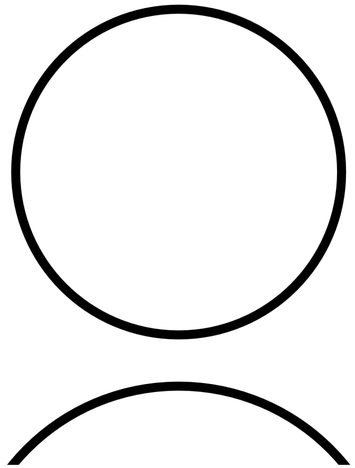
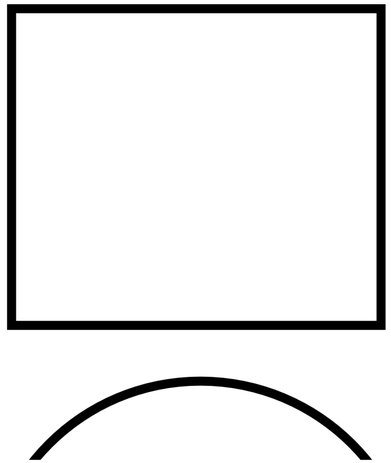
Programming by Navigation

Justin Lubin, Parker Ziegler, Sarah E. Chasins

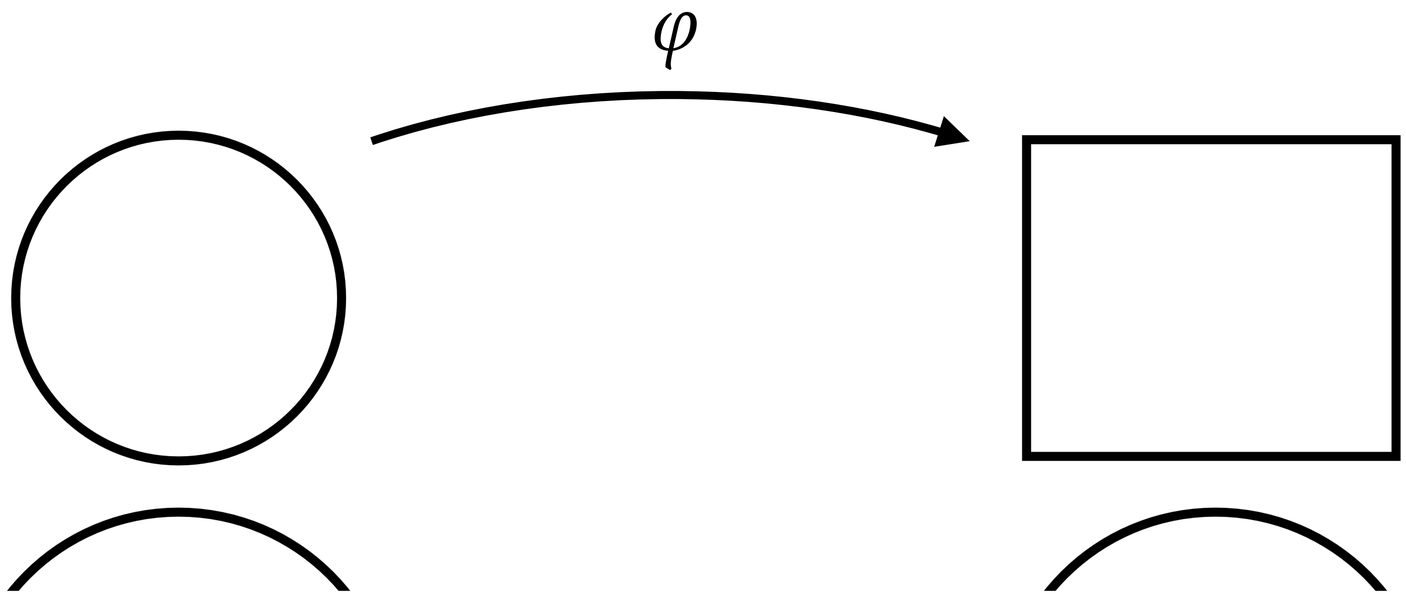
PLDI 2025

In traditional program synthesis, guarantees operate on one round of interaction.

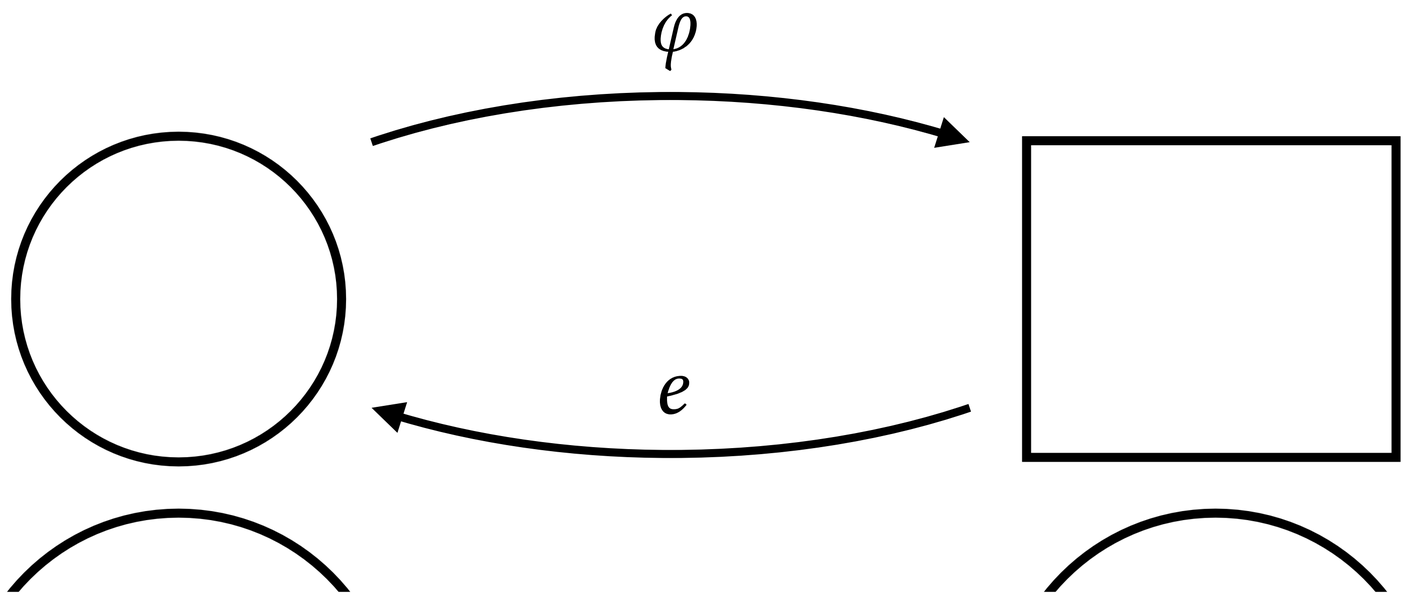




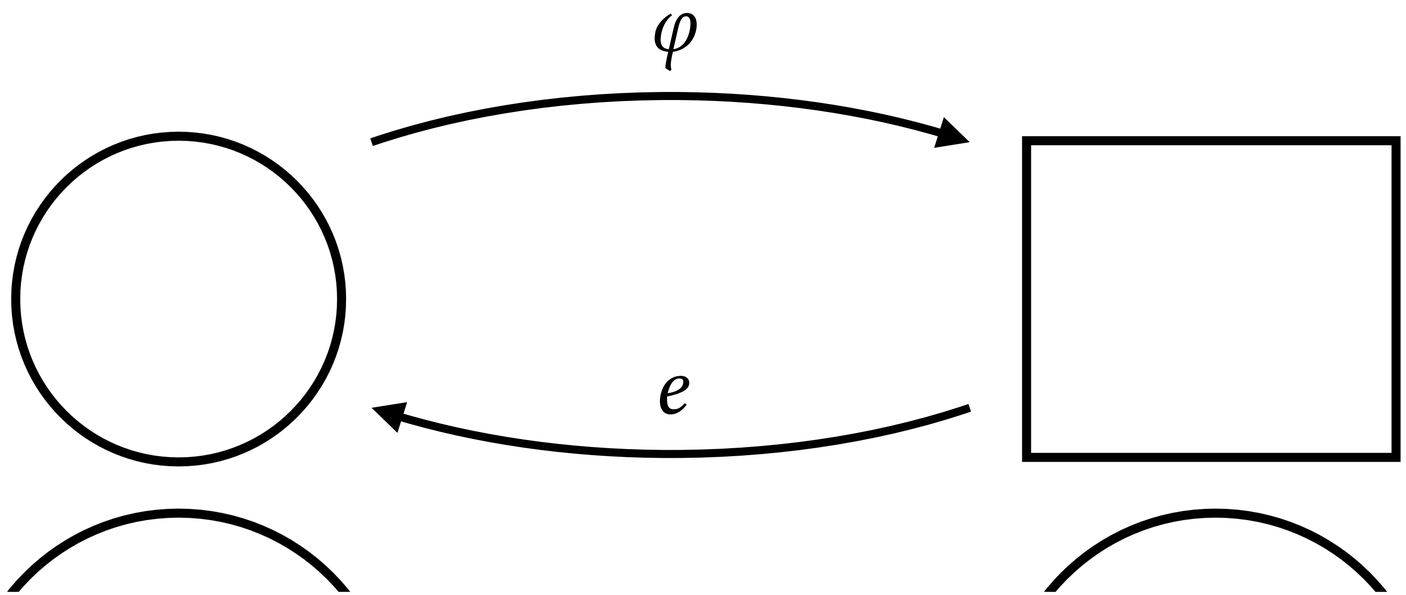
In traditional program synthesis, guarantees operate on one round of interaction.



In traditional program synthesis, guarantees operate on one round of interaction.

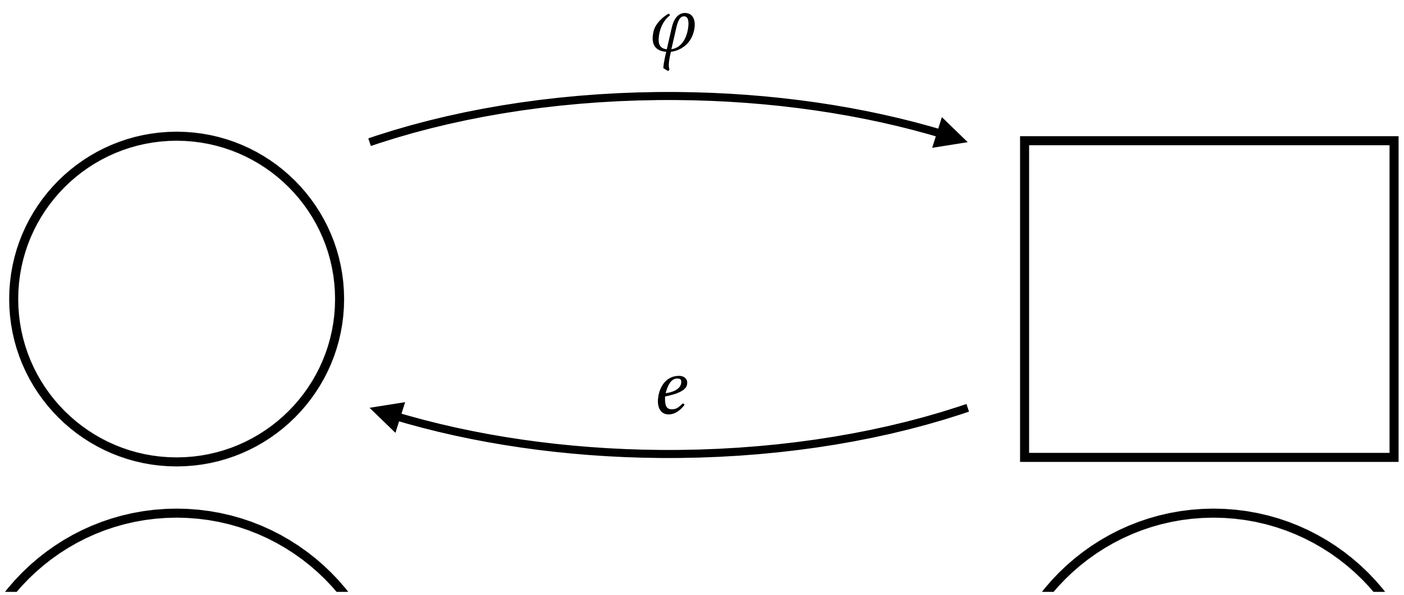


In traditional program synthesis, guarantees operate on one round of interaction.



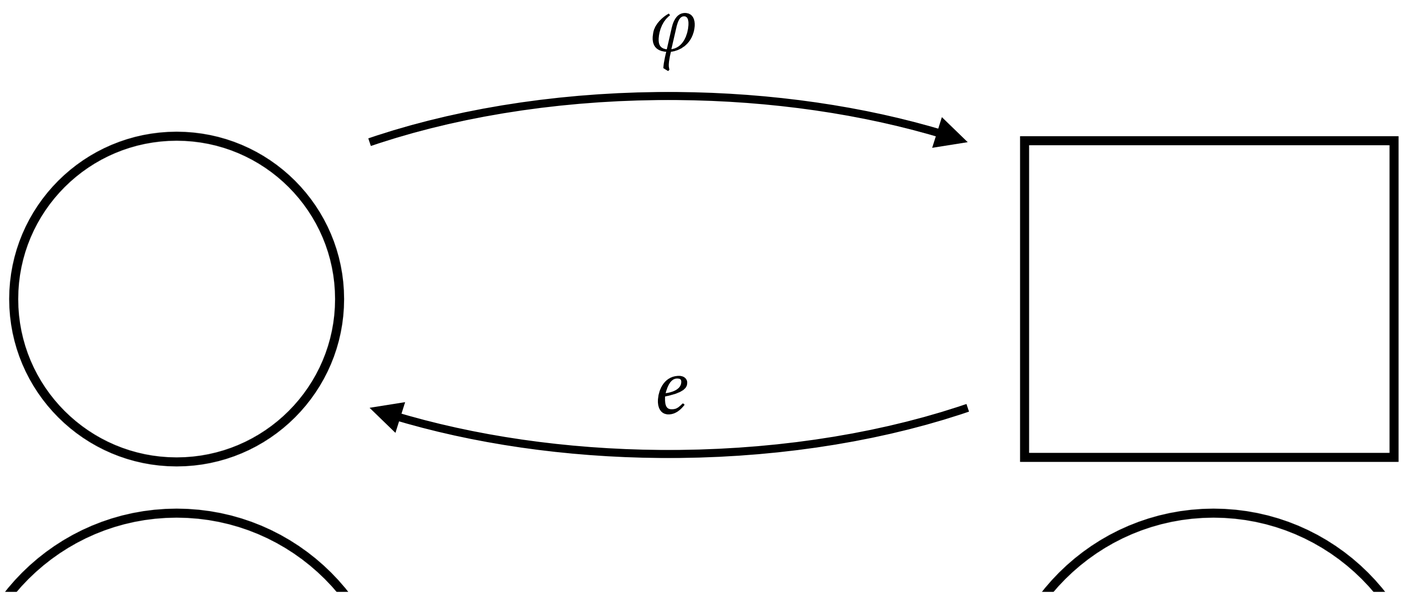
- ***Soundness.*** *Any returned program e satisfies the specification φ.*

In traditional program synthesis, guarantees operate on one round of interaction.



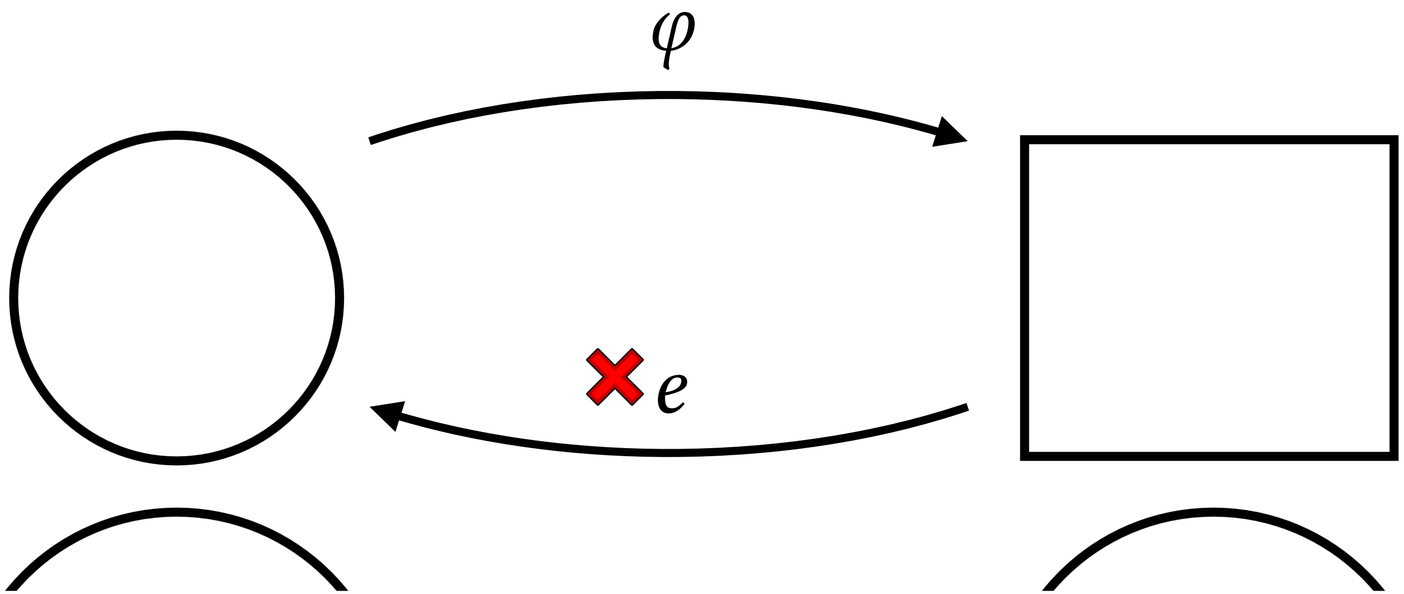
- ***Soundness.*** *Any returned program e satisfies the specification φ.*
- ***Completeness.*** *A program e is returned when at least one satisfying solution to the specification φ exists.*

In traditional program synthesis, guarantees operate on one round of interaction.



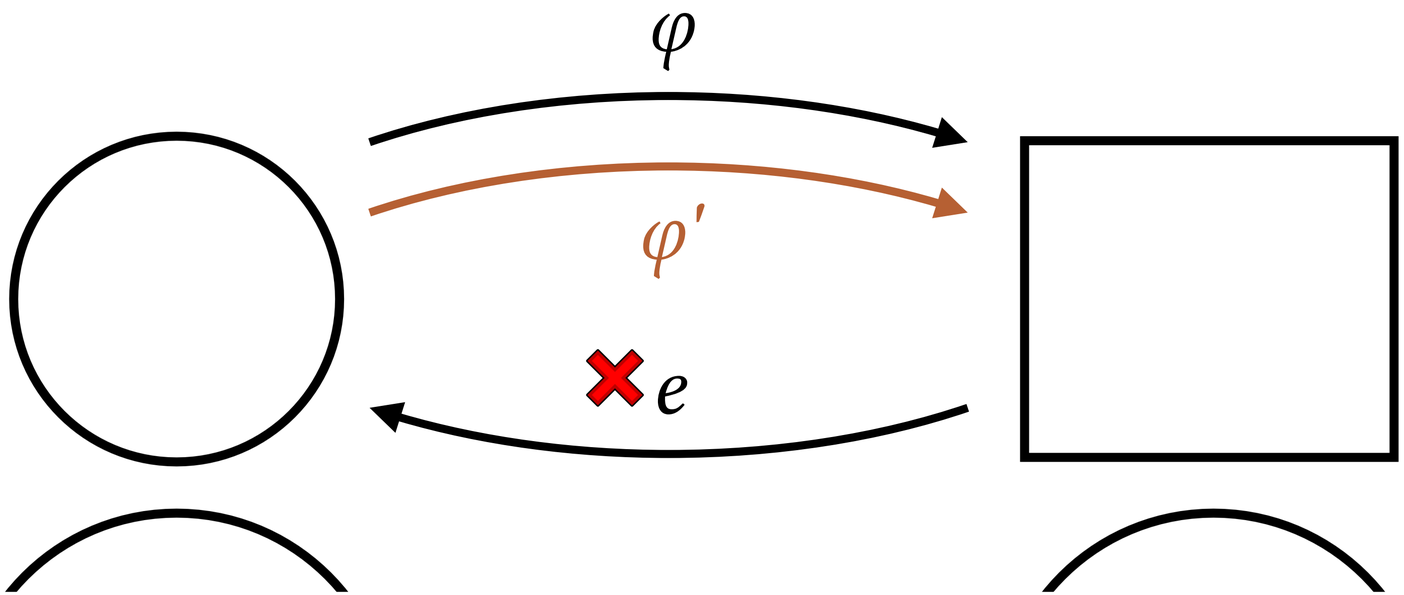
- ***Soundness.*** *Any returned program e satisfies the specification φ.*
- ***Completeness.*** *A program e is returned when at least one satisfying solution to the specification φ exists.*
- What if we want to refine φ?

In traditional program synthesis, guarantees operate on one round of interaction.



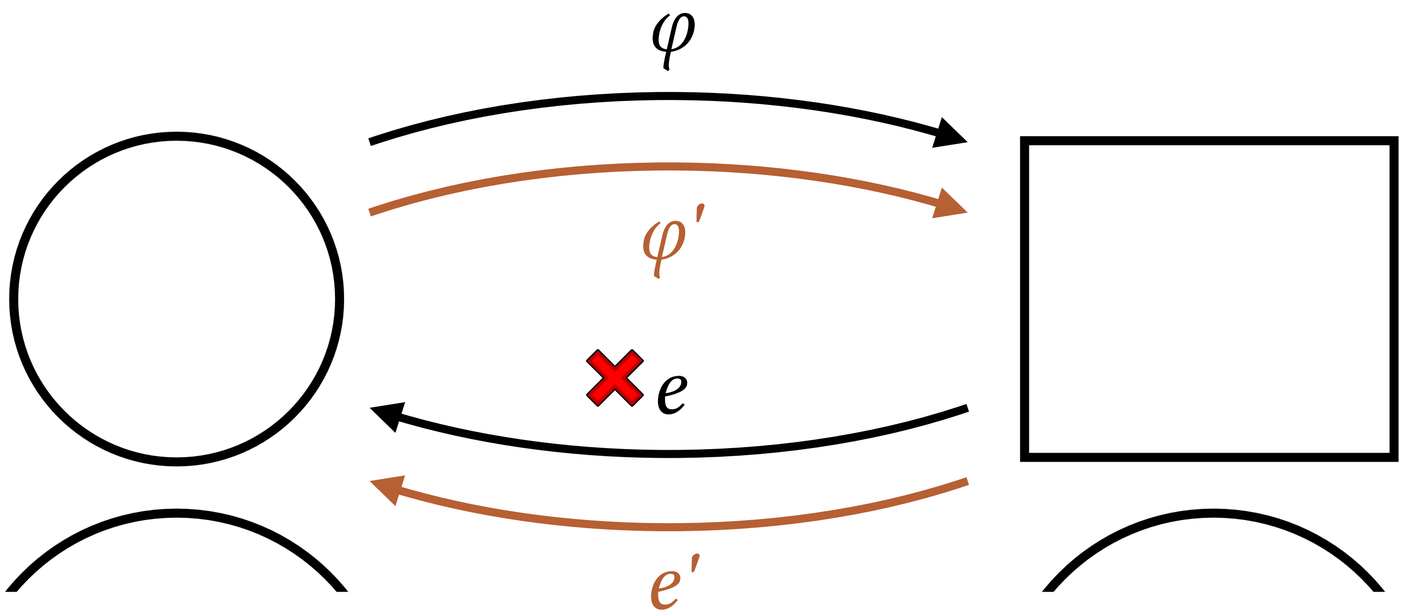
- ***Soundness.*** *Any returned program e satisfies the specification φ.*
- ***Completeness.*** *A program e is returned when at least one satisfying solution to the specification φ exists.*
- What if we want to refine φ?

In traditional program synthesis, guarantees operate on one round of interaction.



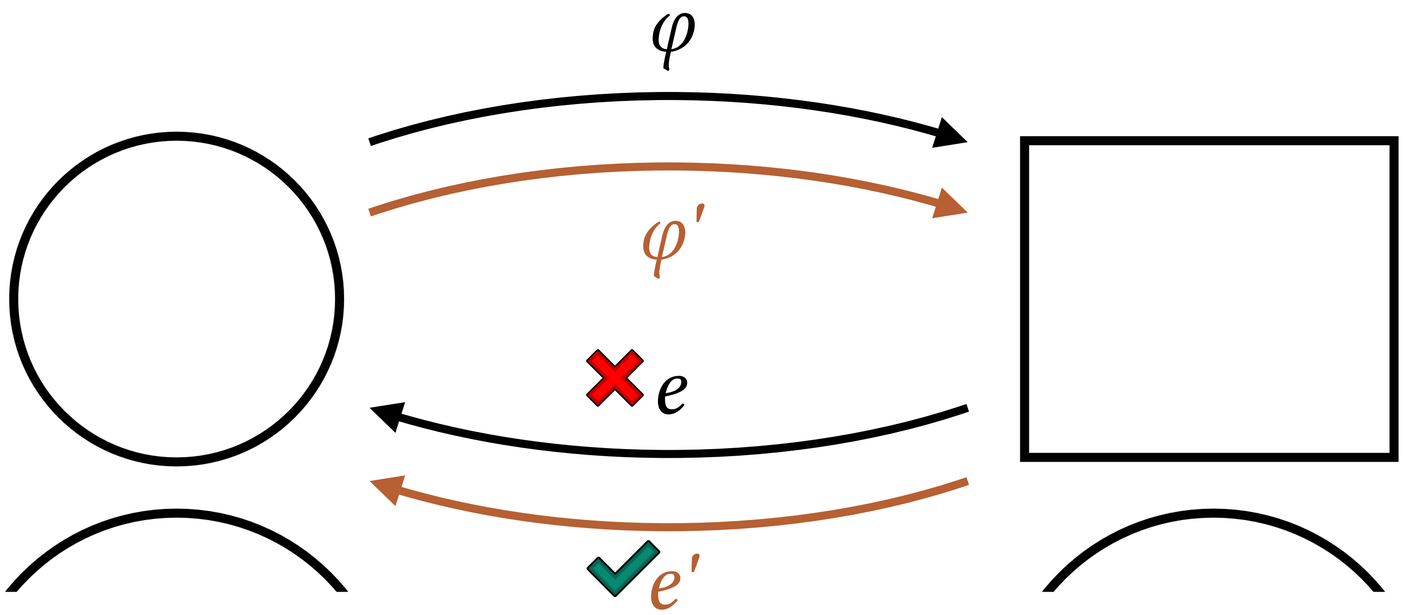
- ***Soundness.*** *Any returned program e satisfies the specification φ.*
- ***Completeness.*** *A program e is returned when at least one satisfying solution to the specification φ exists.*
- What if we want to refine φ?

In traditional program synthesis, guarantees operate on one round of interaction.



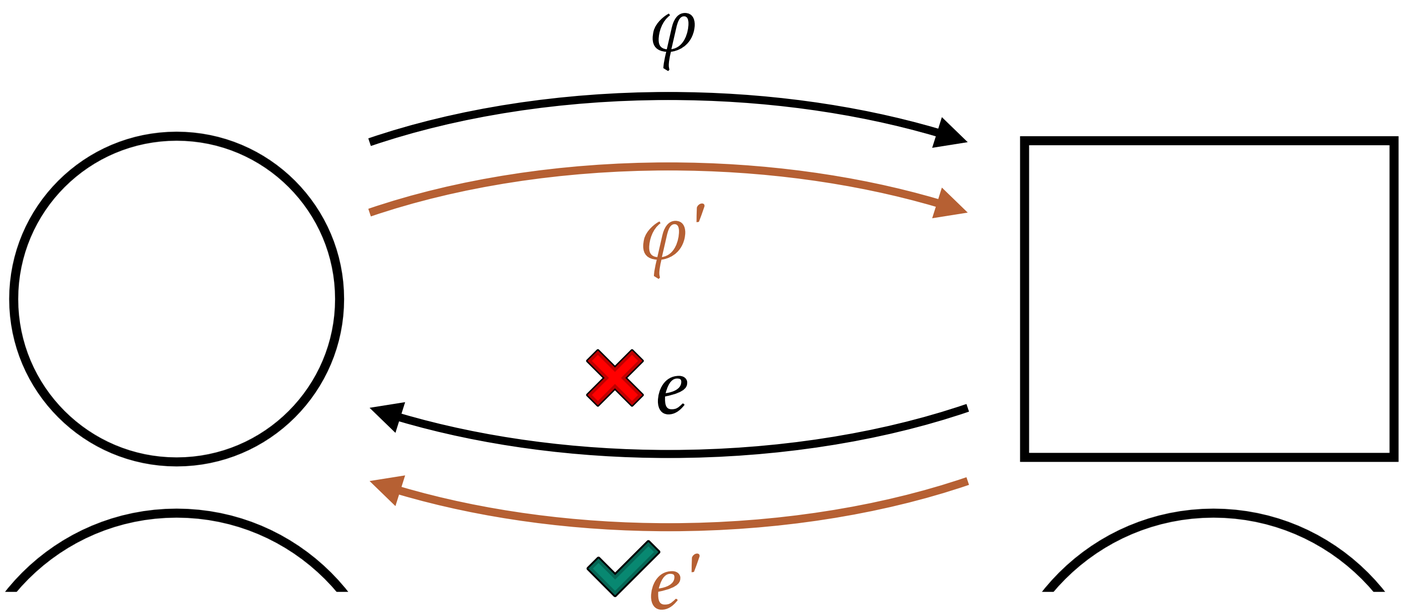
- ***Soundness.*** *Any returned program e satisfies the specification φ.*
- ***Completeness.*** *A program e is returned when at least one satisfying solution to the specification φ exists.*
- What if we want to refine φ?

In traditional program synthesis, guarantees operate on one round of interaction.



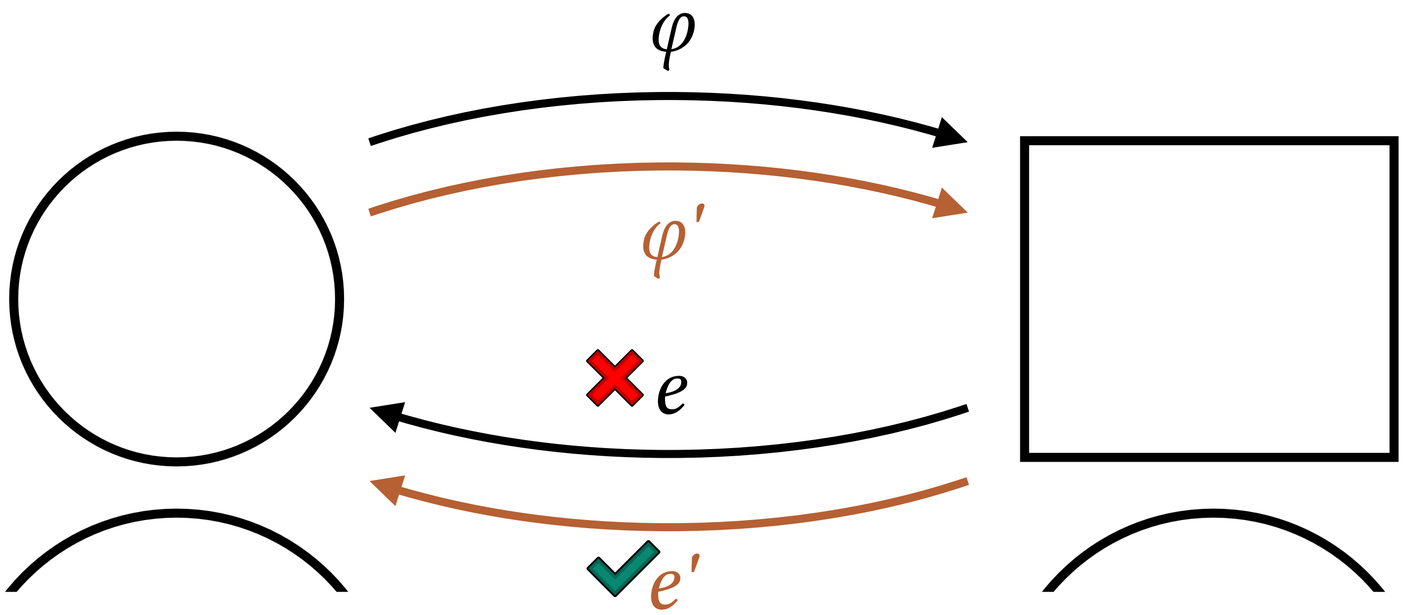
- ***Soundness.*** *Any returned program e satisfies the specification φ.*
- ***Completeness.*** *A program e is returned when at least one satisfying solution to the specification φ exists.*
- What if we want to refine φ?

In traditional program synthesis, guarantees operate on one round of interaction.



- ***Soundness.*** *Any returned program e satisfies the specification φ.*
- ***Completeness.*** *A program e is returned when at least one satisfying solution to the specification φ exists.*
- What if we want to refine φ?
- What guarantees can we get on the interaction as a whole?

In traditional program synthesis, guarantees operate on one round of interaction.



- ***Soundness.*** *Any returned program e satisfies the specification φ.*
- ***Completeness.*** *A program e is returned when at least one satisfying solution to the specification φ exists.*
- What if we want to refine φ?
- What guarantees can we get on the interaction as a whole?
- ***Convergence.*** *User will accept synthesis output in a finite number of rounds.*

Existing interactive synthesis guarantees require well-behaved users.

Existing interactive synthesis guarantees require well-behaved users.

Abstraction-Based Interaction Model for Synthesis

Hila Peleg[1], Shachar Itzhaky[1], and Sharon Shoham[2]

[1] Technion, {hilap,shachari}@cs.technion.ac.il
[2] Tel Aviv University, sharon.shoham@gmail.com

Definition 5 (User correctness). *A user step, providing A_i as an additional specification, is* correct *when* $A_i \subseteq \{p \in \mathcal{P} \mid \exists m \in U^*.m \vDash p\}$.

Definition 6 (Synthesis user). *The* behavior of the user *includes the following guarantees:*

1. *The user is correct for as long as they can be. If the user can no longer provide an answer that is correct, they will answer* $\perp$.
2. *If a user sees a program in* M^**, they will accept it.*

Definition 7 (Feasible synthesis session). *A* feasible synthesis session *is a synthesis session* $\mathcal{S} = (A_0, q1), (A_1, q_2), \ldots$ *that satisfies the following:*

(a) All A_i *are correct steps (definition 5) or* $\perp$,
(b) $q_i = Select(S_{i-1})$*, i.e.* $q_i \in \gamma(S_{i-1}) \cup \{\perp\}$*, where* $\perp$ *signifies no possible program,*
(c) If $q_n \in M^* \cup \{\perp\}$ *then* $\mathcal{S}$ *is finite and of length* n*, and*
(d) In a finite $\mathcal{S}$ *of length* n*,* $q_n \in M^* \cup \{\perp\}$

where item b is a requirements for synthesizer correctness, and items a, c and d are requirements for user correctness.

Definition 8 (Convergence). *A synthesis session* $(A_0, q1), (A_1, q_2), \ldots, (A_n, q_n)$ *is said to* converge *if* $\gamma(S_n) \subseteq M^*$*. It has* converged successfully *if* $\gamma(S_n) \neq \emptyset$.

Existing interactive synthesis guarantees require well-behaved users.

Assumptions about user behavior

Abstraction-Based Interaction Model for Synthesis

Hila Peleg[1], Shachar Itzhaky[1], and Sharon Shoham[2]

[1] Technion, {hilap,shachari}@cs.technion.ac.il
[2] Tel Aviv University, sharon.shoham@gmail.com

Definition 5 (User correctness). *A user step, providing A_i as an additional specification, is* correct *when* $A_i \subseteq \{p \in \mathcal{P} \mid \exists m \in U^*.m \vDash p\}$.

Definition 6 (Synthesis user). *The* behavior of the user *includes the following guarantees:*

1. *The user is correct for as long as they can be. If the user can no longer provide an answer that is correct, they will answer* $\perp$.
2. *If a user sees a program in* M^**, they will accept it.*

Definition 7 (Feasible synthesis session). *A* feasible synthesis session *is a synthesis session* $\mathcal{S} = (A_0, q1), (A_1, q_2), \ldots$ *that satisfies the following:*

(a) All A_i *are correct steps (definition 5) or* $\perp$,
(b) $q_i = Select(S_{i-1})$*, i.e.* $q_i \in \gamma(S_{i-1}) \cup \{\perp\}$*, where* $\perp$ *signifies no possible program,*
(c) If $q_n \in M^* \cup \{\perp\}$ *then* $\mathcal{S}$ *is finite and of length* n*, and*
(d) In a finite $\mathcal{S}$ *of length* n*,* $q_n \in M^* \cup \{\perp\}$

where item b is a requirements for synthesizer correctness, and items a, c and d are requirements for user correctness.

Definition 8 (Convergence). *A synthesis session* $(A_0, q1), (A_1, q_2), \ldots, (A_n, q_n)$ *is said to* converge *if* $\gamma(S_n) \subseteq M^*$*. It has* converged successfully *if* $\gamma(S_n) \neq \emptyset$.

Existing interactive synthesis guarantees require well-behaved users.

Assumptions about user behavior

Convergence guarantee

Abstraction-Based Interaction Model for Synthesis

Hila Peleg[1], Shachar Itzhaky[1], and Sharon Shoham[2]

[1] Technion, {hilap,shachari}@cs.technion.ac.il
[2] Tel Aviv University, sharon.shoham@gmail.com

Definition 5 (User correctness). *A user step, providing A_i as an additional specification, is* correct *when* $A_i \subseteq \{p \in \mathcal{P} \mid \exists m \in U^*.m \vDash p\}$.

Definition 6 (Synthesis user). *The* behavior of the user *includes the following guarantees:*

1. *The user is correct for as long as they can be. If the user can no longer provide an answer that is correct, they will answer* $\perp$.
2. *If a user sees a program in* M^**, they will accept it.*

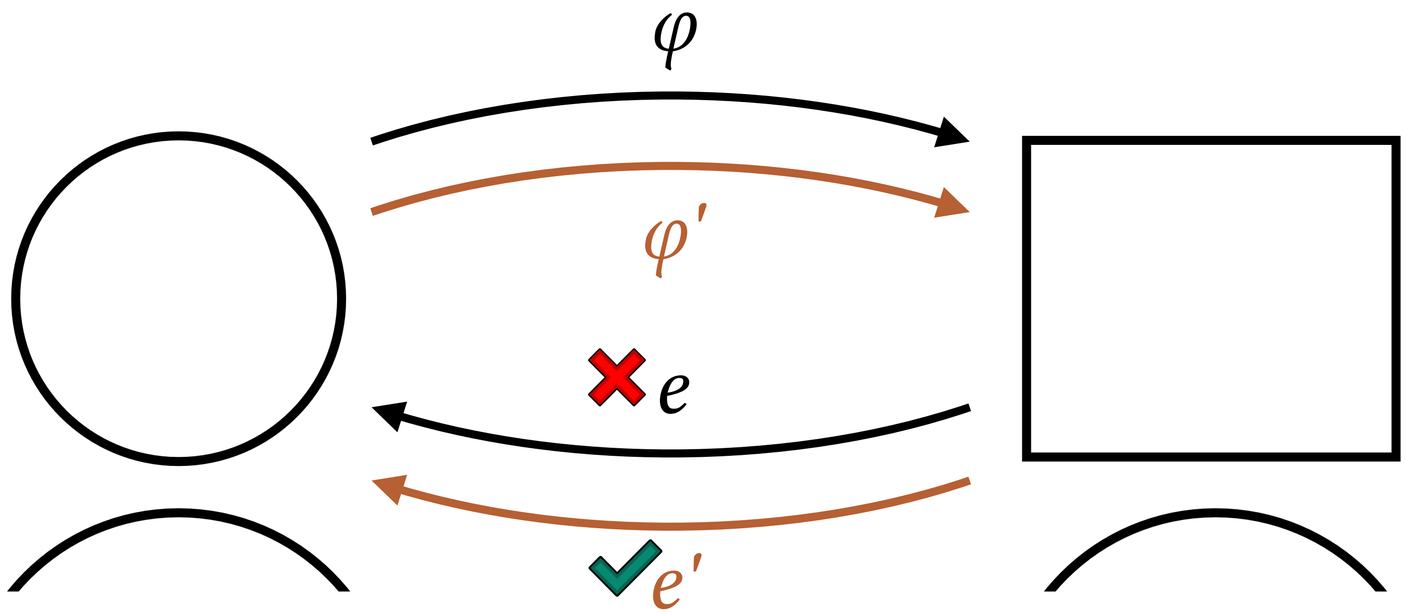
Definition 7 (Feasible synthesis session). *A* feasible synthesis session *is a synthesis session* $\mathcal{S} = (A_0, q1), (A_1, q_2), \ldots$ *that satisfies the following:*

(a) All A_i *are correct steps (definition 5) or* $\perp$,
(b) $q_i = Select(S_{i-1})$*, i.e.* $q_i \in \gamma(S_{i-1}) \cup \{\perp\}$*, where* $\perp$ *signifies no possible program,*
(c) If $q_n \in M^* \cup \{\perp\}$ *then* $\mathcal{S}$ *is finite and of length* n*, and*
(d) In a finite $\mathcal{S}$ *of length* n*,* $q_n \in M^* \cup \{\perp\}$

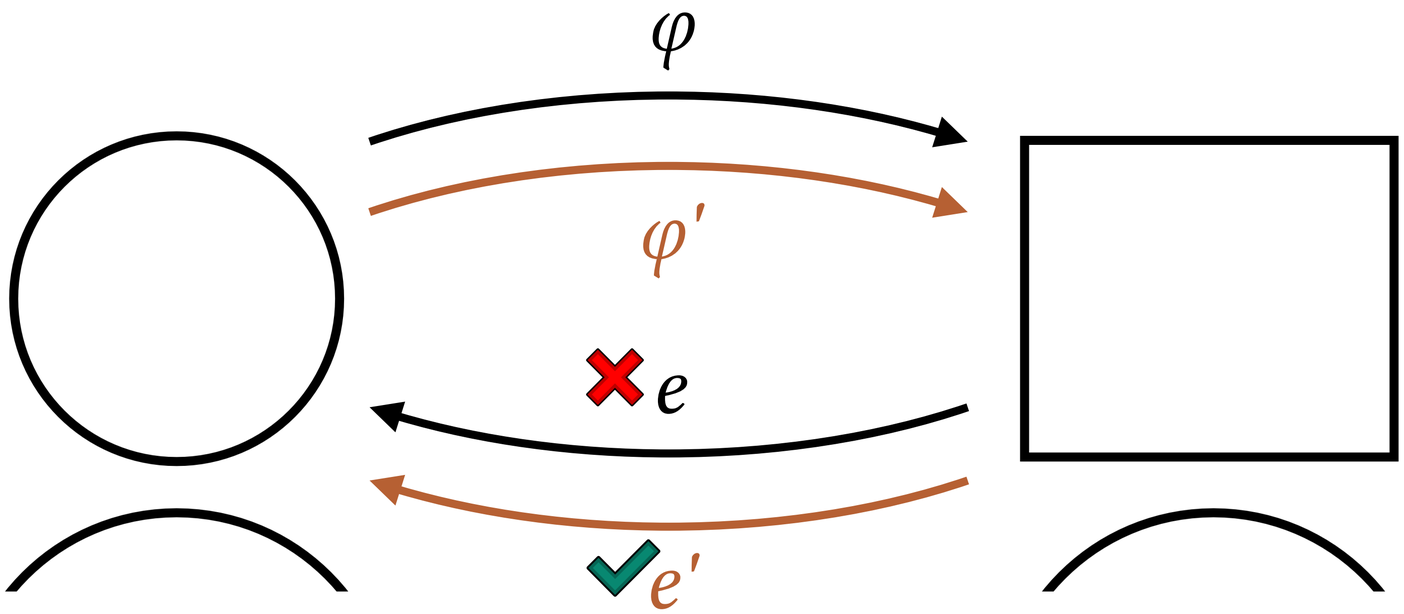
where item b is a requirements for synthesizer correctness, and items a, c and d are requirements for user correctness.

Definition 8 (Convergence). *A synthesis session* $(A_0, q1), (A_1, q_2), \ldots, (A_n, q_n)$ *is said to* converge *if* $\gamma(S_n) \subseteq M^*$*. It has* converged successfully *if* $\gamma(S_n) \neq \emptyset$.

How could the user misbehave, and what could go wrong?

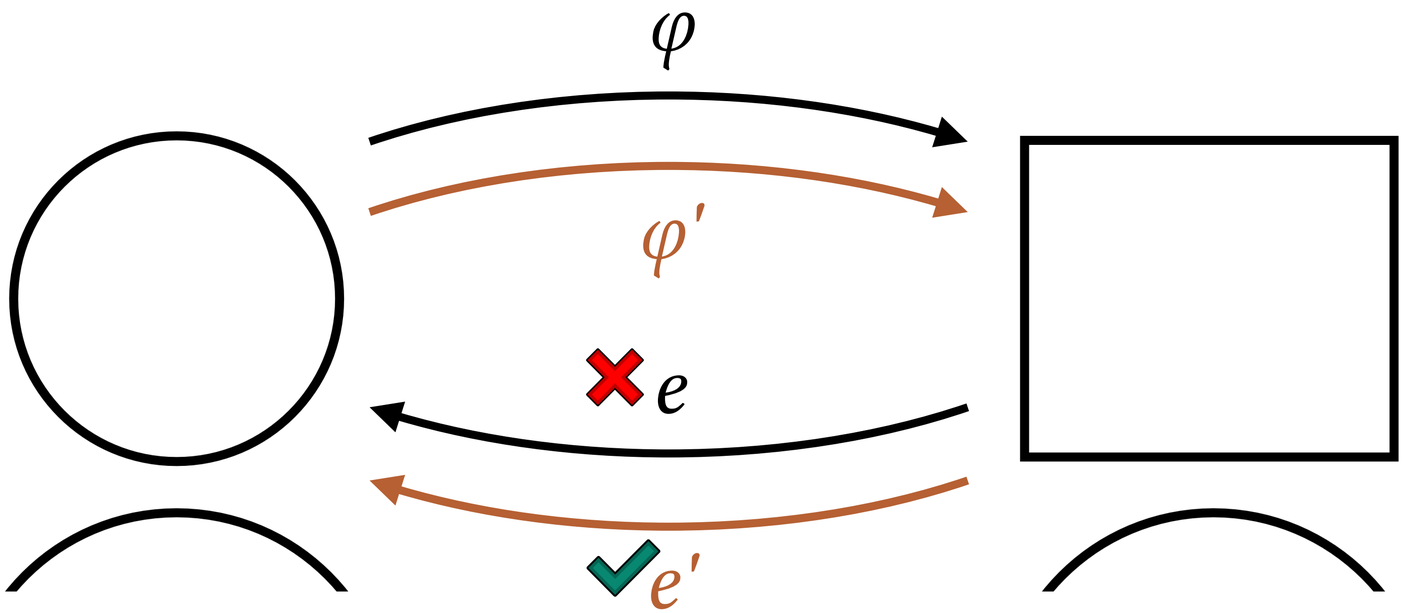


How could the user misbehave, and what could go wrong?



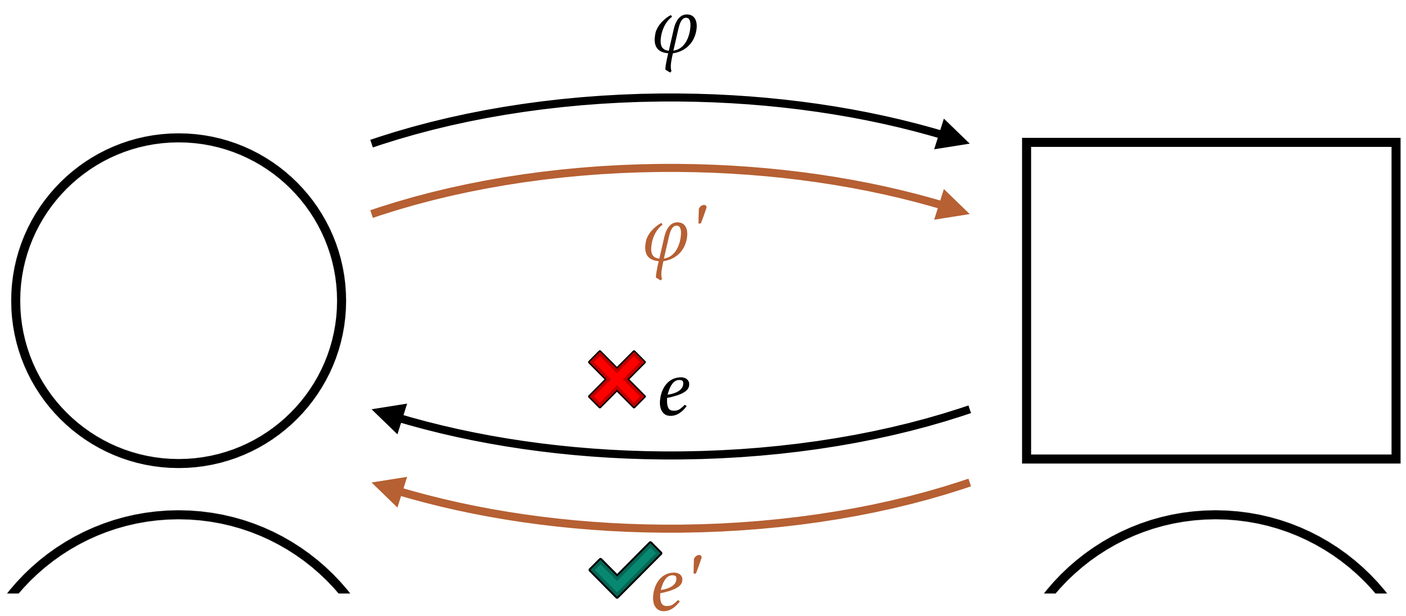
1. Refined specification could be unsatisfiable (φ' collapses to ⊥).

How could the user misbehave, and what could go wrong?



1. Refined specification could be unsatisfiable (φ' collapses to ⊥).
2. Synthesizer could render valid expressions inaccessible.

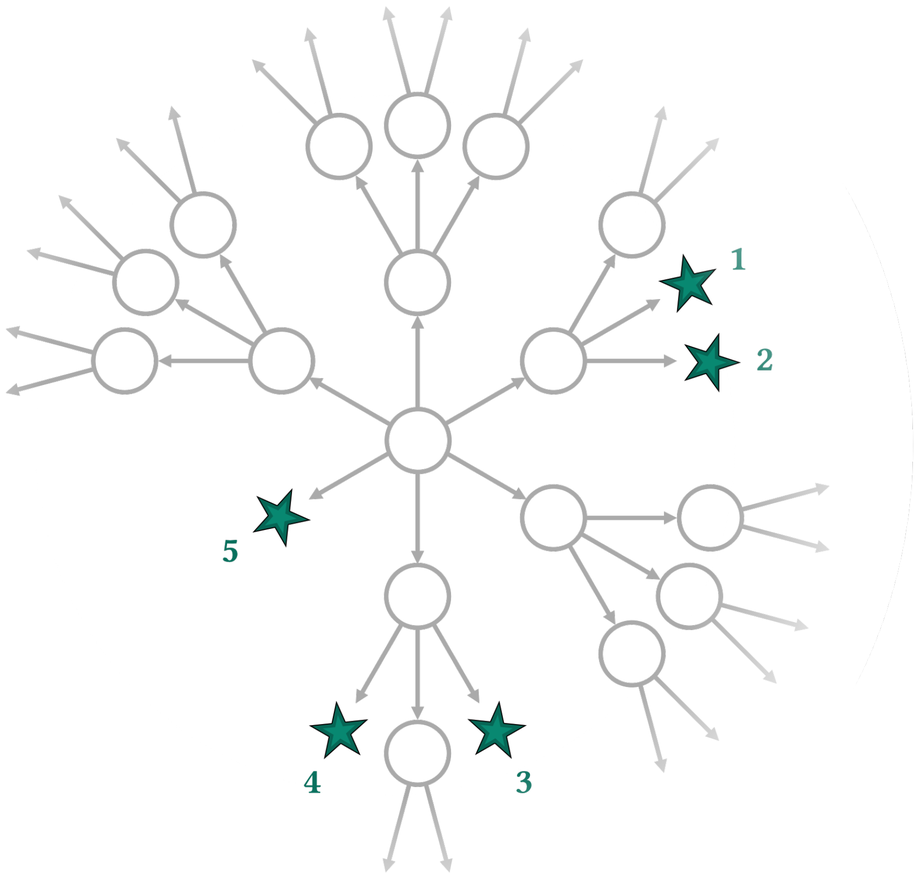
How could the user misbehave, and what could go wrong?



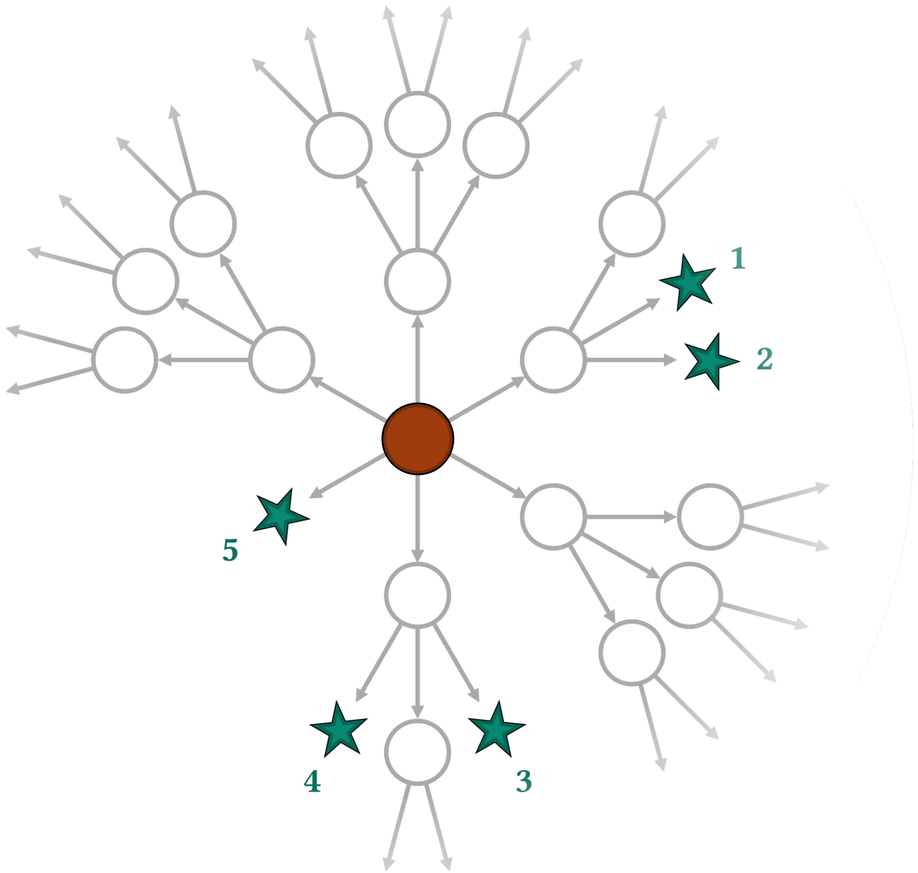
1. Refined specification could be unsatisfiable (φ' collapses to ⊥).
2. Synthesizer could render valid expressions inaccessible.
3. User could go down “rabbit hole” of refining a specification; the changes they’re making never yield the program they want.

Instead, we propose STRONG COMPLETENESS and STRONG SOUNDNESS.

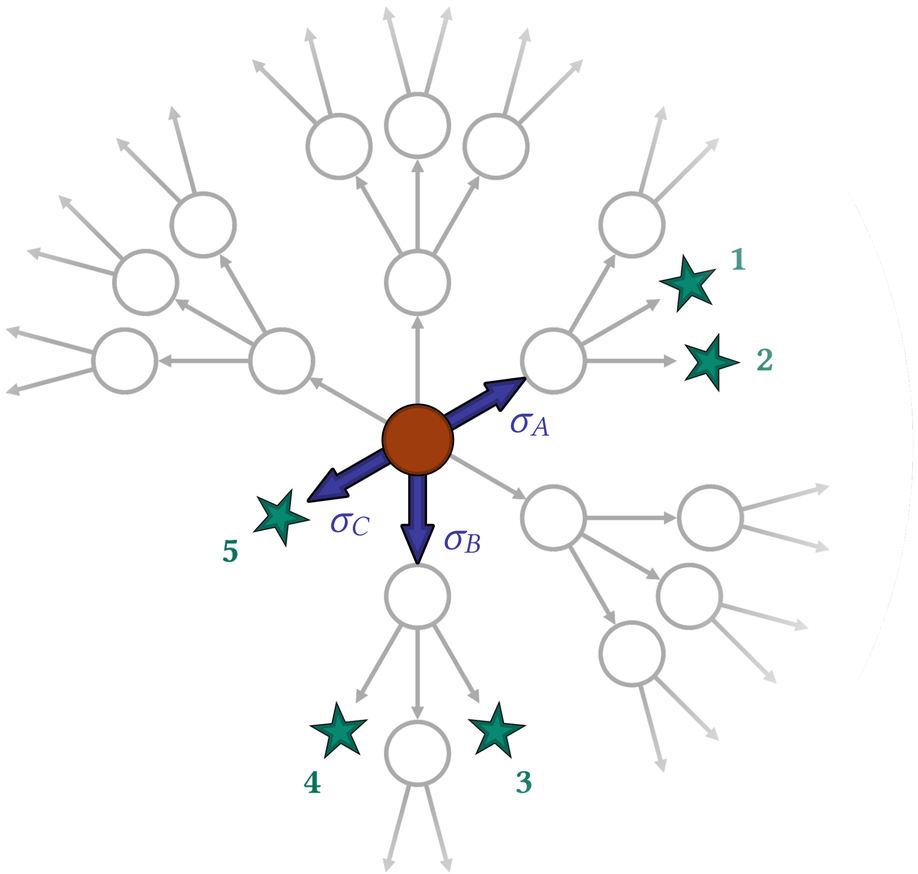
Instead, we propose Strong Completeness and Strong Soundness.



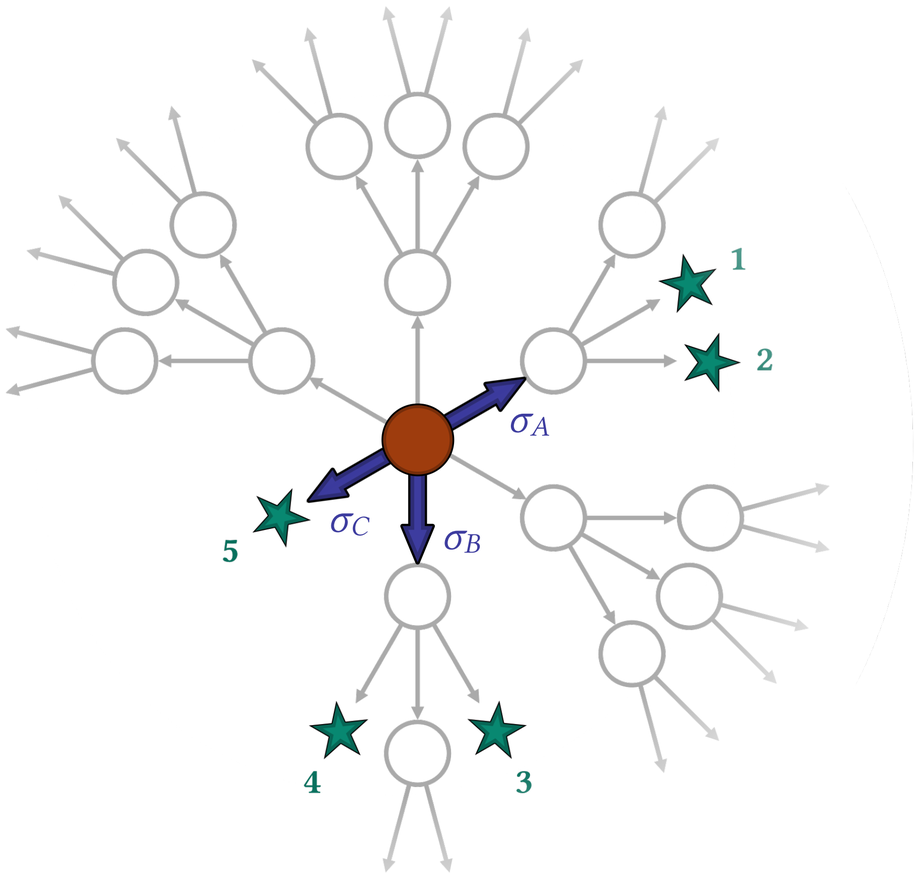
Instead, we propose Strong Completeness and Strong Soundness.



Instead, we propose Strong Completeness and Strong Soundness.

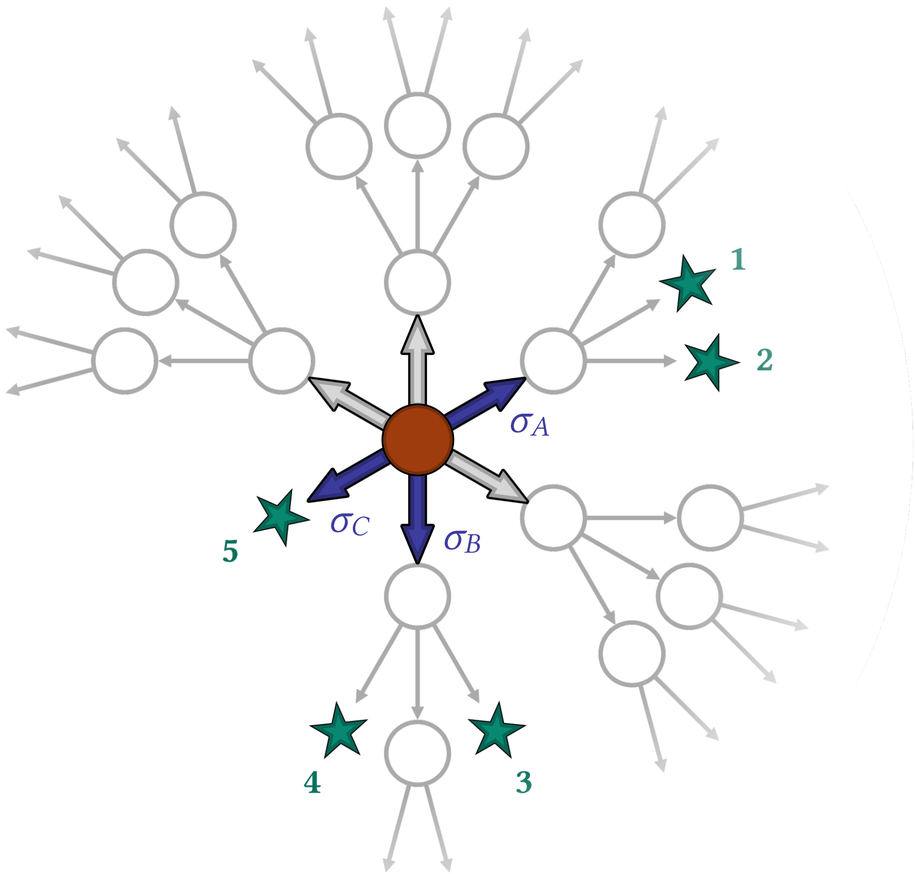


Instead, we propose STRONG COMPLETENESS and STRONG SOUNDNESS.



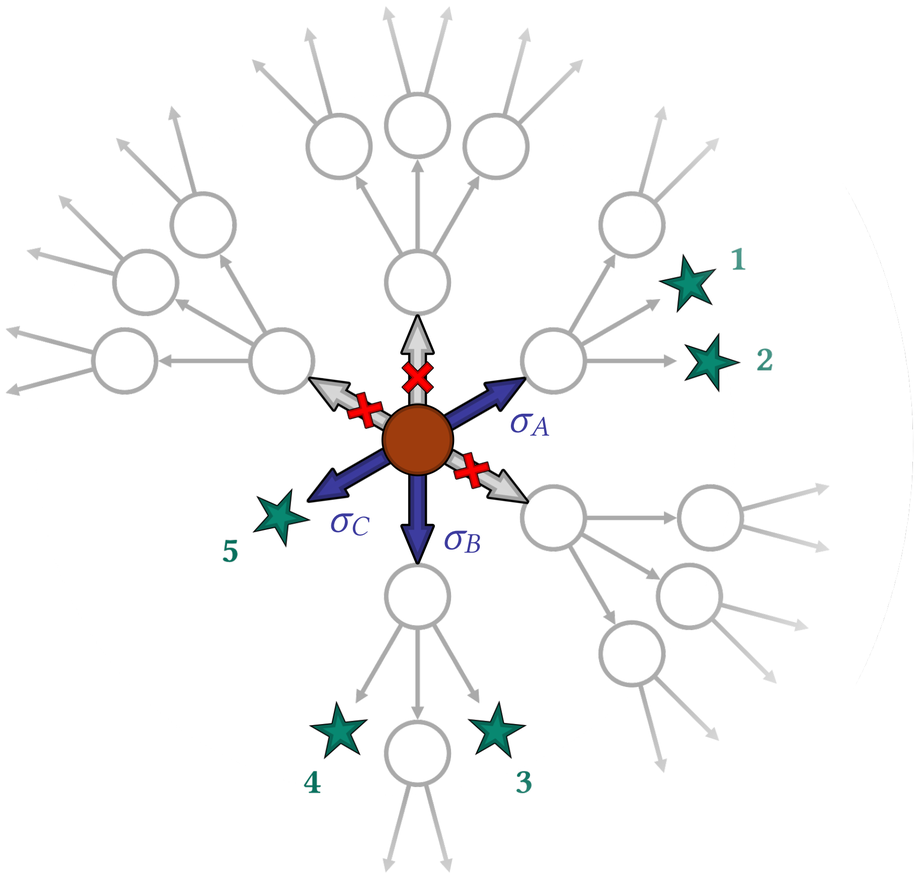
- **STRONG COMPLETENESS.** *All* valid steps are shown.

Instead, we propose STRONG COMPLETENESS and STRONG SOUNDNESS.



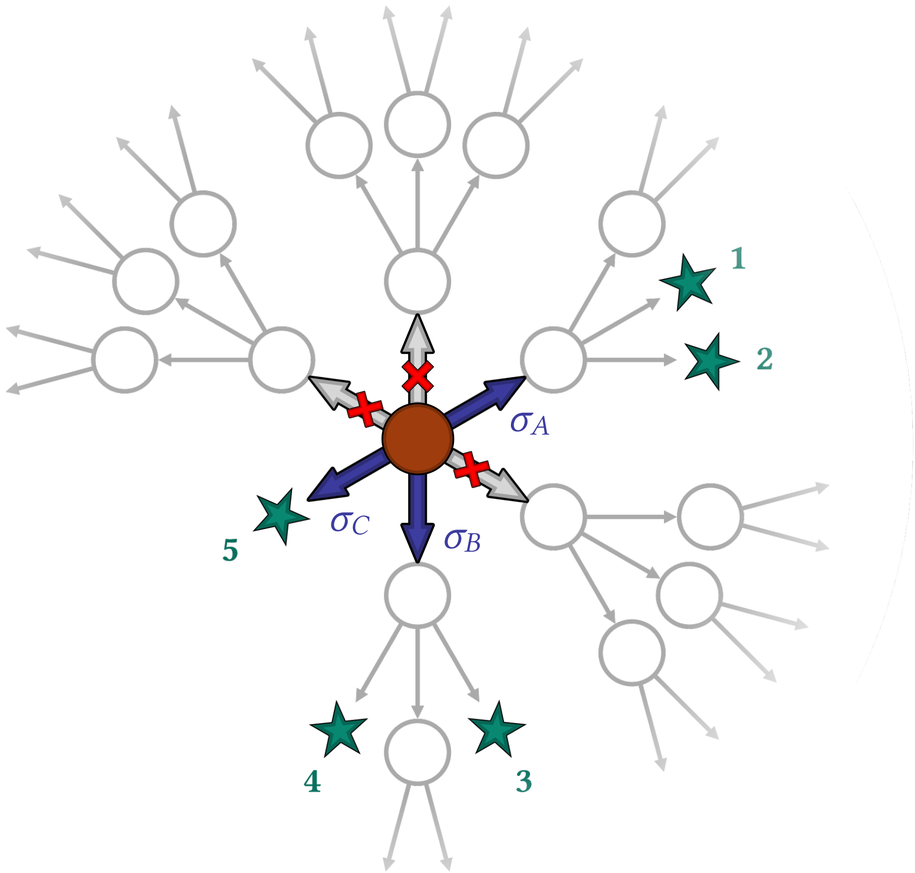
- **STRONG COMPLETENESS.** *All* valid steps are shown.

Instead, we propose STRONG COMPLETENESS and STRONG SOUNDNESS.



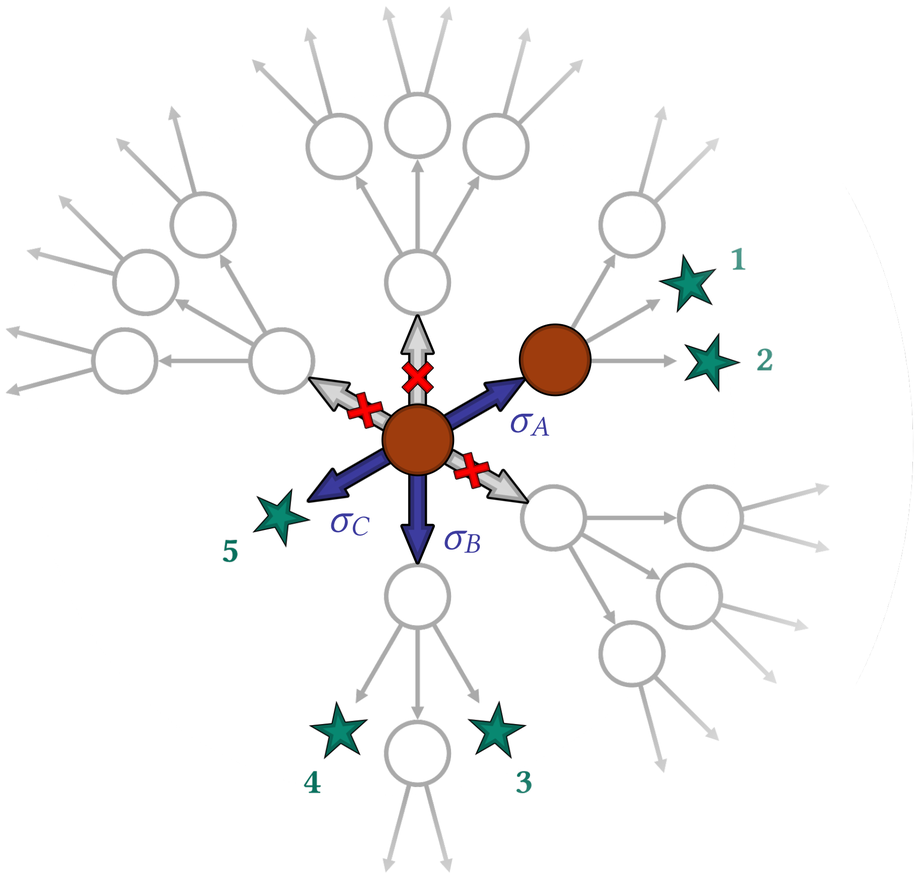
- **STRONG COMPLETENESS.** *All* valid steps are shown.

Instead, we propose STRONG COMPLETENESS and STRONG SOUNDNESS.



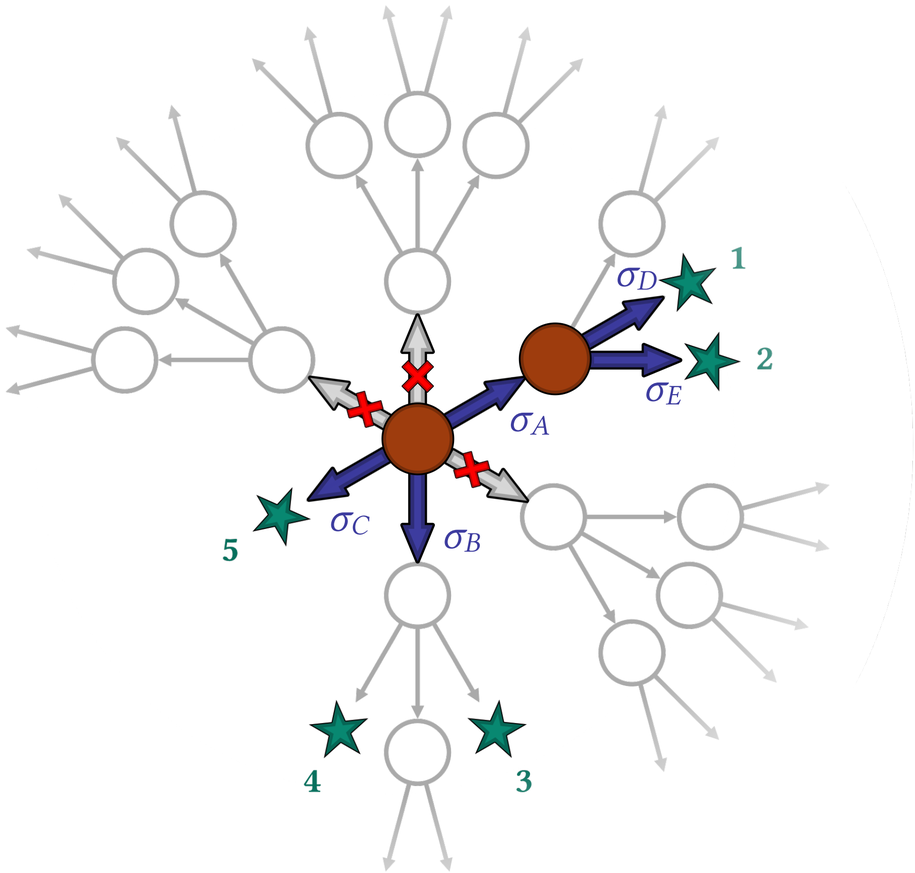
- **STRONG COMPLETENESS.** *All* valid steps are shown.
- **STRONG SOUNDNESS.** *Only* valid steps are shown.

Instead, we propose STRONG COMPLETENESS and STRONG SOUNDNESS.



- **STRONG COMPLETENESS.** *All* valid steps are shown.
- **STRONG SOUNDNESS.** *Only* valid steps are shown.

Instead, we propose STRONG COMPLETENESS and STRONG SOUNDNESS.



- **STRONG COMPLETENESS.** *All* valid steps are shown.
- **STRONG SOUNDNESS.** *Only* valid steps are shown.

Question 1:

What are **STRONG COMPLETENESS** and **STRONG SOUNDNESS**?

Question 1:

What are **STRONG COMPLETENESS** and **STRONG SOUNDNESS**?

Question 2:

How do we achieve **STRONG COMPLETENESS** and **STRONG SOUNDNESS**?

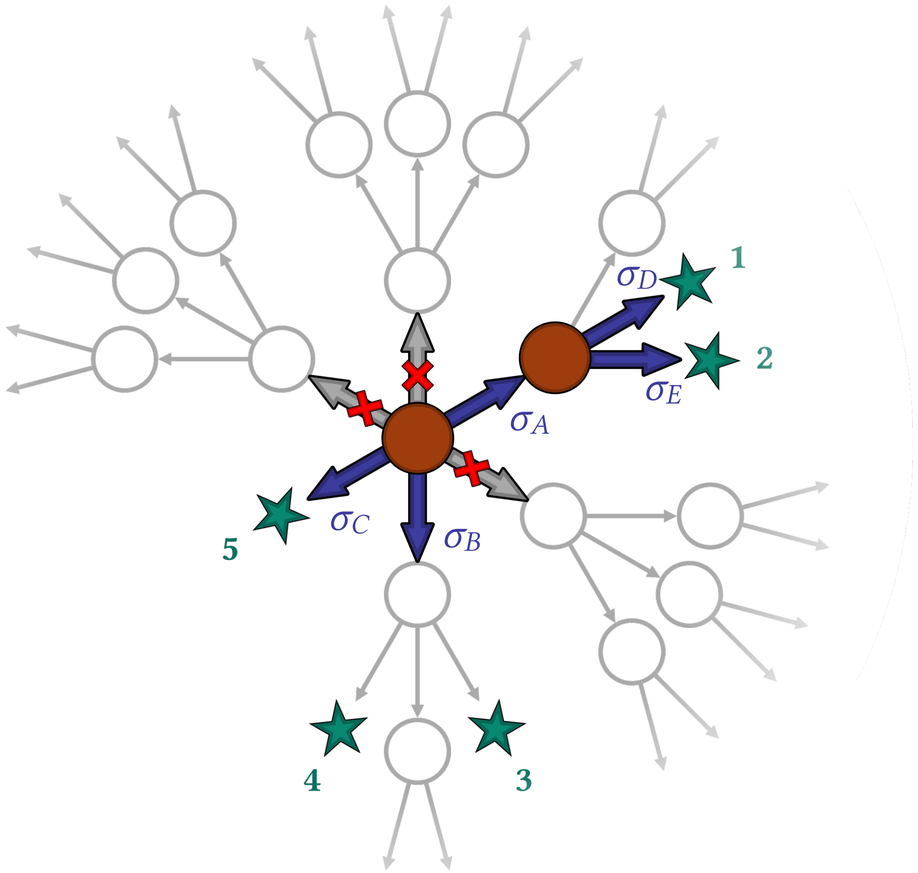
Question 1:

What are STRONG COMPLETENESS and STRONG SOUNDNESS?

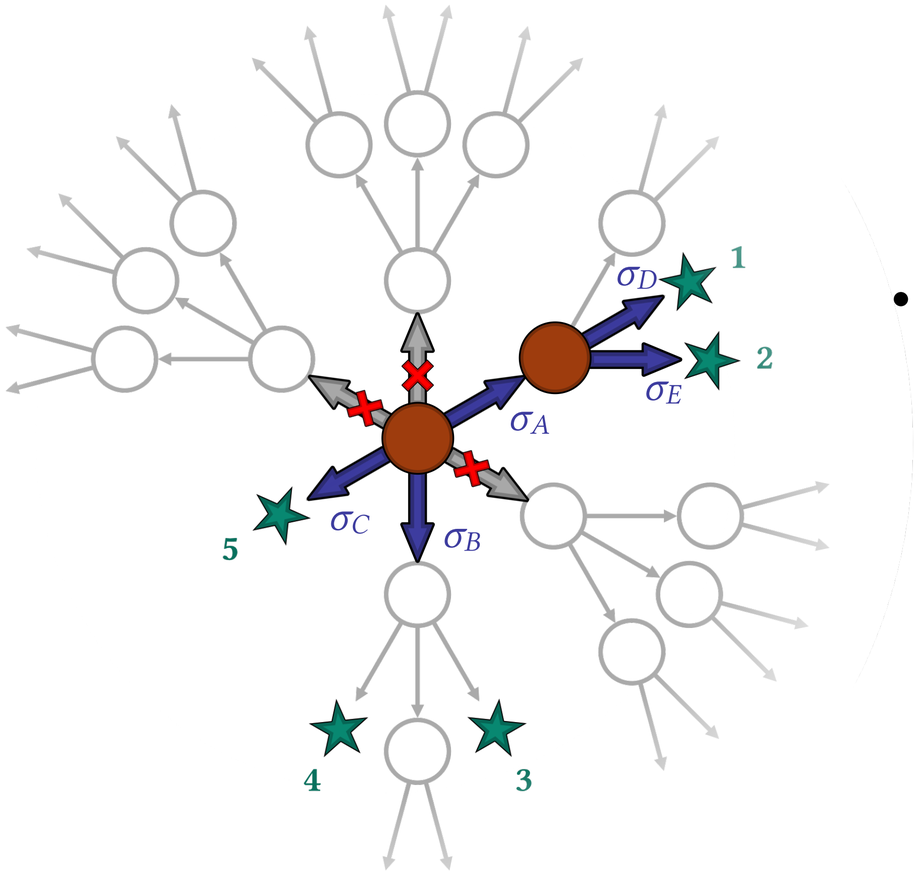
Question 2:

How do we achieve **STRONG COMPLETENESS** and **STRONG SOUNDNESS**?

Steps are the building blocks for STRONG COMPLETENESS and STRONG SOUNDNESS.

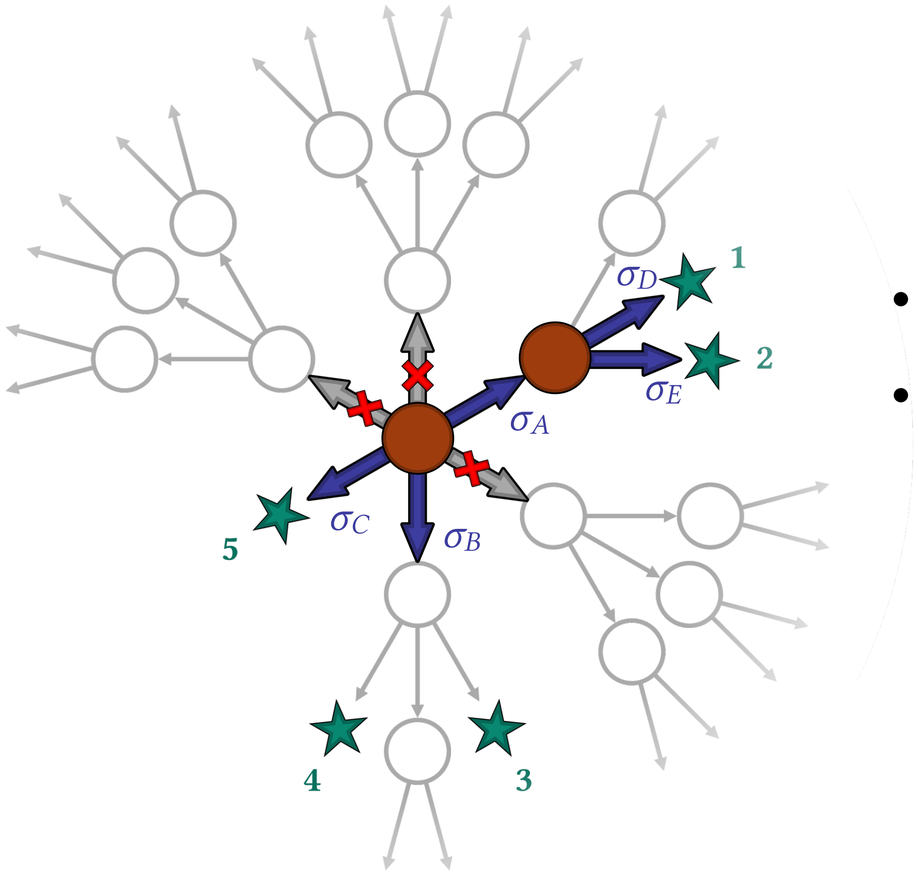


Steps are the building blocks for STRONG COMPLETENESS and STRONG SOUNDNESS.



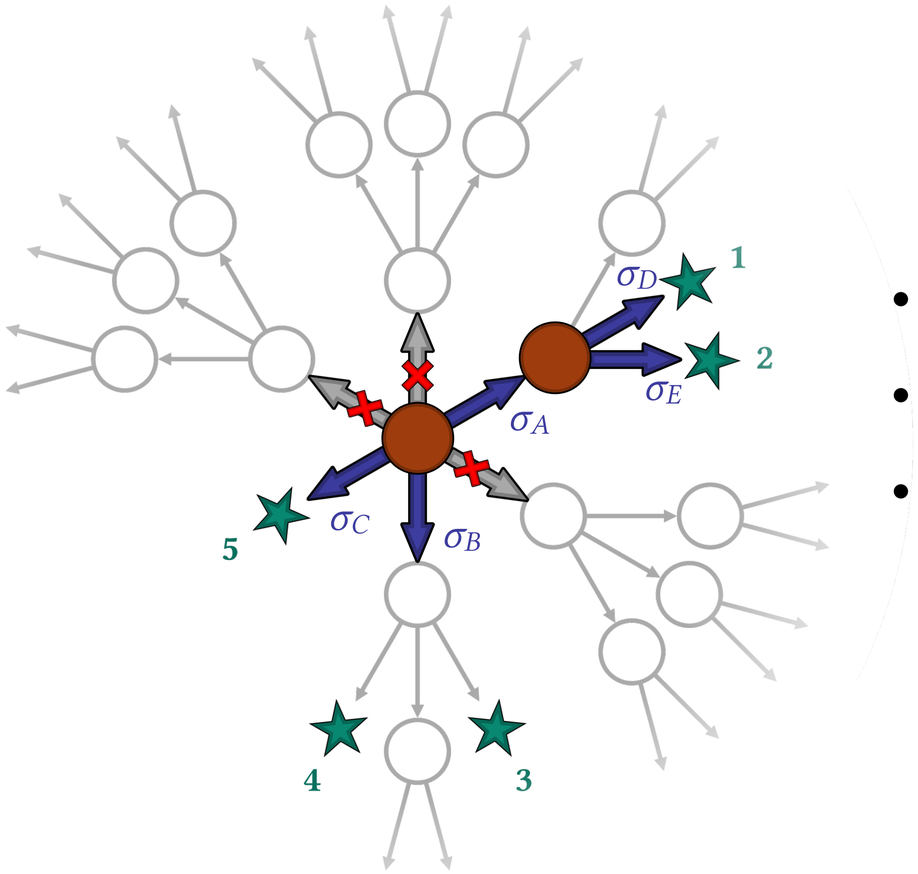
- Expressions e, steps σ

Steps are the building blocks for STRONG COMPLETENESS and STRONG SOUNDNESS.



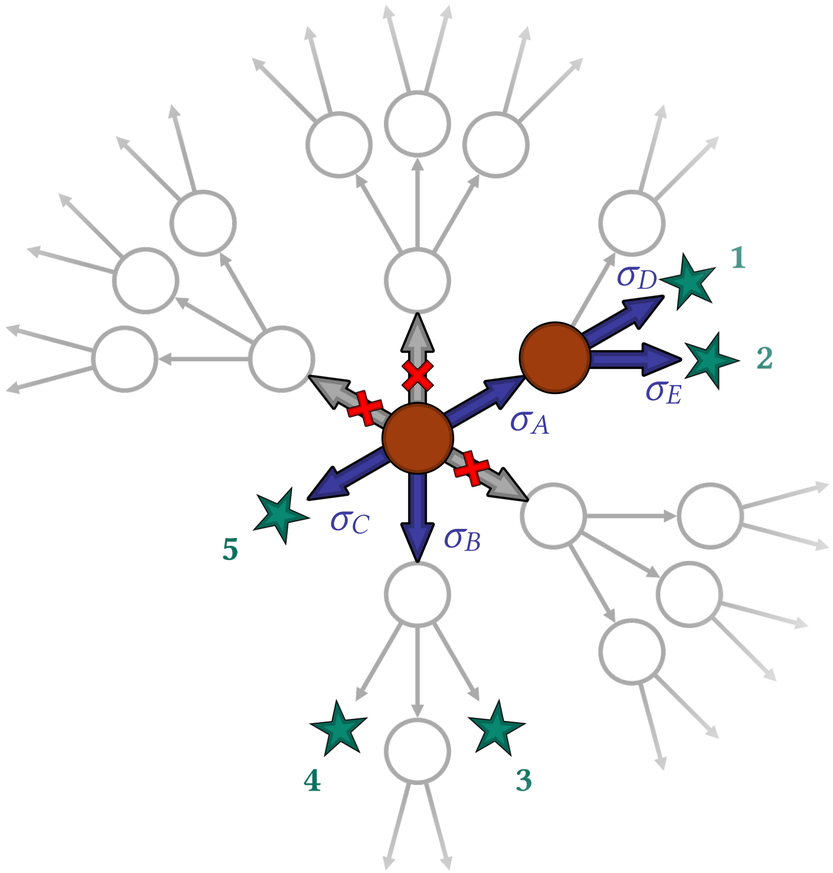
- Expressions e, steps σ
- Validity: e valid

Steps are the building blocks for Strong Completeness and Strong Soundness.



- Expressions e, steps σ
- Validity: e valid
- Step relation $e_1 \xrightarrow{\sigma} e_2$ (also written $\sigma e_1 = e_2$), induced relation $e_1 < e_2$.

Steps are the building blocks for Strong Completeness and Strong Soundness.



- Expressions e, steps σ
- Validity: e valid
- Step relation $e_1 \xrightarrow{\sigma} e_2$ (also written $\sigma e_1 = e_2$), induced relation $e_1 < e_2$.
- Steps need to satisfy mild conditions (such as determinism).

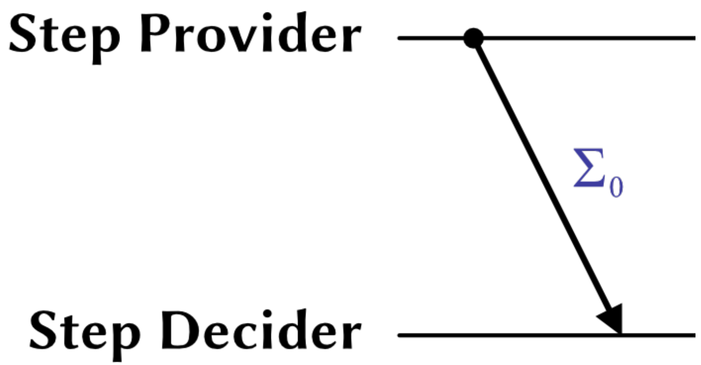
Step providers take the place of synthesizers.

Step providers take the place of synthesizers.

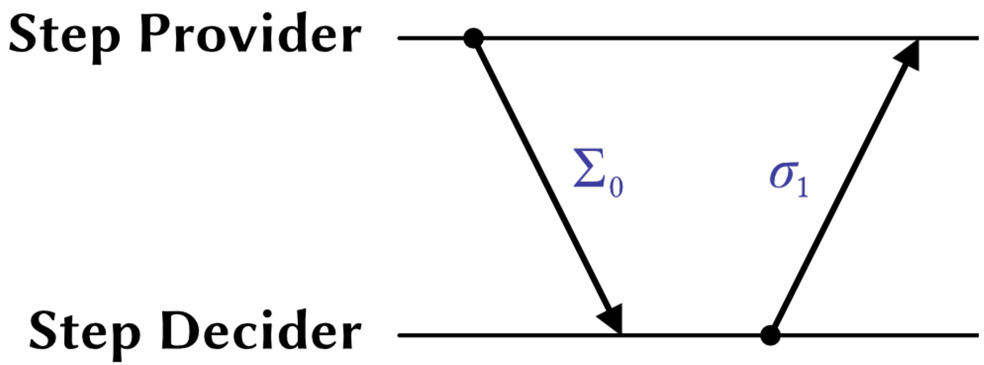
Step Provider

Step Decider

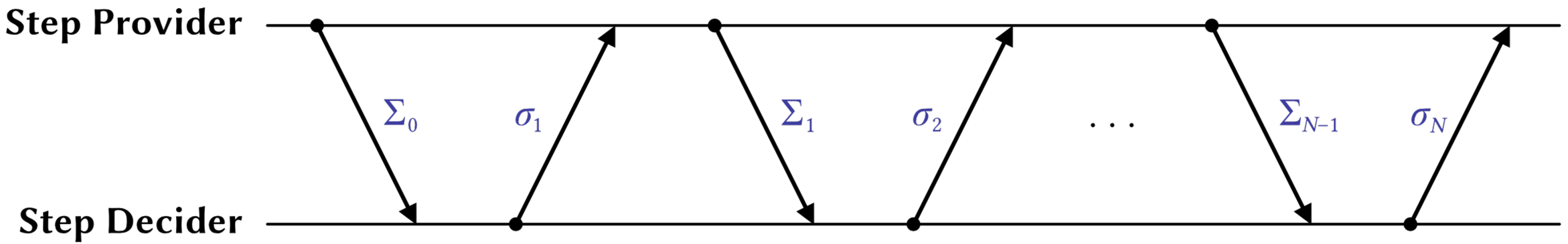
Step providers take the place of synthesizers.



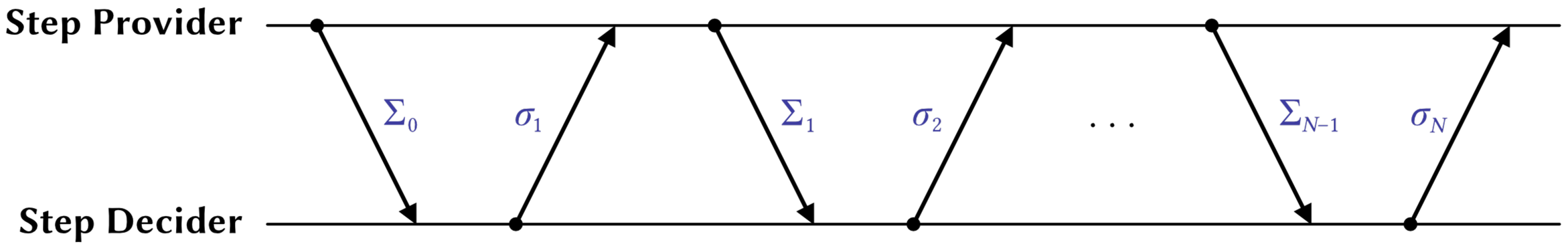
Step providers take the place of synthesizers.



Step providers take the place of synthesizers.

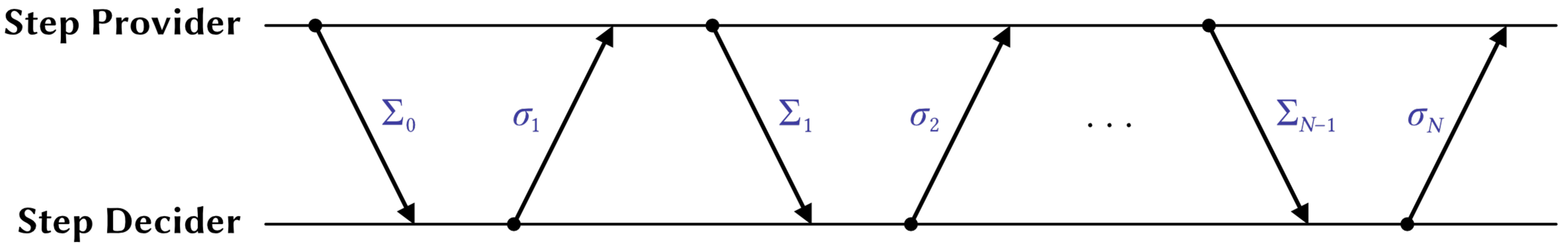


Step providers take the place of synthesizers.



- A **step provider** $\mathbb{S}$ maps expressions to step sets.

Step providers take the place of synthesizers.



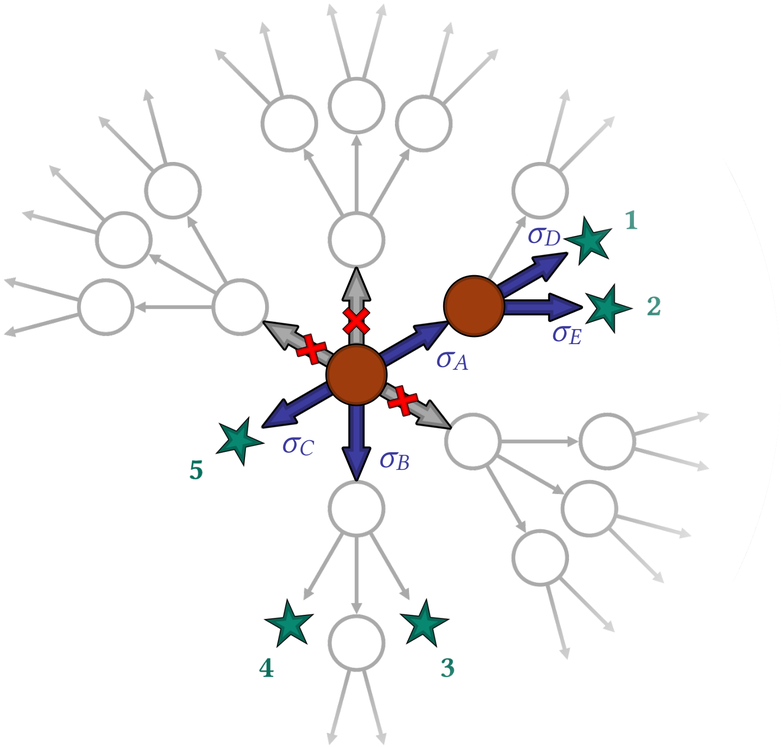
- A **step provider** $\mathbb{S}$ maps expressions to step sets.
- An $\mathbb{S}$**-interaction** is a finite sequence $e_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_N} e_N$ such that $\sigma_{k+1} \in \mathbb{S}(e_k)$ for all $0 \leq k < N$ (and e_0 is a designated "blank program").

The *Programming by Navigation Synthesis Problem* defines STRONG COMPLETENESS and STRONG SOUNDNESS.

The *Programming by Navigation Synthesis Problem* defines Strong Completeness and Strong Soundness.

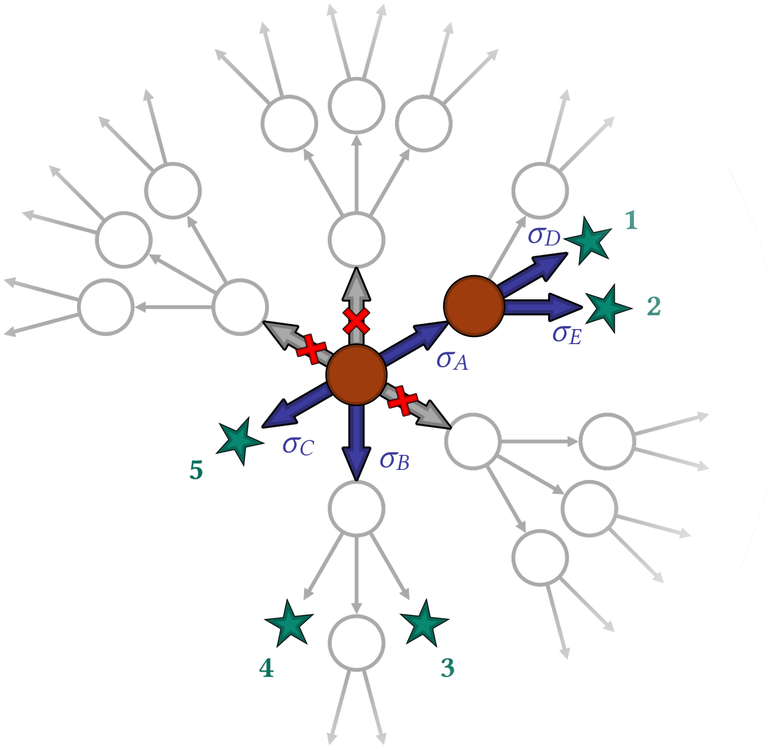
- The **completion** of an expression e is $C(e) = \{e' \mid e \preceq e' \wedge e' \text{ valid}\}$.

The *Programming by Navigation Synthesis Problem* defines Strong Completeness and Strong Soundness.



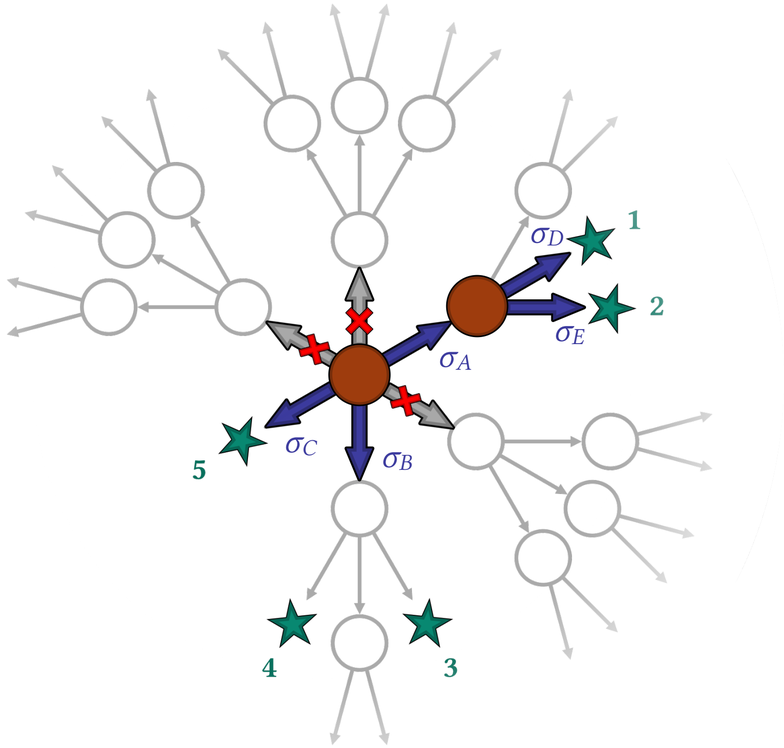
- The **completion** of an expression e is $C(e) = \{e' \mid e \preceq e' \wedge e' \text{ valid}\}$.

The *Programming by Navigation Synthesis Problem* defines Strong Completeness and Strong Soundness.



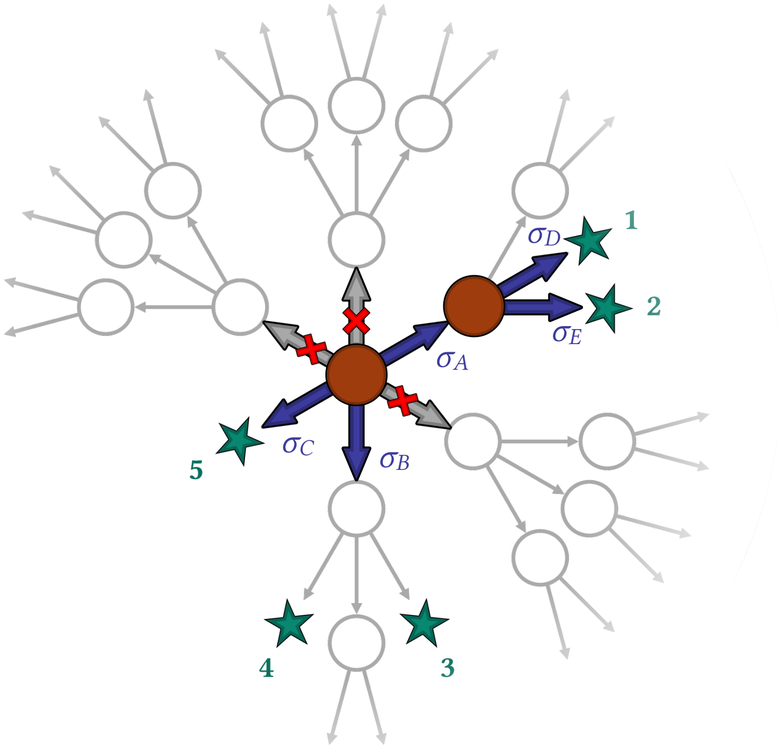
- The **completion** of an expression e is $C(e) = \{e' \mid e \preceq e' \wedge e' \text{ valid}\}$.
- A step set Σ **covers** an expression e if it satisfies:

The *Programming by Navigation Synthesis Problem* defines Strong Completeness and Strong Soundness.



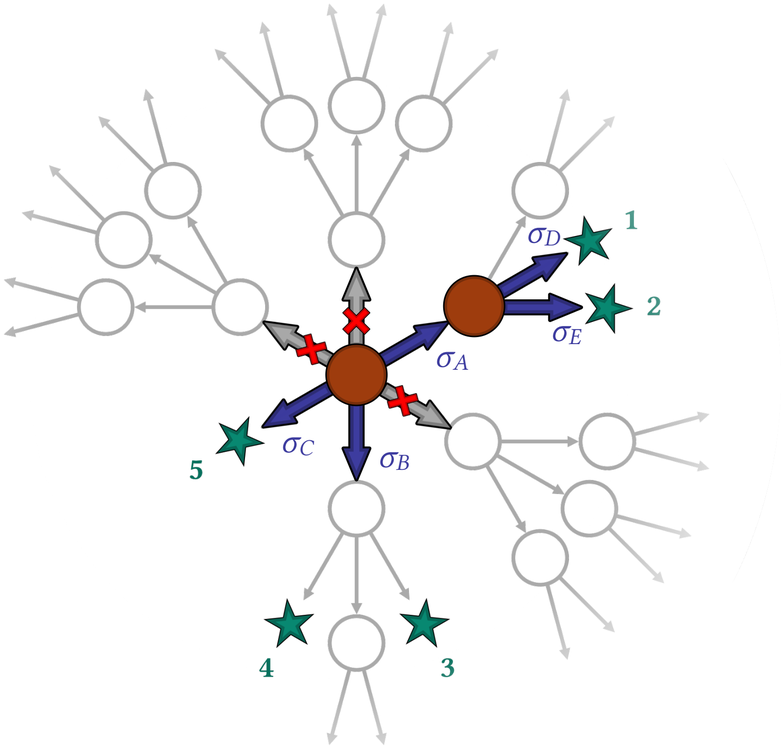
- The **completion** of an expression e is $C(e) = \{e' \mid e \preceq e' \wedge e' \text{ valid}\}$.
- A step set Σ **covers** an expression e if it satisfies:
  - **Strong Completeness.**

The *Programming by Navigation Synthesis Problem* defines Strong Completeness and Strong Soundness.



- The **completion** of an expression e is $C(e) = \{e' \mid e \preceq e' \wedge e' \text{ valid}\}$.
- A step set Σ **covers** an expression e if it satisfies:
  - **Strong Completeness.**
  - **Strong Soundness.**

The *Programming by Navigation Synthesis Problem* defines Strong Completeness and Strong Soundness.

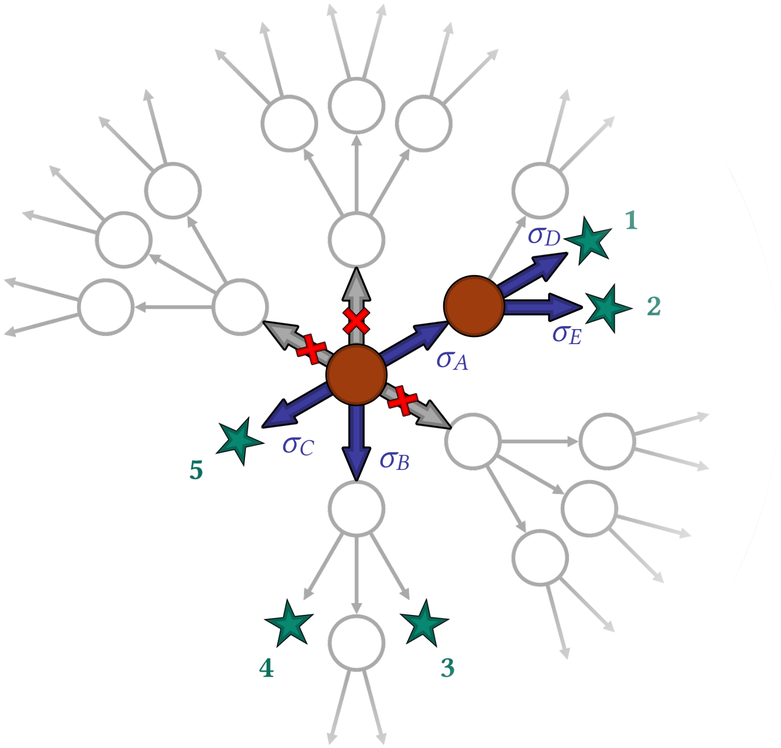


- The **completion** of an expression e is $C(e) = \{e' \mid e \preceq e' \wedge e' \text{ valid}\}$.
- A step set Σ **covers** an expression e if it satisfies:
  - **Strong Completeness.**

$$C(e) \setminus \{e\} \subseteq \bigcup_{\sigma \in \Sigma} C(\sigma e)$$

  - **Strong Soundness.**

The *Programming by Navigation Synthesis Problem* defines Strong Completeness and Strong Soundness.

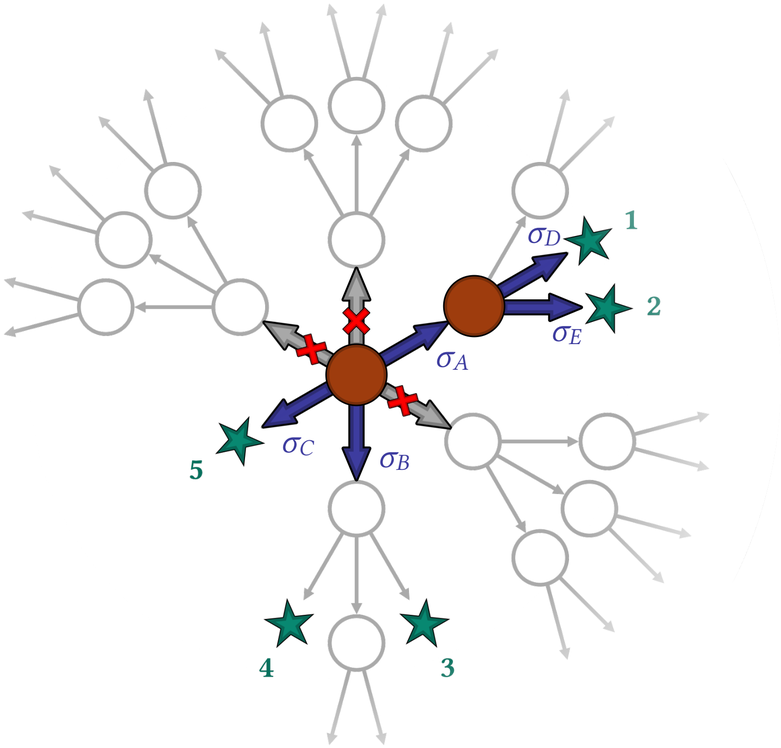


- The **completion** of an expression e is $C(e) = \{e' \mid e \preceq e' \wedge e' \text{ valid}\}$.
- A step set Σ **covers** an expression e if it satisfies:
  - **Strong Completeness.**

$$C(e) \setminus \{e\} \subseteq \bigcup_{\sigma \in \Sigma} C(\sigma e)$$

  - **Strong Soundness.**

The *Programming by Navigation Synthesis Problem* defines Strong Completeness and Strong Soundness.

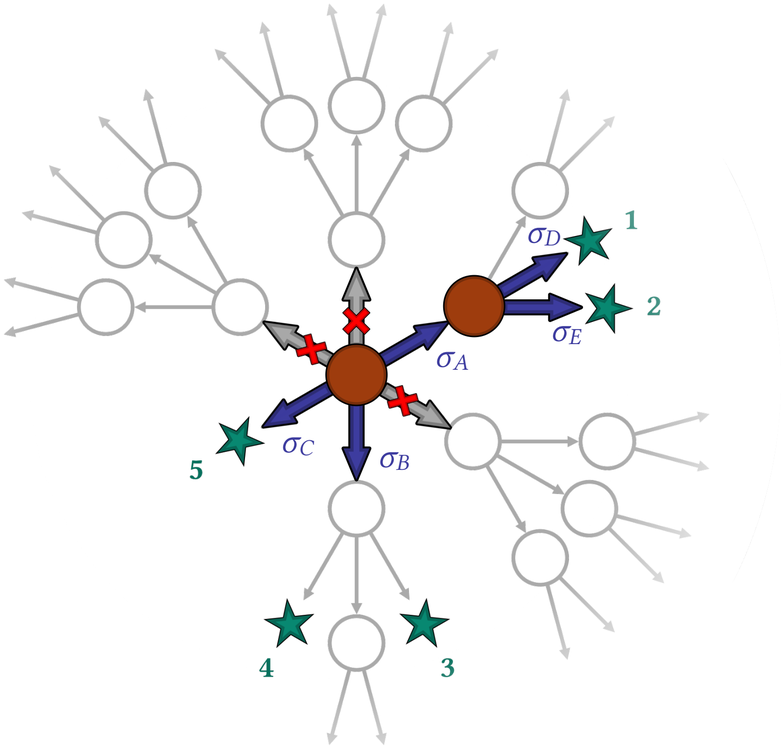


- The **completion** of an expression e is $C(e) = \{e' \mid e \preceq e' \wedge e' \text{ valid}\}$.
- A step set Σ **covers** an expression e if it satisfies:
  - **Strong Completeness.**

$$C(e) \setminus \{e\} \subseteq \bigcup_{\sigma \in \Sigma} C(\sigma e)$$

  - **Strong Soundness.**

The *Programming by Navigation Synthesis Problem* defines Strong Completeness and Strong Soundness.

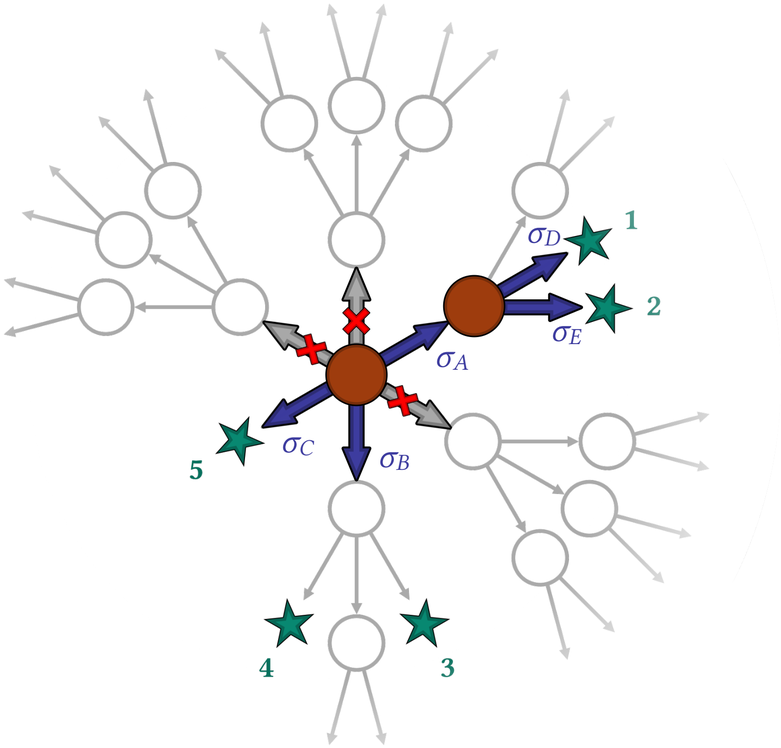


- The **completion** of an expression e is $C(e) = \{e' \mid e \preceq e' \wedge e' \text{ valid}\}$.
- A step set Σ **covers** an expression e if it satisfies:
  - **Strong Completeness.**

$$C(e) \setminus \{e\} \subseteq \bigcup_{\sigma \in \Sigma} C(\sigma e)$$

  - **Strong Soundness.**

The *Programming by Navigation Synthesis Problem* defines Strong Completeness and Strong Soundness.



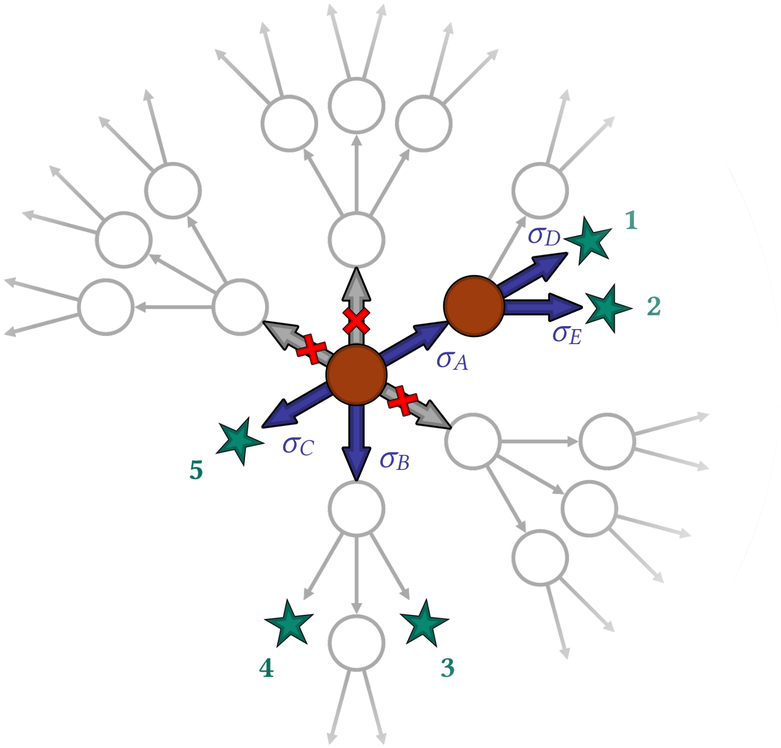
- The **completion** of an expression e is $C(e) = \{e' \mid e \preceq e' \wedge e' \text{ valid}\}$.
- A step set Σ **covers** an expression e if it satisfies:
  - **Strong Completeness.**

$$C(e) \setminus \{e\} \subseteq \bigcup_{\sigma \in \Sigma} C(\sigma e)$$

  - **Strong Soundness.**

$$C(\sigma e) \neq \emptyset \text{ for all } \sigma \in \Sigma$$

The *Programming by Navigation Synthesis Problem* defines Strong Completeness and Strong Soundness.



- The **completion** of an expression e is $C(e) = \{e' \mid e \preceq e' \wedge e' \text{ valid}\}$.
- A step set Σ **covers** an expression e if it satisfies:
  - **Strong Completeness.**

$$C(e) \setminus \{e\} \subseteq \bigcup_{\sigma \in \Sigma} C(\sigma e)$$

  - **Strong Soundness.**

$$C(\sigma e) \neq \emptyset \text{ for all } \sigma \in \Sigma$$

Problem Statement

A step provider $\mathbb{S}$ solves the **Programming by Navigation Synthesis Problem** if $\mathbb{S}(e_N)$ covers e_N for all $\mathbb{S}$-interactions $e_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_N} e_N$.

Question 1:

What are STRONG COMPLETENESS and STRONG SOUNDNESS?

Question 2:

How do we achieve **STRONG COMPLETENESS** and **STRONG SOUNDNESS**?

Question 1:

What are Strong Completeness and Strong Soundness?

Answer: Properties of step providers in Programming by Navigation (in particular, that they must show all and only the valid next steps).

Question 2:

How do we achieve **Strong Completeness** and **Strong Soundness**?

Question 1:

What are **Strong Completeness** and **Strong Soundness**?

Answer: Properties of step providers in Programming by Navigation (in particular, that they must show all and only the valid next steps).

Question 2:

How do we achieve Strong Completeness and Strong Soundness?

Programming by Navigation for component-based synthesis with top-down steps.

Programming by Navigation for component-based synthesis with top-down steps.

Types

Programming by Navigation for component-based synthesis with top-down steps.

Types

Functions

Programming by Navigation for component-based synthesis with top-down steps.

Types **Functions** **Validity Conditions**

Programming by Navigation for component-based synthesis with top-down steps.

Types	Functions	Validity Conditions
$R\langle \text{int} \rangle$		

Programming by Navigation for component-based synthesis with top-down steps.

Types

$R\langle \text{int} \rangle$

$R\langle 2 \rangle$ could be… “the type of an integer vector where each entry is the number of times an RNA transcript appears in tissue sample 2”

Functions

Validity Conditions

Programming by Navigation for component-based synthesis with top-down steps.

Types

$R\langle \text{int} \rangle$

$R\langle 2 \rangle$ could be… “the type of an integer vector where each entry is the number of times an RNA transcript appears in tissue sample 2”

$R\langle 90 \rangle$ could be… “the type of a geospatial raster image with spatial resolution of 90 meters^2 / pixel.”

Functions

Validity Conditions

Programming by Navigation for component-based synthesis with top-down steps.

Types

$R\langle \text{int} \rangle$

$R\langle 2 \rangle$ could be… “the type of an integer vector where each entry is the number of times an RNA transcript appears in tissue sample 2”

$R\langle 90 \rangle$ could be… “the type of a geospatial raster image with spatial resolution of 90 meters^2 / pixel.”

Functions

$f : R \rightarrow_{\varphi} R$

Validity Conditions

Programming by Navigation for component-based synthesis with top-down steps.

Types

$R\langle \text{int} \rangle$

$R\langle 2 \rangle$ could be… "the type of an integer vector where each entry is the number of times an RNA transcript appears in tissue sample 2"

$R\langle 90 \rangle$ could be… "the type of a geospatial raster image with spatial resolution of 90 meters^2 / pixel."

Functions

$f : R \rightarrow_{\varphi} R$

"Perform RNA-seq quality checks."

Validity Conditions

Programming by Navigation for component-based synthesis with top-down steps.

Types	Functions	Validity Conditions
$R\langle \text{int} \rangle$	$f : R \to_{\varphi} R$	
$R\langle 2 \rangle$ could be… "the type of an integer vector where each entry is the number of times an RNA transcript appears in tissue sample 2"	"Perform RNA-seq quality checks."	
$R\langle 90 \rangle$ could be… "the type of a geospatial raster image with spatial resolution of 90 meters^2 / pixel."	"Downsample the raster image."	

Programming by Navigation for component-based synthesis with top-down steps.

Types	Functions	Validity Conditions
$R\langle \text{int} \rangle$	$f : R \rightarrow_{\varphi} R$	
$R\langle 2 \rangle$ could be… "the type of an integer vector where each entry is the number of times an RNA transcript appears in tissue sample 2"	"Perform RNA-seq quality checks."	$\varphi := \text{param}_{1,1} = \text{ret}_1$
$R\langle 90 \rangle$ could be… "the type of a geospatial raster image with spatial resolution of 90 meters^2 / pixel."	"Downsample the raster image."	

Programming by Navigation for component-based synthesis with top-down steps.

Types	Functions	Validity Conditions
$R\langle \text{int} \rangle$	$f : R \rightarrow_{\varphi} R$	
$R\langle 2 \rangle$ could be… "the type of an integer vector where each entry is the number of times an RNA transcript appears in tissue sample 2"	"Perform RNA-seq quality checks."	$\varphi := \text{param}_{1,1} = \text{ret}_1$
$R\langle 90 \rangle$ could be… "the type of a geospatial raster image with spatial resolution of 90 meters^2 / pixel."	"Downsample the raster image."	$\varphi := \text{param}_{1,1} < \text{ret}_1$

Programming by Navigation for component-based synthesis with top-down steps.

Types

$R\langle \text{int} \rangle$
$T\langle \text{int} \rangle$
$A\langle \text{int} \rangle$
$M\langle \text{int}, \text{int}, \text{bool} \rangle$
$D\langle \text{int}, \text{int} \rangle$

Functions

$$l : () \to_{\varphi_1} R$$
$$f : R \to_{\varphi_2} R$$
$$t : R \to_{\varphi_2} R$$
$$q_1, q_2 : R \to_{\varphi_2} T$$
$$a_1, a_2 : R \to_{\varphi_2} A$$
$$s : A \to_{\varphi_2} T$$
$$c : T \times T \to_{\varphi_3} M$$
$$b : M \to_{\varphi_4} M$$
$$d : M \to_{\varphi_5} D$$

Validity conditions

$\varphi_1 := S(\text{ret}_1)$
$\varphi_2 := \text{param}_{1,1} = \text{ret}_1$
$\varphi_3 := \text{param}_{1,1} = \text{ret}_1 \wedge \text{param}_{2,1} = \text{ret}_2 \wedge \neg \text{ret}_3$
$\varphi_4 := \text{param}_{1,1} = \text{ret}_1 \wedge \text{param}_{1,2} = \text{ret}_2 \wedge \text{ret}_3 \wedge \neg \text{param}_{1,3}$
$\varphi_5 := \text{param}_{1,1} = \text{ret}_1 \wedge \text{param}_{2,1} = \text{ret}_2$

Validity is defined via a well-typing judgment for expressions.

Validity is defined via a well-typing judgment for expressions.

- Expressions are function applications or holes

Validity is defined via a well-typing judgment for expressions.

- Expressions are function applications or holes
- Functions applications are well-typed when the function's validity condition is met.

Validity is defined via a well-typing judgment for expressions.

- Expressions are function applications or holes
- Functions applications are well-typed when the function's validity condition is met.
- Only ground terms are well-typed (can't know if validity condition holds with a hole).

Validity is defined via a well-typing judgment for expressions.

- Expressions are function applications or holes
- Functions applications are well-typed when the function's validity condition is met.
- Only ground terms are well-typed (can't know if validity condition holds with a hole).

$$\textsc{Well-Typed/Fun}$$

$$\frac{\begin{array}{c}\Gamma(f) = \tau_1, \ldots, \tau_N \to_\varphi \tau \\ \forall i, j.\, v_{ij} \in \mathsf{vals}(\Gamma) \cup \mathsf{vals}(\Delta) \cup \mathsf{vals}(\overline{v}) \\ \Delta \models \varphi[\overline{v_1}, \ldots, \overline{v_N}; \overline{v}] \\ \forall i.\, \Gamma, \Delta \vdash e_i : \tau_i(\overline{v_i})\end{array}}{\Gamma, \Delta \vdash f^{\overline{v}}(e_1, \ldots, e_N) : \tau(\overline{v})}$$

Validity is defined via a well-typing judgment for expressions.

- Expressions are function applications or holes
- Functions applications are well-typed when the function's validity condition is met.
- Only ground terms are well-typed (can't know if validity condition holds with a hole).

$$\textsc{Well-Typed/Fun}\quad \frac{\begin{array}{c}\Gamma(f) = \tau_1, \ldots, \tau_N \to_\varphi \tau \\ \forall i, j.\, v_{ij} \in \mathsf{vals}(\Gamma) \cup \mathsf{vals}(\Delta) \cup \mathsf{vals}(\overline{v}) \\ \Delta \models \varphi[\overline{v_1}, \ldots, \overline{v_N}; \overline{v}] \\ \forall i.\, \Gamma, \Delta \vdash e_i : \tau_i(\overline{v_i})\end{array}}{\Gamma, \Delta \vdash f^{\overline{v}}(e_1, \ldots, e_N) : \tau(\overline{v})}$$

Validity is defined via a well-typing judgment for expressions.

- Expressions are function applications or holes
- Functions applications are well-typed when the function's validity condition is met.
- Only ground terms are well-typed (can't know if validity condition holds with a hole).

$$\textsc{Well-Typed/Fun}\ \frac{\begin{array}{c}\Gamma(f) = \tau_1, \ldots, \tau_N \to_\varphi \tau \\ \forall i, j.\, v_{ij} \in \mathsf{vals}(\Gamma) \cup \mathsf{vals}(\Delta) \cup \mathsf{vals}(\overline{v}) \\ \Delta \models \varphi[\overline{v_1}, \ldots, \overline{v_N}; \overline{v}] \\ \forall i.\, \Gamma, \Delta \vdash e_i : \tau_i(\overline{v_i})\end{array}}{\Gamma, \Delta \vdash f^{\overline{v}}(e_1, \ldots, e_N) : \tau(\overline{v})}$$

Validity is defined via a well-typing judgment for expressions.

- Expressions are function applications or holes
- Functions applications are well-typed when the function's validity condition is met.
- Only ground terms are well-typed (can't know if validity condition holds with a hole).

$$\textsc{Well-Typed/Fun}\quad \frac{\begin{array}{c}\Gamma(f) = \tau_1, \ldots, \tau_N \to_\varphi \tau \\ \forall i, j.\, v_{ij} \in \mathsf{vals}(\Gamma) \cup \mathsf{vals}(\Delta) \cup \mathsf{vals}(\overline{v}) \\ \Delta \models \varphi[\overline{v_1}, \ldots, \overline{v_N}; \overline{v}] \\ \forall i.\, \Gamma, \Delta \vdash e_i : \tau_i(\overline{v_i})\end{array}}{\Gamma, \Delta \vdash f^{\overline{v}}(e_1, \ldots, e_N) : \tau(\overline{v})}$$

An example Programing by Navigation interaction for top-down steps.

An example Programing by Navigation interaction for top-down steps.

Goal type: $D\langle 1,2\rangle$

An example Programing by Navigation interaction for top-down steps.

Goal type: $D\langle 1,2\rangle$

1

Working sketch: $?_1$

Goal: $?_1 : D$

An example Programing by Navigation interaction for top-down steps.

Goal type: $D\langle 1,2\rangle$

All and only valid next steps

1

Working sketch: $?_1$

Goal: $?_1 : D$

An example Programing by Navigation interaction for top-down steps.

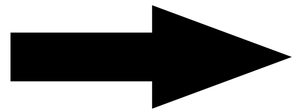
Goal type: $D\langle 1,2 \rangle$

All and only valid next steps

1

Working sketch: $?_1$

Goal: $?_1 : D$



$\{ \bullet \; ?_1 \mapsto d^{1,2}(?_2)$

An example Programing by Navigation interaction for top-down steps.

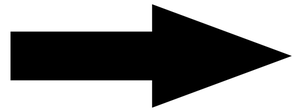
Goal type: $D\langle 1,2 \rangle$

All and only valid next steps

1

Working sketch: $?_1$

Goal: $?_1 : D$



$\{ \bullet \; ?_1 \mapsto d^{1,2}(?_2)$

2

Working sketch: $d^{1,2}(?_2)$

Goal: $?_2 : M$

An example Programing by Navigation interaction for top-down steps.

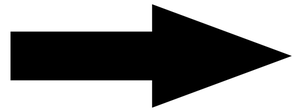
Goal type: $D\langle 1,2 \rangle$

All and only valid next steps

1

Working sketch: $?_1$

Goal: $?_1 : D$

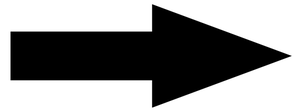


- $?_1 \mapsto d^{1,2}(?_2)$

2

Working sketch: $d^{1,2}(?_2)$

Goal: $?_2 : M$



- $?_2 \mapsto c^{1,2,\bot}(?_3, ?_4)$
- $?_2 \mapsto b^{1,2,\top}(?_3)$

An example Programing by Navigation interaction for top-down steps.

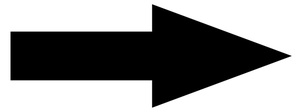
Goal type: $D\langle 1,2\rangle$

All and only valid next steps

1

Working sketch: $?_1$

Goal: $?_1 : D$

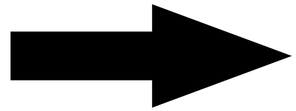


- $?_1 \mapsto d^{1,2}(?_2)$

2

Working sketch: $d^{1,2}(?_2)$

Goal: $?_2 : M$



- $?_2 \mapsto c^{1,2,\bot}(?_3, ?_4)$
- $?_2 \mapsto b^{1,2,\top}(?_3)$

⋮

An example Programing by Navigation interaction for top-down steps.

Synthesizer (step provider) needs to do this.

Goal type: $D\langle 1,2 \rangle$

All and only valid next steps

1

Working sketch: $?_1$

Goal: $?_1 : D$

$$\left\{ \bullet \; ?_1 \mapsto d^{1,2}(?_2) \right.$$

2

Working sketch: $d^{1,2}(?_2)$

Goal: $?_2 : M$

$$\left\{ \begin{array}{l} \bullet \; ?_2 \mapsto c^{1,2,\bot}(?_3, ?_4) \\ \bullet \; ?_2 \mapsto b^{1,2,\top}(?_3) \end{array} \right.$$

⋮

An example Programing by Navigation interaction for top-down steps.

Synthesizer (step provider) needs to do this.

Goal type: $D\langle 1,2 \rangle$

All and only valid next steps

1

Working sketch: $?_1$

Goal: $?_1 : D$

$$\{ \bullet \; ?_1 \mapsto d^{1,2}(?_2)$$

2

Working sketch: $d^{1,2}(?_2)$

Goal: $?_2 : M$

$$\{ \bullet \; ?_2 \mapsto c^{1,2,\bot}(?_3, ?_4) \quad \bullet \; ?_2 \mapsto b^{1,2,\top}(?_3)$$

User chooses among these.

⋮

An example Programing by Navigation interaction for top-down steps.

How?

Synthesizer (step provider) needs to do this.

Goal type: $D\langle 1,2\rangle$

All and only valid next steps

1

Working sketch: $?_1$

Goal: $?_1 : D$

$$\left\{ \bullet \;\; ?_1 \mapsto d^{1,2}(?_2) \right.$$

2

Working sketch: $d^{1,2}(?_2)$

Goal: $?_2 : M$

$$\left\{ \begin{array}{l} \bullet \;\; ?_2 \mapsto c^{1,2,\bot}(?_3, ?_4) \\ \bullet \;\; ?_2 \mapsto b^{1,2,\top}(?_3) \end{array} \right.$$

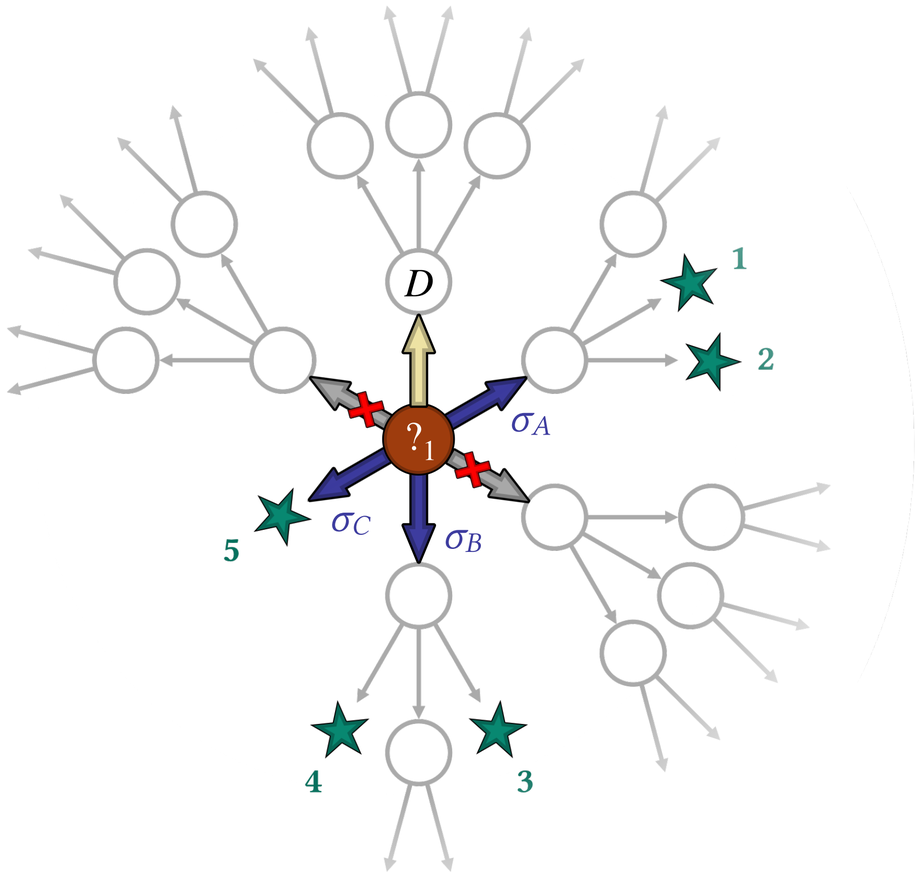
User chooses among these.

⋮

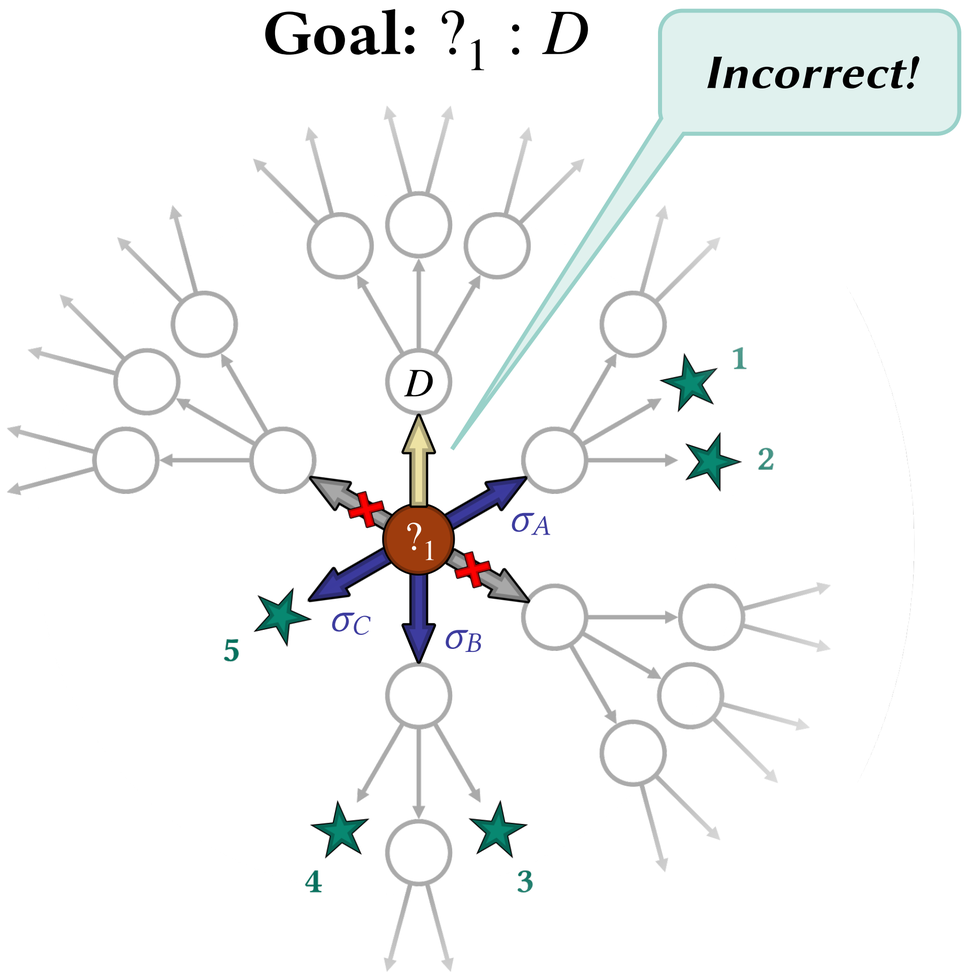
Cannot simply look at grammar induced by the simple types.

Cannot simply look at grammar induced by the simple types.

Goal: $?_1 : D$



Cannot simply look at grammar induced by the simple types.



Cannot enumerate solutions and store in trie-like data structure.

Cannot enumerate solutions and store in trie-like data structure.

$$d^{1,2}(c^{1,2,\perp}(q_1^1(l^1()), q_1^2(l^2())))$$
$$d^{1,2}(c^{1,2,\perp}(q_1^1(l^1()), q_2^2(l^2())))$$
$$d^{1,2}(c^{1,2,\perp}(q_2^1(l^1()), q_1^2(l^2())))$$

Cannot enumerate solutions and store in trie-like data structure.

$d^{1,2}(c^{1,2,\bot}(q_1^1(l^1()), q_1^2(l^2())))$

$d^{1,2}(c^{1,2,\bot}(q_1^1(l^1()), q_2^2(l^2())))$

$d^{1,2}(c^{1,2,\bot}(q_2^1(l^1()), q_1^2(l^2())))$

$d^{1,2}(c^{1,2,\bot}(q_2^1(l^1()), q_2^2(l^2())))$

$d^{1,2}(c^{1,2,\bot}(f^1(q_1^1(l^1())), q_1^2(l^2())))$

$d^{1,2}(c^{1,2,\bot}(q_1^1(l^1())), f^1(q_1^2(l^2()))))$

$d^{1,2}(c^{1,2,\bot}(f^1(q_1^1(l^1())), f^1(q_1^2(l^2()))))$

$d^{1,2}(c^{1,2,\bot}(s^1(a_1^1(l^1())), q_1^2(l^2())))$

$d^{1,2}(c^{1,2,\bot}(s^1(a_2^1(l^1())), q_1^2(l^2())))$

$d^{1,2}(c^{1,2,\bot}(s_1^1(a_1^1(l^1())), q_2^2(l^2())))$

$d^{1,2}(c^{1,2,\bot}(s_1^1(a_2^1(l^1())), q_2^2(l^2())))$

$d^{1,2}(b^{1,2,\top}(c^{1,2,\bot}(q_1^1(l^1()), q_1^2(l^2()))))$

$d^{1,2}(b^{1,2,\top}(c^{1,2,\bot}(q_1^1(l^1()), q_2^2(l^2()))))$

$d^{1,2}(b^{1,2,\top}(c^{1,2,\bot}(q_2^1(l^1()), q_1^2(l^2()))))$

$d^{1,2}(b^{1,2,\top}(c^{1,2,\bot}(q_2^1(l^1()), q_2^2(l^2()))))$

$d^{1,2}(b^{1,2,\top}(c^{1,2,\bot}(f^1(q_1^1(l^1())), q_1^2(l^2()))))$

$d^{1,2}(b^{1,2,\top}(c^{1,2,\bot}(q_1^1(l^1())), f^1(q_1^2(l^2())))))$

$d^{1,2}(b^{1,2,\top}(c^{1,2,\bot}(f^1(q_1^1(l^1())), f^1(q_1^2(l^2()))))))$

Continues infinitely...

We *really* want to extend well-typing to non-ground terms...

$$\textsc{Well-Typed/Fun}\quad \frac{\begin{array}{c}\Gamma(f) = \tau_1, \ldots, \tau_N \to_\varphi \tau \\ \forall i, j.\, v_{ij} \in \mathsf{vals}(\Gamma) \cup \mathsf{vals}(\Delta) \cup \mathsf{vals}(\overline{v}) \\ \Delta \models \varphi[\overline{v_1}, \ldots, \overline{v_N}; \overline{v}] \\ \forall i.\, \Gamma, \Delta \vdash e_i : \tau_i(\overline{v_i})\end{array}}{\Gamma, \Delta \vdash f^{\overline{v}}(e_1, \ldots, e_N) : \tau(\overline{v})}$$

We *really* want to extend well-typing to non-ground terms...

$$\textsc{Well-Typed/Fun}\quad \frac{\begin{array}{c}\Gamma(f) = \tau_1, \ldots, \tau_N \to_\varphi \tau \\ \forall i, j.\, v_{ij} \in \mathsf{vals}(\Gamma) \cup \mathsf{vals}(\Delta) \cup \mathsf{vals}(\overline{v}) \\ \Delta \models \varphi[\overline{v_1}, \ldots, \overline{v_N}; \overline{v}] \\ \forall i.\, \Gamma, \Delta \vdash e_i : \tau_i(\overline{v_i})\end{array}}{\Gamma, \Delta \vdash f^{\overline{v}}(e_1, \ldots, e_N) : \tau(\overline{v})}$$

We *really* want to extend well-typing to non-ground terms...

$$\textsc{Well-Typed/Fun}\quad \frac{\begin{array}{c}\Gamma(f) = \tau_1, \ldots, \tau_N \to_\varphi \tau \\ \forall i, j.\, v_{ij} \in \mathsf{vals}(\Gamma) \cup \mathsf{vals}(\Delta) \cup \mathsf{vals}(\overline{v}) \\ \Delta \models \varphi[\overline{v_1}, \ldots, \overline{v_N}; \overline{v}] \\ \forall i.\, \Gamma, \Delta \vdash e_i : \tau_i(\overline{v_i})\end{array}}{\Gamma, \Delta \vdash f^{\overline{v}}(e_1, \ldots, e_N) : \tau(\overline{v})}$$

Want: "*or* e_i = ? *and* ∃e such that premise holds"

We *really* want to extend well-typing to non-ground terms...

$$\textsc{Well-Typed/Fun}\quad \frac{\begin{array}{c}\Gamma(f) = \tau_1, \ldots, \tau_N \to_\varphi \tau \\ \forall i, j.\, v_{ij} \in \mathrm{vals}(\Gamma) \cup \mathrm{vals}(\Delta) \cup \mathrm{vals}(\overline{v}) \\ \Delta \models \varphi[\overline{v_1}, \ldots, \overline{v_N}; \overline{v}] \\ \forall i.\, \Gamma, \Delta \vdash e_i : \tau_i(\overline{v_i})\end{array}}{\Gamma, \Delta \vdash f^{\overline{v}}(e_1, \ldots, e_N) : \tau(\overline{v})}$$

Want: "*or* e_i = ? *and* ∃e such that premise holds"

...but this existential makes type-checking *very much not* syntax-directed. ☹

We *really* want to extend well-typing to non-ground terms…

$$\textsc{Well-Typed/Fun}\quad \frac{\begin{array}{c}\Gamma(f) = \tau_1, \ldots, \tau_N \to_\varphi \tau \\ \forall i, j.\, v_{ij} \in \mathrm{vals}(\Gamma) \cup \mathrm{vals}(\Delta) \cup \mathrm{vals}(\overline{v}) \\ \Delta \models \varphi[\overline{v_1}, \ldots, \overline{v_N}; \overline{v}] \\ \forall i.\, \Gamma, \Delta \vdash e_i : \tau_i(\overline{v_i})\end{array}}{\Gamma, \Delta \vdash f^{\overline{v}}(e_1, \ldots, e_N) : \tau(\overline{v})}$$

Want: "*or* e_i = ? *and* ∃e such that premise holds"

…but this existential makes type-checking *very much not* syntax-directed. ☹

- ***Key insight:***

We *really* want to extend well-typing to non-ground terms...

$$\textsc{Well-Typed/Fun}\quad \frac{\begin{array}{c}\Gamma(f) = \tau_1, \ldots, \tau_N \to_\varphi \tau \\ \forall i, j.\, v_{ij} \in \mathsf{vals}(\Gamma) \cup \mathsf{vals}(\Delta) \cup \mathsf{vals}(\overline{v}) \\ \Delta \models \varphi[\overline{v_1}, \ldots, \overline{v_N}; \overline{v}] \\ \forall i.\, \Gamma, \Delta \vdash e_i : \tau_i(\overline{v_i})\end{array}}{\Gamma, \Delta \vdash f^{\overline{v}}(e_1, \ldots, e_N) : \tau(\overline{v})}$$

Want: "*or* e_i = ? *and* ∃e such that premise holds"

...but this existential makes type-checking *very much not* syntax-directed. ☹

- ***Key insight:*** *We need a type inhabitation oracle.*

We *really* want to extend well-typing to non-ground terms…

$$\textsc{Well-Typed/Fun}\quad \frac{\Gamma(f) = \tau_1, \ldots, \tau_N \to_\varphi \tau \qquad \forall i, j.\, v_{ij} \in \mathsf{vals}(\Gamma) \cup \mathsf{vals}(\Delta) \cup \mathsf{vals}(\overline{v}) \qquad \Delta \models \varphi[\overline{v_1}, \ldots, \overline{v_N}; \overline{v}] \qquad \forall i.\, \Gamma, \Delta \vdash e_i : \tau_i(\overline{v_i})}{\Gamma, \Delta \vdash f^{\overline{v}}(e_1, \ldots, e_N) : \tau(\overline{v})}$$

Want: "*or* e_i = ? *and* ∃e such that premise holds"

…but this existential makes type-checking *very much not* syntax-directed. ☹

- ***Key insight:*** *We need a type inhabitation oracle.*
- Something that, when asked if a type is inhabited, responds "yes" or "no" (as in classical logic) without needing to say *what* that inhabitant is (as in constructive logic).

We can leverage the Curry-Howard correspondence to implement a type inhabitation oracle with fast proof engines like Datalog.

We can leverage the Curry-Howard correspondence to implement a type inhabitation oracle with fast proof engines like Datalog.

- Represent types as propositions (true if inhabited), expressions as proofs.

We can leverage the Curry-Howard correspondence to implement a type inhabitation oracle with fast proof engines like Datalog.

- Represent types as propositions (true if inhabited), expressions as proofs.
- From this perspective, oracle needs to determine if a proposition is true or false.
  Crucially: *We don't need to know a proof of the proposition, just its truth value.*

We can leverage the Curry-Howard correspondence to implement a type inhabitation oracle with fast proof engines like Datalog.

- Represent types as propositions (true if inhabited), expressions as proofs.
- From this perspective, oracle needs to determine if a proposition is true or false.
  Crucially: *We don't need to know a proof of the proposition, just its truth value.*
- Then, we can use a fast proof engine that doesn't give proofs…

We can leverage the Curry-Howard correspondence to implement a type inhabitation oracle with fast proof engines like Datalog.

- Represent types as propositions (true if inhabited), expressions as proofs.
- From this perspective, oracle needs to determine if a proposition is true or false.
  Crucially: *We don't need to know a proof of the proposition, just its truth value.*
- Then, we can use a fast proof engine that doesn't give proofs… like Datalog!

We can leverage the Curry-Howard correspondence to implement a type inhabitation oracle with fast proof engines like Datalog.

- Represent types as propositions (true if inhabited), expressions as proofs.
- From this perspective, oracle needs to determine if a proposition is true or false.
  Crucially: *We don't need to know a proof of the proposition, just its truth value.*
- Then, we can use a fast proof engine that doesn't give proofs… like Datalog!

$$f : R \to_{\varphi} R \;\text{ with }\; \varphi := \text{param}_{1,1} < \text{ret}_1$$

We can leverage the Curry-Howard correspondence to implement a type inhabitation oracle with fast proof engines like Datalog.

- Represent types as propositions (true if inhabited), expressions as proofs.
- From this perspective, oracle needs to determine if a proposition is true or false.
  Crucially: *We don't need to know a proof of the proposition, just its truth value.*
- Then, we can use a fast proof engine that doesn't give proofs… like Datalog!

$$f : R \to_{\varphi} R \;\text{ with }\; \varphi := \text{param}_{1,1} < \text{ret}_1 \quad \Rightarrow \quad \frac{R(x) \qquad x < y}{R(y)}\ \textsc{Rule}_f$$

We can leverage the Curry-Howard correspondence to implement a type inhabitation oracle with fast proof engines like Datalog.

- Represent types as propositions (true if inhabited), expressions as proofs.
- From this perspective, oracle needs to determine if a proposition is true or false.
  Crucially: *We don't need to know a proof of the proposition, just its truth value.*
- Then, we can use a fast proof engine that doesn't give proofs… like Datalog!

$$f : R \to_{\varphi} R \;\text{ with }\; \varphi := \text{param}_{1,1} < \text{ret}_1 \quad \Rightarrow \quad \frac{R(x) \qquad x < y}{R(y)}\ \textsc{Rule}_f$$

Key invariant. *For any function f, the following are equivalent:*

1. *Datalog proves* $\tau(\overline{v})$ *with a derivation tree ending in* $\textsc{Rule}_f$.
2. *There exist expressions* $e_1, \ldots, e_N$ *such that* $\Gamma, \Delta \vdash f^{\overline{v}}(e_1, \ldots, e_N) : \tau(\overline{v})$.

Question 1:

What are **Strong Completeness** and **Strong Soundness**?

Answer: Properties of step providers in Programming by Navigation (in particular, that they must show all and only the valid next steps).

Question 2:

How do we achieve Strong Completeness and Strong Soundness?

Question 1:

What are **Strong Completeness** and **Strong Soundness**?

Answer: Properties of step providers in Programming by Navigation (in particular, that they must show all and only the valid next steps).

Question 2:

How do we achieve Strong Completeness and Strong Soundness?

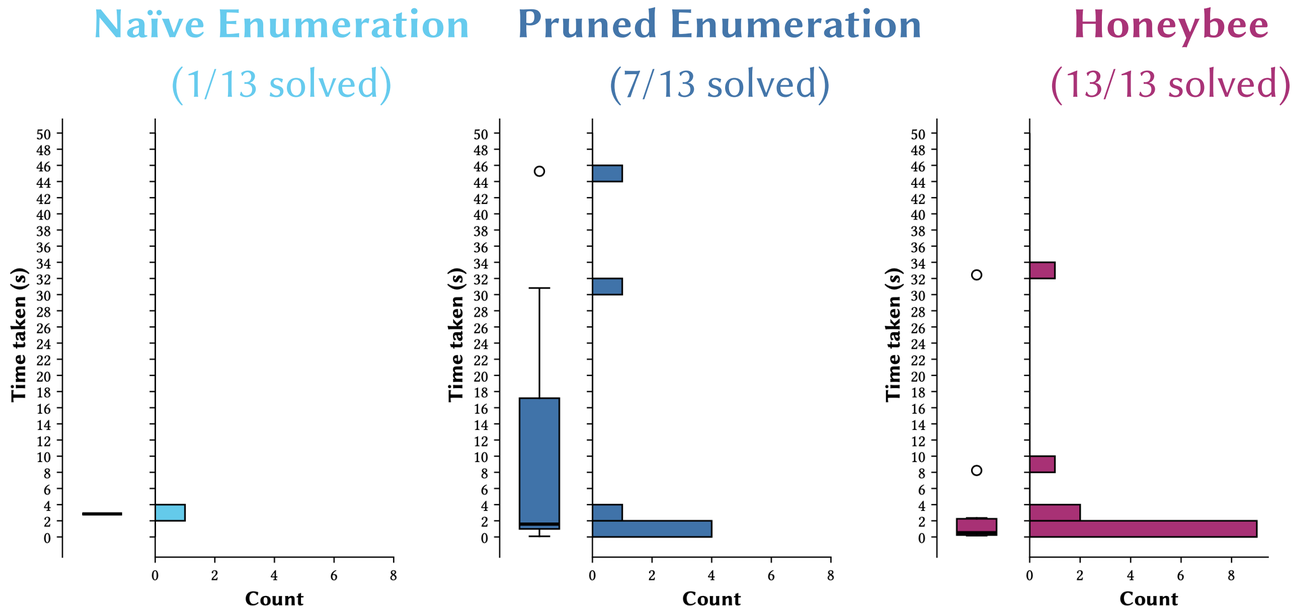
Answer: Use a classical-style inhabitation oracle. Leverage the Curry-Howard correspondence to implement it with Datalog.

Wrap-up

Our Programming by Navigation synthesizer, HONEYBEE, solves tasks that are impossible or too large for baselines to solve.

Our Programming by Navigation synthesizer, HONEYBEE, solves tasks that are impossible or too large for baselines to solve.

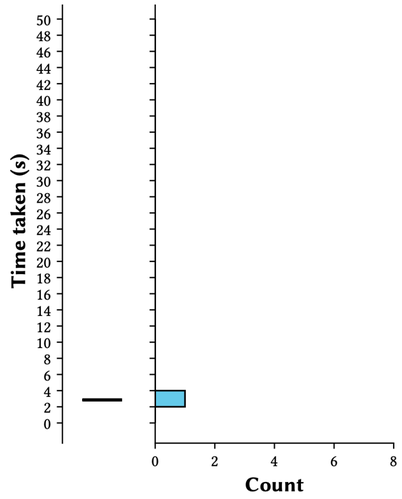
Benchmarks with ***finitely*** many solutions.



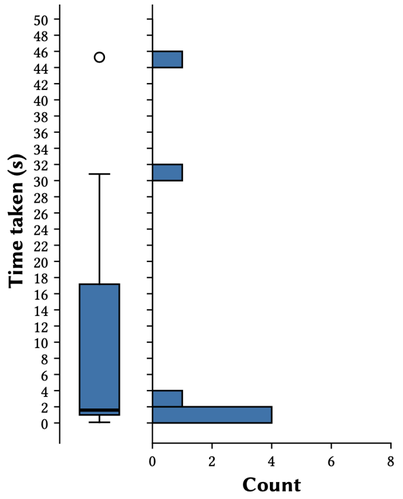
Our Programming by Navigation synthesizer, HONEYBEE, solves tasks that are impossible or too large for baselines to solve.

Benchmarks with ***finitely*** many solutions.

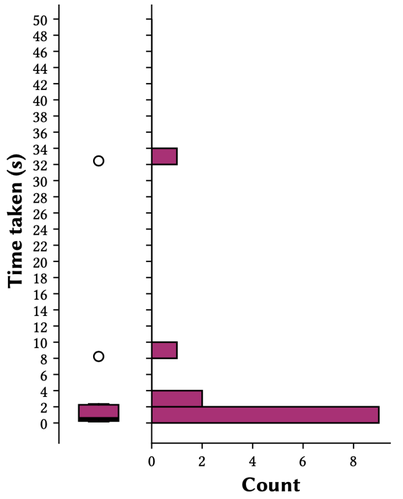
Naïve Enumeration (1/13 solved)



Pruned Enumeration (7/13 solved)



Honeybee (13/13 solved)

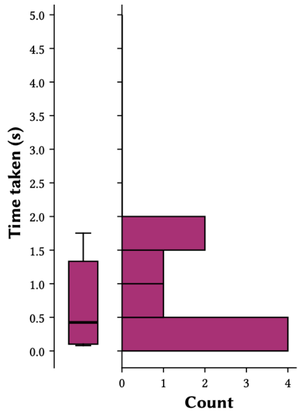


Benchmarks with ***infinitely*** many solutions.

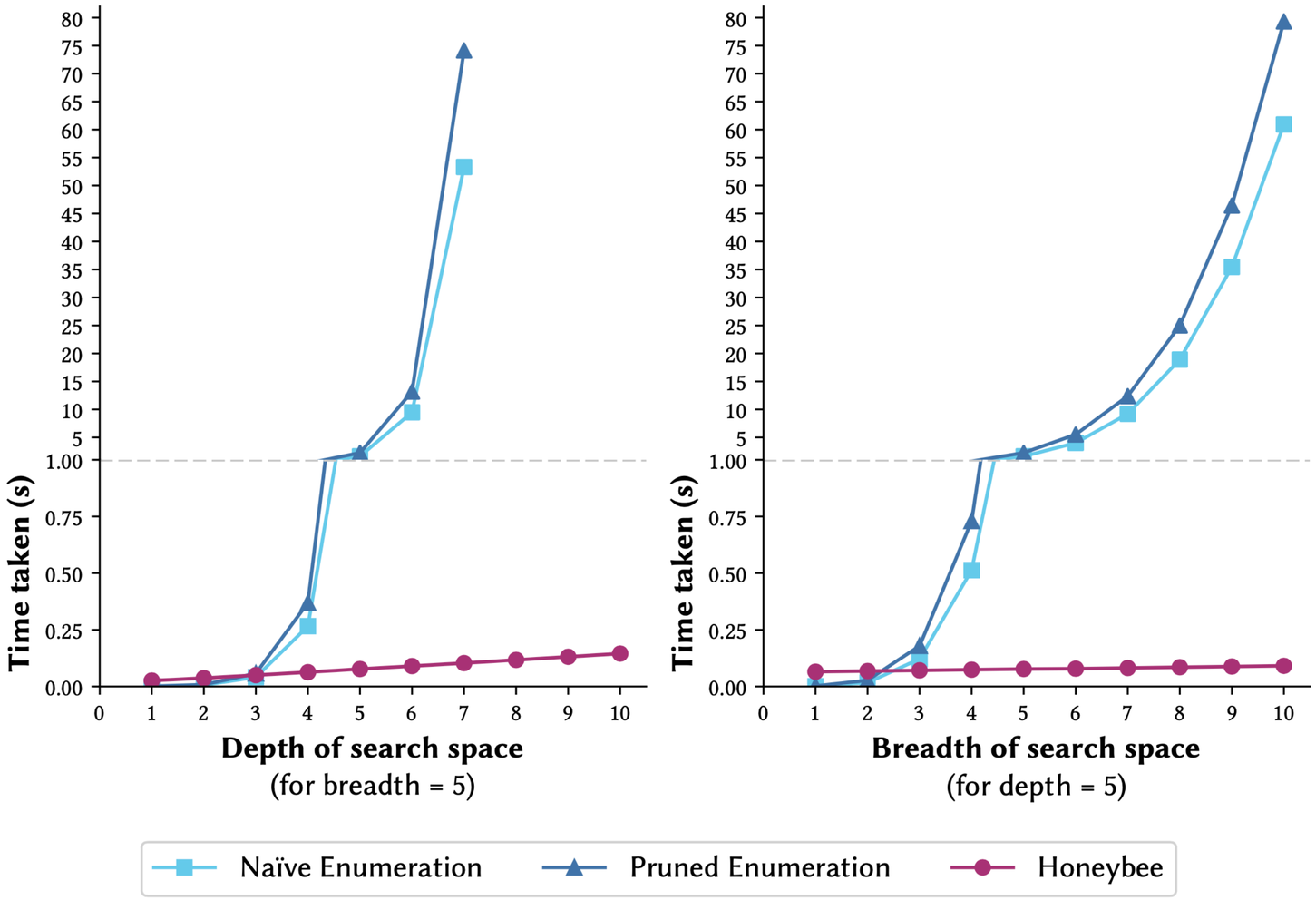
not possible

not possible

(8/8 solved)

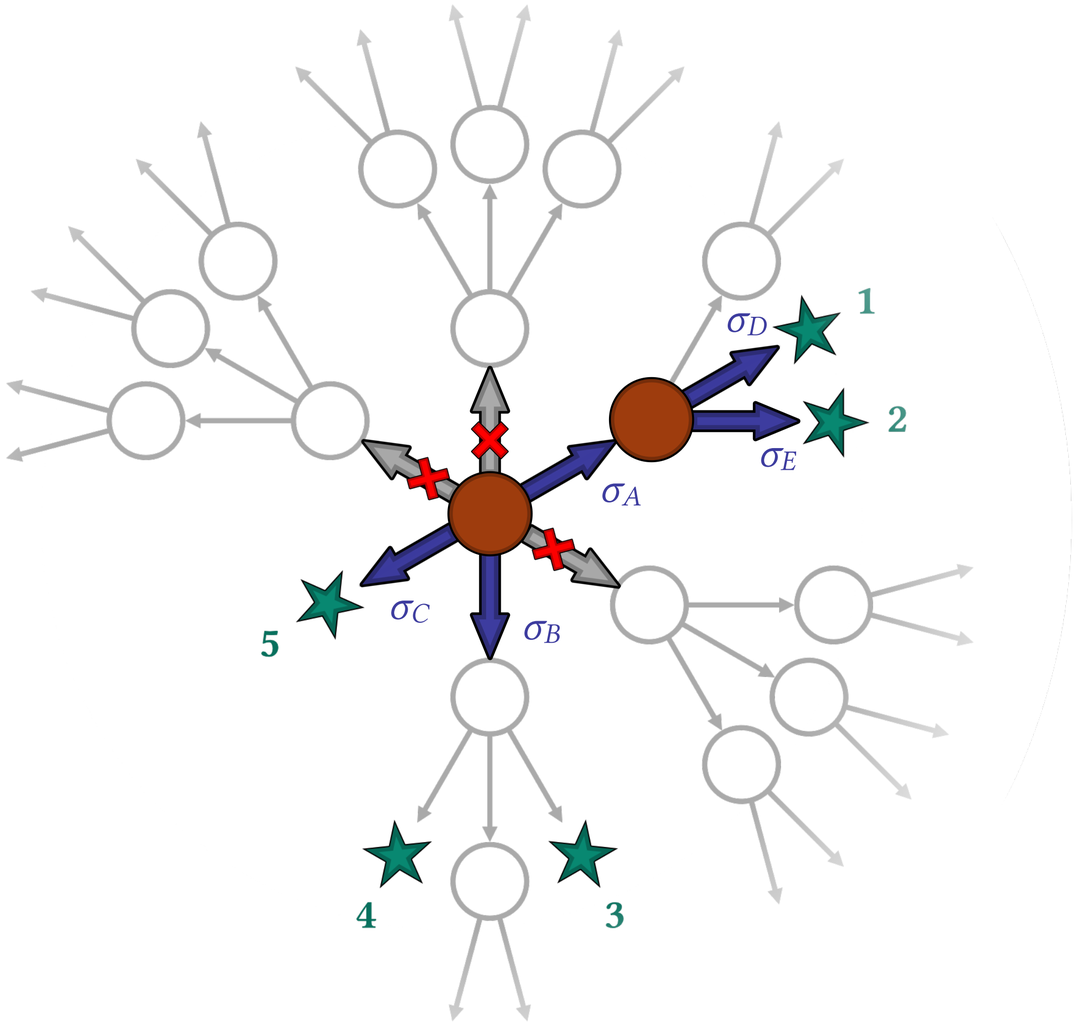


Our Programming by Navigation synthesizer, HONEYBEE, solves tasks that are impossible or too large for baselines to solve.

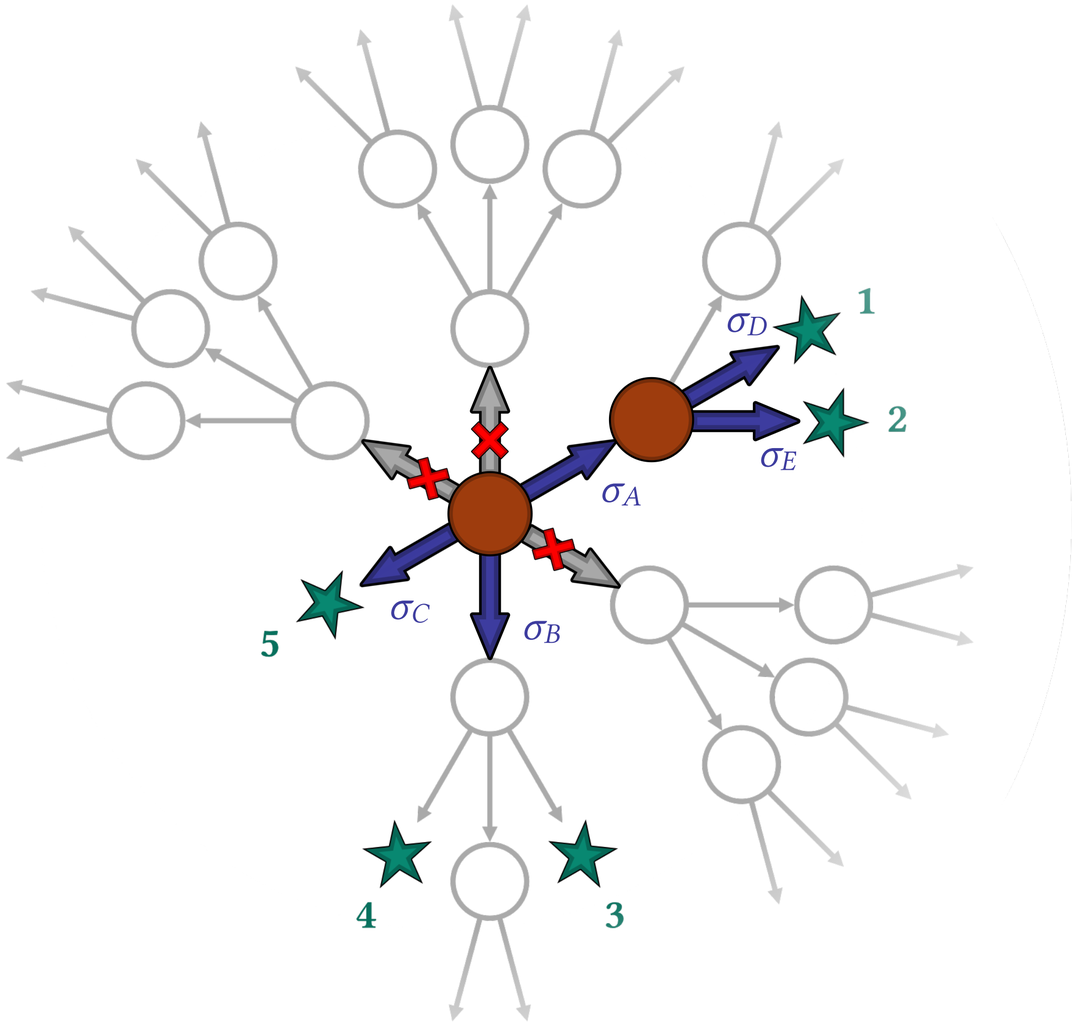


Programming by Navigation

Programming by Navigation

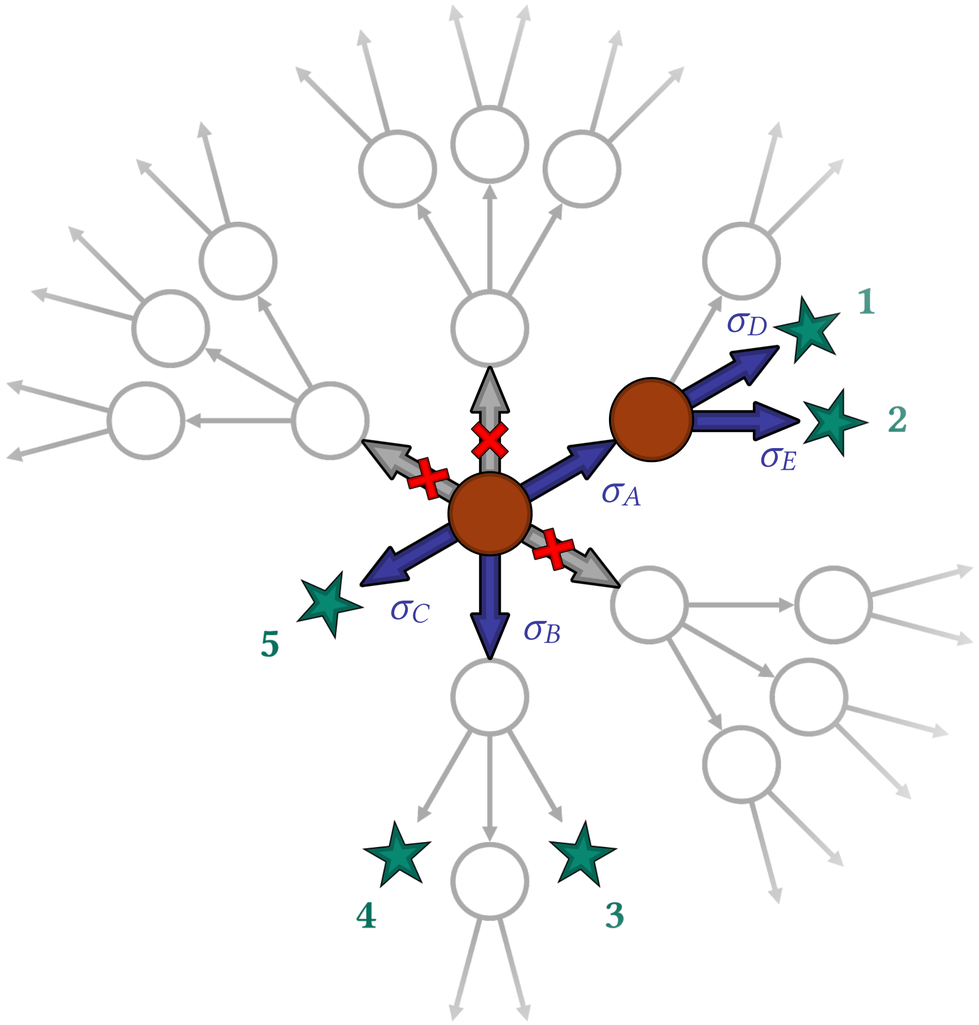


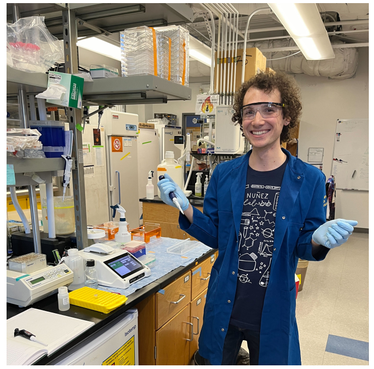
Programming by Navigation



- **Strong Completeness** and **Strong Soundness**:
  Nice goals, even when (provably) not fully possible…
  … but can work even for messy settings!

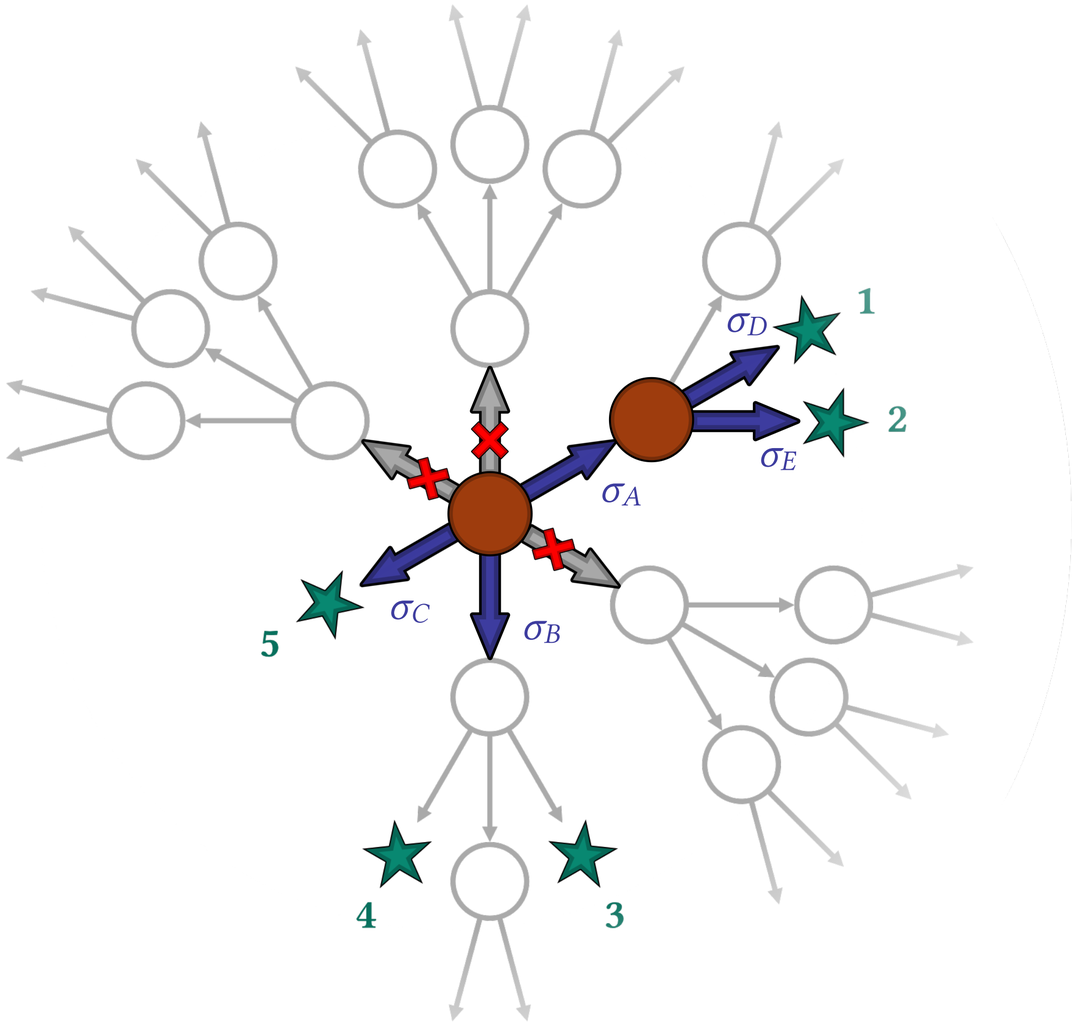
Programming by Navigation

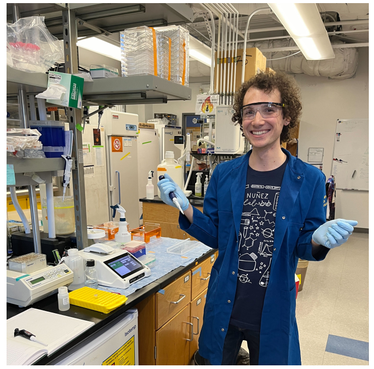




- **Strong Completeness** and **Strong Soundness**:
  Nice goals, even when (provably) not fully possible…
  … but can work even for messy settings!

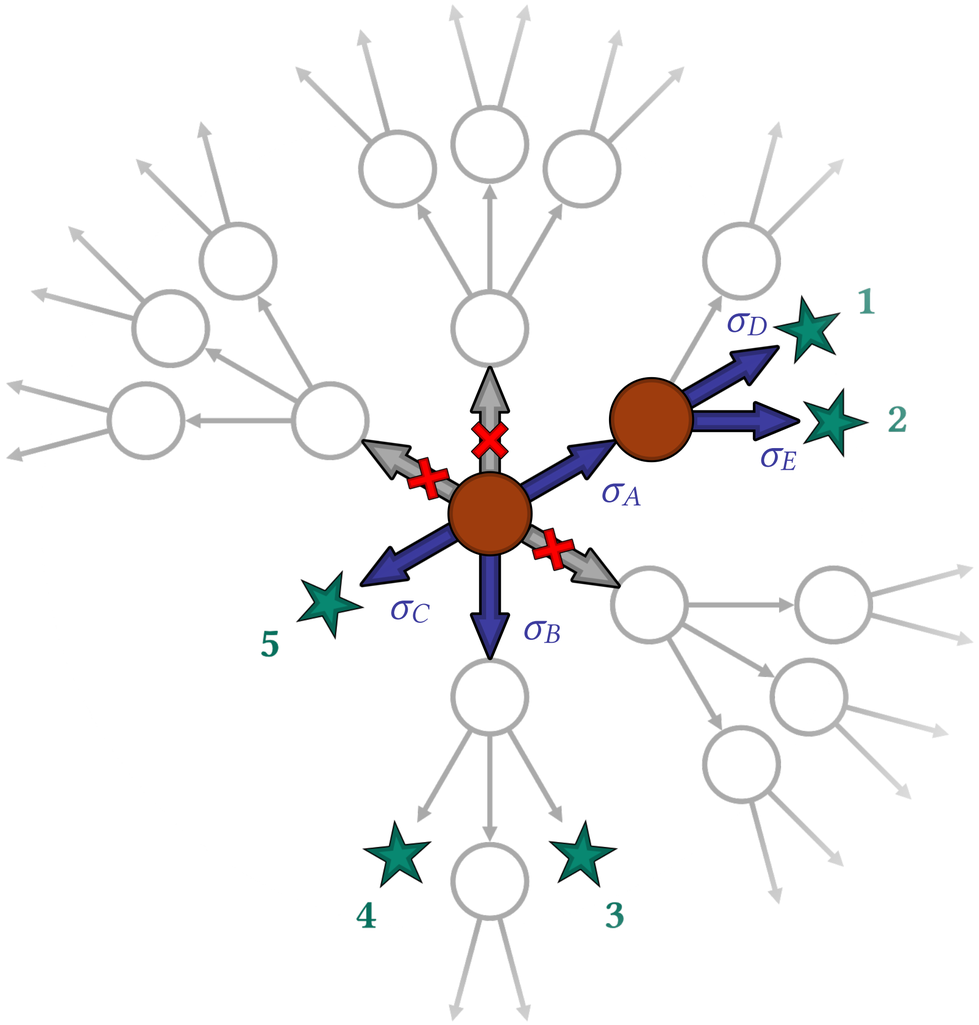
Programming by Navigation

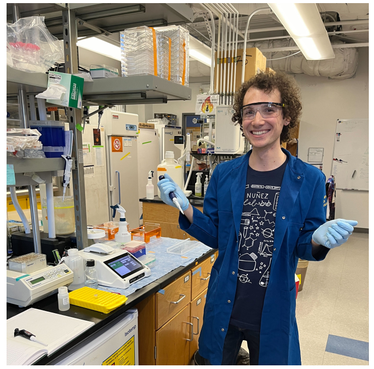




- **STRONG COMPLETENESS** and **STRONG SOUNDNESS**:
  Nice goals, even when (provably) not fully possible…
  … but can work even for messy settings!
- **Classical-style inhabitation oracles for synthesis:**
  Very powerful, if you can get away with it!

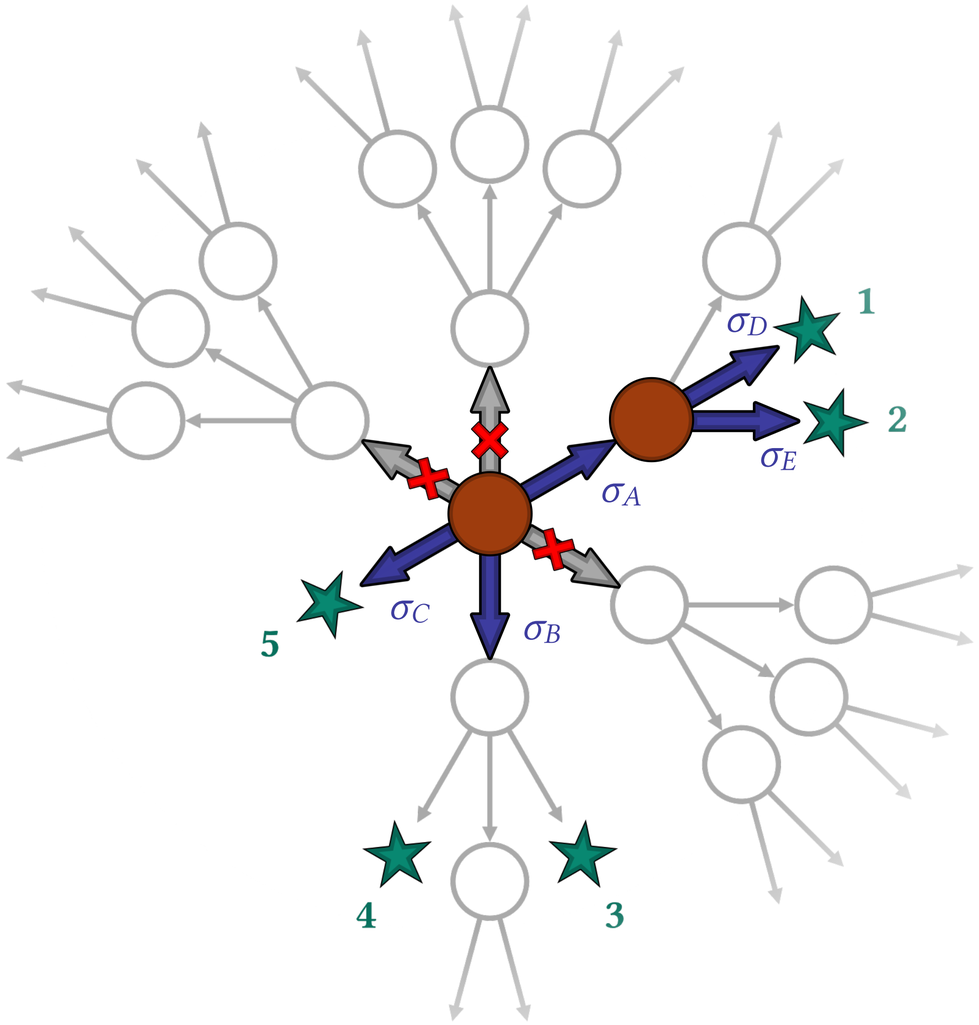
Programming by Navigation

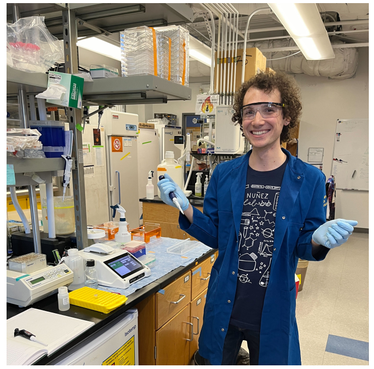




- **Strong Completeness** and **Strong Soundness**:
  Nice goals, even when (provably) not fully possible…
  … but can work even for messy settings!
- **Classical-style inhabitation oracles for synthesis:**
  Very powerful, if you can get away with it!
- ***I would love to collaborate!*** ☺

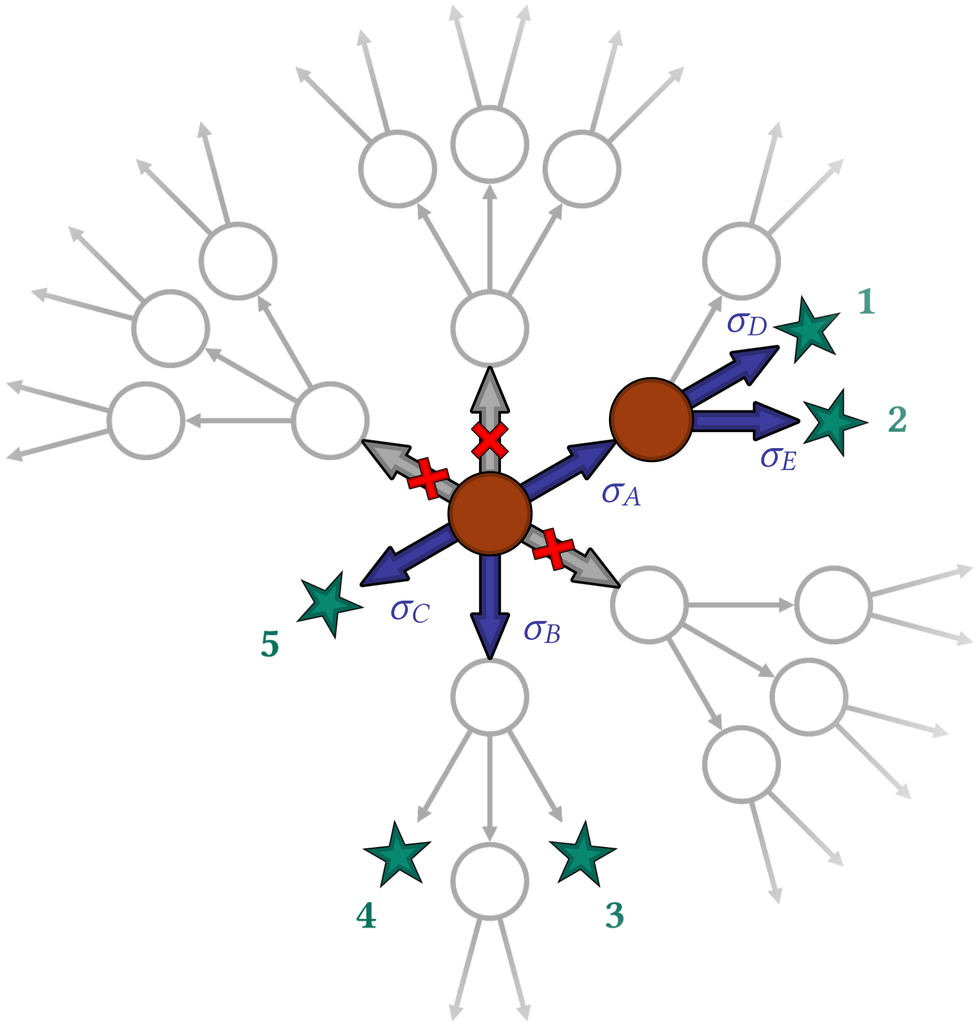
Programming by Navigation

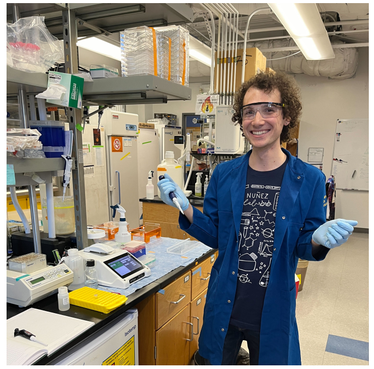




- **STRONG COMPLETENESS** and **STRONG SOUNDNESS**:
  Nice goals, even when (provably) not fully possible…
  … but can work even for messy settings!
- **Classical-style inhabitation oracles for synthesis:**
  Very powerful, if you can get away with it!
- ***I would love to collaborate!*** ☺
  - *Connections we've observed:* structure editors, theorem provers, rewrite systems + e-graphs, …

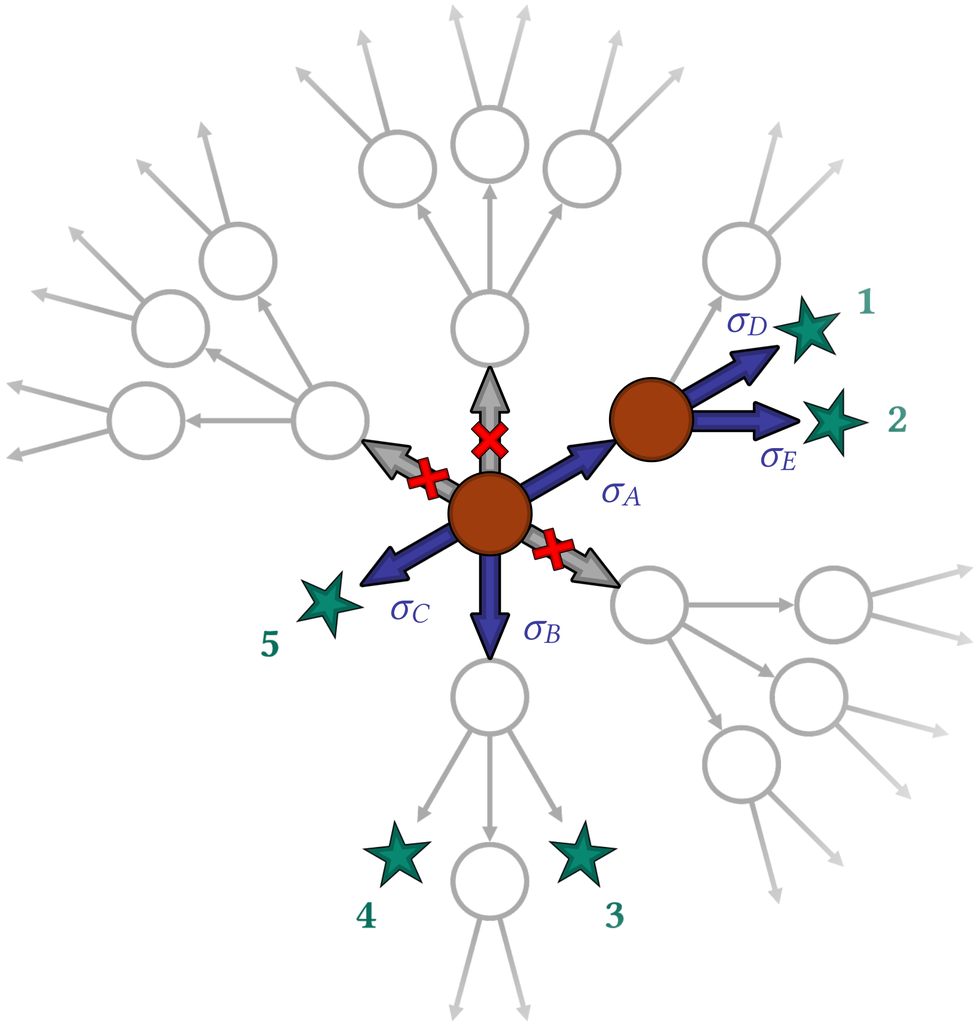
Programming by Navigation

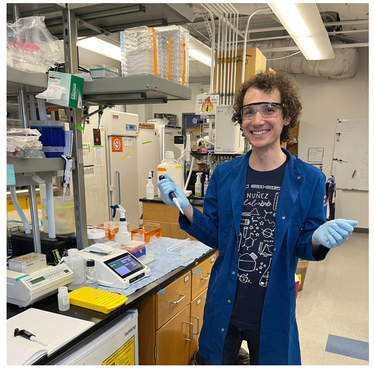




- **STRONG COMPLETENESS** and **STRONG SOUNDNESS**:
  Nice goals, even when (provably) not fully possible…
  … but can work even for messy settings!
- **Classical-style inhabitation oracles for synthesis:**
  Very powerful, if you can get away with it!
- ***I would love to collaborate!*** ☺
  - *Connections we've observed:* structure editors, theorem provers, rewrite systems + e-graphs, …
  - HONEYBEE (and many other PL tools) are based on constraint systems… I'm also very interested in developing new PL theory to make them more usable.

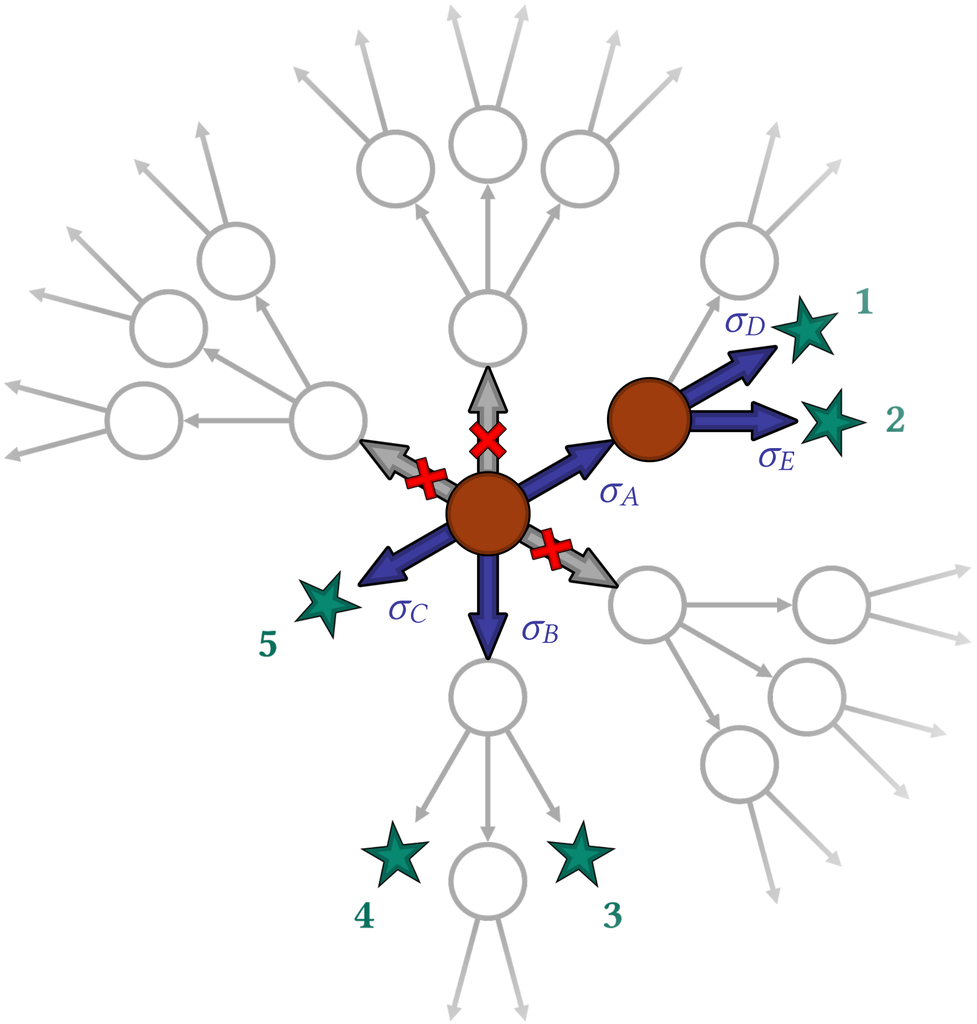
Programming by Navigation

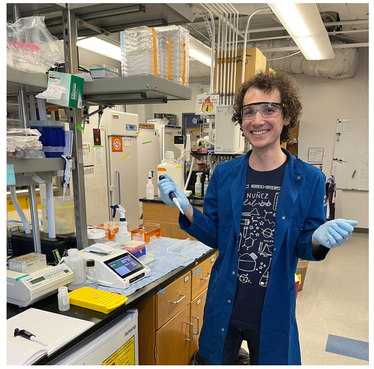




- **STRONG COMPLETENESS** and **STRONG SOUNDNESS**:
  Nice goals, even when (provably) not fully possible…
  … but can work even for messy settings!
- **Classical-style inhabitation oracles for synthesis:**
  Very powerful, if you can get away with it!
- ***I would love to collaborate!*** ☺
  - *Connections we've observed:* structure editors, theorem provers, rewrite systems + e-graphs, …
  - HONEYBEE (and many other PL tools) are based on constraint systems… I'm also very interested in developing new PL theory to make them more usable.

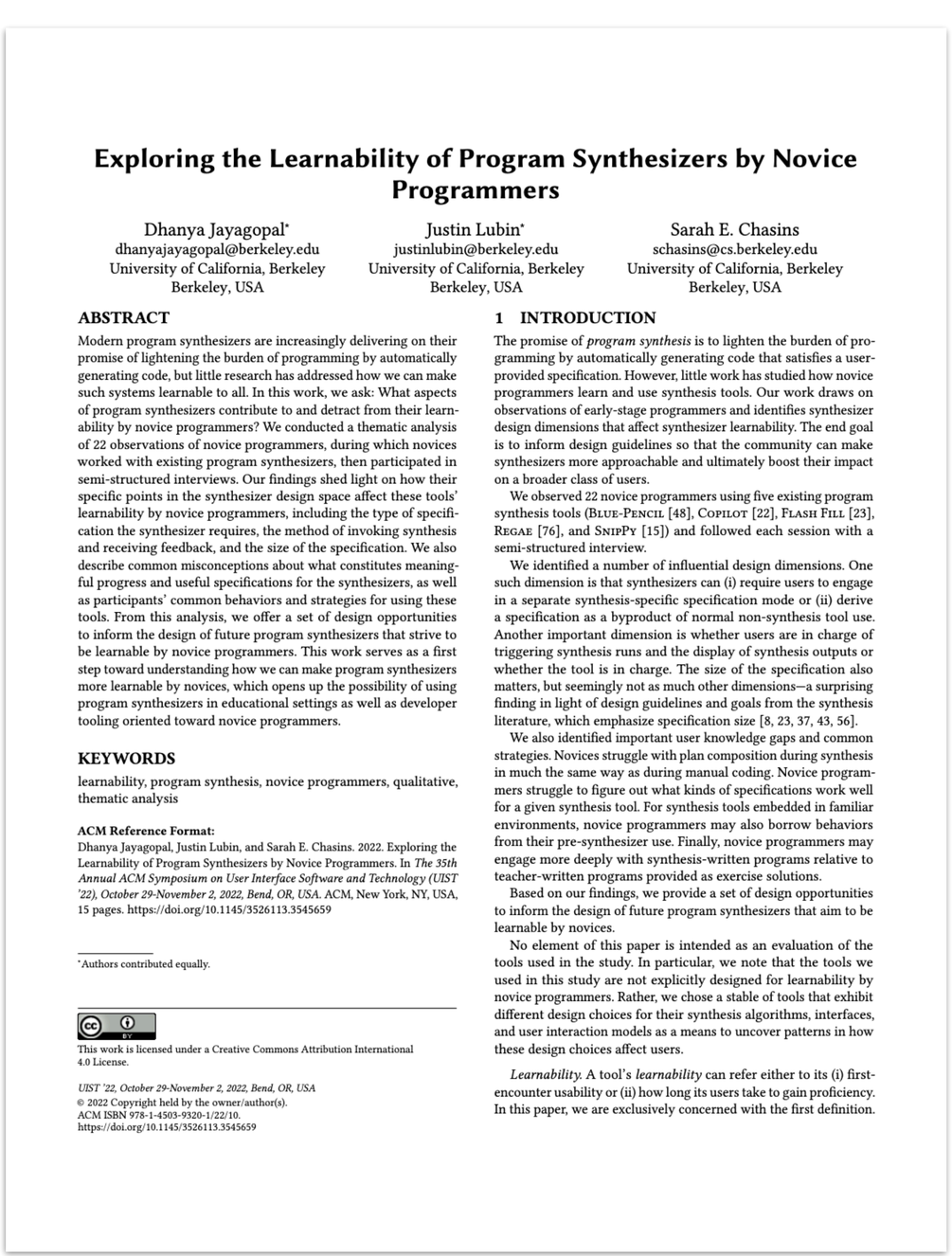
Programming by Navigation





- **STRONG COMPLETENESS** and **STRONG SOUNDNESS**:
  Nice goals, even when (provably) not fully possible…
  … but can work even for messy settings!
- **Classical-style inhabitation oracles for synthesis:**
  Very powerful, if you can get away with it!
- ***I would love to collaborate!*** ☺
  - *Connections we've observed:* structure editors, theorem provers, rewrite systems + e-graphs, …
  - HONEYBEE (and many other PL tools) are based on constraint systems… I'm also very interested in developing new PL theory to make them more usable.

Extra Slides



Exploring the Learnability of Program Synthesizers by Novice Programmers

Dhanya Jayagopal*
dhanyajayagopal@berkeley.edu
University of California, Berkeley
Berkeley, USA

Justin Lubin*
justinlubin@berkeley.edu
University of California, Berkeley
Berkeley, USA

Sarah E. Chasins
schasins@cs.berkeley.edu
University of California, Berkeley
Berkeley, USA

ABSTRACT

Modern program synthesizers are increasingly delivering on their promise of lightening the burden of programming by automatically generating code, but little research has addressed how we can make such systems learnable to all. In this work, we ask: What aspects of program synthesizers contribute to and detract from their learnability by novice programmers? We conducted a thematic analysis of 22 observations of novice programmers, during which novices worked with existing program synthesizers, then participated in semi-structured interviews. Our findings shed light on how their specific points in the synthesizer design space affect these tools' learnability by novice programmers, including the type of specification the synthesizer requires, the method of invoking synthesis and receiving feedback, and the size of the specification. We also describe common misconceptions about what constitutes meaningful progress and useful specifications for the synthesizers, as well as participants' common behaviors and strategies for using these tools. From this analysis, we offer a set of design opportunities to inform the design of future program synthesizers that strive to be learnable by novice programmers. This work serves as a first step toward understanding how we can make program synthesizers more learnable by novices, which opens up the possibility of using program synthesizers in educational settings as well as developer tooling oriented toward novice programmers.

KEYWORDS

learnability, program synthesis, novice programmers, qualitative, thematic analysis

ACM Reference Format:
Dhanya Jayagopal, Justin Lubin, and Sarah E. Chasins. 2022. Exploring the Learnability of Program Synthesizers by Novice Programmers. In *The 35th Annual ACM Symposium on User Interface Software and Technology (UIST '22), October 29-November 2, 2022, Bend, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3526113.3545659

*Authors contributed equally.

This work is licensed under a Creative Commons Attribution International 4.0 License.

UIST '22, October 29-November 2, 2022, Bend, OR, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9320-1/22/10.
https://doi.org/10.1145/3526113.3545659

1 INTRODUCTION

The promise of *program synthesis* is to lighten the burden of programming by automatically generating code that satisfies a user-provided specification. However, little work has studied how novice programmers learn and use synthesis tools. Our work draws on observations of early-stage programmers and identifies synthesizer design dimensions that affect synthesizer learnability. The end goal is to inform design guidelines so that the community can make synthesizers more approachable and ultimately boost their impact on a broader class of users.

We observed 22 novice programmers using five existing program synthesis tools (Blue-Pencil [48], Copilot [22], Flash Fill [23], Regae [76], and SnipPy [15]) and followed each session with a semi-structured interview.

We identified a number of influential design dimensions. One such dimension is that synthesizers can (i) require users to engage in a separate synthesis-specific specification mode or (ii) derive a specification as a byproduct of normal non-synthesis tool use. Another important dimension is whether users are in charge of triggering synthesis runs and the display of synthesis outputs or whether the tool is in charge. The size of the specification also matters, but seemingly not as much other dimensions—a surprising finding in light of design guidelines and goals from the synthesis literature, which emphasize specification size [8, 23, 37, 43, 56].

We also identified important user knowledge gaps and common strategies. Novices struggle with plan composition during synthesis in much the same way as during manual coding. Novice programmers struggle to figure out what kinds of specifications work well for a given synthesis tool. For synthesis tools embedded in familiar environments, novice programmers may also borrow behaviors from their pre-synthesizer use. Finally, novice programmers may engage more deeply with synthesis-written programs relative to teacher-written programs provided as exercise solutions.

Based on our findings, we provide a set of design opportunities to inform the design of future program synthesizers that aim to be learnable by novices.

No element of this paper is intended as an evaluation of the tools used in the study. In particular, we note that the tools we used in this study are not explicitly designed for learnability by novice programmers. Rather, we chose a stable of tools that exhibit different design choices for their synthesis algorithms, interfaces, and user interaction models as a means to uncover patterns in how these design choices affect users.

Learnability. A tool's *learnability* can refer either to its (i) first-encounter usability or (ii) how long its users take to gain proficiency. In this paper, we are exclusively concerned with the first definition.

Exploring the Learnability of Program Synthesizers by Novice Programmers.

Dhanya Jayagopal,* Justin Lubin,* Sarah E. Chasins.

In *UIST 2022.* (* equal contribution)

Existing interactive synthesis guarantees require well-behaved users.

Interactive Program Synthesis

Vu Le
Microsoft Corporation
levu@microsoft.com

Daniel Perelman
Microsoft Corporation
danpere@microsoft.com

Oleksandr Polozov
University of Washington
polozov@cs.washington.edu

Mohammad Raza
Microsoft Corporation
moraza@microsoft.com

Abhishek Udupa
Microsoft Corporation
abudup@microsoft.com

Sumit Gulwani
Microsoft Corporation
sumitg@microsoft.com

Let φ^* be a spec on the output symbol of $\mathcal{L}$, called a *task spec*. A **φ^*-driven interactive program synthesis process** is a finite series of 4-tuples $\langle N_0, \varphi_0, \widetilde{N}_0, \Sigma_0 \rangle, \ldots, \langle N_m, \varphi_m, \widetilde{N}_m, \Sigma_m \rangle$, where

- Each N_i is a nonterminal in $\mathcal{L}$,
- Each φ_i is a spec on N_i,
- Each $\widetilde{N}_i$ is some set of programs rooted at N_i s.t. $\widetilde{N}_i \models \varphi_i$,
- Each Σ_i is an **interaction state**, explained below,

which satisfies the following axioms for any program $P \in \mathcal{L}$:

A. $(P \models \varphi^*) \Rightarrow (P \models \varphi_i)$ for any $0 \leq i \leq m$;
B. $(P \models \varphi_j) \Rightarrow (P \models \varphi_i)$ for any $0 \leq i < j \leq m$ s.t. $N_i = N_j$.

We say that the process is **converging** iff the top-ranked program of the last program set in the process satisfies the task spec:

$$P^* = \mathsf{Top}_h(\widetilde{N}_m, 1) \models \varphi^*$$

Existing interactive synthesis guarantees require well-behaved users.

User correctness requirements

Interactive Program Synthesis

Vu Le
Microsoft Corporation
levu@microsoft.com

Daniel Perelman
Microsoft Corporation
danpere@microsoft.com

Oleksandr Polozov
University of Washington
polozov@cs.washington.edu

Mohammad Raza
Microsoft Corporation
moraza@microsoft.com

Abhishek Udupa
Microsoft Corporation
abudup@microsoft.com

Sumit Gulwani
Microsoft Corporation
sumitg@microsoft.com

Let φ^* be a spec on the output symbol of $\mathcal{L}$, called a *task spec*. A **φ^*-driven interactive program synthesis process** is a finite series of 4-tuples $\langle N_0, \varphi_0, \widetilde{N}_0, \Sigma_0\rangle, \ldots, \langle N_m, \varphi_m, \widetilde{N}_m, \Sigma_m\rangle$, where

- Each N_i is a nonterminal in $\mathcal{L}$,
- Each φ_i is a spec on N_i,
- Each $\widetilde{N}_i$ is some set of programs rooted at N_i s.t. $\widetilde{N}_i \models \varphi_i$,
- Each Σ_i is an **interaction state**, explained below,

which satisfies the following axioms for any program $P \in \mathcal{L}$:

A. $(P \models \varphi^*) \Rightarrow (P \models \varphi_i)$ for any $0 \leq i \leq m$;
B. $(P \models \varphi_j) \Rightarrow (P \models \varphi_i)$ for any $0 \leq i < j \leq m$ s.t. $N_i = N_j$.

We say that the process is **converging** iff the top-ranked program of the last program set in the process satisfies the task spec:

$$P^* = \mathsf{Top}_h(\widetilde{N}_m, 1) \models \varphi^*$$

Existing interactive synthesis guarantees require well-behaved users.

User correctness requirements

Convergence guarantee

Interactive Program Synthesis

Vu Le
Microsoft Corporation
levu@microsoft.com

Daniel Perelman
Microsoft Corporation
danpere@microsoft.com

Oleksandr Polozov
University of Washington
polozov@cs.washington.edu

Mohammad Raza
Microsoft Corporation
moraza@microsoft.com

Abhishek Udupa
Microsoft Corporation
abudup@microsoft.com

Sumit Gulwani
Microsoft Corporation
sumitg@microsoft.com

Let φ^* be a spec on the output symbol of $\mathcal{L}$, called a *task spec*. A **φ^*-driven interactive program synthesis process** is a finite series of 4-tuples $\langle N_0, \varphi_0, \widetilde{N}_0, \Sigma_0\rangle, \ldots, \langle N_m, \varphi_m, \widetilde{N}_m, \Sigma_m\rangle$, where

- Each N_i is a nonterminal in $\mathcal{L}$,
- Each φ_i is a spec on N_i,
- Each $\widetilde{N}_i$ is some set of programs rooted at N_i s.t. $\widetilde{N}_i \models \varphi_i$,
- Each Σ_i is an **interaction state**, explained below,

which satisfies the following axioms for any program $P \in \mathcal{L}$:

A. $(P \models \varphi^*) \Rightarrow (P \models \varphi_i)$ for any $0 \leq i \leq m$;
B. $(P \models \varphi_j) \Rightarrow (P \models \varphi_i)$ for any $0 \leq i < j \leq m$ s.t. $N_i = N_j$.

We say that the process is **converging** iff the top-ranked program of the last program set in the process satisfies the task spec:

$$P^* = \mathsf{Top}_h(\widetilde{N}_m, 1) \models \varphi^*$$

Jha *et al.* (2010). **Oracle-Guided Component-Based Program Synthesis.** In *ICSE*.

3. PROBLEM DEFINITION

The goal is to synthesize a loop-free program using a given set of base components and using input-ouput examples. We assume the presence of an I/O oracle that can be queried on any input. The I/O oracle, when given an input, returns the output of the desired program (that we wish to synthesize) on that input. We also assume the presence of a

```
IterativeSynthesis():
1  // Input:  Set of base components used in
2  // construction of Behave_E and Distinct_{E,L}
3  // Output:  Candidate Program
4  E := {(α0, I(α0))} // Pick any value α0 for I⃗
5  while (1) {
6    L := T-SAT(Behave_E(L));
7    if (L == ⊥) return "Components insufficient";
8    α := T-SAT(Distinct_{E,L}(I⃗));
9    if (α == ⊥) {
10       P := Lval2Prog(L);
11       if (V(P)) return P;
12       else return "Components insufficient"; }
13   E := E ∪ {α, I(α)}; }
```

Figure 3: Oracle-guided Synthesis Procedure

Blinn *et al.* (2022). **An Integrative Human-Centered Architecture for Interactive Programming Assistants.** In *VL/HCC*.

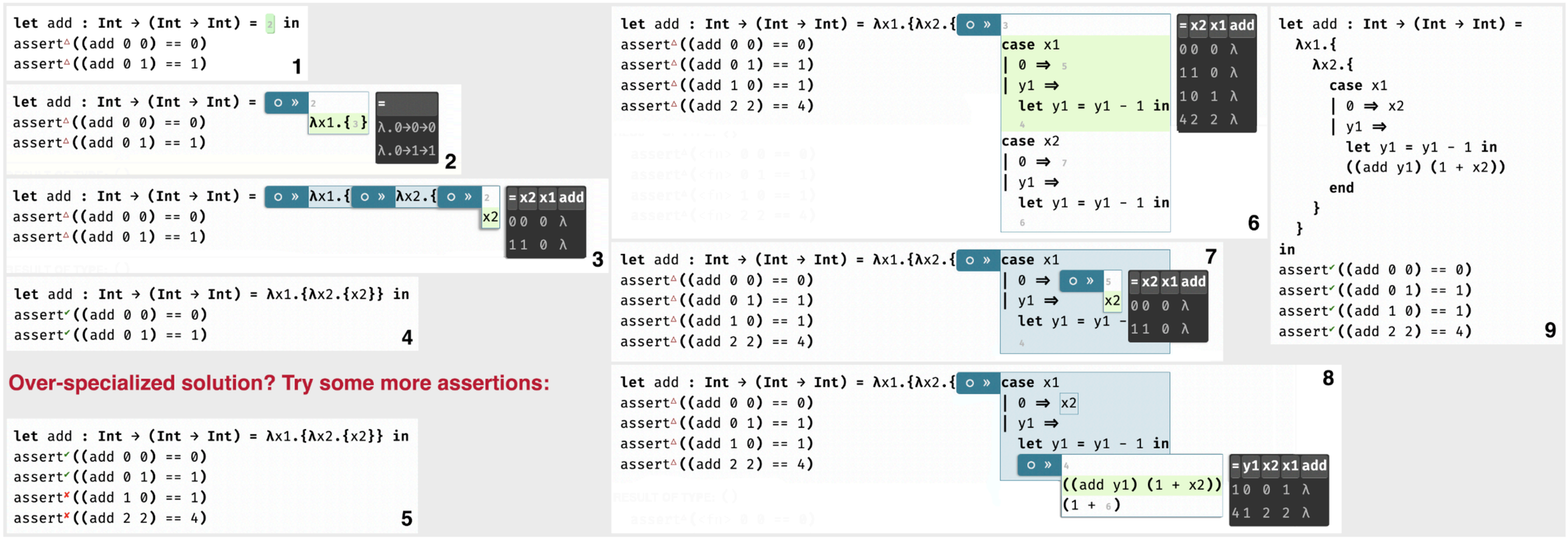


Fig. 3. Hazel Live Assistant: Here we collaborate with the Smyth synthesizer to write a function to add Peano-representation integers. Here we are working around the fact that the Smyth synthesizer supports only algebraic data types, which are not supported by Hazel; we translate the successor constructor to "+ 1" and destructure a successor by subtracting 1. (3.1) portrays a stubbed-out function with two user-provided examples. (3.2-3.4) show the process of *stepping through* a synthesis refinement tree: The user is offered a menu of options; at these stages there is only one suggested completion. The black panel displays the unevaluated constraints which must be satisfied. (3.5) shows a finished but overspecialized solution; the user resolves this by stepping out of synthesis and adding two more examples. (3.6-3.8) represent the result of deleting the "x2" reference and resuming synthesis. This time the user has more options; either casing on x1 or x2 (3.6), and adding 1 before or after the recursive call (3.8). (3.9) shows the completed function

Mayer *et al.* (2015). **User Interaction Models for Disambiguation in Programming by Example.** In *UIST*.

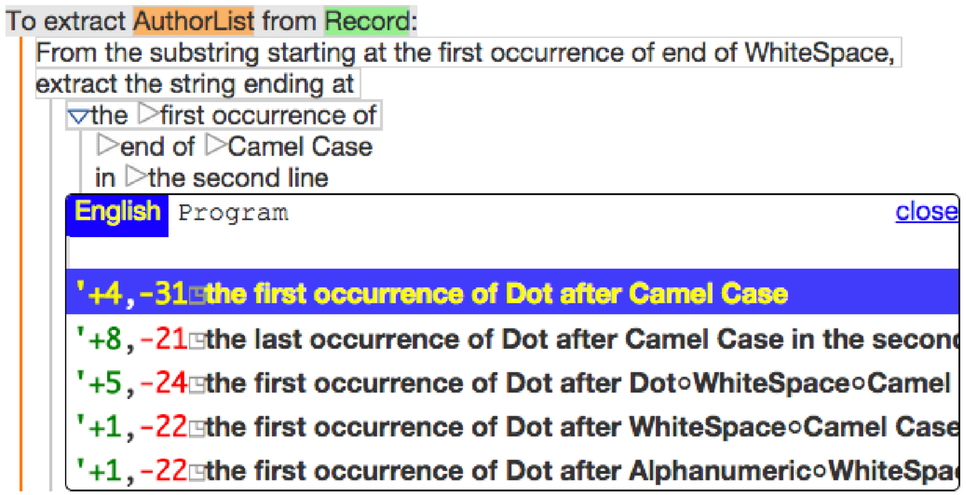


Figure 6: Program Viewer tab & alternative subexpressions.

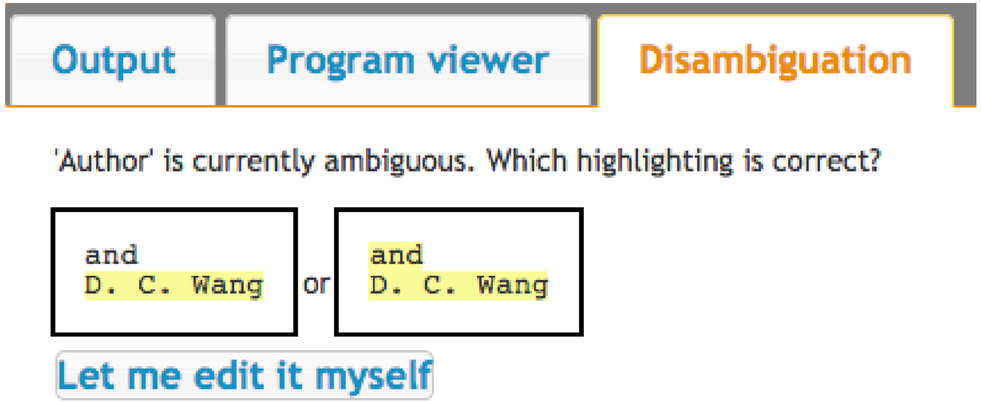
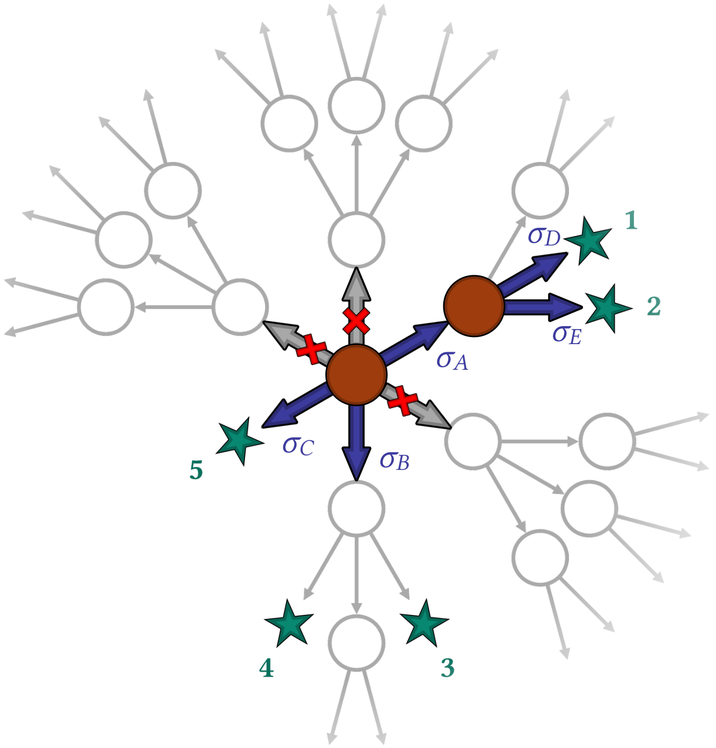


Figure 7: Conversational Clarification being used to disambiguate different programs that extract individual authors.

Extended Slides

Programming by Navigation can be thought of as a "semantic" structure editor.

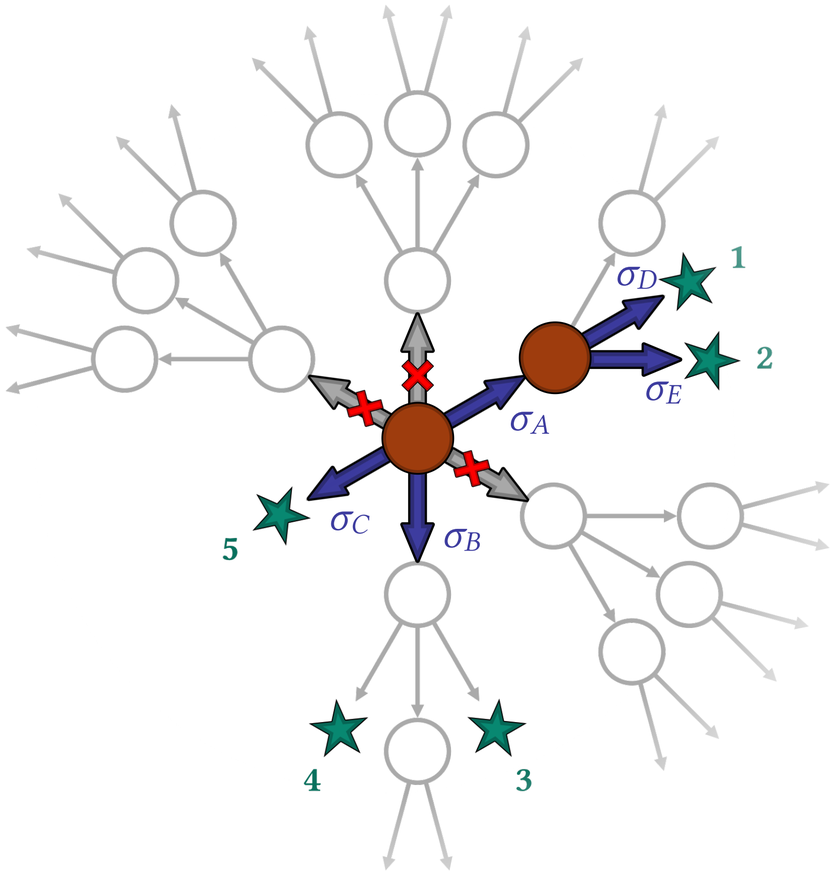


- Programming by Navigation synthesizers satisfy the following theorem:
- ***Constructability.*** *If e valid, then there exists an* $\mathbb{S}$*-interaction*

$$e_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_N} e$$

- Analogous to Omar *et al.* (2017)'s constructability theorem for the Hazelnut structure editor calculus.
- Structure editors prevent steps that are **syntactically** invalid.
- Programming by Navigation prevents steps that are **semantically** invalid (*i.e.*, won't lead to a valid solution).

Steps are the building blocks for Strong Completeness and Strong Soundness.



- Validity: e valid
- Steps: $e_1 \xrightarrow{\sigma} e_2$ (with induced relation $e_1 < e_2$)
- Requirements for steps:
  - **Determinism.** There is at most one e' for each e and σ such that $e \xrightarrow{\sigma} e'$.
  - **No loops.** $<$ is a strict partial order.
  - **Reachability.** There exists a lower bound e_{start} on the set of valid expressions (a "blank program").
  - **Finite Between.** Every infinite ascending chain $e_0 < e_1 < \cdots$ is unbounded.

Programming by Navigation Synthesizers have some nice properties for free!

- ***Fail Fast.***

  If there are no valid expressions, then $\mathbb{S}(e_{\text{start}}) = \emptyset$.

- ***Progress*** (analogous to traditional progress theorem for λ-calculus).

  If there is at least one valid expression, and $e_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_N} e_N$ *is an* $\mathbb{S}$*-interaction, then either* e_N valid *or* $\mathbb{S}(e_{\text{start}}) \neq \emptyset$.

- ***Constructability*** (analogous to Omar et al. 2017's constructability theorem for the Hazelnut structure editor calculus)**.**

  If e valid*, then there exists an* $\mathbb{S}$*-interaction* $e_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_N} e$.

- Structure editors prevent steps that are **syntactically** invalid.
- Programming by Navigation prevents steps that are **semantically** invalid (*i.e.*, won't lead to a valid solution).

Syntax and semantics for top-down steps.

Expressions $e ::= f(e_1, \ldots, e_N)$ **Steps** $\sigma ::= ?_h \rightsquigarrow f(e_1, \ldots, e_N)$

$\boxed{e_1 \xrightarrow{\sigma} e_2}$ σ top-down steps e_1 to e_2

STEP/EXTEND

$$\frac{\text{arity}(f) = N \qquad ?_h \triangleleft e}{e \xrightarrow{?_h \rightsquigarrow f(e_1,\ldots,e_N)} [?_h \mapsto f(e_1, \ldots, e_N)]e}$$

STEP/SEQ

$$\frac{e \xrightarrow{\sigma_1} e' \qquad e' \xrightarrow{\sigma_2} e''}{e \xrightarrow{\sigma_1 ; \sigma_2} e''}$$

An example Programing by Navigation interaction for top-down steps.

How?

Synthesizer (step provider) needs to do this.

Goal type: $D\langle 1,2\rangle$

All and only valid next steps

1

Working sketch: $?_1$

Goal: $?_1 : D$

$$\left\{ \bullet \;\; ?_1 \mapsto d^{1,2}(?_2) \right.$$

2

Working sketch: $d^{1,2}(?_2)$

Goal: $?_2 : M$

$$\left\{ \begin{array}{l} \bullet \;\; ?_2 \mapsto c^{1,2,\bot}(?_3, ?_4) \\ \bullet \;\; ?_2 \mapsto b^{1,2,\top}(?_3) \end{array} \right.$$

User chooses among these.

3

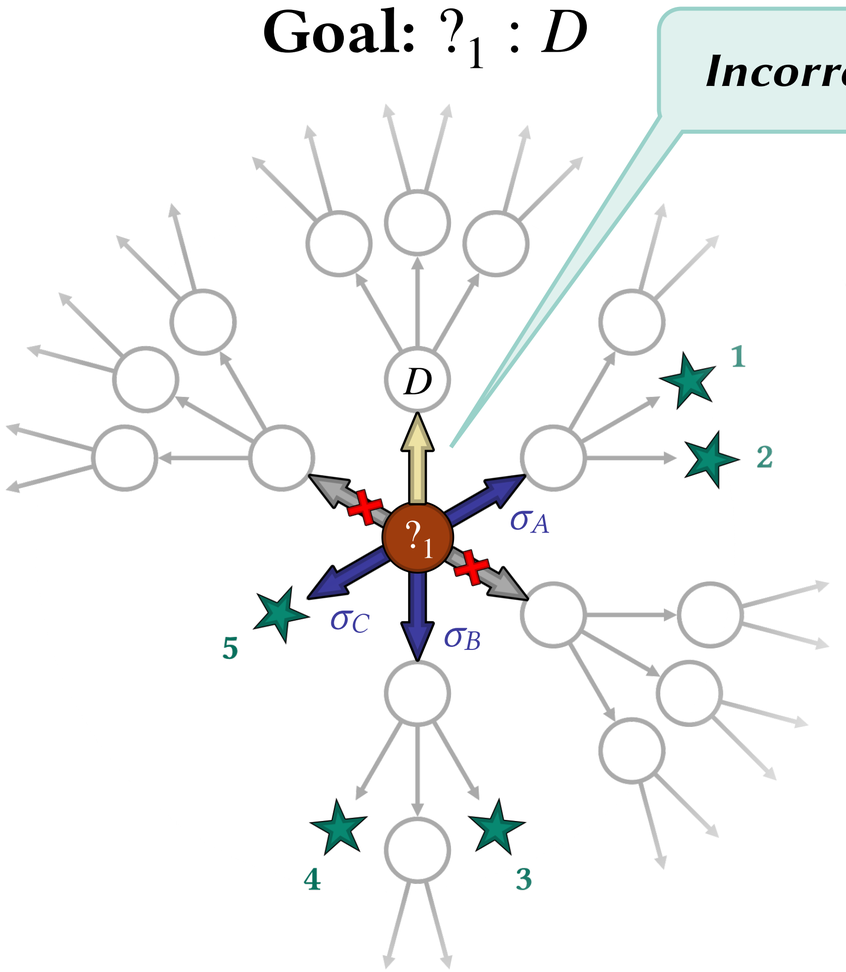
Working sketch: $d^{1,2}(b^{1,2,\top}(?_3))$

Goal: $?_3 : M$

$$\left\{ \bullet \;\; ?_3 \mapsto c^{1,2,\bot}(?_3, ?_4) \right.$$

⋮

Cannot simply look at grammar induced by the simple types.



2 **Working sketch:** $d^{1,2}(?_2)$
Goal: $?_2 : M$
→ • $c^{1,2,\perp}(?_3, ?_4)$
• $b^{1,2,\top}(?_3)$

3 **Working sketch:** $d^{1,2}(b^{1,2,\top}(?_3))$
Goal: $?_3 : M$
→ • $c^{1,2,\perp}(?_3, ?_4)$

The paper irons out a some wrinkles...

Programming by Navigation

JUSTIN LUBIN, University of California, Berkeley, USA
PARKER ZIEGLER, University of California, Berkeley, USA
SARAH E. CHASINS, University of California, Berkeley, USA

When a program synthesis task starts from an ambiguous specification, the synthesis process often involves an iterative specification refinement process. We introduce the Programming by Navigation Synthesis Problem, a new synthesis problem adapted specifically for supporting iterative specification refinement in order to find a *particular* target solution. In contrast to prior work, we prove that synthesizers that solve the Programming by Navigation Synthesis Problem show *all* valid next steps (**Strong Completeness**) and *only* valid next steps (**Strong Soundness**). To meet the demands of the Programming by Navigation Synthesis Problem, we introduce an algorithm to turn a type inhabitation oracle (in the style of classical logic) into a fully constructive program synthesizer. We then define such an oracle via sound compilation to Datalog. Our empirical evaluation shows that this technique results in an efficient Programming by Navigation synthesizer that solves tasks that are either impossible or too large for baselines to solve. Our synthesizer is the first to guarantee that its specification refinement process satisfies both **Strong Completeness** and **Strong Soundness**.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: Interactive Program Synthesis, Component-Based Synthesis, Datalog

ACM Reference Format:
Justin Lubin, Parker Ziegler, and Sarah E. Chasins. 2025. Programming by Navigation. *Proc. ACM Program. Lang.* 9, PLDI, Article 165 (June 2025), 28 pages. https://doi.org/10.1145/3729264

1 Introduction

Program synthesis tasks often begin with an underspecification of a target program [38]. If we care about refining this underspecification to reach not just any program but a *particular* program, then program synthesizers can employ an iterative specification refinement process [55, 83]. Our work starts from the observation that no existing technique for specification refinement offers what we will call **Strong Completeness** and **Strong Soundness**; that is, the guarantee that, at each round of synthesis, the synthesizer presents *all* the valid next steps (**Strong Completeness**) and *only* the valid next steps (**Strong Soundness**).

The paper irons out a some wrinkles...

Programming by Navigation

JUSTIN LUBIN, University of California, Berkeley, USA
PARKER ZIEGLER, University of California, Berkeley, USA
SARAH E. CHASINS, University of California, Berkeley, USA

When a program synthesis task starts from an ambiguous specification, the synthesis process often involves an iterative specification refinement process. We introduce the Programming by Navigation Synthesis Problem, a new synthesis problem adapted specifically for supporting iterative specification refinement in order to find a *particular* target solution. In contrast to prior work, we prove that synthesizers that solve the Programming by Navigation Synthesis Problem show *all* valid next steps (**Strong Completeness**) and *only* valid next steps (**Strong Soundness**). To meet the demands of the Programming by Navigation Synthesis Problem, we introduce an algorithm to turn a type inhabitation oracle (in the style of classical logic) into a fully constructive program synthesizer. We then define such an oracle via sound compilation to Datalog. Our empirical evaluation shows that this technique results in an efficient Programming by Navigation synthesizer that solves tasks that are either impossible or too large for baselines to solve. Our synthesizer is the first to guarantee that its specification refinement process satisfies both **Strong Completeness** and **Strong Soundness**.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: Interactive Program Synthesis, Component-Based Synthesis, Datalog

ACM Reference Format:
Justin Lubin, Parker Ziegler, and Sarah E. Chasins. 2025. Programming by Navigation. *Proc. ACM Program. Lang.* 9, PLDI, Article 165 (June 2025), 28 pages. https://doi.org/10.1145/3729264

1 Introduction

Program synthesis tasks often begin with an underspecification of a target program [38]. If we care about refining this underspecification to reach not just any program but a *particular* program, then program synthesizers can employ an iterative specification refinement process [55, 83]. Our work starts from the observation that no existing technique for specification refinement offers what we will call **Strong Completeness** and **Strong Soundness**; that is, the guarantee that, at each round of synthesis, the synthesizer presents *all* the valid next steps (**Strong Completeness**) and *only* the valid next steps (**Strong Soundness**).

- **Problem:** Need to handle entire sketches with multiple interdependent holes

The paper irons out a some wrinkles...

Programming by Navigation

JUSTIN LUBIN, University of California, Berkeley, USA
PARKER ZIEGLER, University of California, Berkeley, USA
SARAH E. CHASINS, University of California, Berkeley, USA

When a program synthesis task starts from an ambiguous specification, the synthesis process often involves an iterative specification refinement process. We introduce the Programming by Navigation Synthesis Problem, a new synthesis problem adapted specifically for supporting iterative specification refinement in order to find a *particular* target solution. In contrast to prior work, we prove that synthesizers that solve the Programming by Navigation Synthesis Problem show *all* valid next steps (**Strong Completeness**) and *only* valid next steps (**Strong Soundness**). To meet the demands of the Programming by Navigation Synthesis Problem, we introduce an algorithm to turn a type inhabitation oracle (in the style of classical logic) into a fully constructive program synthesizer. We then define such an oracle via sound compilation to Datalog. Our empirical evaluation shows that this technique results in an efficient Programming by Navigation synthesizer that solves tasks that are either impossible or too large for baselines to solve. Our synthesizer is the first to guarantee that its specification refinement process satisfies both **Strong Completeness** and **Strong Soundness**.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: Interactive Program Synthesis, Component-Based Synthesis, Datalog

ACM Reference Format:
Justin Lubin, Parker Ziegler, and Sarah E. Chasins. 2025. Programming by Navigation. *Proc. ACM Program. Lang.* 9, PLDI, Article 165 (June 2025), 28 pages. https://doi.org/10.1145/3729264

1 Introduction

Program synthesis tasks often begin with an underspecification of a target program [38]. If we care about refining this underspecification to reach not just any program but a *particular* program, then program synthesizers can employ an iterative specification refinement process [55, 83]. Our work starts from the observation that no existing technique for specification refinement offers what we will call **Strong Completeness** and **Strong Soundness**; that is, the guarantee that, at each round of synthesis, the synthesizer presents *all* the valid next steps (**Strong Completeness**) and *only* the valid next steps (**Strong Soundness**).

- **Problem:** Need to handle entire sketches with multiple interdependent holes
- **Solution:** Define “query” rules with a corresponding key invariants

The paper irons out a some wrinkles...

Programming by Navigation

JUSTIN LUBIN, University of California, Berkeley, USA
PARKER ZIEGLER, University of California, Berkeley, USA
SARAH E. CHASINS, University of California, Berkeley, USA

When a program synthesis task starts from an ambiguous specification, the synthesis process often involves an iterative specification refinement process. We introduce the Programming by Navigation Synthesis Problem, a new synthesis problem adapted specifically for supporting iterative specification refinement in order to find a *particular* target solution. In contrast to prior work, we prove that synthesizers that solve the Programming by Navigation Synthesis Problem show *all* valid next steps (**Strong Completeness**) and *only* valid next steps (**Strong Soundness**). To meet the demands of the Programming by Navigation Synthesis Problem, we introduce an algorithm to turn a type inhabitation oracle (in the style of classical logic) into a fully constructive program synthesizer. We then define such an oracle via sound compilation to Datalog. Our empirical evaluation shows that this technique results in an efficient Programming by Navigation synthesizer that solves tasks that are either impossible or too large for baselines to solve. Our synthesizer is the first to guarantee that its specification refinement process satisfies both **Strong Completeness** and **Strong Soundness**.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: Interactive Program Synthesis, Component-Based Synthesis, Datalog

ACM Reference Format:
Justin Lubin, Parker Ziegler, and Sarah E. Chasins. 2025. Programming by Navigation. *Proc. ACM Program. Lang.* 9, PLDI, Article 165 (June 2025), 28 pages. https://doi.org/10.1145/3729264

1 Introduction

Program synthesis tasks often begin with an underspecification of a target program [38]. If we care about refining this underspecification to reach not just any program but a *particular* program, then program synthesizers can employ an iterative specification refinement process [55, 83]. Our work starts from the observation that no existing technique for specification refinement offers what we will call **Strong Completeness** and **Strong Soundness**; that is, the guarantee that, at each round of synthesis, the synthesizer presents *all* the valid next steps (**Strong Completeness**) and *only* the valid next steps (**Strong Soundness**).

- **Problem:** Need to handle entire sketches with multiple interdependent holes
- **Solution:** Define “query” rules with a corresponding key invariants

- **Problem:** Need to determine exactly *which* functions are valid expansions to show as a step (not just that *some* expansion exists).

The paper irons out a some wrinkles…

Programming by Navigation

JUSTIN LUBIN, University of California, Berkeley, USA
PARKER ZIEGLER, University of California, Berkeley, USA
SARAH E. CHASINS, University of California, Berkeley, USA

When a program synthesis task starts from an ambiguous specification, the synthesis process often involves an iterative specification refinement process. We introduce the Programming by Navigation Synthesis Problem, a new synthesis problem adapted specifically for supporting iterative specification refinement in order to find a *particular* target solution. In contrast to prior work, we prove that synthesizers that solve the Programming by Navigation Synthesis Problem show *all* valid next steps (**Strong Completeness**) and *only* valid next steps (**Strong Soundness**). To meet the demands of the Programming by Navigation Synthesis Problem, we introduce an algorithm to turn a type inhabitation oracle (in the style of classical logic) into a fully constructive program synthesizer. We then define such an oracle via sound compilation to Datalog. Our empirical evaluation shows that this technique results in an efficient Programming by Navigation synthesizer that solves tasks that are either impossible or too large for baselines to solve. Our synthesizer is the first to guarantee that its specification refinement process satisfies both **Strong Completeness** and **Strong Soundness**.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: Interactive Program Synthesis, Component-Based Synthesis, Datalog

ACM Reference Format:
Justin Lubin, Parker Ziegler, and Sarah E. Chasins. 2025. Programming by Navigation. *Proc. ACM Program. Lang.* 9, PLDI, Article 165 (June 2025), 28 pages. https://doi.org/10.1145/3729264

1 Introduction

Program synthesis tasks often begin with an underspecification of a target program [38]. If we care about refining this underspecification to reach not just any program but a *particular* program, then program synthesizers can employ an iterative specification refinement process [55, 83]. Our work starts from the observation that no existing technique for specification refinement offers what we will call **Strong Completeness** and **Strong Soundness**; that is, the guarantee that, at each round of synthesis, the synthesizer presents *all* the valid next steps (**Strong Completeness**) and *only* the valid next steps (**Strong Soundness**).

- **Problem:** Need to handle entire sketches with multiple interdependent holes
- **Solution:** Define “query” rules with a corresponding key invariants

- **Problem:** Need to determine exactly *which* functions are valid expansions to show as a step (not just that *some* expansion exists).
- **Solution:** Use logical cuts (in the sense of program fusion) to specialize the proof rules appropriately.