Verification and Synthesis for Data Structures

Anders Miltner

Data Structures!

Key part of large software systems

Provides…

1. Data persistence
2. Mechanisms to restrict access to underlying data

And yet…

There's relatively little work surrounding them!

Why? 🤔

It's **HARD**

➡ 1. Data persistence

Must ensure the data structure is *always* safe

➡ 2. Mechanisms to restrict access to underlying data

Sometimes safety of the overall system relies on the data access restrictions

Proving a data structure maintains a proper invariant solves both of these problems!

This talk…

Hanoi

- Tool for data structure verification
- Focuses on the important task of *invariant generation*

Burst

- Tool for data structure synthesis
- Can synthesize recursive functions on data structures that ensure the invariants are upheld

But first…

What do I mean when I say *Data Structure*

SET

```
module type SET = sig
    type t
    val empty  : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
end
```

$$
\begin{aligned}
\forall i : \texttt{int}.\,\forall s : \texttt{t}.\,\forall s' : \texttt{t}.\quad & \neg(\text{lookup empty } i) \\
& \wedge\ (\text{lookup } (\text{insert } s\ i)\ i) \\
& \wedge\ \neg(\text{lookup } (\text{delete } s\ i)\ i)
\end{aligned}
$$

Implementing SET

```
module type SET = sig
    type t
    val empty  : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
end
```

$$\forall i : \texttt{int}.\forall s : \texttt{t}.\forall s' : \texttt{t}.\quad \neg(\text{lookup empty } i) \land (\text{lookup } (\text{insert } s\ i)\ \ i) \land \neg(\text{lookup } (\text{delete } s\ i)\ \ i)$$

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Hanoi
for Verification

Time to Verify!

$$\forall i : \texttt{int}. \forall s : \texttt{t}. \quad \neg(\texttt{lookup empty } i)$$
$$\wedge \ (\texttt{lookup (insert } s\ i)\ i)$$
$$\wedge \ \neg(\texttt{lookup (delete } s\ i)\ i)$$

```
t = int list
```

Time to Verify!

$$\forall i : \texttt{int}. \forall s : \texttt{t}. \quad \neg(\texttt{lookup empty } i) \land (\texttt{lookup } (\texttt{insert } s\ i)\ i) \land \neg(\texttt{lookup } (\texttt{delete } s\ i)\ i)$$

```
t = int list
```

s = [0;0]

i = 0

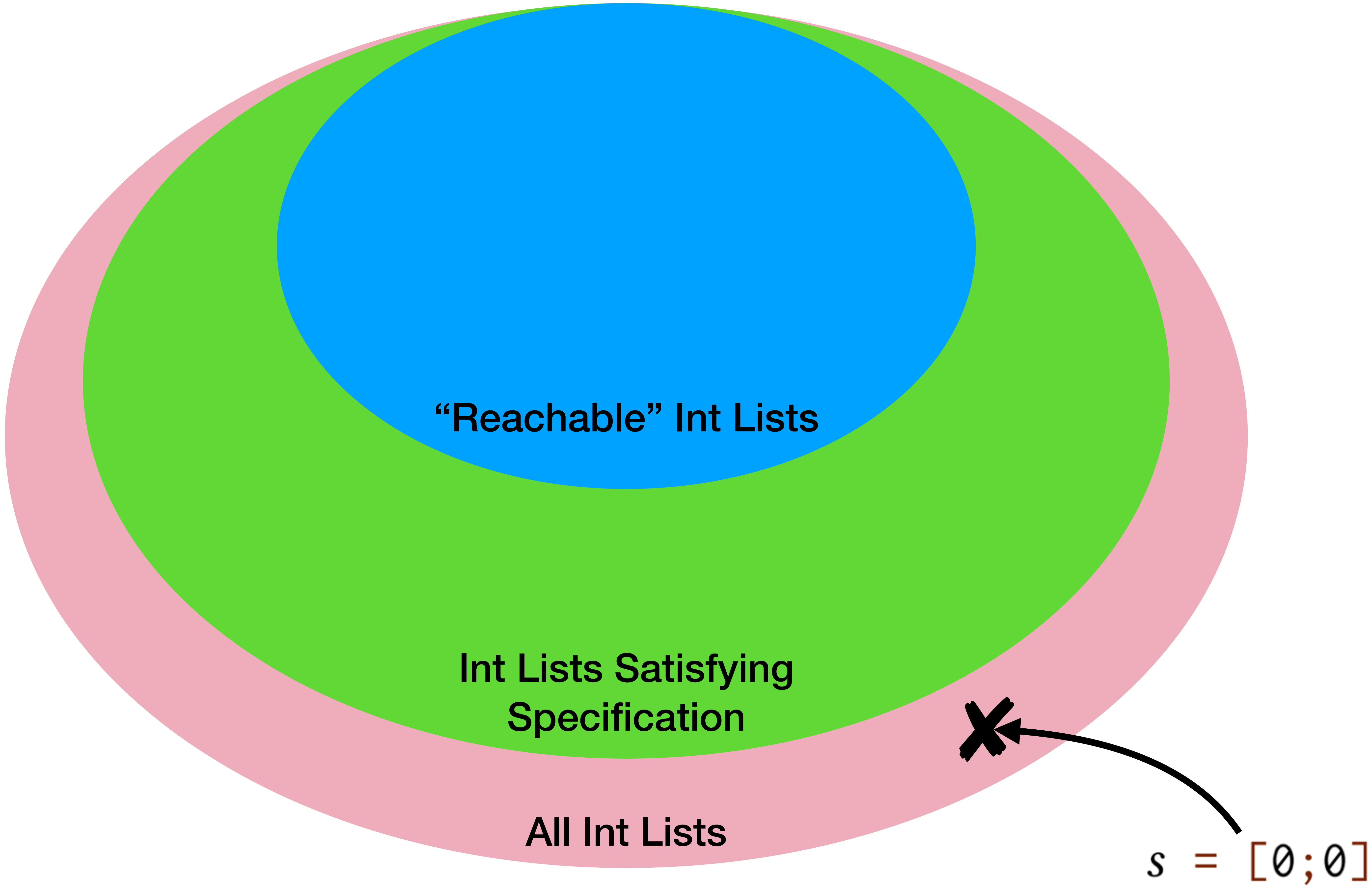
Is ListSet wrong?

```
module type SET = sig
    type t
    val empty  : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
end
```

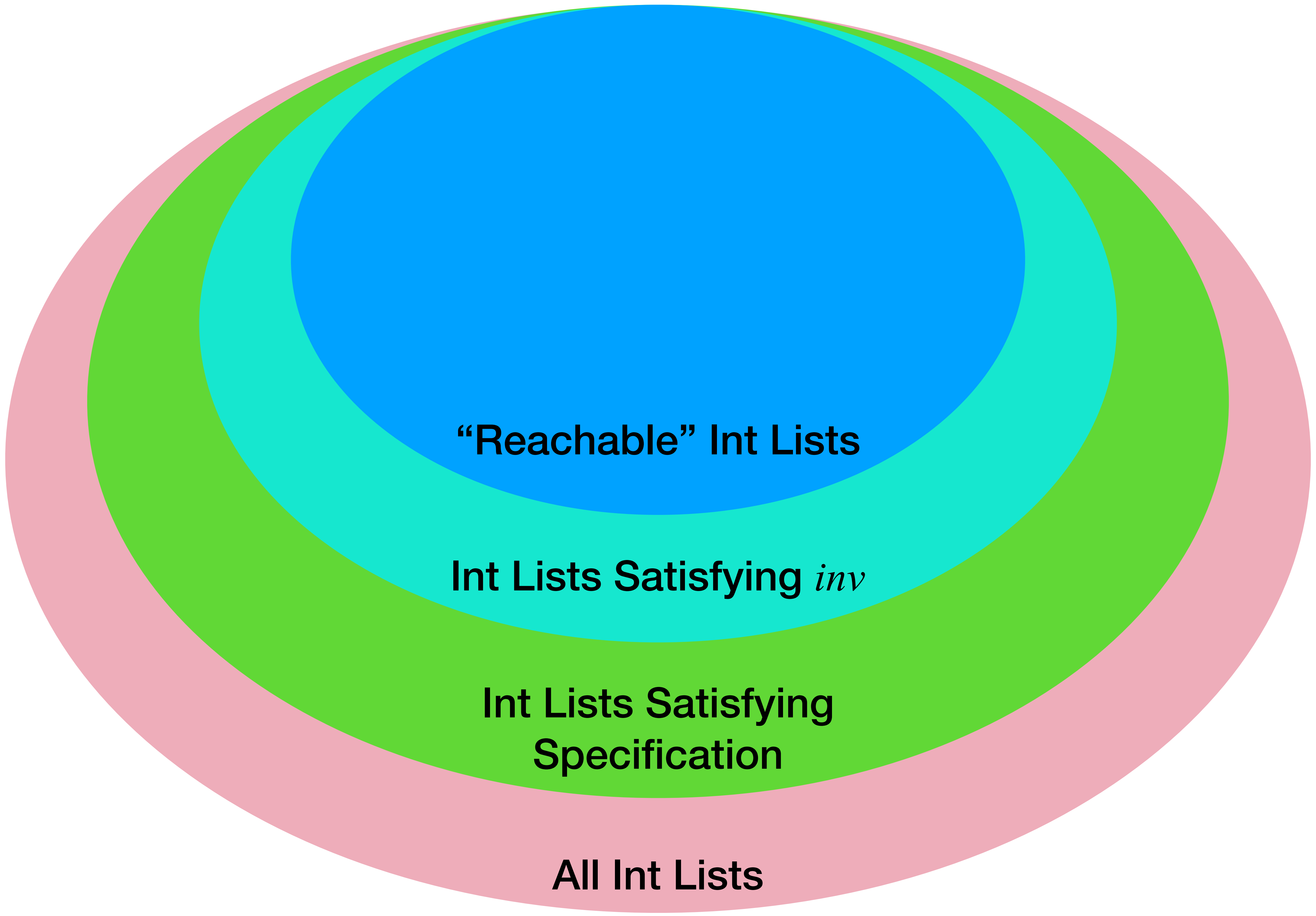
$$\forall i:\mathtt{int}.\forall s:\mathtt{t}.\forall s':\mathtt{t}.\quad \neg(\mathtt{lookup}\ \mathtt{empty}\ i) \wedge (\mathtt{lookup}\ (\mathtt{insert}\ s\ i)\ i) \wedge \neg(\mathtt{lookup}\ (\mathtt{delete}\ s\ i)\ i)$$

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

No, it isn't.


"Reachable" Int Lists
Int Lists Satisfying Specification
All Int Lists
s = [0;0]

How do you verify this?



How do you verify this?

Step 1: Find *inv*

How do you verify this?

Step 1: Find inv

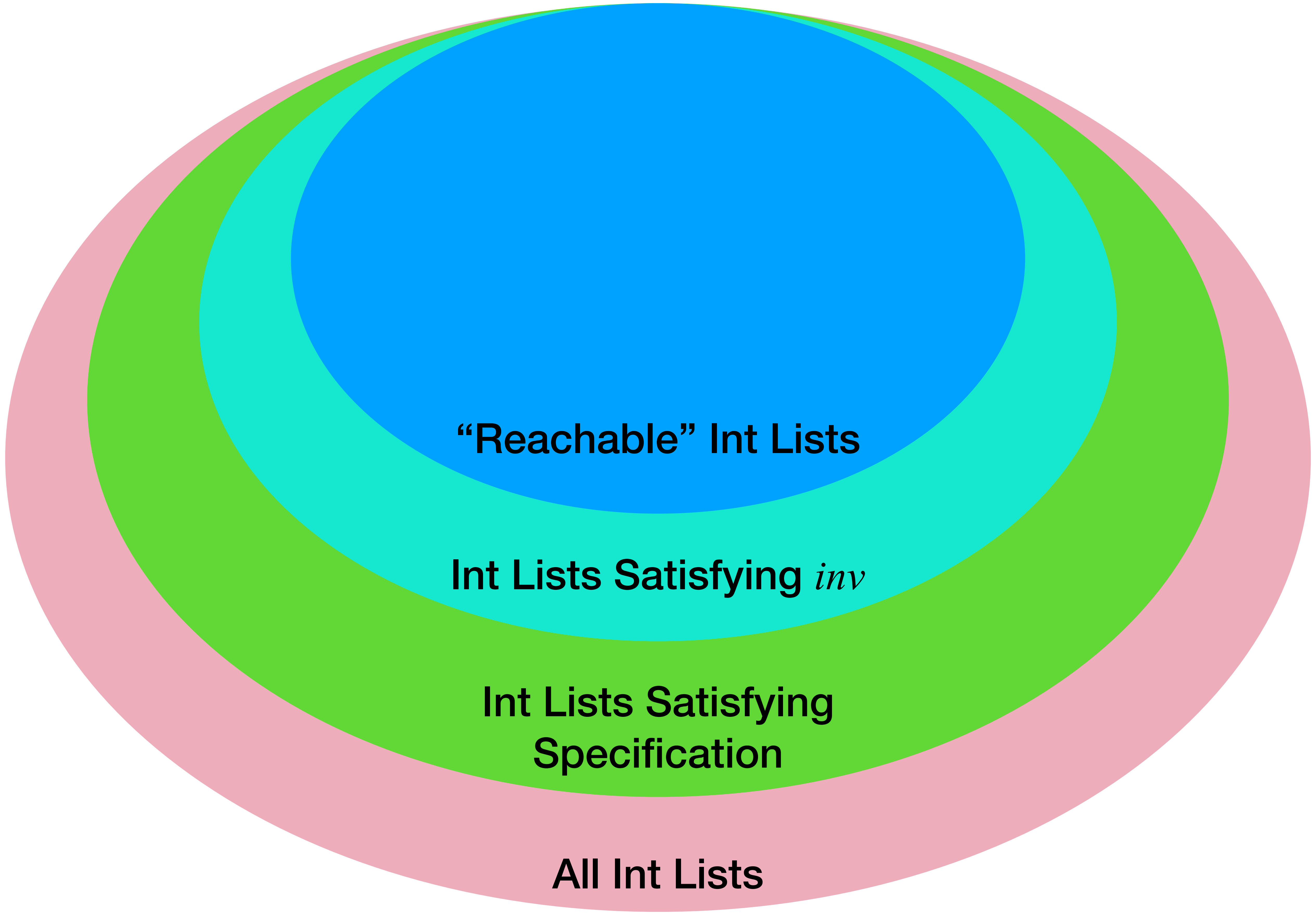
Step 2: Prove lists satisfying inv **satisfy the specification**

How do you verify this?

Step 1: Find *inv*

Step 2: Prove lists satisfying *inv* **satisfy the specification**

Step 3: Prove *inv* **is inductive — preserved by module operations**



How do you verify this? ✓

Step 1: Find *inv*

Step 2: Prove lists satisfying *inv* **satisfy the specification**

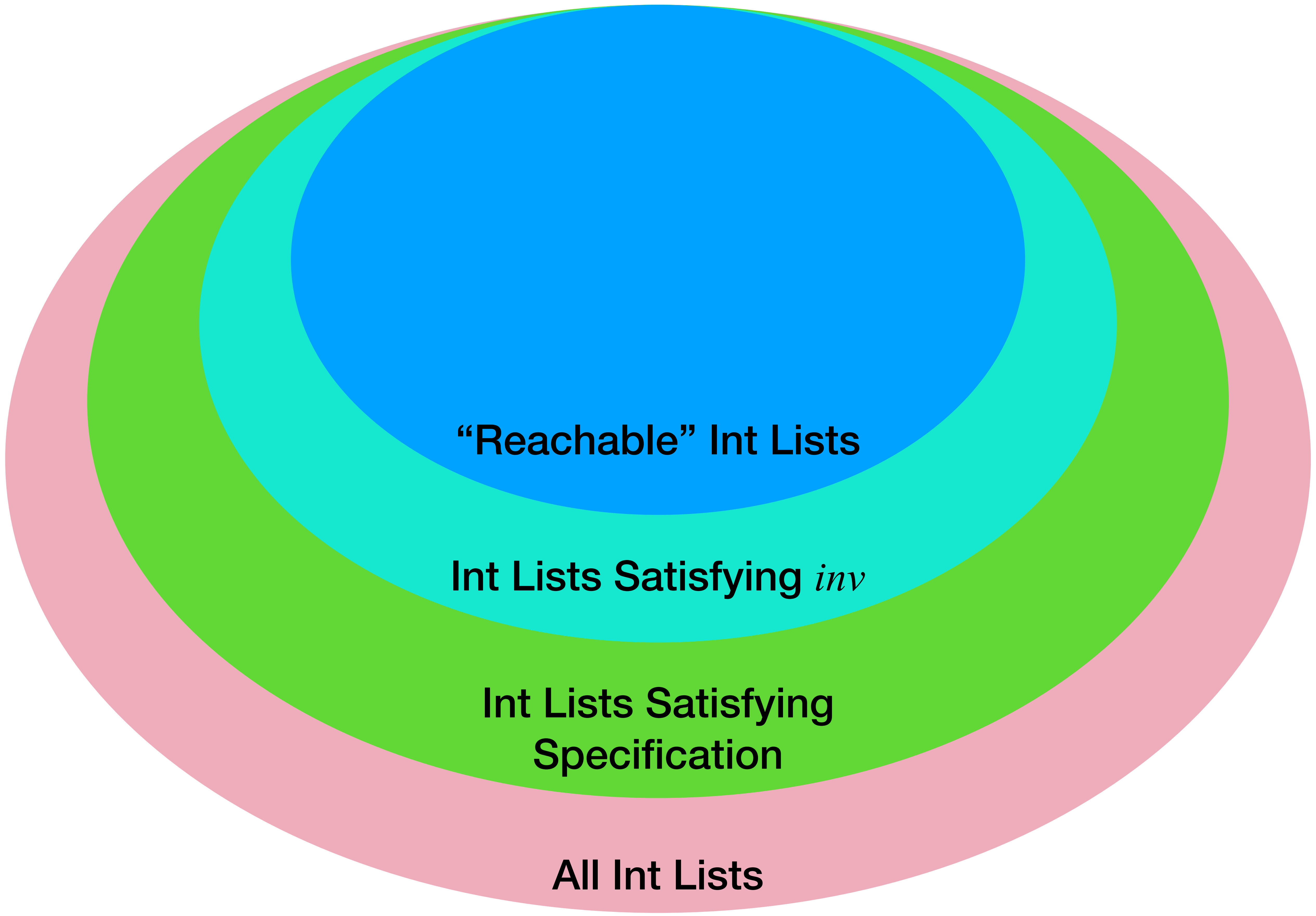
Step 3: Prove *inv* **is inductive — preserved by module operations**

How do you verify this?

Step 1: Find inv

Step 2: Prove lists satisfying inv **satisfy the specification**

Step 3: Prove inv **is inductive — preserved by module operations**

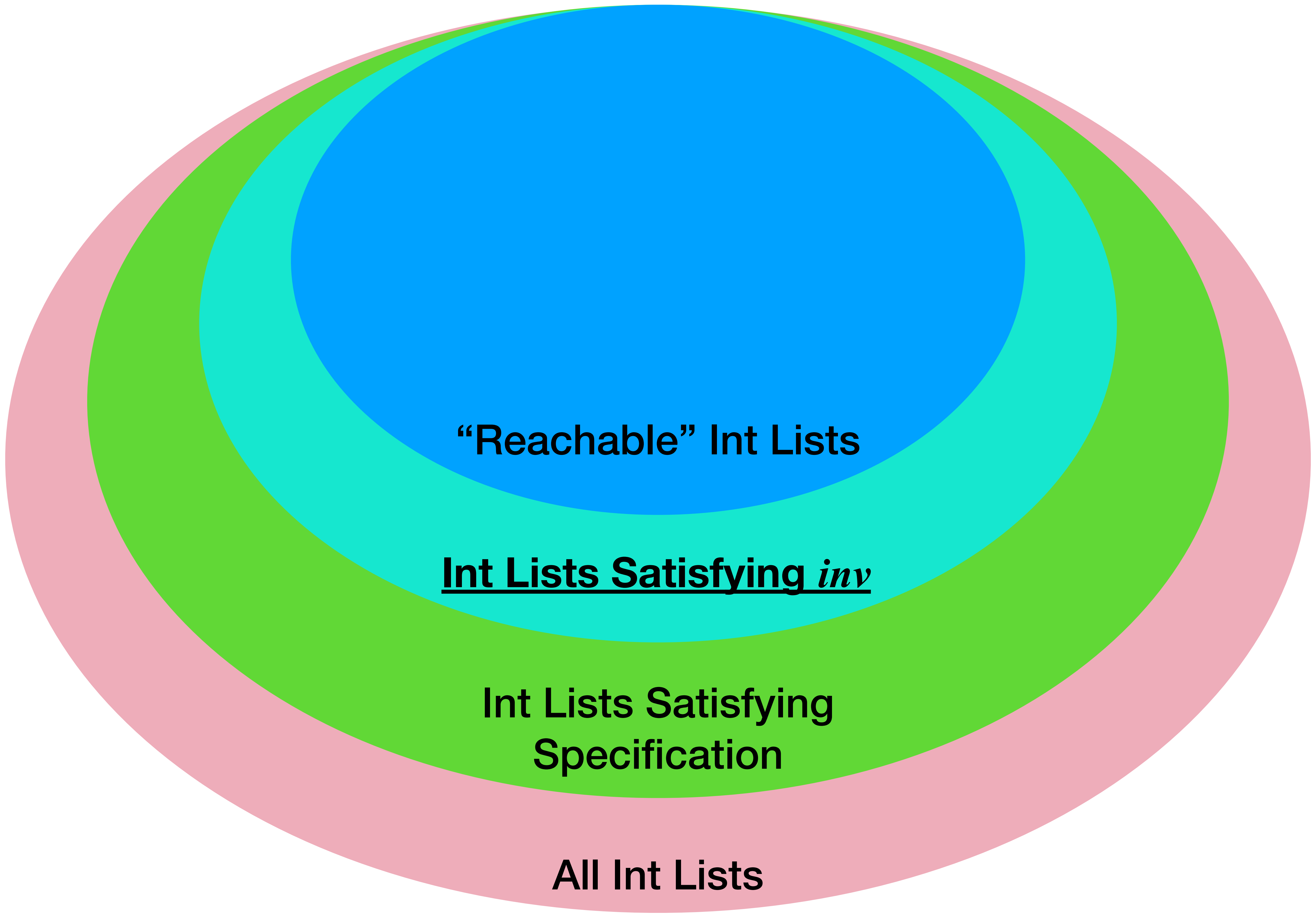


How do you verify this?

Step 1: Find *inv*

Step 2: Prove lists satisfying *inv* satisfy the specification

Step 3: Prove *inv* is inductive — preserved by module operations

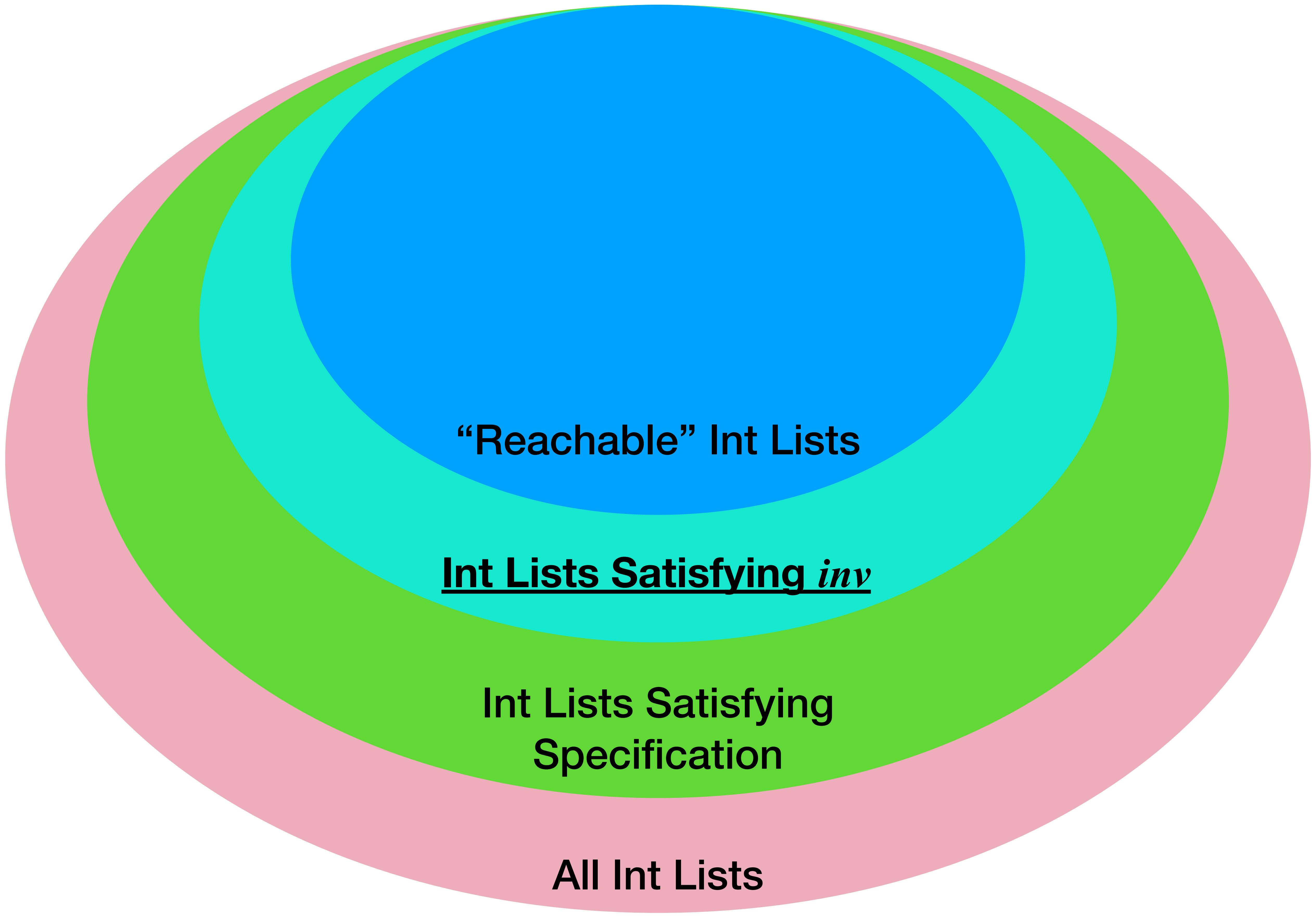


How do you verify this?

Step 1: Find *inv*

Step 2: Prove lists satisfying *inv* satisfy the specification

Step 3: Prove *inv* is inductive — preserved by module operations




```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

Time to Prove Correct!

Step 1: Find inv

Step 2: Prove lists satisfying inv **satisfy the specification**

Step 3: Prove inv **is inductive — preserved by module operations**

```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

Time to Prove Correct!

Step 1: Find inv

Step 2: Prove lists satisfying inv **satisfy the specification**

Step 3: Prove inv **is inductive — preserved by module operations**

$$\forall i : \texttt{int}.\forall s : \texttt{t}.\quad \neg(\texttt{lookup empty } i) \wedge (\texttt{lookup (insert } s\ i)\ i) \wedge \neg(\texttt{lookup (delete } s\ i)\ i)$$

```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

Time to Prove Correct!

Step 1: Find inv

$$\forall i : \texttt{int}.\ \forall s : \texttt{t}.\quad \neg(\texttt{lookup empty } i) \wedge (\texttt{lookup } (\texttt{insert } s\ i)\ i) \wedge \neg(\texttt{lookup } (\texttt{delete } s\ i)\ i)$$

Step 2: Prove lists satisfying inv **satisfy the specification**

$$\forall i : \texttt{int}.\ \forall s : \texttt{t}.\quad (inv\ s) \Rightarrow \neg(\texttt{lookup empty } i) \wedge (\texttt{lookup } (\texttt{insert } s\ i)\ i) \wedge \neg(\texttt{lookup } (\texttt{delete } s\ i)\ i)$$

Step 3: Prove inv **is inductive — preserved by module operations**

```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

Time to Prove Correct!

Step 1: Find *inv*

Step 2: Prove lists satisfying *inv* **satisfy the specification**

Step 3: Prove *inv* **is inductive — preserved by module operations**

```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

Time to Prove Correct!

Step 1: Find inv

Step 2: Prove lists satisfying inv **satisfy the specification**

Step 3: Prove inv **is inductive — preserved by module operations**

```
module type SET = sig
    type t
    val empty  : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
end
```

```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

Time to Prove Correct!

Step 1: Find inv

Step 2: Prove lists satisfying inv **satisfy the specification**

Step 3: Prove inv **is inductive — preserved by module operations**

```
module type SET = sig
    type t
    val empty  : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
end
```

```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

Time to Prove Correct!

Step 1: Find *inv*

Step 2: Prove lists satisfying *inv* satisfy the specification

Step 3: Prove *inv* is inductive — preserved by module operations

```
module type SET = sig
    type t
    val empty  : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
end
```

? *inv* [] = true

```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

Time to Prove Correct!

Step 1: Find *inv*

Step 2: Prove lists satisfying *inv* satisfy the specification

Step 3: Prove *inv* is inductive — preserved by module operations

```
module type SET = sig
    type t
    val empty  : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
end
```

✓ *inv* [] = true

```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

Time to Prove Correct!

Step 1: Find *inv*

Step 2: Prove lists satisfying *inv* satisfy the specification

Step 3: Prove *inv* is inductive — preserved by module operations

```
module type SET = sig
    type t
    val empty  : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
end
```

✔ inv `[] = true`

? $\forall s.\forall i.inv$ `s = true` $\Rightarrow$ inv `(insert s i) = true`

```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

Time to Prove Correct!

Step 1: Find inv

Step 2: Prove lists satisfying inv **satisfy the specification**

Step 3: Prove inv **is inductive — preserved by module operations**

```
module type SET = sig
    type t
    val empty  : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
end
```

✓ inv `[] = true`

✓ $\forall s. \forall i. inv$ `s = true` $\Rightarrow$ inv `(insert s i) = true`

```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

Time to Prove Correct!

Step 1: Find inv

Step 2: Prove lists satisfying inv **satisfy the specification**

Step 3: Prove inv **is inductive — preserved by module operations**

```
module type SET = sig
    type t
    val empty  : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
end
```

✔ inv `[] = true`

✔ $\forall s. \forall i. inv$ `s = true` $\Rightarrow$ inv `(insert s i) = true`

⋮

```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

Time to Prove Correct!

Step 1: Find *inv*

Step 2: Prove lists satisfying *inv* satisfy the specification

Step 3: Prove *inv* is inductive — preserved by module operations

```
module type SET = sig
    type t
    val empty  : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
end
```

✔ inv `[] = true`

✔ $\forall s. \forall i. inv$ `s = true` $\Rightarrow$ inv `(insert s i) = true`

⋮

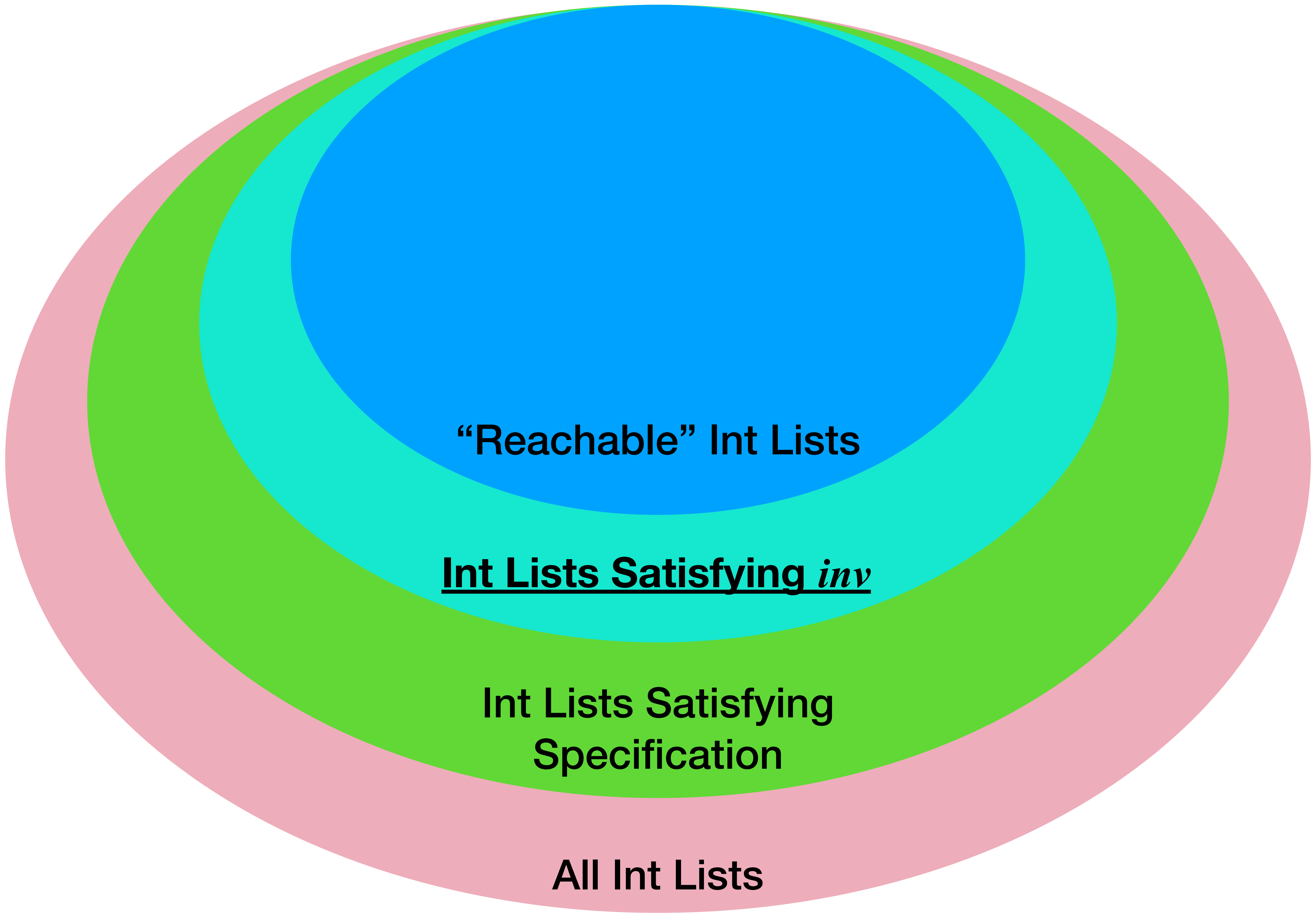
```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

Time to Prove Correct!

Step 1: Find *inv*

Step 2: Prove lists satisfying *inv* satisfy the specification

Step 3: Prove *inv* is inductive — preserved by module operations




```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

```ocaml
module type SET = sig
    type t
    val empty : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
end
```

```ocaml
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

$$\forall i : \mathtt{int}. \forall s : \mathtt{t}. \quad \neg(\mathtt{lookup\ empty}\ i) \wedge (\mathtt{lookup}\ (\mathtt{insert}\ s\ i)\ i) \wedge \neg(\mathtt{lookup}\ (\mathtt{delete}\ s\ i)\ i)$$

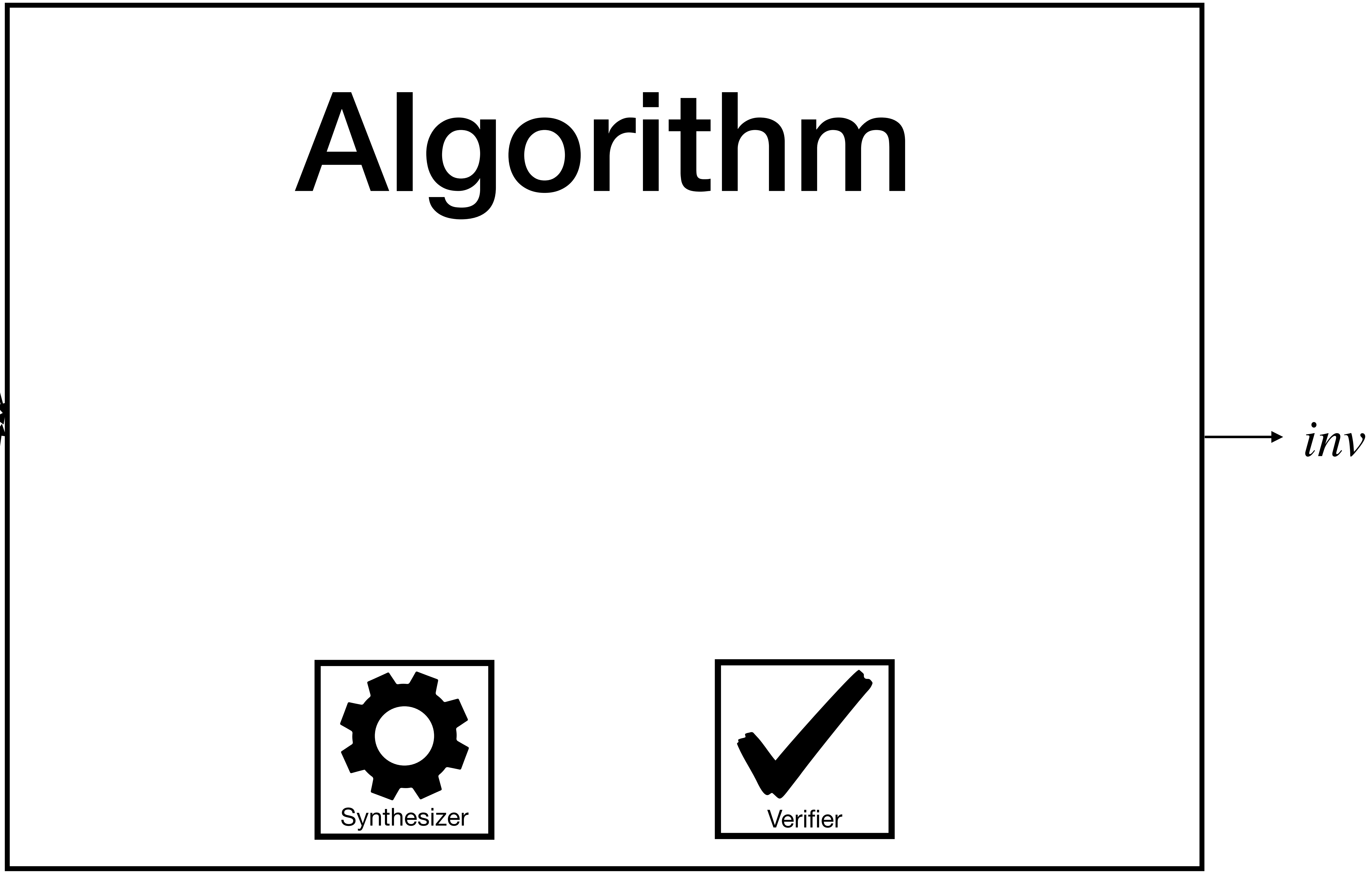
Hanoi

```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

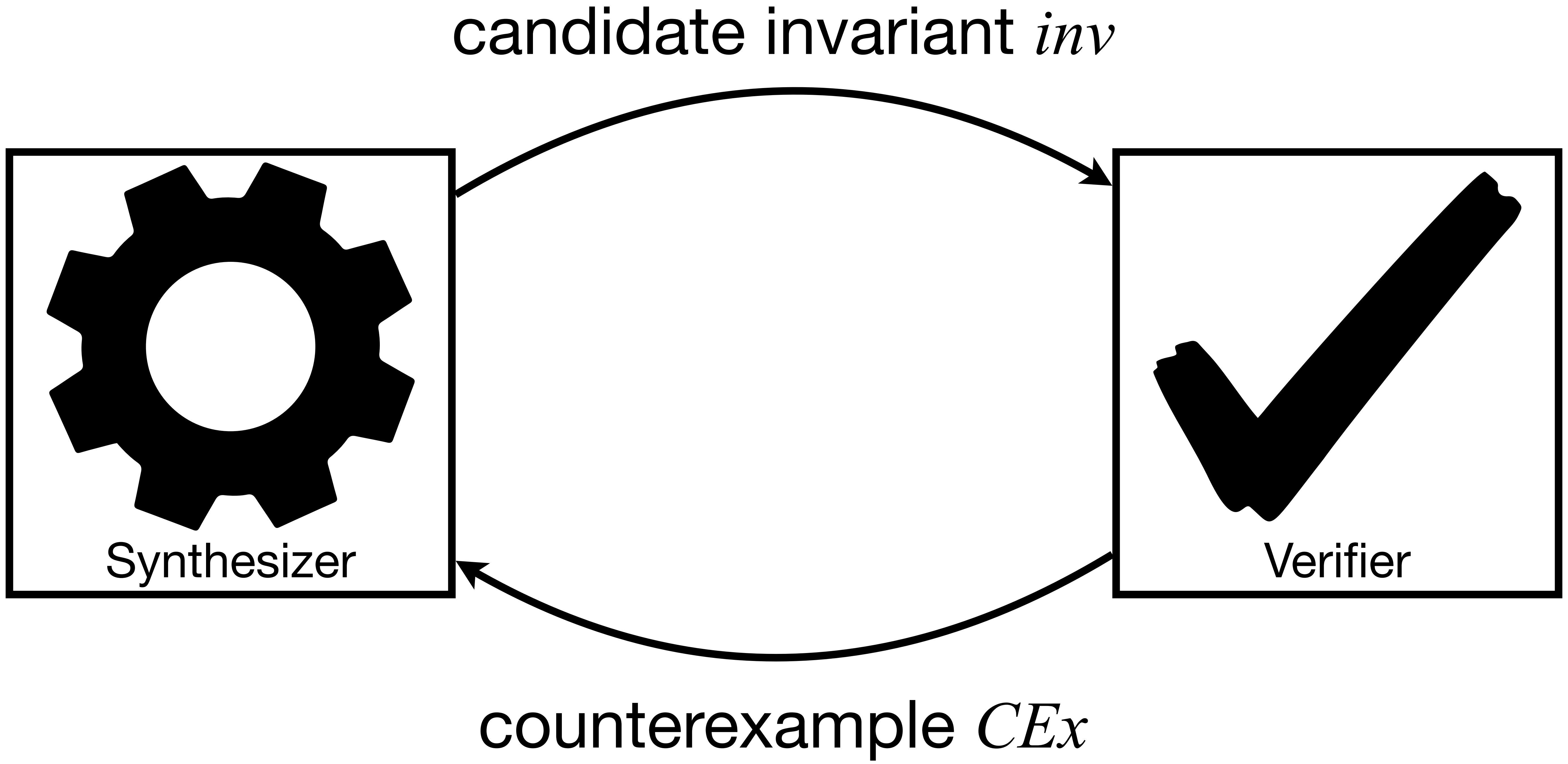
```
module type SET = sig
    type t
    val empty : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
end
```

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

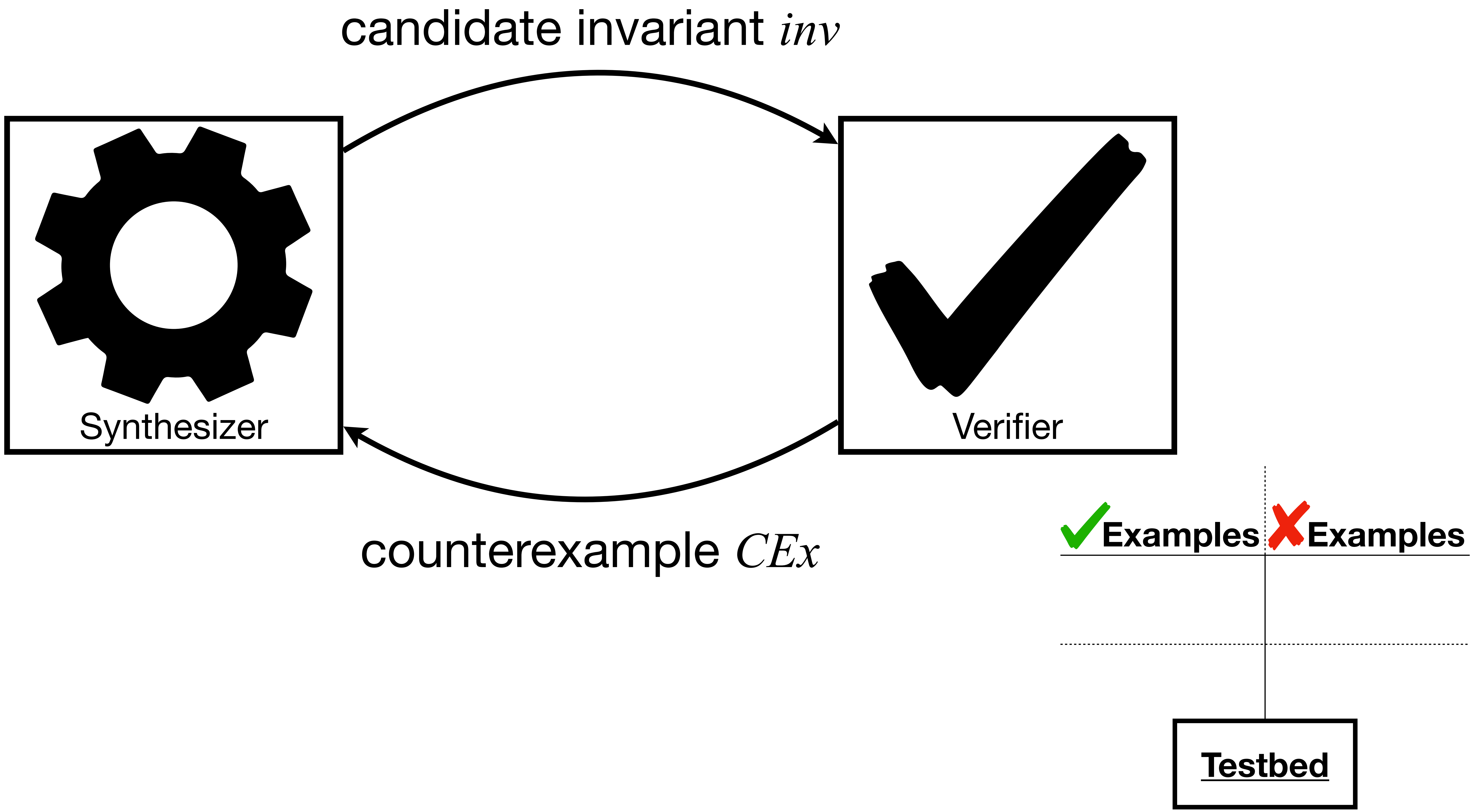
$$
\begin{aligned}
\forall i : \texttt{int}.\, \forall s : \texttt{t}.\quad & \neg(\texttt{lookup empty } i) \\
& \wedge\ (\texttt{lookup } (\texttt{insert } s\ i)\ i) \\
& \wedge\ \neg(\texttt{lookup } (\texttt{delete } s\ i)\ i)
\end{aligned}
$$



Algorithm

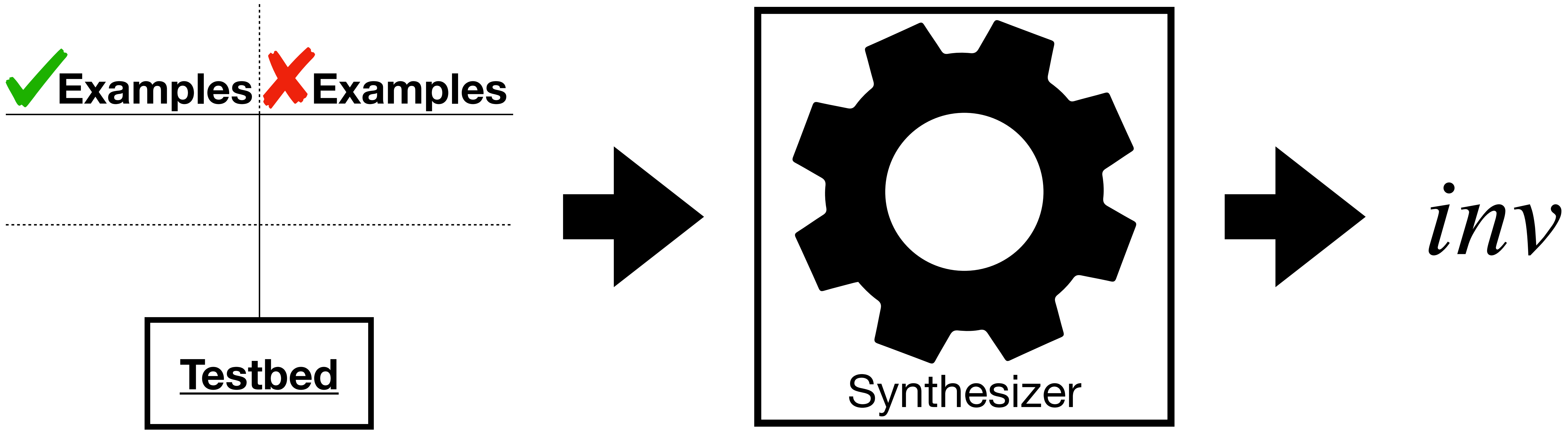

candidate invariant *inv*
Synthesizer
Verifier
counterexample *CEx*

Algorithm



candidate invariant inv
Synthesizer
Verifier
counterexample CEx
Examples
Examples
Testbed

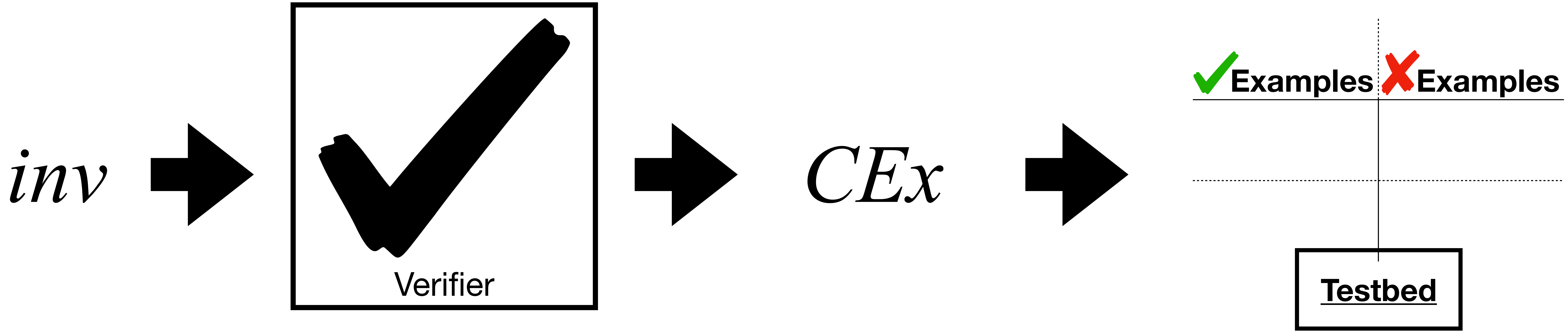
Synthesizer



$$\mathit{inv}(\checkmark) = \texttt{true}$$

$$\mathit{inv}(\text{✗}) = \texttt{false}$$

Verifier

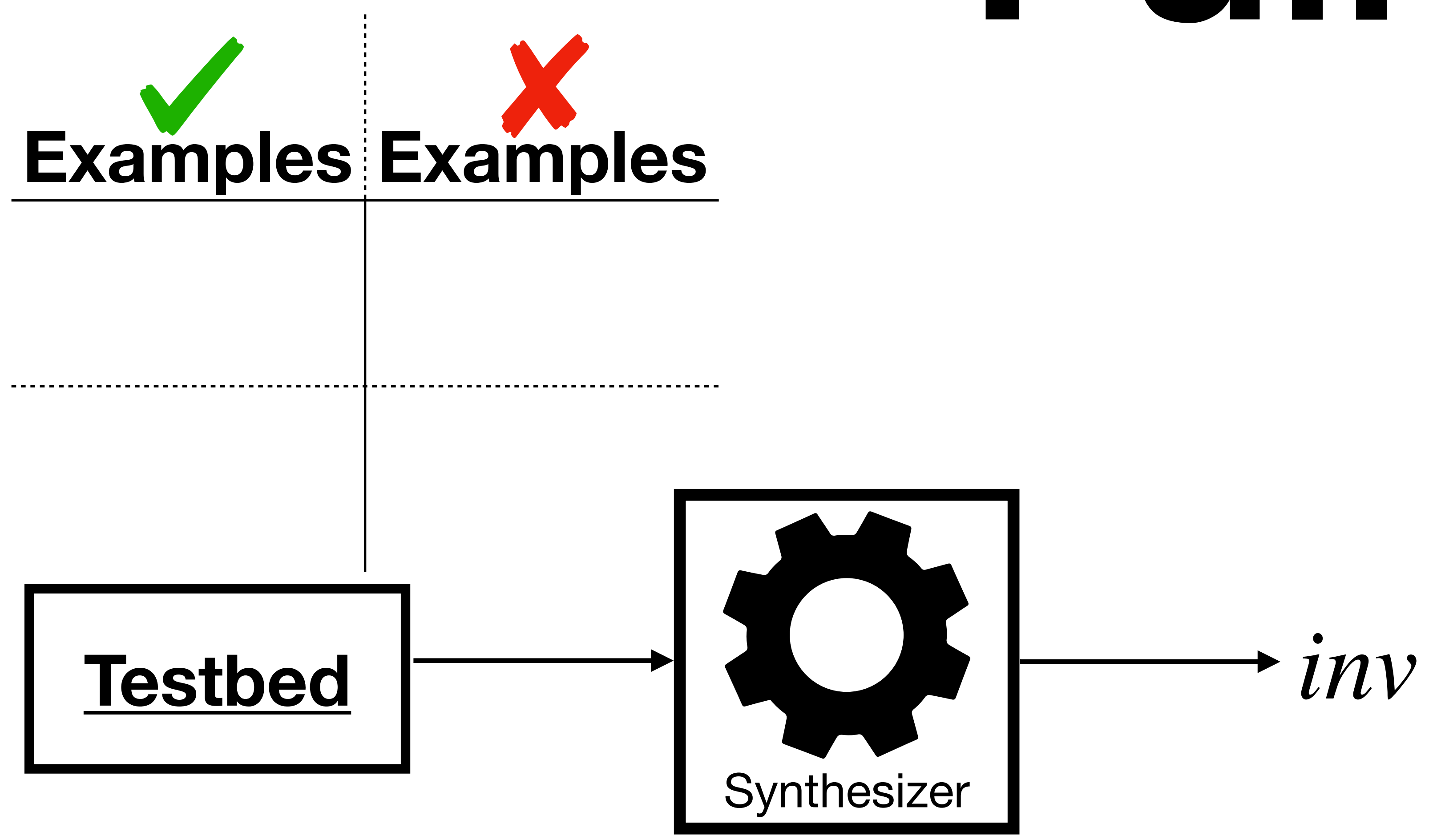


Full Algorithm

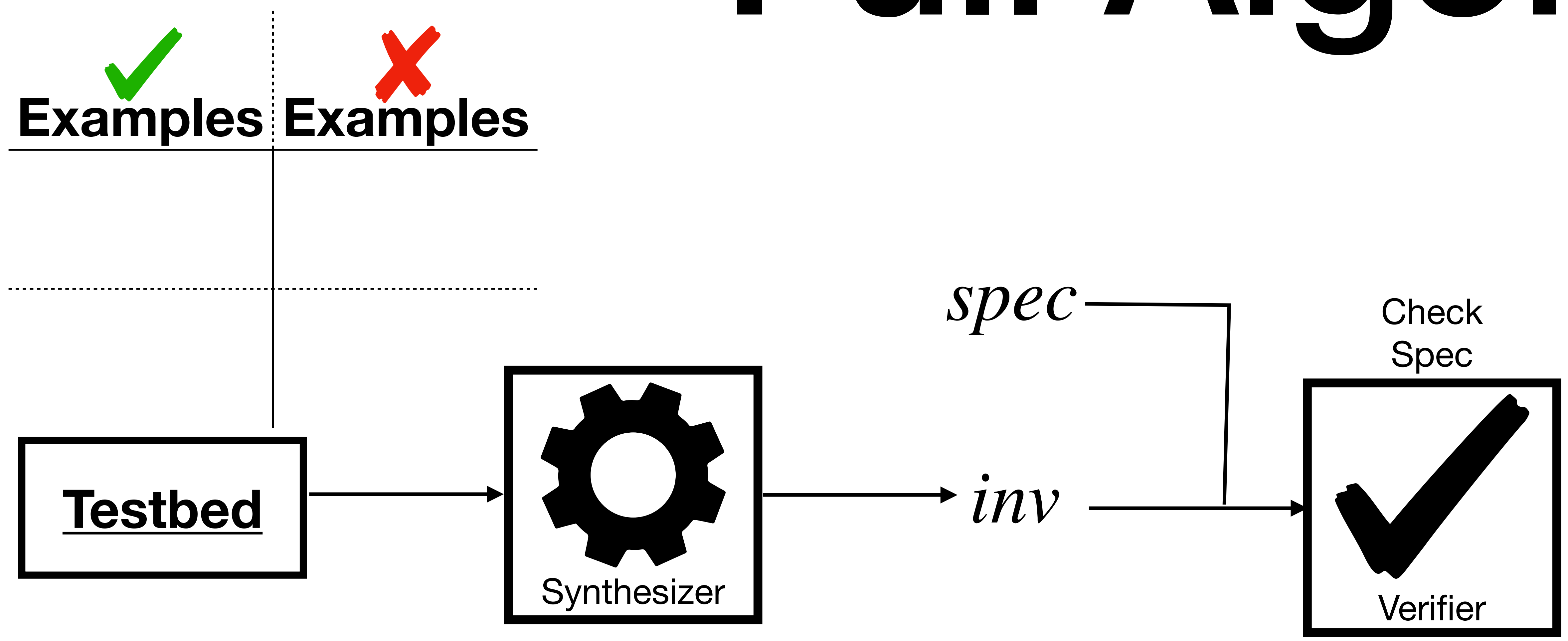
✓ Examples	✗ Examples

Testbed

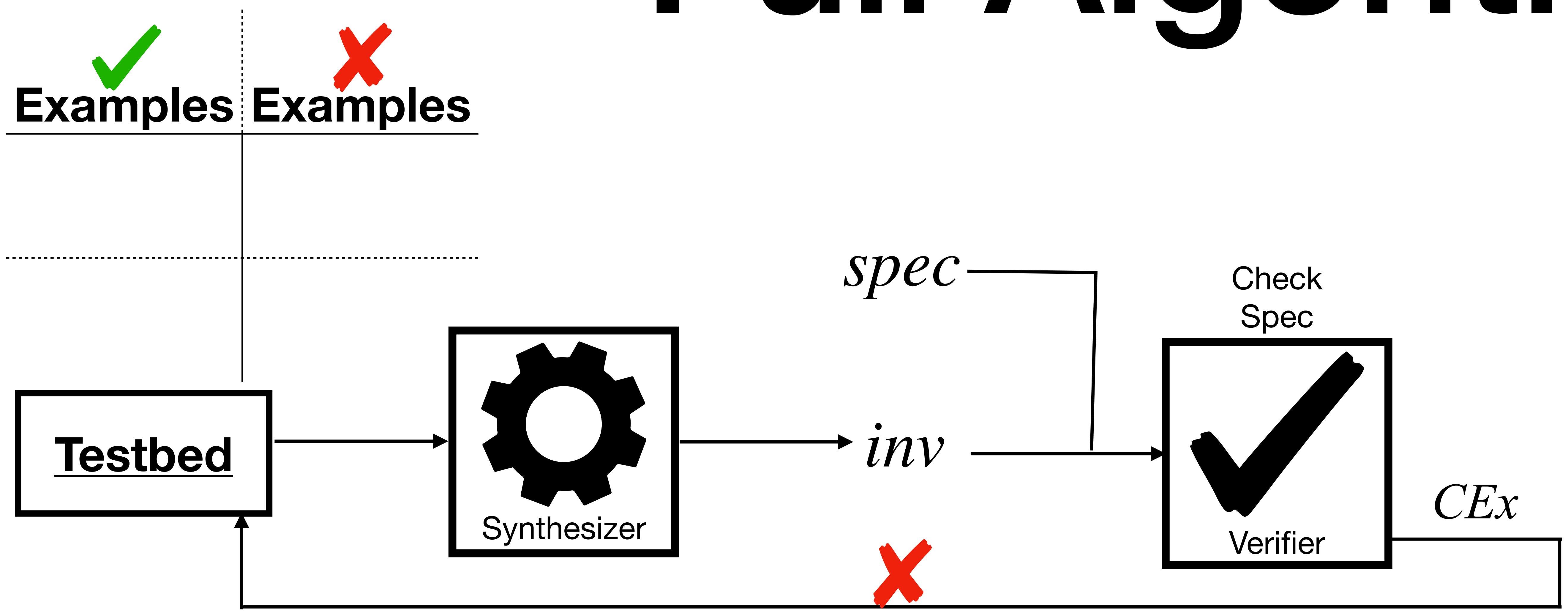
Full Algorithm



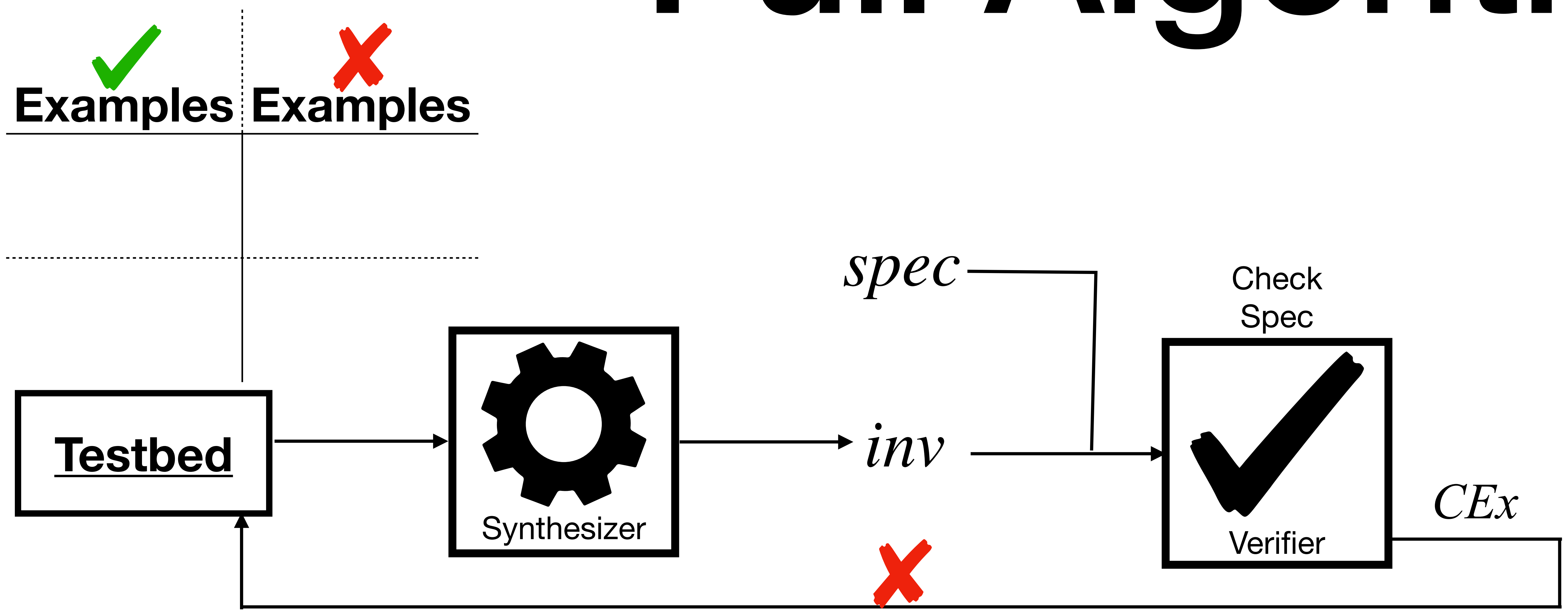
Full Algorithm



Full Algorithm



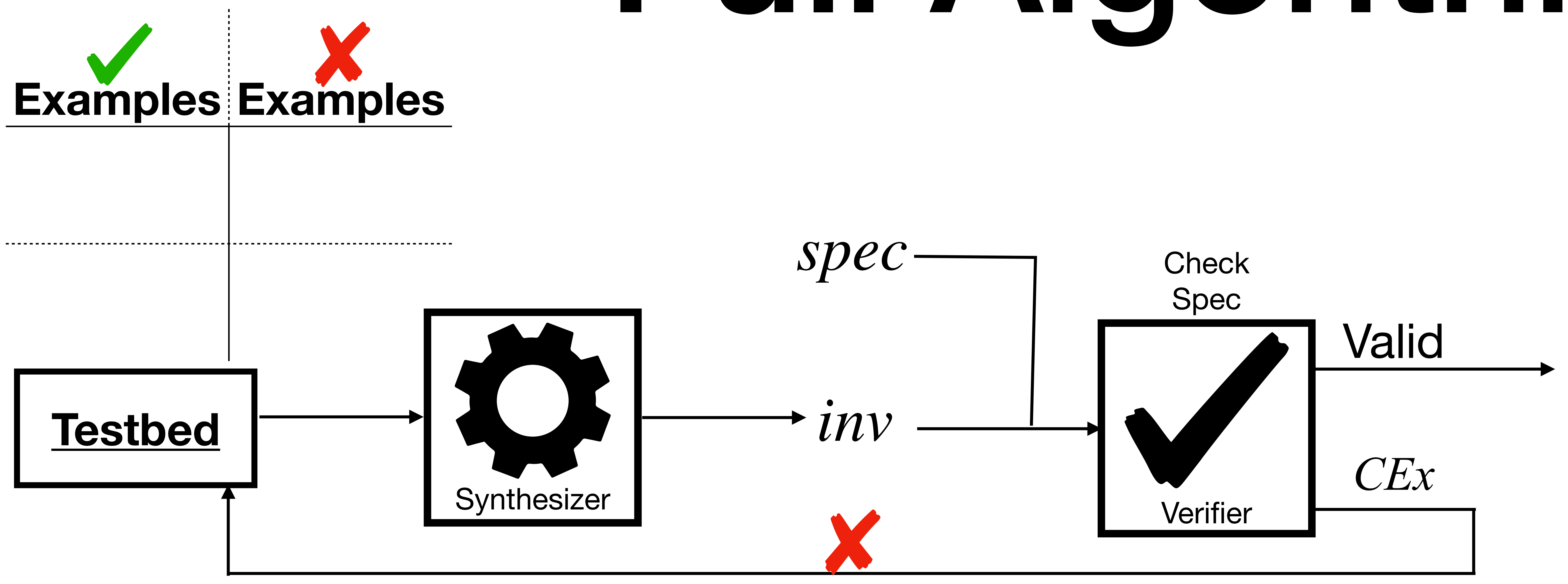
Full Algorithm



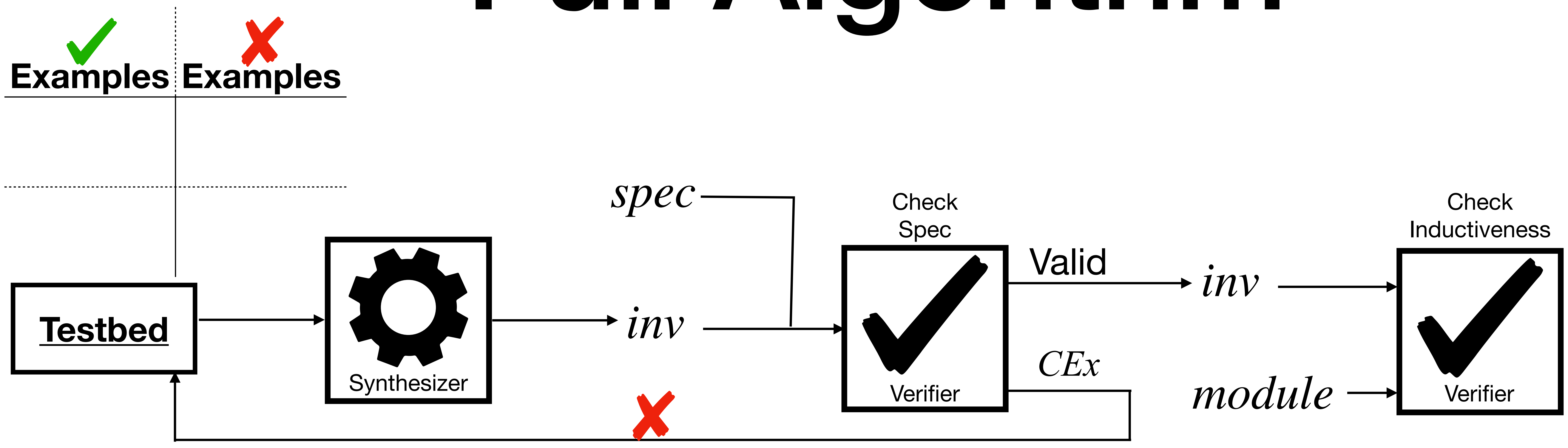
$$\forall i : \texttt{int}.\forall s : \texttt{t}. \quad (inv\ s) \Rightarrow \neg(\texttt{lookup empty}\ i) \land (\texttt{lookup}\ (\texttt{insert}\ s\ i)\ i) \land \neg(\texttt{lookup}\ (\texttt{delete}\ s\ i)\ i)$$

s = [0;0]
i = 0

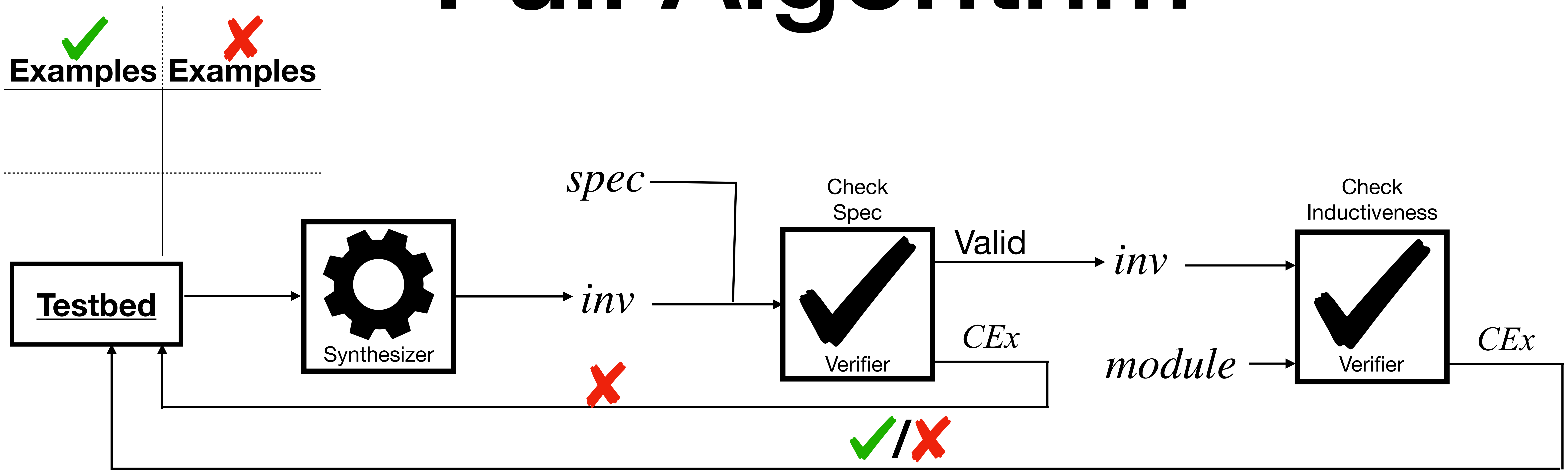
Full Algorithm



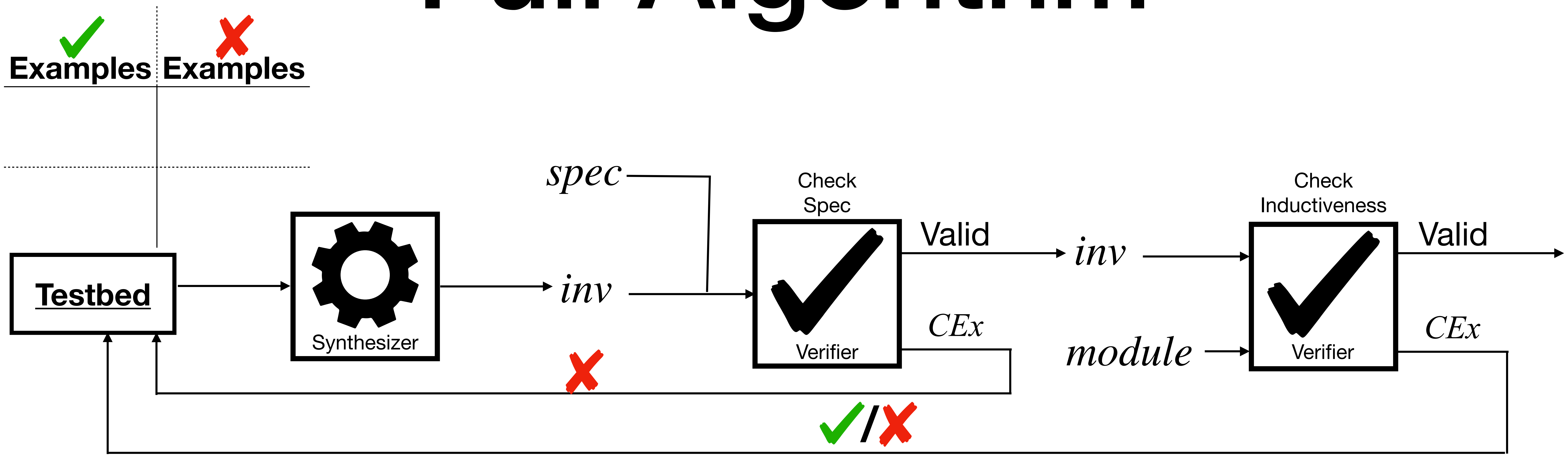
Full Algorithm



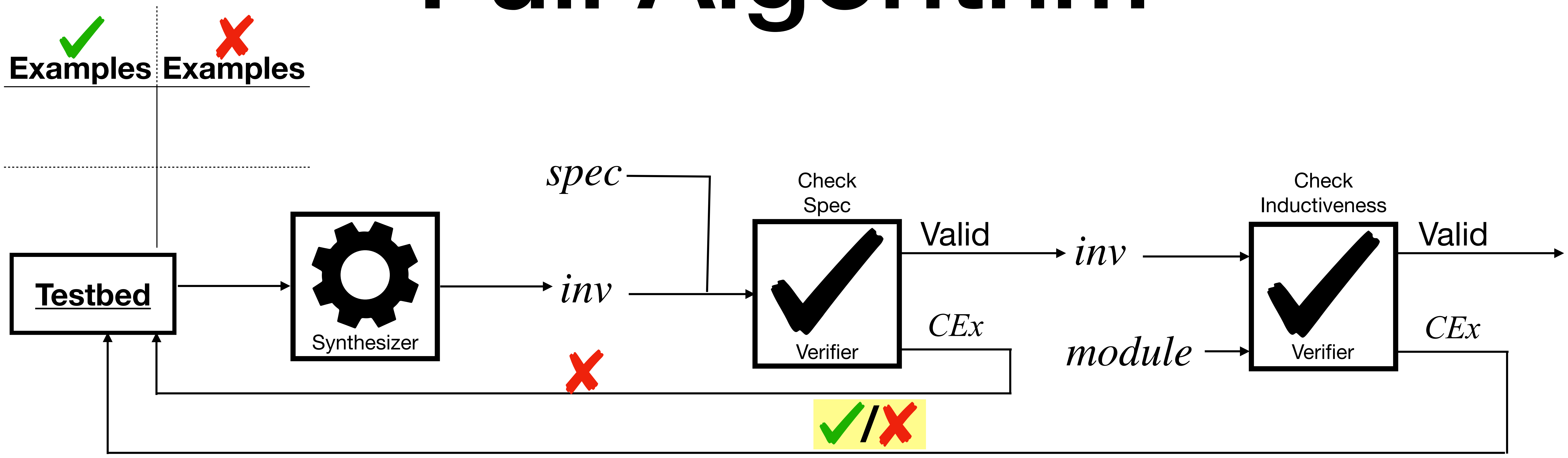
Full Algorithm



Full Algorithm



Full Algorithm



Verifying Inductiveness

inv `l = length l < 2`

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Verifying Inductiveness

inv `l = length l < 2`

? inv `[] = true`

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Verifying Inductiveness

```
inv l = length l < 2
```

✓ `inv [] = true`

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Verifying Inductiveness

inv `l = length l < 2`

✔ inv `[] = true`

? $\forall s. \forall i. inv$ `s = true` $\Rightarrow$
inv `(insert s i) = true`

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Verifying Inductiveness

inv `l = length l < 2`

✓ inv `[] = true`

✗ $\forall s.\forall i.inv$ `s = true` $\Rightarrow$
inv `(insert s i) = true`

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Verifying Inductiveness

inv l = length l < 2

✔ *inv* [] = true

✘ ∀s.∀i.*inv* s = true ⇒
inv (insert s i) = true

```
module ListSet : SET = struct
  type t = int list
  let empty = []
  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)
  let insert l x =
    if (lookup l x) then l else (x :: l)
  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

insert [1] 0 = [0;1]

Verifying Inductiveness

```
inv l = length l < 2
```

✓ *inv* `[] = true`

✗ $\forall s.\forall i.$*inv* `s = true` $\Rightarrow$
inv `(insert s i) = true`

```
insert [1] 0 = [0;1]
```

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Verifying Inductiveness

inv `l = length l < 2`

✓ inv `[] = true`

✗ $\forall s. \forall i. inv$ `s = true` $\Rightarrow$
inv `(insert s i) = true`

`insert [1] 0 = [0;1]`

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Verifying Inductiveness

inv `l = length l < 2`

✔ inv `[] = true`

✘ $\forall s. \forall i. inv$ `s = true` $\Rightarrow$
inv `(insert s i) = true`

`insert [1] 0 = [0;1]`

```
module ListSet : SET = struct
  type t = int list
  let empty = []
  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)
  let insert l x =
    if (lookup l x) then l else (x :: l)
  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Verifying Inductiveness

inv `l = length l < 2`

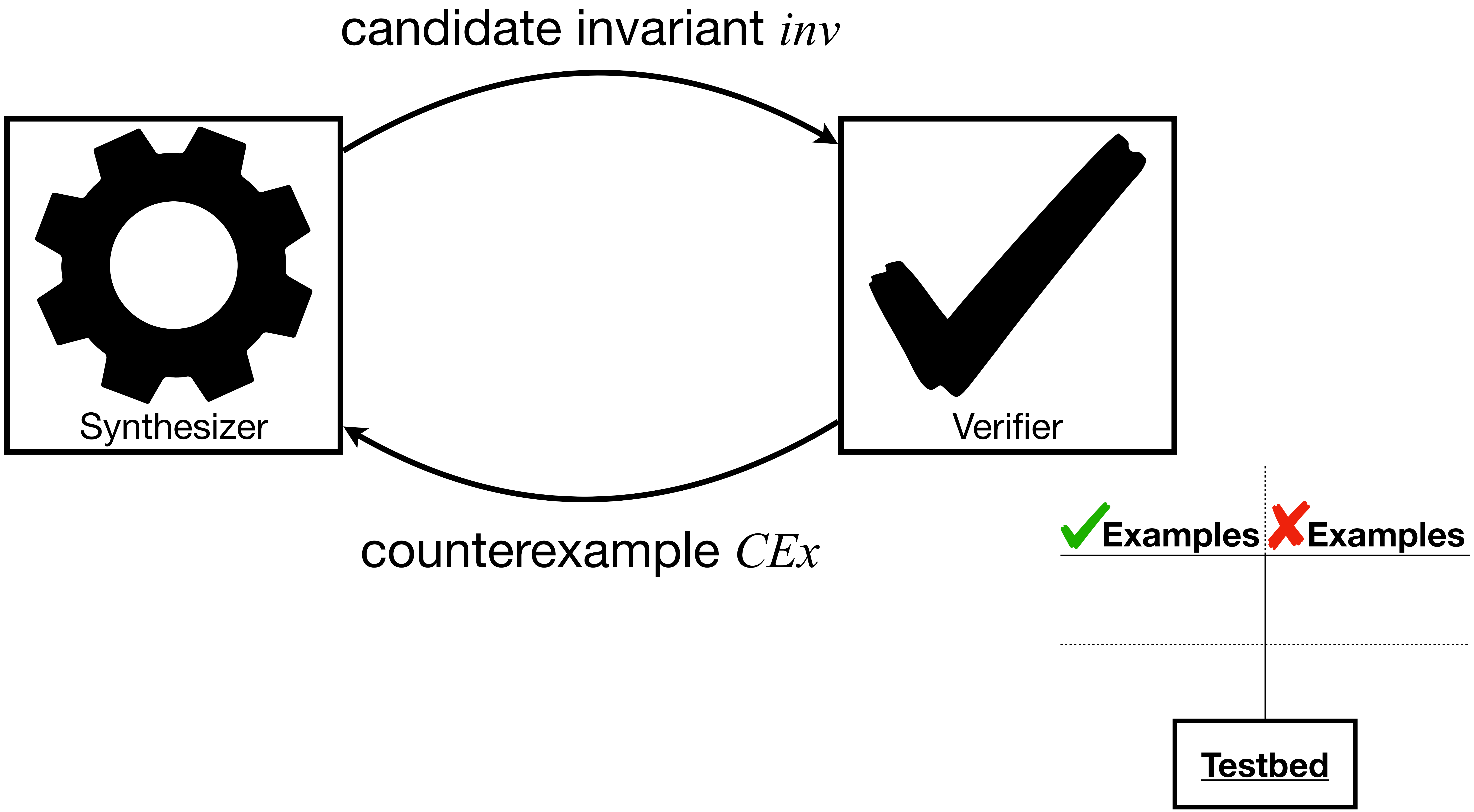
✔ inv `[] = true`

✘ $\forall s.\forall i.inv$ `s = true` $\Rightarrow$
inv `(insert s i) = true`

`insert [1] 0 = [0;1]`

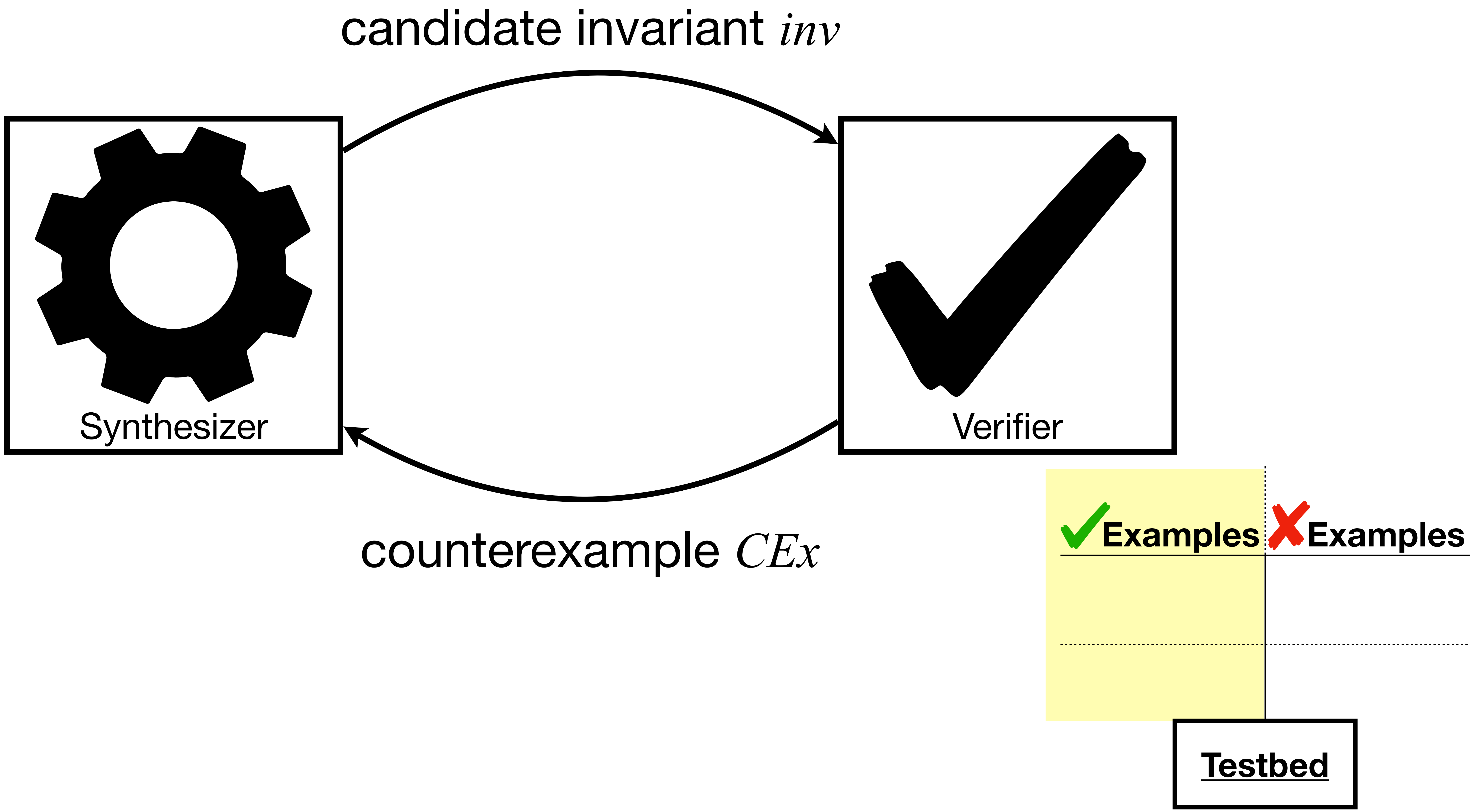
```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Algorithm

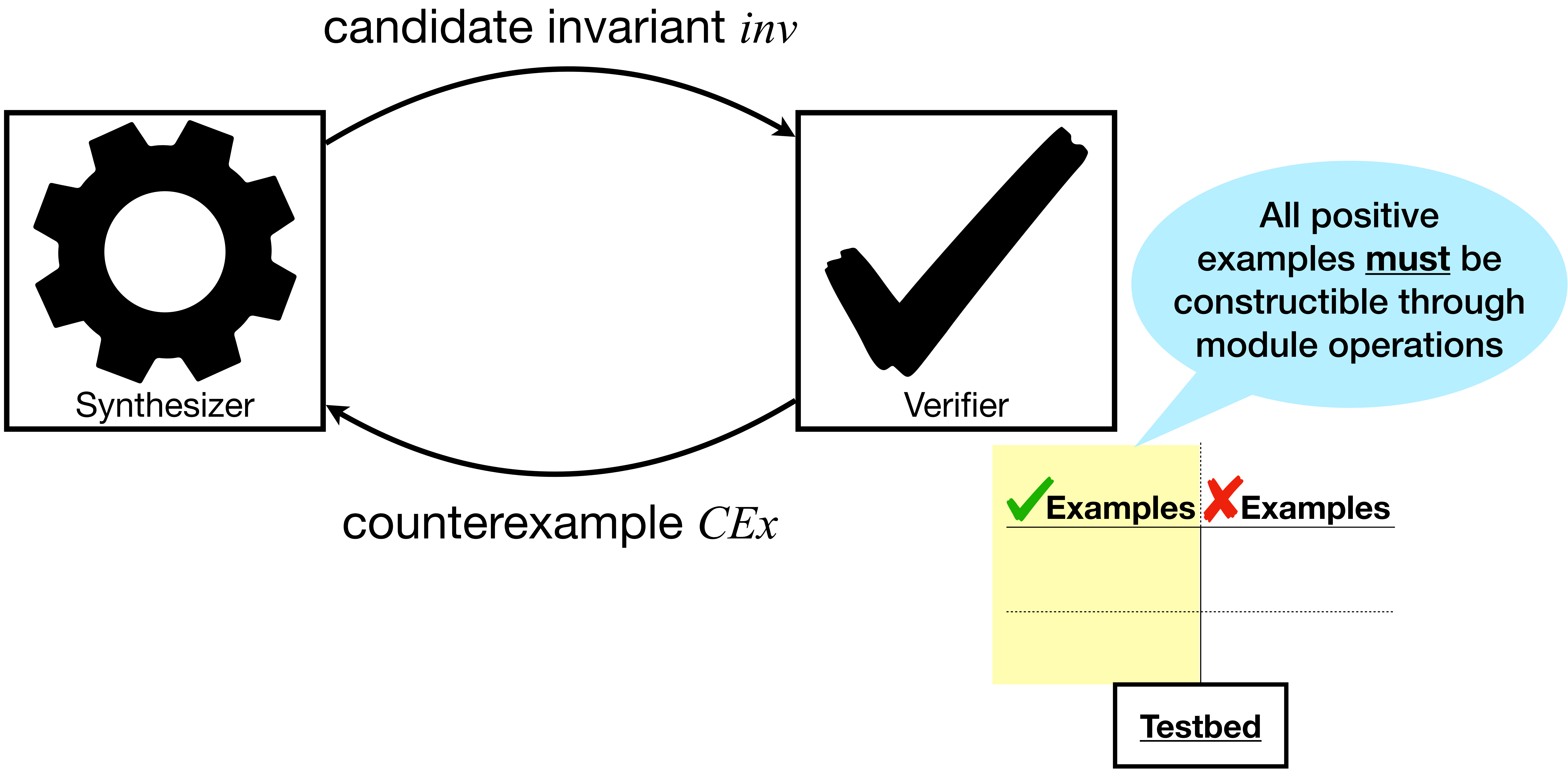


candidate invariant *inv*
Synthesizer
Verifier
counterexample *CEx*
Examples
Examples
Testbed

Algorithm


candidate invariant *inv*
Synthesizer
Verifier
counterexample *CEx*
✓Examples
✗Examples
Testbed

Algorithm


candidate invariant *inv*
Synthesizer
Verifier
counterexample *CEx*
All positive examples **must** be constructible through module operations
Examples
Examples
Testbed

Visible Inductiveness

✓ Examples	✗ Examples
[]	
[1]	

Can we find an inductiveness counterexample where the inputs must be ✓ examples

$$inv\ \mathtt{l} = \mathtt{length\ l} < 2$$

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Visible Inductiveness

✓ Examples	✗ Examples
[]	
[1]	

Can we find an inductiveness counterexample where the inputs must be ✓ examples

inv `l = length l < 2`

? *inv* `[] = true`

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Visible Inductiveness

✔ Examples	✘ Examples
[]	
[1]	

Can we find an inductiveness counterexample where the inputs must be ✔ examples

inv l = length l < 2

✔ inv [] = true

```
module ListSet : SET = struct
  type t = int list
  let empty = []
  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)
  let insert l x =
    if (lookup l x) then l else (x :: l)
  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Visible Inductiveness

✓ Examples	✗ Examples
[]	
[1]	

Can we find an inductiveness counterexample where the inputs must be ✓ examples

inv `l = length l < 2`

✓ inv `[] = true`

? $\forall s. \forall i.\, inv$ `s = true` $\Rightarrow$ inv `(insert s i) = true`

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Visible Inductiveness

✓ Examples	✗ Examples
[]	
[1]	

Can we find an inductiveness counterexample where the inputs must be ✓ examples

$$\mathit{inv}\ \texttt{l} = \texttt{length l < 2}$$

✓ $\mathit{inv}\ \texttt{[]} = \texttt{true}$

✗ $\forall s.\forall i.\mathit{inv}\ \texttt{s} = \texttt{true} \Rightarrow \mathit{inv}\ \texttt{(insert s i)} = \texttt{true}$

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Visible Inductiveness

✓ Examples	✗ Examples
[]	
[1]	

Can we find an inductiveness counterexample where the inputs must be ✓ examples

inv `l = length l < 2`

✓ inv `[] = true`

✗ $\forall s. \forall i. inv$ `s = true` $\Rightarrow$
inv `(insert s i) = true`

`insert [1] 0 = [0;1]`

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Visible Inductiveness

✓ Examples	✗ Examples
[]	
[1]	

Can we find an inductiveness counterexample where the inputs must be ✓ examples

inv l = length l < 2

✓ *inv* [] = true

✗ ∀s.∀i.*inv* s = true ⇒ *inv* (insert s i) = true

insert [1] 0 = [0;1]

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Visible Inductiveness

✓ Examples	✗ Examples
[]	
[1]	

Can we find an inductiveness counterexample where the inputs must be ✓ examples

$$\mathit{inv}\ \mathtt{l} = \mathtt{length\ l} < 3$$

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Visible Inductiveness

✓ Examples	✗ Examples
[]	
[1]	

Can we find an inductiveness counterexample where the inputs must be ✓ examples

inv `l = length l < 3`

? inv `[] = true`

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Visible Inductiveness

✓ Examples	✗ Examples
[]	
[1]	

Can we find an inductiveness counterexample where the inputs must be ✓ examples

inv `l = length l < 3`

✓ inv `[] = true`

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Visible Inductiveness

✓ Examples	✗ Examples
[]	
[1]	

Can we find an inductiveness counterexample where the inputs must be ✓ examples

$$\mathit{inv}\ \mathtt{l} = \mathtt{length\ l} < 3$$

✓ $\mathit{inv}\ \mathtt{[]} = \mathtt{true}$

? $\forall s. \forall i. \mathit{inv}\ \mathtt{s} = \mathtt{true} \Rightarrow \mathit{inv}\ \mathtt{(insert\ s\ i)} = \mathtt{true}$

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Visible Inductiveness

✓ Examples	✗ Examples
`[]`	
`[1]`	

Can we find an inductiveness counterexample where the inputs must be ✓ examples

inv `l = length l < 3`

✓ inv `[] = true`

✓ $\forall s. \forall i. inv$ `s = true` $\Rightarrow$ inv `(insert s i) = true`

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Visible Inductiveness

✓ Examples	✗ Examples
[]	
[1]	

Can we find an inductiveness counterexample where the inputs must be ✓ examples

inv `l = length l < 3`

✓ inv `[] = true`

✓ $\forall s.\forall i.inv$ `s = true` $\Rightarrow$ inv `(insert s i) = true`

⋮

Visibly Inductive

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Say we are visibly inductive...

✓ Examples	Examples
[]	
[1]	

$$inv\ \texttt{l} = \texttt{length l} < 3$$

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Then do a full inductiveness check!

Examples	Examples
[]	
[1]	

inv `l = length l < 3`

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Verifying Full Inductiveness

inv `l = length l < 3`

? *inv* `[] = true`

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Verifying Full Inductiveness

inv l = length l < 3

✓ *inv* [] = true

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Verifying Full Inductiveness

inv `l = length l < 3`

✓ inv `[] = true`

? $\forall s. \forall i. inv$ `s = true` $\Rightarrow$ inv `(insert s i) = true`

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Verifying Full Inductiveness

$$inv\ \texttt{l} = \texttt{length l} < 3$$

✓ $inv\ \texttt{[]} = \texttt{true}$

✗ $\forall s. \forall i. inv\ \texttt{s} = \texttt{true} \Rightarrow inv\ \texttt{(insert s i)} = \texttt{true}$

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Verifying Full Inductiveness

inv `l = length l < 3`

✓ inv `[] = true`

✗ $\forall s. \forall i. inv$ `s = true` $\Rightarrow$ inv `(insert s i) = true`

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

`insert [0;1] 2 = [2;0;1]`

Verifying Full Inductiveness

inv `l = length l < 3`

✓ inv `[] = true`

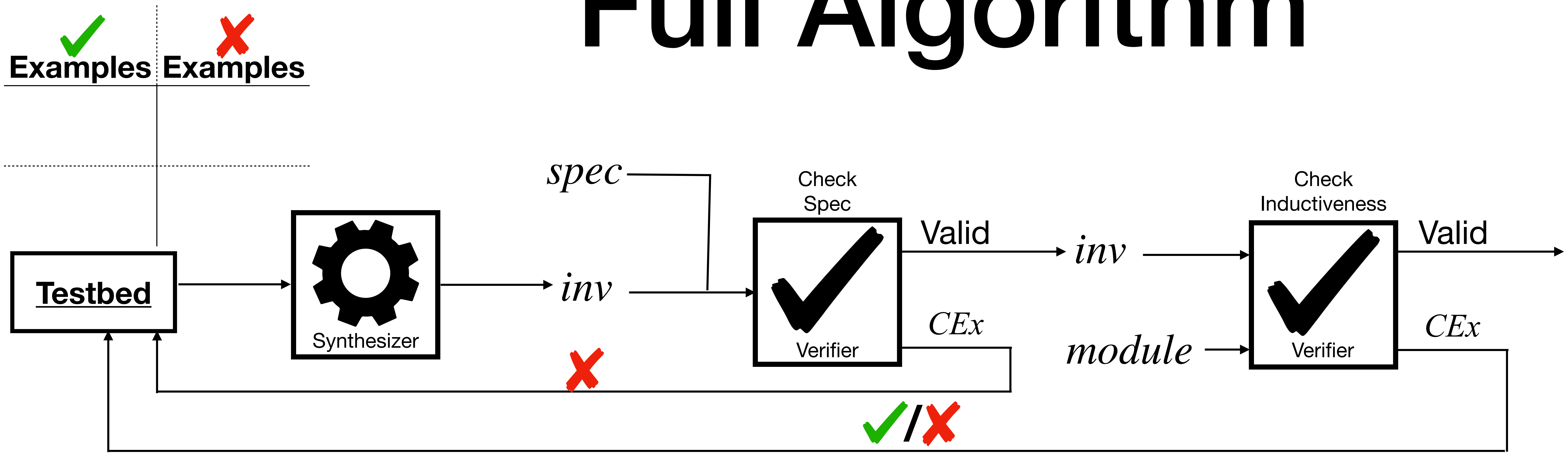
✗ $\forall s. \forall i. inv$ `s = true` $\Rightarrow$
inv `(insert s i) = true`

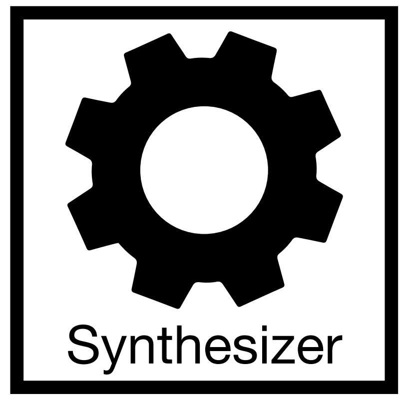
Add it to negative set

`insert [0;1] 2 = [2;0;1]`

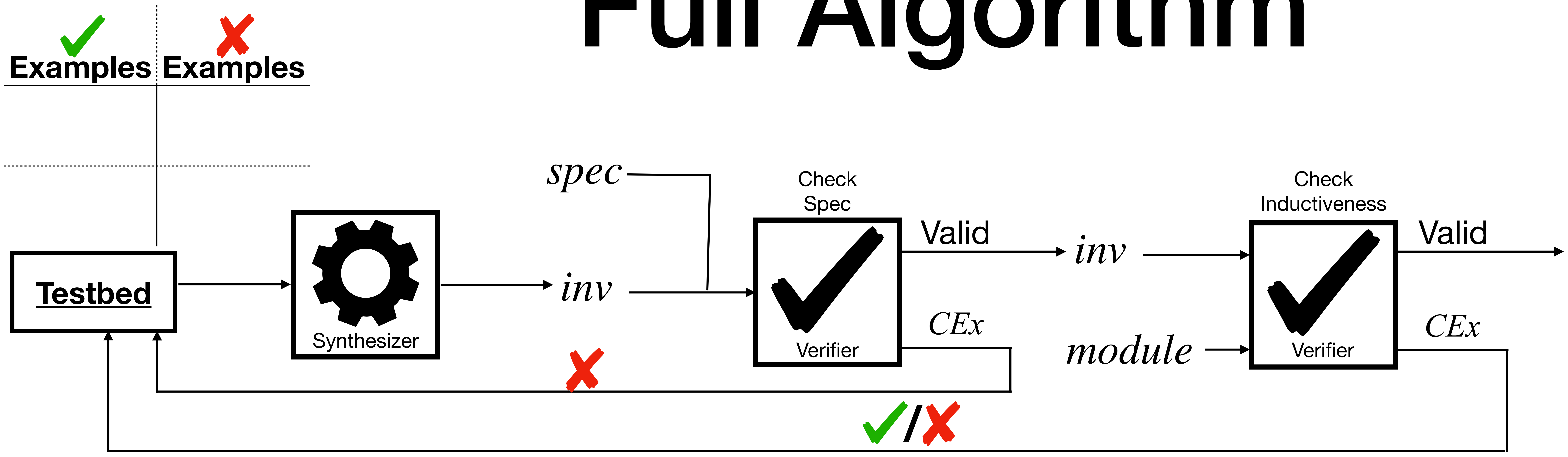
```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Full Algorithm

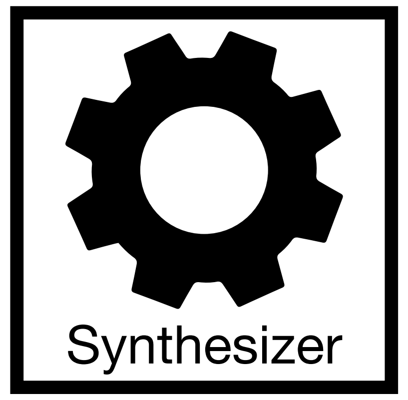




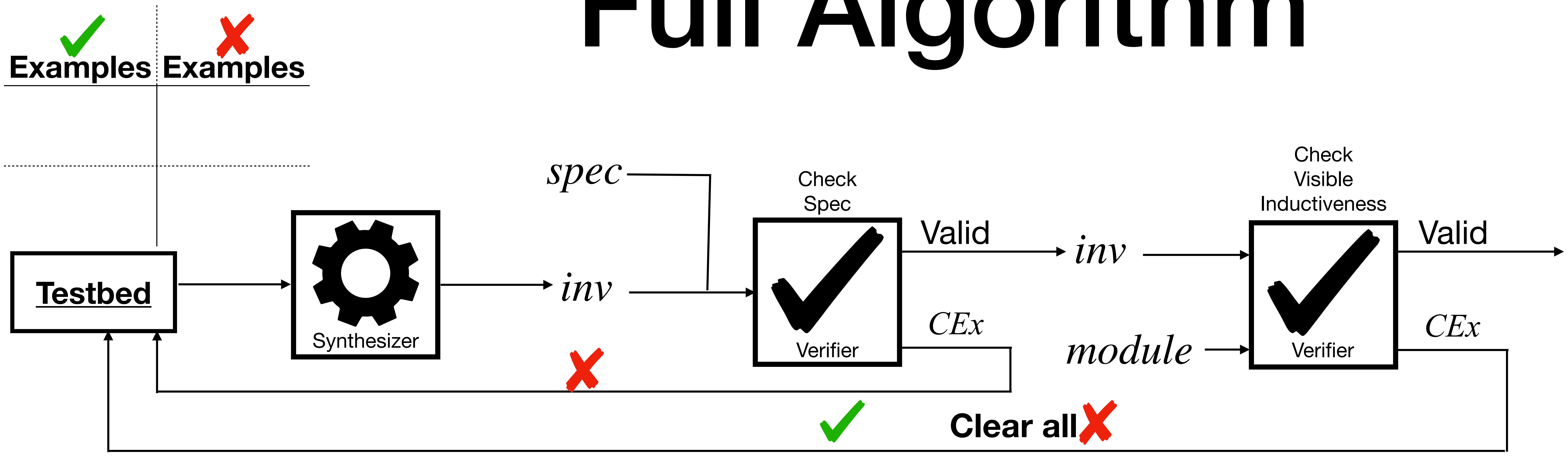
Full Algorithm



Constraint:
✓ are always reachable

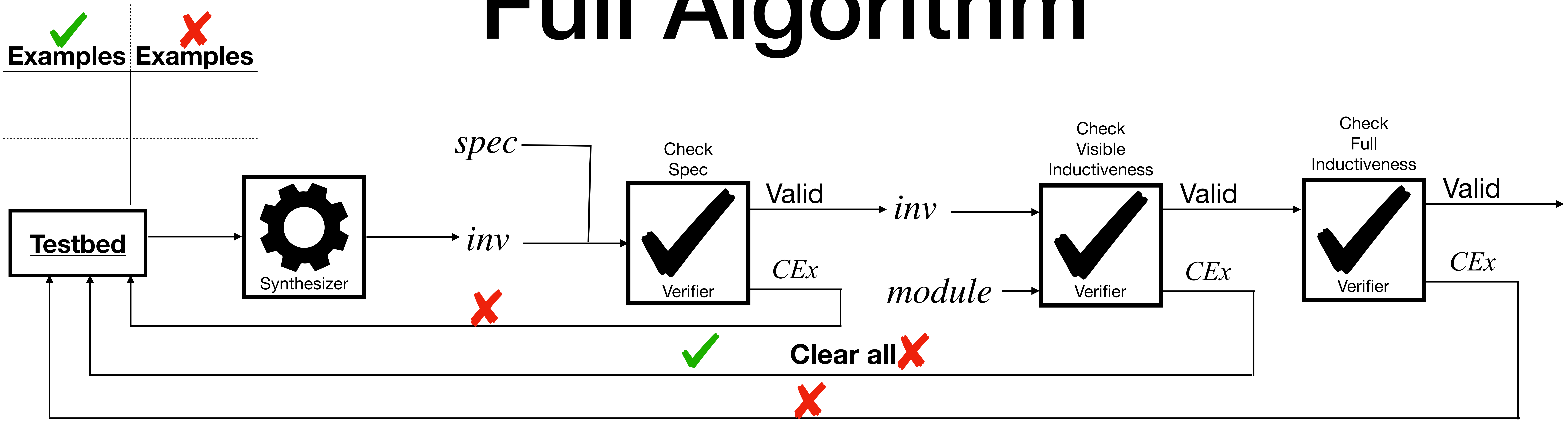


Full Algorithm

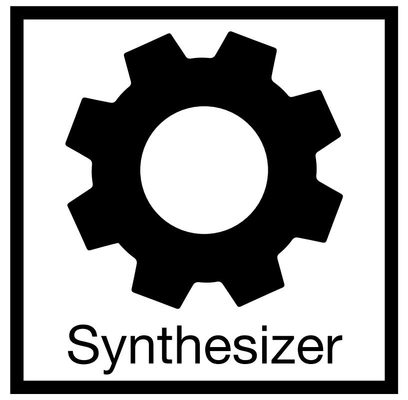


Constraint:
✓ are always reachable

Full Algorithm



Constraint:
✓ are always reachable



Correctness Theorem

If:

1. Our verifier is sound and complete

2. Our synthesizer is sound and complete

3. Our concrete data type only has a finite number of elements

Then our algorithm will find a sufficient representation invariant, if one exists

Hanoi

Hanoi

Synthesizer

MYTH

[Osera and Zdancewic 2015]

Hanoi

Synthesizer

Myth

[Osera and Zdancewic 2015]

```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

Hanoi

Synthesizer

MYTH

[Osera and Zdancewic 2015]

Verifier

Enumerative Tester

Hanoi

Synthesizer

MYTH

[Osera and Zdancewic 2015]

Verifier

Enumerative Tester

👎 Unsound

Hanoi

Synthesizer

MYTH

[Osera and Zdancewic 2015]

Verifier

Enumerative Tester

- 👎 Unsound
- 👍 Fast
- 👍 Guaranteed to terminate

Hanoi

Synthesizer

MYTH

[Osera and Zdancewic 2015]

Verifier

Enumerative Tester

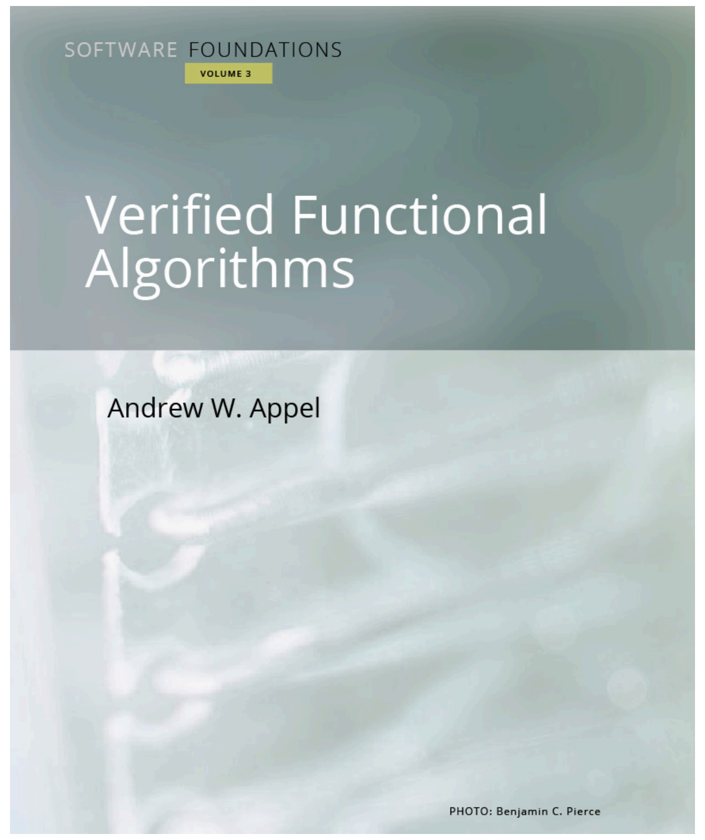
- 👎 Unsound
- 👍 Fast
- 👍 Guaranteed to terminate

Theory doesn't address higher-order functions.
Hanoi does (using higher-order contracts).

Evaluation

Benchmark Suite Construction

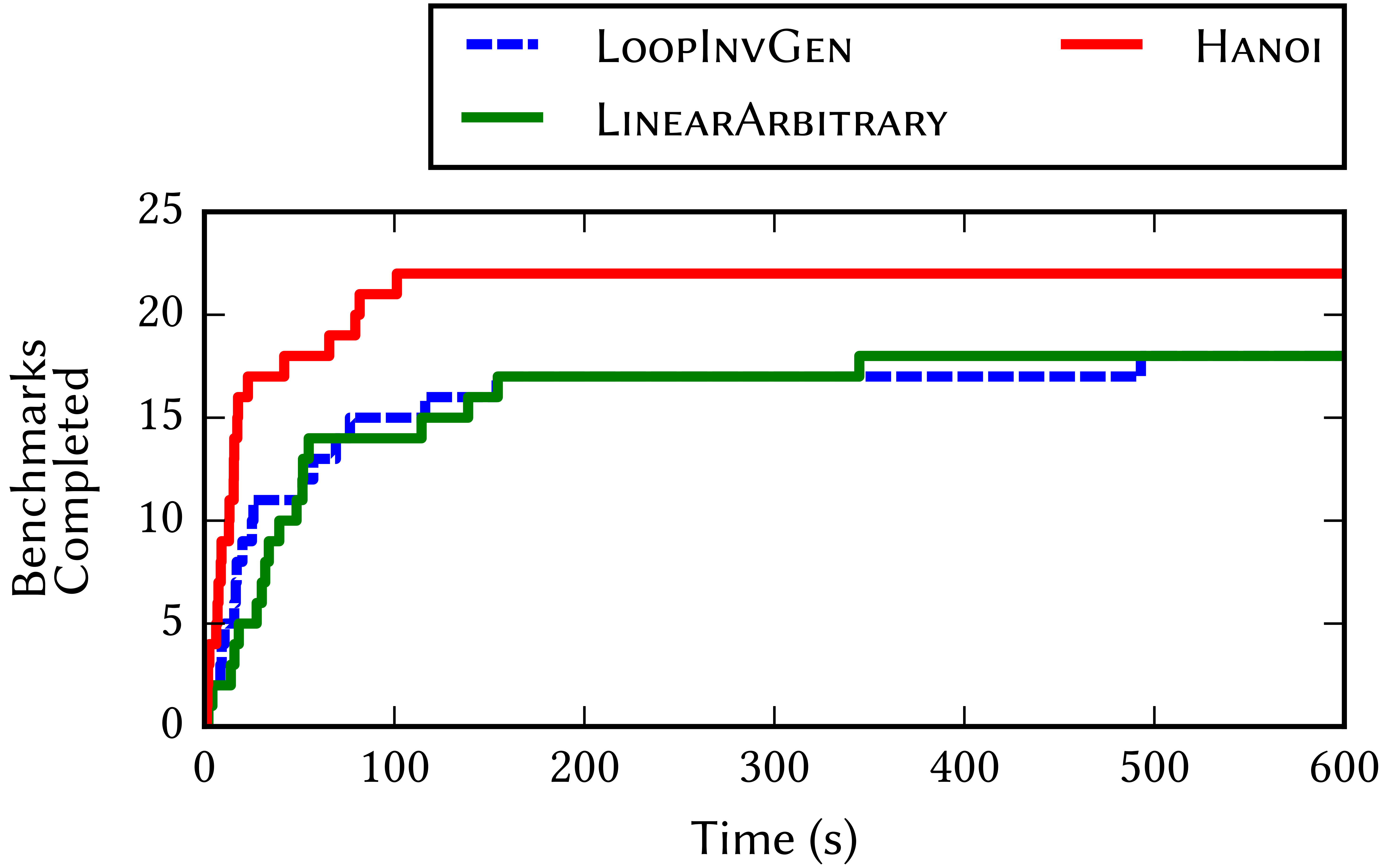
- Verified Function Algorithms (5)
- VFAExt (3)
- Coq (14)
- Other (6)



Numbers

- Timeout of 30 minutes
- Inferred 22/28 Invariants
- All of our inferred invariants were correct, despite using a tester for verification

Comparisons



LoopInvGen

[Padhi et al. 2016]

LinearArbitrary

[Zhu et al. 2018]

Burst
for Synthesis

ListSet

```ocaml
module type SET = sig
    type t
    val empty  : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
end
```

$$
\begin{aligned}
\forall i : \mathtt{int}. \forall s : \mathtt{t}. \forall s' : \mathtt{t}. \quad & \neg(\mathtt{lookup\ empty}\ i) \\
\wedge\ & (\mathtt{lookup}\ (\mathtt{insert}\ s\ i)\ i) \\
\wedge\ & \neg(\mathtt{lookup}\ (\mathtt{delete}\ s\ i)\ i)
\end{aligned}
$$

```ocaml
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

ListSet

NOW WITH INVARIANTS!

```
module type SET = sig
    type t
    val empty  : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
end
```

$$
\begin{aligned}
\forall i : \mathtt{int}.\forall s : \mathtt{t}.\forall s' : \mathtt{t}.\quad & \neg(\mathtt{lookup\ empty}\ i) \\
\wedge\ & (\mathtt{lookup}\ (\mathtt{insert}\ s\ i)\ i) \\
\wedge\ & \neg(\mathtt{lookup}\ (\mathtt{delete}\ s\ i)\ i)
\end{aligned}
$$

```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

ListSet++

```
module type SET = sig
    type t
    val empty  : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
    val insert_all : t -> int list -> t
end
```

$$\forall i:\mathtt{int}.\forall s:\mathtt{t}.\forall s':\mathtt{t}.\quad \neg(\mathtt{lookup\ empty}\ i) \wedge (\mathtt{lookup}\ (\mathtt{insert}\ s\ i)\ i) \wedge \neg(\mathtt{lookup}\ (\mathtt{delete}\ s\ i)\ i)$$

```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

ListSet++

```
module type SET = sig
    type t
    val empty  : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
    val insert_all : t -> int list -> t
end
```

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))

end
```

$$\forall i : \mathtt{int}. \forall s : \mathtt{t}. \forall s' : \mathtt{t}. \quad \neg(\mathtt{lookup\ empty}\ i) \wedge (\mathtt{lookup}\ (\mathtt{insert}\ s\ i)\ i) \wedge \neg(\mathtt{lookup}\ (\mathtt{delete}\ s\ i)\ i)$$

inv l = match l with
| [] -> true
| h::t -> ¬(lookup h t) ∧ *inv* t

ListSet++

```
module type SET = sig
    type t
    val empty  : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
    val insert_all : t -> int list -> t
end
```

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

$$\forall i:\texttt{int}.\forall s:\texttt{t}.\forall s':\texttt{t}.\quad \neg(\texttt{lookup empty } i) \wedge (\texttt{lookup }(\texttt{insert } s\ i)\ i) \wedge \neg(\texttt{lookup }(\texttt{delete } s\ i)\ i)$$

$$\forall is:\texttt{int list}.\forall s:\texttt{t}.\forall out:\texttt{t}.\quad out = (\texttt{insert\_all } s\ is) \Rightarrow (\texttt{size } out) = (\texttt{size } s) + (\texttt{size }(\texttt{dedup } is))$$

```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

ListSet++

```
inv l = match l with
        | [] -> true
        | h::t -> ¬(lookup h t) ∧ inv t
```

$$\forall is : \mathsf{int\ list}.\, \forall s : \mathsf{t}.\, \forall out : \mathsf{t}.\quad out = (\mathsf{insert\_all}\ s\ is) \Rightarrow (\mathsf{size}\ out) = (\mathsf{size}\ s) + (\mathsf{size}\ is)$$

$$\forall is : \mathsf{int\ list}.\, \forall s : \mathsf{t}.\, \forall out : \mathsf{t}.\quad out = (\mathsf{insert\_all}\ s\ is) \Rightarrow (inv\ s) \Rightarrow ((\mathsf{size}\ out) = (\mathsf{size}\ s) + (\mathsf{size}\ (\mathsf{dedup}\ is)) \wedge (inv\ out))$$

$$
\begin{aligned}
\forall is : \mathtt{int\ list}.\ \forall s : \mathtt{t}.\ \forall out : \mathtt{t}.\quad & out = (\mathtt{insert\_all}\ s\ is) \\
\Rightarrow\ & (inv\ s) \\
\Rightarrow\ & ((\mathtt{size}\ out) = (\mathtt{size}\ s) + (\mathtt{size}\ \ (\mathtt{dedup}\ is)) \\
\wedge\ & (inv\ out))
\end{aligned}
$$

Burst

```
let rec insert_all (s:t) (is:int list) =
  match is with
  | [] -> s
  | h::t -> insert h (insert_all s t)
```

What can do this?

Leon

SyGuS
Syntax-Guided Synthesis

What else do we want?

1. Recursion

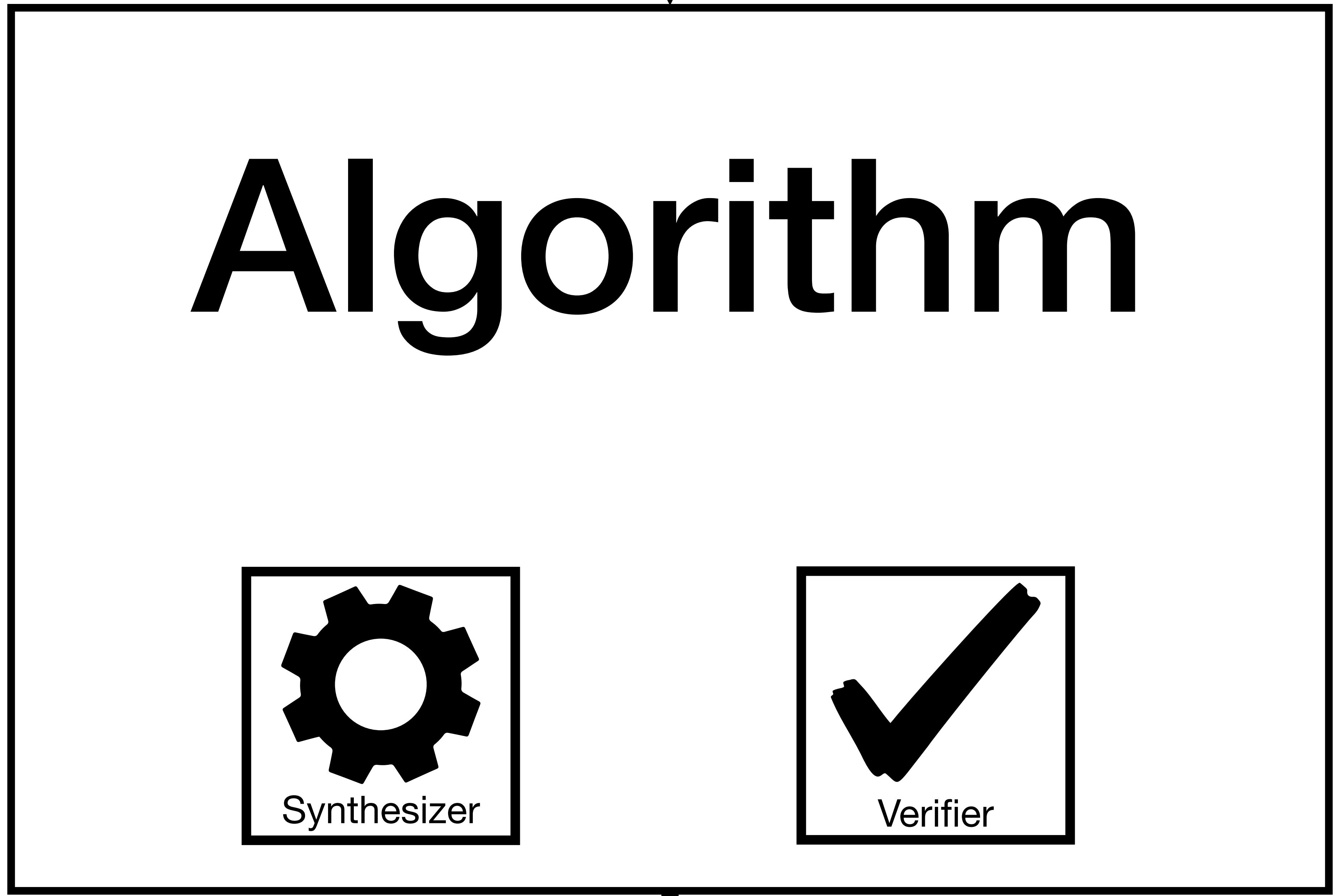
What else do we want?

1. Recursion
    a. Non-inductive Specifications

What else do we want?

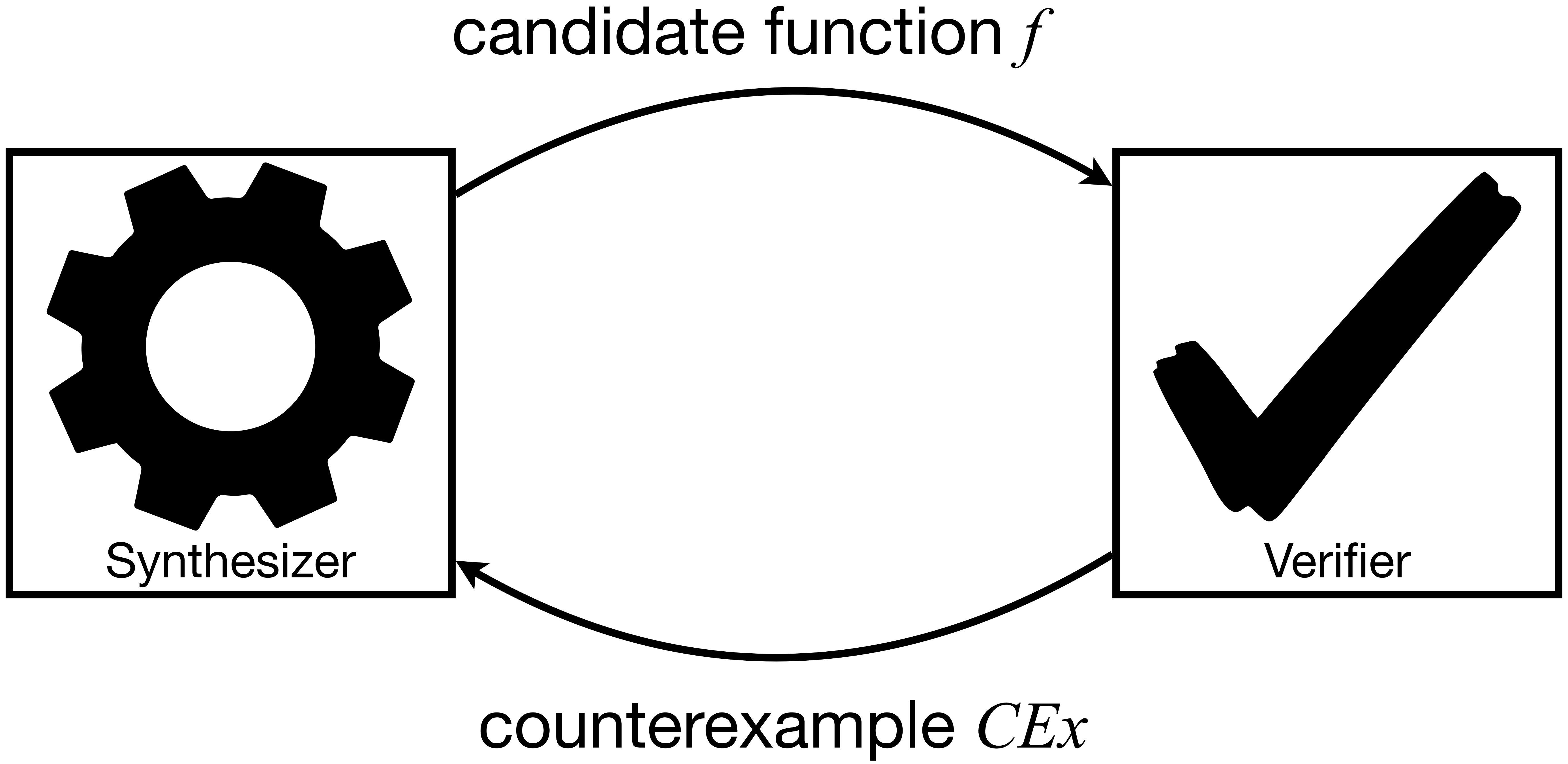
1. Recursion
    a. Non-inductive Specifications
2. No Top-Down Reasoning

$$
\begin{aligned}
\forall is : \texttt{int list}.\ \forall s : \texttt{t}.\ \forall out : \texttt{t}.\quad & out = (\texttt{insert\_all}\ s\ is) \\
\Rightarrow\ & (inv\ s) \\
\Rightarrow\ & ((\texttt{size}\ out) = (\texttt{size}\ s) + (\texttt{size}\ \ (\texttt{dedup}\ is)) \\
\land\ & (inv\ out))
\end{aligned}
$$

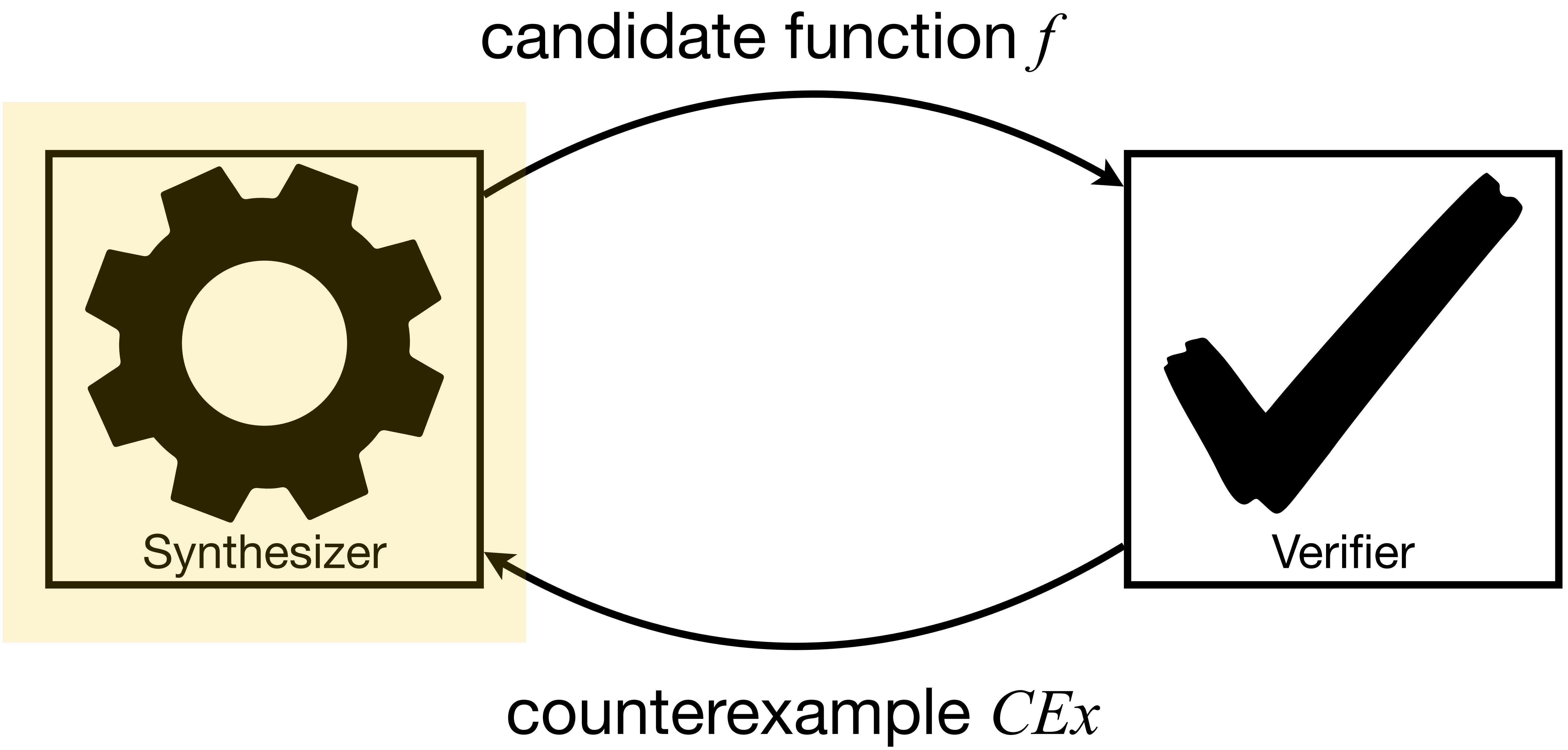



```
let rec insert_all (s:t) (is:int list) =
  match is with
  | [] -> s
  | h::t -> insert h (insert_all s t)
```

Algorithm


candidate function f
Synthesizer
Verifier
counterexample CEx

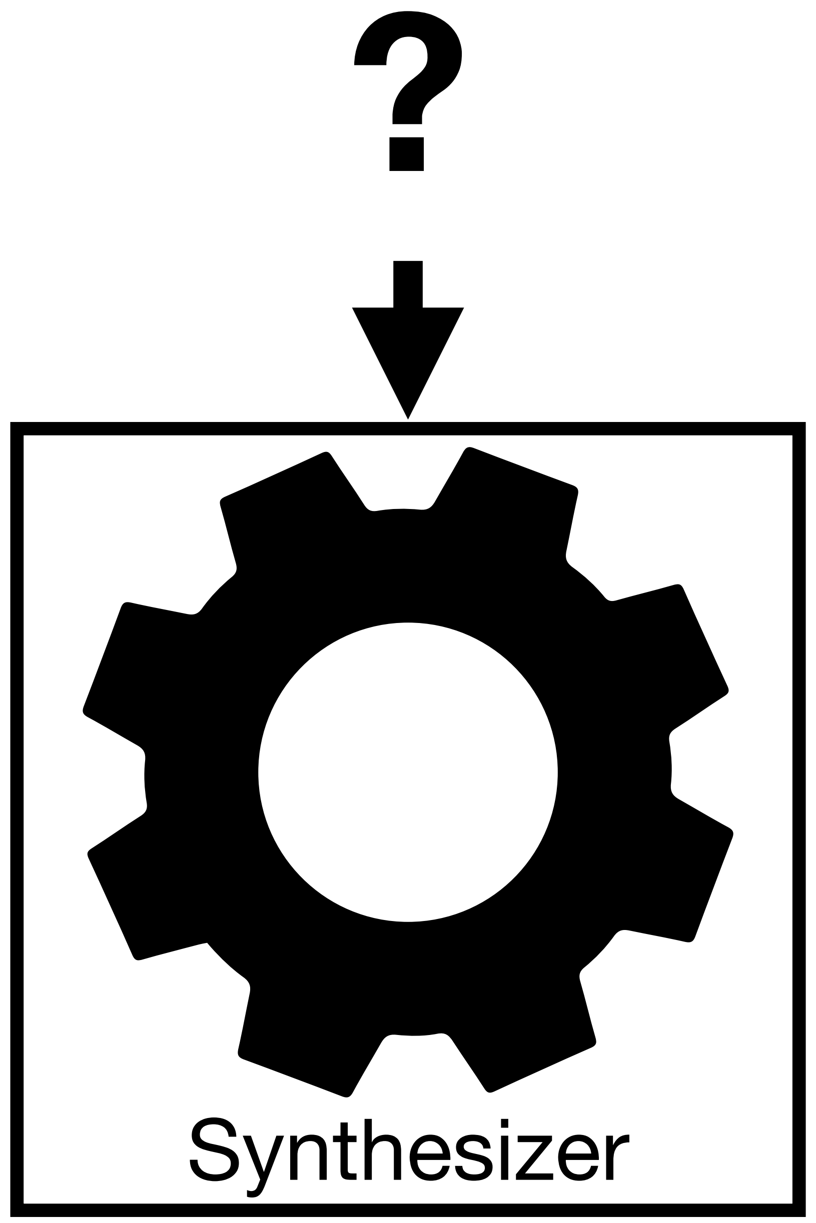
Algorithm


candidate function f
Synthesizer
Verifier
counterexample CEx

$$s = [],\ is = [0]$$

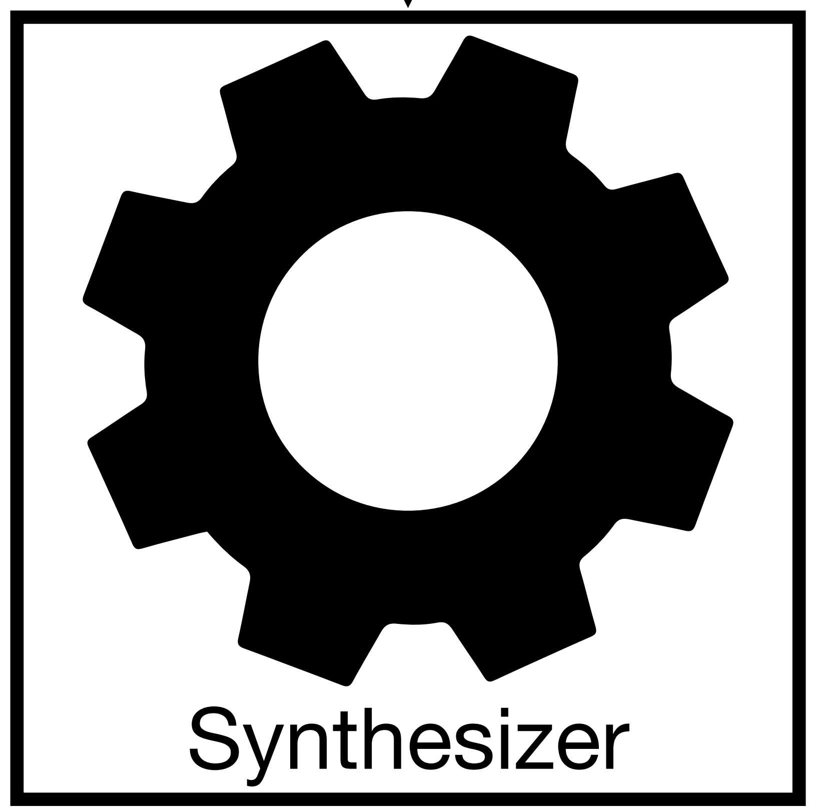
$$\begin{aligned}\forall is : \texttt{int list}.\ \forall s : \texttt{t}.\ \forall out : \texttt{t}.\quad & out = (\texttt{insert\_all}\ s\ is)\\ \Rightarrow\ & (inv\ s)\\ \Rightarrow\ & ((\texttt{size}\ out) = (\texttt{size}\ s) + (\texttt{size}\ is)\\ \wedge\ & (inv\ out))\end{aligned}$$

$$\varphi_{[],[0]}(out) = (\texttt{size}\ out) = (\texttt{size}\ []) + (\texttt{size}\ [0]) \wedge inv\ \texttt{out}$$



Ground Formulas

$\varphi_{[],[0]} \wedge (\varphi_{[0],[]} \vee \psi_{[0],[1]})$



`s = [], is = [0]`

$$\forall is : \mathtt{int\ list}. \forall s : \mathtt{t}. \forall out : \mathtt{t}. \quad out = (\mathtt{insert\_all}\ s\ is) \Rightarrow (inv\ s) \Rightarrow ((\mathtt{size}\ out) = (\mathtt{size}\ s) + (\mathtt{size}\ is) \wedge (inv\ out))$$

$$\varphi_{[],[0]}(out) = (\mathtt{size}\ out) = (\mathtt{size}\ [\,]) + (\mathtt{size}\ [0]) \wedge inv\ \mathtt{out}$$

Let’s Simplify First

1. Recursion
   a. Non-inductive Specifications
2. No Top-Down Reasoning

Let's Simplify First

1. Recursion

    ~~a. Non-inductive Specifications~~ Inductive IO Specs

2. No Top-Down Reasoning

FTA Synthesis [Wang 2017]

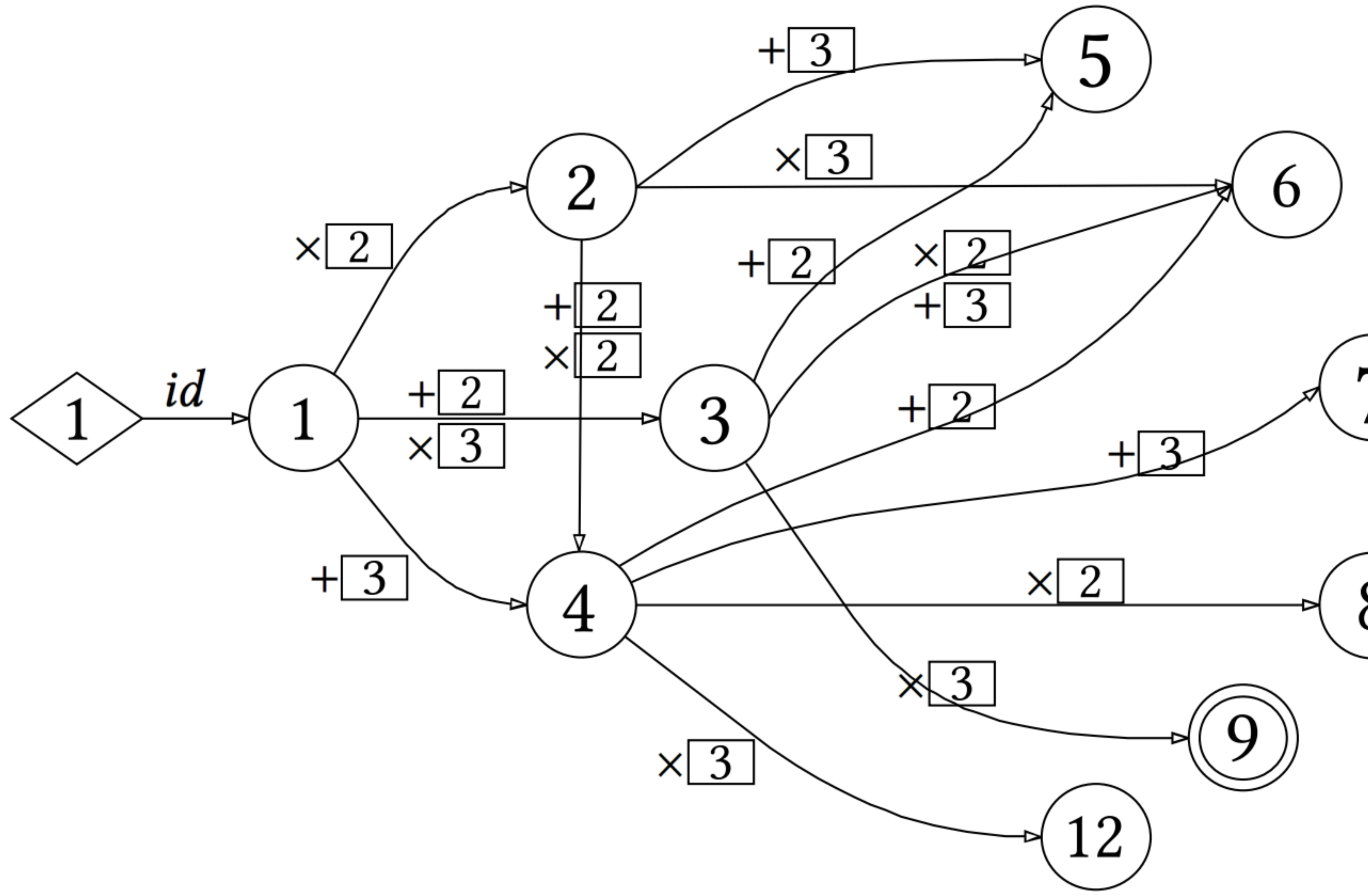
FTAs or *Finite Tree Automata* are automata

Where DFAs represent sets of strings, FTAs represent sets of *trees*

Programs are trees

FTAs can describe the set of all valid programs

1 ↦ 9

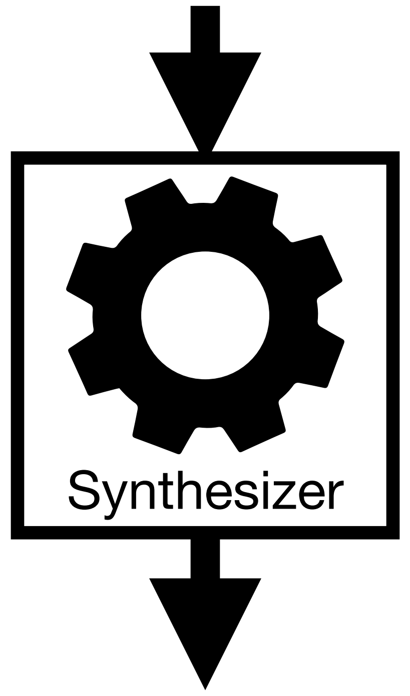


Guarantee

$e \mapsto e'$ and $f \in A$ if, and only if $f\ e \to^* e'$

Example Task

```
insert_all [0] [2,1,2] = [0,1,2]
insert_all [0] [1,2] = [0,1,2]
insert_all [0] [2] = [0,2]
insert_all [0] [] = [0]
```




```
let rec insert_all (s:t) (is:int list) =
  match is with
  | [] -> s
  | h::t -> insert h (insert_all s t)
```

FTA Synthesis [Wang 2017]

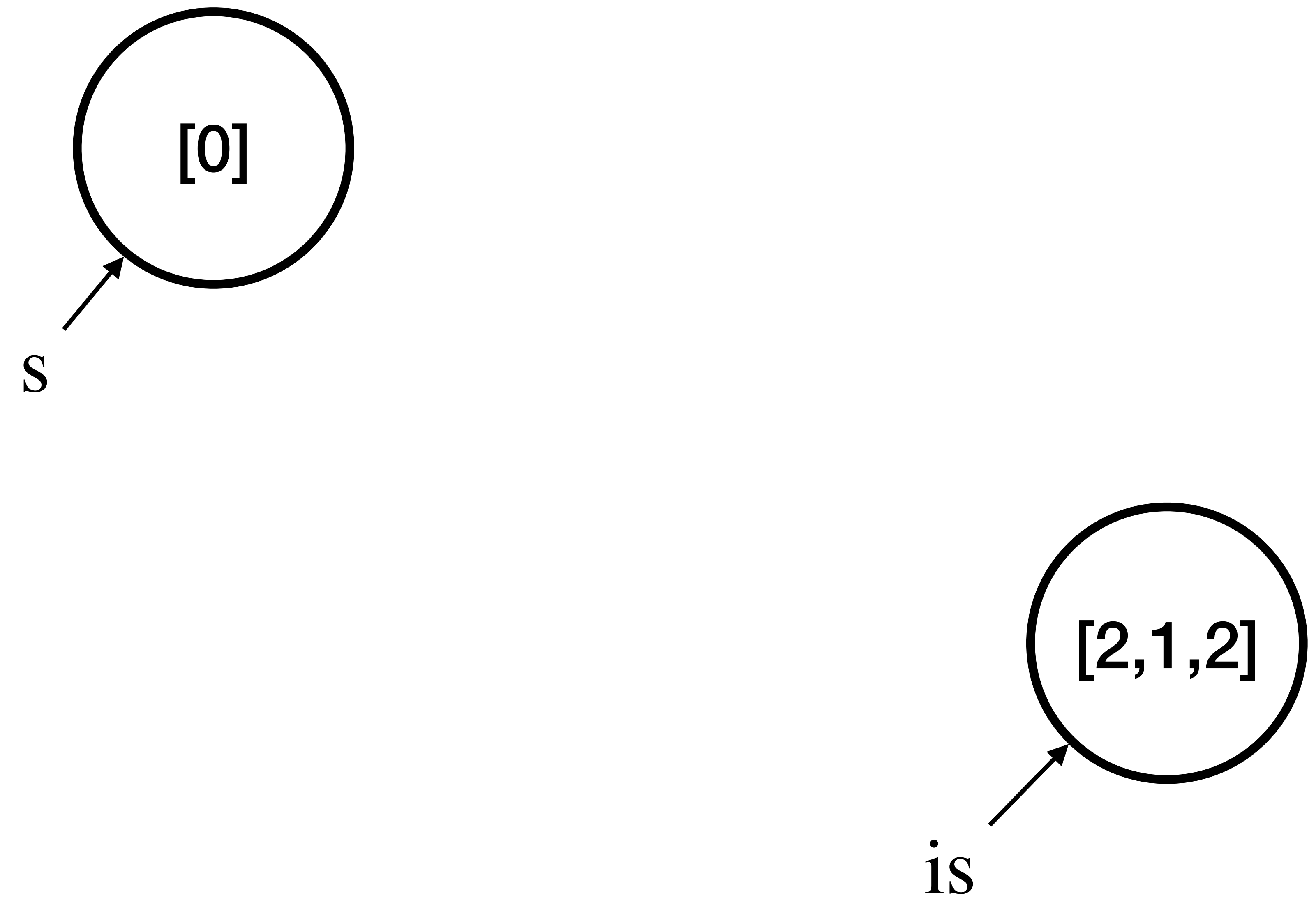
```
insert_all [0] [2,1,2] = [0,1,2]
insert_all [0] [1,2] = [0,1,2]
insert_all [0] [2] = [0,2]
insert_all [0] [] = [0]
```

FTA Synthesis [Wang 2017]

```
→insert_all [0] [2,1,2] = [0,1,2]
 insert_all [0] [1,2] = [0,1,2]
 insert_all [0] [2] = [0,2]
 insert_all [0] [] = [0]
```

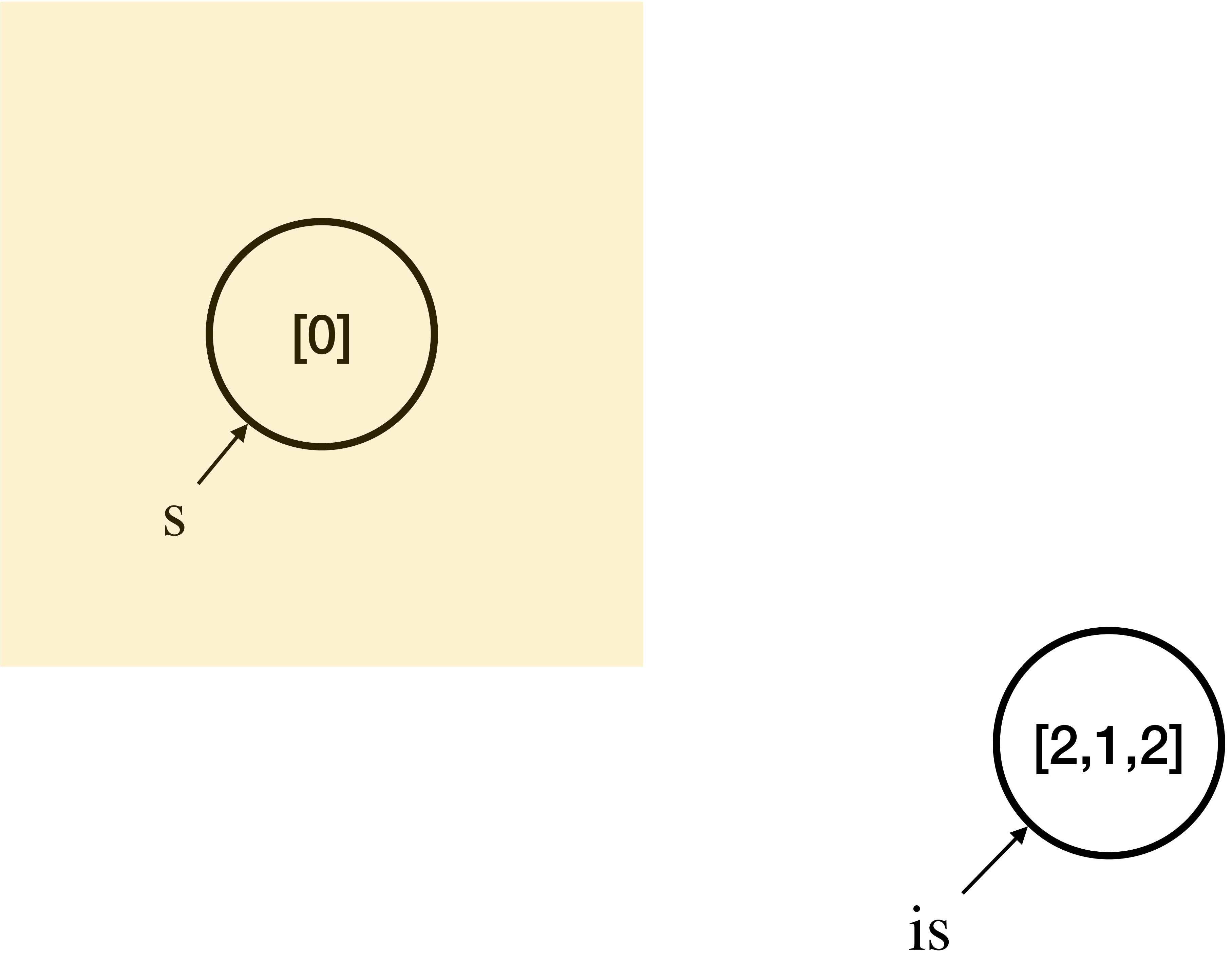
FTA Synthesis [Wang 2017]

```
→insert_all [0] [2,1,2] = [0,1,2]
 insert_all [0] [1,2] = [0,1,2]
 insert_all [0] [2] = [0,2]
 insert_all [0] [] = [0]
```



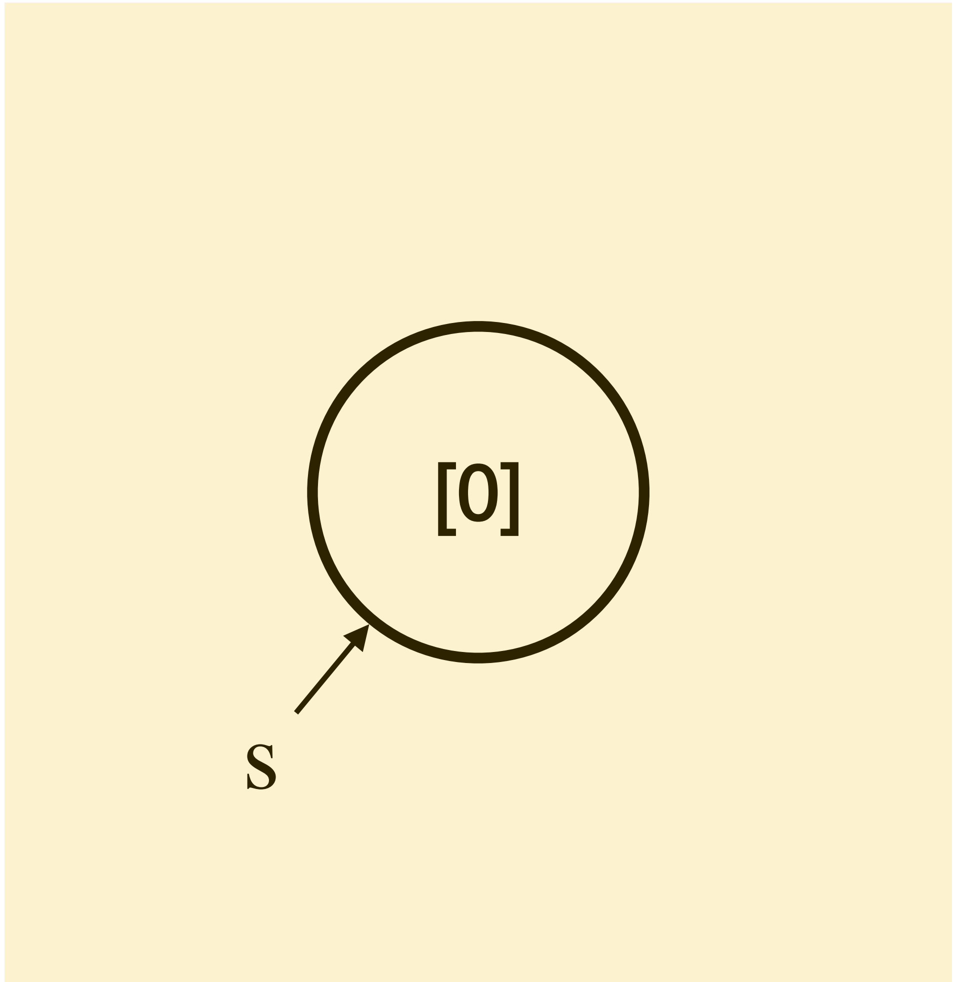
FTA Synthesis [Wang 2017]

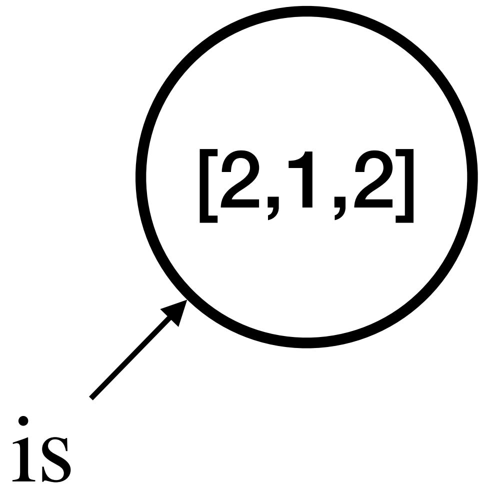
```
→insert_all [0] [2,1,2] = [0,1,2]
 insert_all [0] [1,2] = [0,1,2]
 insert_all [0] [2] = [0,2]
 insert_all [0] [] = [0]
```



FTA Synthesis [Wang 2017]

```
insert_all [0] [2,1,2] = [0,1,2]
insert_all [0] [1,2] = [0,1,2]
insert_all [0] [2] = [0,2]
insert_all [0] [] = [0]
```

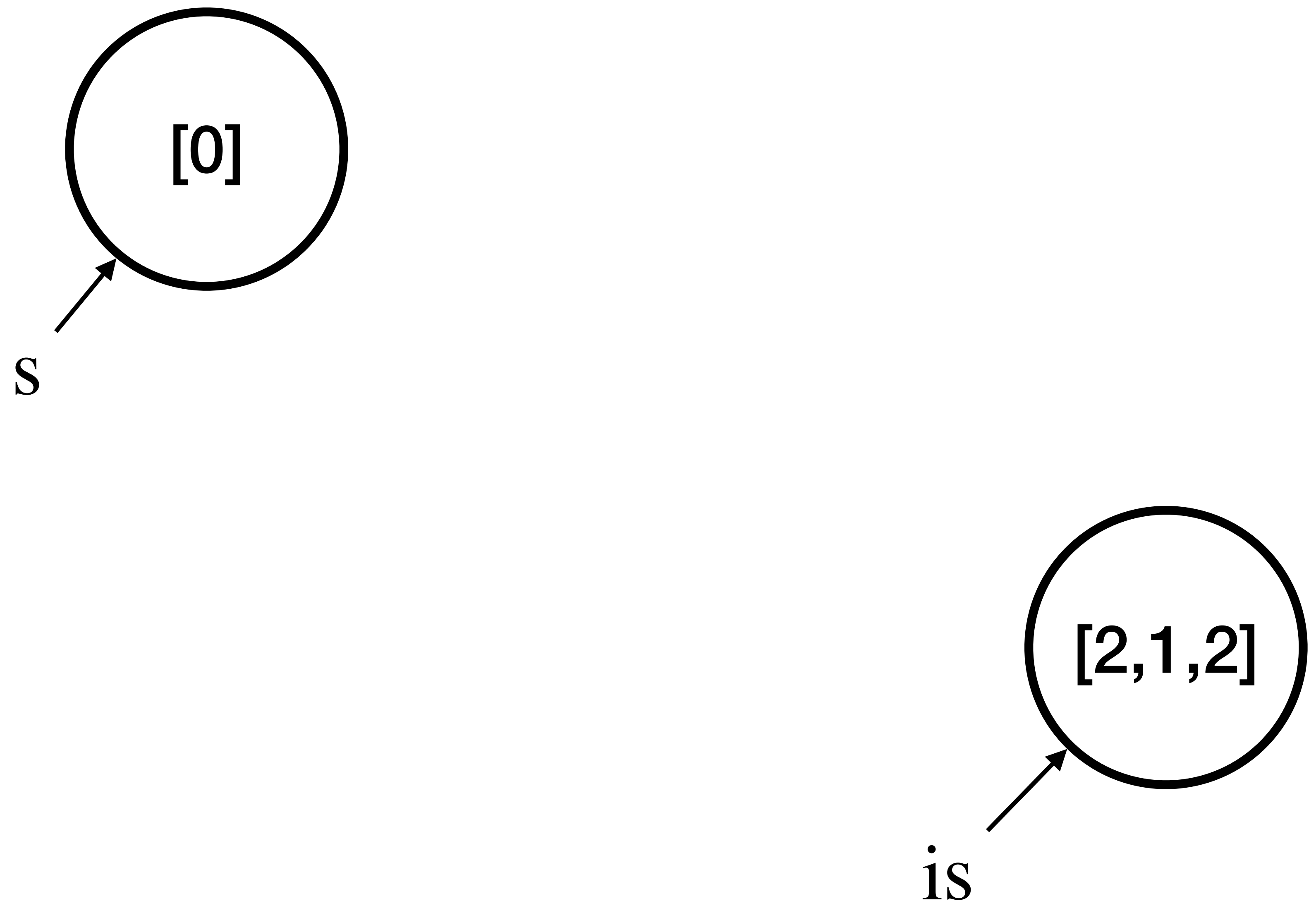





```
let rec insert_all (s:t) (is:int list) =
  s
```

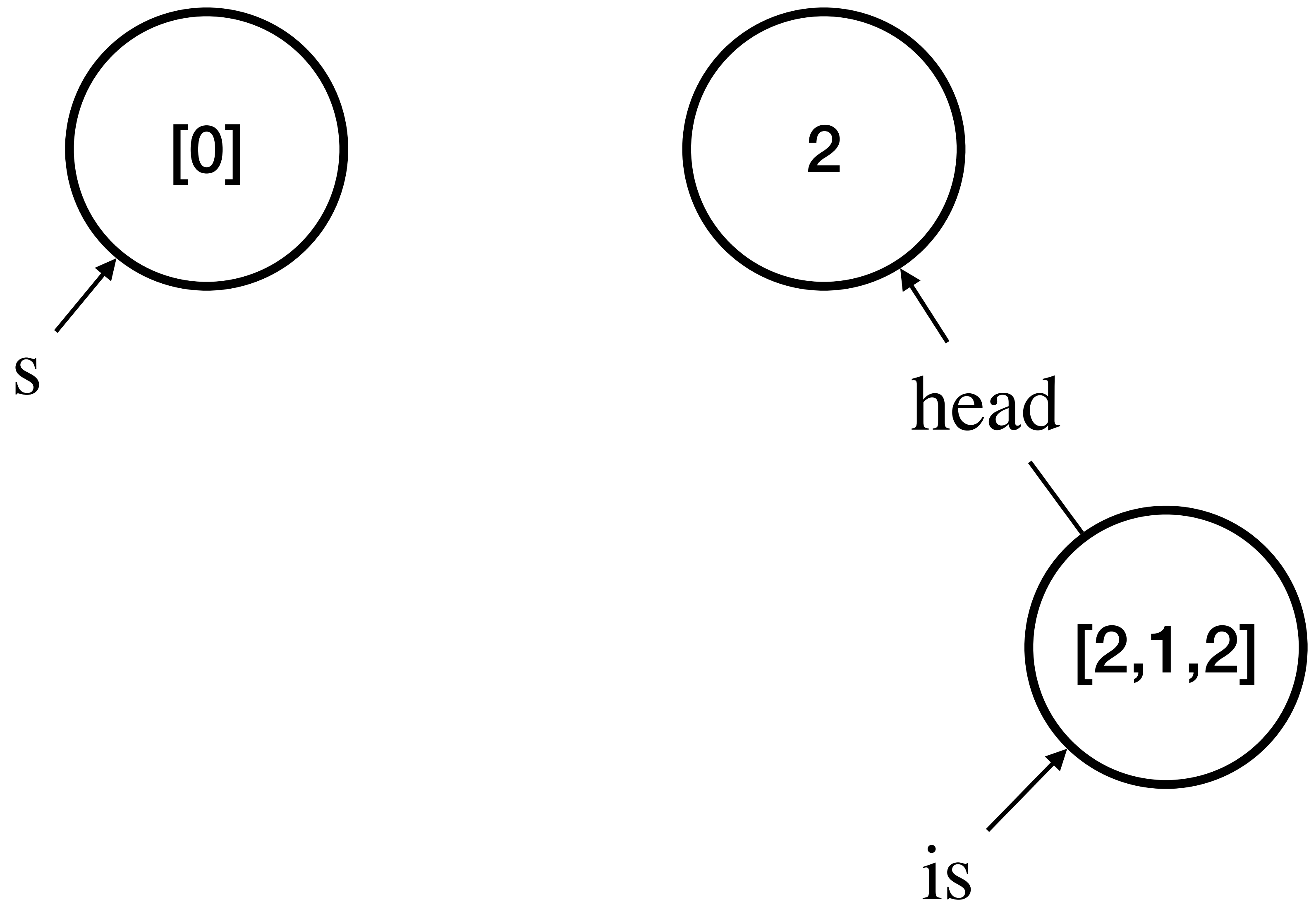
FTA Synthesis [Wang 2017]

```
→insert_all [0] [2,1,2] = [0,1,2]
 insert_all [0] [1,2] = [0,1,2]
 insert_all [0] [2] = [0,2]
 insert_all [0] [] = [0]
```



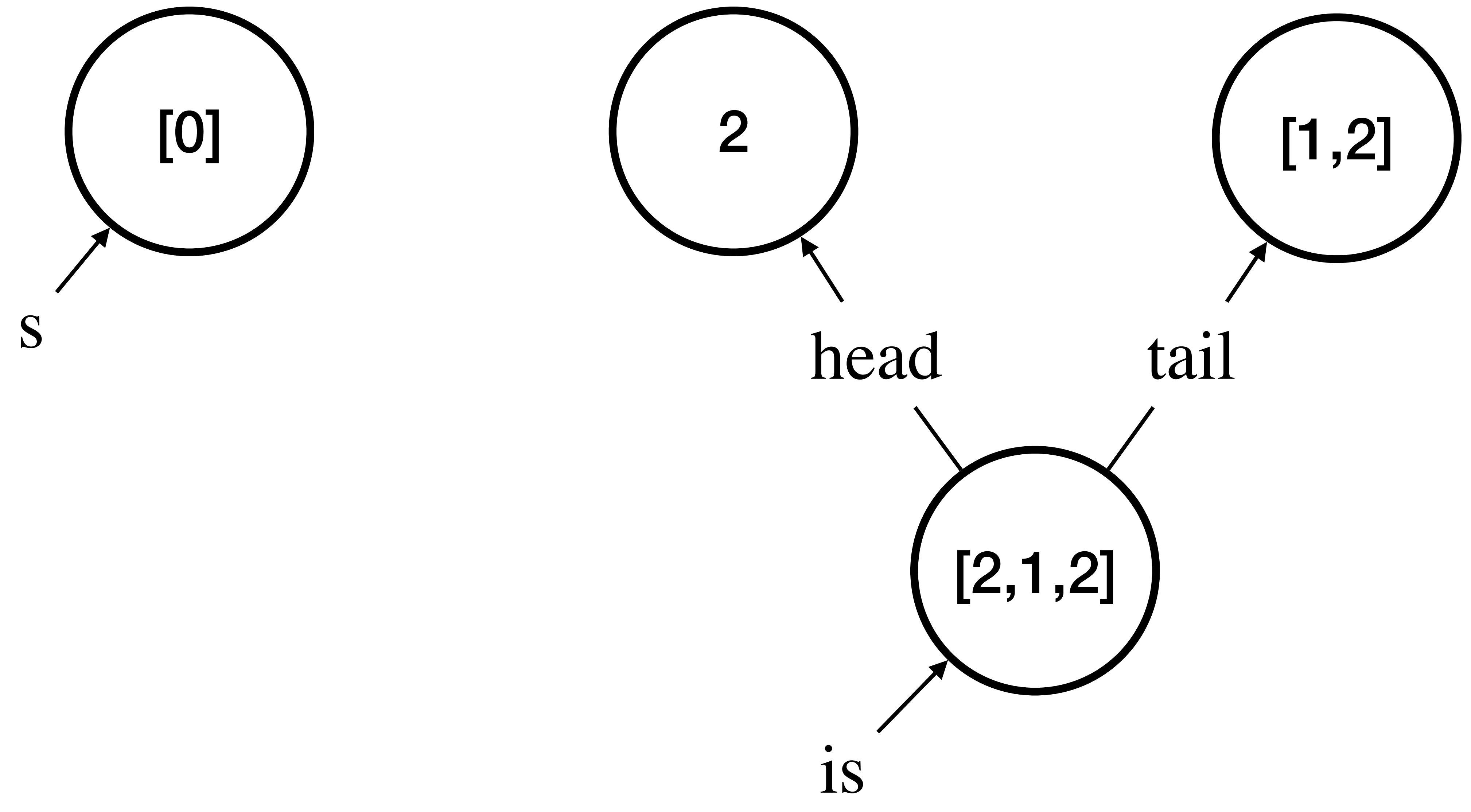
FTA Synthesis [Wang 2017]

```
→insert_all [0] [2,1,2] = [0,1,2]
 insert_all [0] [1,2] = [0,1,2]
 insert_all [0] [2] = [0,2]
 insert_all [0] [] = [0]
```



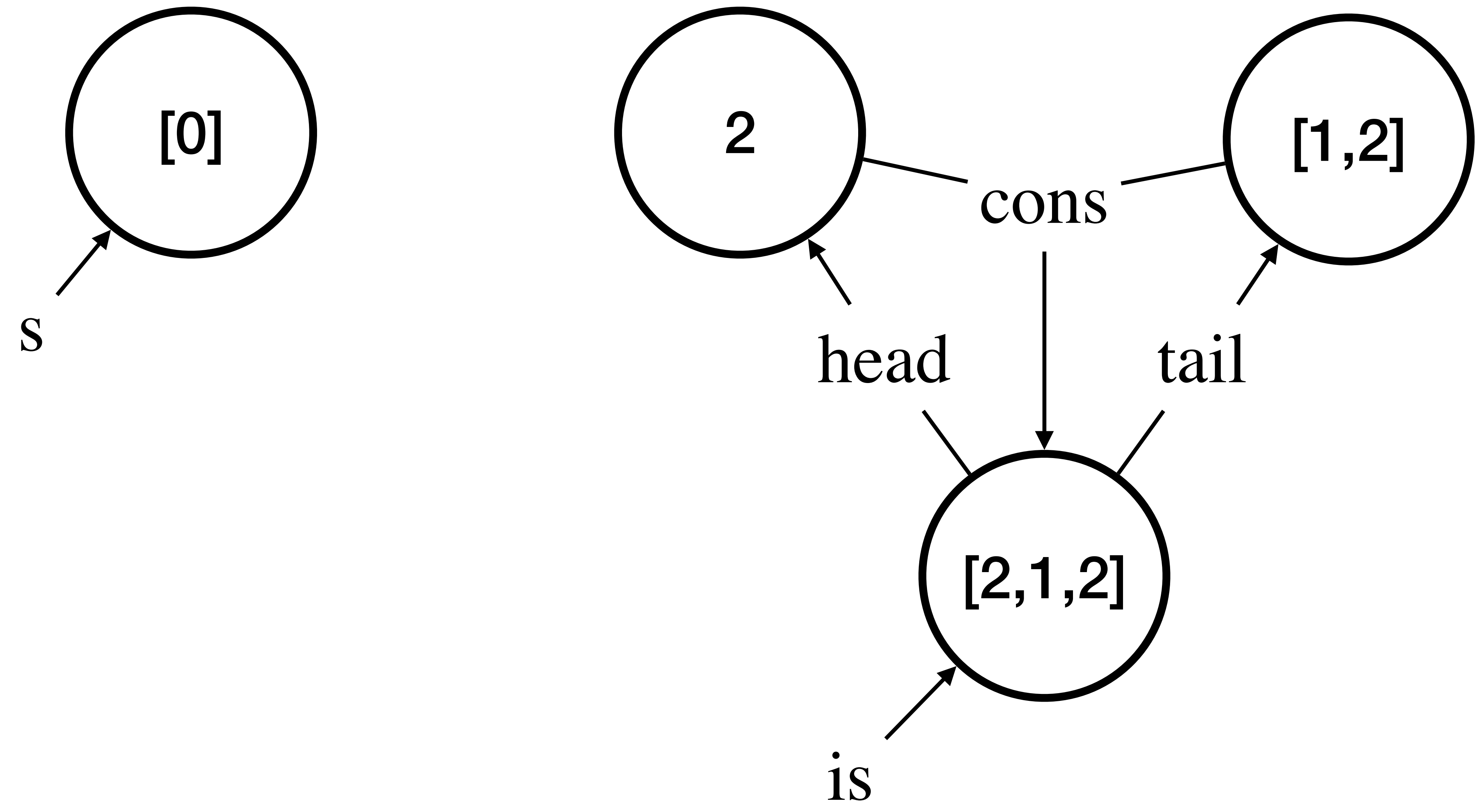
FTA Synthesis [Wang 2017]

```
→insert_all [0] [2,1,2] = [0,1,2]
 insert_all [0] [1,2] = [0,1,2]
 insert_all [0] [2] = [0,2]
 insert_all [0] [] = [0]
```



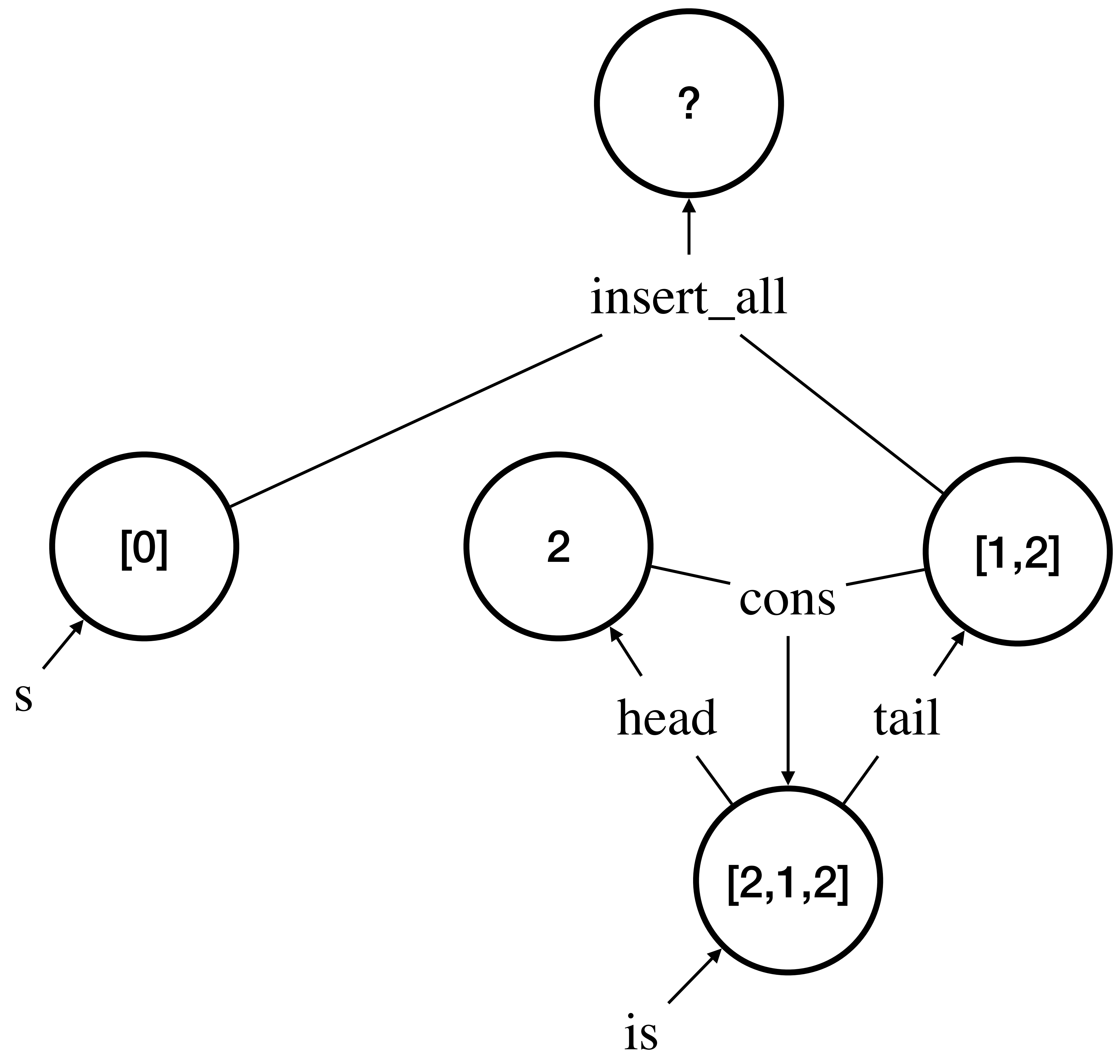
FTA Synthesis [Wang 2017]

```
→insert_all [0] [2,1,2] = [0,1,2]
 insert_all [0] [1,2] = [0,1,2]
 insert_all [0] [2] = [0,2]
 insert_all [0] [] = [0]
```



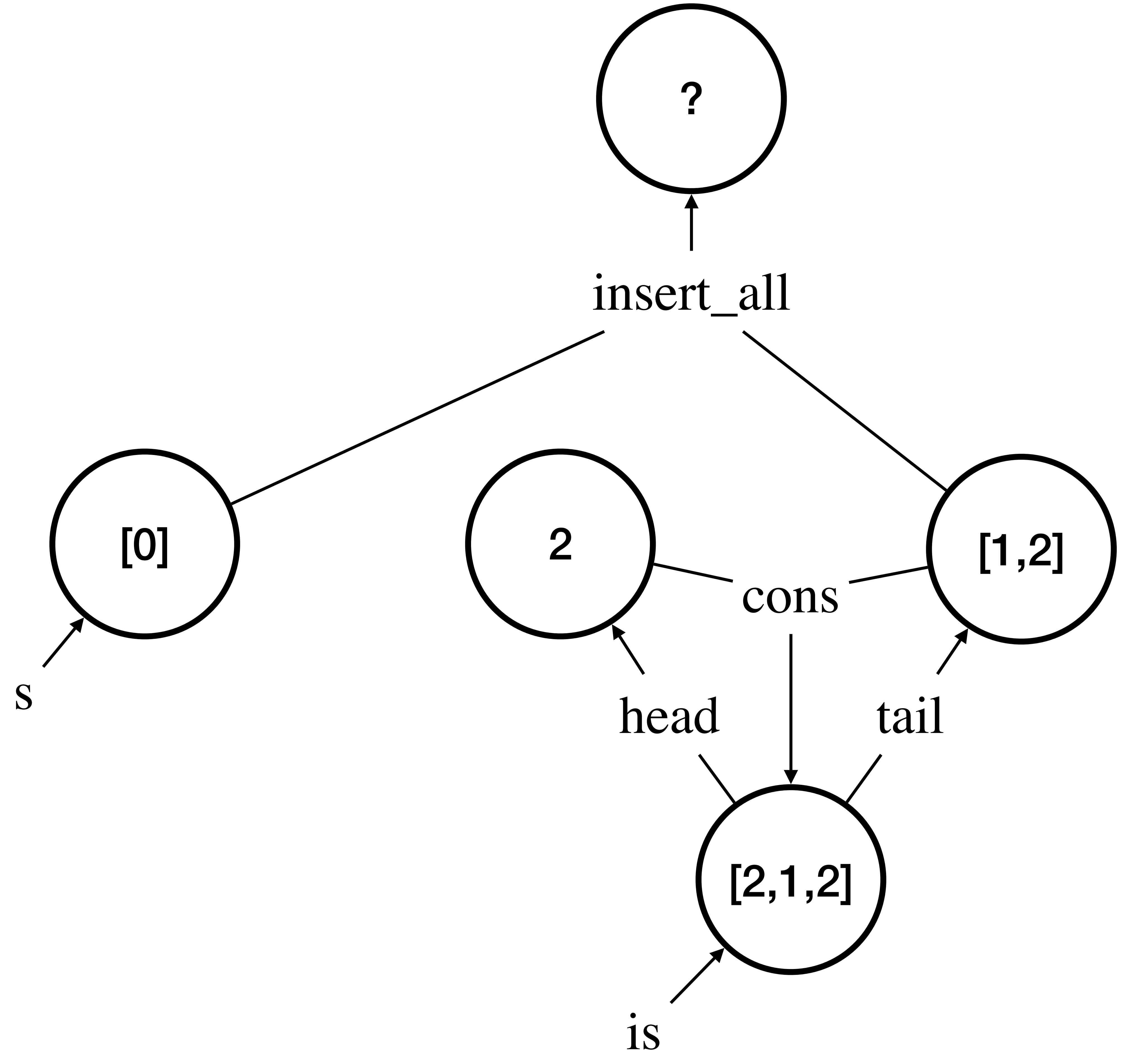
FTA Synthesis with Recursion

```
→insert_all [0] [2,1,2] = [0,1,2]
 insert_all [0] [1,2] = [0,1,2]
 insert_all [0] [2] = [0,2]
 insert_all [0] [] = [0]
```



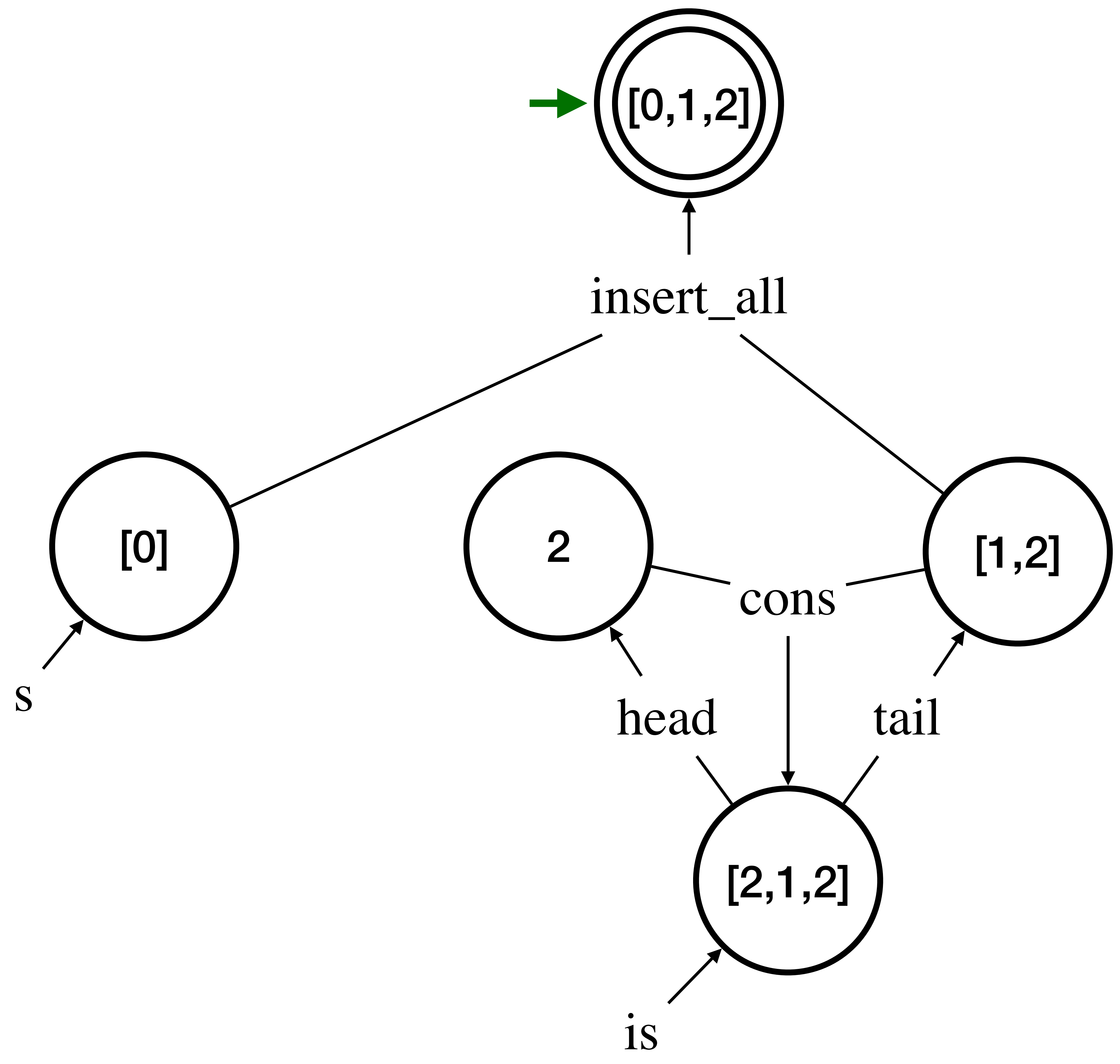
FTA Synthesis with Recursion

```
→insert_all [0] [2,1,2] = [0,1,2]
→insert_all [0] [1,2] = [0,1,2]
 insert_all [0] [2] = [0,2]
 insert_all [0] [] = [0]
```



FTA Synthesis with Recursion

```
→insert_all [0] [2,1,2] = [0,1,2]
→insert_all [0] [1,2] = [0,1,2]
 insert_all [0] [2] = [0,2]
 insert_all [0] [] = [0]
```



FTA Synthesis over Ground Specs

FTA Synthesis over Ground Specs

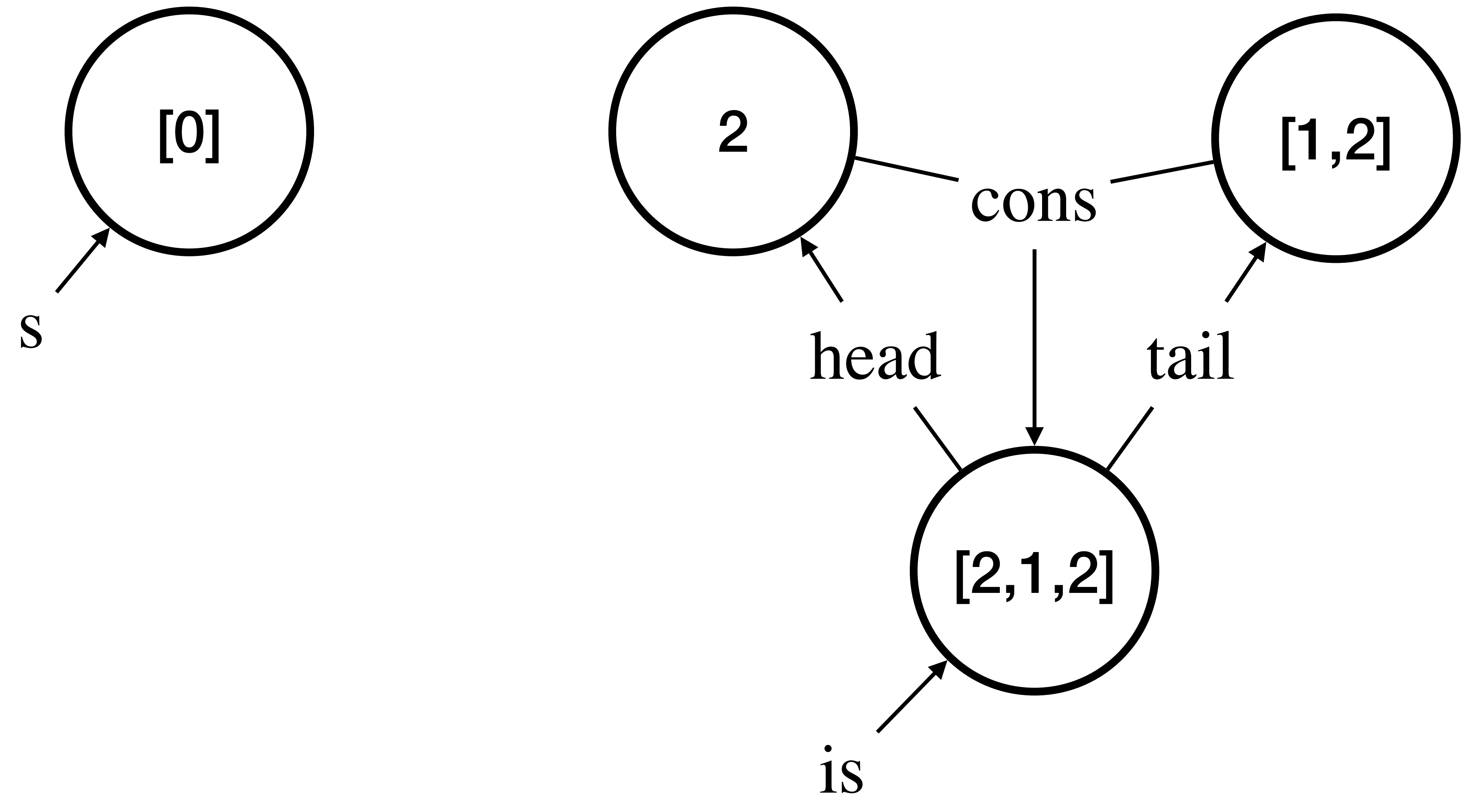
$\varphi_{[0],[2,1,2]}(out) =$

```
      (size out) = (size [0]) + (size (dedup [2,1,2]))
      ∧ inv out
```

FTA Synthesis over Ground Specs

$\varphi_{[0],[2,1,2]}(out) =$

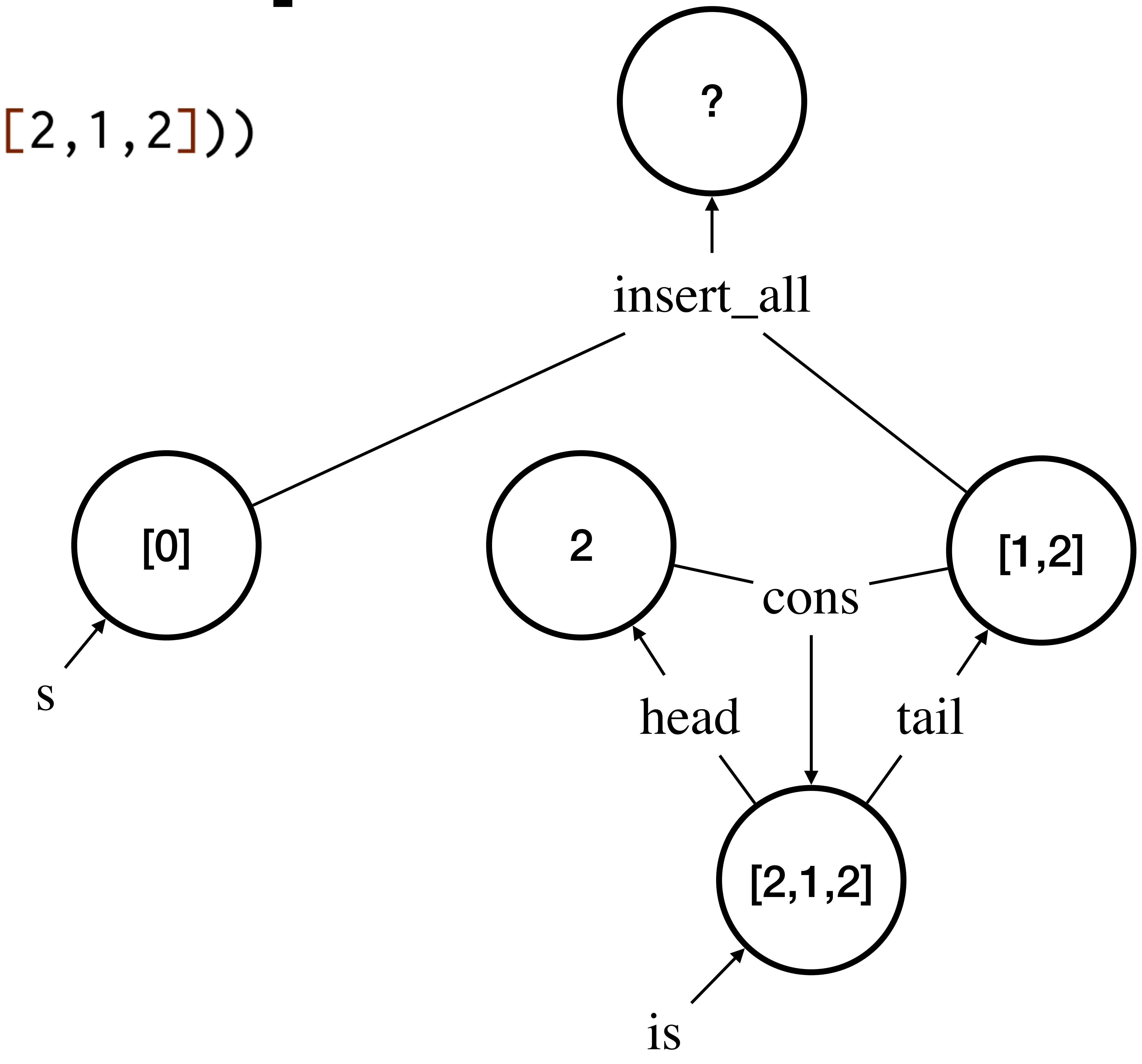
```
(size out) = (size [0]) + (size (dedup [2,1,2]))
∧ inv out
```



FTA Synthesis over Ground Specs

$\varphi_{[0],[2,1,2]}(out) =$

```
(size out) = (size [0]) + (size (dedup [2,1,2]))
∧ inv out
```

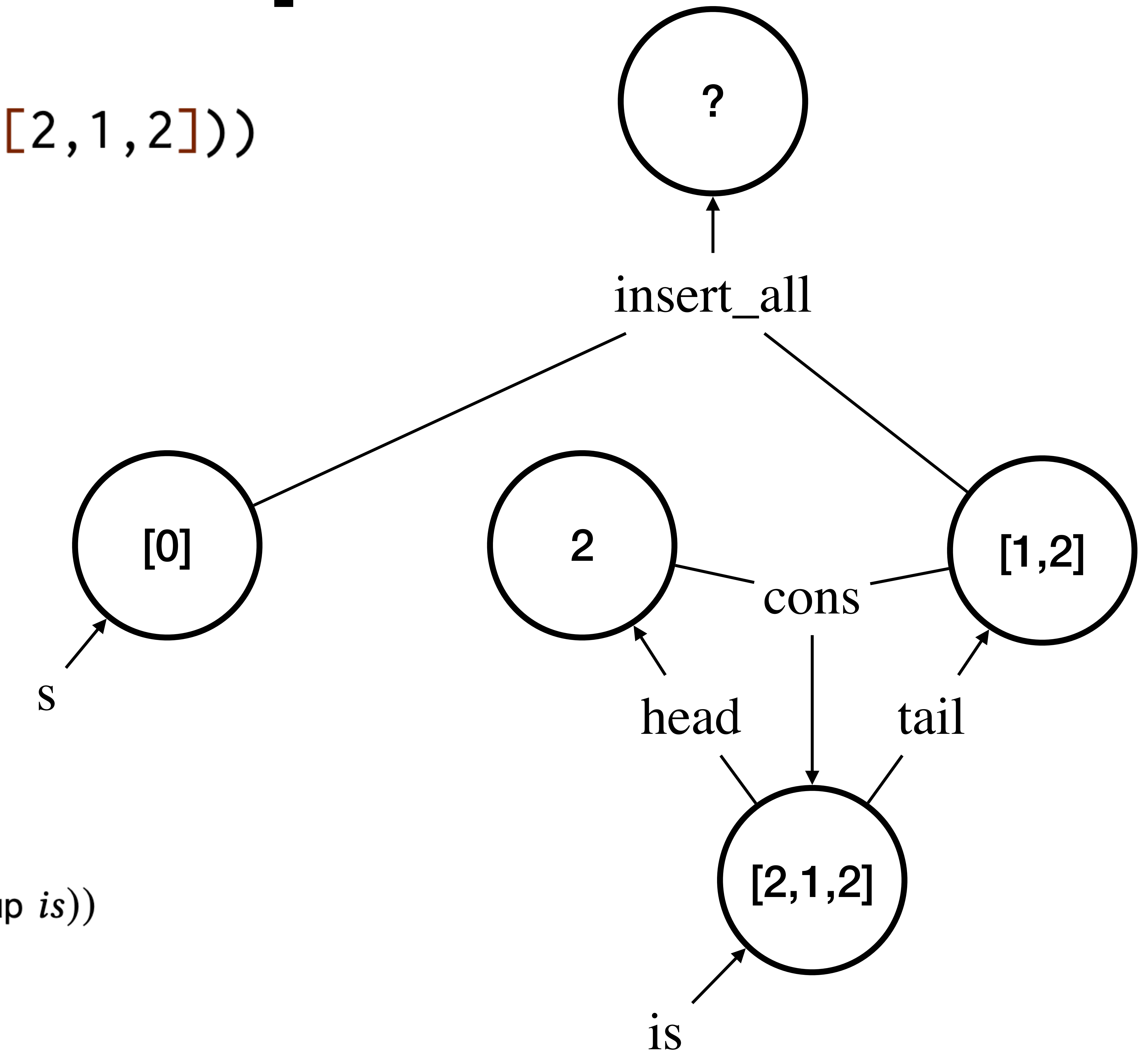


FTA Synthesis over Ground Specs

$\varphi_{[0],[2,1,2]}(out) =$

$\quad (\texttt{size}\ out) = (\texttt{size [0]}) + (\texttt{size (dedup [2,1,2])})$

$\quad \wedge\ inv\ \texttt{out}$



$$\begin{aligned}\forall is : \texttt{int list}. \forall s : \texttt{t}. \forall out : \texttt{t}.\quad & out = (\texttt{insert\_all}\ s\ is) \\ \Rightarrow\ & (inv\ s) \\ \Rightarrow\ & ((\texttt{size}\ out) = (\texttt{size}\ s) + (\texttt{size}\ \ (\texttt{dedup}\ is)) \\ \wedge\ & (inv\ out))\end{aligned}$$

We Add Everything.

$$SAT(\chi \wedge (\texttt{insert\_all [0] [1,2] = } l))$$

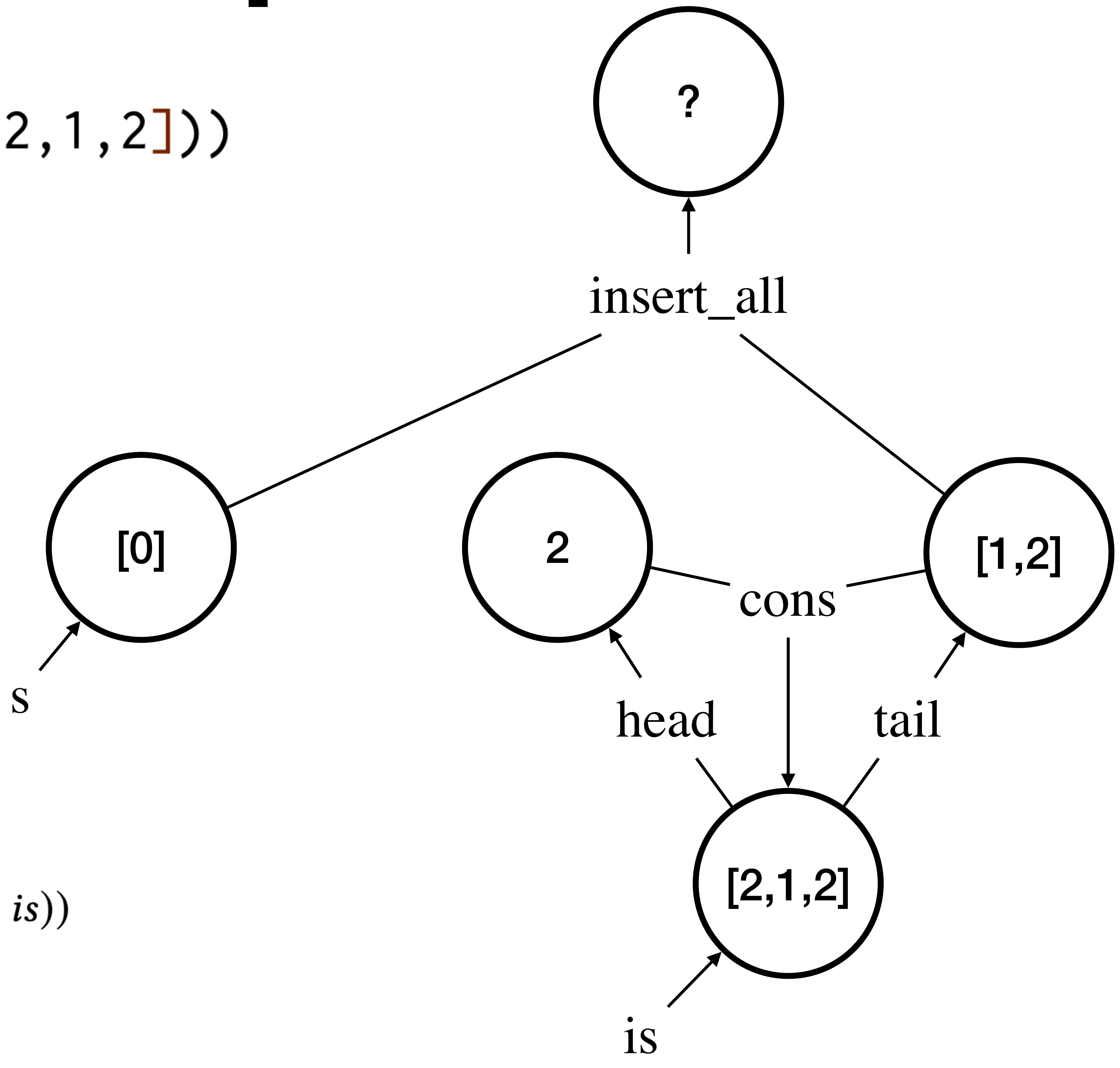
where χ is the ground specification

FTA Synthesis over Ground Specs

$$\varphi_{[0],[2,1,2]}(out) =$$

$$\quad (\texttt{size } out) = (\texttt{size } [0]) + (\texttt{size } (\texttt{dedup } [2,1,2]))$$

$$\quad \wedge\ inv\ \texttt{out}$$



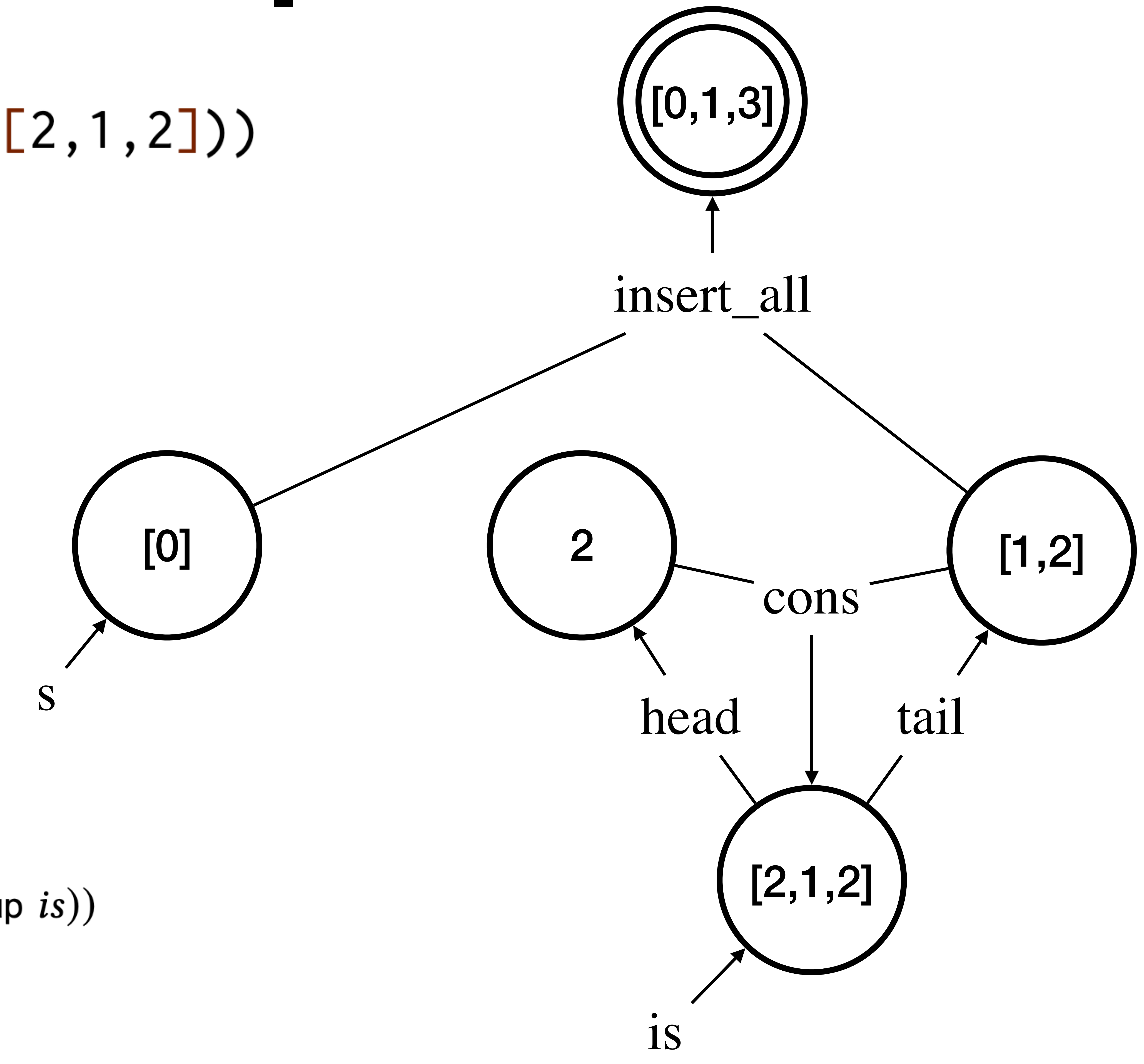
$$\forall is : \texttt{int list}. \forall s : \texttt{t}. \forall out : \texttt{t}. \quad out = (\texttt{insert\_all}\ s\ is)$$
$$\Rightarrow (inv\ s)$$
$$\Rightarrow ((\texttt{size}\ out) = (\texttt{size}\ s) + (\texttt{size}\ (\texttt{dedup}\ is))$$
$$\wedge\ (inv\ out))$$

FTA Synthesis over Ground Specs

$\varphi_{[0],[2,1,2]}(out) =$

`(size` out`) = (size [0]) + (size (dedup [2,1,2]))`

$\wedge$ inv `out`



$\forall is : \texttt{int list}. \forall s : \texttt{t}. \forall out : \texttt{t}.\quad out = (\texttt{insert\_all}\ s\ is)$

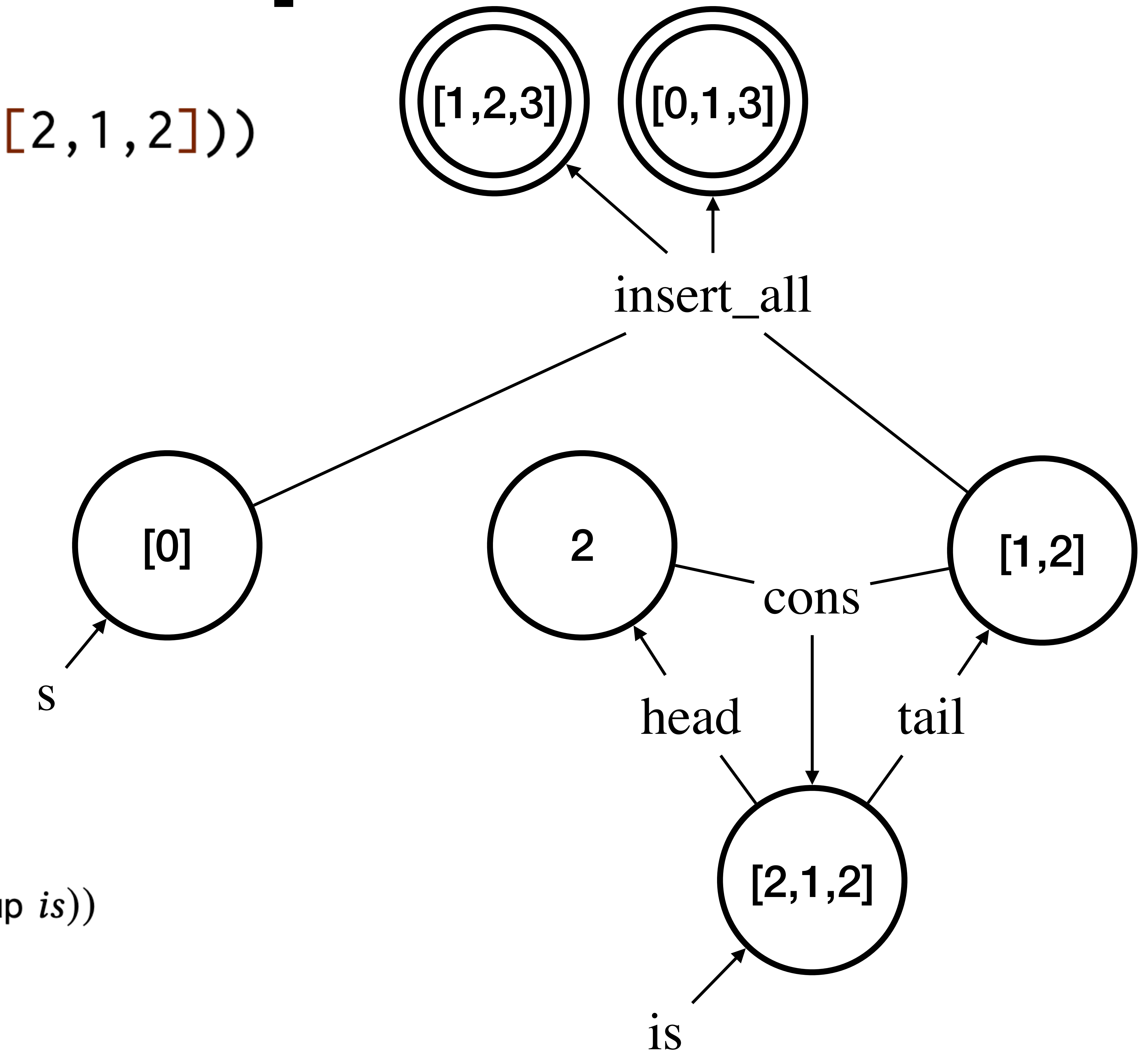
$\Rightarrow (inv\ s)$

$\Rightarrow ((\texttt{size}\ out) = (\texttt{size}\ s) + (\texttt{size}\ (\texttt{dedup}\ is))$

$\wedge\ (inv\ out))$

FTA Synthesis over Ground Specs

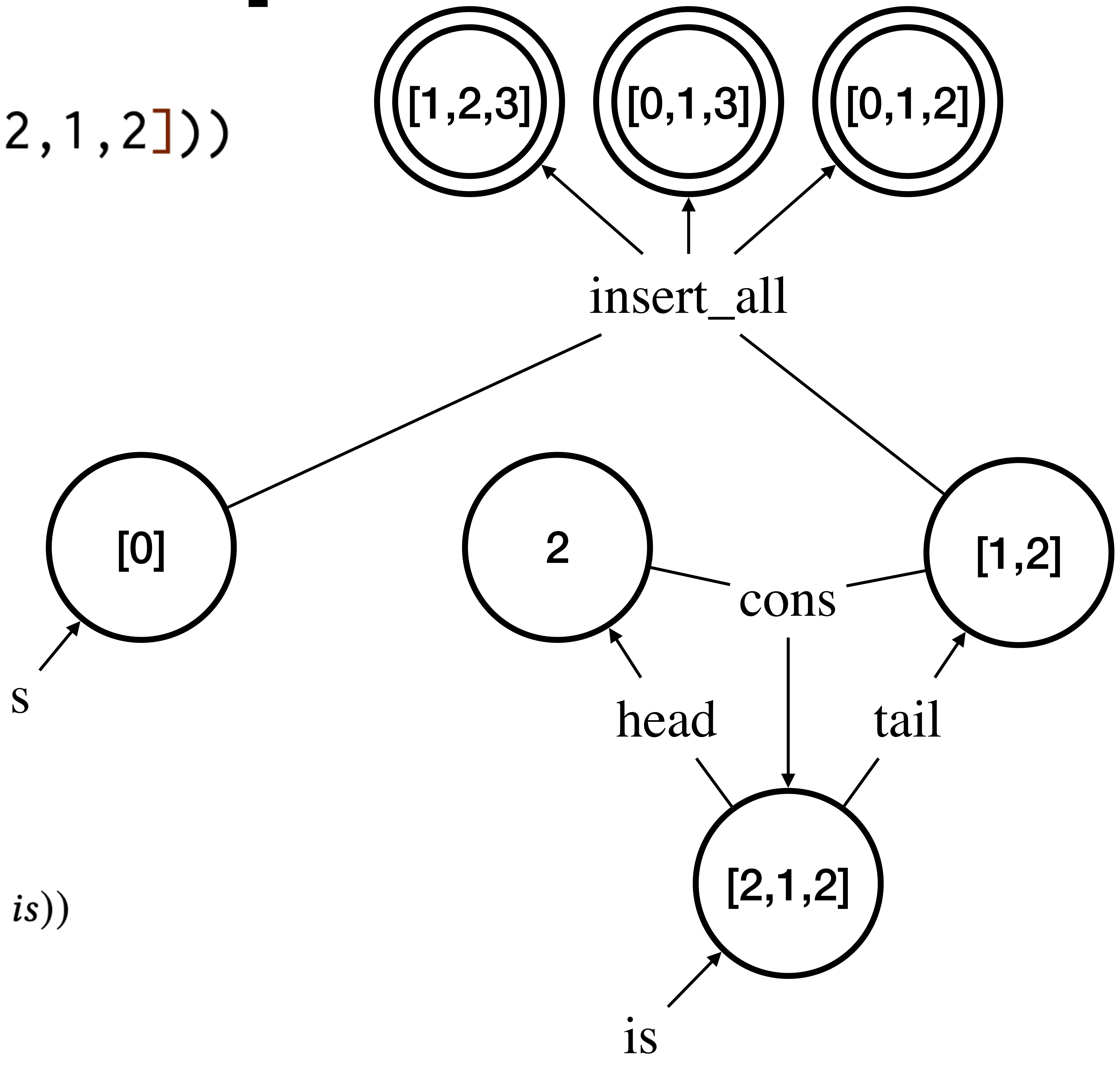
$\varphi_{[0],[2,1,2]}(out) =$
  `(size` out`) = (size [0]) + (size (dedup [2,1,2]))`
  $\wedge$ inv `out`



$\forall is$: `int list`. $\forall s$: `t`. $\forall out$: `t`. out = (`insert_all` s is)
$\Rightarrow$ (inv s)
$\Rightarrow$ ((`size` out) = (`size` s) + (`size` (`dedup` is))
$\wedge$ (inv out))

FTA Synthesis over Ground Specs

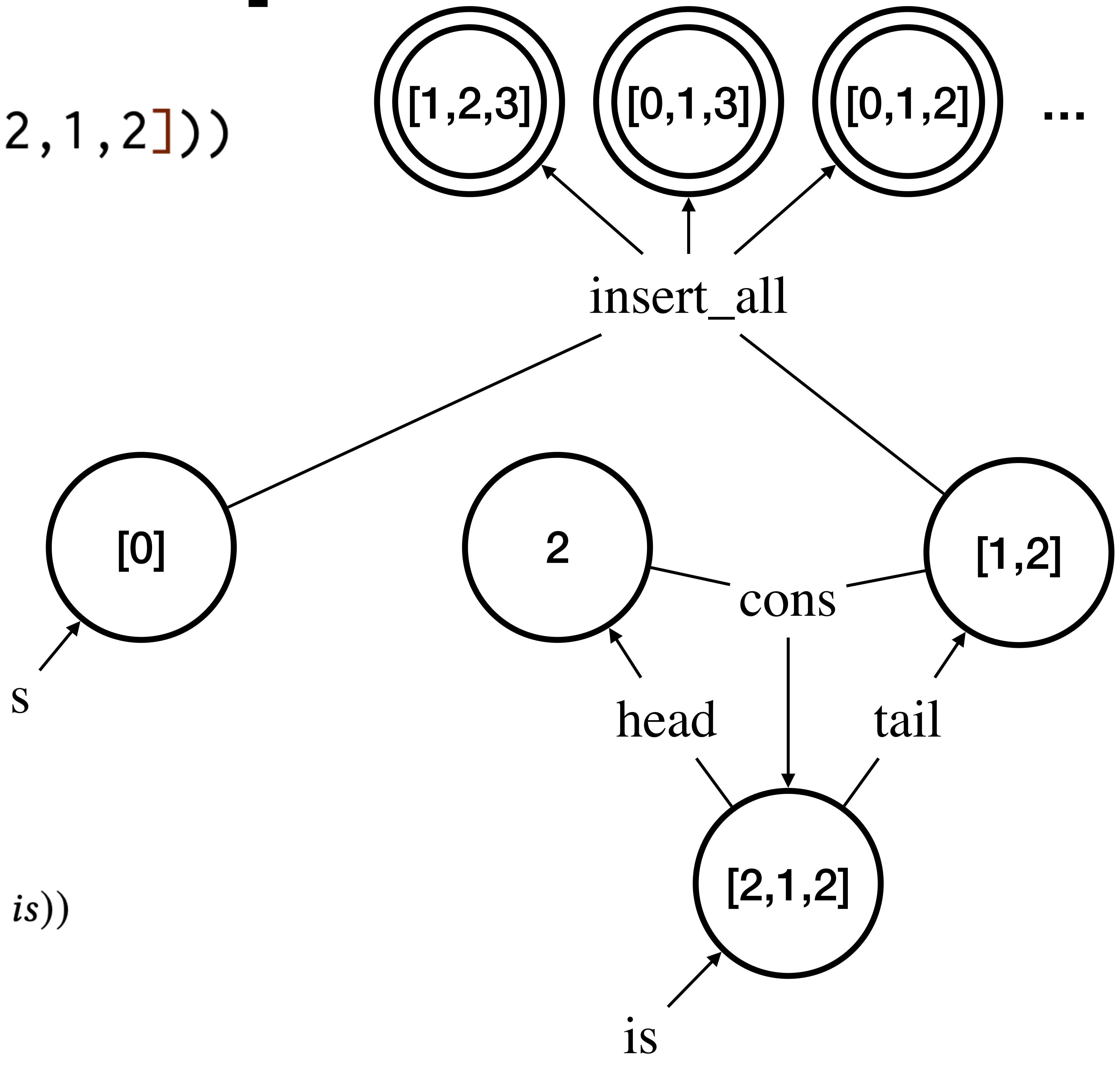
$$\varphi_{[0],[2,1,2]}(out) = \\ \quad (\texttt{size}\ out) = (\texttt{size}\ [0]) + (\texttt{size}\ (\texttt{dedup}\ [2,1,2])) \\ \quad \wedge\ inv\ \texttt{out}$$



$$\forall is : \texttt{int list}.\forall s : \texttt{t}.\forall out : \texttt{t}.\quad out = (\texttt{insert\_all}\ s\ is) \\ \Rightarrow (inv\ s) \\ \Rightarrow ((\texttt{size}\ out) = (\texttt{size}\ s) + (\texttt{size}\ (\texttt{dedup}\ is)) \\ \wedge\ (inv\ out))$$

FTA Synthesis over Ground Specs

$$\varphi_{[0],[2,1,2]}(out) = $$
$$(\texttt{size}\ out) = (\texttt{size}\ [0]) + (\texttt{size}\ (\texttt{dedup}\ [2,1,2]))$$
$$\wedge\ inv\ \texttt{out}$$



$$\forall is : \texttt{int list}. \forall s : \texttt{t}. \forall out : \texttt{t}. \quad out = (\texttt{insert\_all}\ s\ is)$$
$$\Rightarrow (inv\ s)$$
$$\Rightarrow ((\texttt{size}\ out) = (\texttt{size}\ s) + (\texttt{size}\ (\texttt{dedup}\ is))$$
$$\wedge\ (inv\ out))$$

Before, there was a meaning to our FTAs

$e \mapsto e'$ and $f \in A$ if, and only if $f\ e \rightarrow^* e'$

Is there any meaning to these FTAs?

Now, we ensure

$$SAT(\chi \land (\mathsf{f}\ \ e = e'))$$

and $f \in$ A

if, and only if

$$f\, e \rightarrow^{*}_{\chi} e'$$

Problem:

We are over approximating

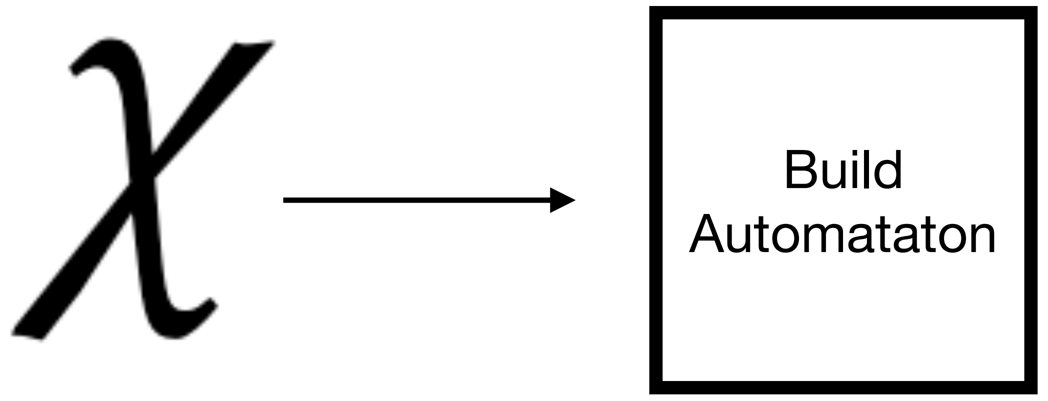
Solution:

Backtracking Search

Synthesis Algorithm

χ

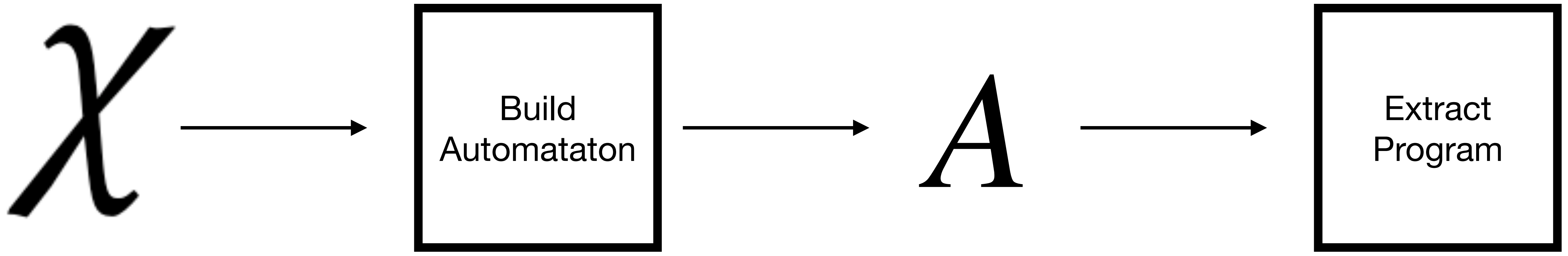
Synthesis Algorithm



Synthesis Algorithm

$$\chi \longrightarrow \boxed{\text{Build Automataton}} \longrightarrow A$$

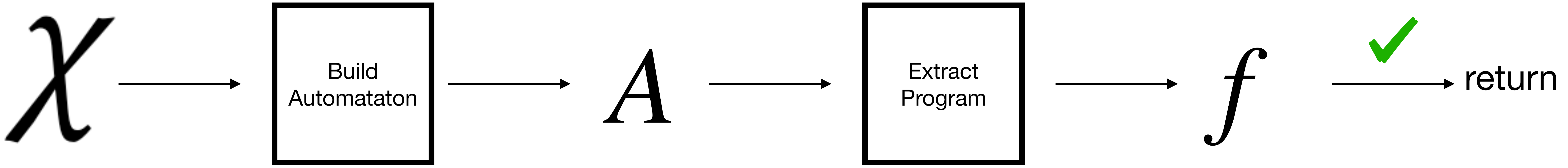
Synthesis Algorithm



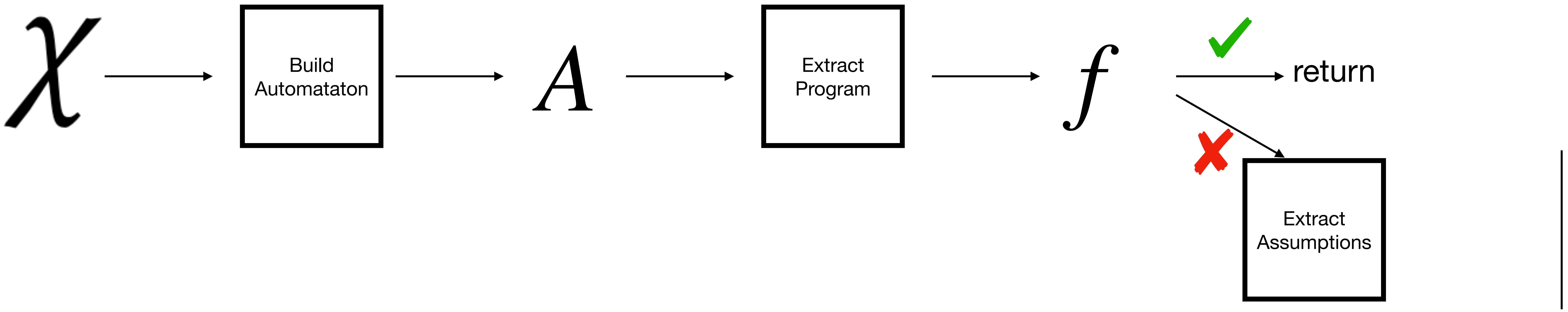
Synthesis Algorithm

$$\chi \longrightarrow \boxed{\begin{array}{c}\text{Build}\\ \text{Automataton}\end{array}} \longrightarrow A \longrightarrow \boxed{\begin{array}{c}\text{Extract}\\ \text{Program}\end{array}} \longrightarrow f$$

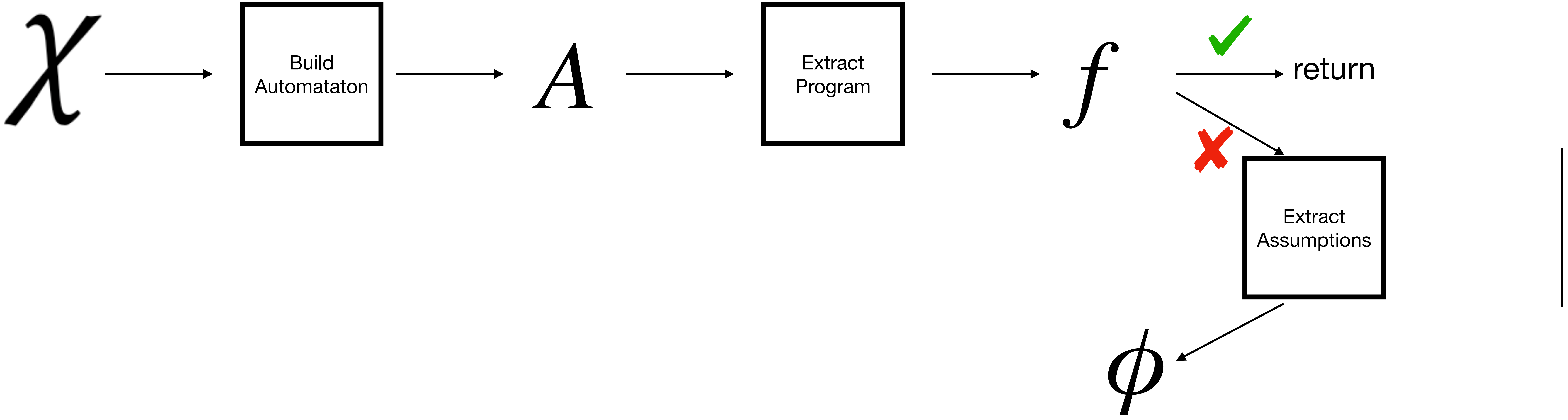
Synthesis Algorithm



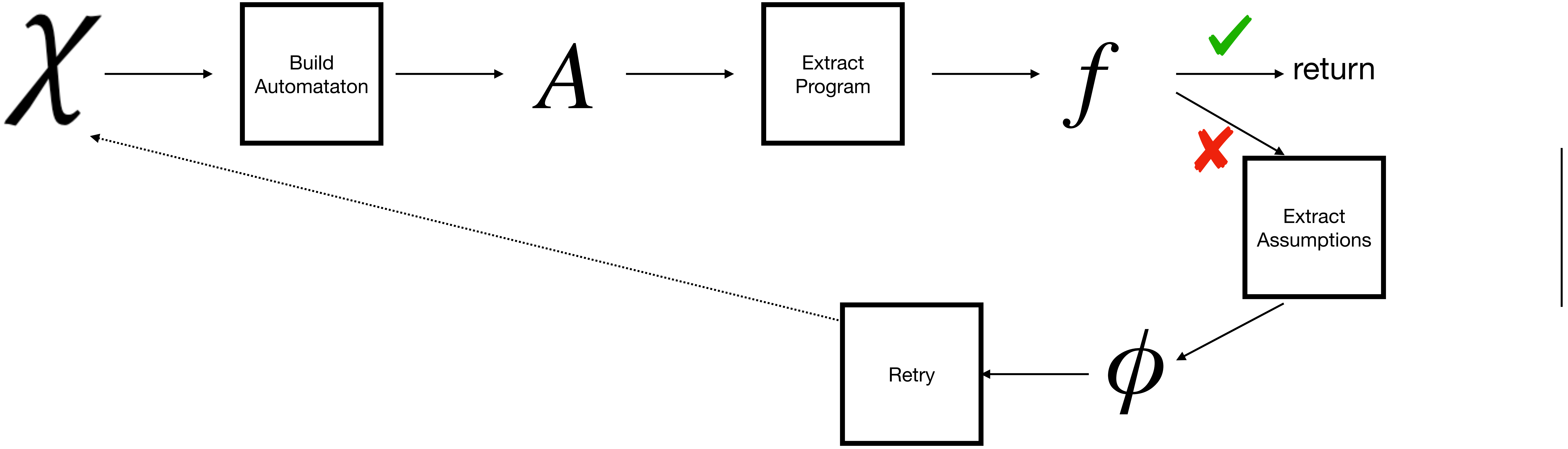
Synthesis Algorithm



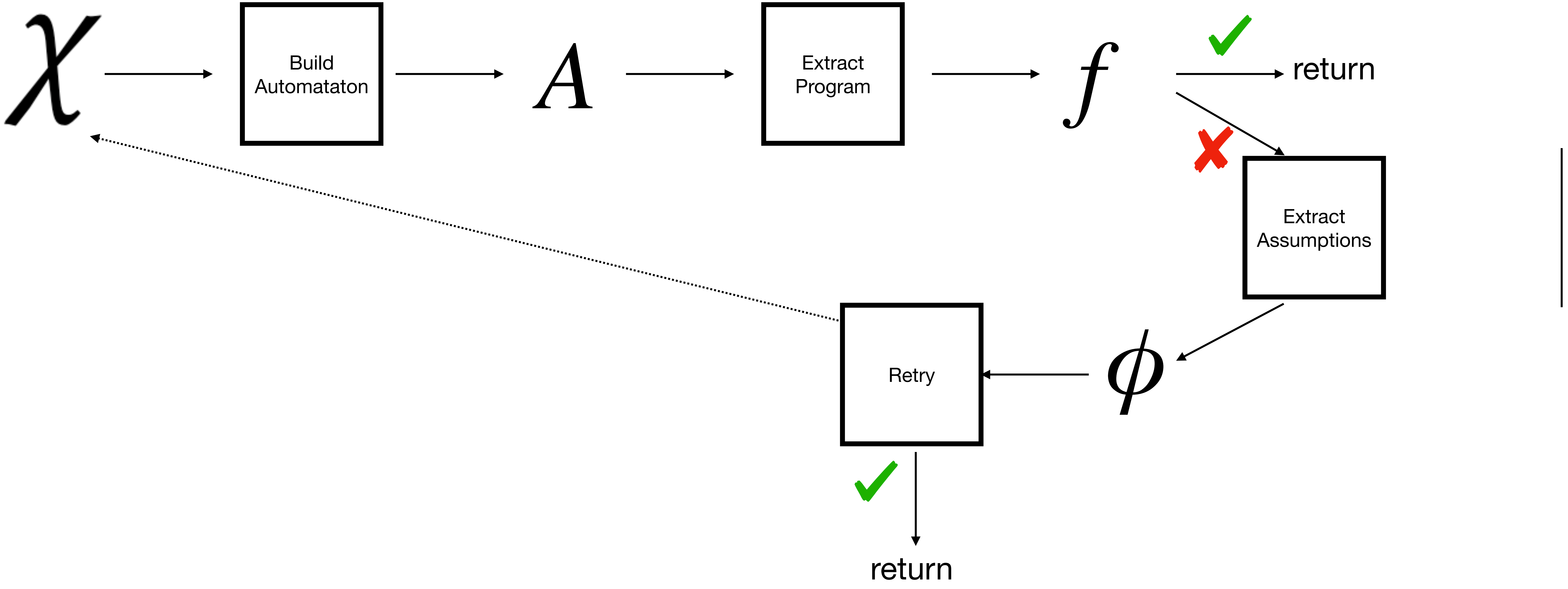
Synthesis Algorithm



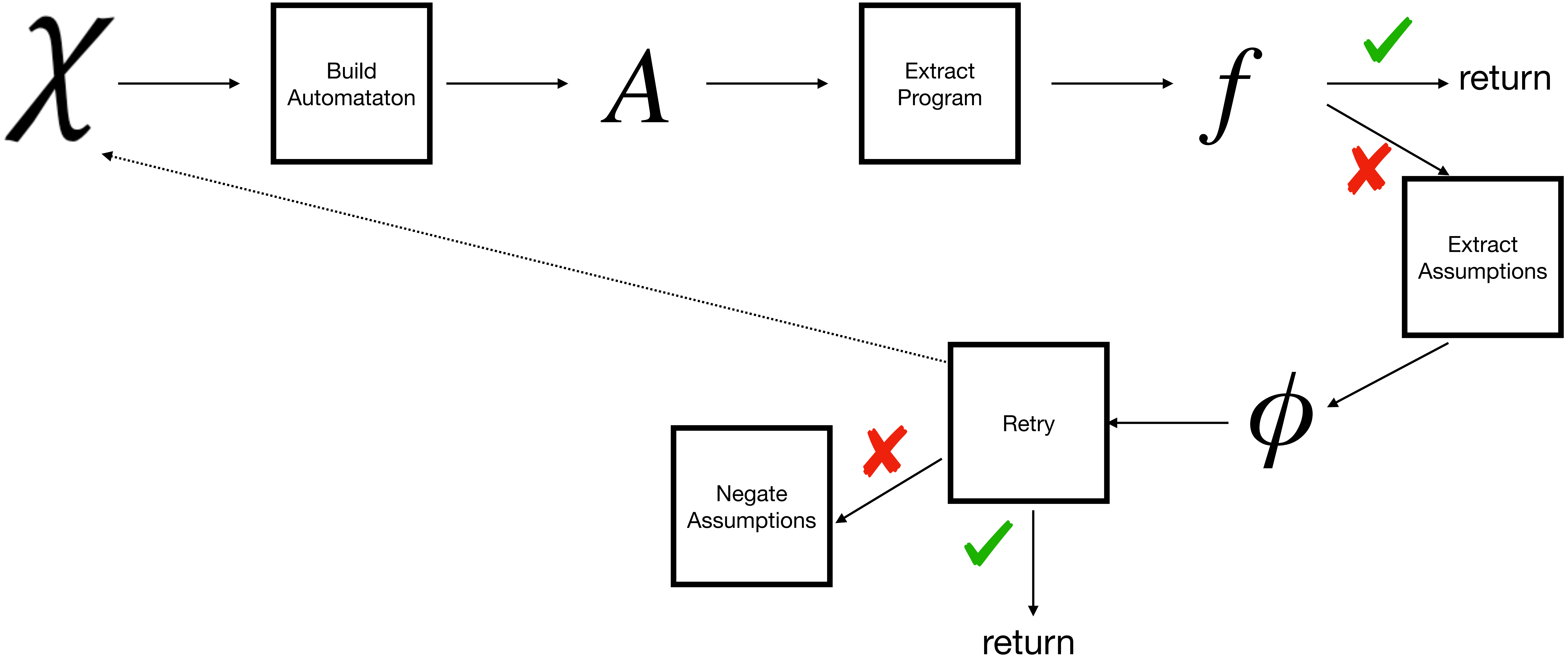
Synthesis Algorithm



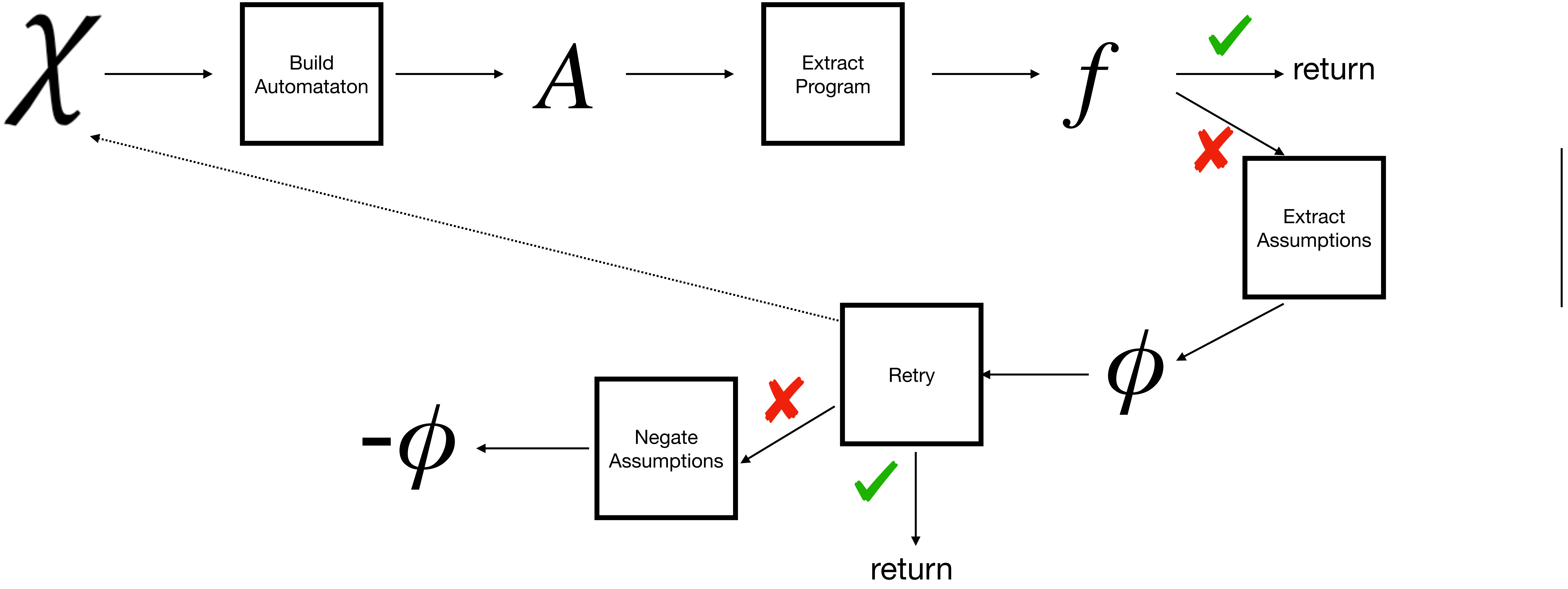
Synthesis Algorithm



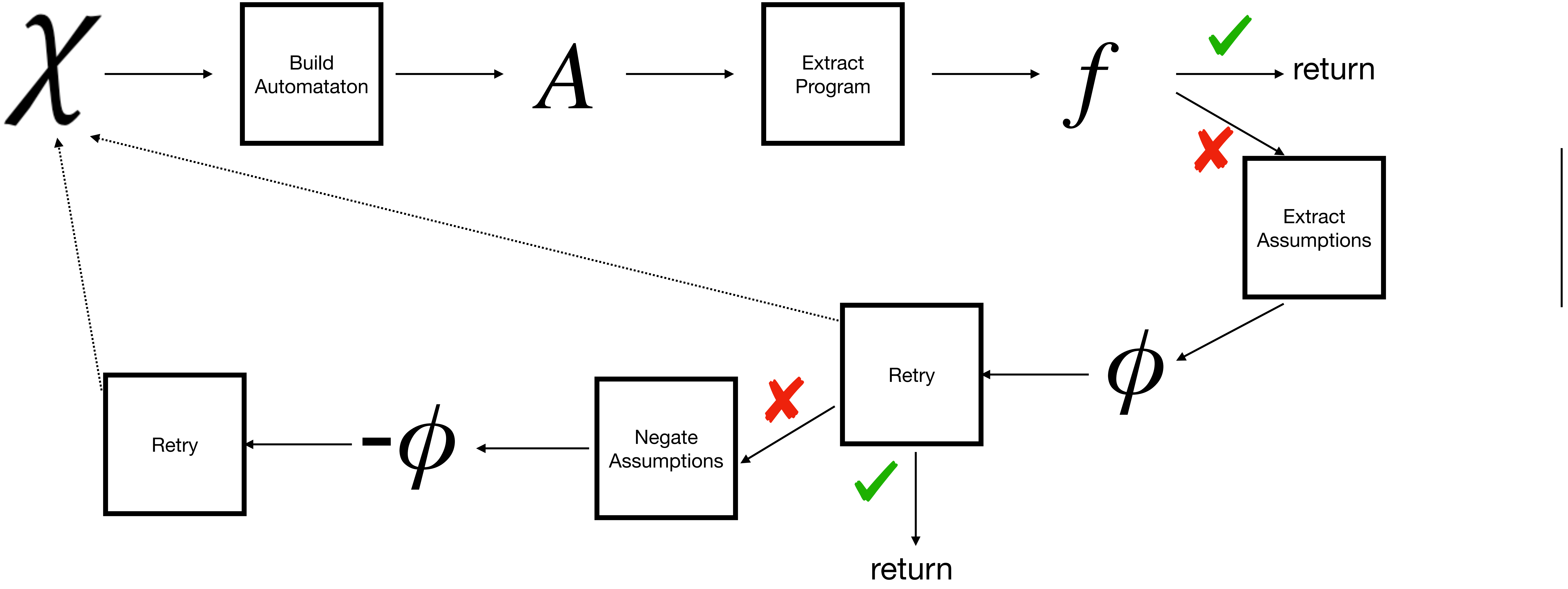
Synthesis Algorithm



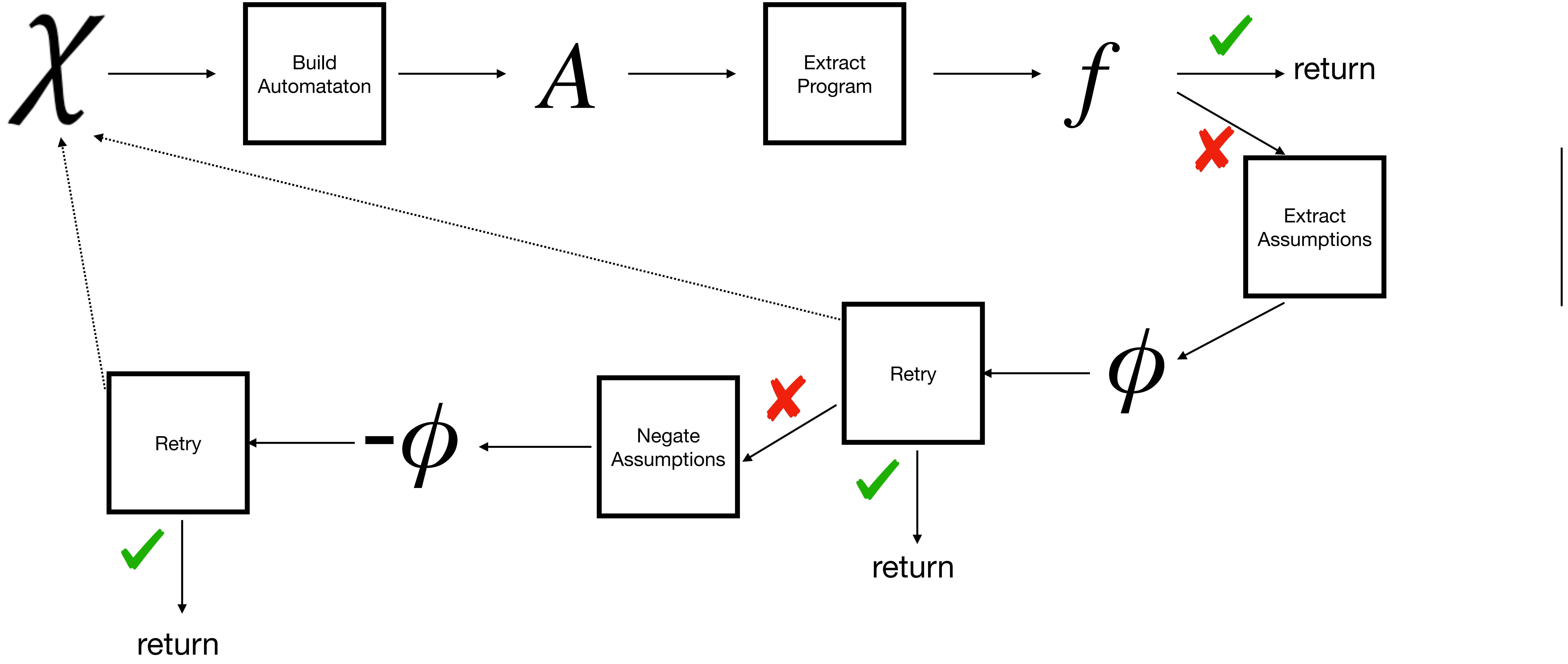
Synthesis Algorithm



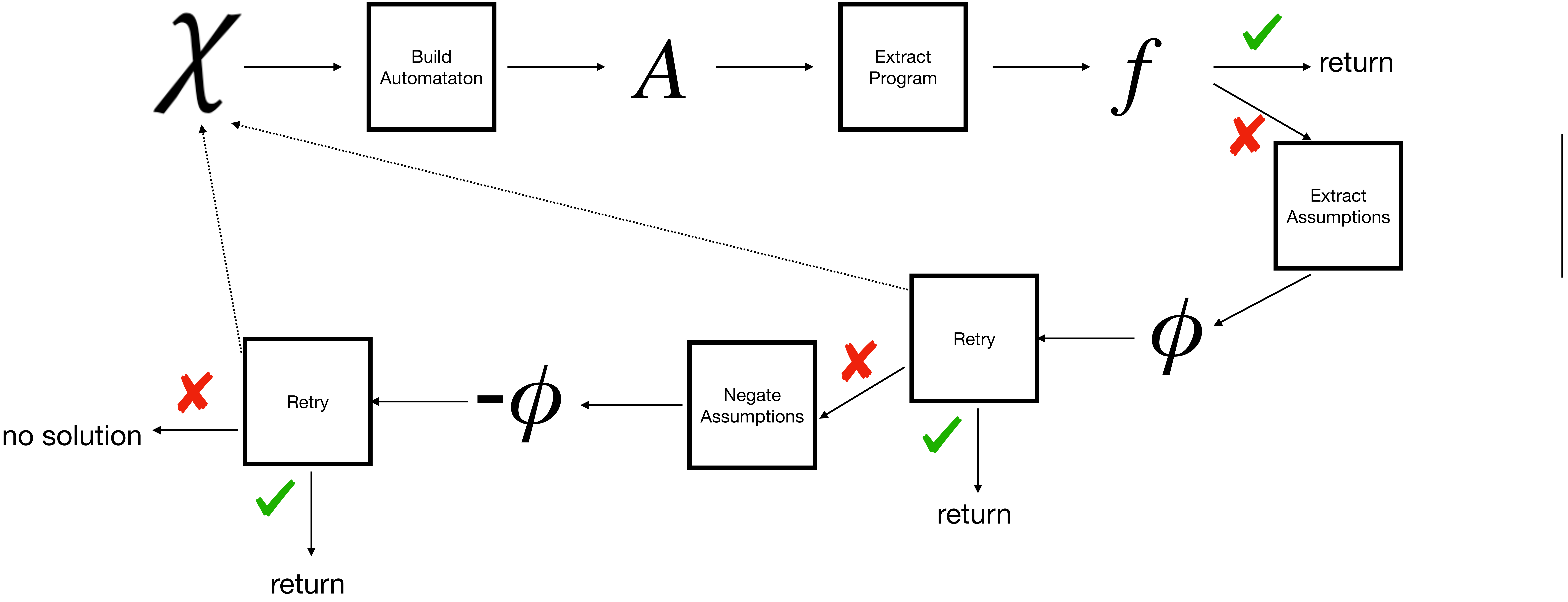
Synthesis Algorithm



Synthesis Algorithm



Synthesis Algorithm



Benchmark Suite Construction

- Myth Benchmark Suite (45)
  - Example-Based
  - Reimplementation
  - Logical Specification

Numbers

- Timeout of 2 minutes
- Inferred:
  - 43/45 for Example-based
  - 43/45 for Reimplementation
  - 41/45 for Logical

Future Work
Full Data Structure Synthesis

Future Work
Full Data Structure Synthesis

```
module type SET = sig
    type t
    val empty  : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
end
```

$$\forall i : \texttt{int}. \forall s : \texttt{t}. \forall s' : \texttt{t}. \quad \neg(\texttt{lookup empty } i) \\ \wedge\ (\texttt{lookup } (\texttt{insert } s\ i)\ i) \\ \wedge\ \neg(\texttt{lookup } (\texttt{delete } s\ i)\ i)$$

Future Work
Full Data Structure Synthesis

```
module type SET = sig
    type t
    val empty  : t
    val insert : t -> int -> t
    val delete : t -> int -> t
    val lookup : t -> int -> bool
end
```

$$\forall i : \mathtt{int}. \forall s : \mathtt{t}. \forall s' : \mathtt{t}. \quad \neg(\mathtt{lookup\ empty}\ i) \wedge (\mathtt{lookup}\ (\mathtt{insert}\ s\ i)\ i) \wedge \neg(\mathtt{lookup}\ (\mathtt{delete}\ s\ i)\ i)$$

```
module ListSet : SET = struct
  type t = int list

  let empty = []

  let rec lookup l x =
    match l with
    | [] -> false
    | hd :: tl -> (hd = x) || (lookup tl x)

  let insert l x =
    if (lookup l x) then l else (x :: l)

  let rec delete l x =
    match l with
    | [] -> []
    | hd :: tl -> if (hd = x) then tl
                  else (hd :: (delete tl x))
end
```

Future Work
Full Data Structure Synthesis

$$
\begin{aligned}
\forall i : \texttt{int}.\forall s : \texttt{t}.\forall s' : \texttt{t}. \quad & \neg(\mathrm{lookup\ empty}\ i) \\
\wedge\ & (\mathrm{lookup}\ (\mathtt{insert}\ s\ i)\ \ i) \\
\wedge\ & \neg(\mathrm{lookup}\ (\mathtt{delete}\ s\ i)\ \ i) \\
\wedge\ & (\mathrm{lookup}\ i\ s) \Rightarrow (\mathrm{lookup}\ i\ \ (\mathtt{union}\ s\ s'))
\end{aligned}
$$

Future Work
Full Data Structure Synthesis

$$
\begin{aligned}
\forall i : \texttt{int}. \forall s : \texttt{t}. \forall s' : \texttt{t}. \quad & \neg(\texttt{lookup empty } i) \\
\wedge\ & (\texttt{lookup } (\texttt{insert } s\ i)\ i) \\
\wedge\ & \neg(\texttt{lookup } (\texttt{delete } s\ i)\ i) \\
\wedge\ & (\texttt{lookup } i\ s) \Rightarrow (\texttt{lookup } i\ (\texttt{union } s\ s'))
\end{aligned}
$$

This is a relational specification

Future Work
Full Data Structure Synthesis

$$
\begin{aligned}
\forall i : \mathtt{int}. \forall s : \mathtt{t}. \forall s' : \mathtt{t}. \quad & \neg\,(\mathtt{lookup}\ \mathtt{empty}\ i) \\
\wedge\ & (\mathtt{lookup}\ \ (\mathtt{insert}\ s\ i)\ \ i) \\
\wedge\ & \neg\,(\mathtt{lookup}\ \ (\mathtt{delete}\ s\ i)\ \ i) \\
\wedge\ & (\mathtt{lookup}\ i\ s) \Rightarrow (\mathtt{lookup}\ i\ \ (\mathtt{union}\ s\ s'))
\end{aligned}
$$

This is a relational specification

Relish [Wang 2018] synthesizes functions from relational specifications

Future Work
Full Data Structure Synthesis

$$
\begin{aligned}
\forall i : \mathtt{int}. \forall s : \mathtt{t}. \forall s' : \mathtt{t}. \quad & \neg(\mathtt{lookup}\ \mathtt{empty}\ i) \\
\wedge\ & (\mathtt{lookup}\ (\mathtt{insert}\ s\ i)\ \ i) \\
\wedge\ & \neg(\mathtt{lookup}\ (\mathtt{delete}\ s\ i)\ \ i) \\
\wedge\ & (\mathtt{lookup}\ i\ s) \Rightarrow (\mathtt{lookup}\ i\ \ (\mathtt{union}\ s\ s'))
\end{aligned}
$$

This is a relational specification

Relish [Wang 2018] synthesizes functions from relational specifications

Can we integrate relational specification synthesis with recursive synthesis?

Summary

- Data structure verification & synthesis is an important problem
- Find Representation Invariants with Hanoi
  - Visible inductiveness tames the ambiguity of inductiveness counterexamples
- Synthesize functions that respect the invariant with Burst
  - Angelic synthesis extends prior approaches to work with logical specifications

Collaborators Slide

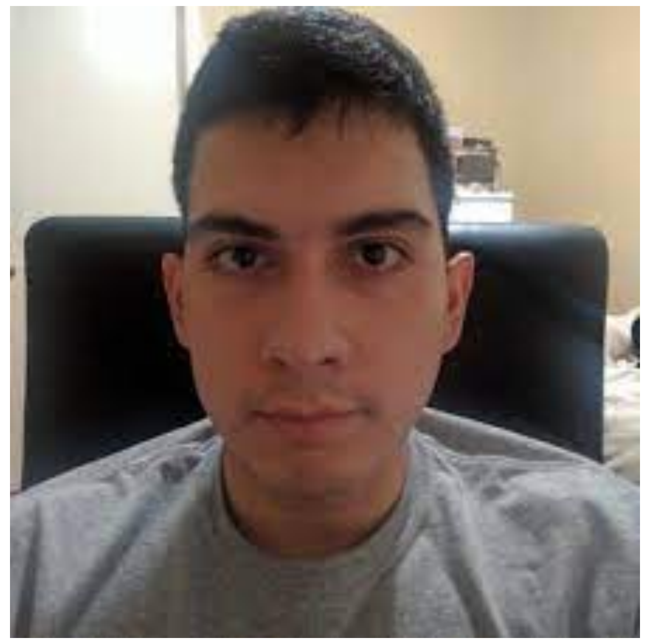
Verification


Saswat Padhi
(AWS)


Todd Millstein
(UCLA)


David Walker
(Princeton)

Synthesis


Adrian Trejo Nuñez
(UT Austin)


Ana Brendel
(UT Austin)


Swarat Chaudhuri
(UT Austin)


Işil Dillig
(UT Austin)

Summary

- Data structure verification & synthesis is an important problem
- Find Representation Invariants with Hanoi
  - Visible inductiveness tames the ambiguity of inductiveness counterexamples
- Synthesize functions that respect the invariant with Burst
  - Angelic synthesis extends prior approaches to work with logical specifications