Automatically Learning Shape Specifications

He Zhu Gustavo Petri Suresh Jagannathan

(S^3)

PURDUE UNIVERSITY

Software Verification ...

Software Verification ...

Software Verifier

Software Verification ...

```
let ocaml_merge_tree
(t1:tree) (t2:tree) =
(//recursively merge t1, t2;
 ... ...;
 ... ...;
 ... ...;
)
```

Software Verification ...

Specifications
Programs
Software Verifier

```
let ocaml_merge_tree
(t1:tree) (t2:tree) =
_(requires BST(t1) && BST(t2))
_(requires Set(t1) < Set(t2))
_(ensures BST(\res))
_(ensures Set(\res)=
                Set(t1)++Set(t2))
 (//recursively merge t1, t2;
  ... ...;
  ... ...;
  ... ...;
 )
```

Specifications are as *complex* as code

Software Verification ...

Inductive Inv
Specifications
Programs
Software Verifier

```
let ocaml_merge_tree
(t1:tree) (t2:tree) =
_(requires BST(t1) && BST(t2))
_(requires Set(t1) < Set(t2))
_(ensures BST(\res))
_(ensures Set(\res)=
                Set(t1)++Set(t2))
 (//recursively merge t1, t2;
  ... ...;
  ... ...;
  ... ...;
 )
```

Specifications are as *complex* as code

Software Verification ...

Inductive Inv
Specifications
Programs
Software Verifier

```
let ocaml_merge_tree
(t1:tree) (t2:tree) =
_(requires BST(t1) && BST(t2))
_(requires Set(t1) < Set(t2))
_(ensures BST(\res))
_(ensures Set(\res)=
                Set(t1)++Set(t2))
 (//recursively merge t1, t2;
  ... ...;
  ... ...;
  ... ...;
 )
```

Specifications are as *complex* as code

A Programmer's Day ...

A Programmer's Day ...

Defining data structures ...

A Programmer's Day ...

Defining data structures ...

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

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
```

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
```

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
```

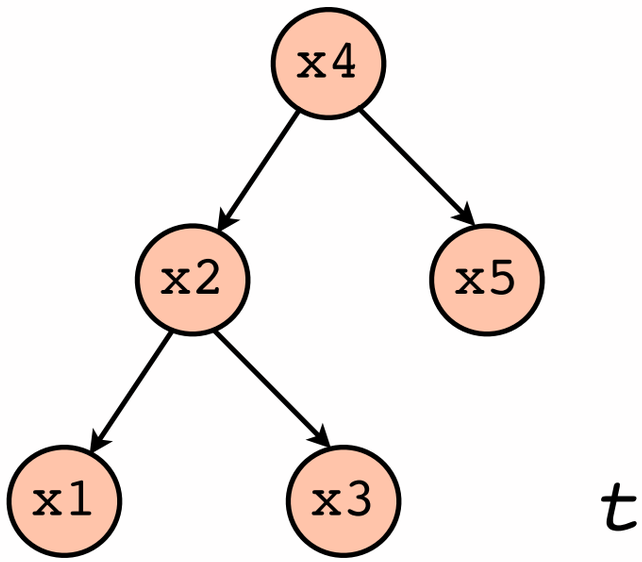
No assertions / pre-conditions / post-conditions!

A Programmer's Day ...

Testing code ...

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

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

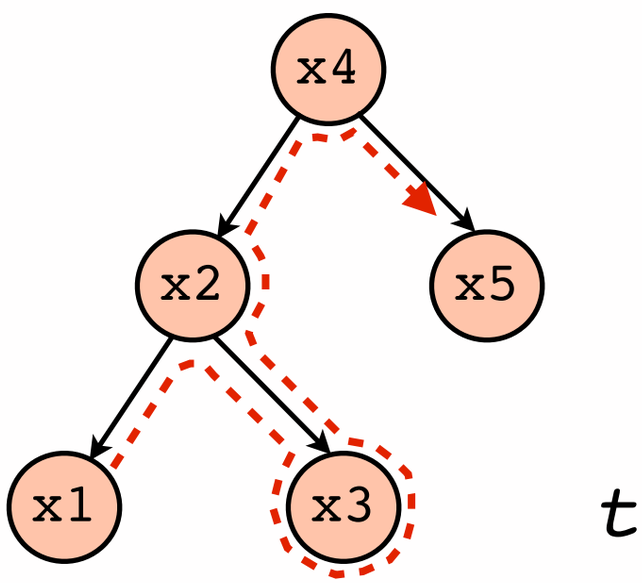


A Programmer's Day ...

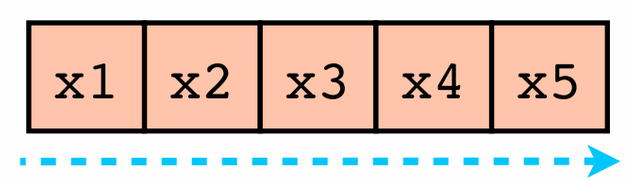
Testing code ...

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

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



t



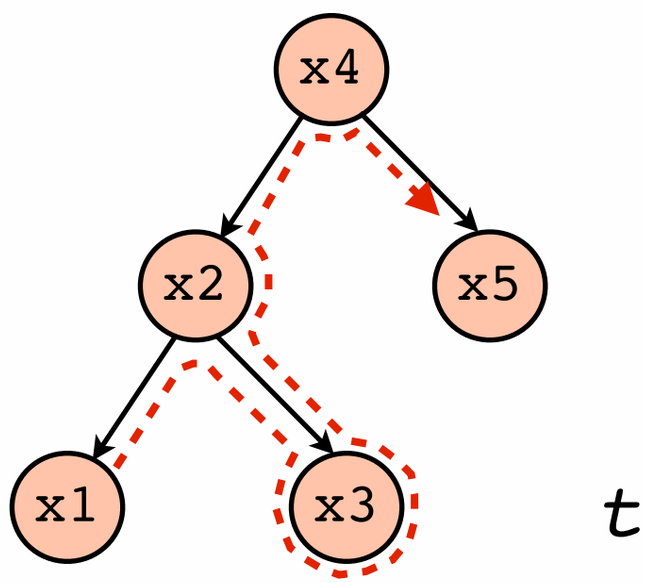
l

A Programmer's Day ...

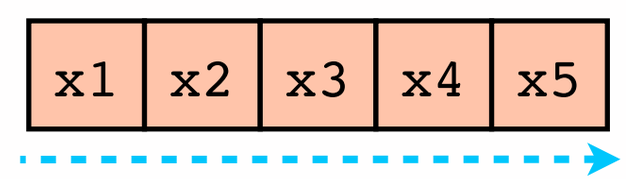
Testing code ...

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

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



t



l

```
// specification:
// elements: 'a tree -> 'a list
// l = elements t
//
```

// ⋯⋯⋯⋯⋯⋯ ≡ ⋯⋯⋯⋯⋯⋯

```
// in-order(t)    forward-order(l)
```

From a programmer's day ...

From a programmer's day ...

Programmers
write tests

From a programmer's day ...

Programmers
write tests

Verifiers
require specifications

From a programmer's day ...

Programmers
write tests

Gap

Verifiers
require specifications

From a programmer's day ...

Programmers
write tests

Gap

Verifiers
require specifications

<u>Vision</u>:

A general specification can be discovered from a small number of tests

From a programmer's day ...

Programmers write tests	Gap	Verifiers require specifications

<u>Vision</u>:

A general specification can be discovered from a small number of tests

Goal: Design a learner to automatically <u>discover software specifications</u>

From a programmer's day ...

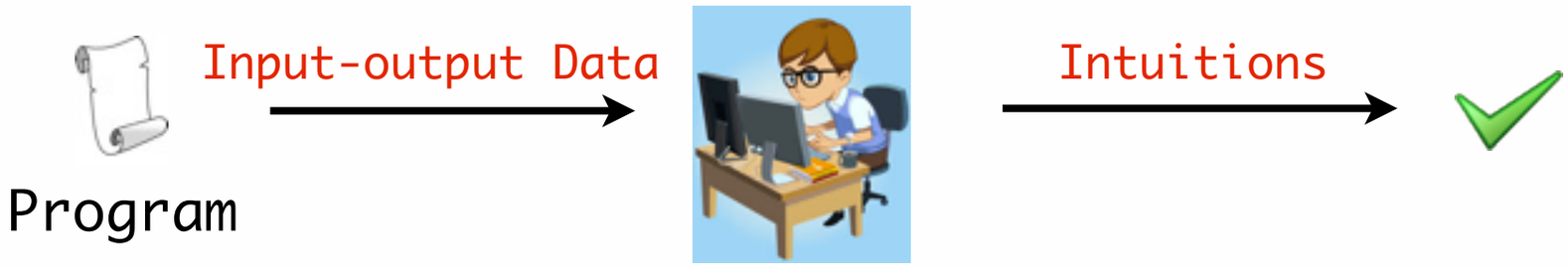
Programmers
write tests

Verifiers
require specifications

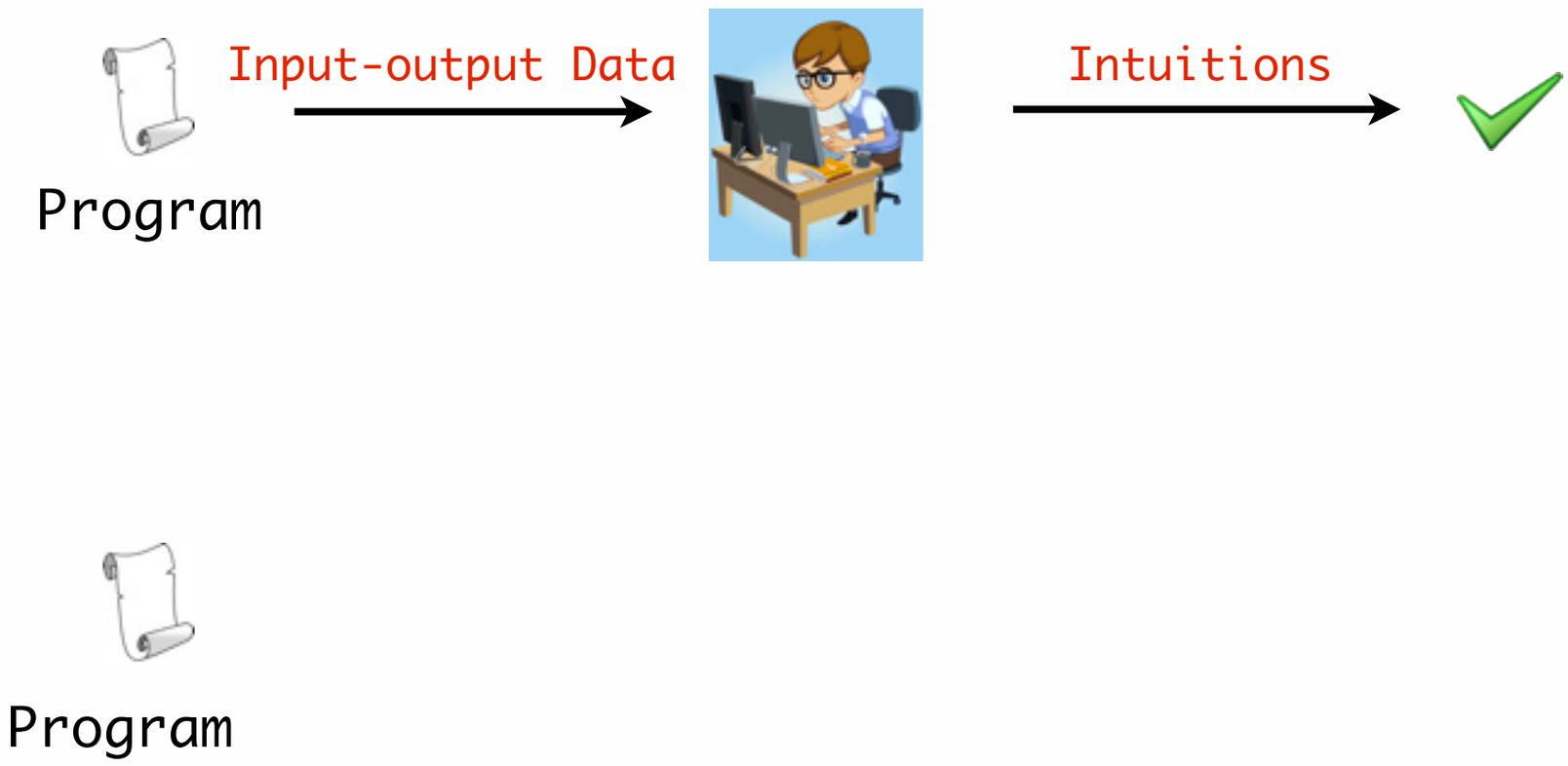
Vision:
A general specification can be discovered from a small number of tests

Goal: Design a learner to automatically discover software specifications

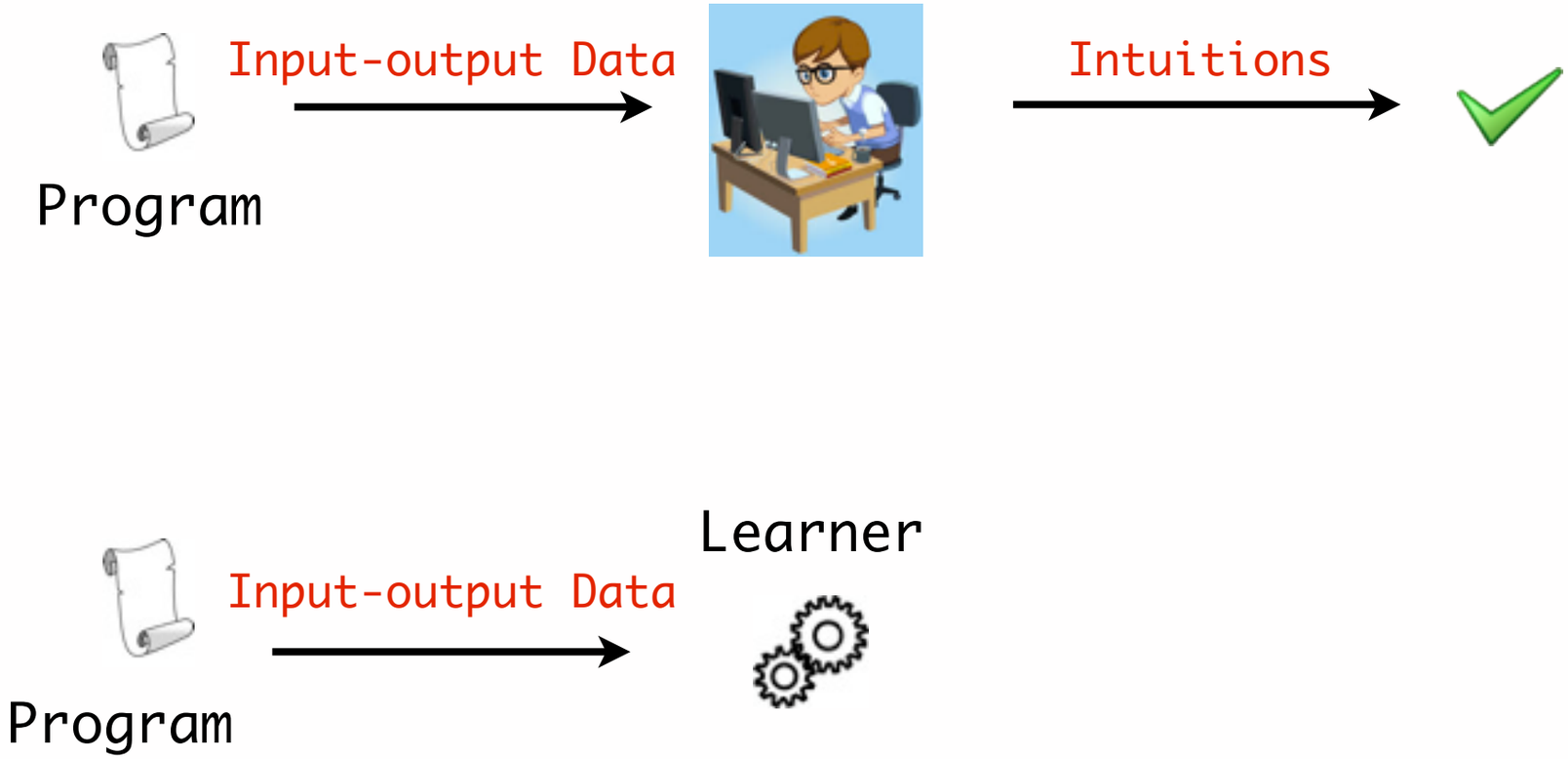
A Learner's Day ...



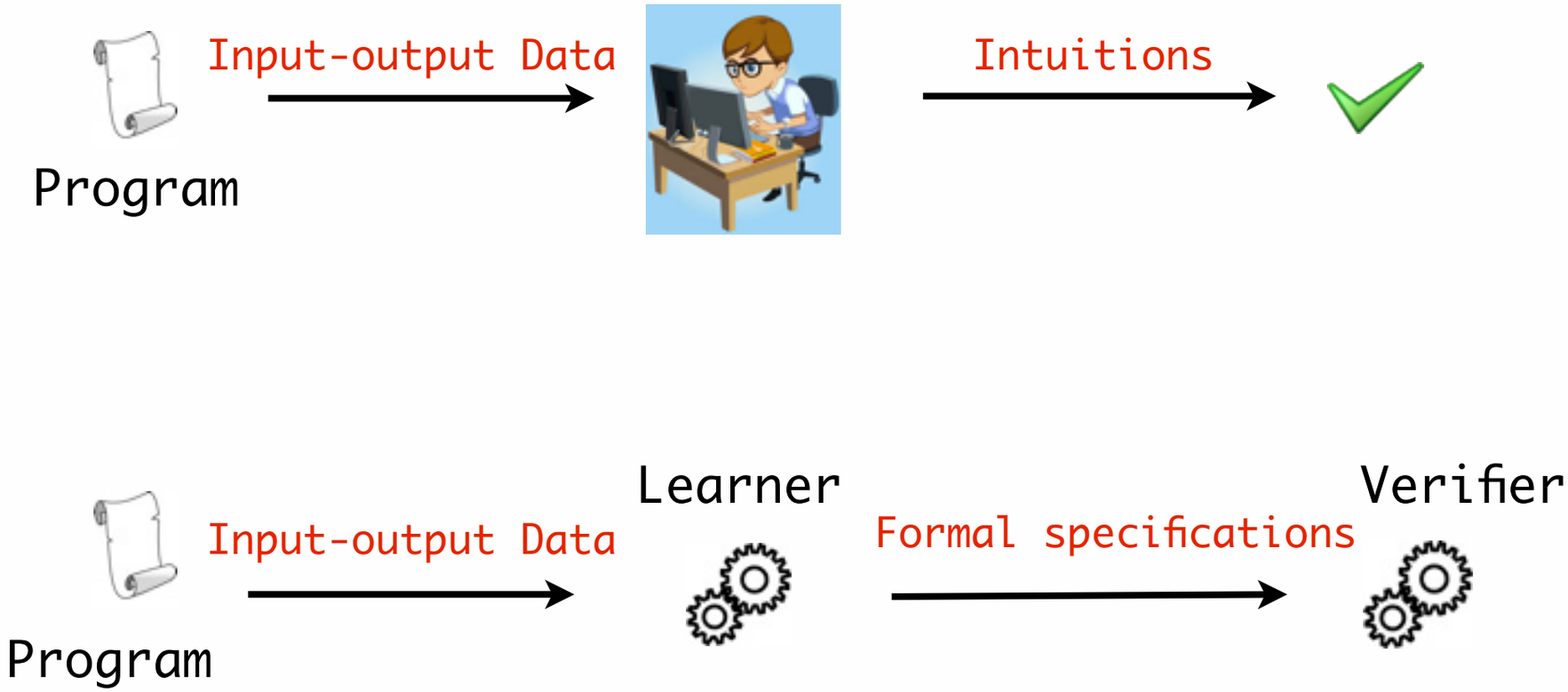
A Learner's Day ...



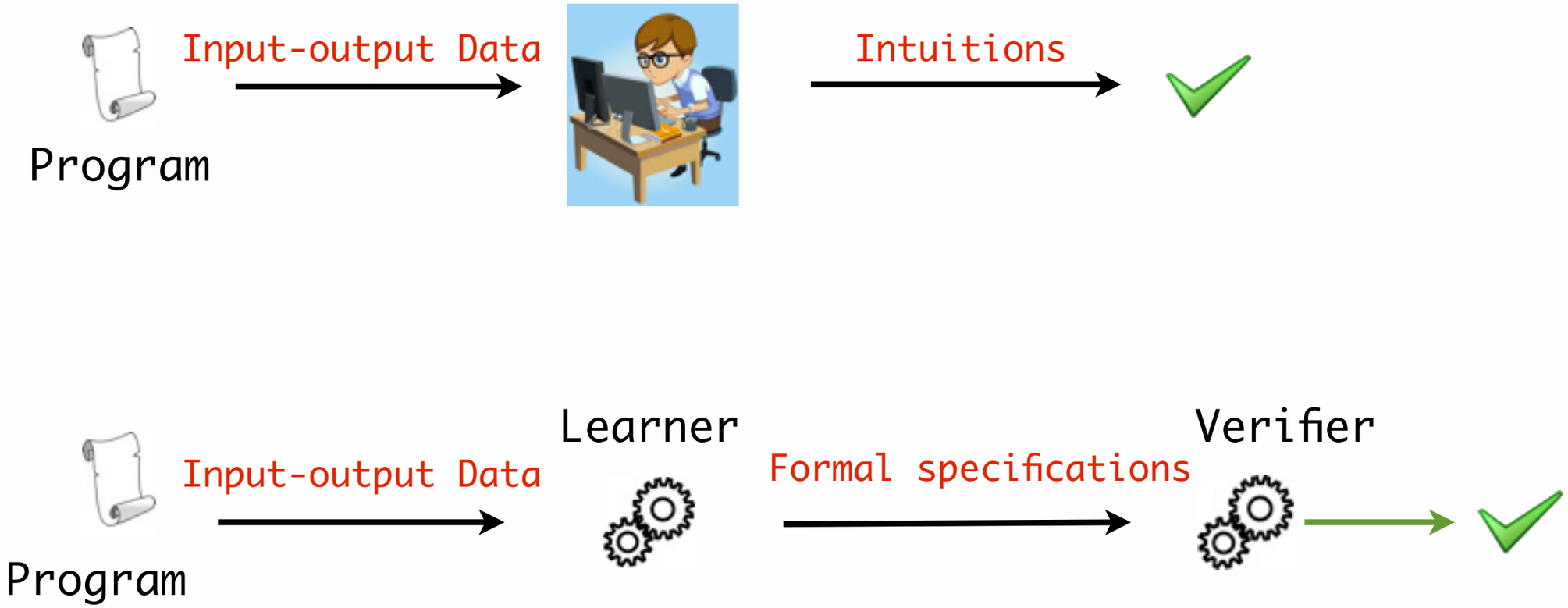
A Learner's Day ...



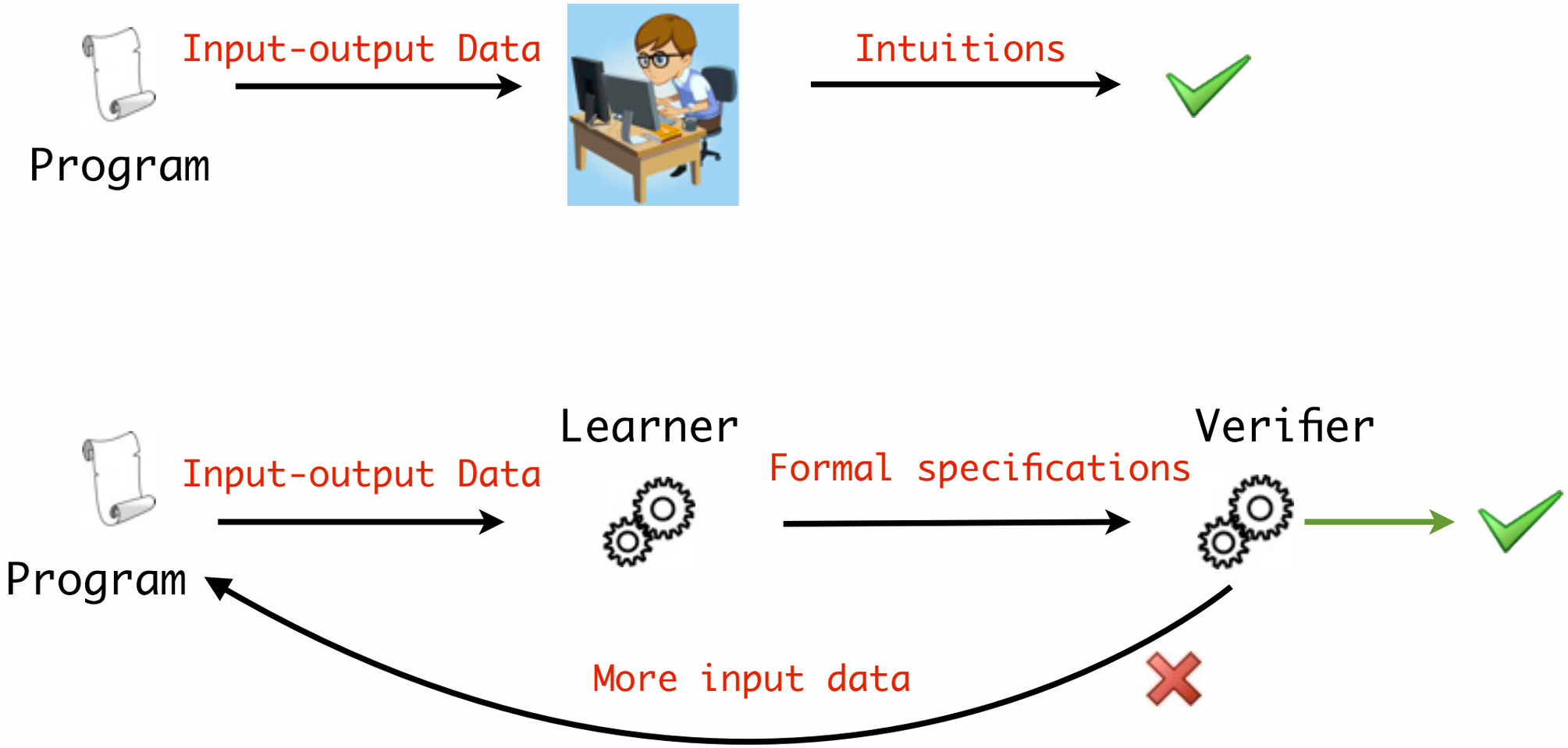
A Learner's Day ...



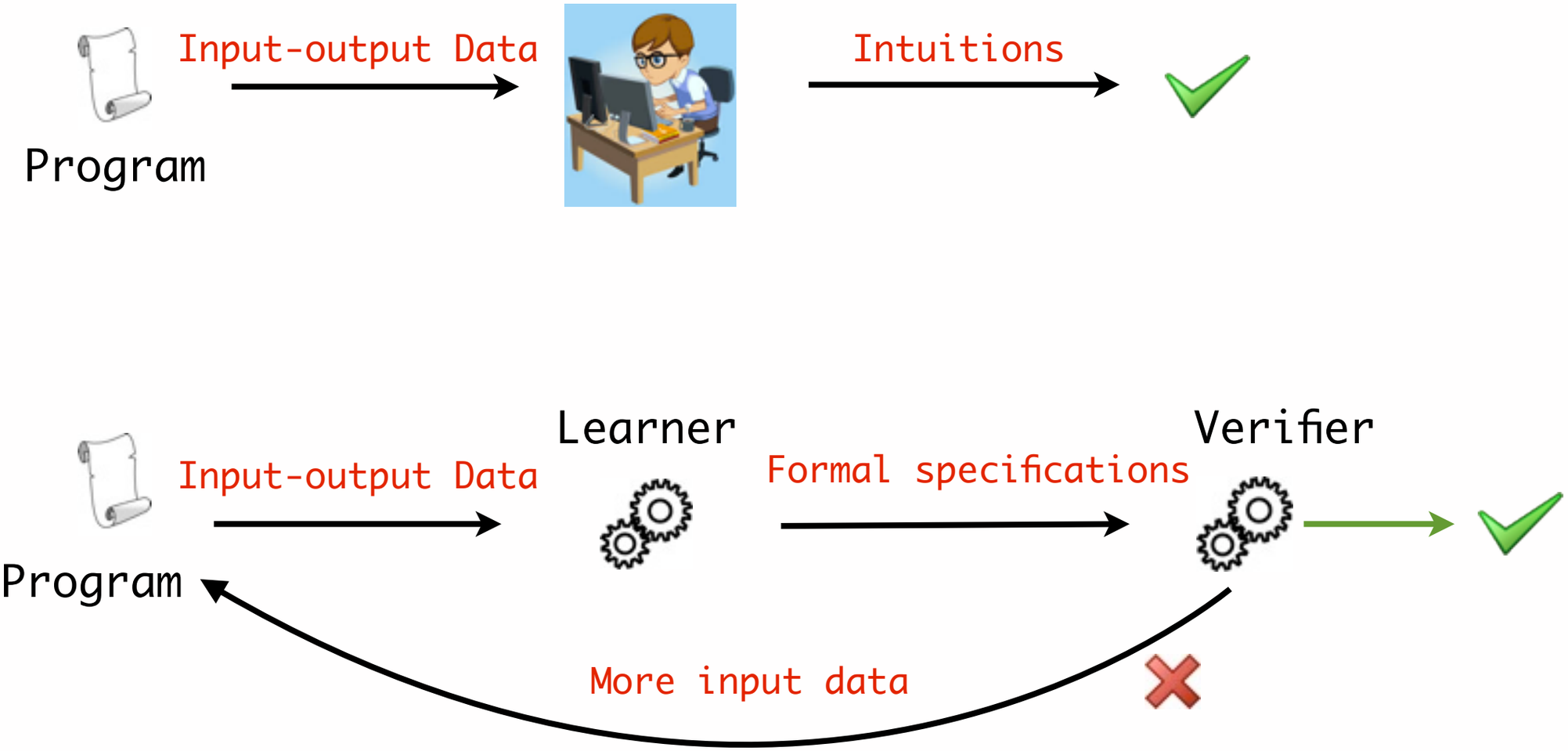
A Learner's Day ...



A Learner's Day ...

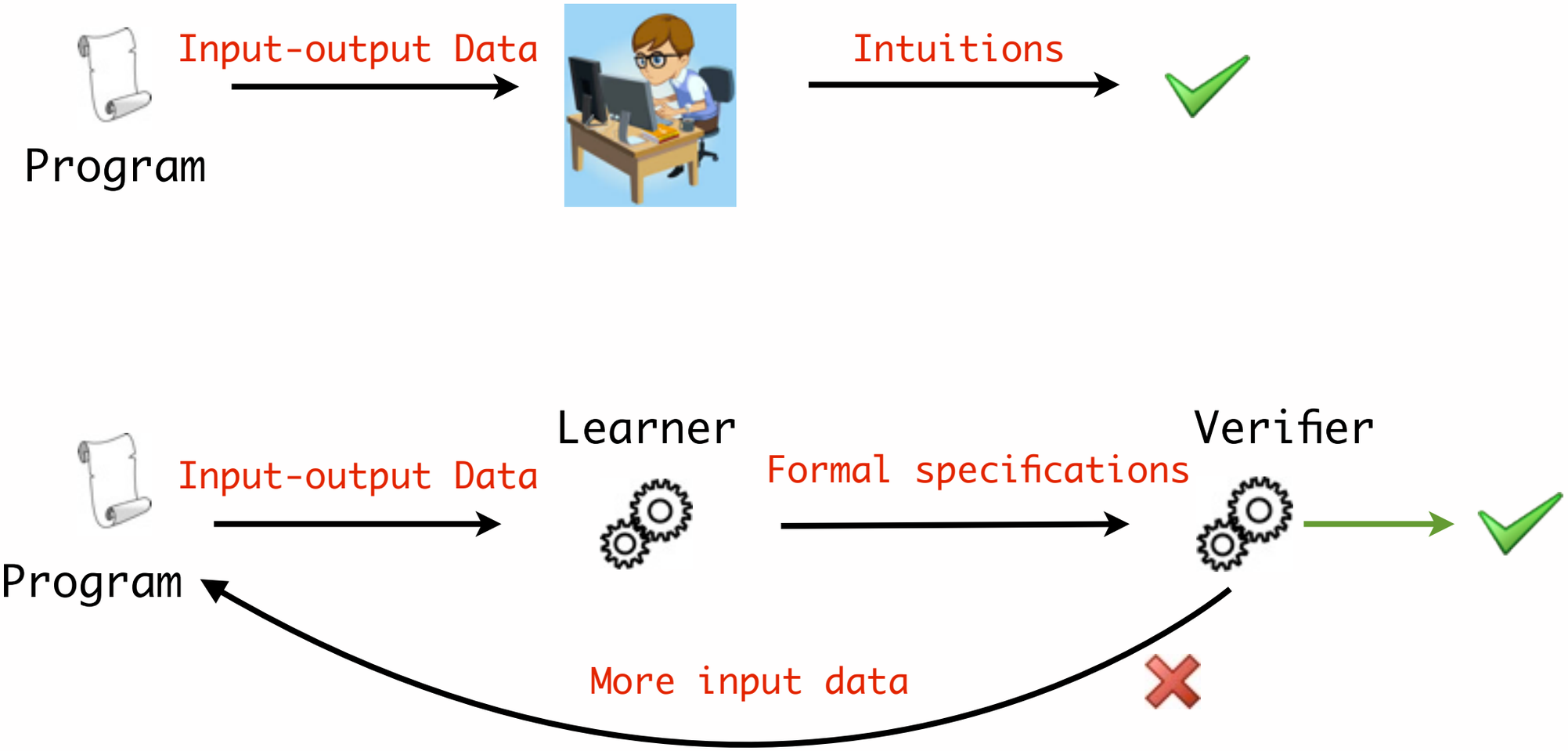


A Learner's Day ...



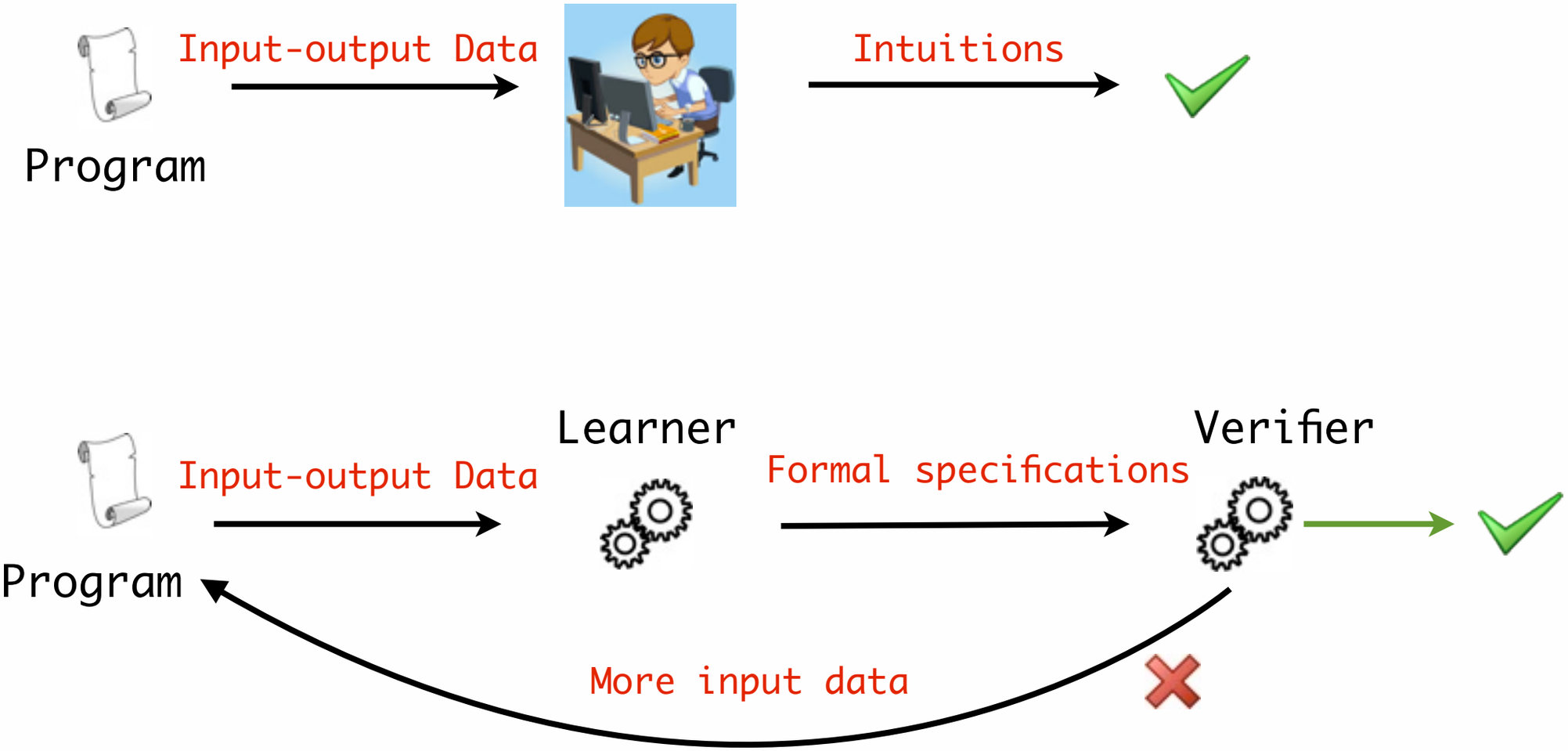
- Automated test inputs generation vs. Manual tests

A Learner's Day ...



- Automated test inputs generation vs. Manual tests
- Formal specifications vs. Intuitions

A Learner's Day ...



- Automated test inputs generation vs. Manual tests
- Formal specifications vs. Intuitions
- Robust verification result vs. Programmers' belief

In this talk ...

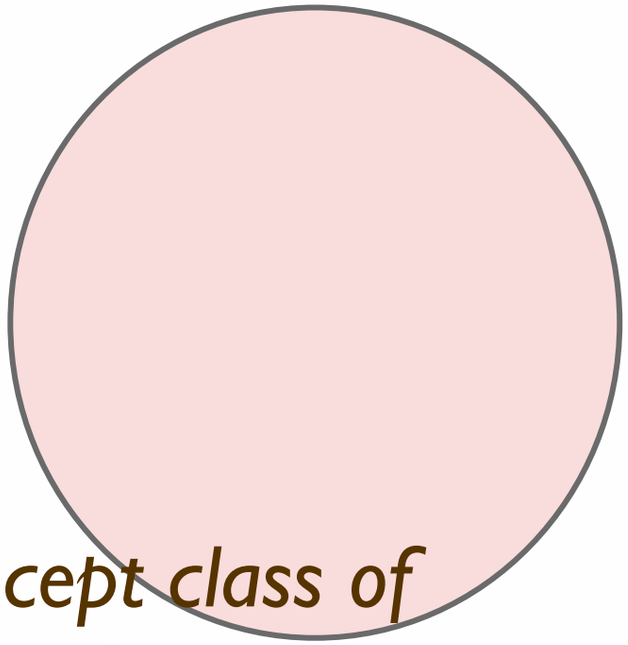
A Convergent Learning Technique for Specification Synthesis

Learning

Learning

Specification Learning ...

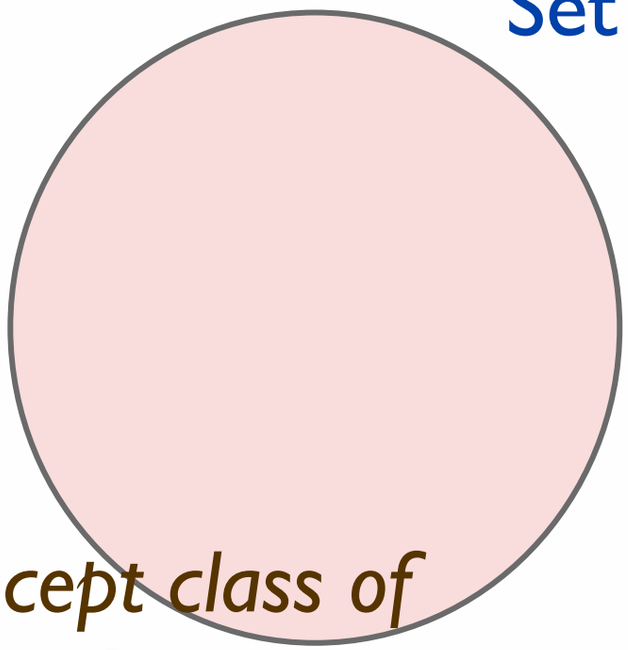
Specification Learning ...



C: Concept class of program P:
Data structures,
Numeric domains,
...

Specification Learning ...

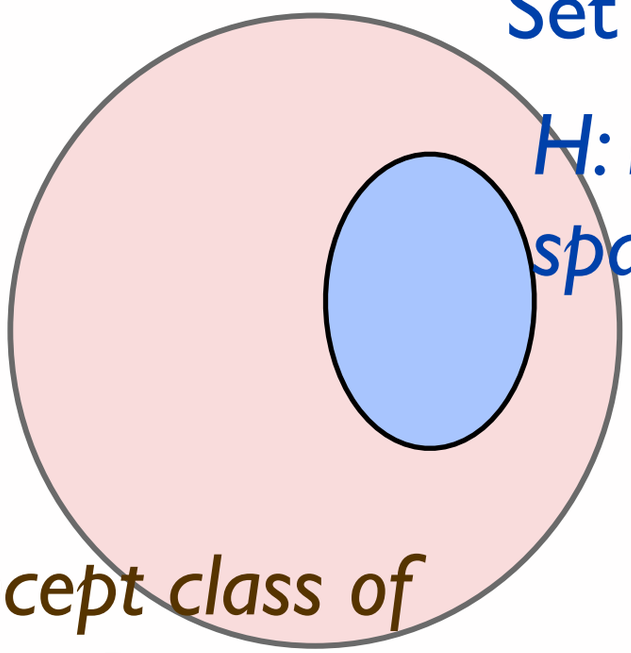
Feature extraction: $P \rightarrow F$
Set of features: F



C: Concept class of program P:
Data structures,
Numeric domains,
...

Specification Learning ...

Feature extraction: *P* → *F*
Set of features: *F*

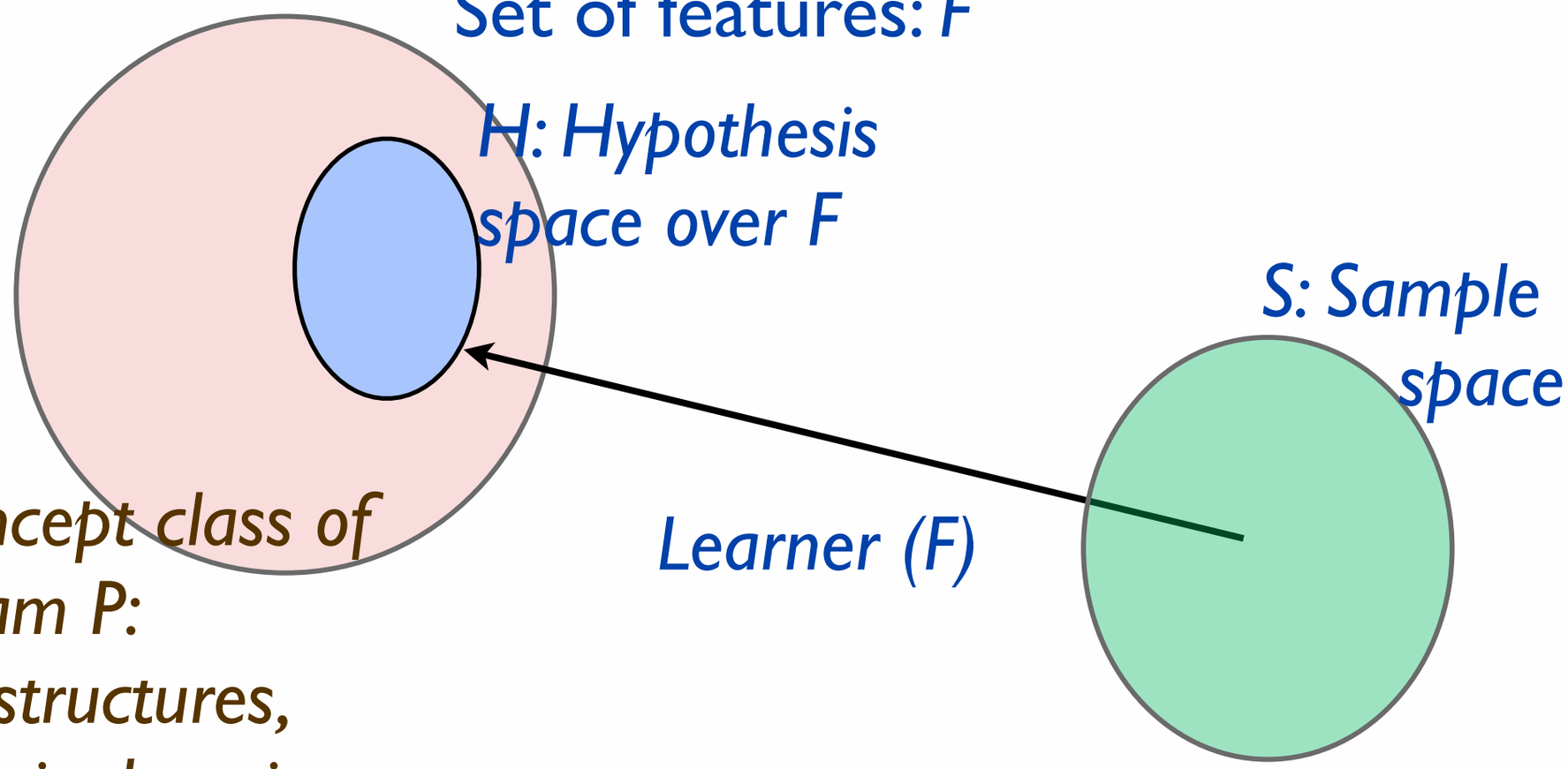


H: Hypothesis space over F

C: Concept class of program P:
Data structures,
Numeric domains,
...

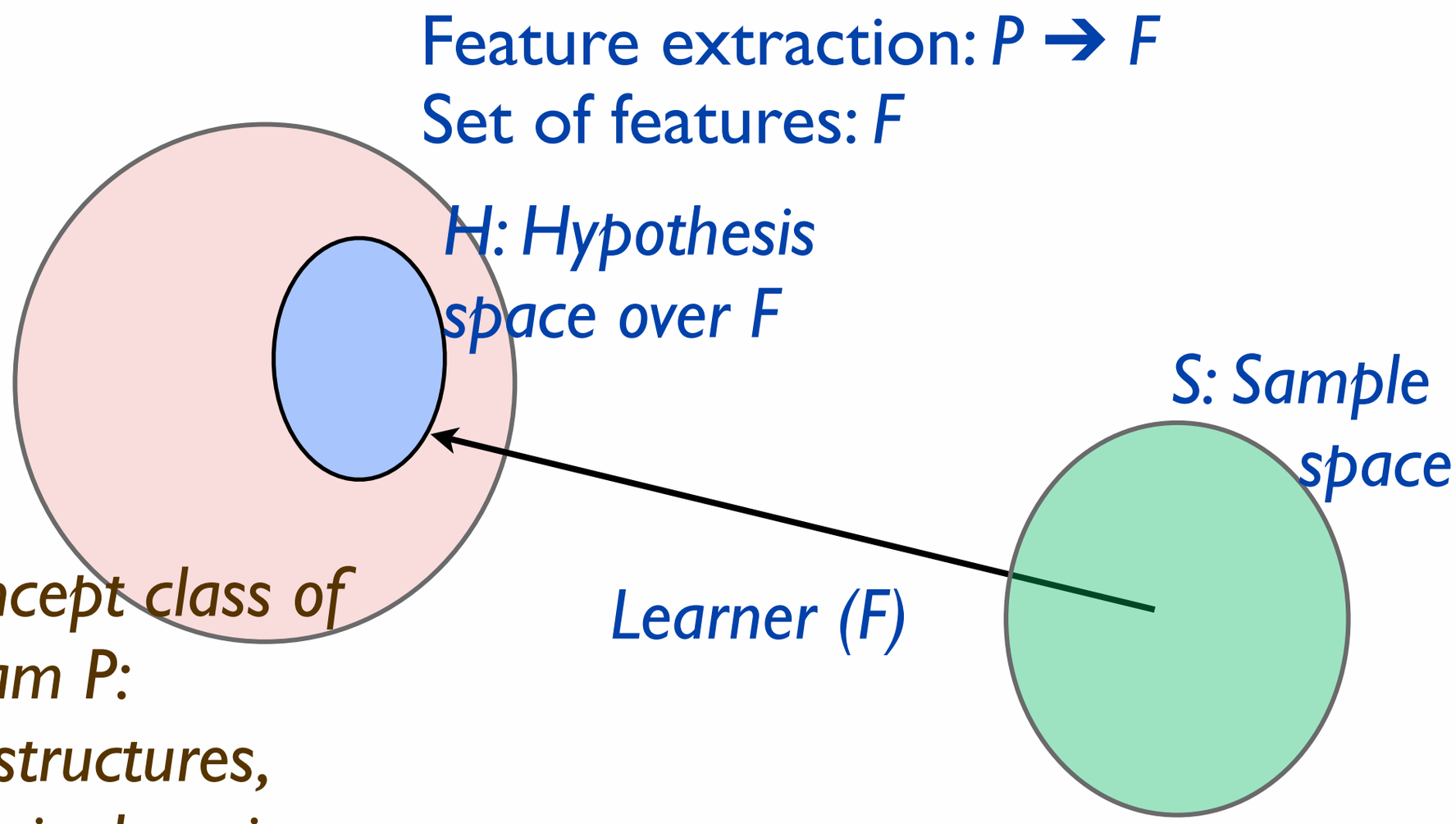
Specification Learning ...

Feature extraction: *P* → *F*
Set of features: *F*



C: Concept class of program P:
Data structures,
Numeric domains,
...

Specification Learning ...



C: Concept class of program P:
Data structures,
Numeric domains,
...

- Assumptions:
  - Inductive Functional Data Structures

Features of Data Structures ...

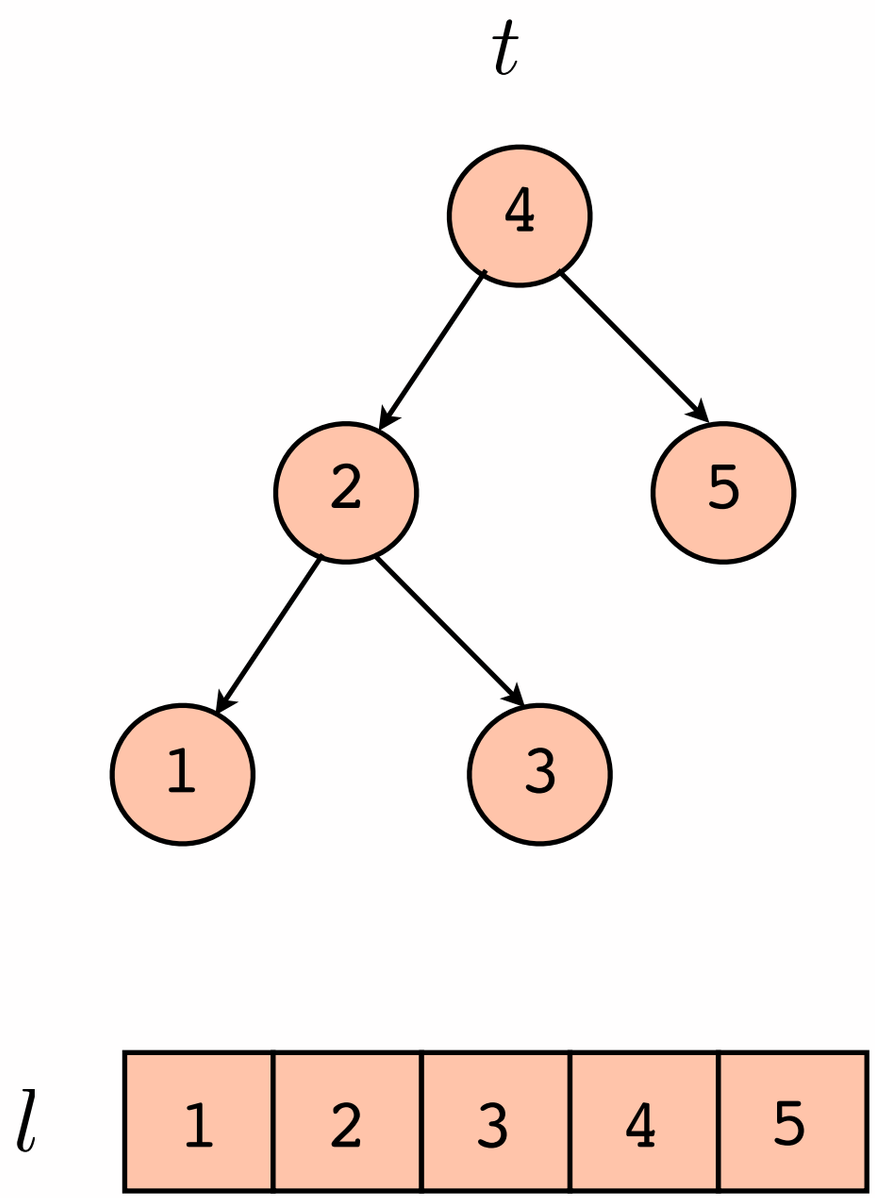
```
// elements: 'a tree -> 'a list
let elements t = flat [] t
```

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

Features of Data Structures ...

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

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



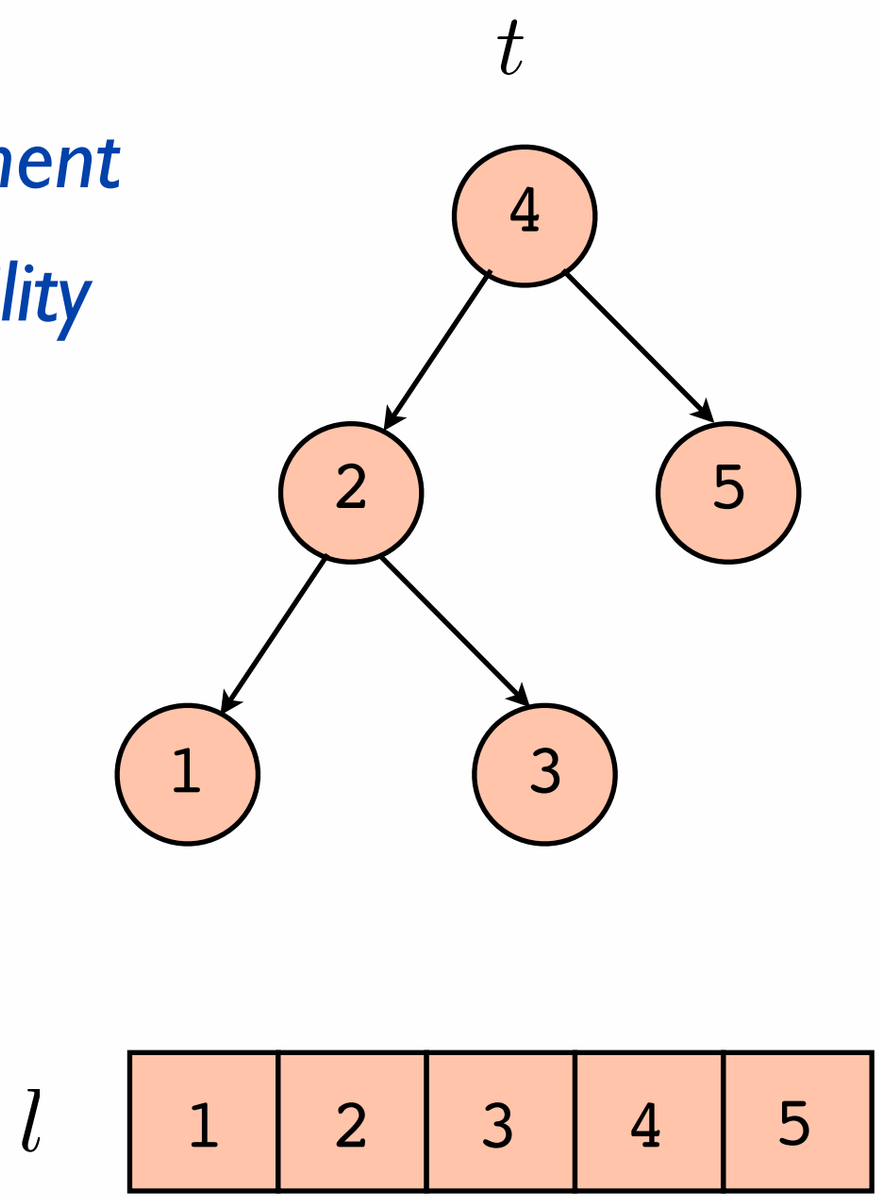
Features of Data Structures ...

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

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

$t \dashrightarrow u$ *Containment*

$t : u \swarrow v$ *Reachability*



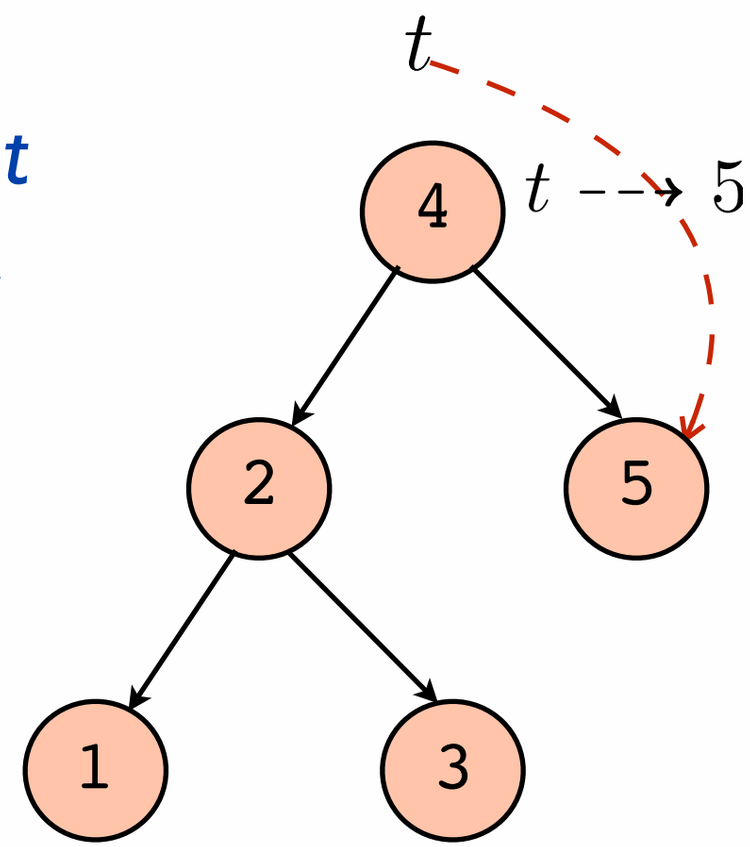
Features of Data Structures ...

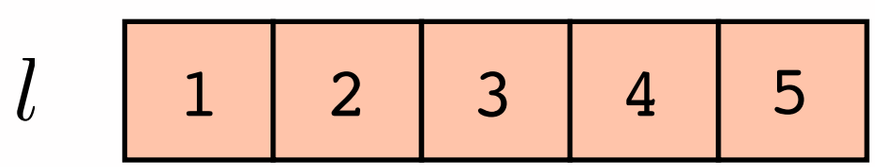
```
// elements: 'a tree -> 'a list
let elements t = flat [] t
```

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

$t \dashrightarrow u$ *Containment*

$t : u \swarrow v$ *Reachability*





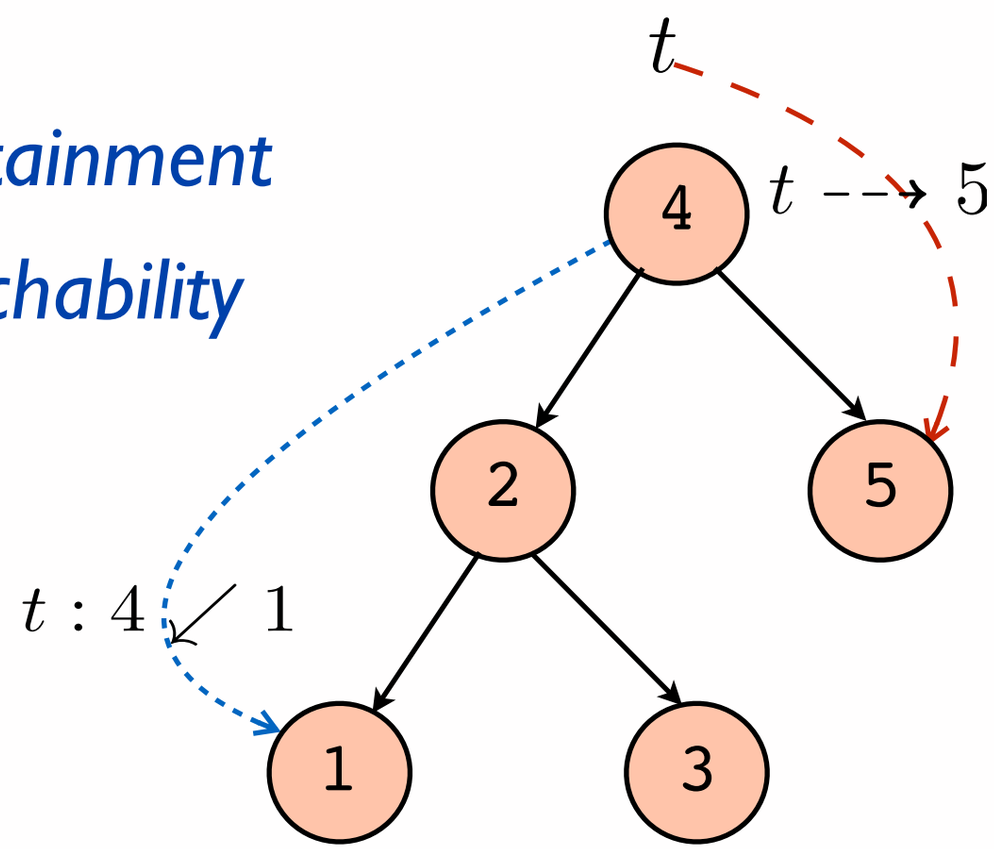
Features of Data Structures ...

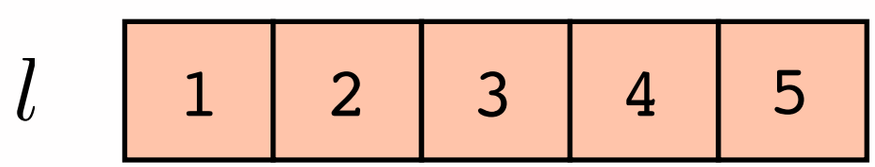
```
// elements: 'a tree -> 'a list
let elements t = flat [] t
```

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

$t \dashrightarrow u$ *Containment*

$t : u \swarrow v$ *Reachability*





Features of Data Structures ...

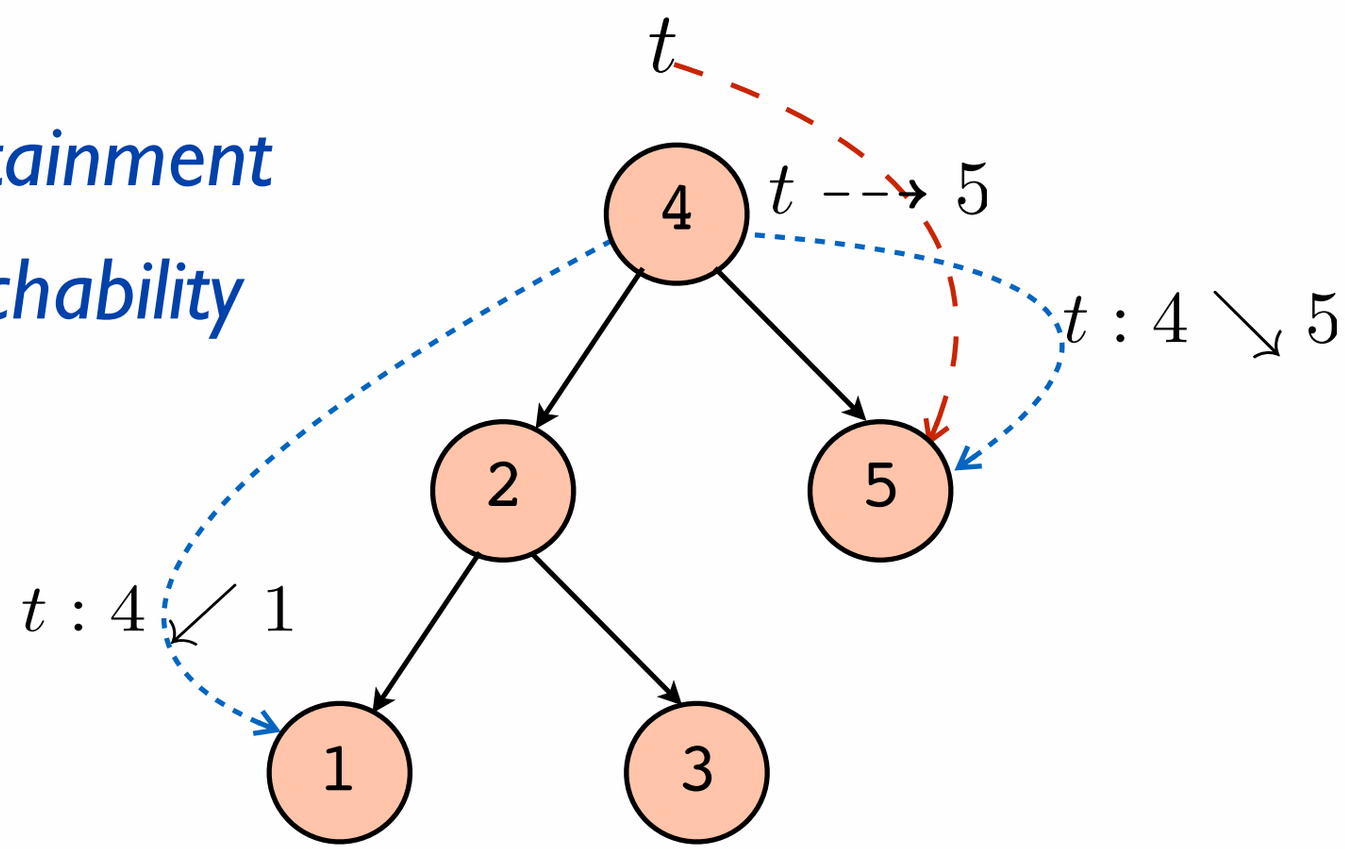
```
// elements: 'a tree -> 'a list
let elements t = flat [] t
```

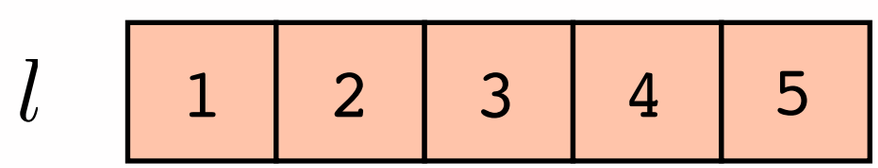
$$l = \text{elements } t$$

$t \dashrightarrow u$ *Containment*

$t : u \swarrow v$ *Reachability*

$t : u \searrow v$





Features of Data Structures ...

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

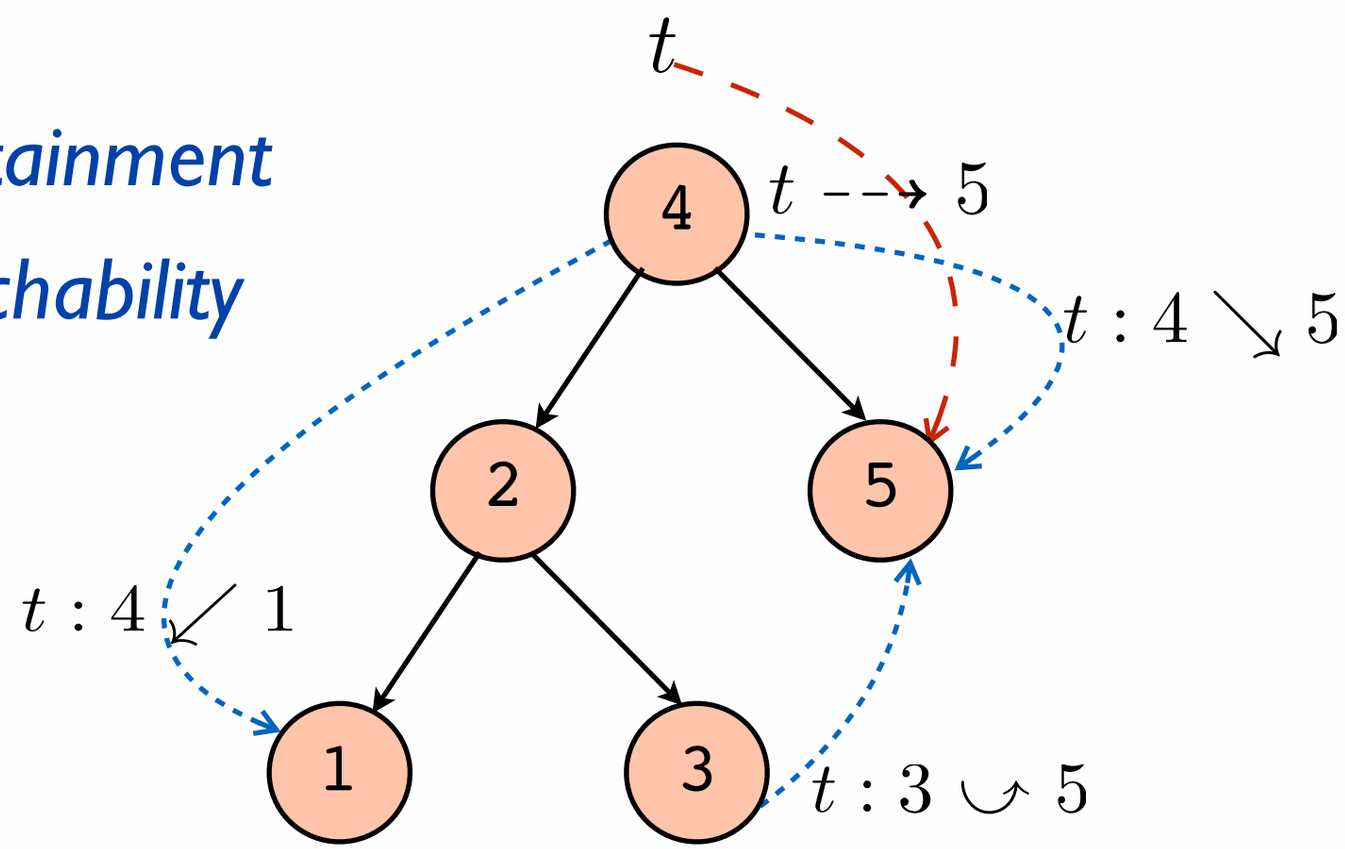
$$l = \text{elements } t$$

$t \dashrightarrow u$ *Containment*

$t : u \swarrow v$ *Reachability*

$t : u \searrow v$

$t : u \circlearrowright v$



l

1	2	3	4	5

Features of Data Structures ...

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

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

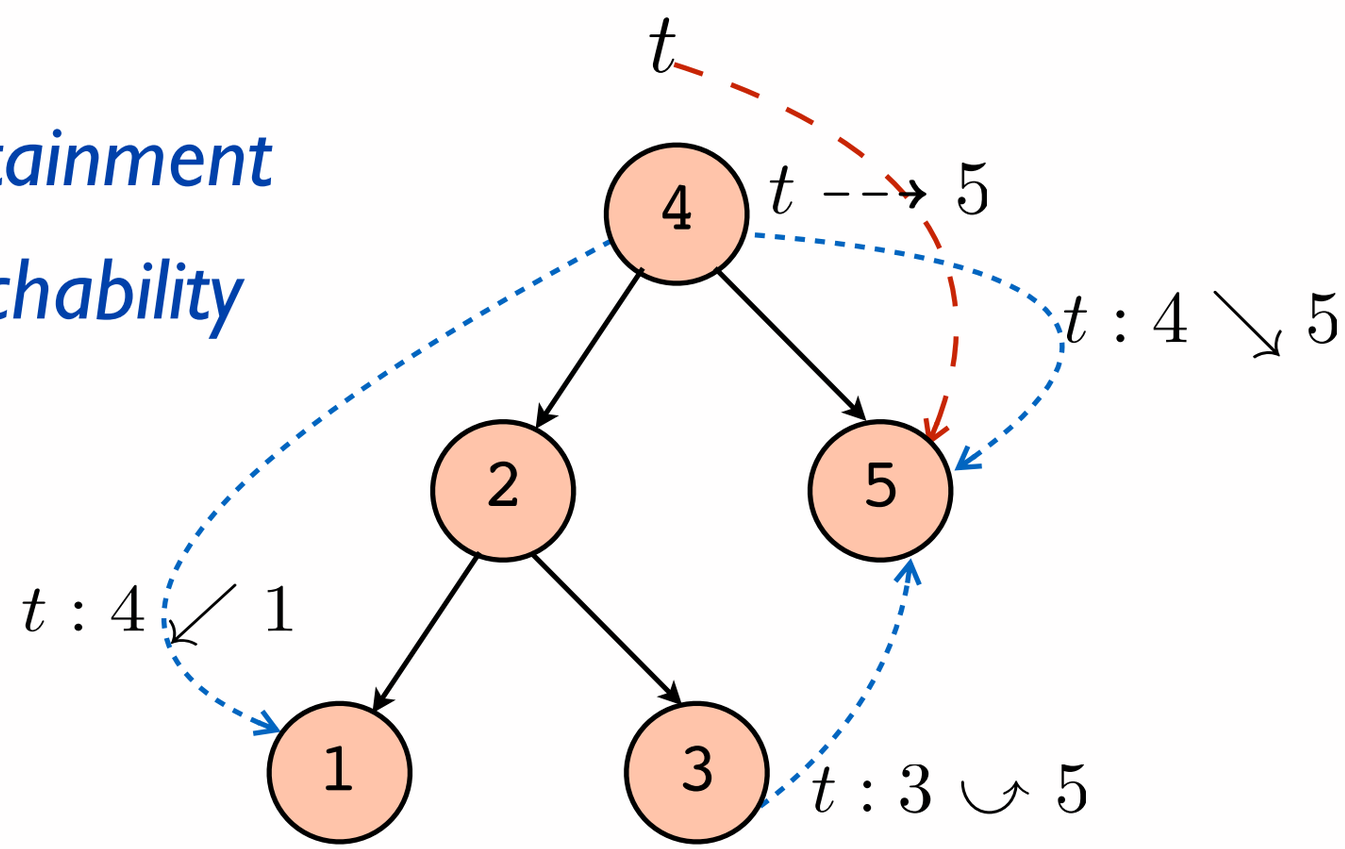
$t \dashrightarrow u$ *Containment*

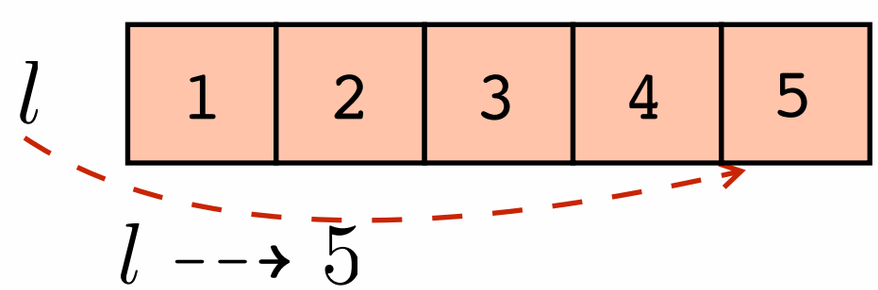
$t : u \swarrow v$ *Reachability*

$t : u \searrow v$

$t : u \circlearrowright v$

$l \dashrightarrow u$





Features of Data Structures ...

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

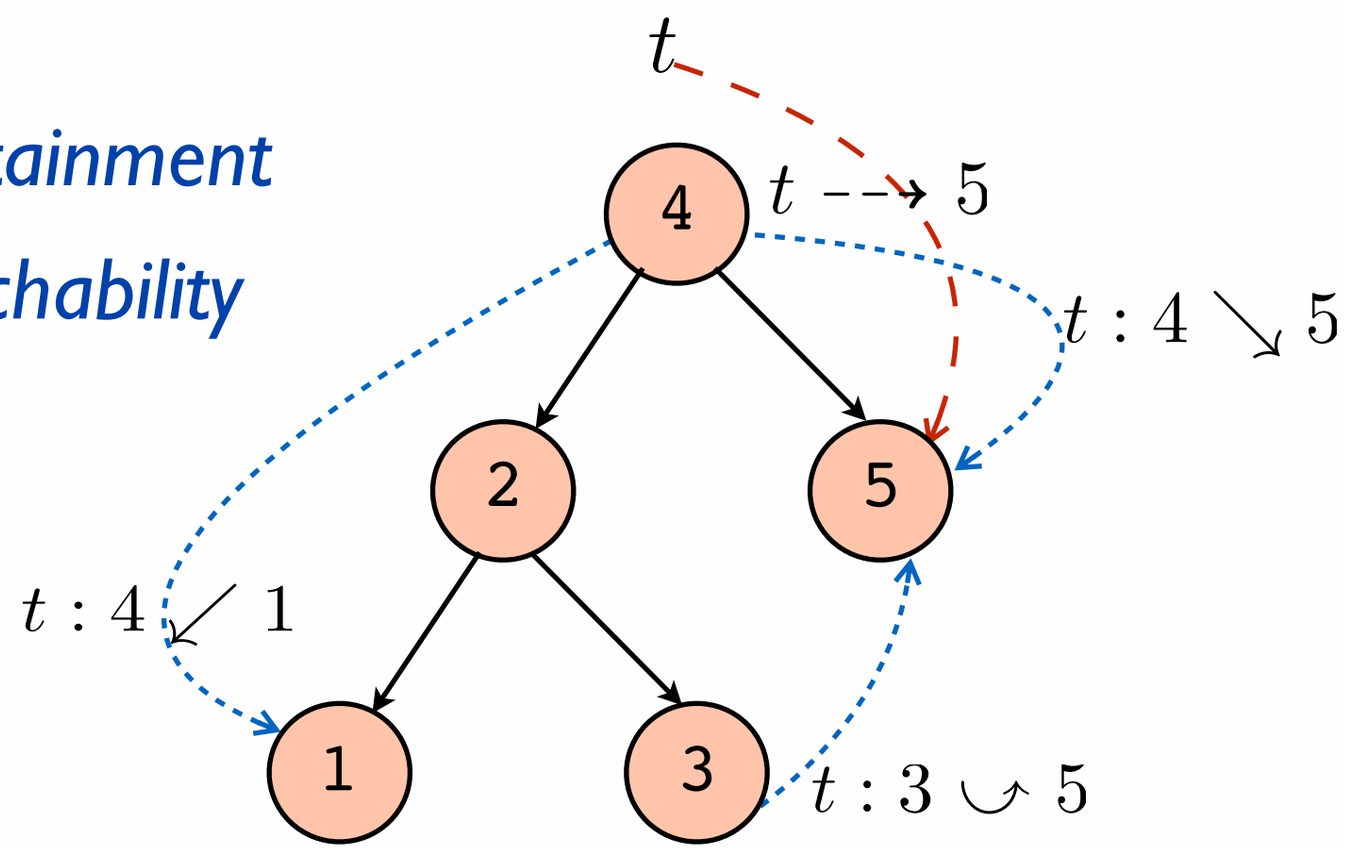
$$l = \text{elements } t$$

$t \dashrightarrow u$ *Containment*

$t : u \swarrow v$ *Reachability*

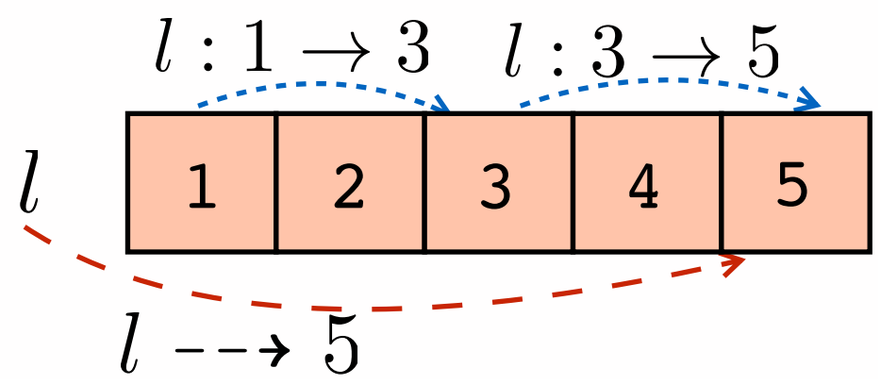
$t : u \searrow v$

$t : u \circlearrowright v$



$l \dashrightarrow u$

$l : u \rightarrow v$



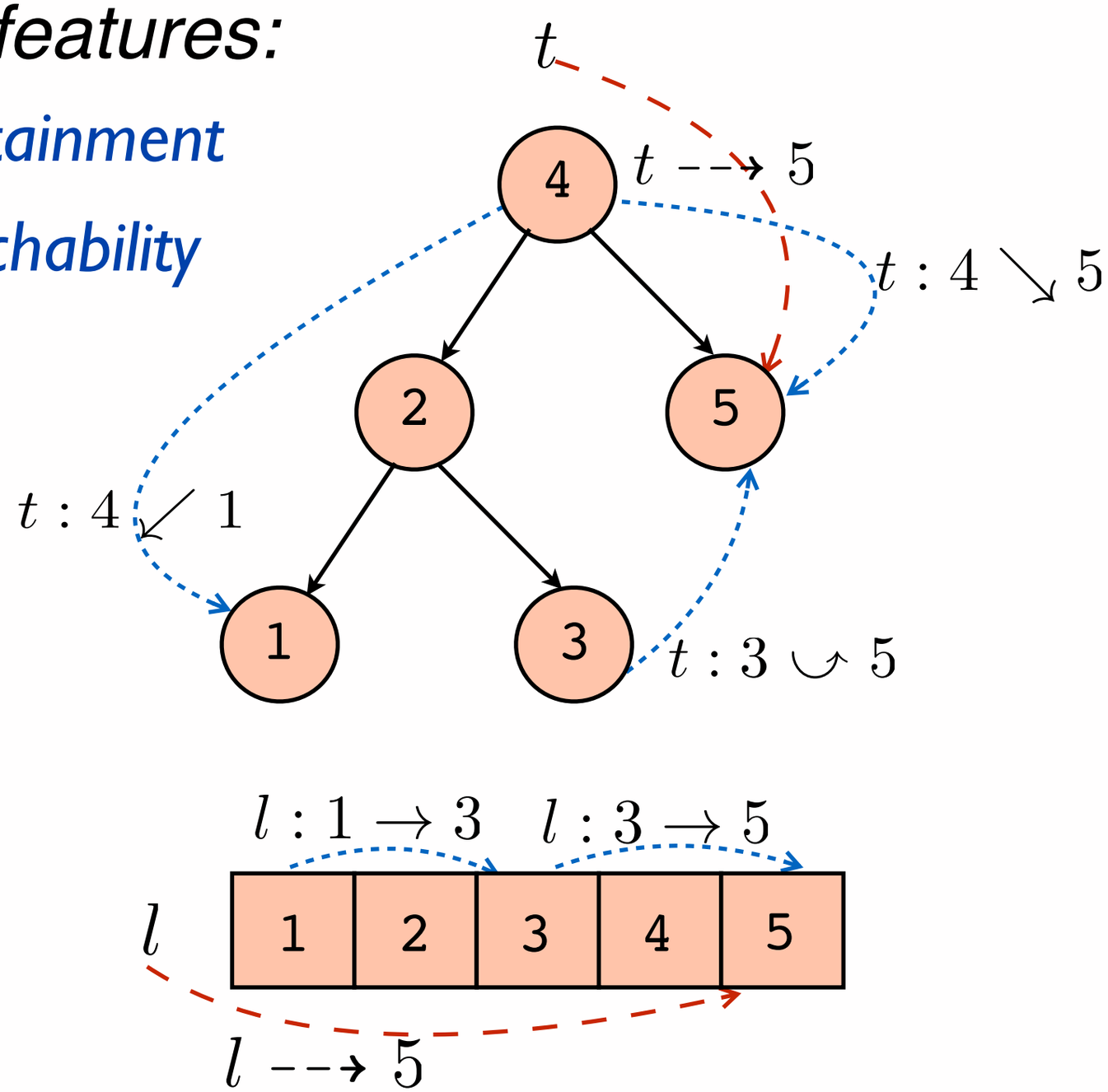
Features of Data Structures ...

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

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

Hypothesis Domain *over data structure features:*

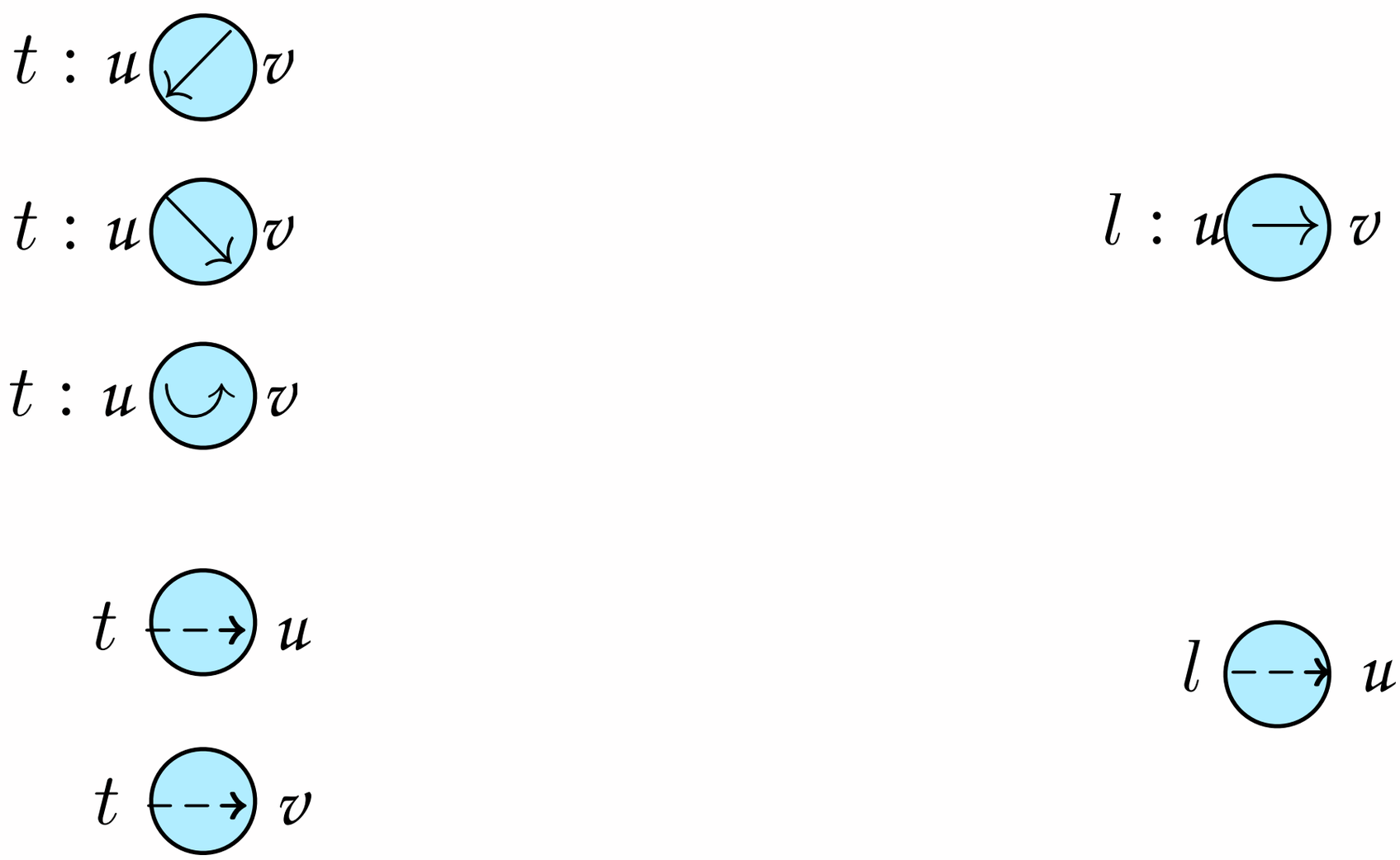
$t \dashrightarrow u$	*Containment*
$t : u \swarrow v$	*Reachability*
$t : u \searrow v$	
$t : u \circlearrowright v$	
$l \dashrightarrow u$	
$l : u \rightarrow v$	



From features to specifications ...

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

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

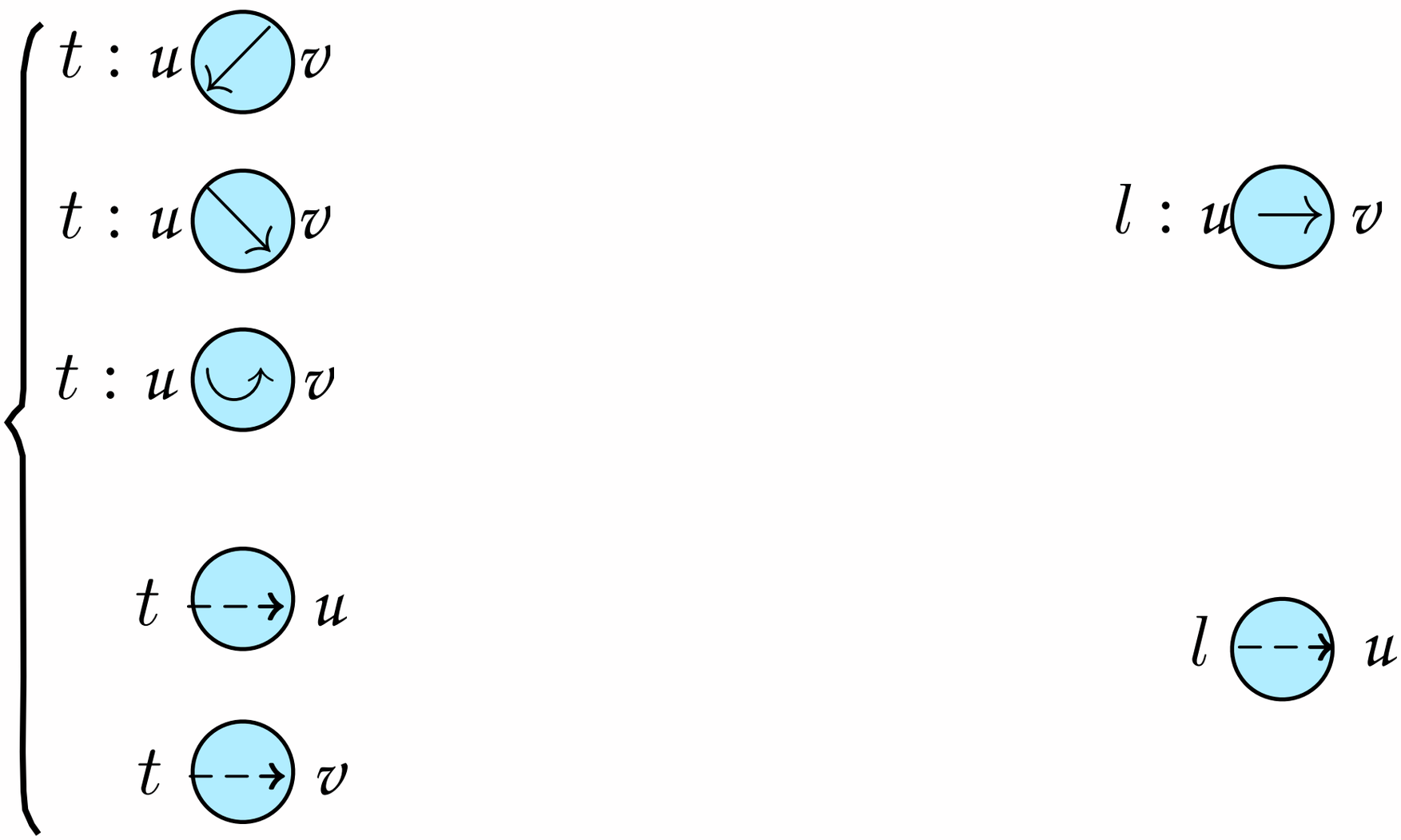


From features to specifications ...

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

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

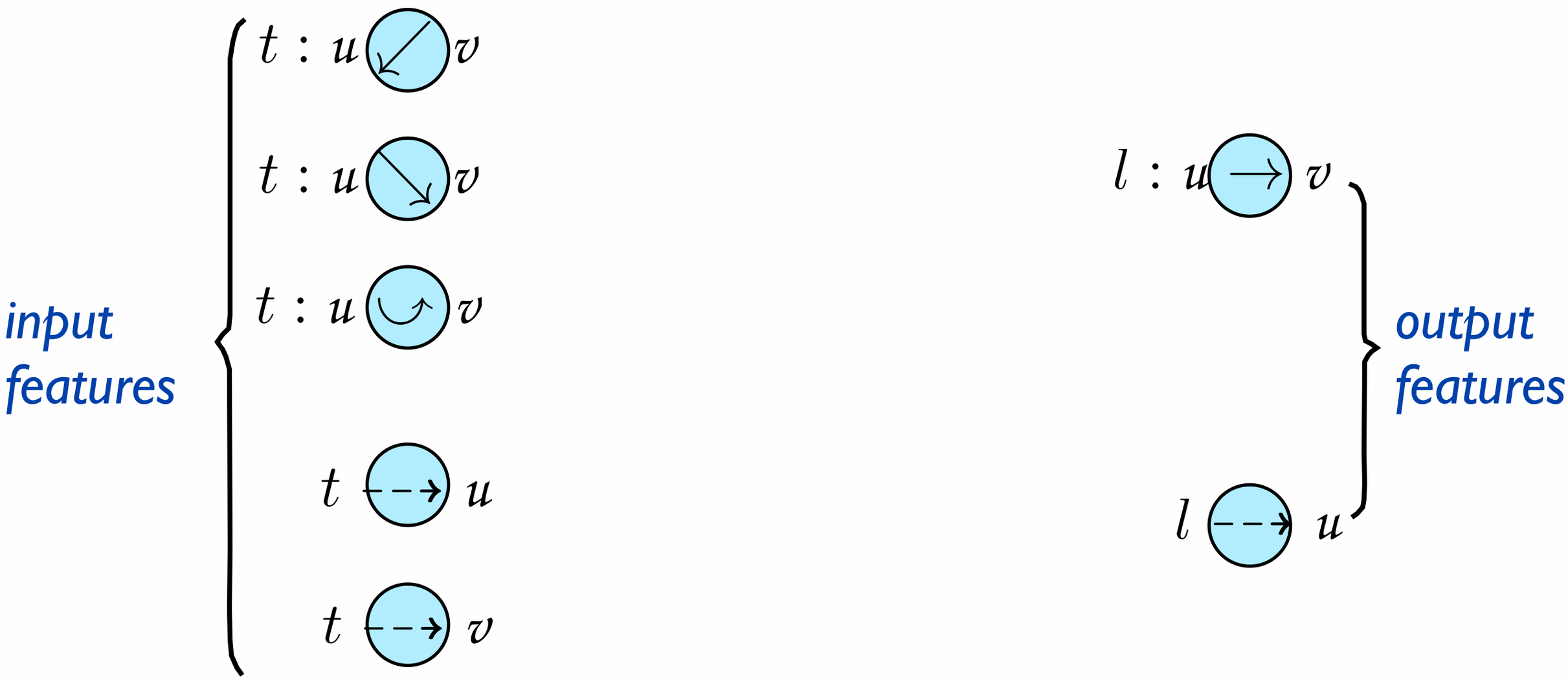
input features



From features to specifications ...

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

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

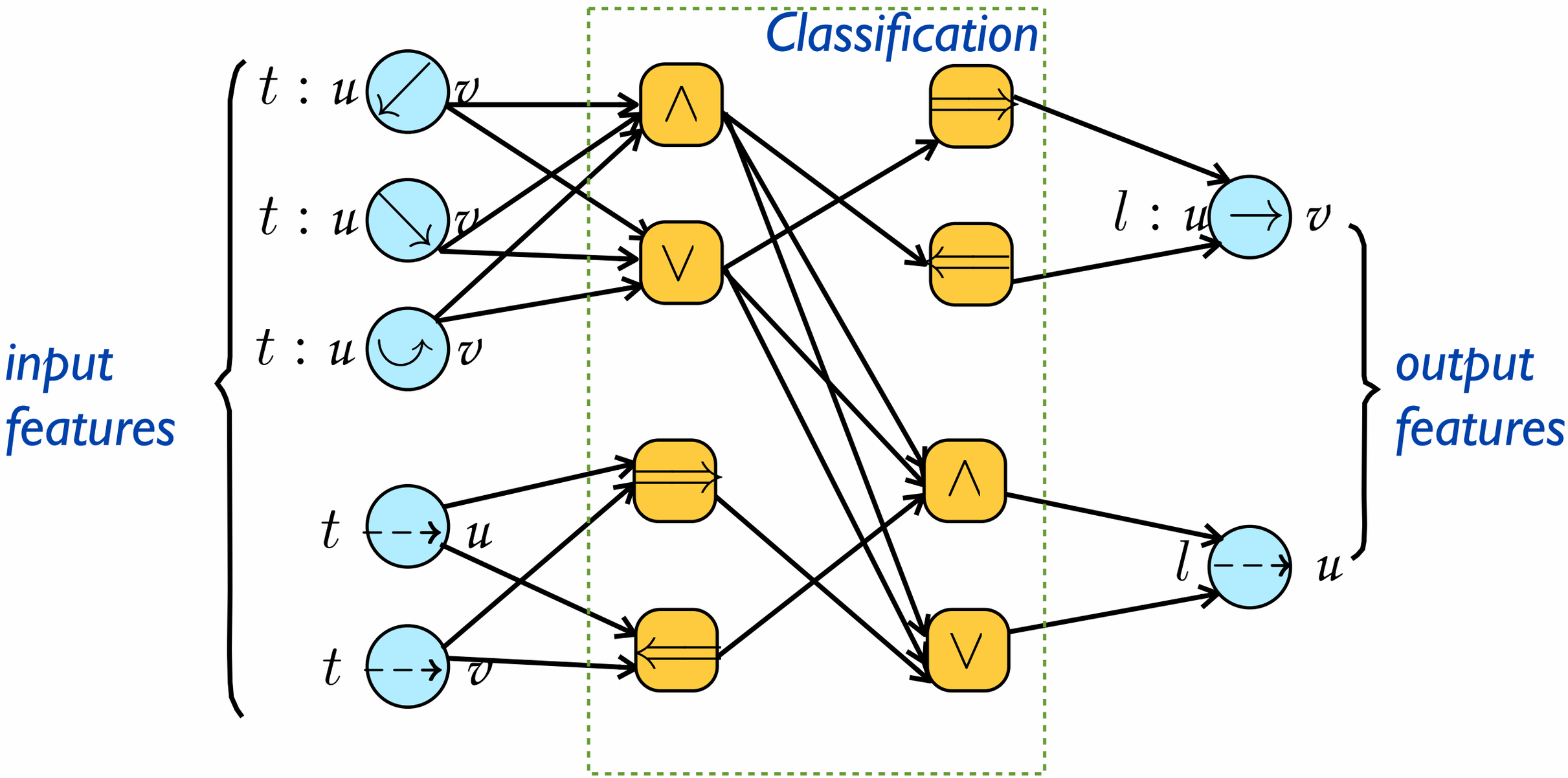


From features to specifications ...

```
// 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 ...

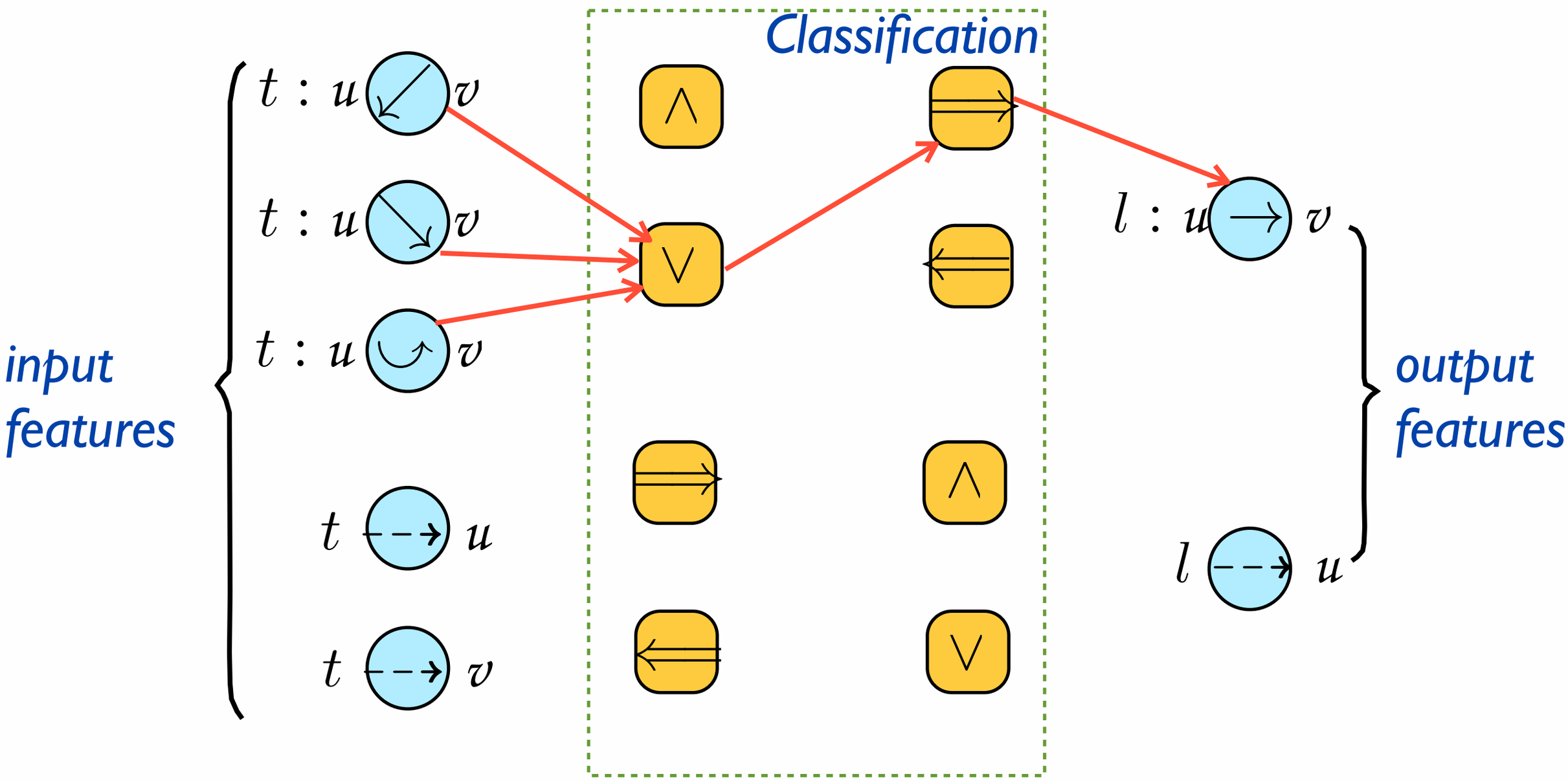


From features to specifications ...

```
// 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 ...

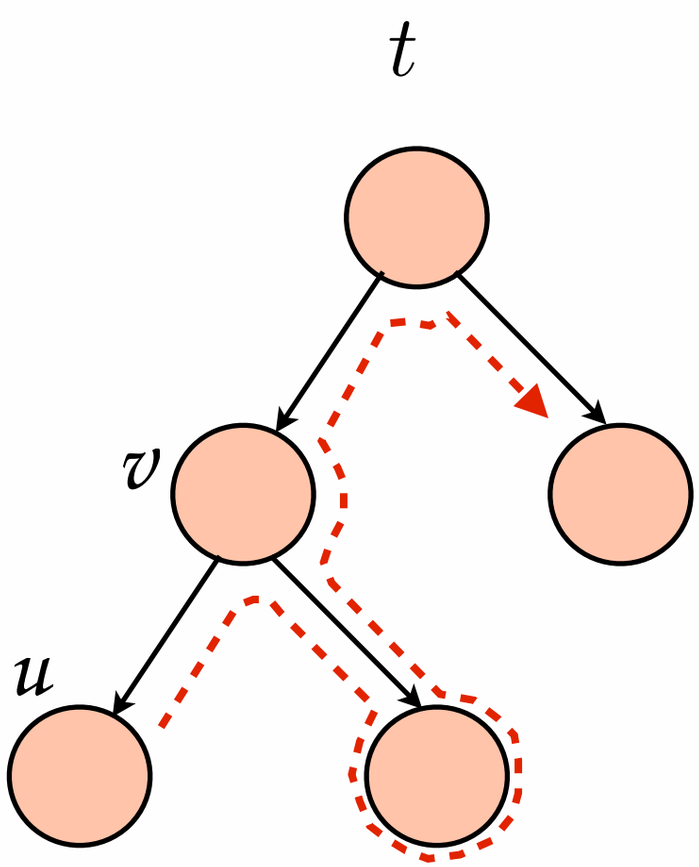
```
// specification:
// in-order of t ≡ forward-order of l
 l:list = elements (t:tree)
```

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

Specifications of Data Structures ...

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

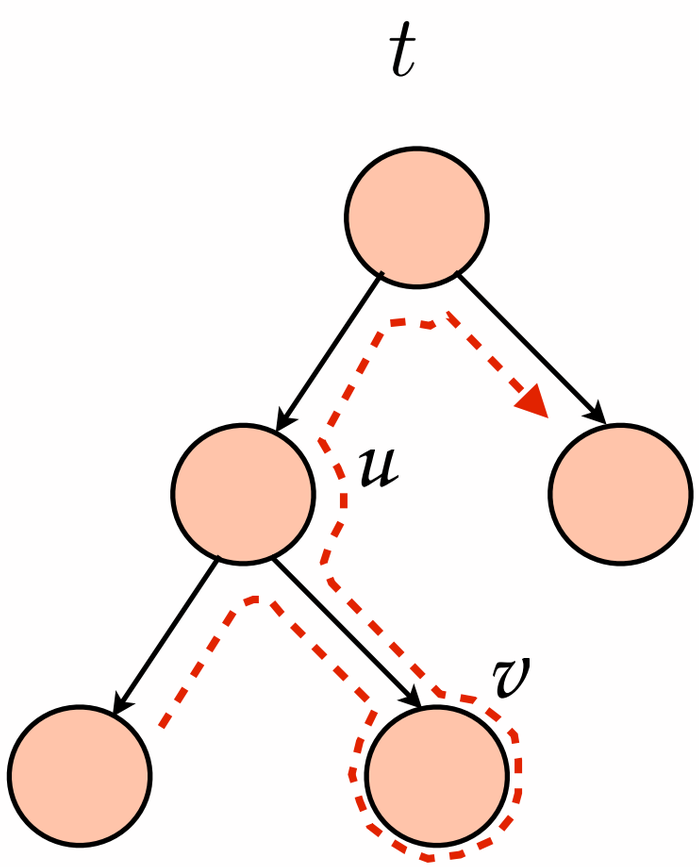
$$\left(\forall u\ v,\ \begin{pmatrix} t : v \swarrow u \vee \\ t : u \searrow v \vee \\ t : u \circlearrowright v \end{pmatrix} \iff l : u \to v\right)$$



Specifications of Data Structures ...

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

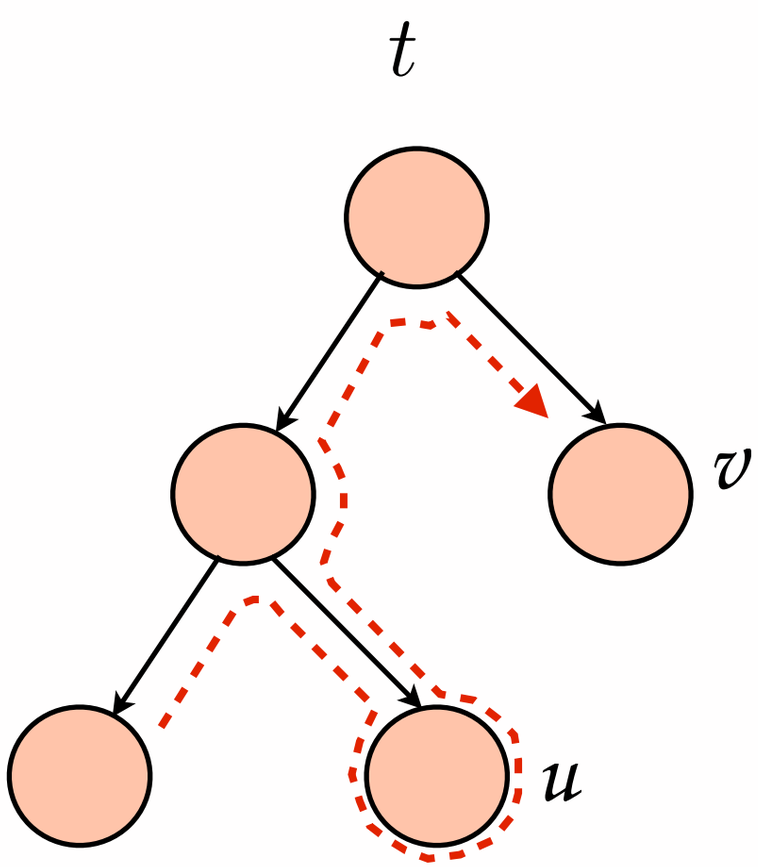
$$\left(\forall u\ v,\ \begin{pmatrix} t : v \swarrow u \vee \\ t : u \searrow v \vee \\ t : u \circlearrowright v \end{pmatrix} \iff l : u \to v\right)$$



Specifications of Data Structures ...

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

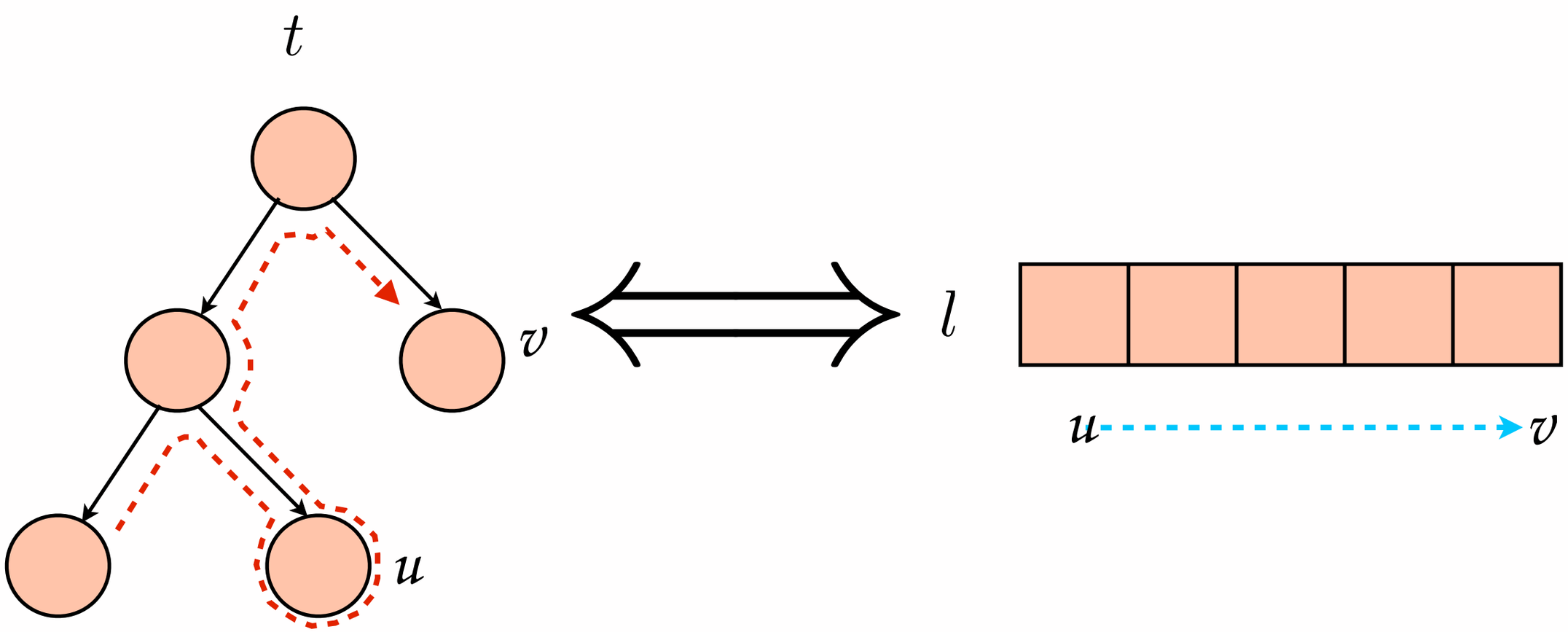
$$\left(\forall u\ v,\ \begin{pmatrix} t : v \swarrow u \ \vee \\ t : u \searrow v \ \vee \\ t : u \circlearrowright v \end{pmatrix} \iff l : u \to v\right)$$



Specifications of Data Structures ...

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

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



Feature Extraction ...

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

Feature Extraction ...

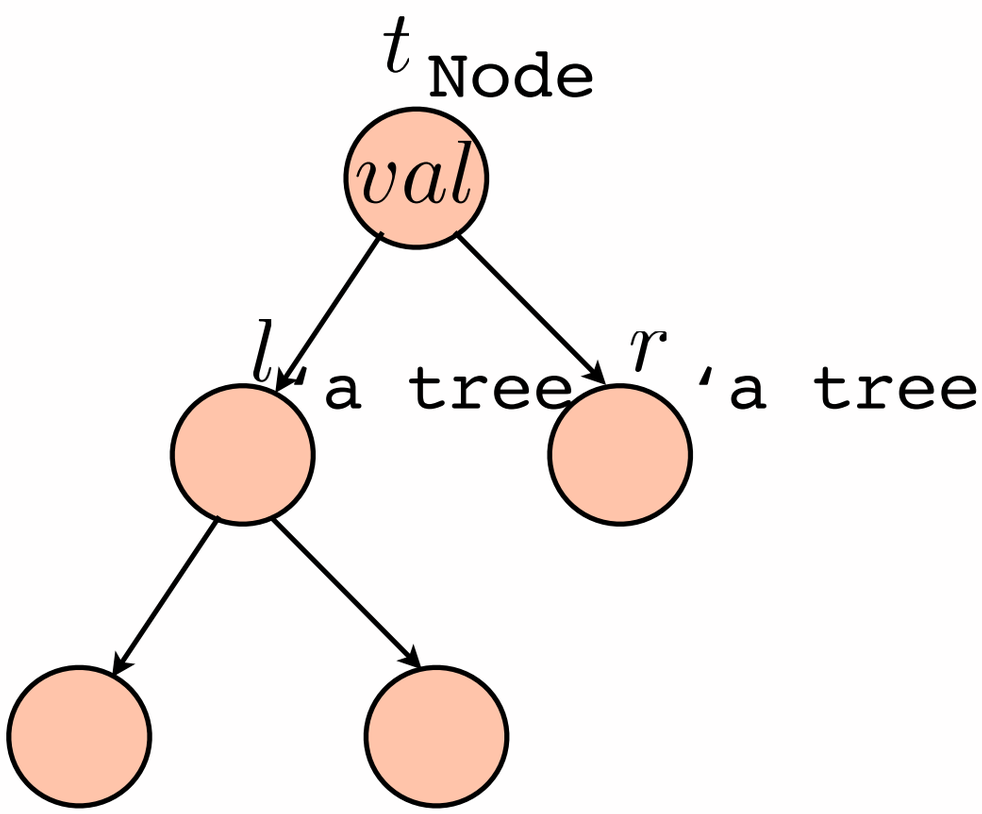
```
type ‘a tree =
  | Leaf
  | Node ‘a * ‘a tree * ‘a tree
```

Static code analysis

Feature Extraction ...

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

Static code analysis

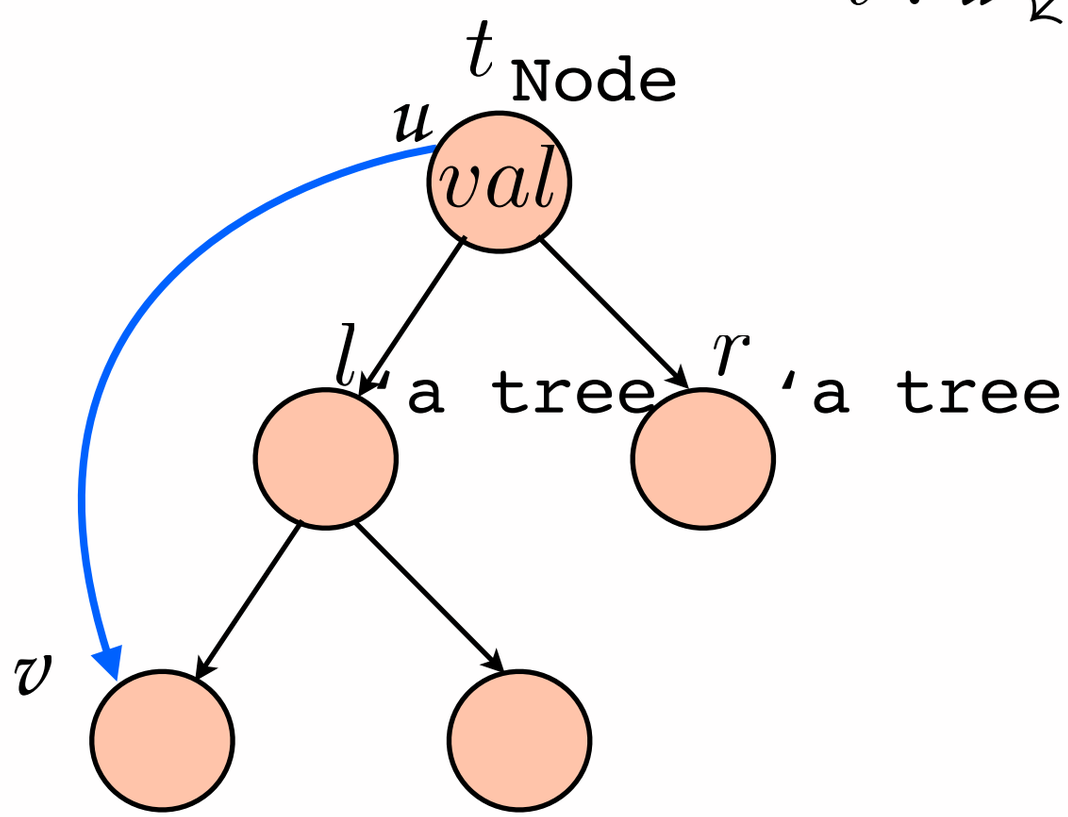


Feature Extraction ...

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

Static code analysis

$t : u \swarrow v$

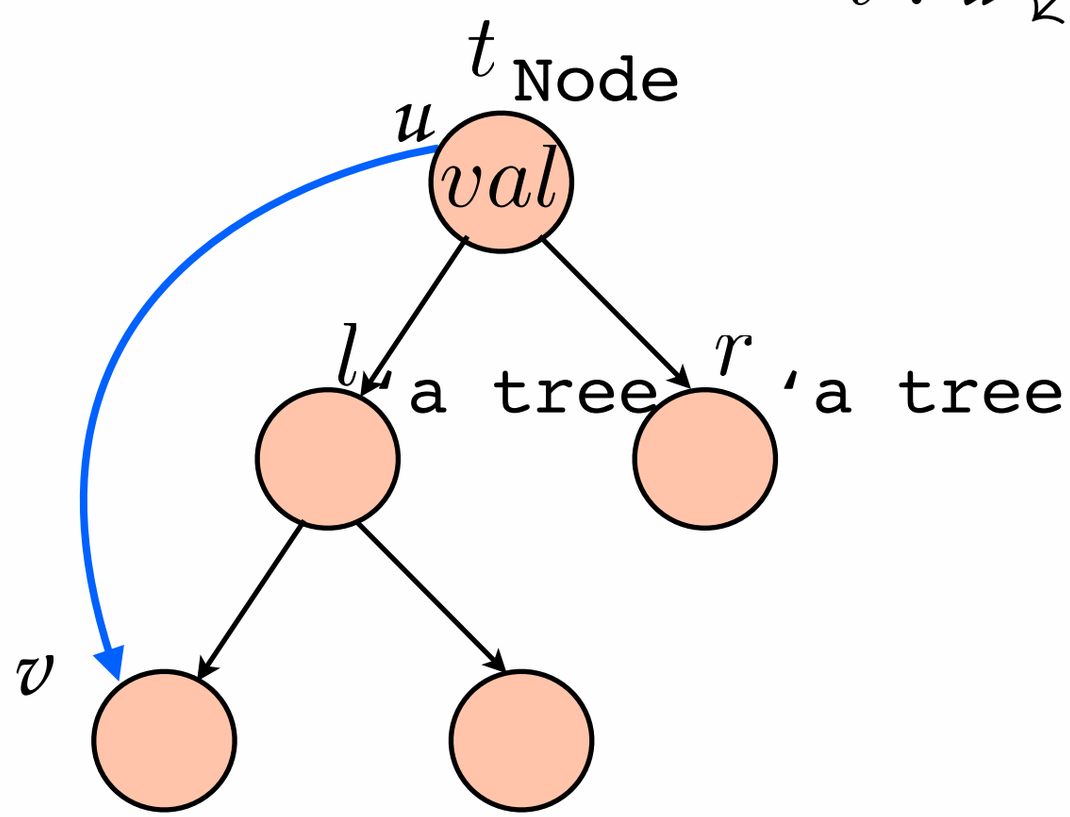


Feature Extraction ...

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

Static code analysis

$t : u \swarrow v$



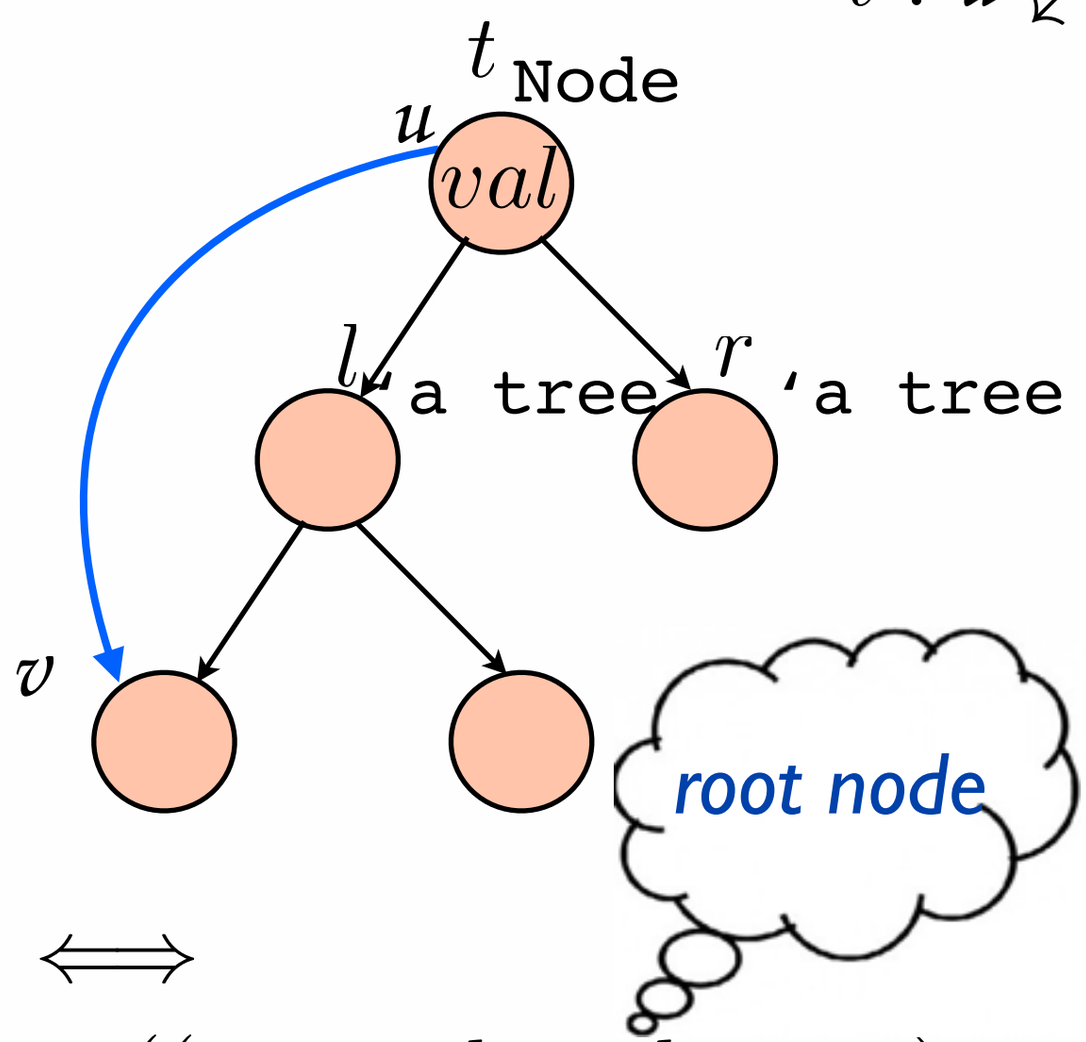
$$t : u \swarrow v \iff ((u = val \wedge l \dashrightarrow v) \vee l : u \swarrow v \vee r : u \swarrow v)$$

Feature Extraction ...

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

Static code analysis

$t : u \swarrow v$



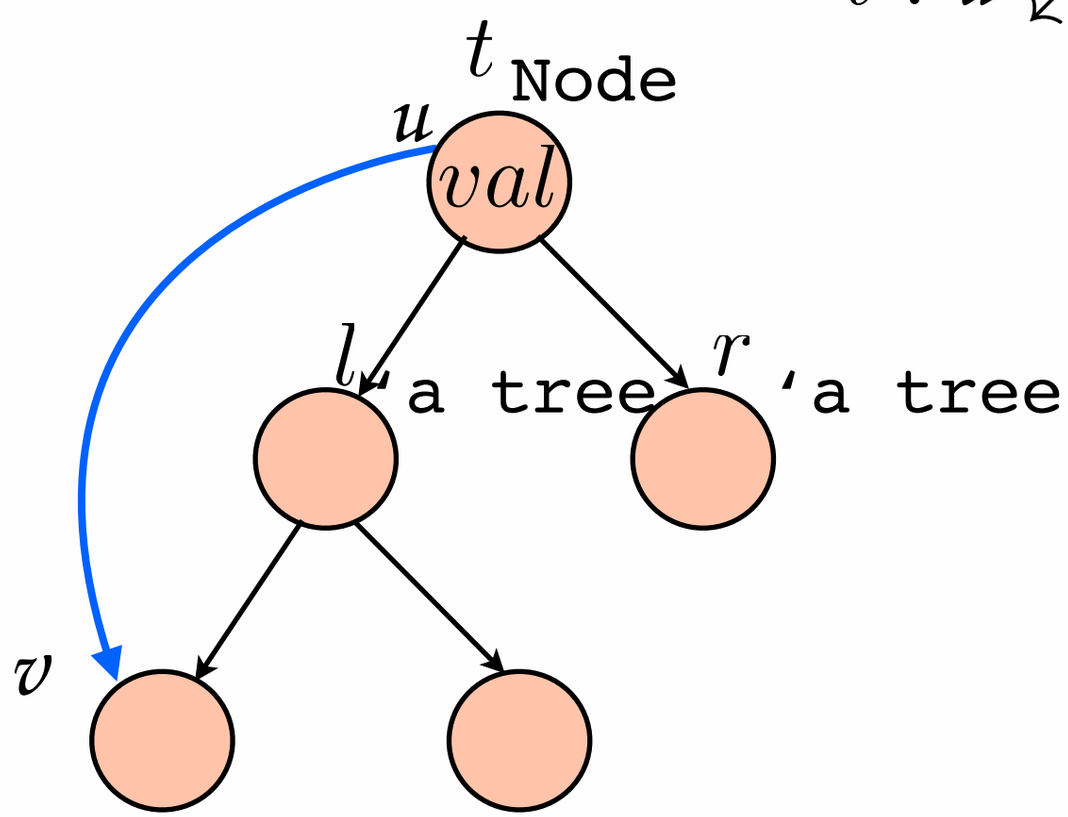
$$t : u \swarrow v \iff ((u = val \wedge l \dashrightarrow v) \vee l : u \swarrow v \vee r : u \swarrow v)$$

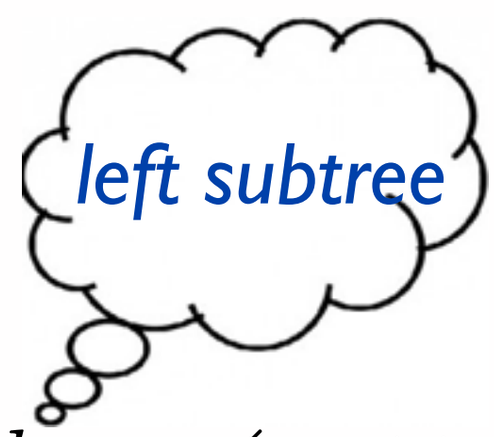
Feature Extraction ...

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

Static code analysis

$t : u \swarrow v$





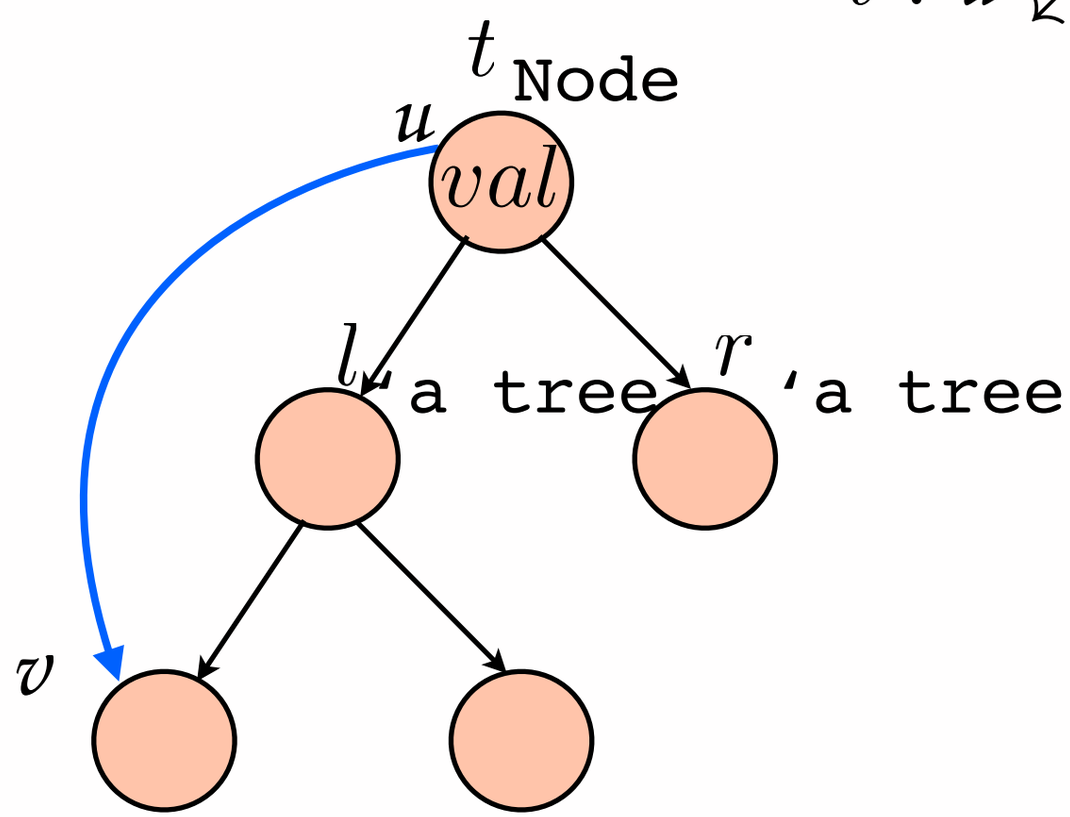
$$t : u \swarrow v \iff ((u = val \wedge l \dashrightarrow v) \vee l : u \swarrow v \vee r : u \swarrow v)$$

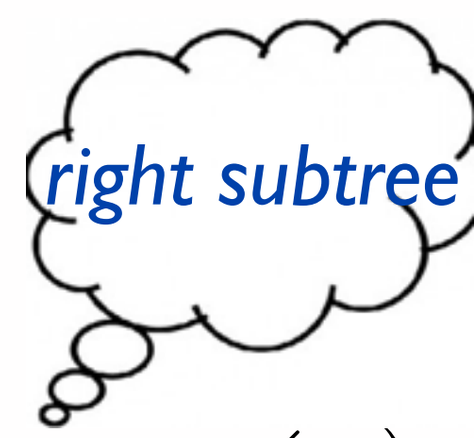
Feature Extraction ...

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

Static code analysis

$t : u \swarrow v$





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

Feature Extraction ...

Static code analysis

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

$t : u \swarrow v$

$t : u \searrow v$

Feature Extraction ...

Static code analysis

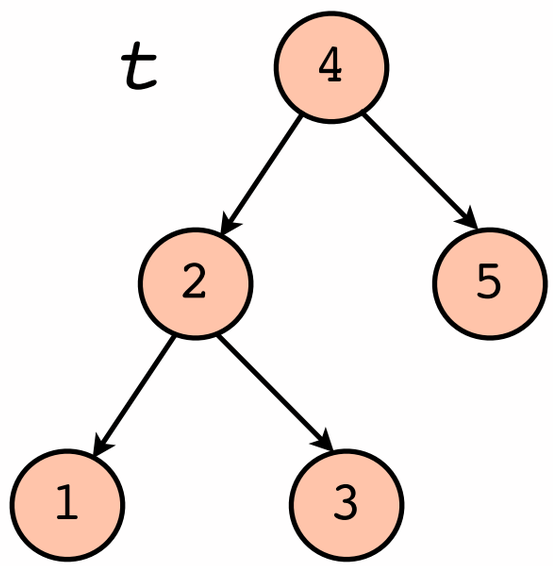
```
type 'a tree =
  | Leaf
  | Node 'a * 'a tree * 'a tree
```

$t : u \circlearrowright v$

$t : u \swarrow v$

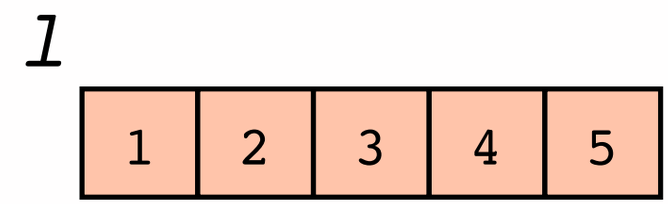
$t : u \searrow v$

Learner ...




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

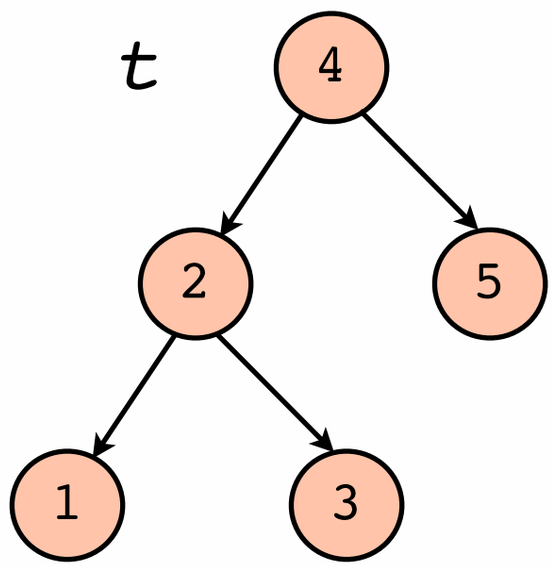
$$l = \text{elements } t$$

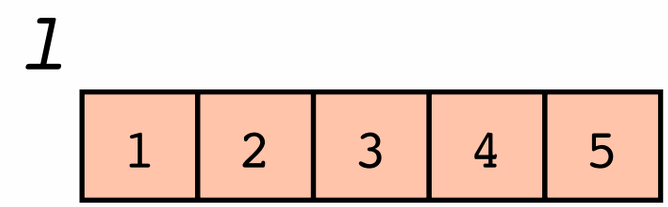


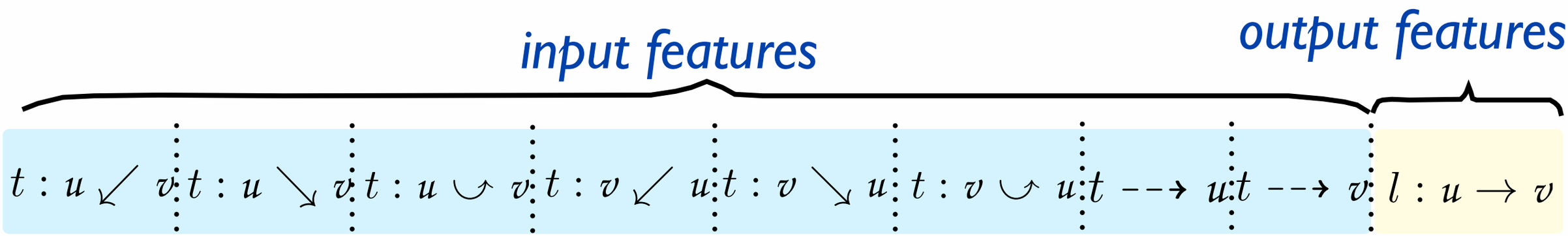
Learner ...

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

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



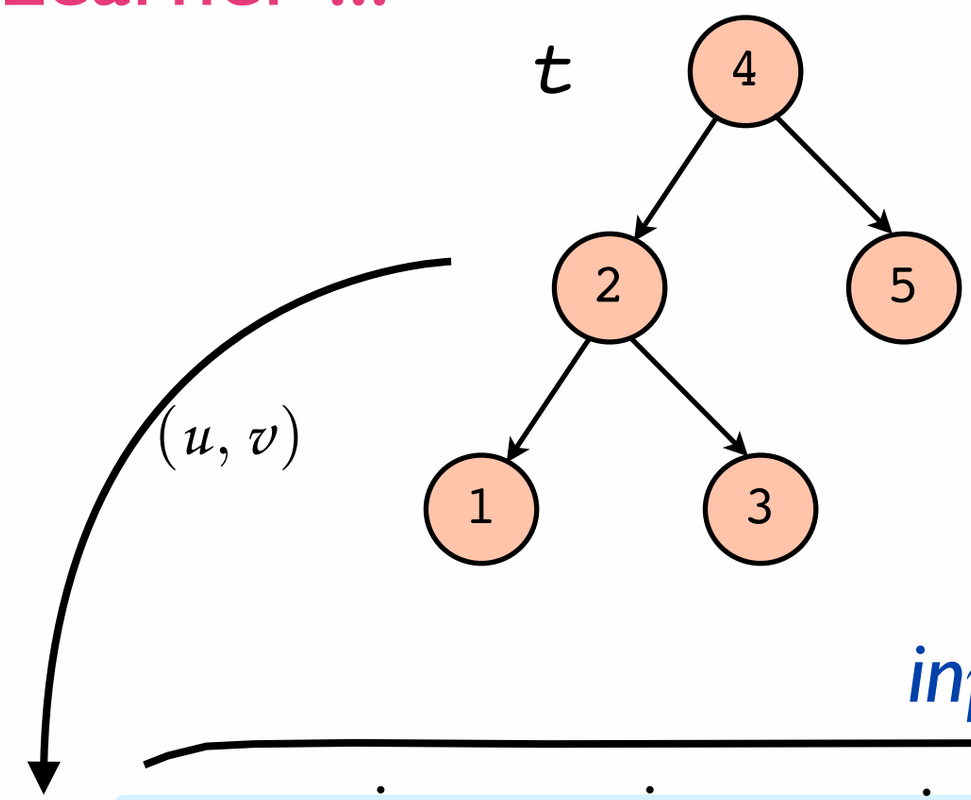


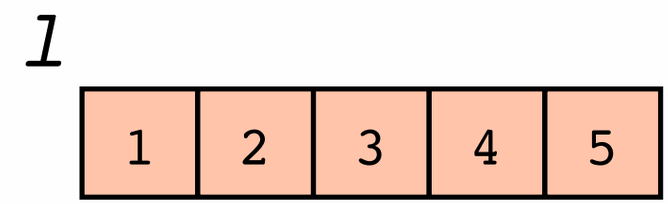


Learner ...

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

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





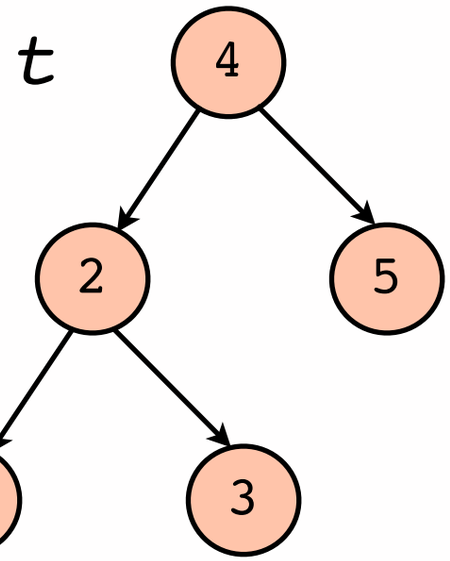
input features | *output features*

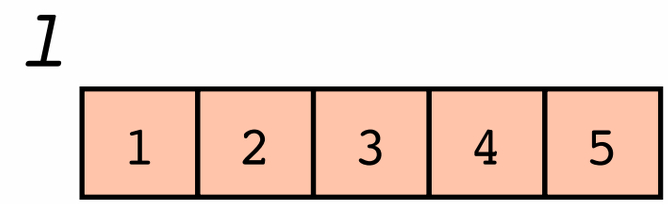
(u, v)	$t : u \swarrow v$	$t : u \searrow v$	$t : u \circlearrowleft v$	$t : v \swarrow u$	$t : v \searrow u$	$t : v \circlearrowleft u$	$t \dashrightarrow u$	$t \dashrightarrow v$	$l : u \rightarrow v$
(1,2)									
(4,5)									
(2,5)									
(3,1)									
(3,2)									
(4,1)									
⋮									

Learner ...

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

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





(u, v)

input features | *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)
(4,5)
(2,5)
(3,1)
(3,2)
(4,1)

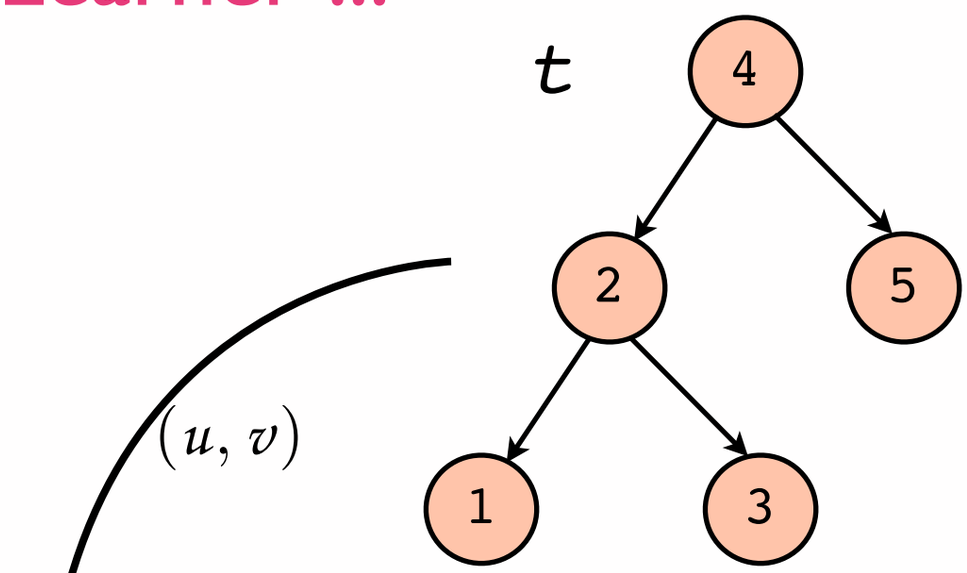
⋮

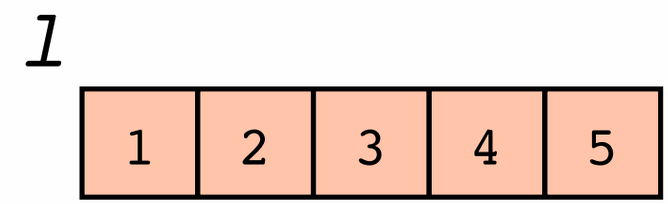
Sample space

Learner ...

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

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





input features | *output features*

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

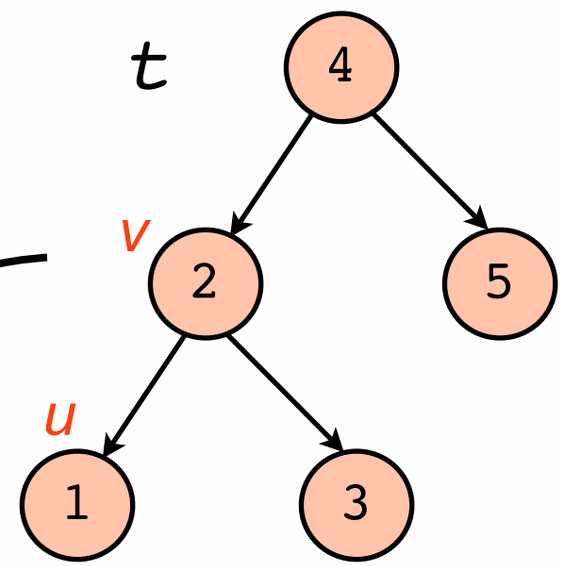
⋮

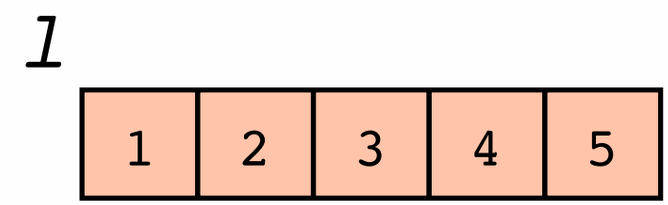
Sample space

Learner ...

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

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





(u, v)

input features | *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	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	0
(4,1)	1	0	0	0	0	0	1	1	0

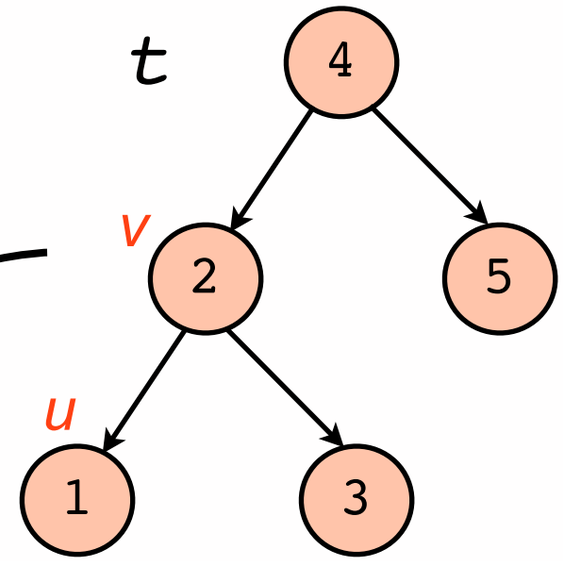
⋮

Sample space

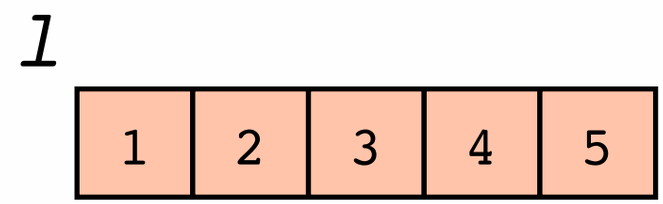
Learner ...

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

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



(u, v)



input features | *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	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	0
(4,1)	1	0	0	0	0	0	1	1	0

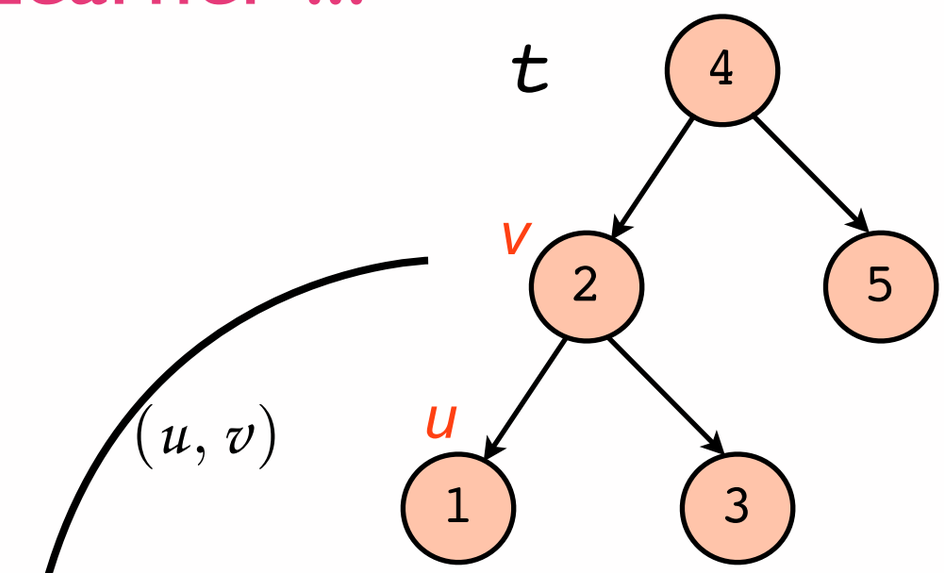
⋮

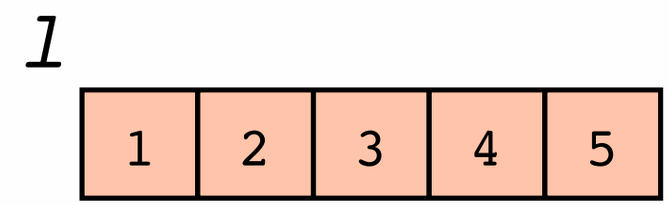
Sample space

Learner ...

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

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





input features | *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	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	0
(4,1)	1	0	0	0	0	0	1	1	0

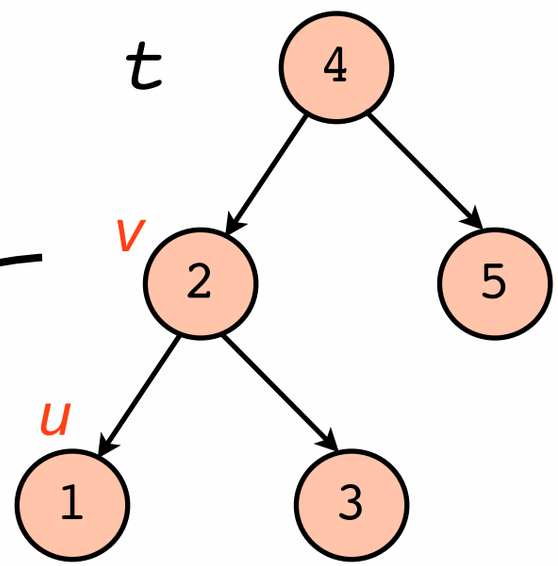
⋮

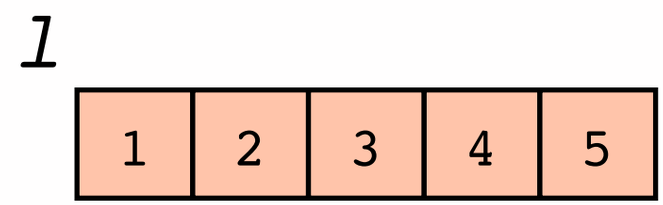
Sample space

Learner ...

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

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





(u, v)

input features | *output feature*

(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	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	0
(4,1)	1	0	0	0	0	0	1	1	0

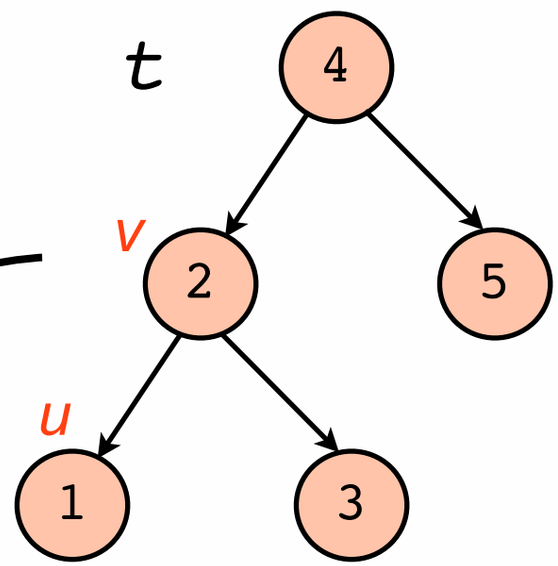
⋮

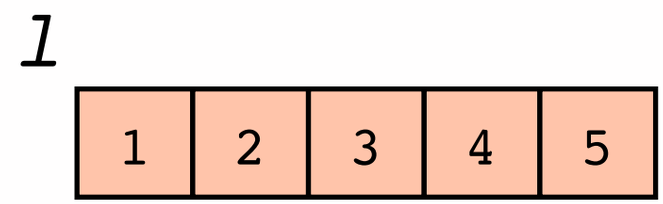
Sample space

Learner ...

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

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





(u, v)

input features | *output feature*

(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		0
(4,1)	1	0	0	0	0	0	1	1		0

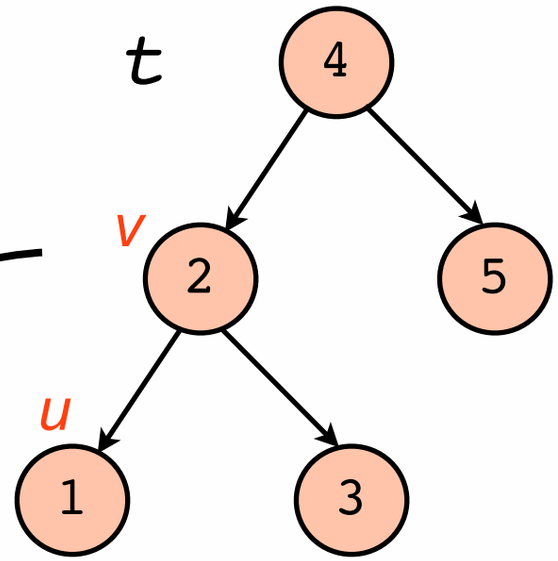
⋮

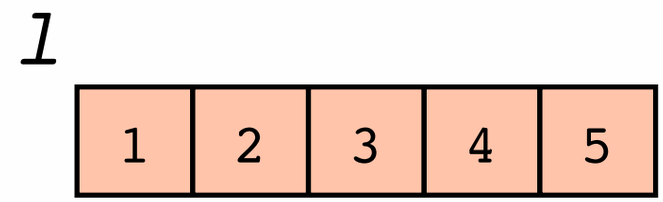
Sample space

Learner ...

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

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





(u, v)

input features | *output feature*

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

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	φ
$l : u \to v$	(4,5)	0	1	0	0	0	0	1	1	
	(2,5)	0	0	1	0	0	0	1	1	
neg samples	(3,1)	0	0	0	0	0	1	1	1	$\neg\varphi$
$\neg\, l : u \to v$	(3,2)	0	0	0	0	1	0	1	1	
	(4,1)	1	0	0	0	0	0	1	1	

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	φ
$l : u \to v$	(4,5)	0	1	0	0	0	0	1	1	
	(2,5)	0	0	1	0	0	0	1	1	
neg samples	(3,1)	0	0	0	0	0	1	1	1	$\neg\varphi$
$\neg\, l : u \to v$	(3,2)	0	0	0	0	1	0	1	1	
	(4,1)	1	0	0	0	0	0	1	1	

SPECS

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

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										
$l:u \to v$	(1,2)	0	0	0	1	0	0	1	1	φ
	(4,5)	0	1	0	0	0	0	1	1	
	(2,5)	0	0	1	0	0	0	1	1	
neg samples										
$\neg\, l:u \to v$	(3,1)	0	0	0	0	0	1	1	1	$\neg\varphi$
	(3,2)	0	0	0	0	1	0	1	1	
	(4,1)	1	0	0	0	0	0	1	1	

SPECS

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

Learner ...

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

Learner ...

(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	pos
(4,5)	0	1	0	0	0	0	1	1	1	pos
(2,5)	0	0	1	0	0	0	1	1	1	pos
(3,1)	0	0	0	0	0	1	1	1	0	neg
(3,2)	0	0	0	0	1	0	1	1	0	neg
(4,1)	1	0	0	0	0	0	1	1	0	neg

- Optimization task:
  - Constraint solvers

Learner ...

(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	pos
(4,5)	0	1	0	0	0	0	1	1	1	pos
(2,5)	0	0	1	0	0	0	1	1	1	pos
(3,1)	0	0	0	0	0	1	1	1	0	neg
(3,2)	0	0	0	0	1	0	1	1	0	neg
(4,1)	1	0	0	0	0	0	1	1	0	neg

- Optimization task:
  - Constraint solvers

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	1	pos
(4,5)	0	1	0	0	0	0	1	1	1	pos
(2,5)	0	0	1	0	0	0	1	1	1	pos
(3,1)	0	0	0	0	0	1	1	1	0	neg
(3,2)	0	0	0	0	1	0	1	1	0	neg
(4,1)	1	0	0	0	0	0	1	1	0	neg

- Optimization task:
  - Constraint solvers

Learner ...

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

Truth Table

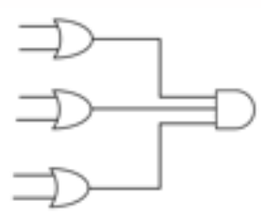
$t : u \searrow v$	$t : u \circlearrowleft v$	$t : v \swarrow u$	$l : u \to v$
0	0	1	1
1	0	0	1
0	1	0	1
0	0	0	0
0	0	0	0
0	0	0	0

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



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

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

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!

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!

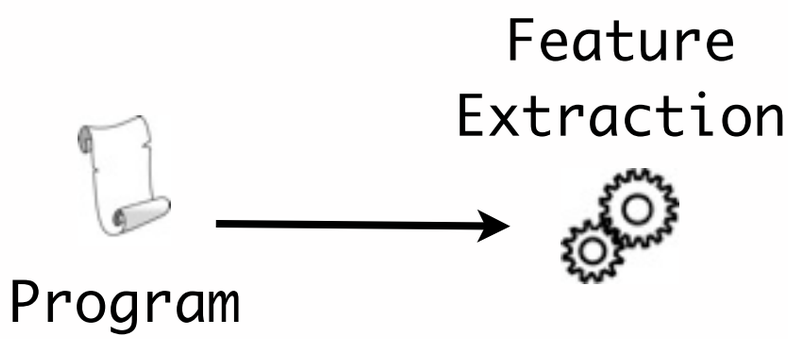
?

$\Longleftarrow \quad \Longrightarrow$

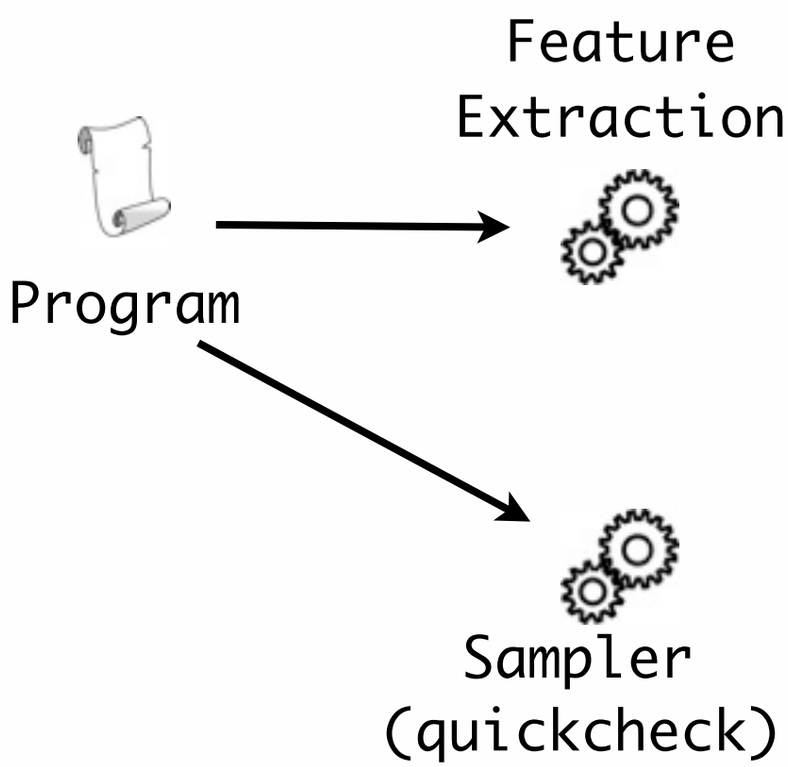
Verification Framework ...

Program

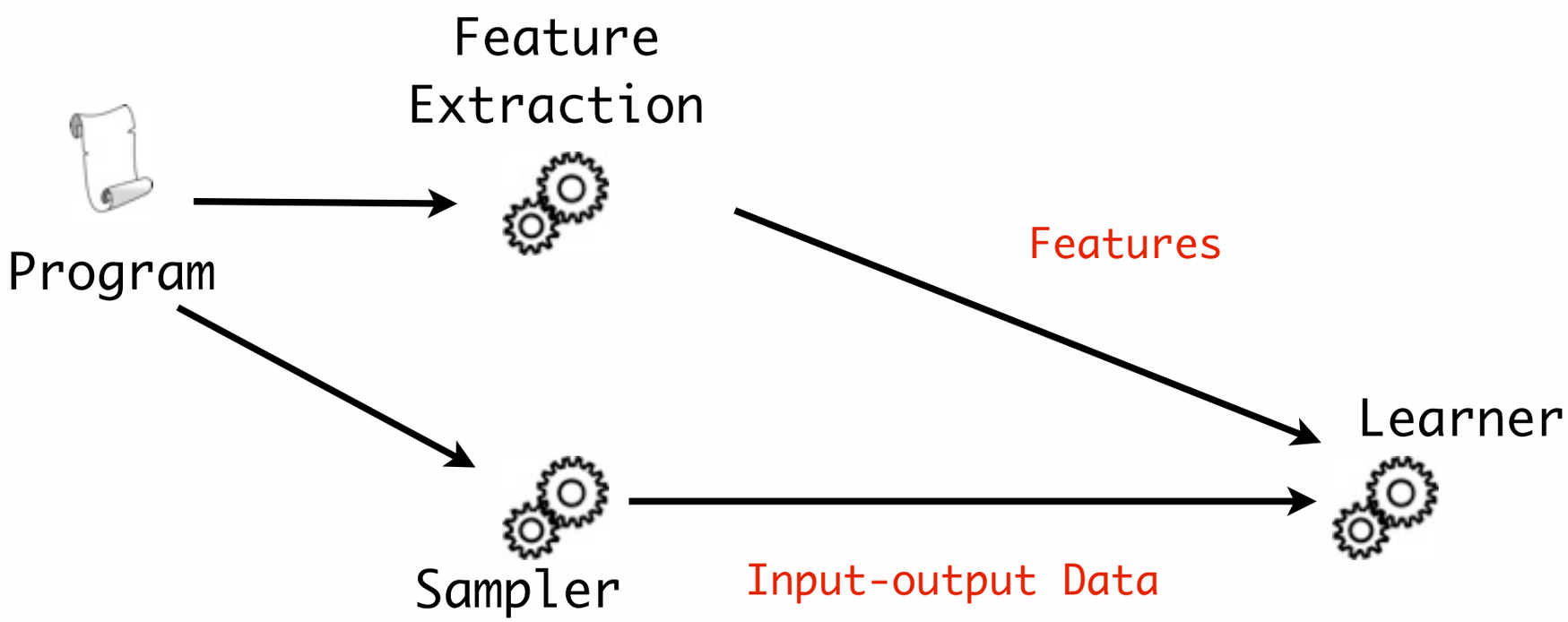
Verification Framework ...



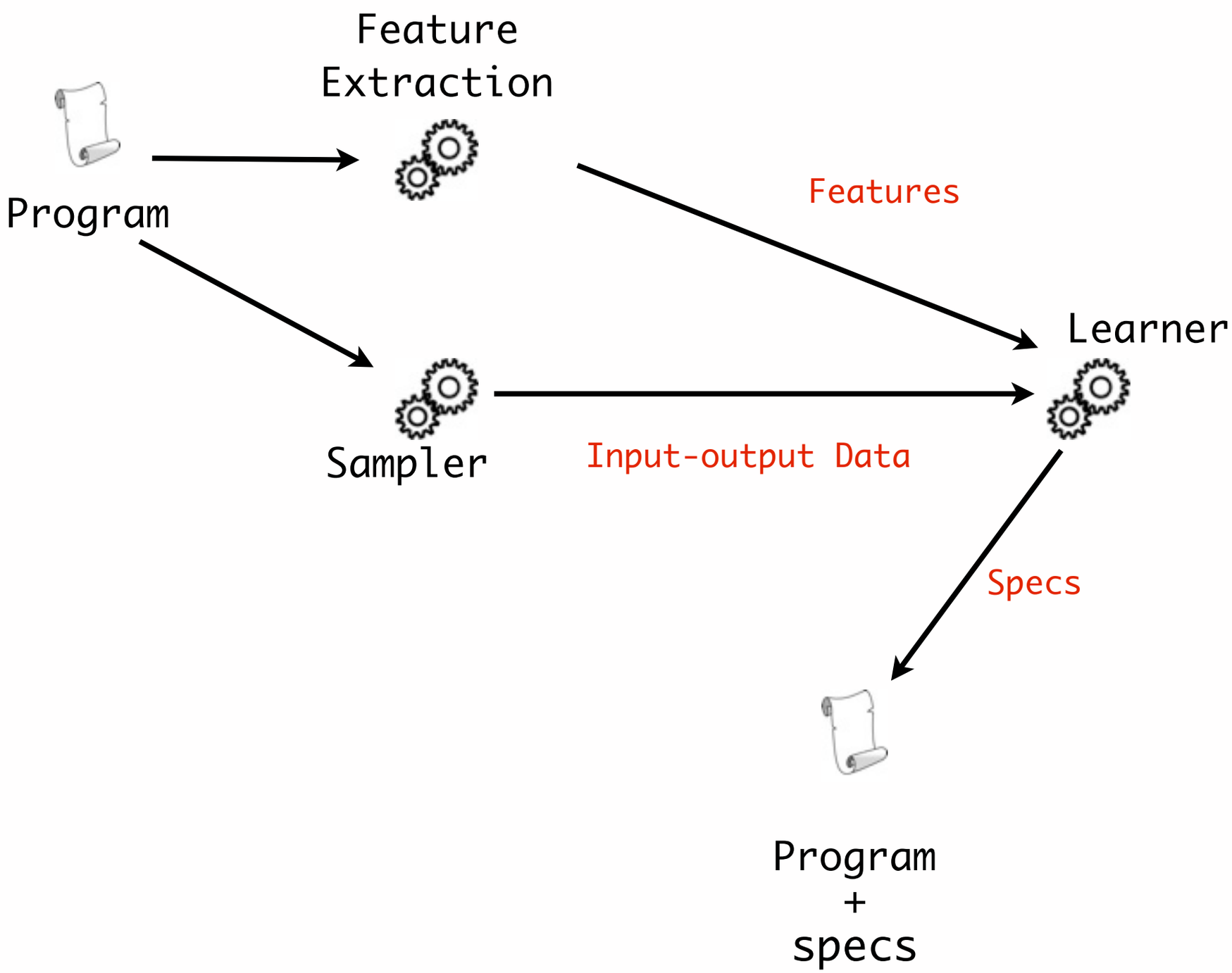
Verification Framework ...



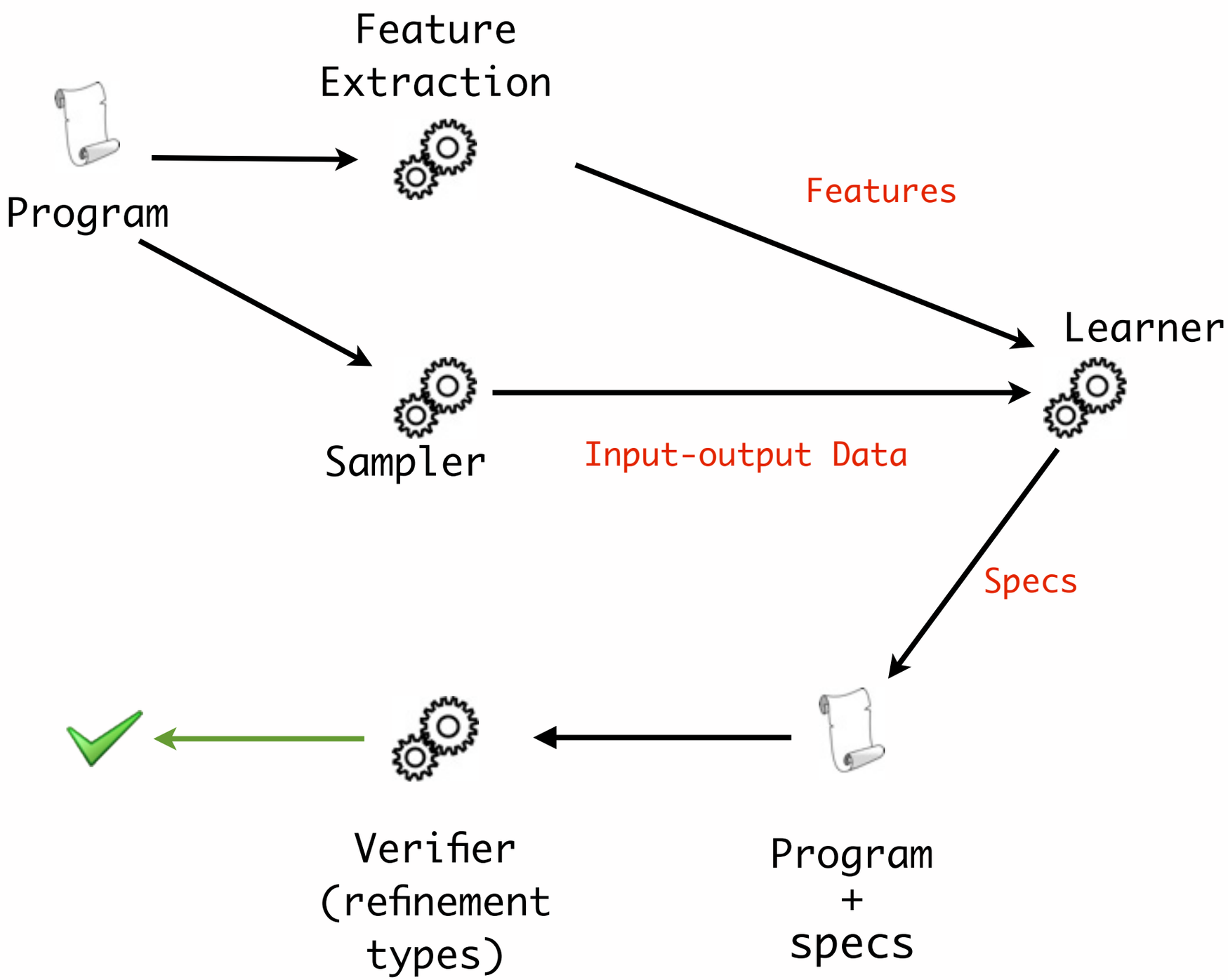
Verification Framework ...



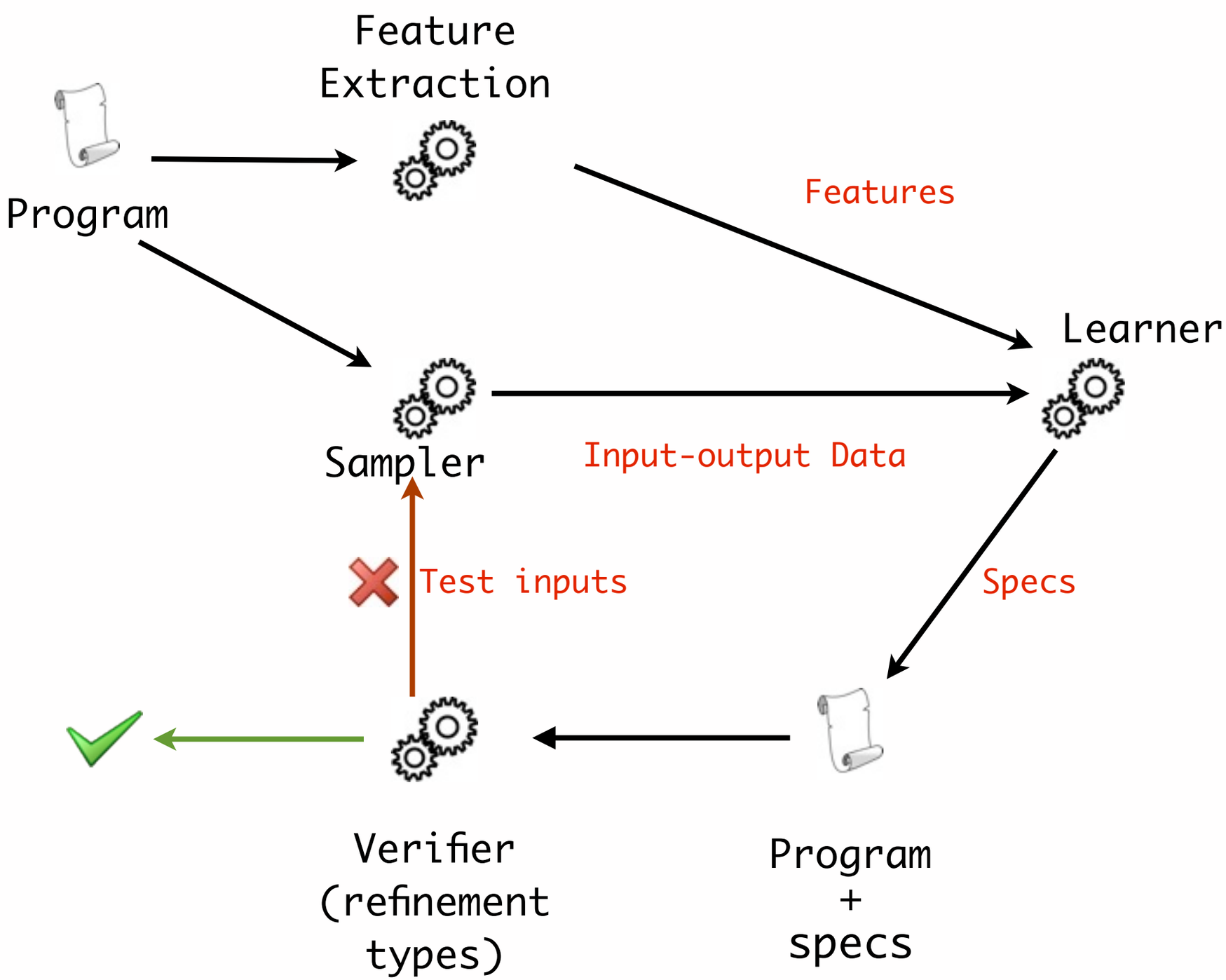
Verification Framework ...



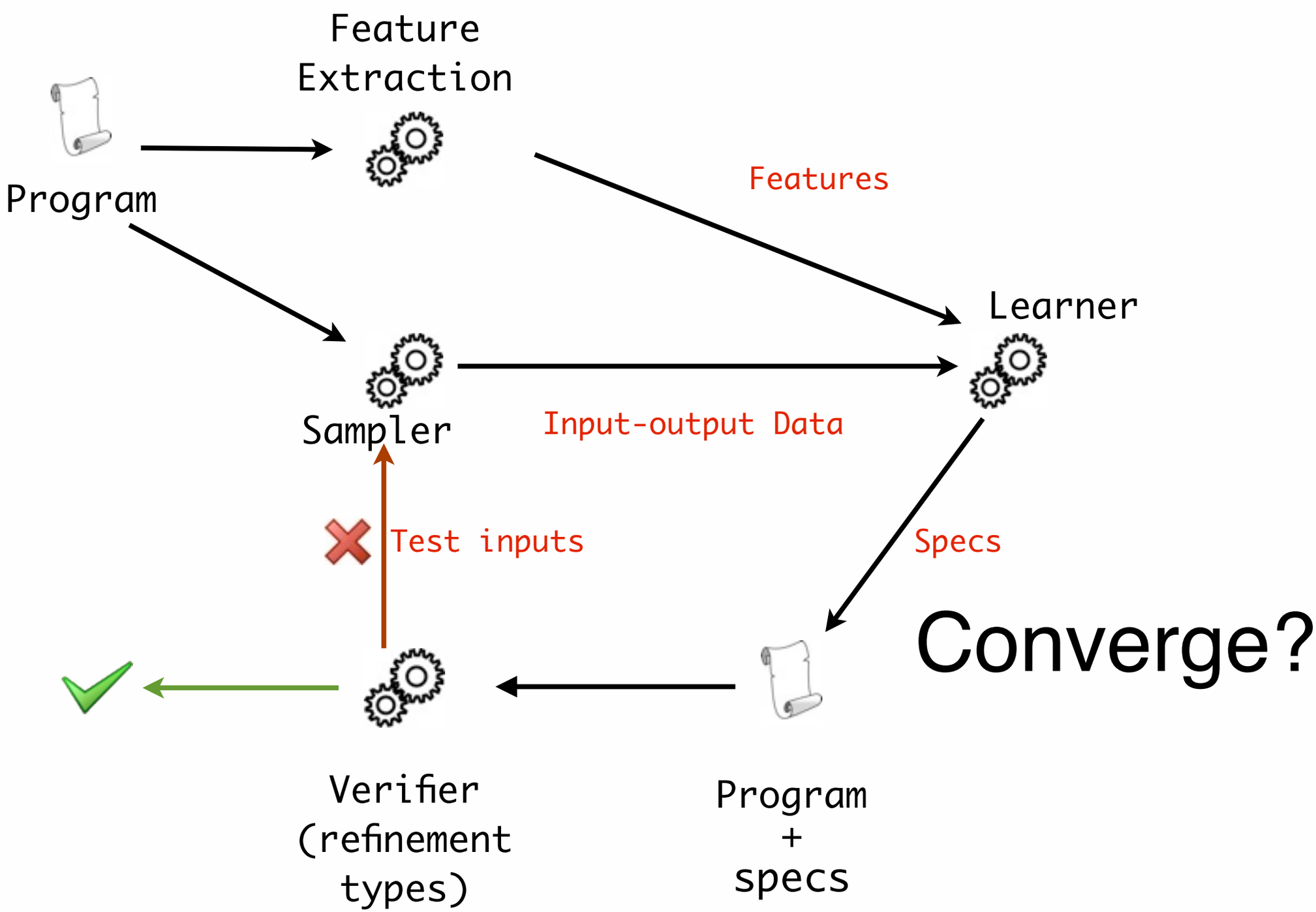
Verification Framework ...



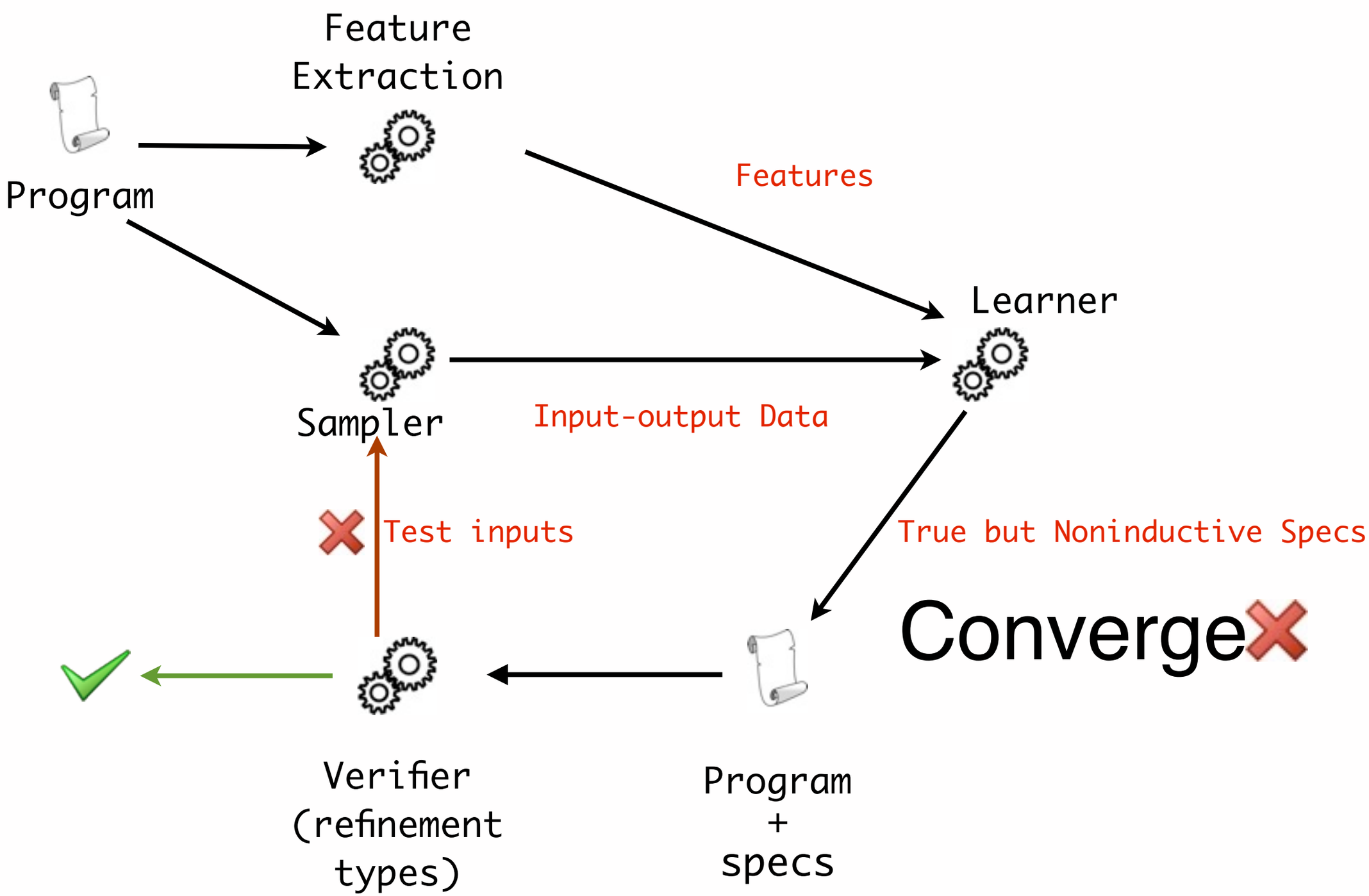
Verification Framework ...



Verification Framework ...

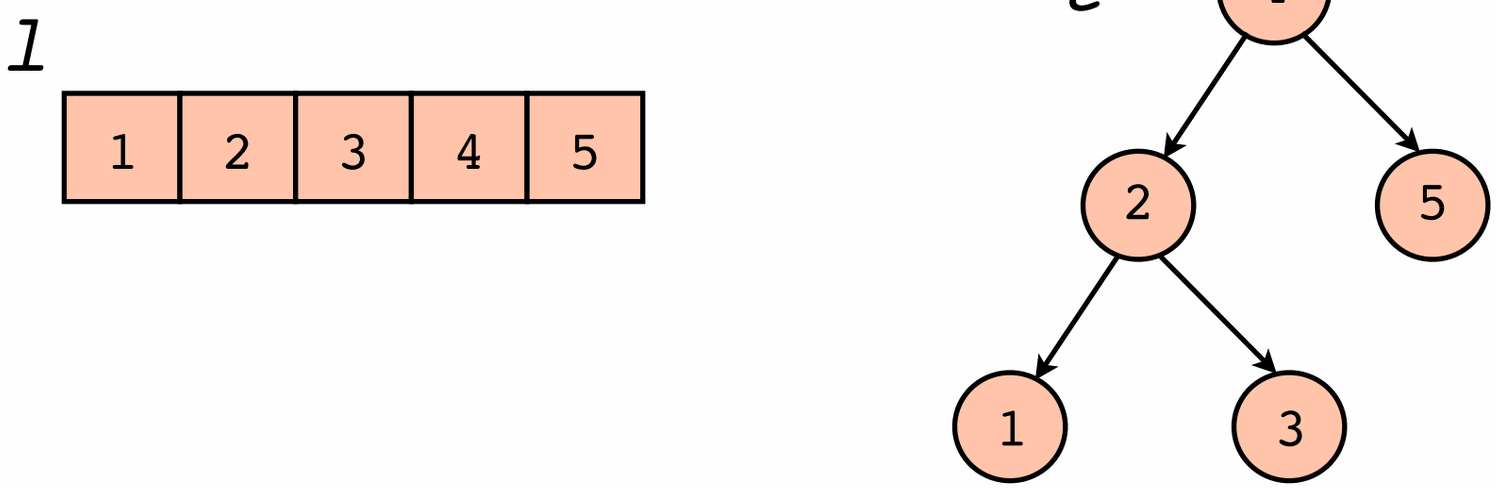


Verification Framework ...



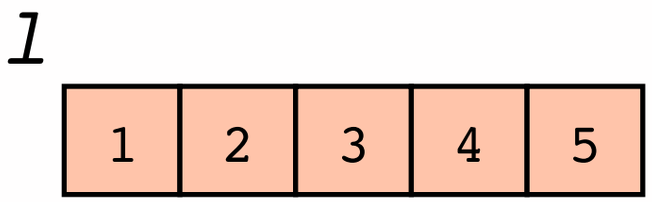
Verification and Convergence ...

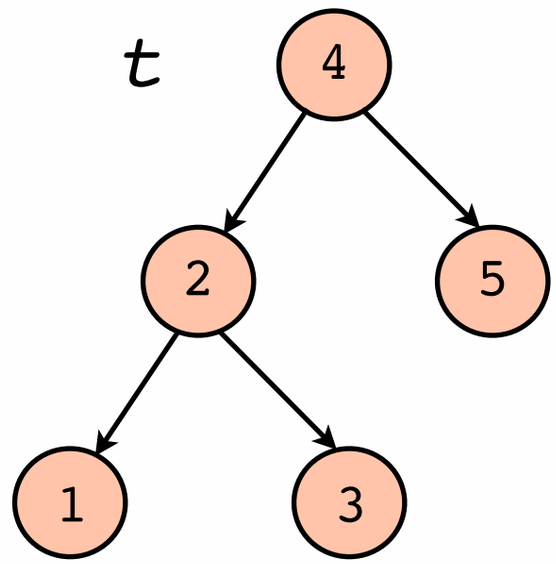
```
l:list = elements (t:tree)
```



Verification and Convergence ...

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





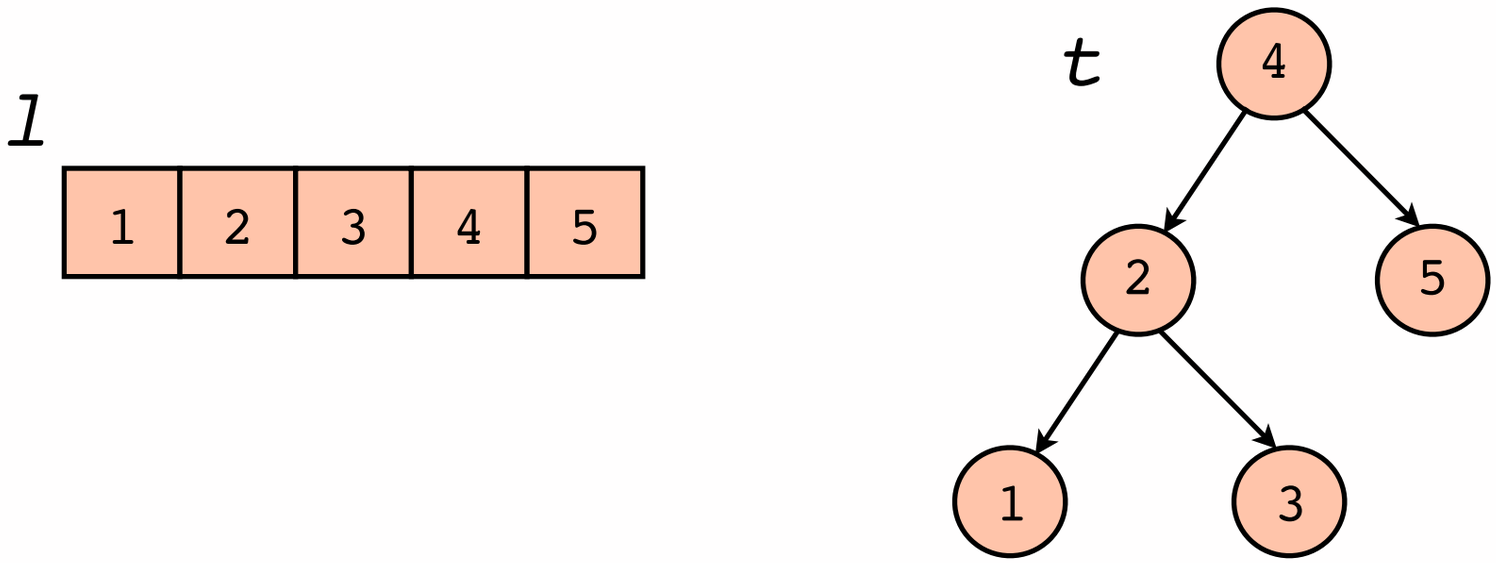
A hypothesis specification with a minimal number of features:

$$\left(\forall u\ v,\ l : u \rightarrow v \Rightarrow\ \left(t \dashrightarrow u \wedge t \dashrightarrow v\right)\right)$$

Non inductive invariant

Verification and Convergence ...

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



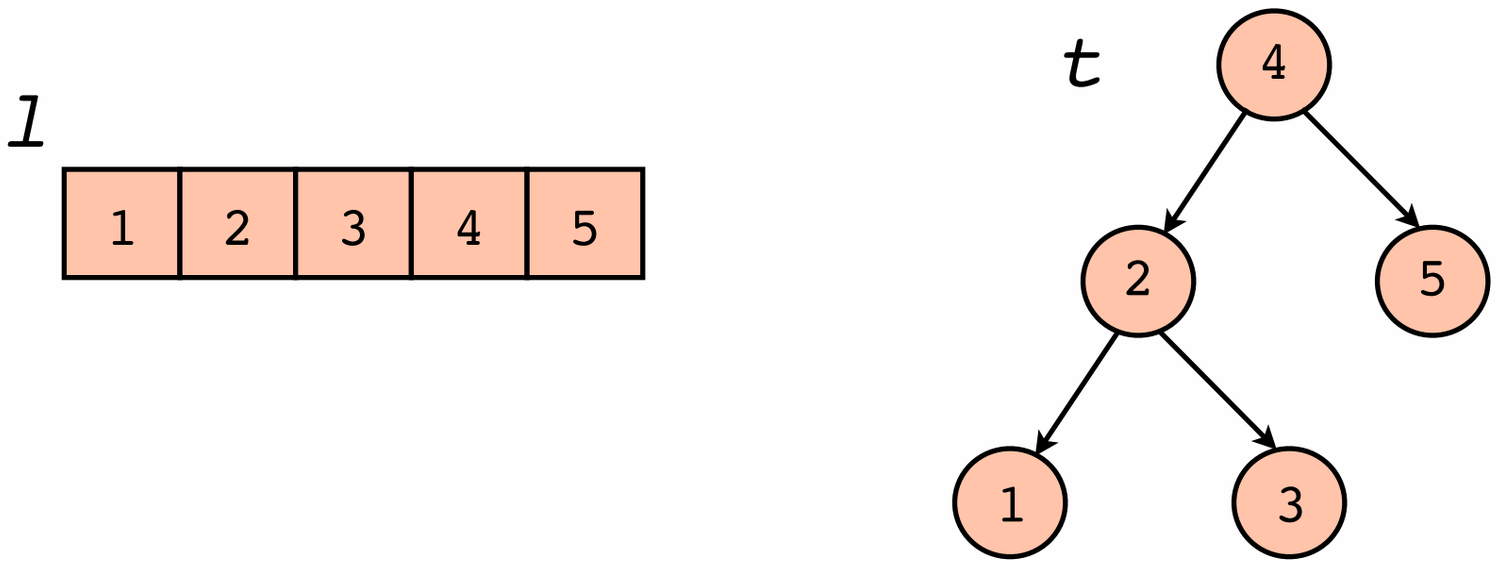
A hypothesis specification with a minimal number of features:

✗ $(\forall u\ v,\ l : u \rightarrow v \Rightarrow\ (t \dashrightarrow u \wedge t \dashrightarrow v))$ ✗

	(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									
$l : u \rightarrow v$	(1,2)	0	0	0	1	0	0	1	1
	(4,5)	0	1	0	0	0	0	1	1
	(2,5)	0	0	1	0	0	0	1	1
neg samples									
$\neg\ l : u \rightarrow v$	(3,1)	0	0	0	0	0	1	1	1
	(3,2)	0	0	0	0	1	0	1	1
	(4,1)	1	0	0	0	0	0	1	1

Verification and Convergence ...

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

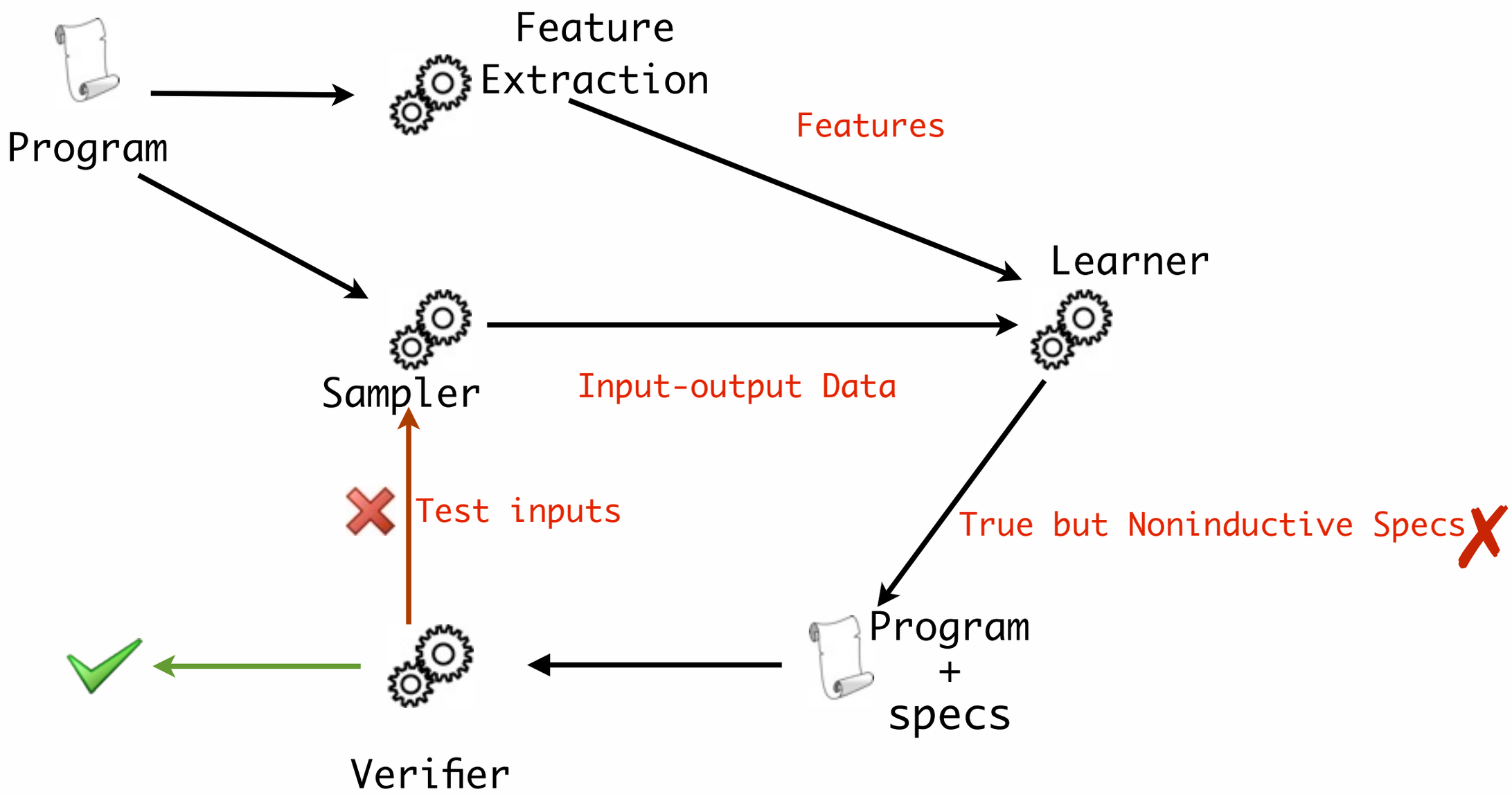


A hypothesis specification with a minimal number of features:

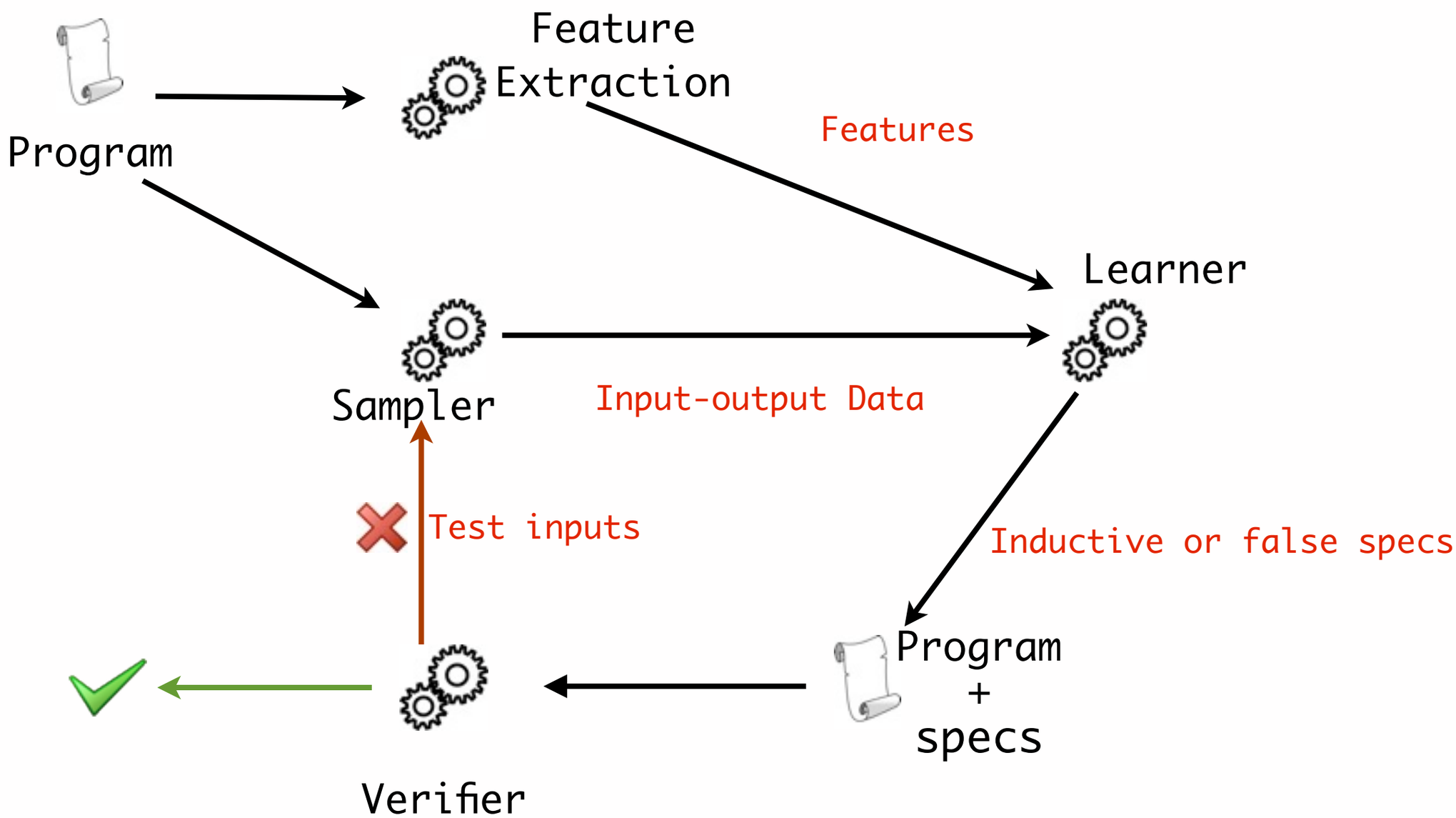
✗ $(\forall u\ v,\ l : u \rightarrow v \Rightarrow\ (t \dashrightarrow u \wedge t \dashrightarrow v))$

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

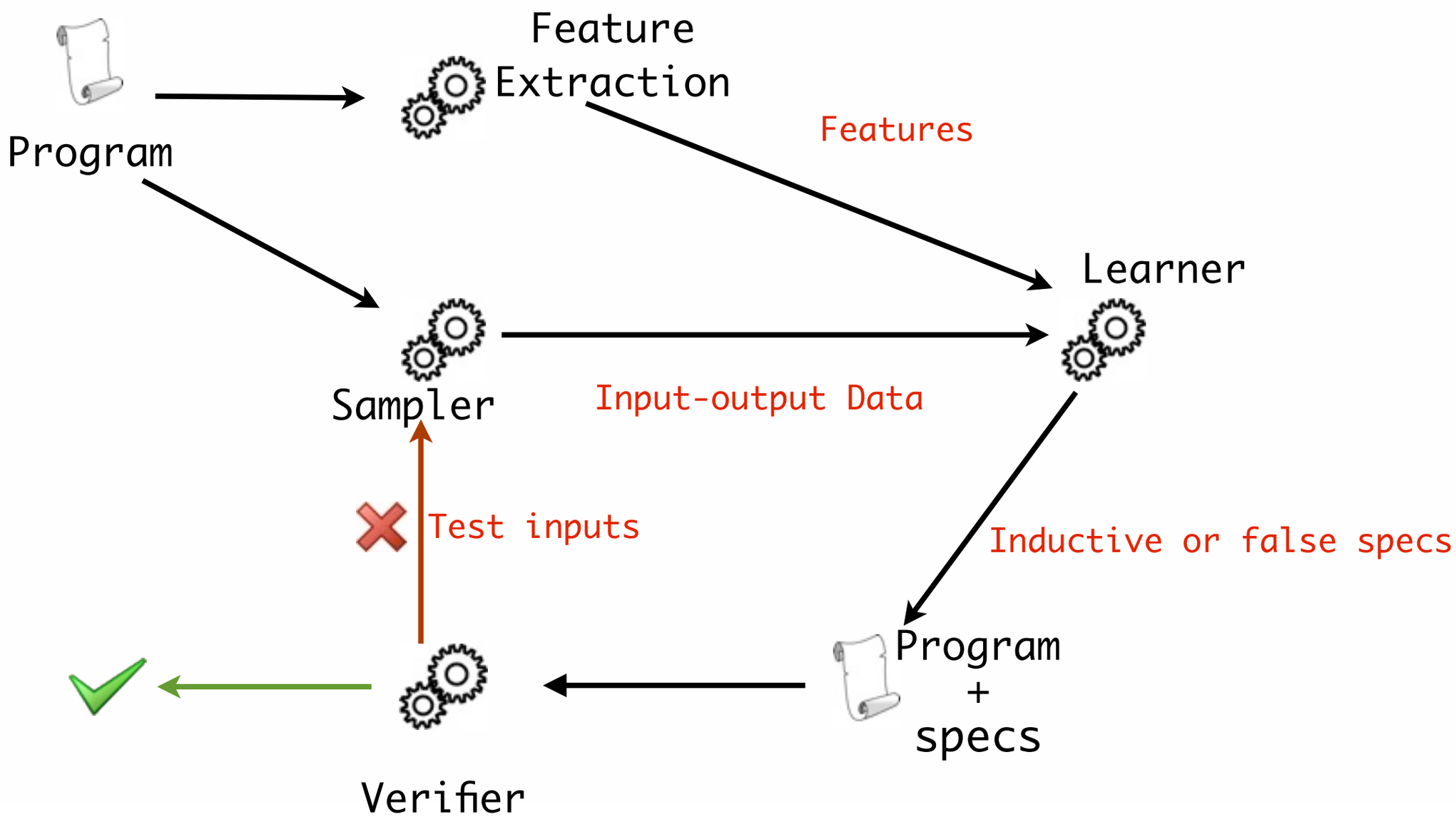
Verification and Convergence ...



Verification and Convergence ...



Verification and Convergence ...



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

In the paper ...

In the paper ...

Predicting binary search tree?

In the paper ...

Predicting binary search tree?

Shape-data specifications

In the paper ...

Predicting binary search tree?

Shape-data specifications

Predicting balanced tree?

In the paper ...

Predicting binary search tree?

Shape-data specifications

Predicting balanced tree?

Numeric specifications

In the paper ...

Predicting binary search tree?

Shape-data specifications

Predicting balanced tree?

Numeric specifications

General Framework for Specification Synthesis

Experimental Results ...

- DOrder -- implemented within the OCaml tool chain.
- Programmers write code as usual (with no annotation burden) while the tool reports program specifications.

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

Experimental Results ...

- Verification is fast.
  - Redblack tree in 2mins.
  - AVL tree in 1min.
  - Most benchmarks verified in less than 30s.

Experimental Results ...

- Verification is fast.
  - Redblack tree in 2mins.
  - AVL tree in 1min.
  - Most benchmarks verified in less than 30s.
- Specifications can be synthesized from a small number of tests.
  - Redblack tree with ~100 samples.
  - AVL tree with ~40 samples.
  - Most benchmarks verified with ~20 samples.

Unique Features ...

Unique Features ...

- High-Automation.
  - *Without* requiring assertions, pre- and post-conditions.

Unique Features ...

- High-Automation.
  - *Without* requiring assertions, pre- and post-conditions.

- High-Quality Guarantee.
  - The *strongest* specification (up to a hypothesis domain).

Unique Features ...

- High-Automation.
  - *Without* requiring assertions, pre- and post-conditions.
- High-Quality Guarantee.
  - The *strongest* specification (up to a hypothesis domain).
- Strong Convergence Guarantee.
  - Ensure there *always* exists a test to refine an unverifiable specification (if hypothesis space is sufficient).

Unique Features ...

- High-Automation.
  - *Without* requiring assertions, pre- and post-conditions.
- High-Quality Guarantee.
  - The *strongest* specification (up to a hypothesis domain).
- Strong Convergence Guarantee.
  - Ensure there *always* exists a test to refine an unverifiable specification (if hypothesis space is sufficient).
- Demonstrated applicable to real-world programs.

Thanks!

Q & A

https://github.com/rowangithub/DOrder

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
```

Binary Search Tree Insertion ...

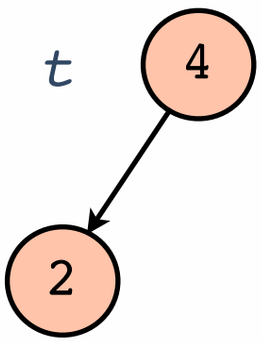
```
let rec insert x t =
 match t with
 | Leaf -> Node (x, Leaf, Leaf)
 | Node (y, l, r) ->
    if x < y then
        Node (y, insert x l, r)
     else if y < x then
        Node (y, l, insert x r)
     else t
```

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

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
```

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

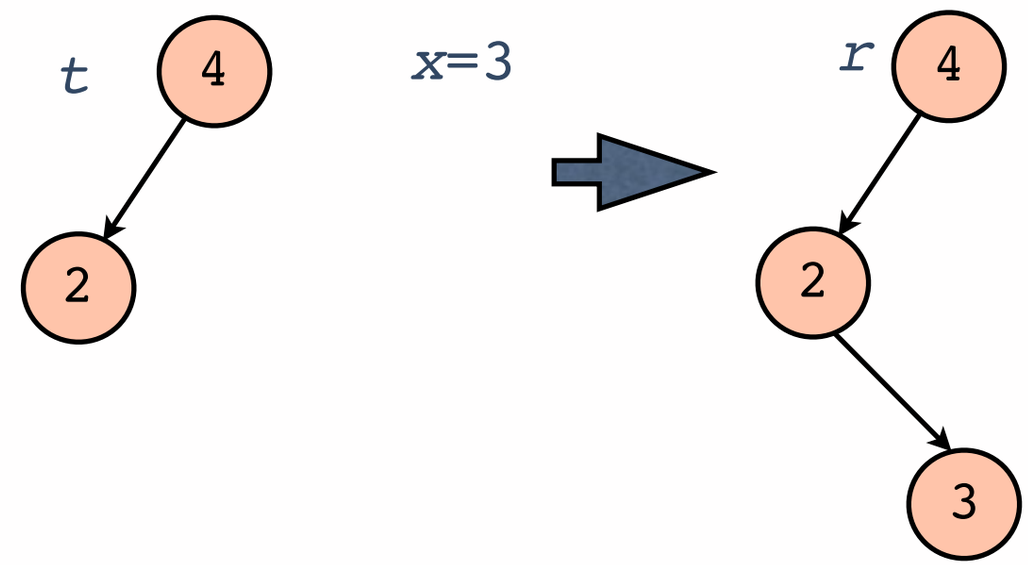


x=3

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
```

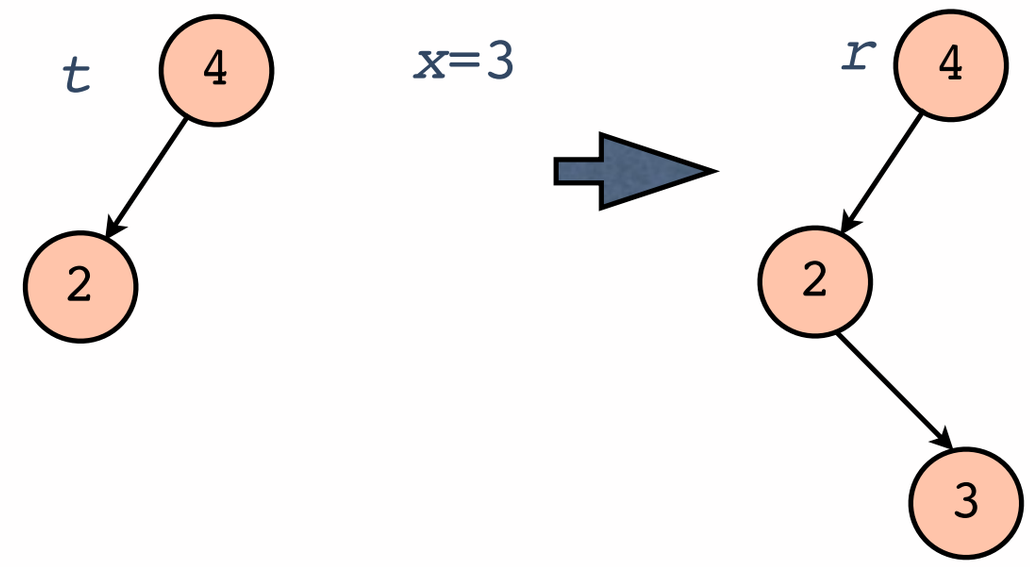
$$r = \text{insert } 3\ t$$



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
```

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



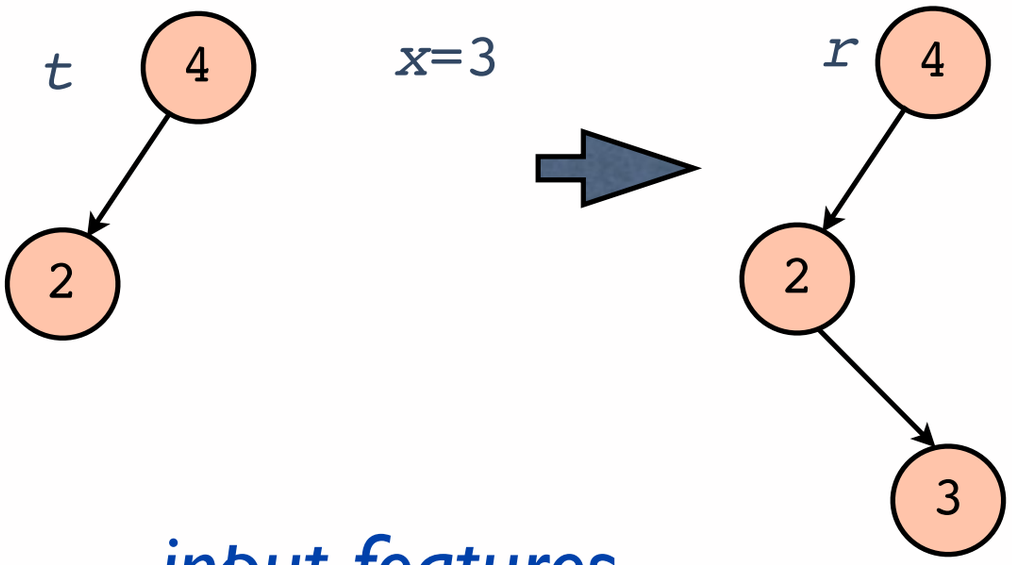
input features | *output features*

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

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
```

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



input features | *output features*

$\Pi_0\ t : u \swarrow v$

$\Pi_1\ t : u \searrow v$

$\Pi_2\ t : u \circlearrowleft v$

$\Pi_3\ t : v \swarrow u$

$\Pi_4\ t : v \searrow u$

$\Pi_5\ t : v \circlearrowleft u$

$\Pi_6\ t \dashrightarrow u$

$\Pi_7\ t \dashrightarrow v$

$\Pi_8\ u = x$

$\Pi_9\ v = x$

Π_{10}

$r : u \swarrow v$

$\vdots$

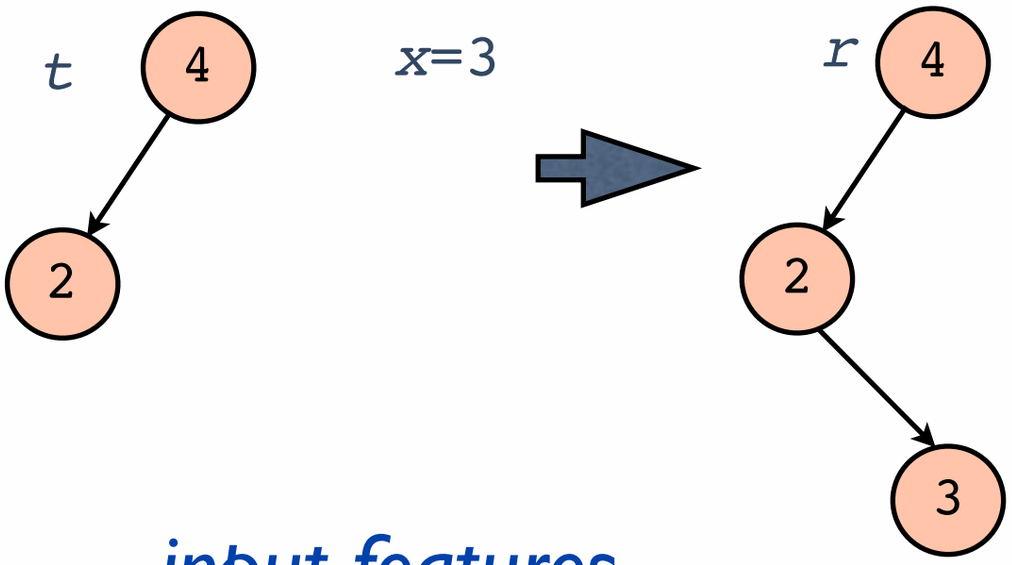
input features

(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	1

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
```

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



input features | *output features*

$\Pi_0\ t : u \swarrow v$

$\Pi_1\ t : u \searrow v$

$\Pi_2\ t : u \circlearrowleft v$

$\Pi_3\ t : v \swarrow u$

$\Pi_4\ t : v \searrow u$

$\Pi_5\ t : v \circlearrowleft u$

$\Pi_6\ t \dashrightarrow u$

$\Pi_7\ t \dashrightarrow v$

$\Pi_8\ u = x$

$\Pi_9\ v = x$

Π_{10}

$r : u \swarrow v$

$\vdots$

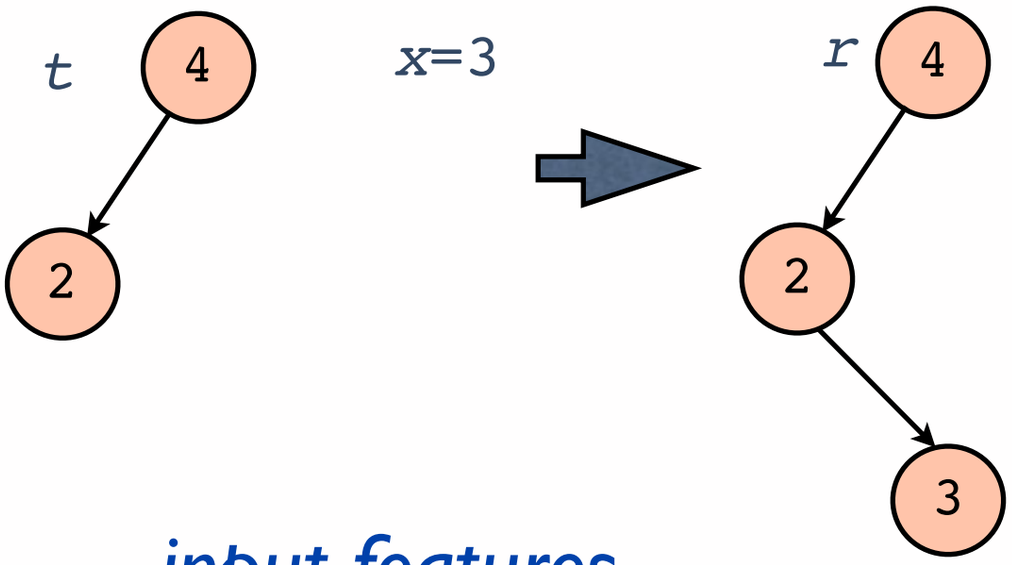
input features

(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	1
(2,3)	0	0	0	0	0	0	1	0	0	1	0

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
```

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



input features | *output features*

$\Pi_0\ t : u \swarrow v$

$\Pi_1\ t : u \searrow v$

$\Pi_2\ t : u \circlearrowleft v$

$\Pi_3\ t : v \swarrow u$

$\Pi_4\ t : v \searrow u$

$\Pi_5\ t : v \circlearrowleft u$

$\Pi_6\ t \dashrightarrow u$

$\Pi_7\ t \dashrightarrow v$

$\Pi_8\ u = x$

$\Pi_9\ v = x$

Π_{10}

$r : u \swarrow v$

$\vdots$

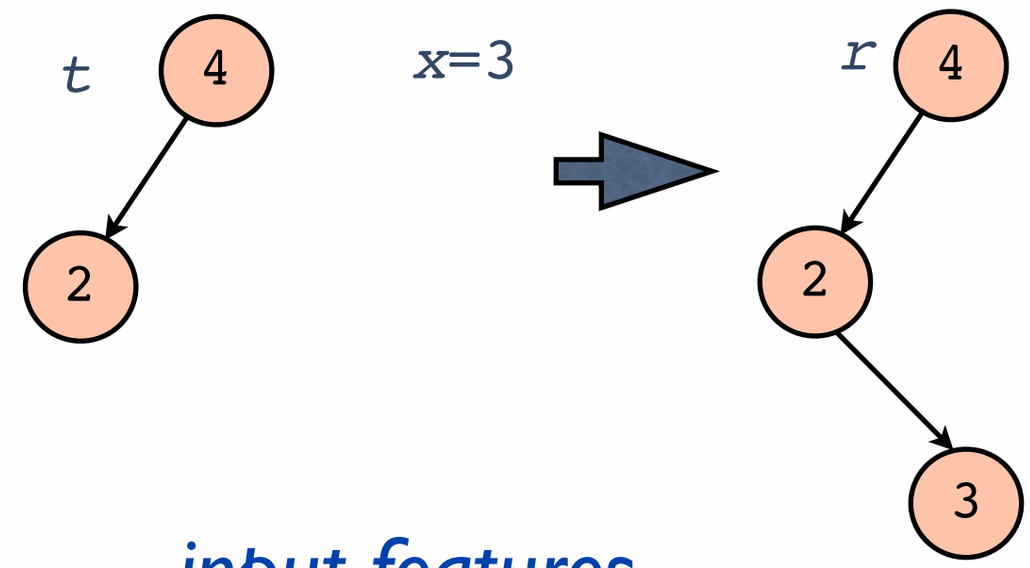
input features

(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

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
```

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



input features | *output features*

$\Pi_0\ t : u \swarrow v$

$\Pi_1\ t : u \searrow v$

$\Pi_2\ t : u \circlearrowleft v$

$\Pi_3\ t : v \swarrow u$

$\Pi_4\ t : v \searrow u$

$\Pi_5\ t : v \circlearrowleft u$

$\Pi_6\ t \dashrightarrow u$

$\Pi_7\ t \dashrightarrow v$

$\Pi_8\ u = x$

$\Pi_9\ v = x$

Π_{10}

$r : u \swarrow v$

$\vdots$

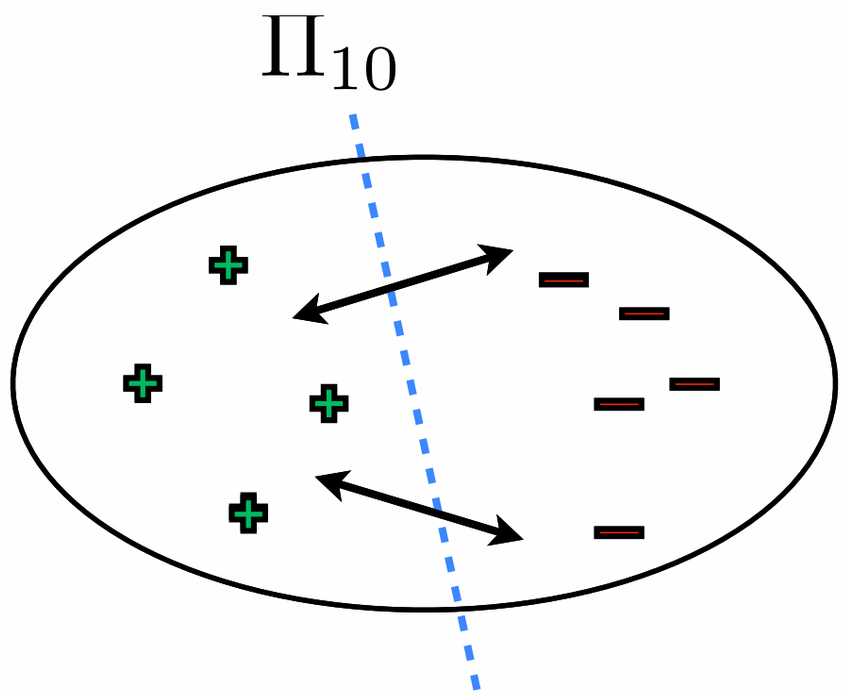
input features

(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

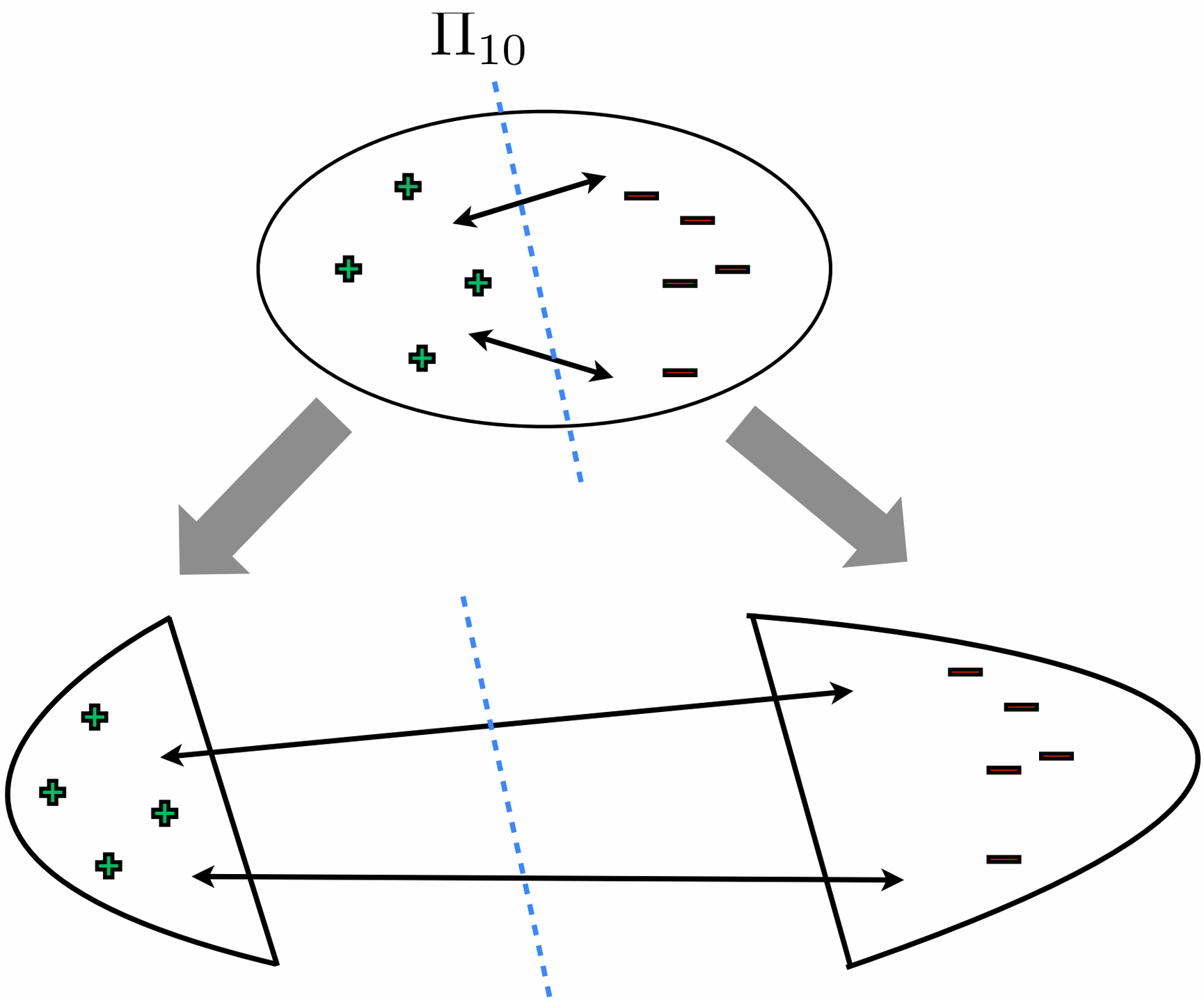
Problem:

Samples are not separable with existing features

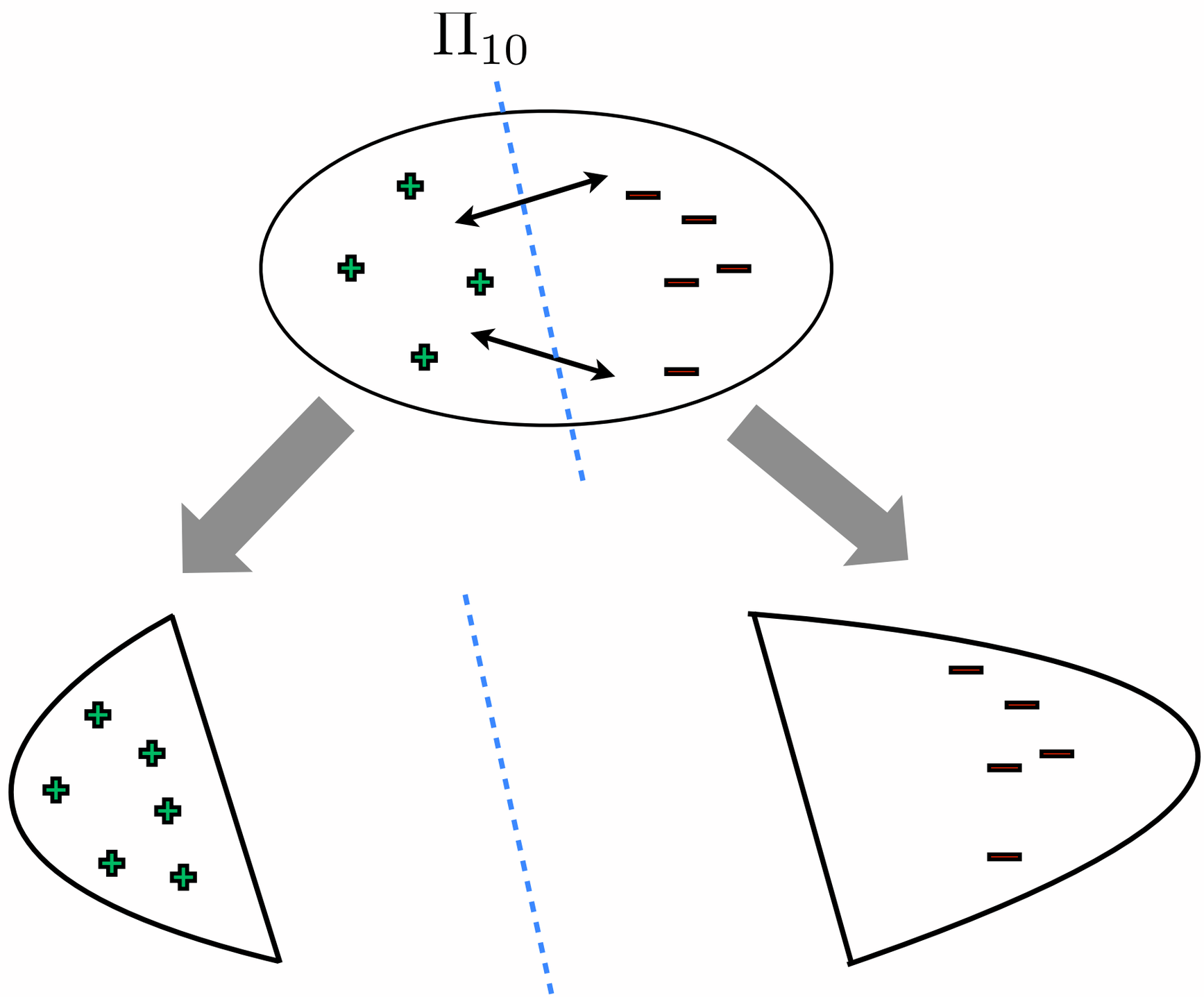
Learner ...



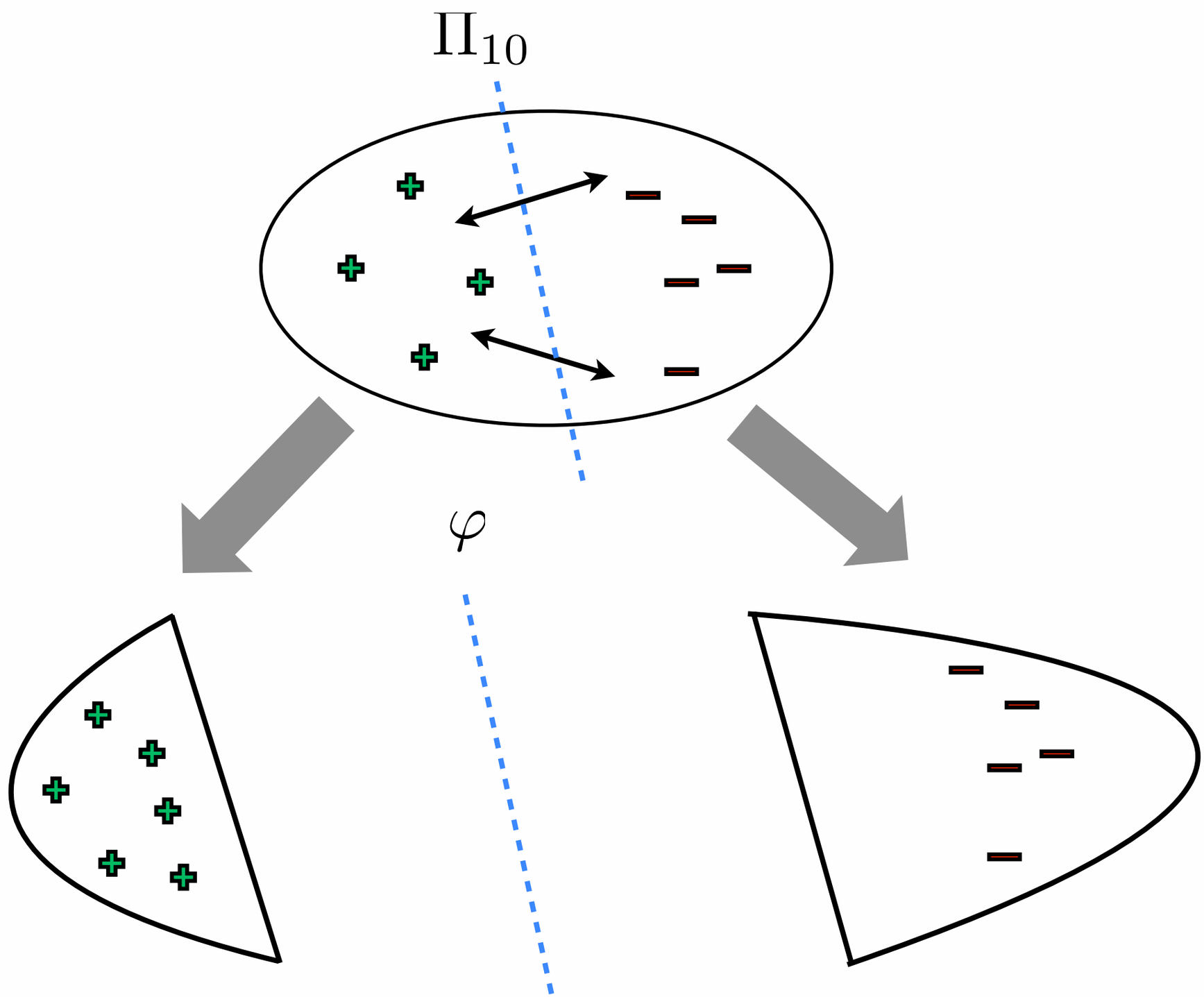
Learner ...



Learner ...

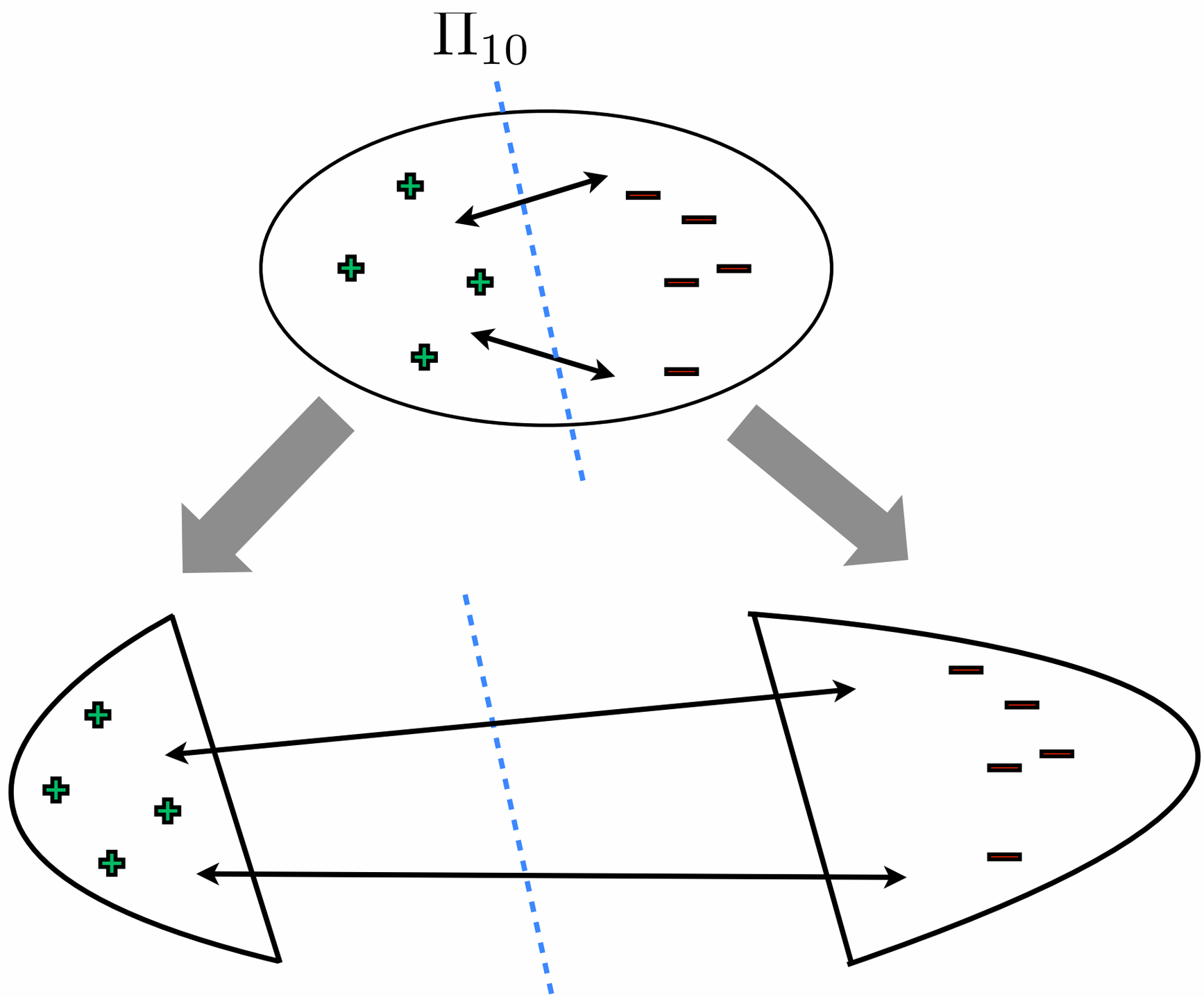


Learner ...



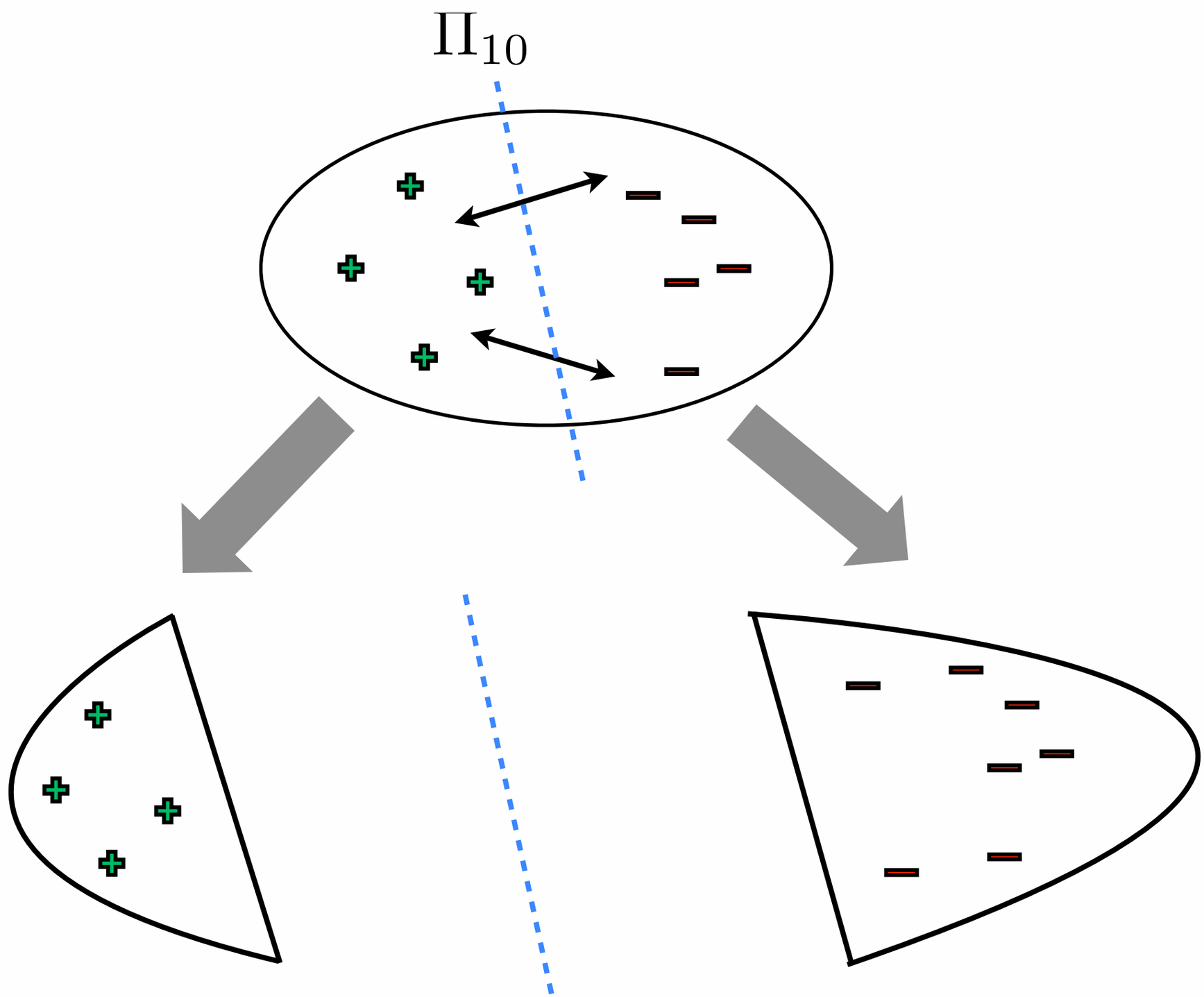
$$\Pi_{10} \Rightarrow \varphi(\Pi_0, \cdots, \Pi_9)$$

Learner ...



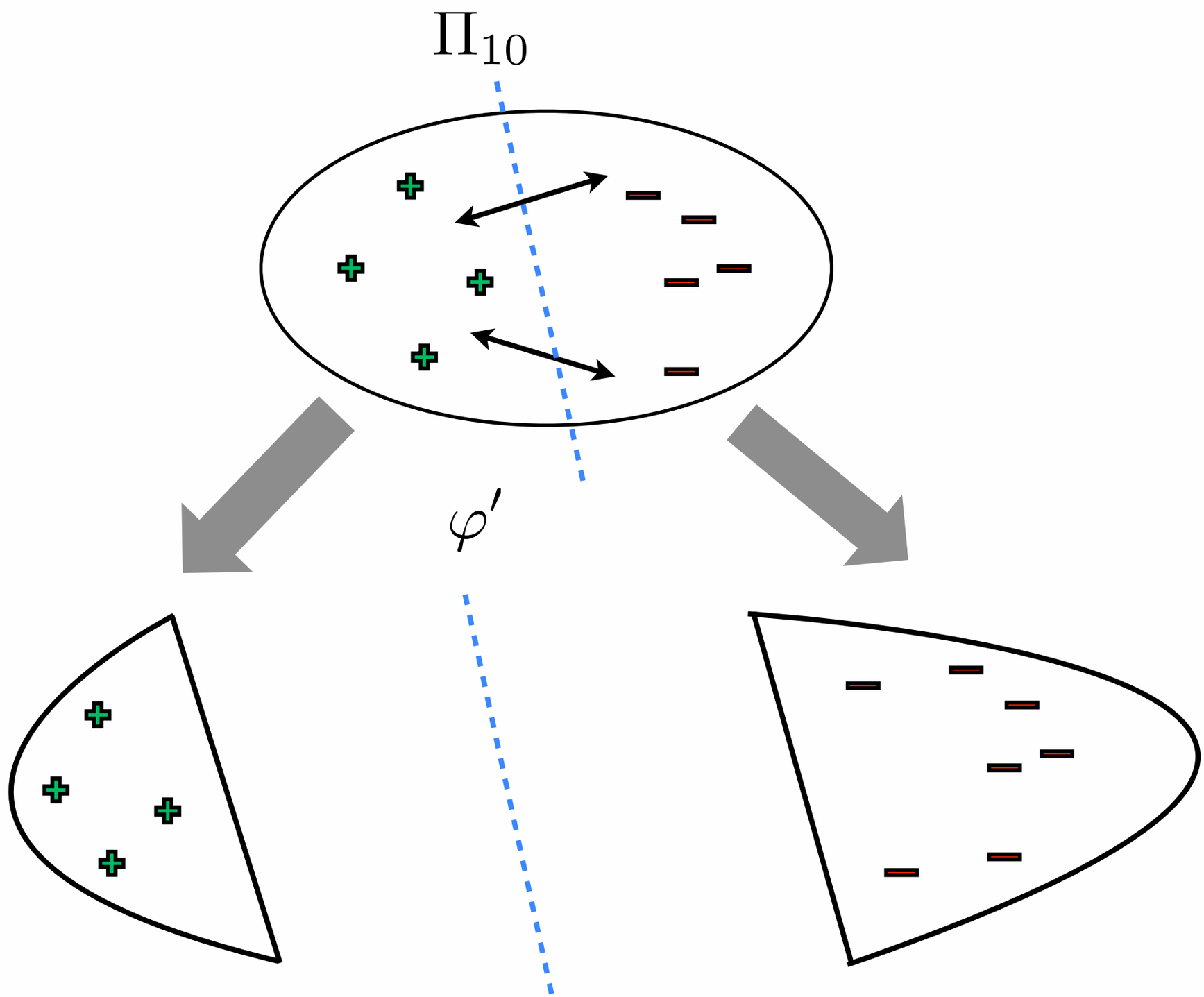
$$\Pi_{10} \Rightarrow \varphi(\Pi_0, \cdots, \Pi_9)$$

Learner ...



$$\Pi_{10} \Rightarrow \varphi(\Pi_0, \cdots, \Pi_9)$$

Learner ...



$$\Pi_{10} \Rightarrow \varphi(\Pi_0, \cdots, \Pi_9)$$

$$\varphi'(\Pi_0, \cdots, \Pi_9) \Rightarrow \Pi_{10}$$

Binary Search Tree Insertion ...

input features *output features*

$\Pi_0\ t : u \swarrow v$ $\Pi_8\ u = x$ Π_{10}

$\Pi_1\ t : u \searrow v$ $\Pi_9\ v = x$ $r : u \swarrow v$

$\Pi_2\ t : u \circlearrowleft v$ $\vdots$

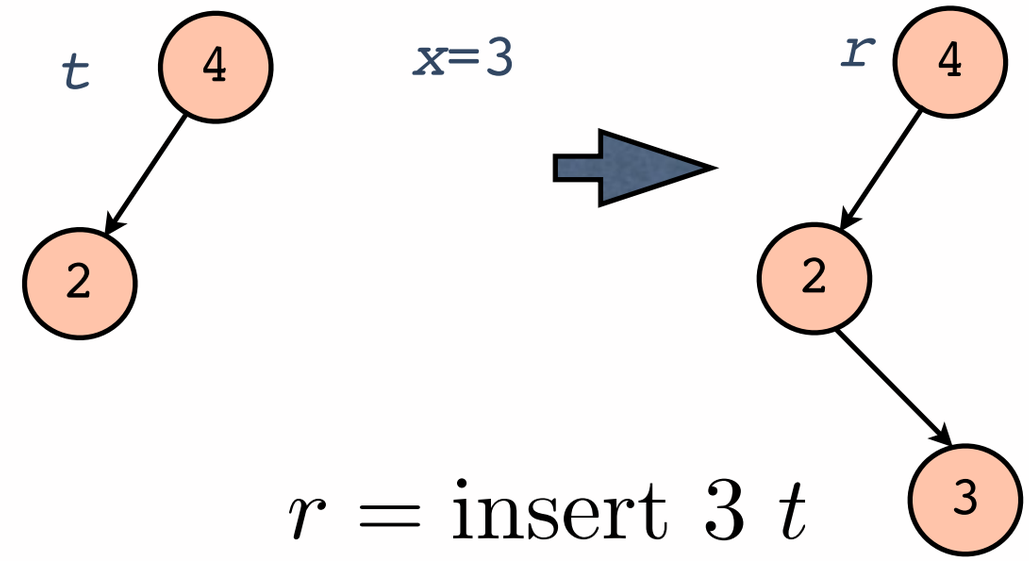
$\Pi_3\ t : v \swarrow u$

$\Pi_4\ t : v \searrow u$

$\Pi_5\ t : v \circlearrowleft u$

$\Pi_6\ t \dashrightarrow u$

$\Pi_7\ t \dashrightarrow v$



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

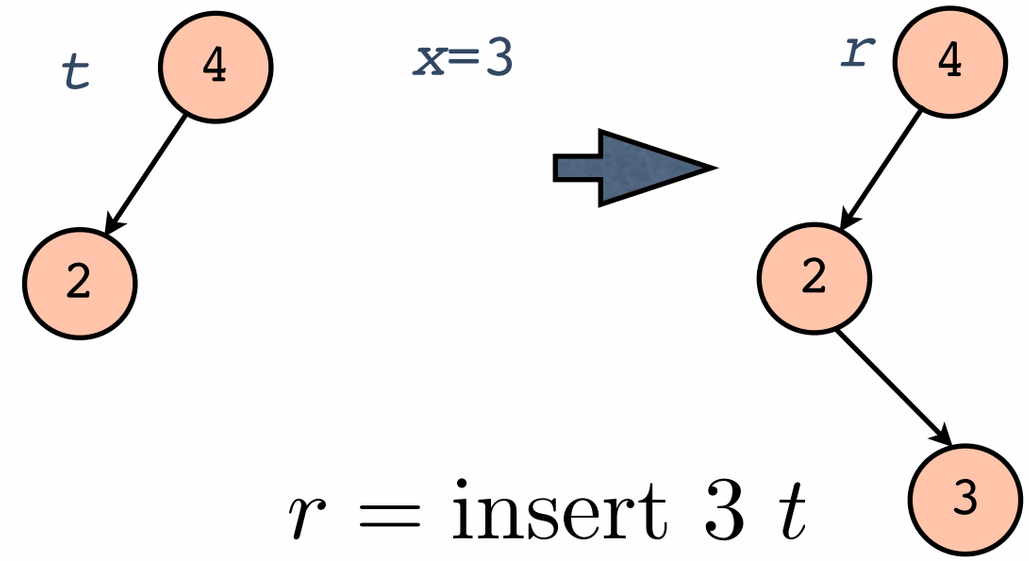
input features

	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

Binary Search Tree Insertion ...

input features *output features*

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



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

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

Binary Search Tree Insertion ...

input features *output features*

$\Pi_0\ t : u \swarrow v$ $\Pi_8\ u = x$ Π_{10}

$\Pi_1\ t : u \searrow v$ $\Pi_9\ v = x$ $r : u \swarrow v$

$\Pi_2\ t : u \circlearrowleft v$ $\vdots$

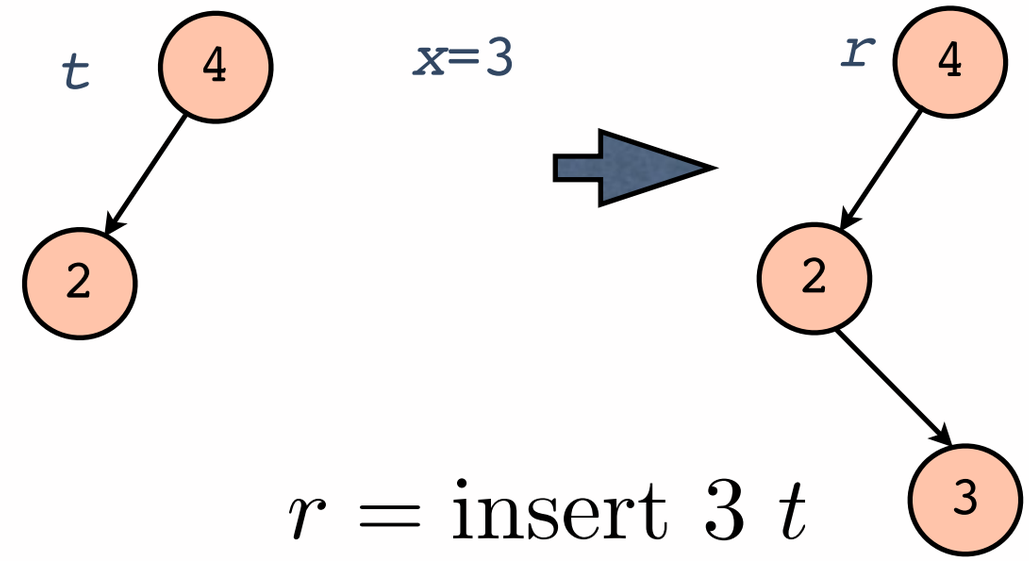
$\Pi_3\ t : v \swarrow u$

$\Pi_4\ t : v \searrow u$

$\Pi_5\ t : v \circlearrowleft u$

$\Pi_6\ t \dashrightarrow u$

$\Pi_7\ t \dashrightarrow v$



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

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

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

Binary Search Tree Insertion ...

input features *output features*

$\Pi_0\ t : u \swarrow v$ $\Pi_8\ u = x$ Π_{10}

$\Pi_1\ t : u \searrow v$ $\Pi_9\ v = x$ $r : u \swarrow v$

$\Pi_2\ t : u \circlearrowright v$ $\vdots$

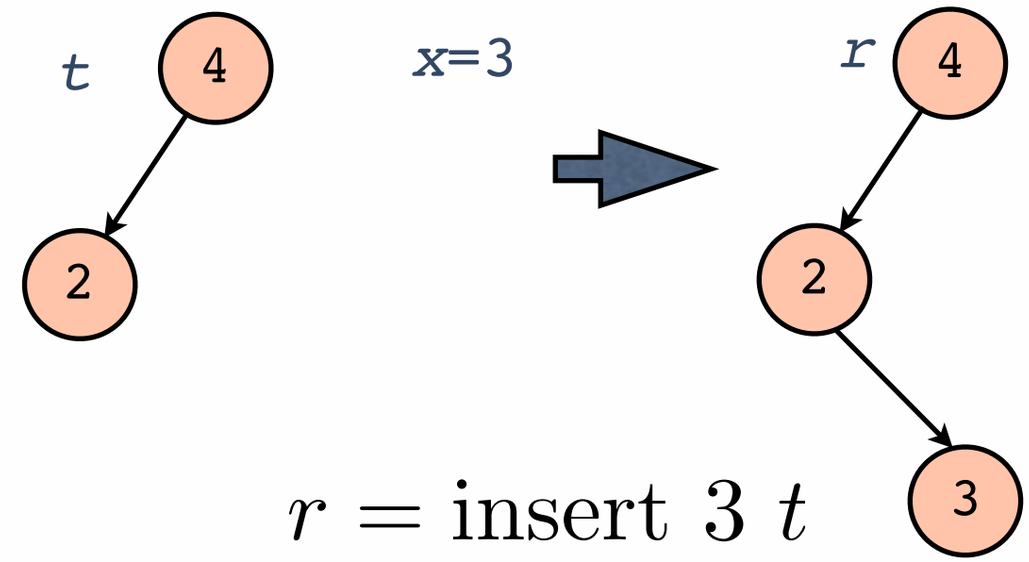
$\Pi_3\ t : v \swarrow u$

$\Pi_4\ t : v \searrow u$

$\Pi_5\ t : v \circlearrowright u$

$\Pi_6\ t \dashrightarrow u$

$\Pi_7\ t \dashrightarrow v$



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

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

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

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