Abstraction-Based Interaction Model for Synthesis

Hila Peleg, Technion
Shachar Itzhaky, Technion
Sharon Shoham, Tel Aviv University

The research leading to these results has received funding from the European Union's - Seventh Framework Programme (FP7) under grant agreement n° 615688 – ERC- COG-PRIME.

Programming by Example

Task: find the most frequent bigram in a string.

Lieberman, H. (2000). Programming by example. *Communications of the ACM*, *43*(3), 72-72.

Programming by Example

Task: find the most frequent bigram in a string.

"a**bd**fibfcfde**bd**fde**bd**ihgfkjfde**bd**"

⇓

"bd"

Lieberman, H. (2000). Programming by example. *Communications of the ACM*, *43*(3), 72-72.

Programming by Example

Task: find the most frequent bigram in a string.

"a**bd**fibfcfde**bd**fde**bd**ihgfkjfde**bd**"

⇓

"bd"

```
input.takeRight(2)
```

Lieberman, H. (2000). Programming by example. *Communications of the ACM*, *43*(3), 72-72.

Programming by Example

Task: find the most frequent bigram in a string.

"a**bd**fibfcfde**bd**fde**bd**ihgfkjfde**bd**"

⇓

"bd"

`input.takeRight(2)`

Lieberman, H. (2000). Programming by example. *Communications of the ACM*, *43*(3), 72-72.

Programming by Example

Task: find the most frequent bigram in a string.

```
"abdfibfcfdebdfdebdihgfkjfdebd"
 ⇓
"bd"
```

```
input.takeRight(2)
```

```
"abbba"
 ⇓
"bb"
```

Lieberman, H. (2000). Programming by example. *Communications of the ACM*, *43*(3), 72-72.

Programming by Example

Task: find the most frequent bigram in a string.

"a**bd**fibfcfde**bd**fde**bd**ihgfkjfde**bd**"
⇓
"bd"

`input.takeRight(2)`

"abbba"
⇓
"bb"

`input.substring(1,3)`

Lieberman, H. (2000). Programming by example. *Communications of the ACM*, *43*(3), 72-72.

Programming Not Only by Example

Input
⇓
Output

Peleg, H., Shoham, S., & Yahav, E. (2017). Programming Not Only by Example. *ICSE 2018 (Upcoming)*.

Programming Not Only by Example

$$\{m \in M \mid [\![m]\!](input) = output\}$$

Peleg, H., Shoham, S., & Yahav, E. (2017). Programming Not Only by Example. *ICSE 2018 (Upcoming)*.

Programming Not Only by Example

$$\{m \in M \mid [\![m]\!](input) = output\}$$

```
input.takeRight(2)
```

Peleg, H., Shoham, S., & Yahav, E. (2017). Programming Not Only by Example. *ICSE 2018 (Upcoming)*.

Programming Not Only by Example

$\{m \in M \mid [\![m]\!](input) = output\}$

`input.takeRight(2)`

Exclude programs with `takeRight`

Peleg, H., Shoham, S., & Yahav, E. (2017). Programming Not Only by Example. *ICSE 2018 (Upcoming)*.

Programming Not Only by Example

$\{m \in M \mid \textcolor{red}{\boldsymbol{p}}(m)\}$

`input.takeRight(2)`

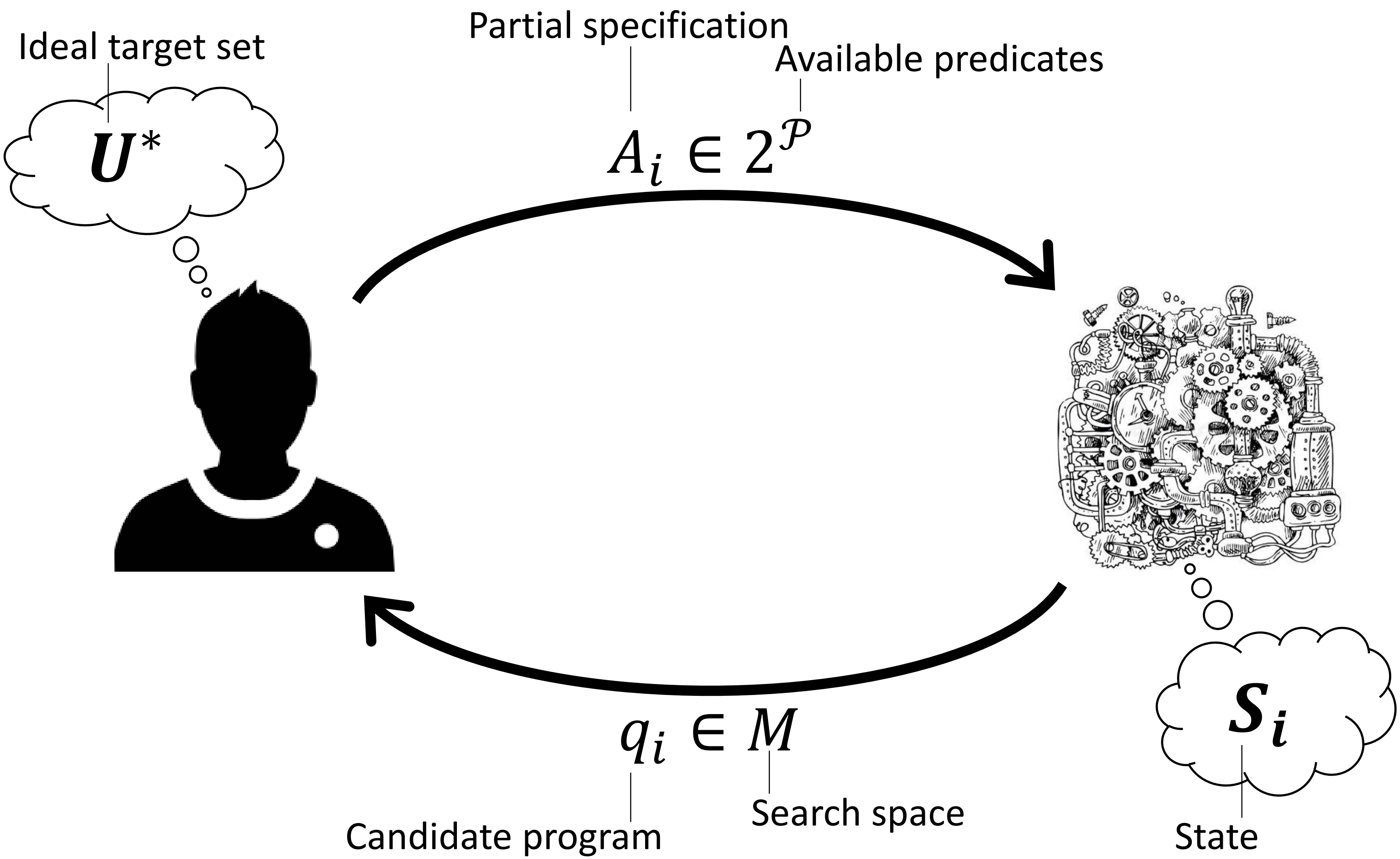
Exclude programs with `takeRight`

Peleg, H., Shoham, S., & Yahav, E. (2017). Programming Not Only by Example. *ICSE 2018 (Upcoming)*.

Our Goal

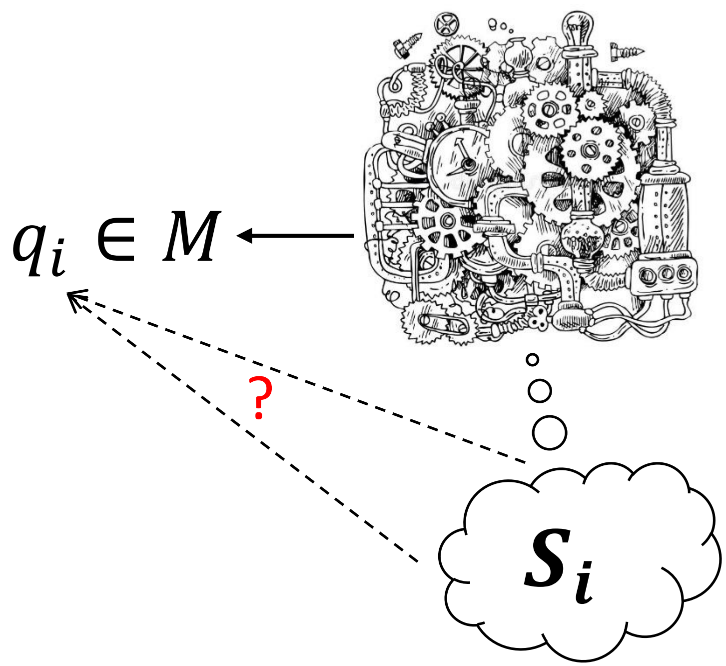
- To model user-driven synthesis
  - Works in practice but we do not understand its limitations
- Properties
  - Of the synthesizer
  - Of the user
- Guarantees
  - Termination (in paper)
  - Are “bad sessions” recoverable

Iterative, interactive synthesis



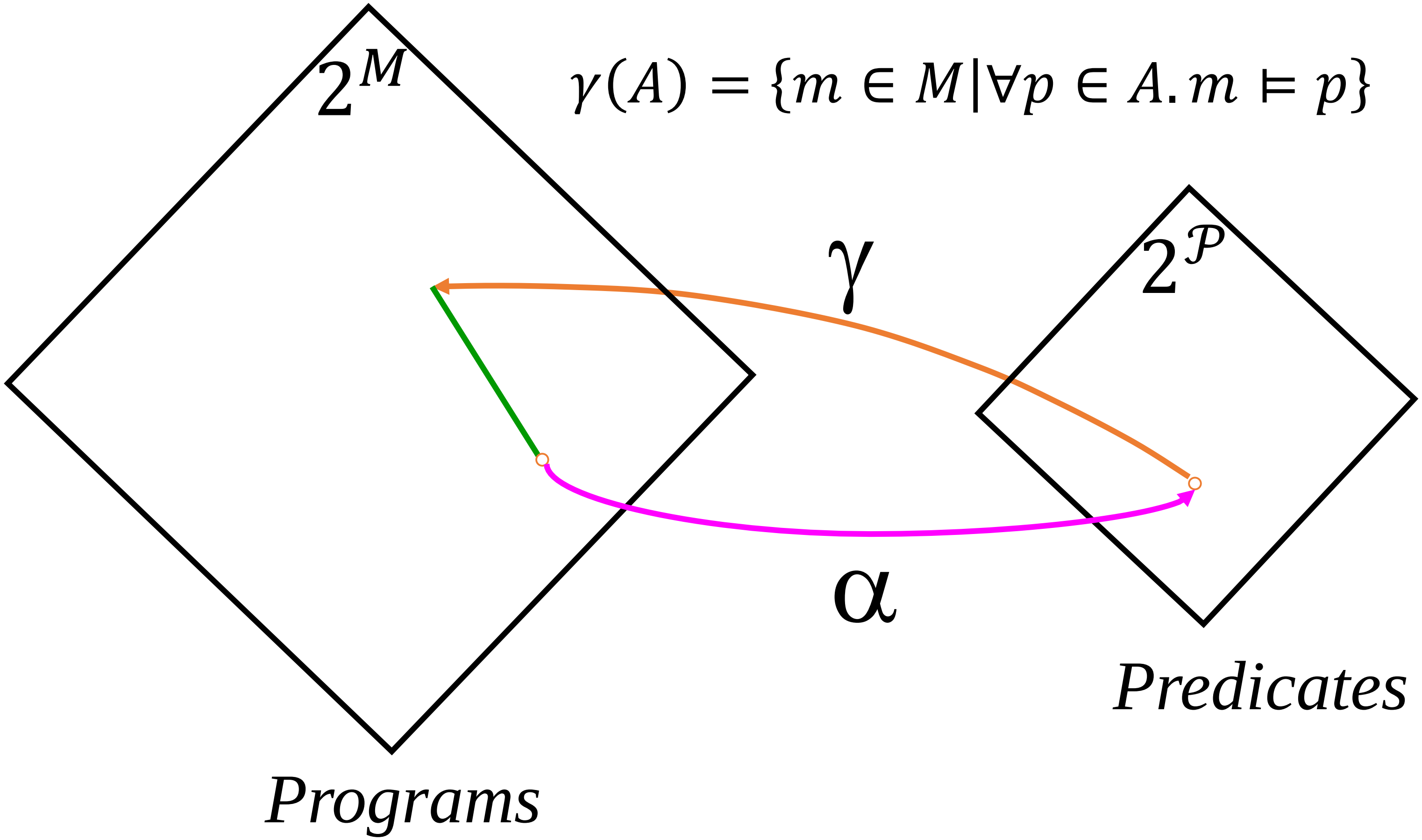
Select

- Candidate program is selected via *some* selection criterion: $Select$
- $Select$ usually designed to return a program from U^* ASAP (in 1-2 iterations)
- There is little theoretical work about *the long run*



An abstract domain

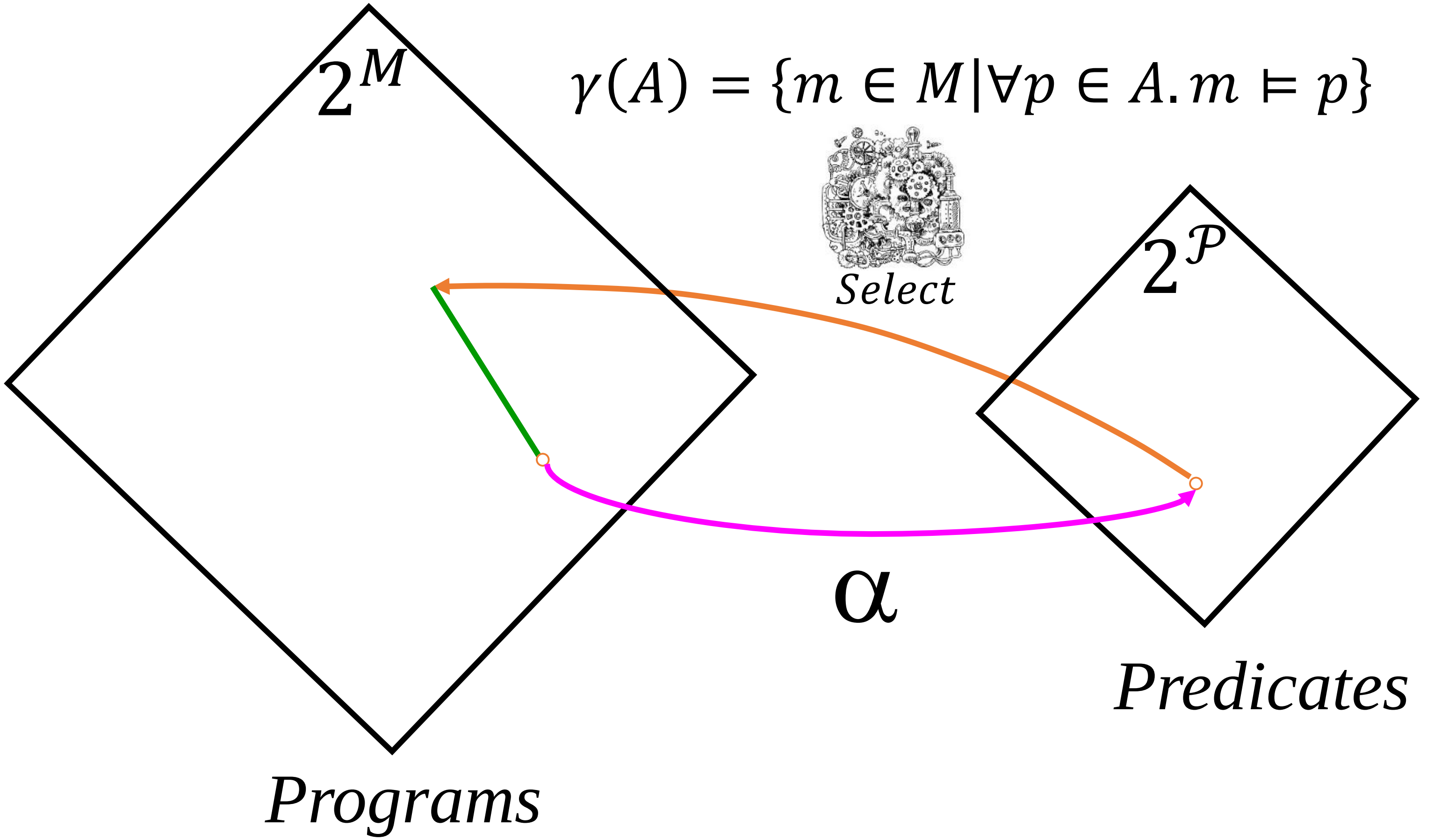
$$\gamma(A) = \{m \in M | \forall p \in A.\, m \vDash p\}$$



$$\alpha(C) = \{p \in \mathcal{P} | \forall m \in C.\, m \vDash p\}$$

An abstract domain

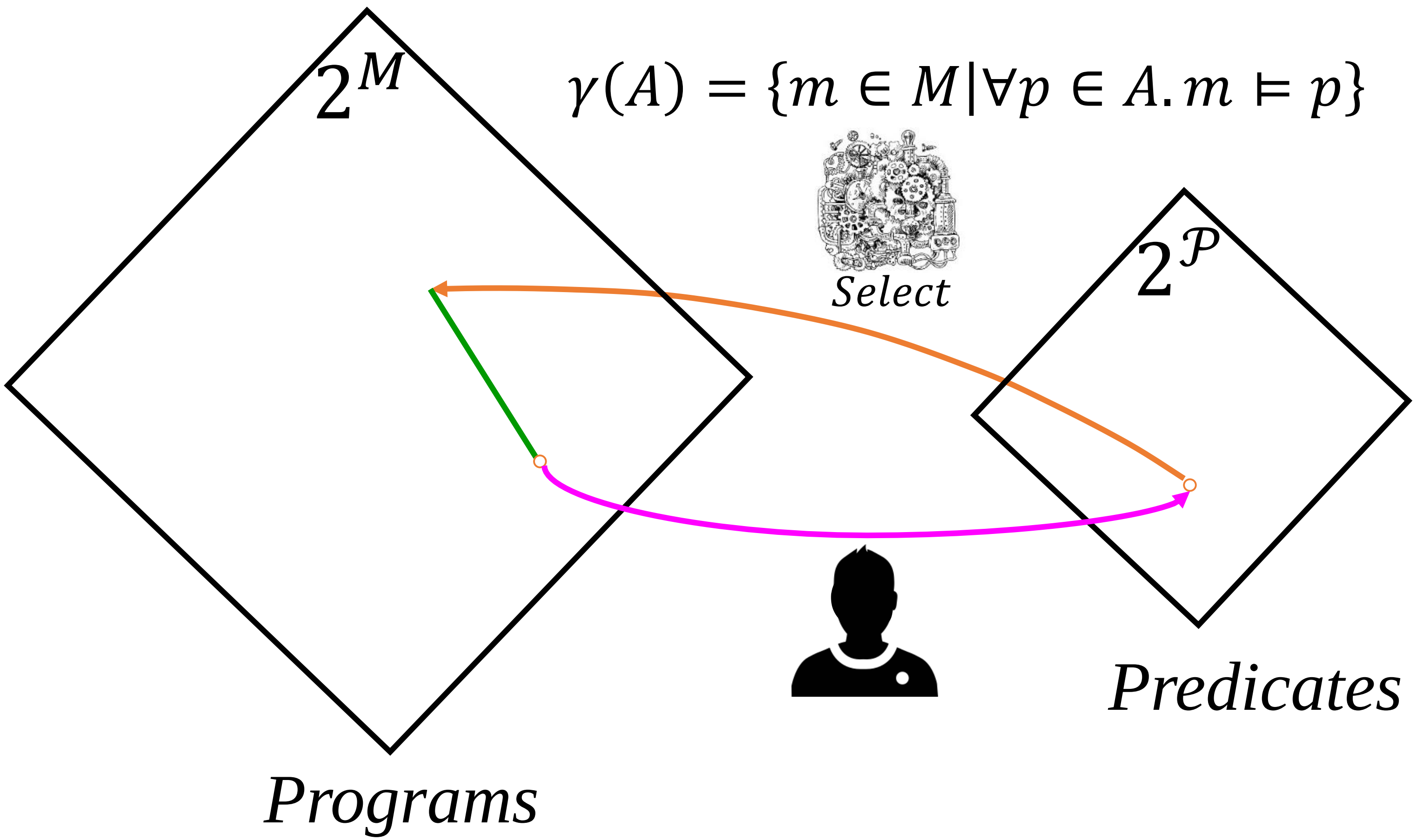
$$\gamma(A) = \{m \in M \mid \forall p \in A.\, m \vDash p\}$$



$$\alpha(C) = \{p \in \mathcal{P} \mid \forall m \in C.\, m \vDash p\}$$

An abstract domain

$$\gamma(A) = \{m \in M \mid \forall p \in A.\, m \vDash p\}$$



$$\alpha(C) = \{p \in \mathcal{P} \mid \forall m \in C.\, m \vDash p\}$$

Synthesis Session

- A synthesis session:

Candidate program

$$\mathcal{S} = (A_0, q_1)(A_1, q_2)\ \dots$$

Initial specifications

User answer

  - Synthesizer state: $S_i = S_{i-1} \sqcap A_i$
  - $q_i = Select(S_{i-1})$, or $q_i \in \gamma(S_{i-1}) \cup \{\perp\}$
  - If $\mathrm{q_i} \in U^* \cup \{\perp\}$, the session terminates

Synthesis user

- The user is aiming for some *ideal* set of programs $U^* \subseteq U$ (where U is the universe of all programs)
- The realizable target set is $M^* = M \cap U^*$
- **Correctness:** A user step is correct when

$$A_i \subseteq \{p \in \mathcal{P} \mid \exists m \in U^*. m \vDash p\}$$

Synthesis user

- The user is aiming for some *ideal* set of programs $U^* \subseteq U$ (where U is the universe of all programs)
- The realizable target set is $M^* = M \cap U^*$
- **Correctness:** A user step is correct when

$$A_i \subseteq \{p \in \mathcal{P} \mid \exists m \in U^*.\, m \vDash p\}$$

$\boldsymbol{U^*}$

```
(1 until n).fold((x,z)=>x+z)
```

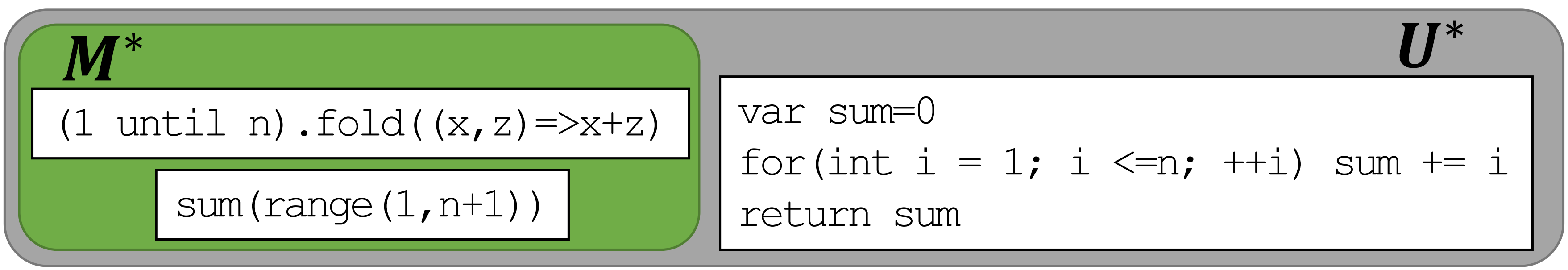
```
sum(range(1,n+1))
```

```
var sum=0
for(int i = 1; i <=n; ++i) sum += i
return sum
```

Synthesis user

- The user is aiming for some *ideal* set of programs $U^* \subseteq U$ (where U is the universe of all programs)
- The realizable target set is $M^* = M \cap U^*$
- **Correctness:** A user step is correct when

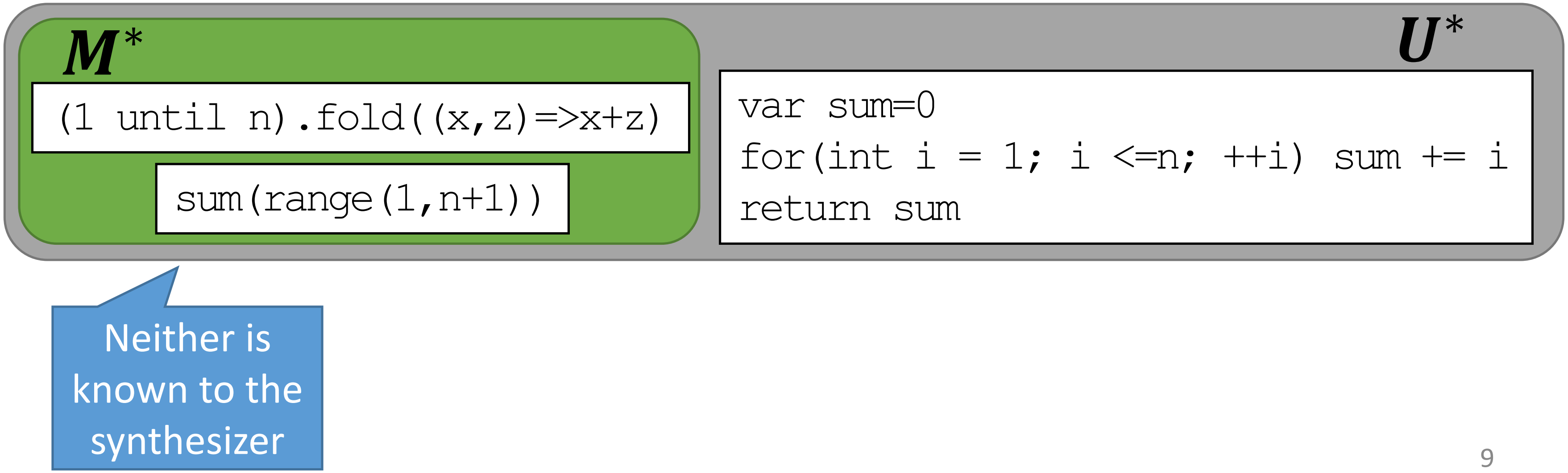
$$A_i \subseteq \{p \in \mathcal{P} \mid \exists m \in U^*. m \vDash p\}$$



Synthesis user

- The user is aiming for some *ideal* set of programs $U^* \subseteq U$ (where U is the universe of all programs)
- The realizable target set is $M^* = M \cap U^*$
- **Correctness:** A user step is correct when

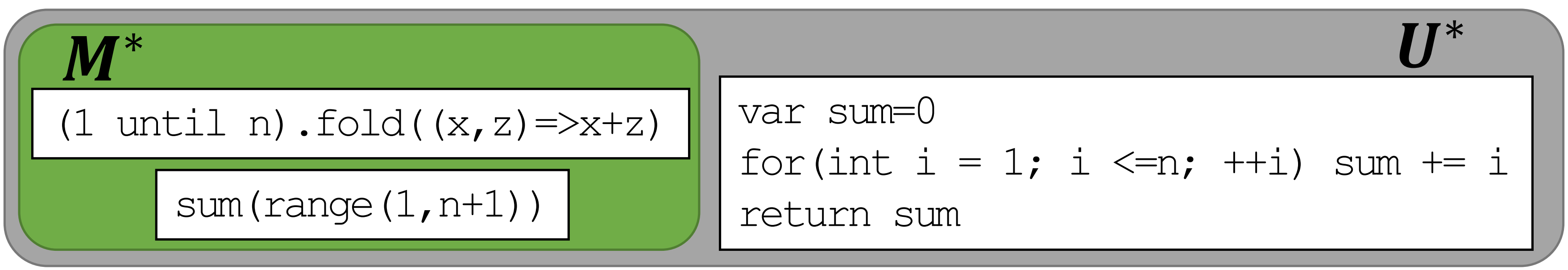
$$A_i \subseteq \{p \in \mathcal{P} \mid \exists m \in U^*. m \vDash p\}$$



Synthesis user

- The user is aiming for some *ideal* set of programs $U^* \subseteq U$ (where U is the universe of all programs)
- The realizable target set is $M^* = M \cap U^*$
- **Correctness:** A user step is correct when

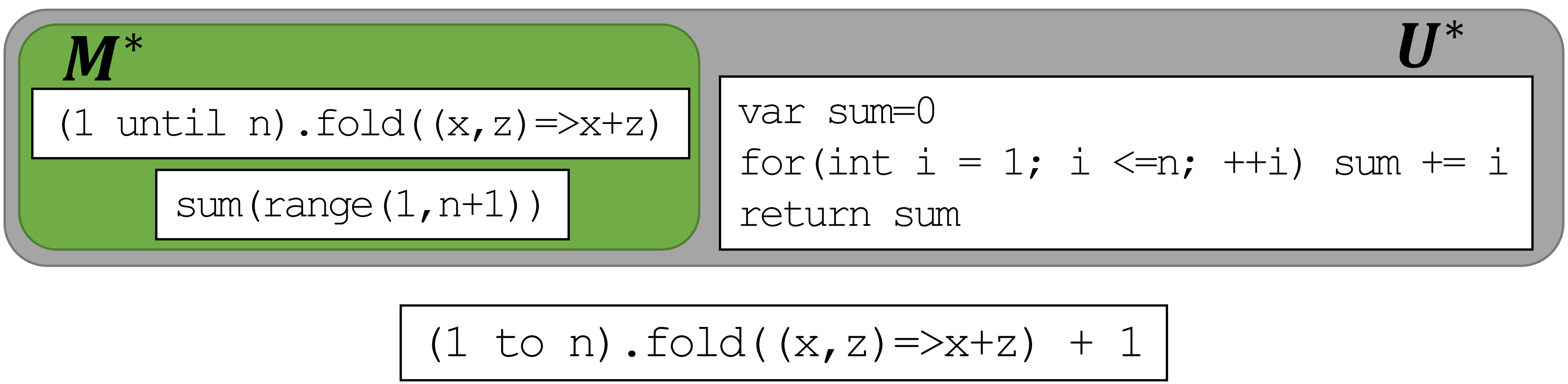
$$A_i \subseteq \{p \in \mathcal{P} \mid \exists m \in U^*. m \vDash p\}$$



Synthesis user

- The user is aiming for some *ideal* set of programs $U^* \subseteq U$ (where U is the universe of all programs)
- The realizable target set is $M^* = M \cap U^*$
- **Correctness:** A user step is correct when

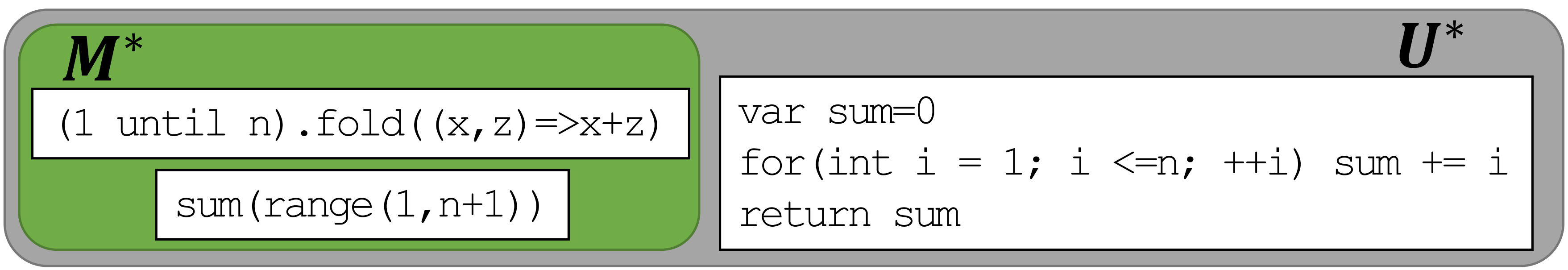
$$A_i \subseteq \{p \in \mathcal{P} \mid \exists m \in U^*. m \vDash p\}$$



Synthesis user

- The user is aiming for some *ideal* set of programs $U^* \subseteq U$ (where U is the universe of all programs)
- The realizable target set is $M^* = M \cap U^*$
- **Correctness:** A user step is correct when

$$A_i \subseteq \{p \in \mathcal{P} \mid \exists m \in U^*. m \vDash p\}$$




```
(1 to n).fold((x,z)=>x+z) + 1
```

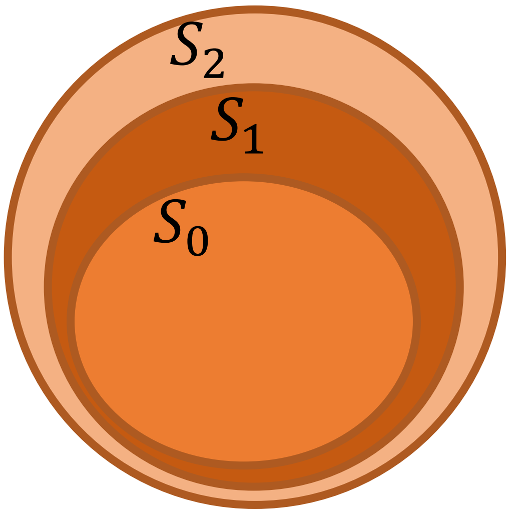
✓ ✗

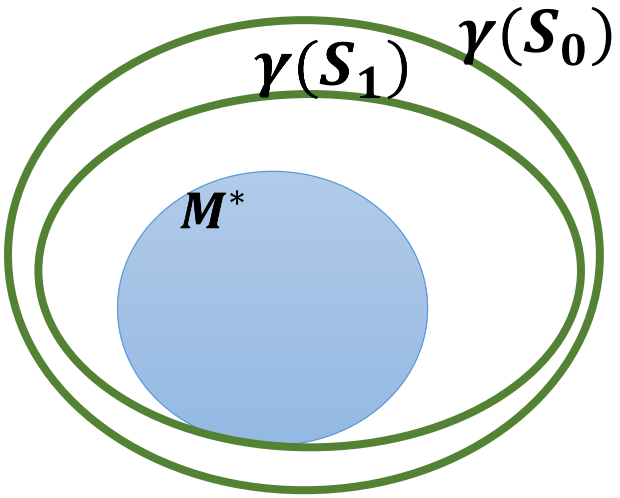
User guarantees

- In a synthesis session:
  1. The user is correct for as long as possible. When not possible, $A_i = \perp$
  2. The user will always accept a program from M^*

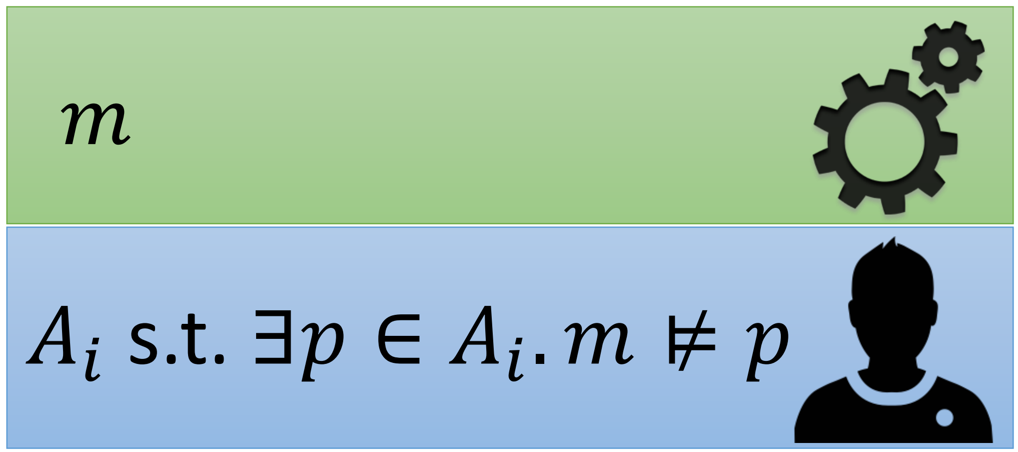
Progress

- Adding new predicate doesn't guarantee that the session is progressing
- $S_{n-1} \sqsubset S_n$ could still mean that $\gamma(S_n) = \gamma(S_{n-1})$
- Synthesizers usually don't check





Easiest way to make progress



Easiest way to make progress

m

A_i s.t. $\exists p \in A_i . m \nvDash p$

- In other words: something that's wrong with the current program
- Rule out *at least* the current program

Easiest way to make progress

m

A_i s.t. $\exists p \in A_i . m \nvDash p$

- In other words: something that's wrong with the current program
- Rule out *at least* the current program
- Important when $q_i \in \gamma(S_{i+1}) \Rightarrow Select(S_{i+1}) = q_i$
- Easy to check, but too strong

A Different Model of Progress

- A_i makes *weak progress* if
  $$\gamma(S_{n-1} \sqcap A_i) = \gamma(S_n) \subsetneq \gamma(S_{n-1})$$
- We can provide positive reinforcement

A Different Model of Progress

- A_i makes *weak progress* if
$$\gamma(S_{n-1} \sqcap A_i) = \gamma(S_n) \subsetneq \gamma(S_{n-1})$$
- We can provide positive reinforcement

```
(1 to n).fold((x,z)=>x+z) + 1
```

A Different Model of Progress

- A_i makes *weak progress* if
  $$\gamma(S_{n-1} \sqcap A_i) = \gamma(S_n) \subsetneq \gamma(S_{n-1})$$
- We can provide positive reinforcement

```
(1 to n).fold((x,z)=>x+z) + 1
```

✓

A Different Model of Progress

- A_i makes *weak progress* if

$$\gamma(S_{n-1} \sqcap A_i) = \gamma(S_n) \subsetneq \gamma(S_{n-1})$$

- We can provide positive reinforcement

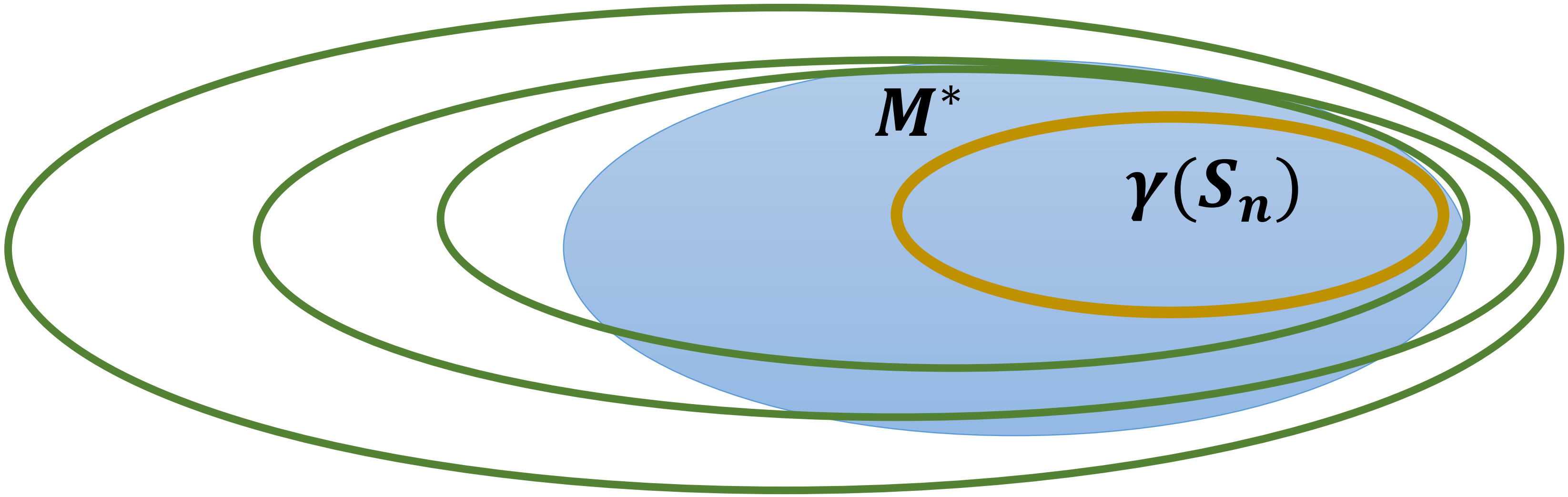
```
(1 to n).fold((x,z)=>x+z) + 1
```

✓

- Harder to check: $S_i \nRightarrow A_i$
- Once we know there is progress, we know some things about **termination** (see paper)

Convergence

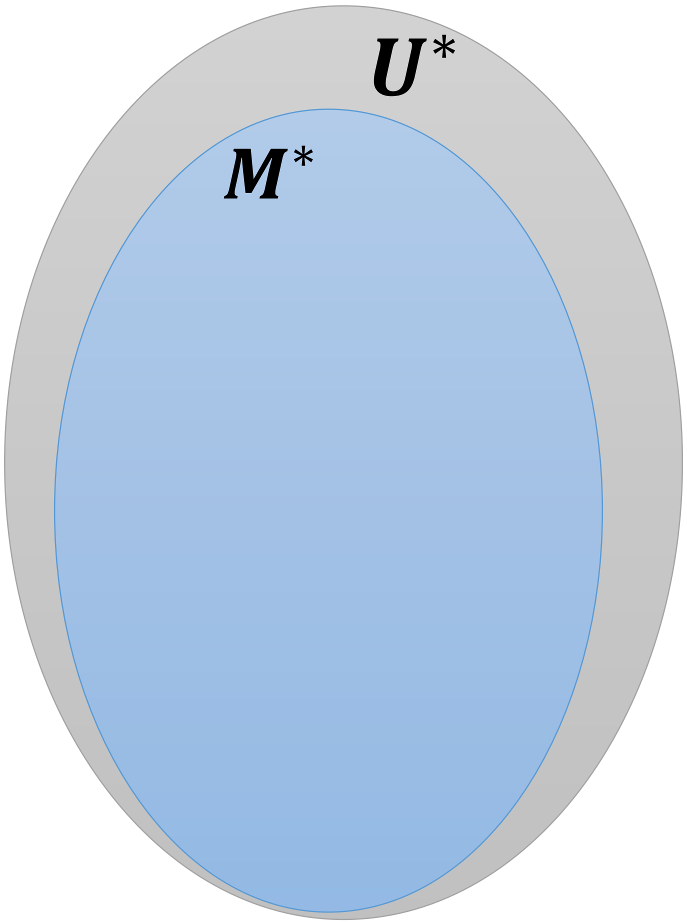
- A session **converges** if $\gamma(S_n) \subseteq M^*$
  - User correctness means the session ends at state n
- **Converges successfully**: $\emptyset \neq \gamma(S_n) \subseteq M^*$



Core set

- Core set: the set of **finite** underapproximations of M^*

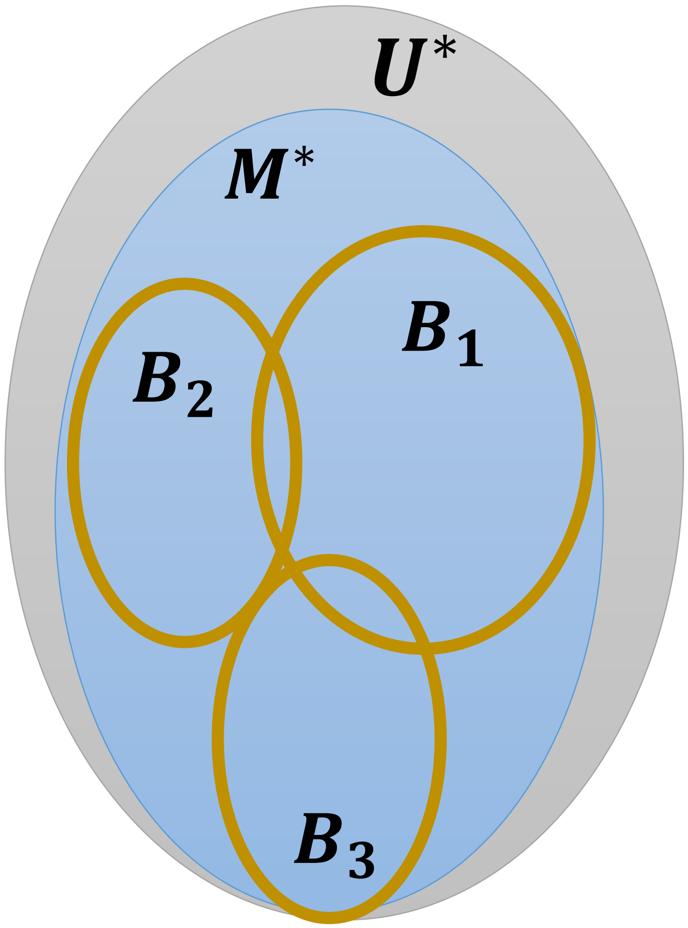
$$\mathcal{B} = \left\{ B \subseteq \mathcal{P} \,\middle|\, \begin{array}{c} 0 \neq \gamma(B) \subseteq M^* \\ \wedge\, |B| \in \mathbb{N} \end{array} \right\}$$



Core set

- Core set: the set of **finite** underapproximations of M^*

$$\mathcal{B} = \left\{ B \subseteq \mathcal{P} \middle| \begin{array}{c} 0 \neq \gamma(B) \subseteq M^* \\ \wedge\, |B| \in \mathbb{N} \end{array} \right\}$$

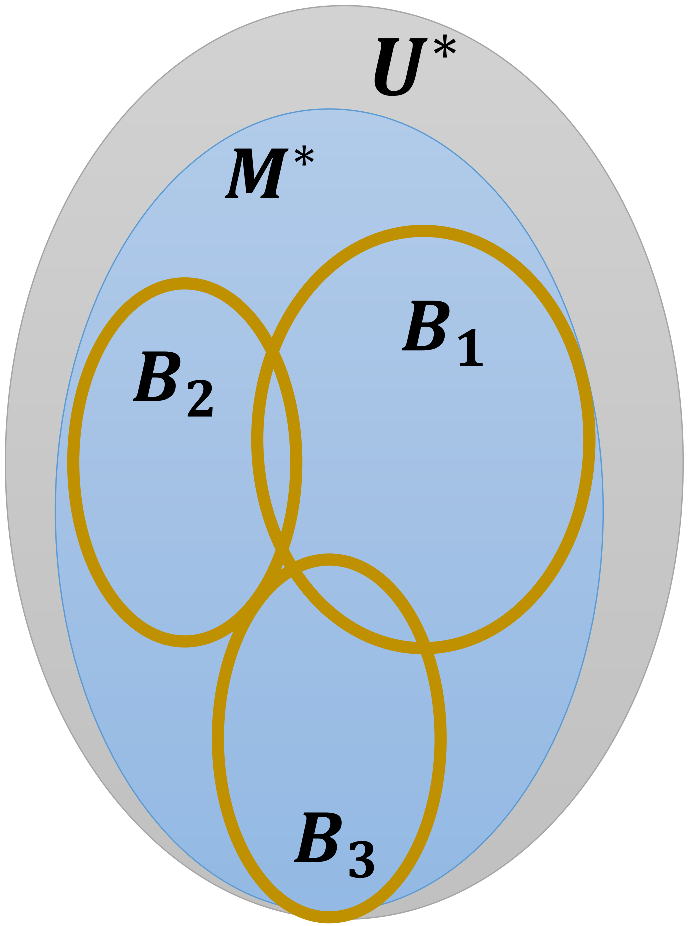


Core set

- Core set: the set of **finite** underapproximations of M^*

$$\mathcal{B} = \left\{ B \subseteq \mathcal{P} \middle| \begin{array}{c} 0 \neq \gamma(B) \subseteq M^* \\ \wedge \, |B| \in \mathbb{N} \end{array} \right\}$$

- User intention is realizable if $M^* \neq \emptyset$
- $\boldsymbol{\mathcal{P}}$**-realizability**: can converge under $\mathcal{P}$

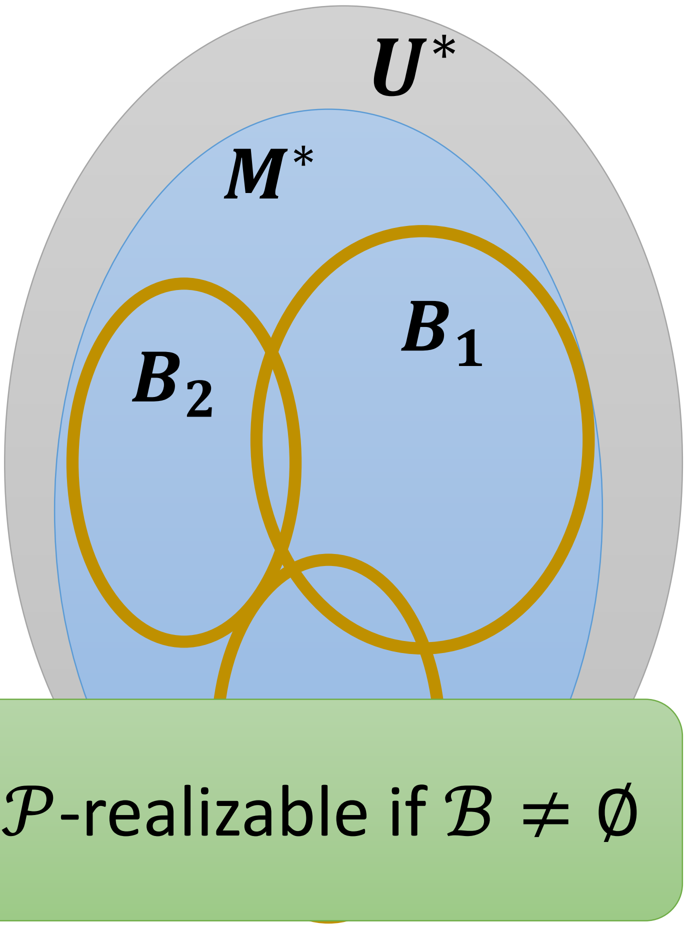


Core set

- Core set: the set of **finite** underapproximations of M^*

$$\mathcal{B} = \left\{ B \subseteq \mathcal{P} \;\middle|\; \begin{array}{c} 0 \neq \gamma(B) \subseteq M^* \\ \wedge \, |B| \in \mathbb{N} \end{array} \right\}$$

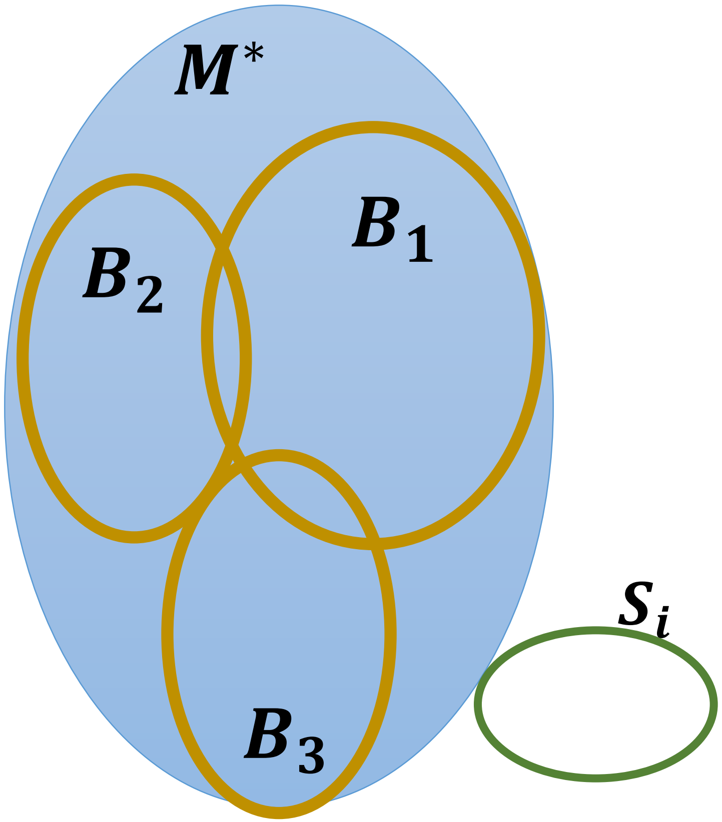
- User intention is realizable if $M^* \neq \emptyset$
- **$\boldsymbol{\mathcal{P}}$-realizability**: can converge under $\mathcal{P}$



M^* is $\mathcal{P}$-realizable if $\mathcal{B} \neq \emptyset$

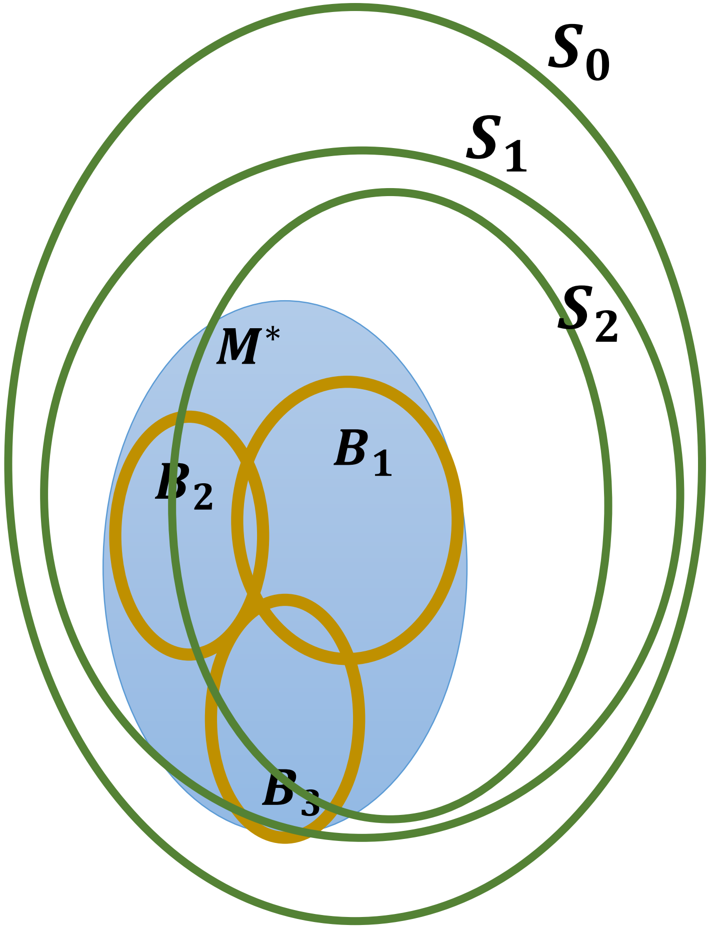
Infeasible point

- A state where
$$\gamma(S_i) \cap M^* = \emptyset$$
- $Select$ can't succeed, even in best case
- The **First Infeasible Point** is the first point of **failure**
- Can we **backtrack** before an infeasible point?



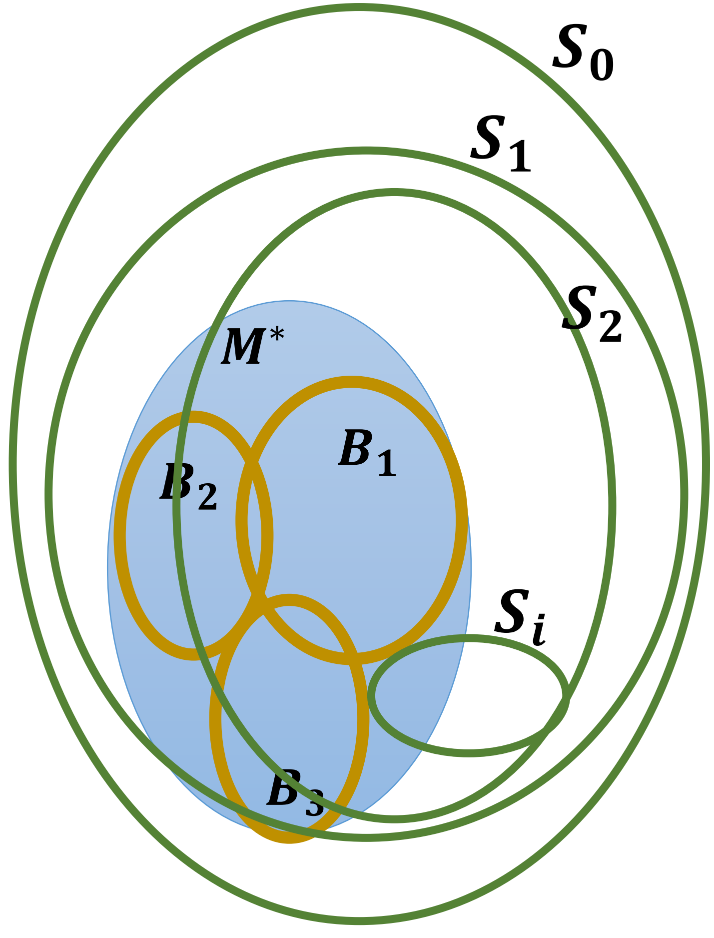
Point of inevitable failure

- State S_i is an POIF if $\forall B \in \mathcal{B}. \gamma(S_i) \cap \gamma(B) = \emptyset$
- Specifically, S_i is an POIF if $\gamma(S_i) \cap M^* = \emptyset$



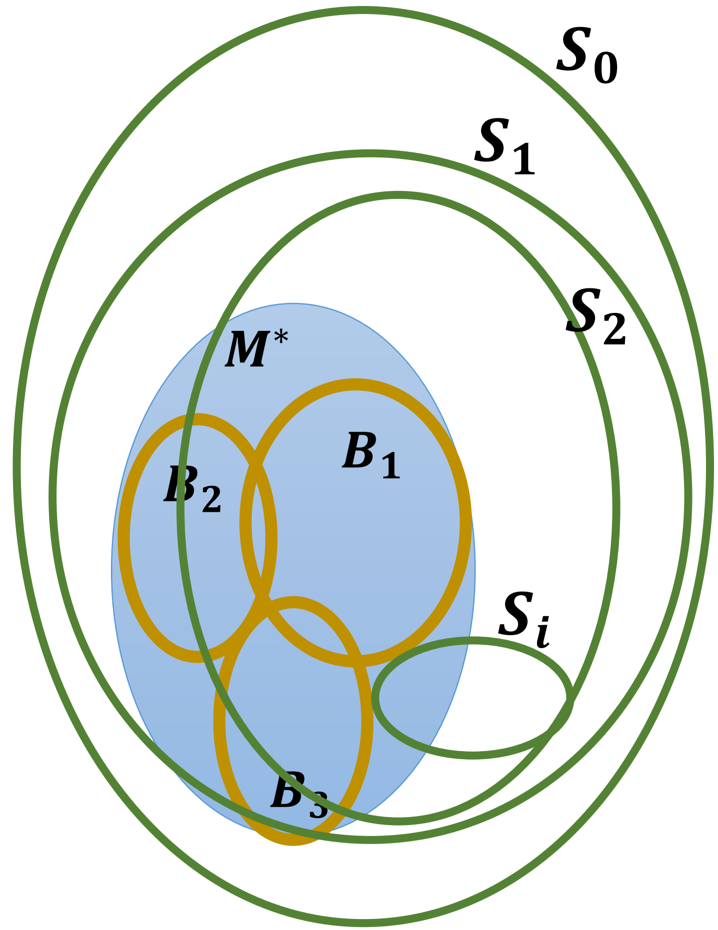
Point of inevitable failure

- State S_i is an POIF if $\forall B \in \mathcal{B}. \gamma(S_i) \cap \gamma(B) = \emptyset$
- Specifically, S_i is an POIF if $\gamma(S_i) \cap M^* = \emptyset$
- But not necessarily



Point of inevitable failure

- State S_i is an POIF if $\forall B \in \mathcal{B}. \gamma(S_i) \cap \gamma(B) = \emptyset$
- Specifically, S_i is an POIF if $\gamma(S_i) \cap M^* = \emptyset$
- But not necessarily



- As long as S_i is not an POIF we can still converge
- Can we **backtrack** before an inevitable point of failure?

Backtracking from failure

Theorem: for any $k \in \mathbb{N}$ there exists a session $\mathcal{S}$ where state S_i is an point of inevitable failure and only state S_{k+i} is the first infeasible point.

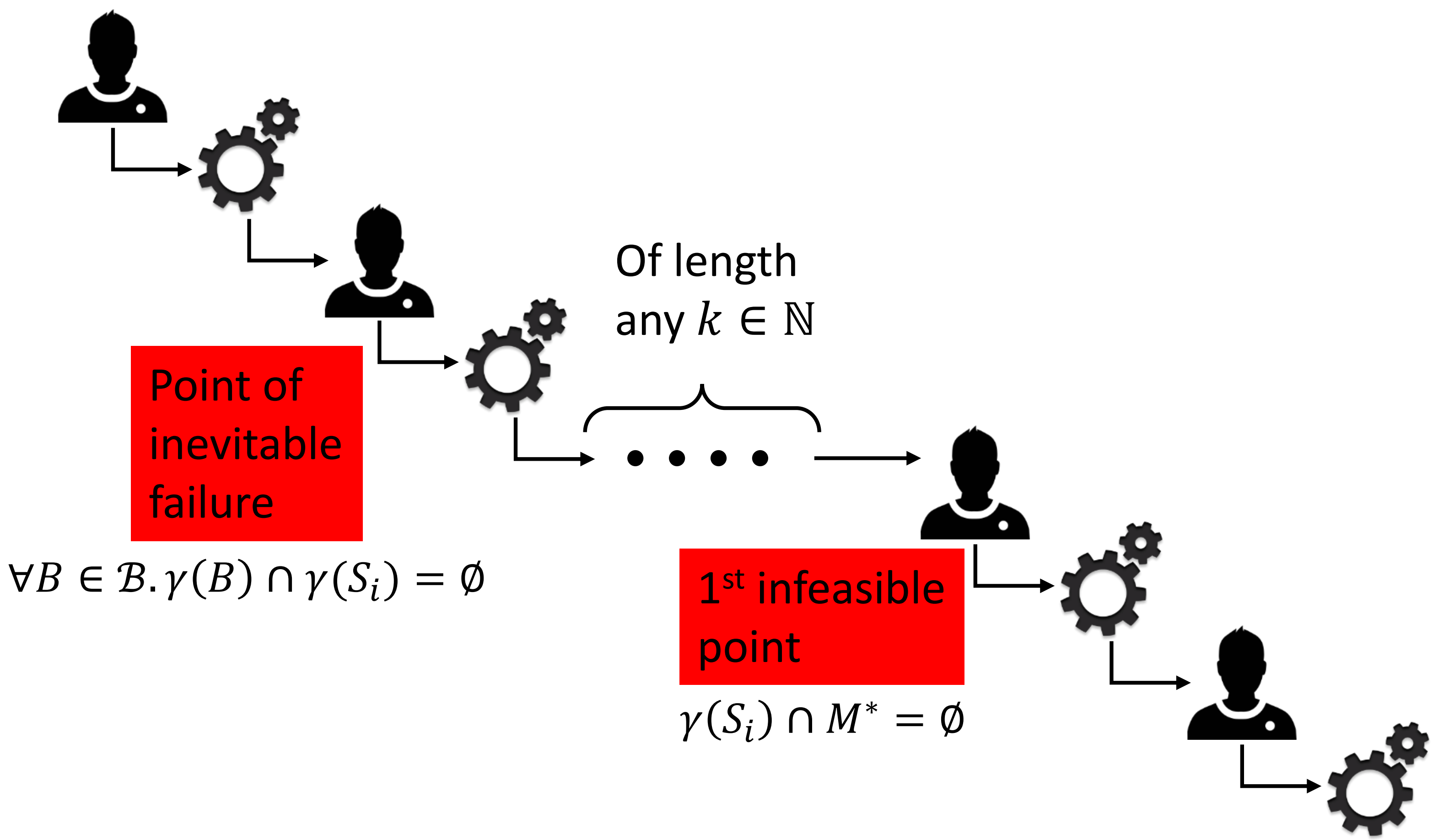
Backtracking from failure

Theorem: for any $k \in \mathbb{N}$ there exists a session $\mathcal{S}$ where state S_i is an point of inevitable failure and only state S_{k+i} is the first infeasible point.

Essentially: there is no bound on the number of steps to backtrack once failure is apparent.

Proof: by construction

An unbounded session



Construction

- Candidate program space M is spanned by:
  - if-else
  - ==
  - lists of ints ([],[1,2], etc.)
  - recursive call `f`
  - the input variable `i`
  - cons
  - max
  - remove
  - sort
  - reverse

Construction

- Candidate program space M is spanned by:
  - if-else
  - ==
  - lists of ints ([],[1,2], etc.)
  - recursive call `f`
  - the input variable `i`
  - cons
  - max
  - remove
  - sort
  - reverse

Example candidate:

```
if (i==[]) []
else cons(max(i),
  f(remove(i,max(i)))
```

Construction

- Candidate program space M is spanned by:
  - if-else
  - ==
  - lists of ints ([],[1,2], etc.)
  - recursive call `f`
  - the input variable `i`
  - cons
  - max
  - remove
  - sort
  - reverse

Example candidate:

```
if (i==[]) []
else cons(max(i),
   f(remove(i,max(i)))
```

Example candidate:

```
reverse(sort(i))
```

Construction

- Candidate program space M is spanned by:
  - if-else
  - ==
  - lists of ints ([],[1,2], etc.)
  - recursive call `f`
  - the input variable `i`
  - cons
  - max
  - remove
  - sort
  - reverse

- Available predicates in $\mathcal{P}$:
  - Input-output examples
  - $exclude(e)$, for any program element e

Example candidate:

```
if (i==[]) []
else cons(max(i),
  f(remove(i,max(i)))
```

Example candidate:

```
reverse(sort(i))
```

Construction

Task: sort a list in descending order.

Construction

Task: sort a list in descending order.

Input: `[]`, Output: `[]`

Input: `[1,2]`, Output: `[2,1]`

Construction

Task: sort a list in descending order.

Input: `[]`, Output: `[]`
Input: `[1,2]`, Output: `[2,1]`

```
reverse(i)
```

Construction

Task: sort a list in descending order.

Input: `[]`, Output: `[]`

Input: `[1,2]`, Output: `[2,1]`

`reverse(i)`

exclude(`reverse`)

Construction

Task: sort a list in descending order.

Input: `[]`, Output: `[]`

Input: `[1,2]`, Output: `[2,1]`

`reverse(i)`

exclude(`reverse`)

Point of Inevitable Failure

Construction

Task: sort a list in descending order.

Input: `[]`, Output: `[]`

Input: `[1,2]`, Output: `[2,1]`

```
reverse(i)
```

exclude(`reverse`)

Point of Inevitable Failure

```
if (i==[1,2])[2,1]
else i
```

Construction

```
if (i==[1,2])[2,1]
else i
```

Construction

```
if (i==[1,2])[2,1]
else i
```

Input: [1,3], **Output:** [3,1]

Construction

```
if (i==[1,2])[2,1]
else i
```

Input: `[1,3],` **Output:** `[3,1]`

```
if (i==[1,2]) [2,1]
else if (i==[1,3]) [3,1]
else i
```

Construction

```
if (i==[1,2])[2,1]
else i
```

Input: `[1,3],` **Output:** `[3,1]`

```
if (i==[1,2]) [2,1]
else if (i==[1,3]) [3,1]
else i
```

⋮

Construction

```
if (i==[1,2])[2,1]
else i
```

Input: `[1,3]`, **Output:** `[3,1]`

```
if (i==[1,2]) [2,1]
else if (i==[1,3]) [3,1]
else i
```

⋮

Example candidate:

```
if (i==[]) []
else cons(max(i),
   f(remove(i,max(i))
```

Construction

```
if (i==[1,2])[2,1]
else i
```

Input: `[1,3],` **Output:** `[3,1]`

```
if (i==[1,2]) [2,1]
else if (i==[1,3]) [3,1]
else i
```

⋮

Construction

```
if (i==[1,2])[2,1]
else i
```

Input: `[1,3]`, **Output:** `[3,1]`

```
if (i==[1,2]) [2,1]
else if (i==[1,3]) [3,1]
else i
```

⋮

exclude(==)

Construction

```
if (i==[1,2])[2,1]
else i
```

Input: `[1,3]`, **Output:** `[3,1]`

```
if (i==[1,2]) [2,1]
else if (i==[1,3]) [3,1]
else i
```

⋮

exclude(==)

$\perp$

Conclusion

- An iterative, interactive model of synthesis
- An abstract domain of predicates
- Progress
- Convergence
- The unboundedness of backtracking
- We hope these results help future synthesizer designers