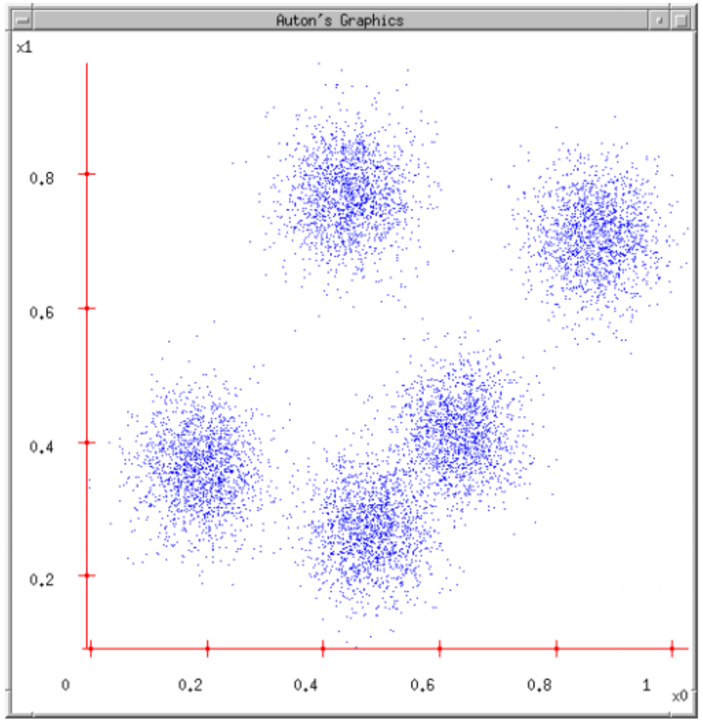
- Clustering!





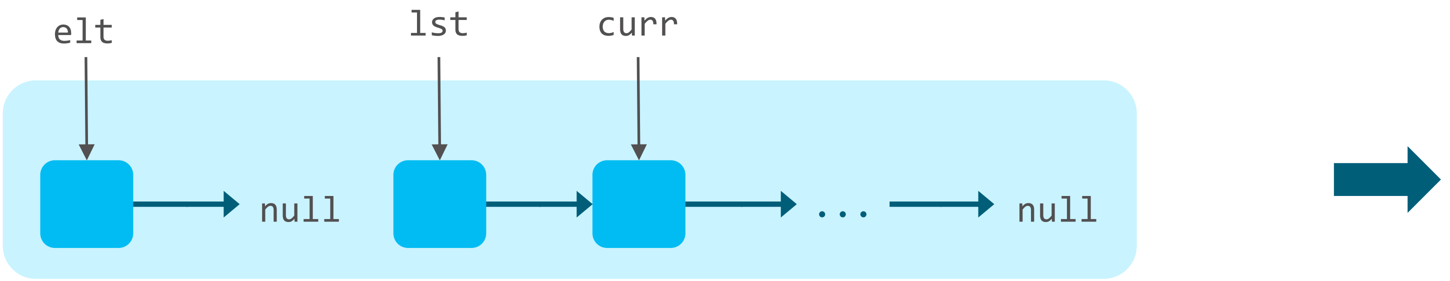
Clustering

- Group together similar points
  - Similar: distance measure
  - We use Euclidean distance
- Input:
  - $\hat{H} \to \phi^{\hat{H}}$ as points in $\mathbb{R}^n$
- Output:
  - Disjunction of prediction for each cluster



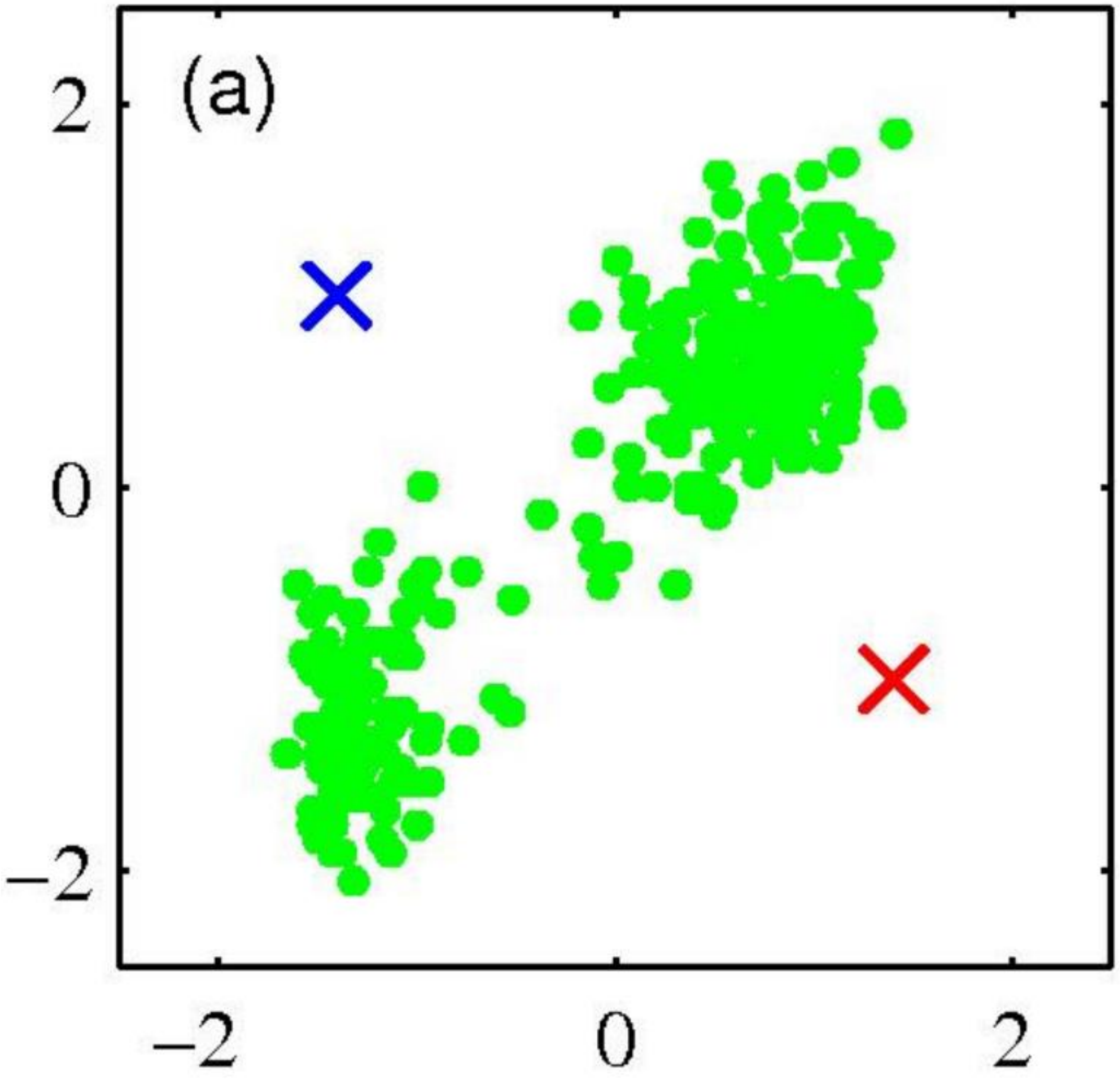
Features

- Reachability
  - Between every two variables
  - Reflexive, without passing through other variables



(0, 0, 0, 1, 0, 0)

K-means

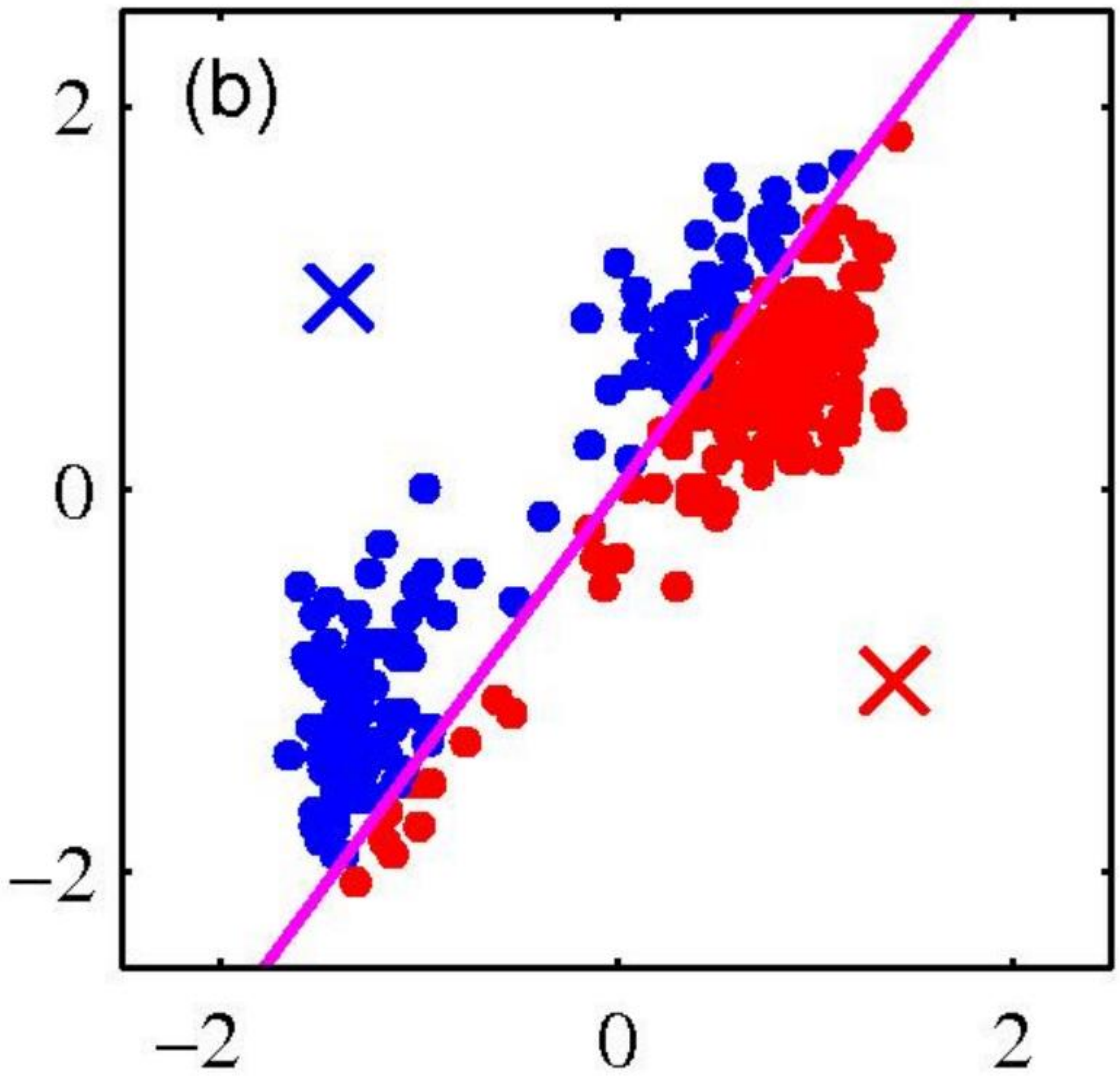


- Pick K random points as cluster centers (means)

Shown here for K=2

These slides from David Sontag, adapted from Luke Zettlemoyer, Vibhav Gogate, Carlos Guestrin, Andrew Moore, Dan Klein

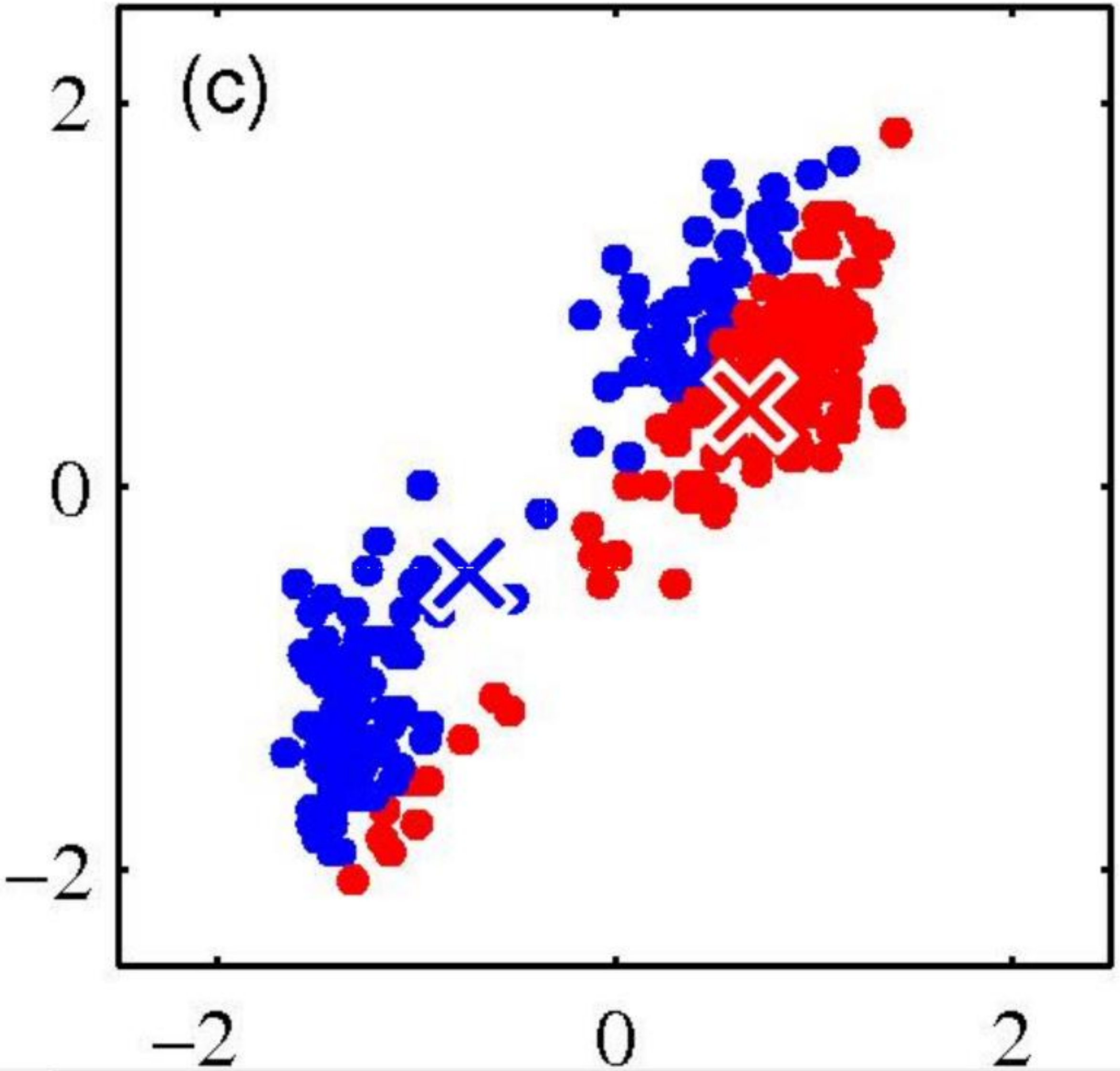
K-means



Iterative Step 1

- Assign data points to closest cluster center

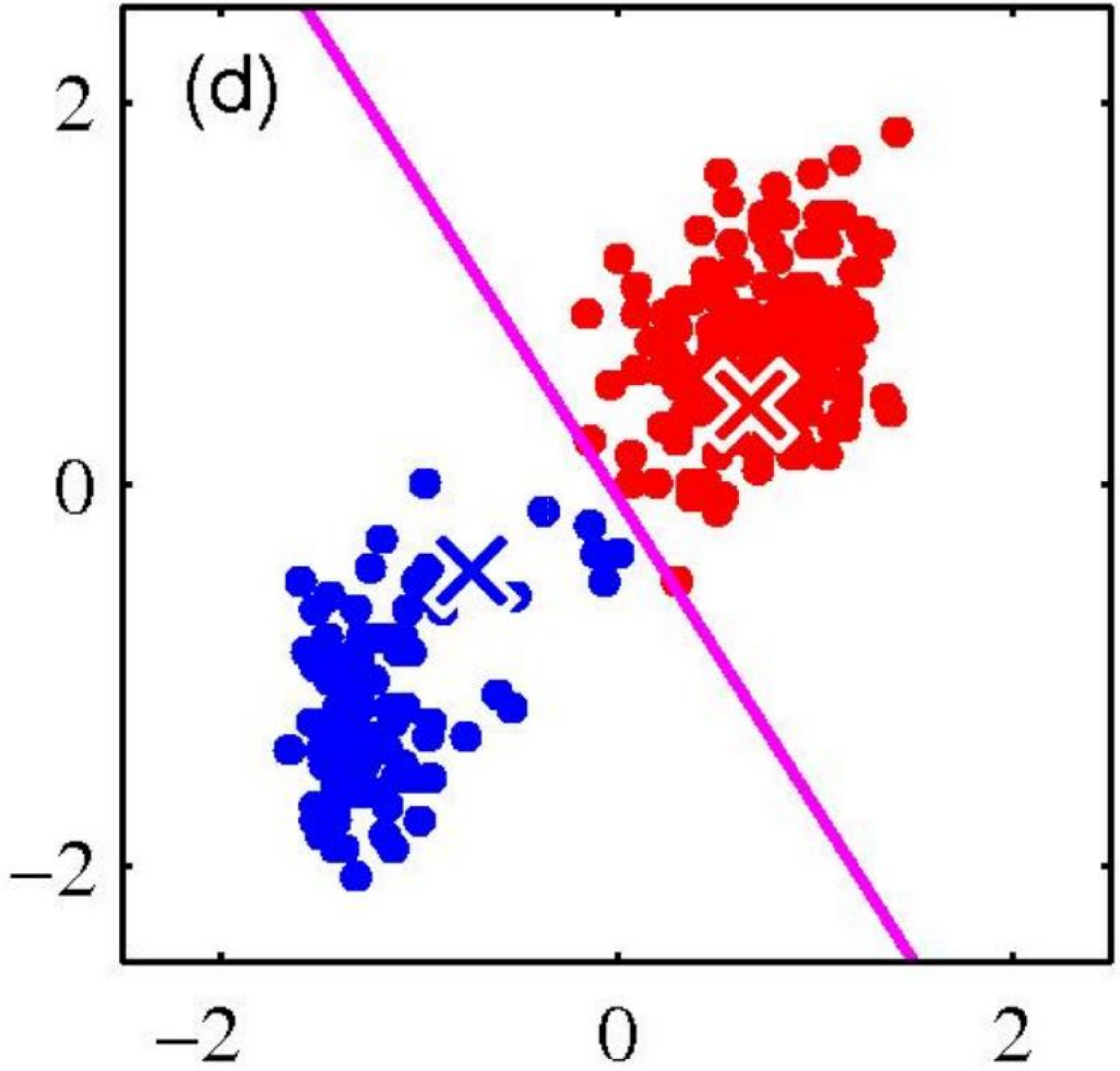
K-means



Iterative Step 2

- Change the cluster center to the average of the assigned points

K-means



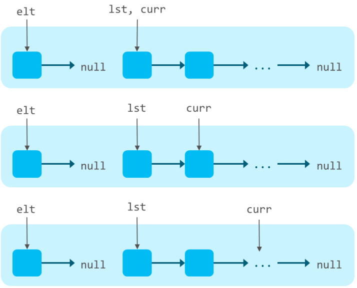
- Repeat until convergence

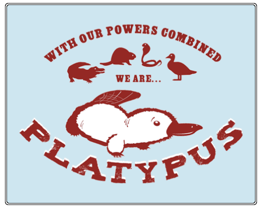
Refinement

Are we done?

Introducing... Locust

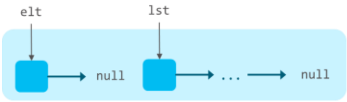
```
procedure insert(lst: Node, elt: Node)
  returns (res: Node)
  requires elt ↦ null ∗ lseg(lst,null)
  ensures lseg(res,null)
{
  if (lst != null)
  {
    var curr := lst;
    while ( ? && curr.next != null)
    {
      curr := curr.next;
    }
    elt.next := curr.next;
    curr.next := elt;
    return lst;
  }
}
```



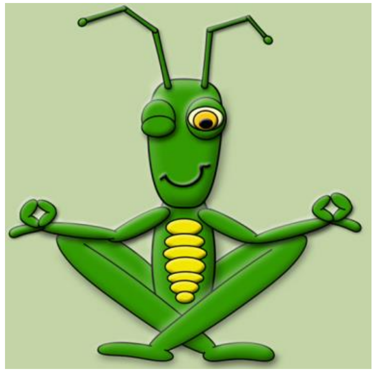


$curr \neq null:$
$elt \mapsto null \,*\, lseg(lst,curr) * lseg(curr,null)$

```
procedure insert(lst: Node, elt: Node)
  returns (res: Node)
  requires elt ↦ null ∗ lseg(lst,null)
  ensures lseg(res,null)
{
  if (lst != null)
  {
    var curr := lst;
    while ( ? && curr.next != null)
      invariant curr ≠ null : elt ↦ null
      ∗ lseg(lst,curr) ∗ lseg(curr,null)
    {
      curr := curr.next;
    }
    elt.next := curr.next;
    curr.next := elt;
    return lst;
  }
}
```



no



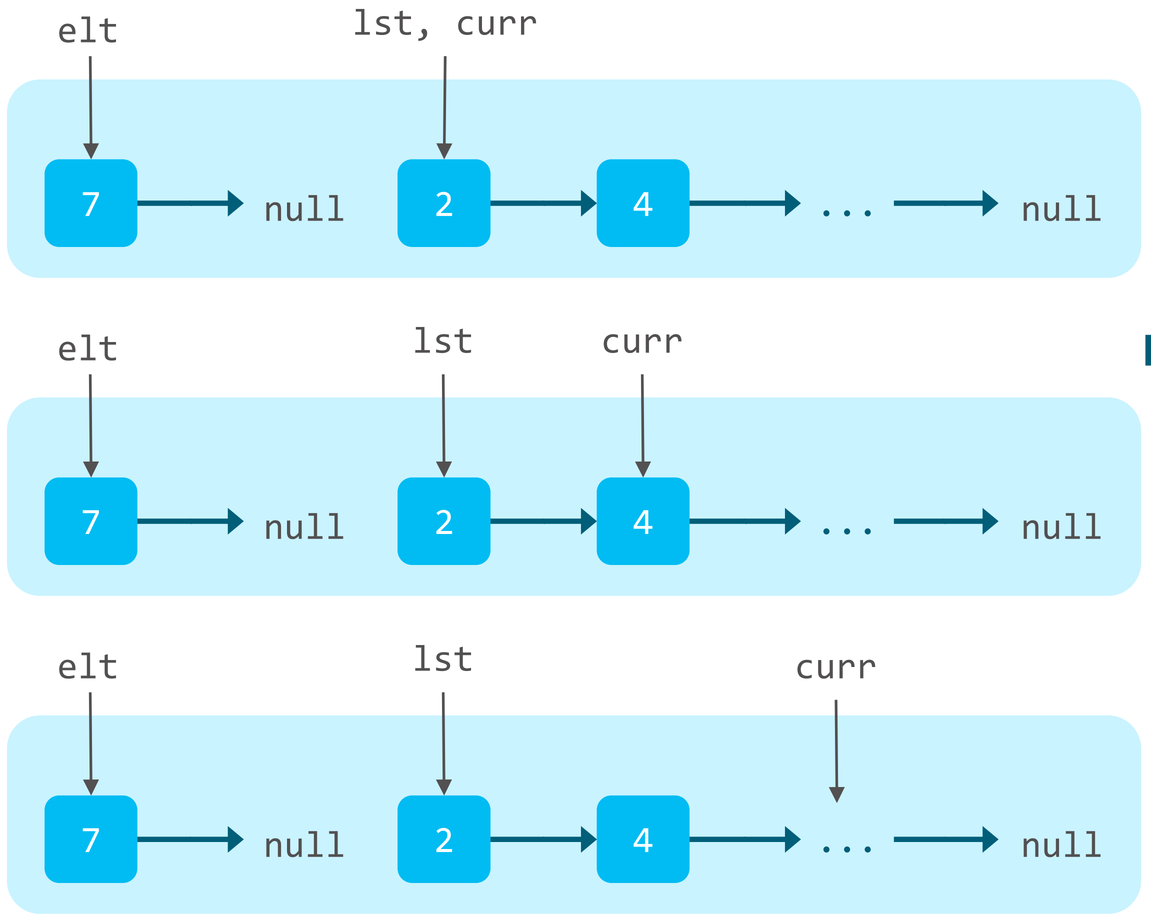
GRASShopper

yes

Grasshopper picture from this blog

Data invariants

Beetle



lseg(lst, curr)	[]	[2]	[2, 4]
lseg(curr, null)	[2, 4, 6, 9]	[4, 6, 9]	[6, 9]
lseg(elt, null)	[7]	[7]	[7]
lst.data	2	2	2
curr.data	2	4	6
elt.data	7	7	7

$$\forall u.\, u \in FP\big(lseg(lst, curr)\big) \Rightarrow u.data < curr.data$$

$$\forall u, v.\, FP\big(lseg(lst, curr)\big): u \to^{+} v \Rightarrow u.data \leq v.data$$

$$\forall u, v.\, FP\big(lseg(curr, null)\big): u \to^{+} v \Rightarrow u.data \leq v.data$$

$$curr.data \leq elt.data$$

Beetle

- Abstract domain:
  - Quantified Octagons ($x \leq y$) + Reachability ($u \rightarrow^{+} v$)

$$\Omega_{\text{shape}}(\varphi_\ell) = \{u \in \mathsf{FP}(d), \mathsf{FP}(d) : u \rightarrow^{+} v, \mathsf{FP}(d) : u \searrow v \mid d \in Mem(\varphi_\ell)\}$$

$$\begin{aligned}\Omega_{fld}(\ell) = &\{u.fld \leq x, x \leq u.fld, u.fld \leq x.fld, x.fld \leq u.fld \mid x \in Vars(\ell)\} \\ &\cup \{u.fld \leq v.fld, v.fld \leq u.fld\}\end{aligned}$$

- Look for invariants of the form: $\forall u, v.\, \omega_{shape} \Rightarrow \psi_{fld}$

Analog to functional version from:

He Zhu, Gustavo Petri, Suresh Jagannathan, *Automatically Learning Shape Specifications*, PLDI 2016

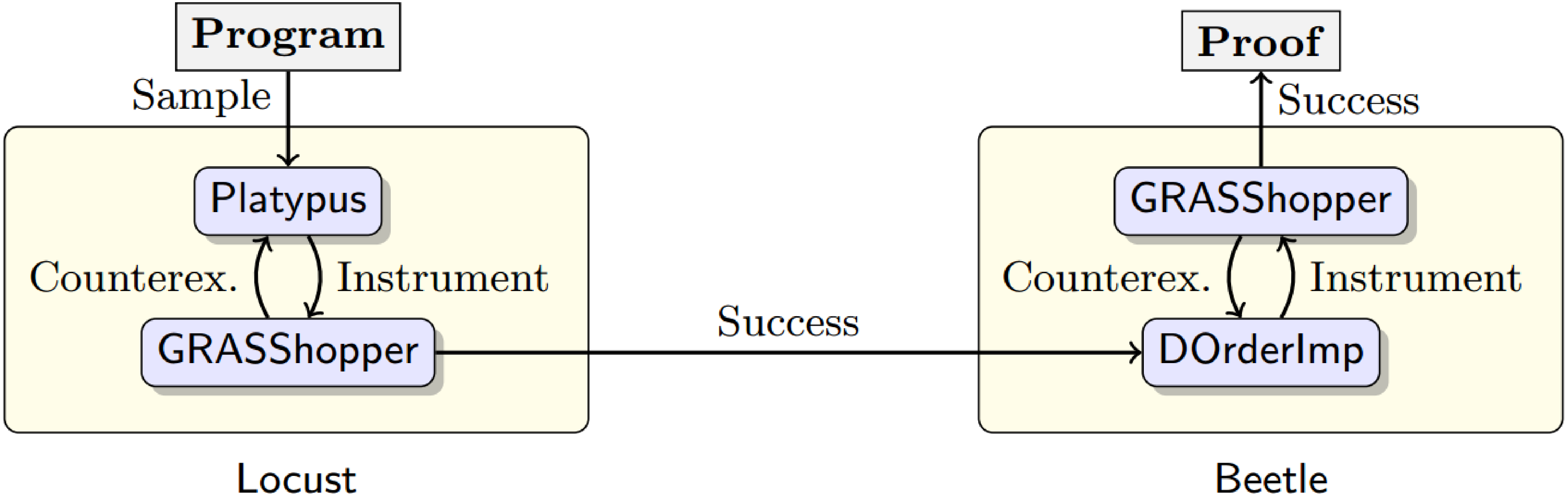
CEGAR loop

Algorithm 3 Pseudocode for Beetle

Input: Program P and entry procedure p with precondition φ_p, locations L requiring program annotations, shape annotations φ_ℓ for $\ell \in L$

```
I ← sample initial states satisfying φ_p                                  {see Sect. 4.1}
S⁺ ← execute p on I to map location ℓ ∈ L to set of observed states
while true do
    for all ℓ ∈ L do
        φ_ℓ^sd ← DOrderImp(φ_ℓ, S⁺(ℓ), S⁻(ℓ))
    P′ ← annotate P with inferred φ_ℓ^sd
    if GRASShopper(P′) returns counterexample s then
        if s is new counterexample then
            update S⁺, S⁻ to contain s for correct location               {see Sect. 4.2}
        else return FAIL
    else return SUCCESS
```

Cricket



Picture from [Aqua Teen Hunger Force Wikia]

Results:

Memory safety + functional correctness for linked list programs:

Example	Platypus	Locust	DOrderImp	Beetle	Cricket
`concat`	2s	✓ 68s (1 it.)	32s	✓ 34s (4 it.)	102s
`copy`	2s	✓ 30s (1 it.)	8s	✓ 18s (2 it.)	52s
`dispose`	1s	✓ 4s (1 it.)	0s	✓ 1s (2 it.)	4s
`double_all`	2s	✓ 29s (1 it.)	–	✗	–
`filter`	2s	✓ 6s (1 it.)	1s	✓ 5s (2 it.)	12s
`insert`	2s	✓ 8s (1 it.)	1s	✓ 3s (1 it.)	11s
`insertion_sort`	5s	✓ 23s (1 it.)	41s	✓ 207s (30 it.)	249s
`merge_sort`	15s	✓ 83s (4 it.)	21s	✓ 159s (41 it.)	273s
`pairwise_sum`	2s	✓ 254s (1 it.)	–	✗	–
`quicksort`	5s	✓ 21s (1 it.)	24s	✓ 41s (11 it.)	67s
`remove`	2s	✓ 7s (1 it.)	19s	✓ 24s (5 it.)	31s
`reverse`	2s	✓ 8s (1 it.)	1s	✓ 4s (1 it.)	12s
`strand_sort`	17s	✓ 85s (5 it.)	–	✗	–
`traverse`	2s	✓ 6s (1 it.)	1s	✓ 1s (1 it.)	7s

Going beyond:

- Platypus: features picked manually
- Automate?
- Gated Graph Sequence Neural Networks
  - Heap graphs as input
  - Can learn reachability properties
  - Accuracy: 89.11% -> 89.96%

Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. *Gated graph sequence neural networks*. In ICLR '16, 2016.

In this talk

- ML based formula prediction
  - Arbitrary (pre-defined) inductive predicates
  - Nested predicates
  - Disjunctions
  - Predictions are not training $\Rightarrow$ fast
- Refinement loop with program verifier
- Data invariants
  - Functional correctness
- Fully automatically verify programs
  - merge & quick sort