

Computer-Aided Reasoning for Software

CSE 507 Metasketches

courses.cs.washington.edu/courses/cse507/17wi/

James Bornholt

bornholt@cs.washington.edu

Today

Last lecture

- Program synthesis
- Solver-aided languages

Today

- Metasketches: building effective synthesis-aided tools
- MemSynth: an example of a metasketch-based tool

Reminders

- Course feedback form is open
- Project presentations on Friday
- Project reports and prototypes due Friday at 11:00pm

Metasketches

James Bornholt, Emin Torlak, Luis Ceze, and Dan Grossman.
Optimizing Synthesis with Metasketches. POPL 2016.

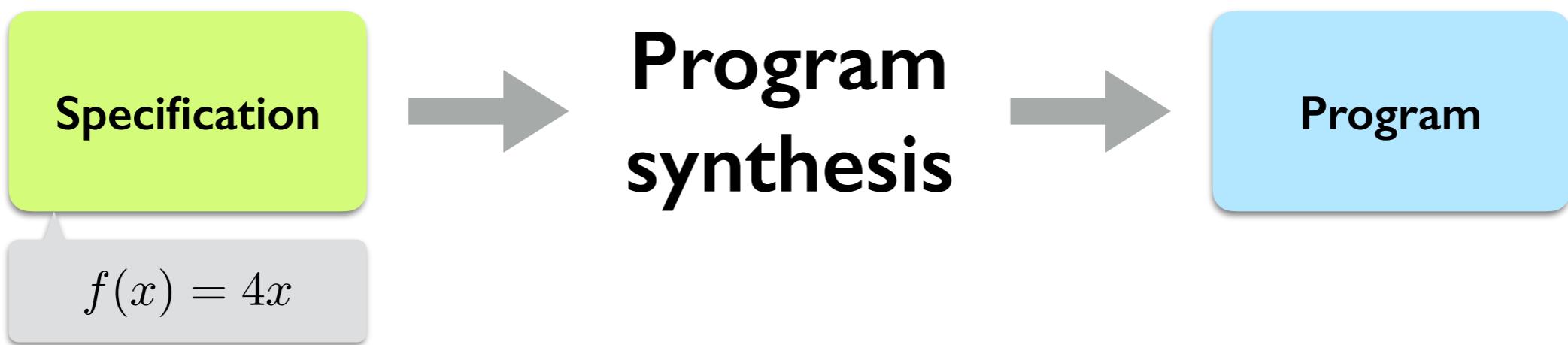
Program synthesis

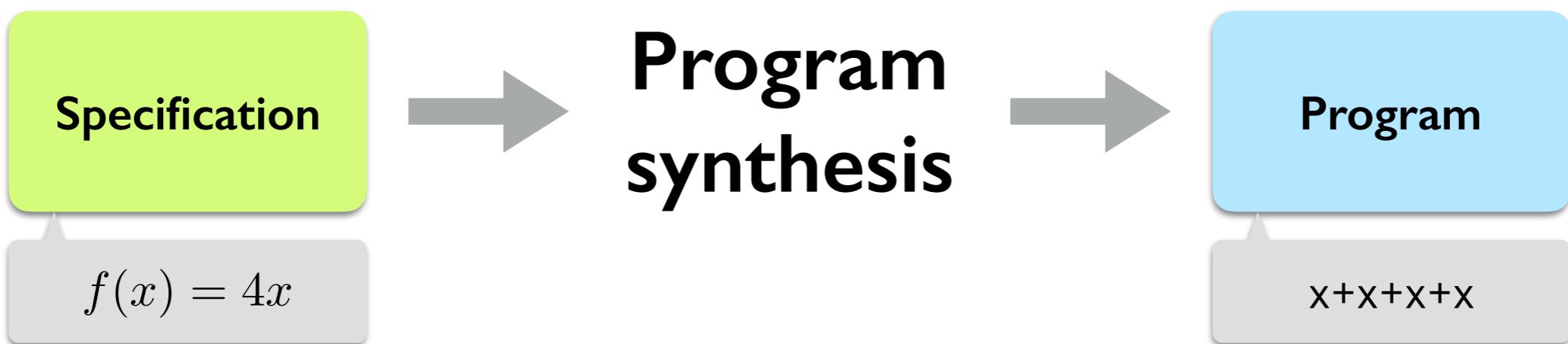
Specification



**Program
synthesis**







Compilation
[PLDI'14]

Data Structures
[PLDI'15]

End-user Programming
[POPL'11]

Specification

$$f(x) = 4x$$

Program synthesis

Program

$$x+x+x+x$$

Executable Biology
[POPL'13]

Browser Layout
[PPoPP'13]

Cache
Protocols
[PLDI'13]

Compilation
[PLDI'14]

Data Structures
[PLDI'15]

End-user Programming
[POPL'11]

Specification



Program synthesis

Program

Often looking for an
optimal solution, not just
any correct program

Executable Biology
[POPL'13]

Browser Layout
[PPoPP'13]

Cache
Protocols
[PLDI'13]

Compilation
[PLDI'14]

Data Structures
[PLDI'15]

End-user Programming
[POPL'11]

Specification



Program synthesis

Program

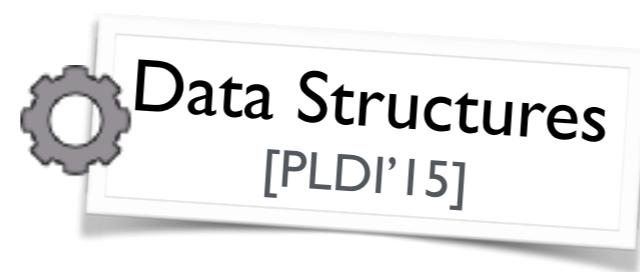
Often looking for an *optimal* solution, not just any correct program

There are *many* programs, so tools must control search strategy

Executable Biology
[POPL'13]

Browser Layout
[PPoPP'13]

Cache Protocols
[PLDI'13]



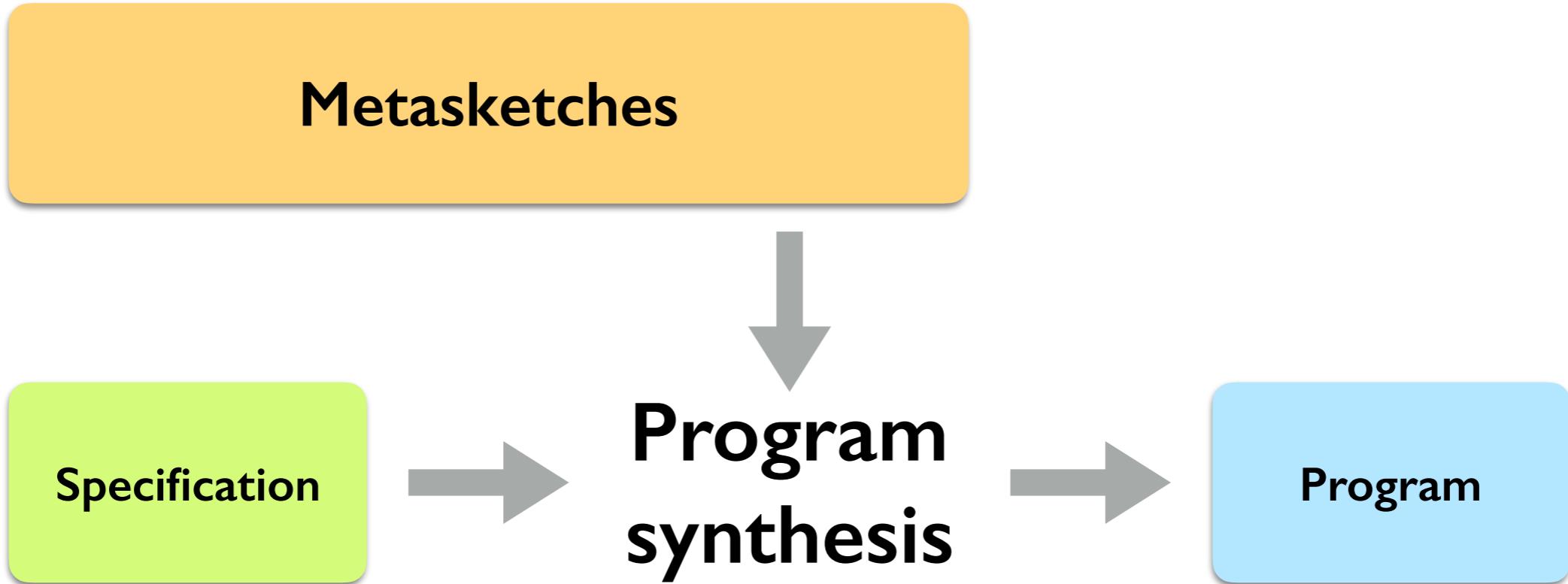
Often looking for an *optimal* solution, not just any correct program



There are *many* programs, so tools must control search strategy







Metasketches

A framework that makes search strategy and optimality part of the problem definition

Specification



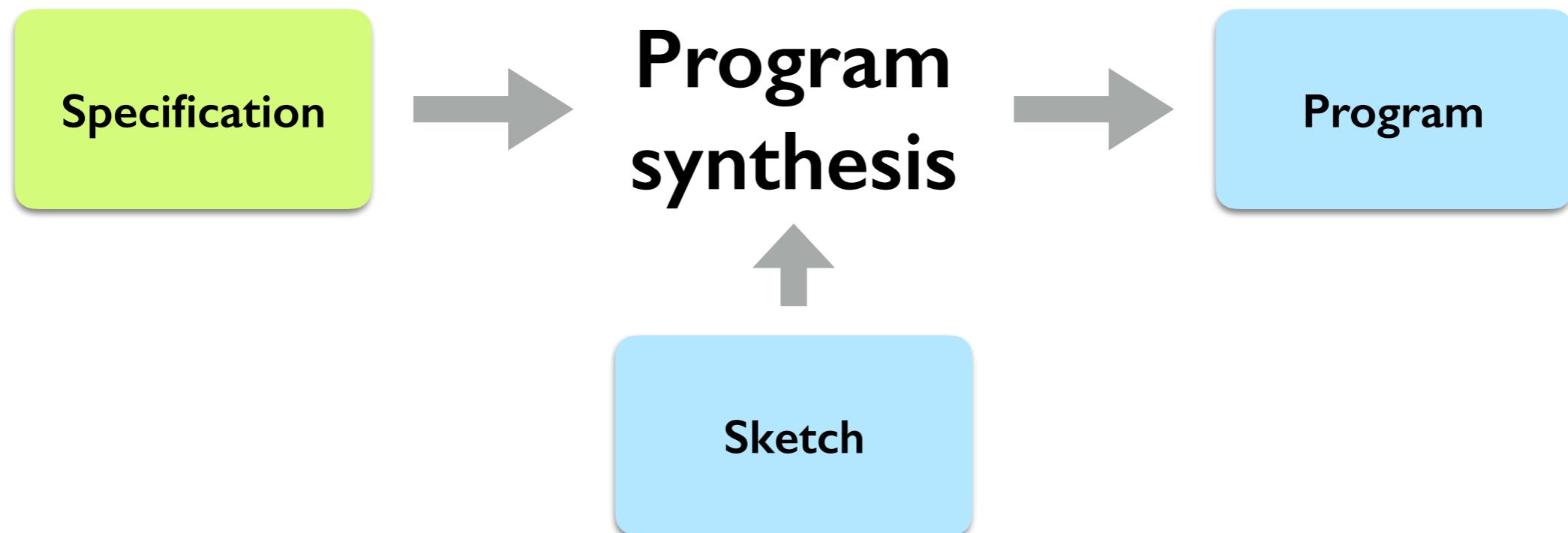
Program synthesis

Program

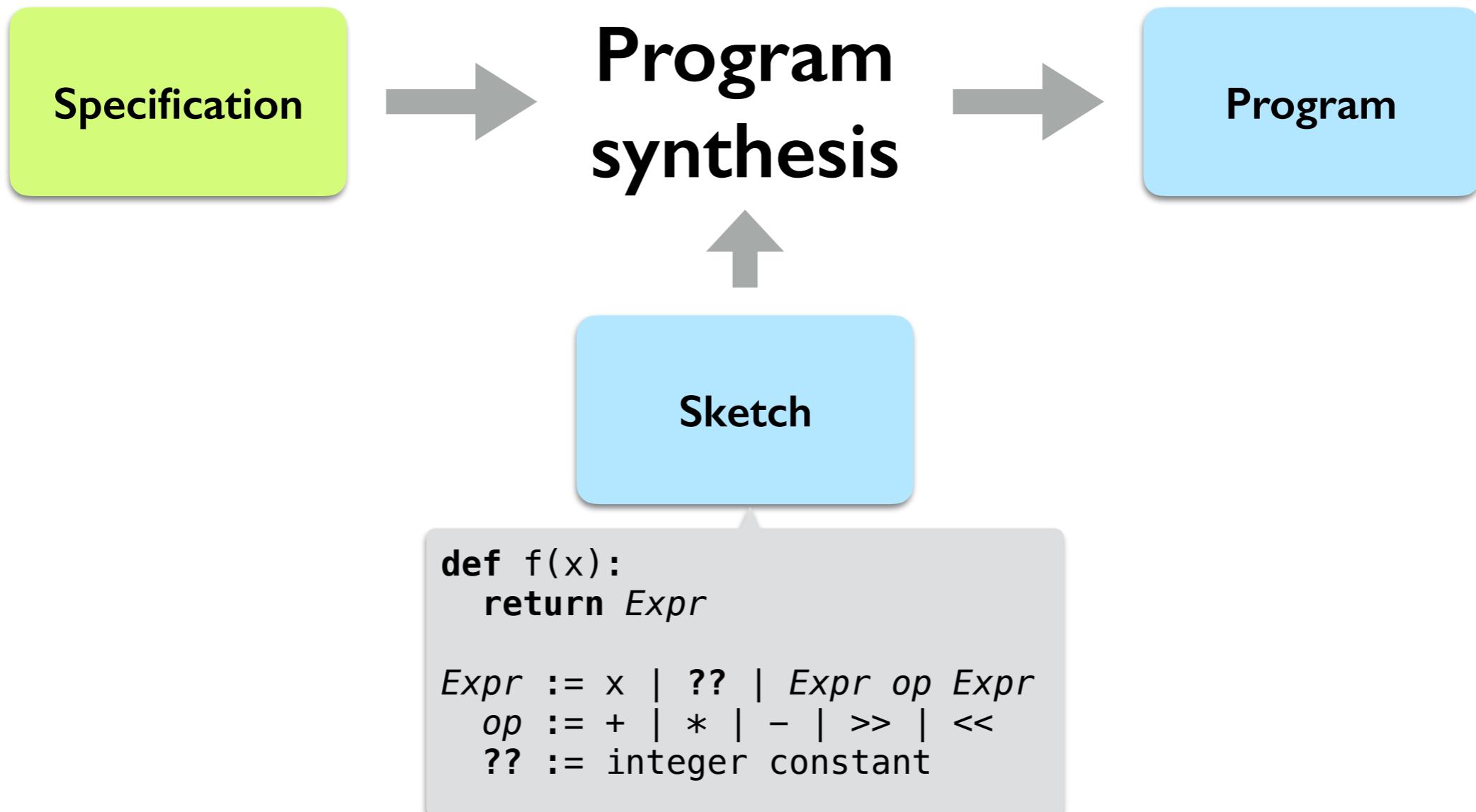
Syntax-guided synthesis



Syntax-guided synthesis



Syntax-guided synthesis



Syntax-guided synthesis: guess, check, learn

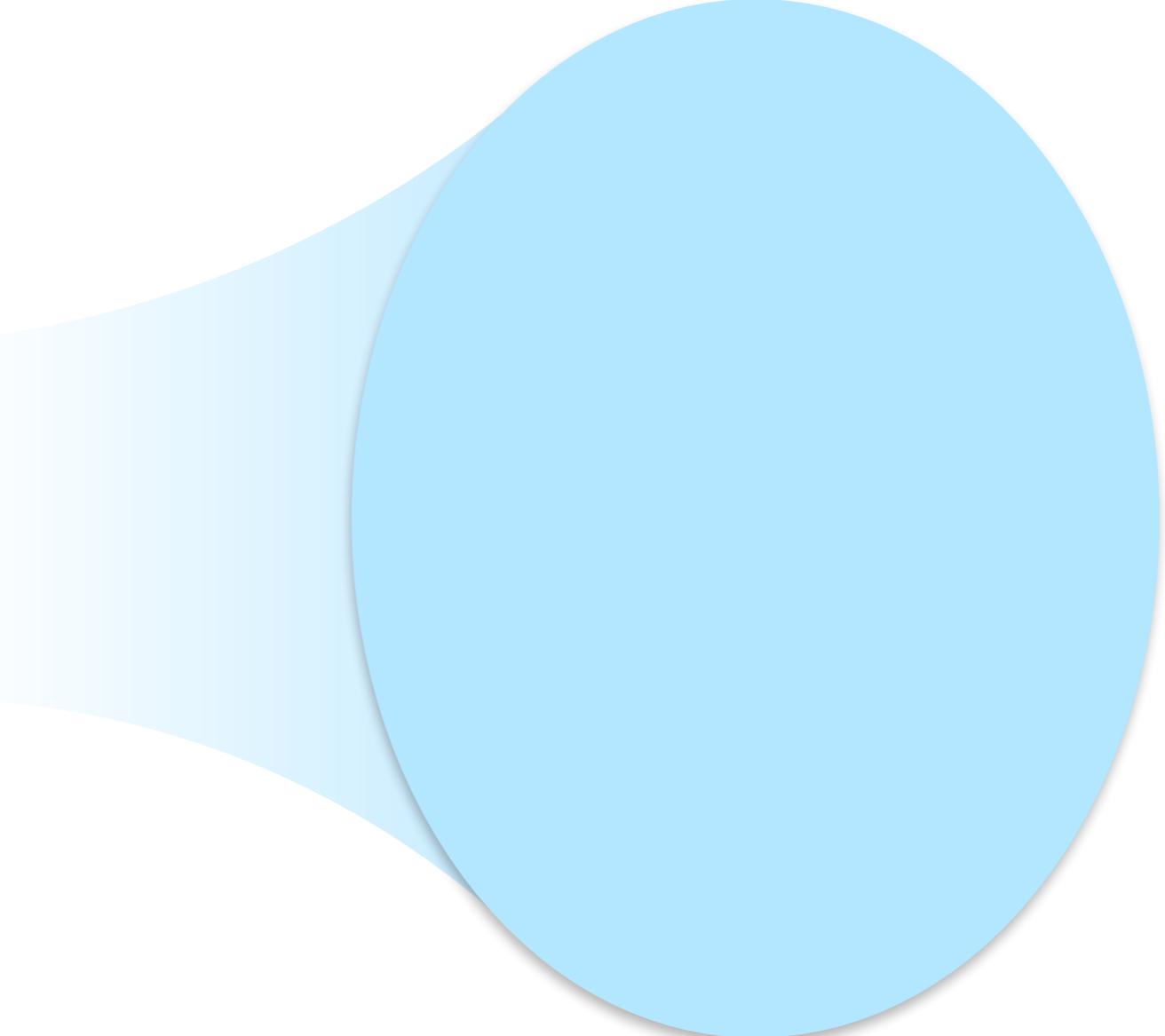
```
def f(x):
    return Expr

Expr ::= x | ?? | Expr op Expr
op ::= + | * | - | >> | <<
?? ::= integer constant
```

Syntax-guided synthesis: guess, check, learn

```
def f(x):  
    return Expr
```

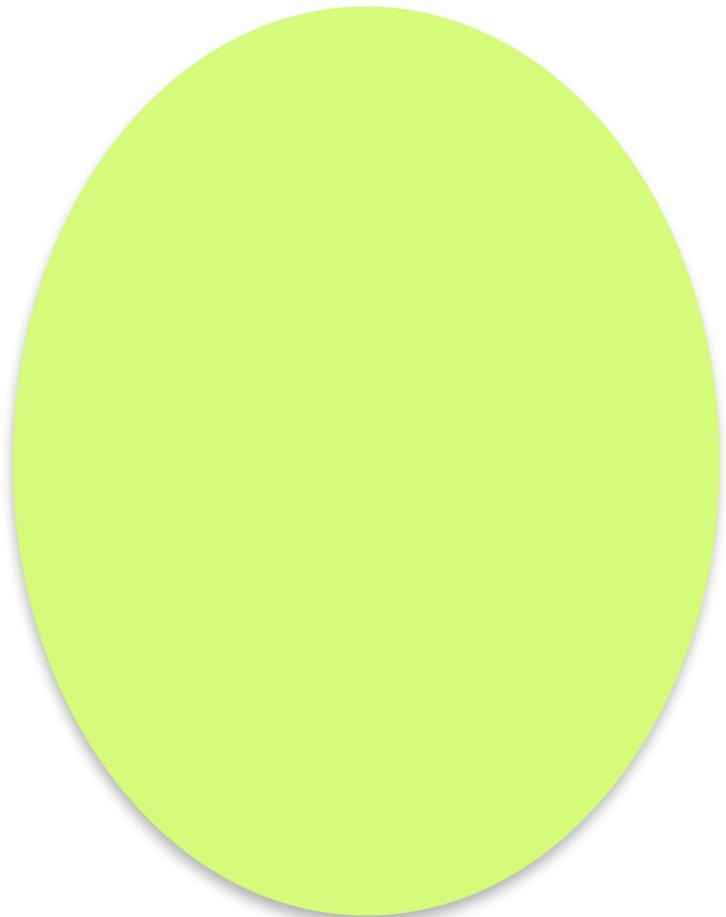
Expr := *x* | *??* | *Expr op Expr*
op := + | * | - | >> | <<
?? := integer constant



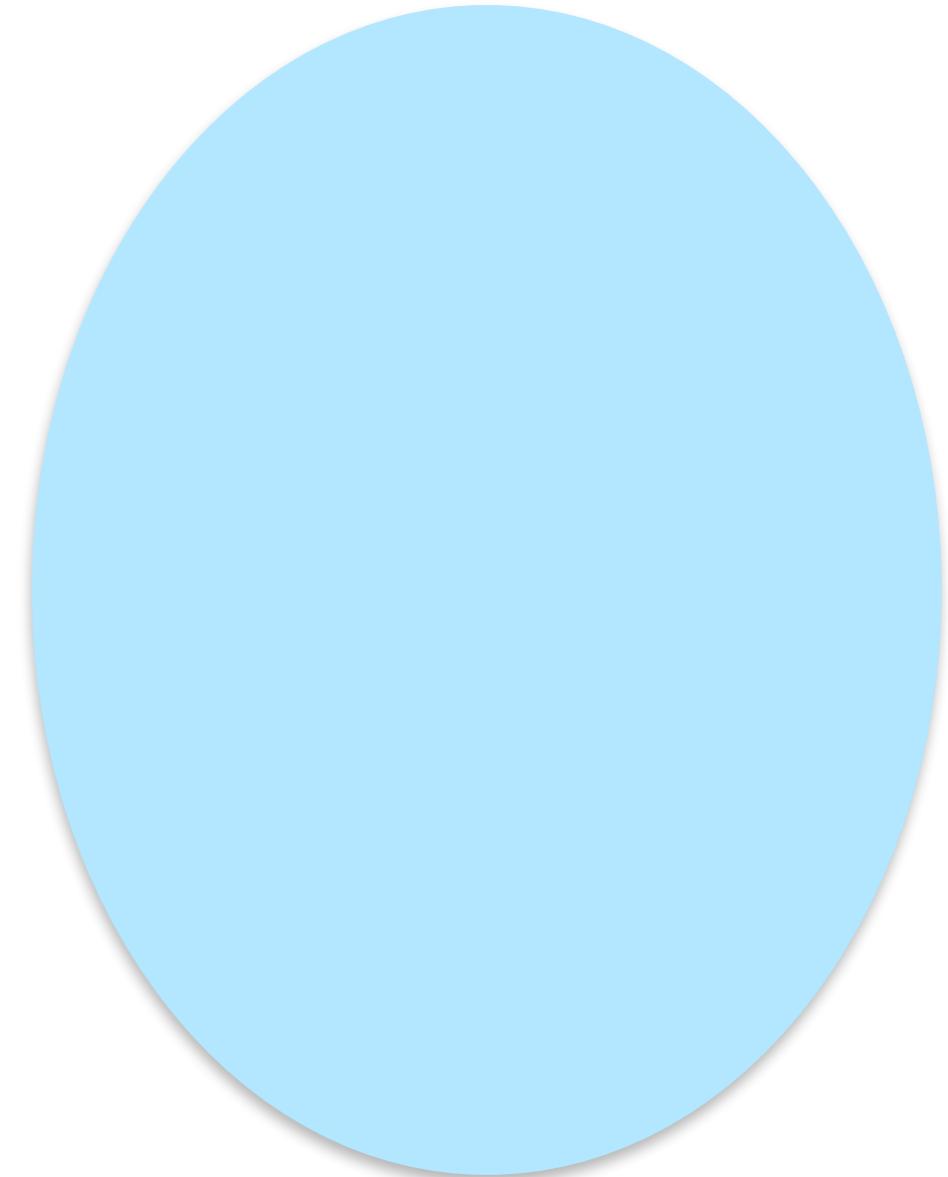
Syntax

Syntax-guided synthesis: guess, check, learn

Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]



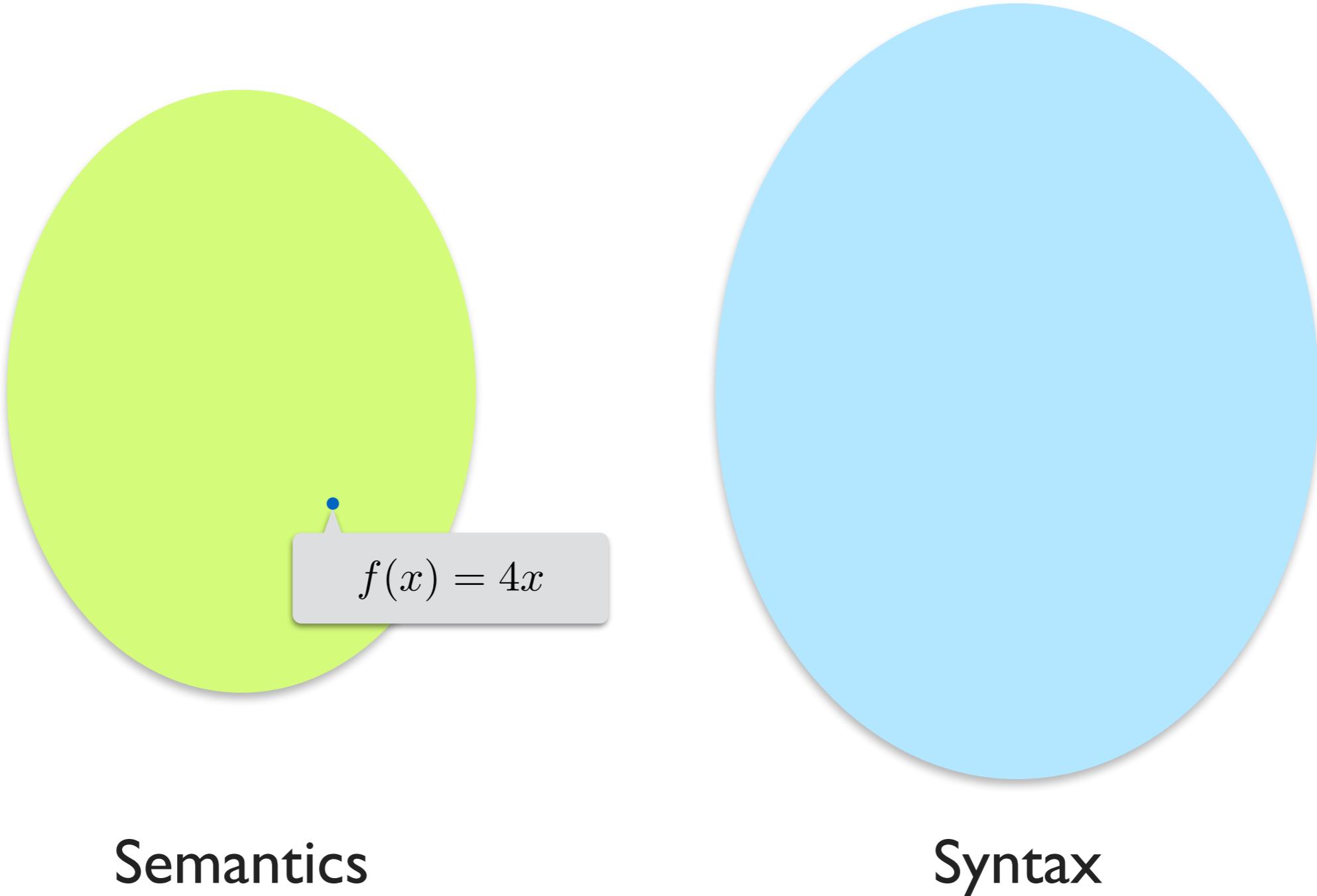
Semantics



Syntax

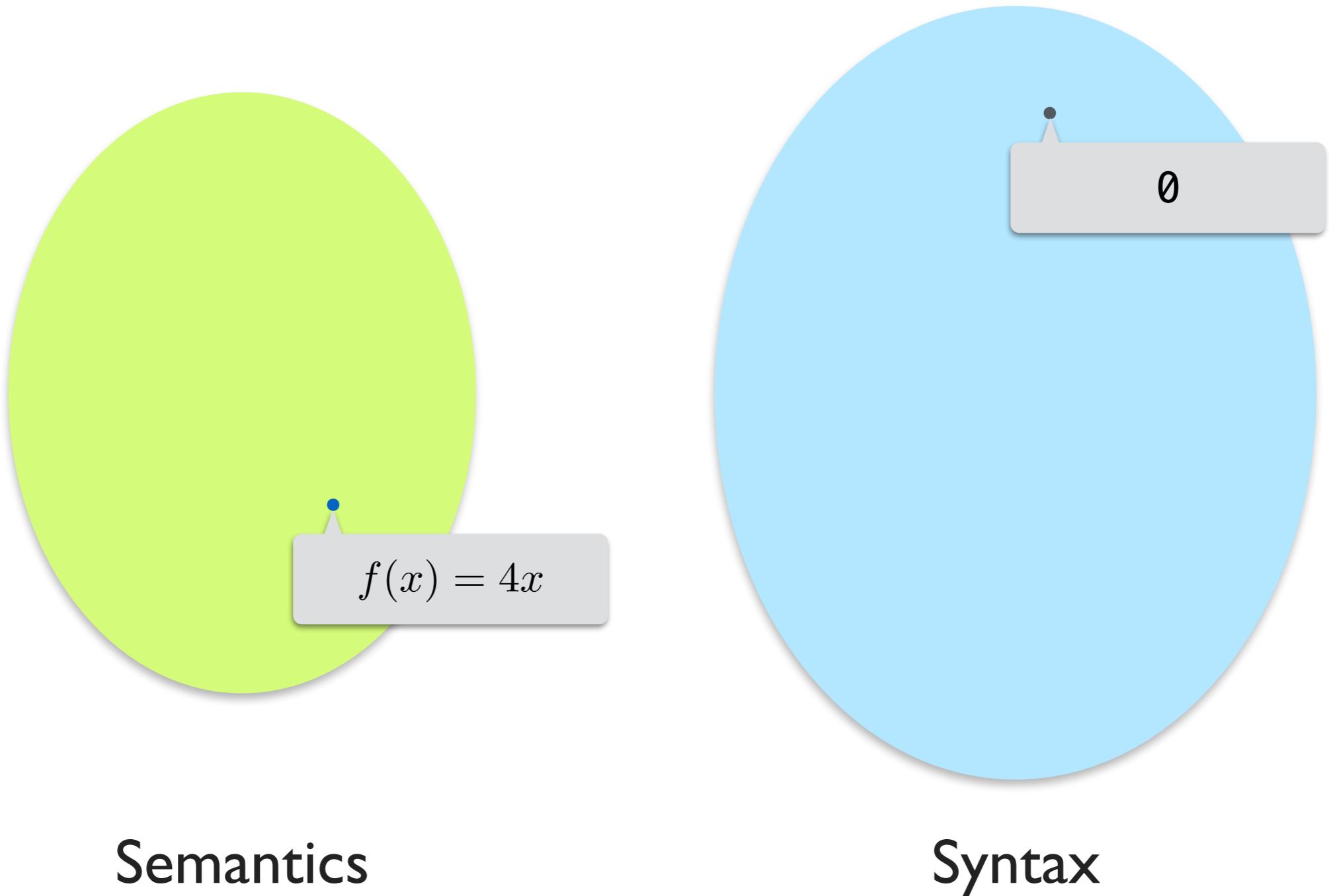
Syntax-guided synthesis: guess, check, learn

Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]



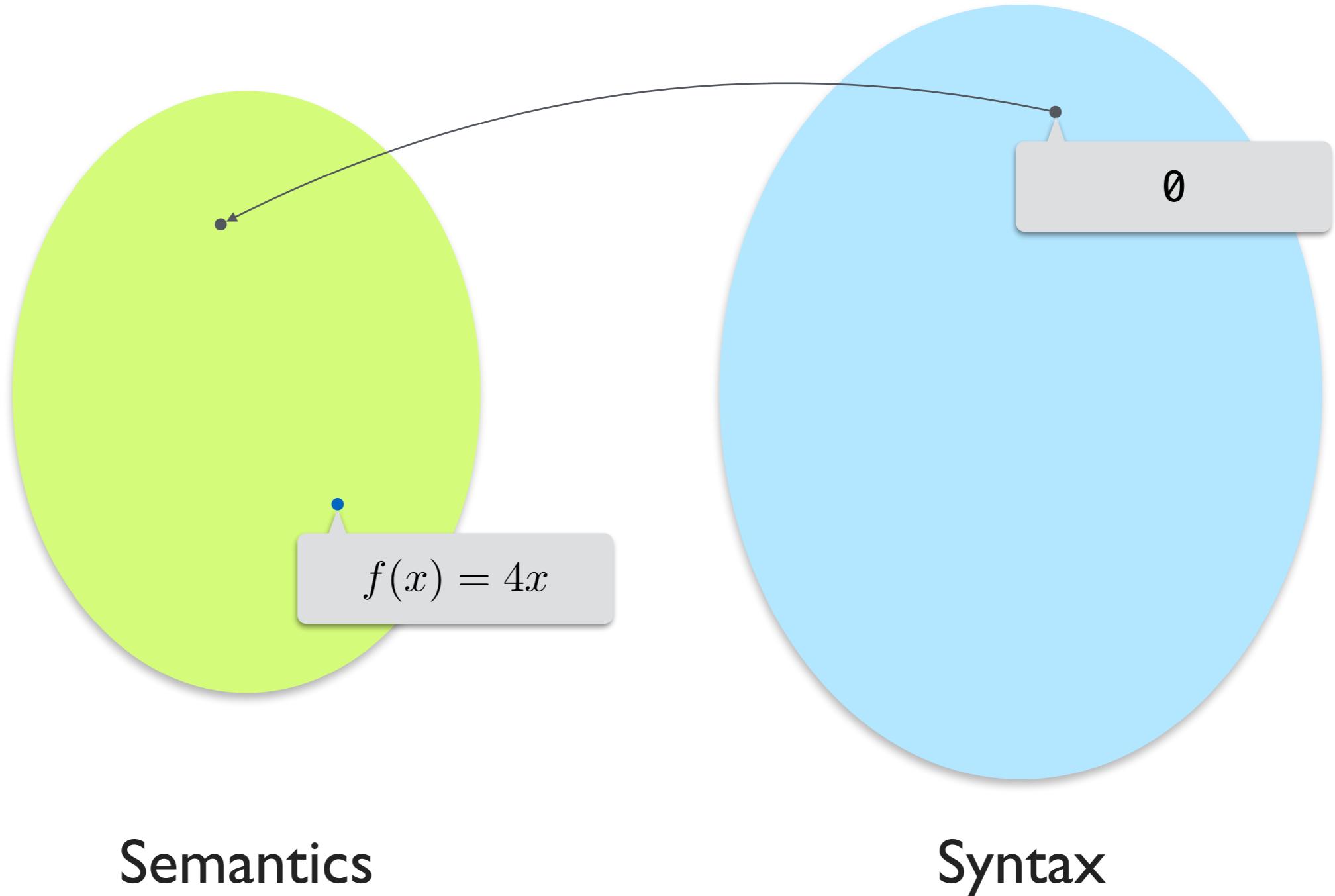
Syntax-guided synthesis: guess, check, learn

Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]



Syntax-guided synthesis: guess, check, learn

Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]

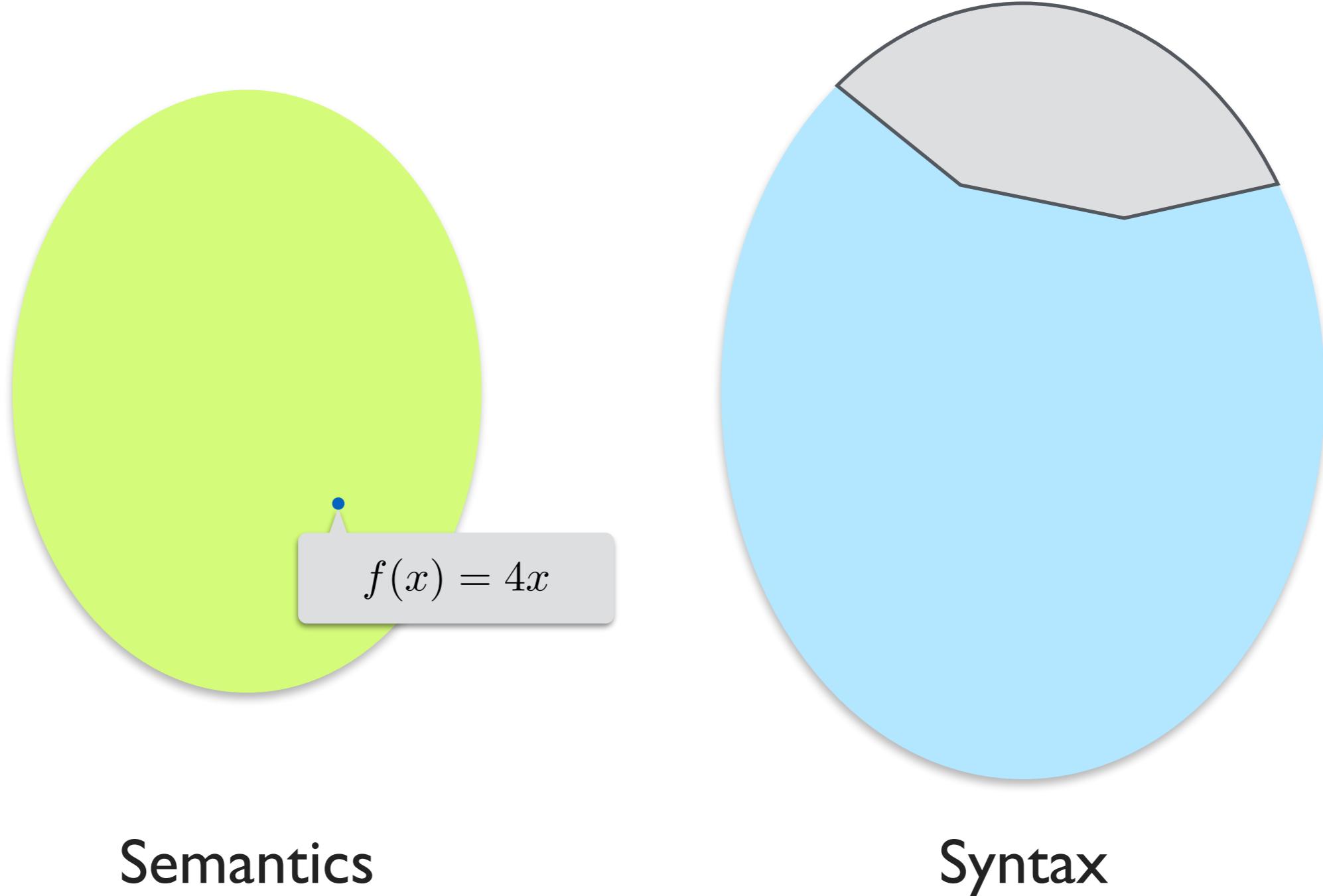


Semantics

Syntax

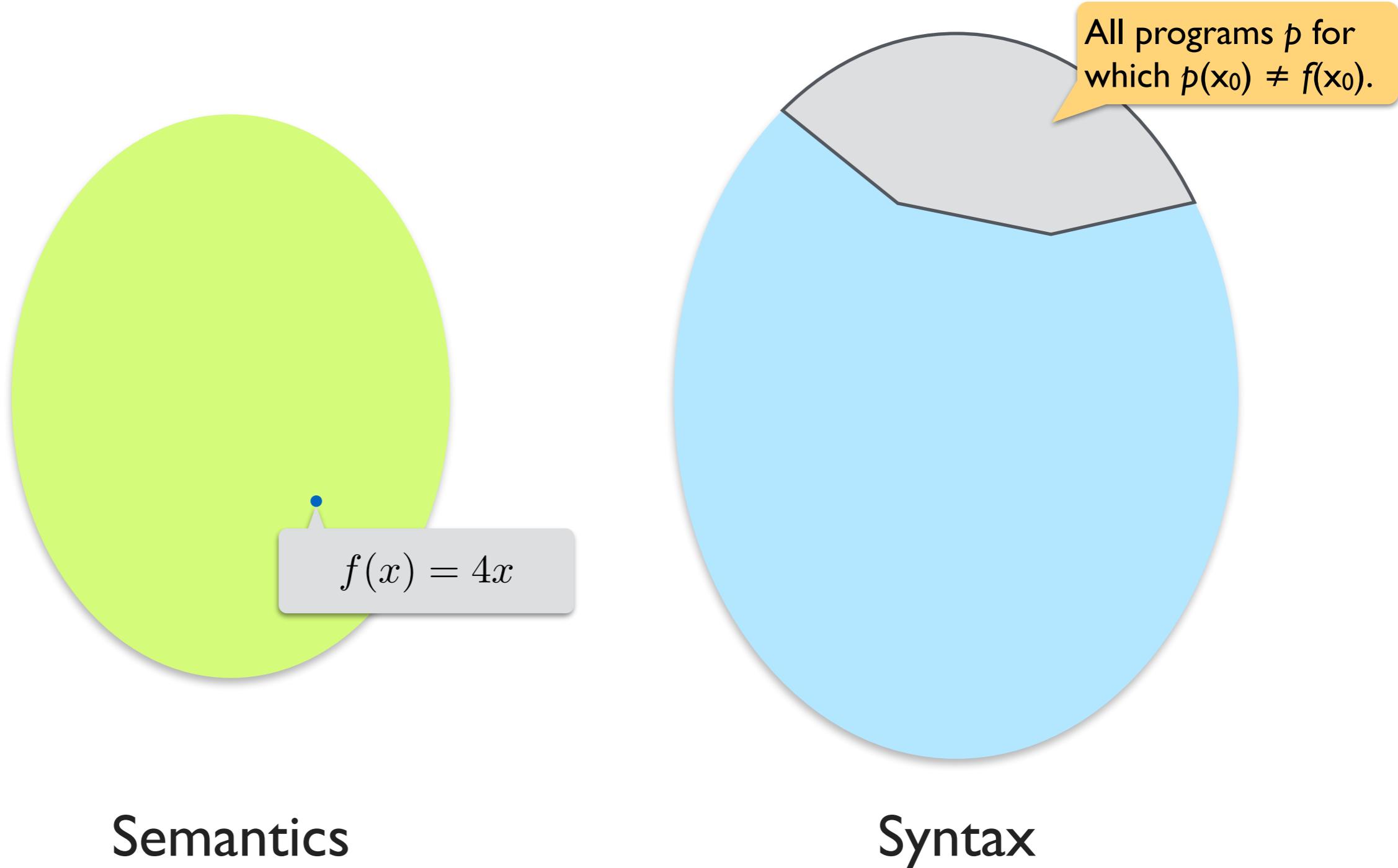
Syntax-guided synthesis: guess, check, learn

Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]



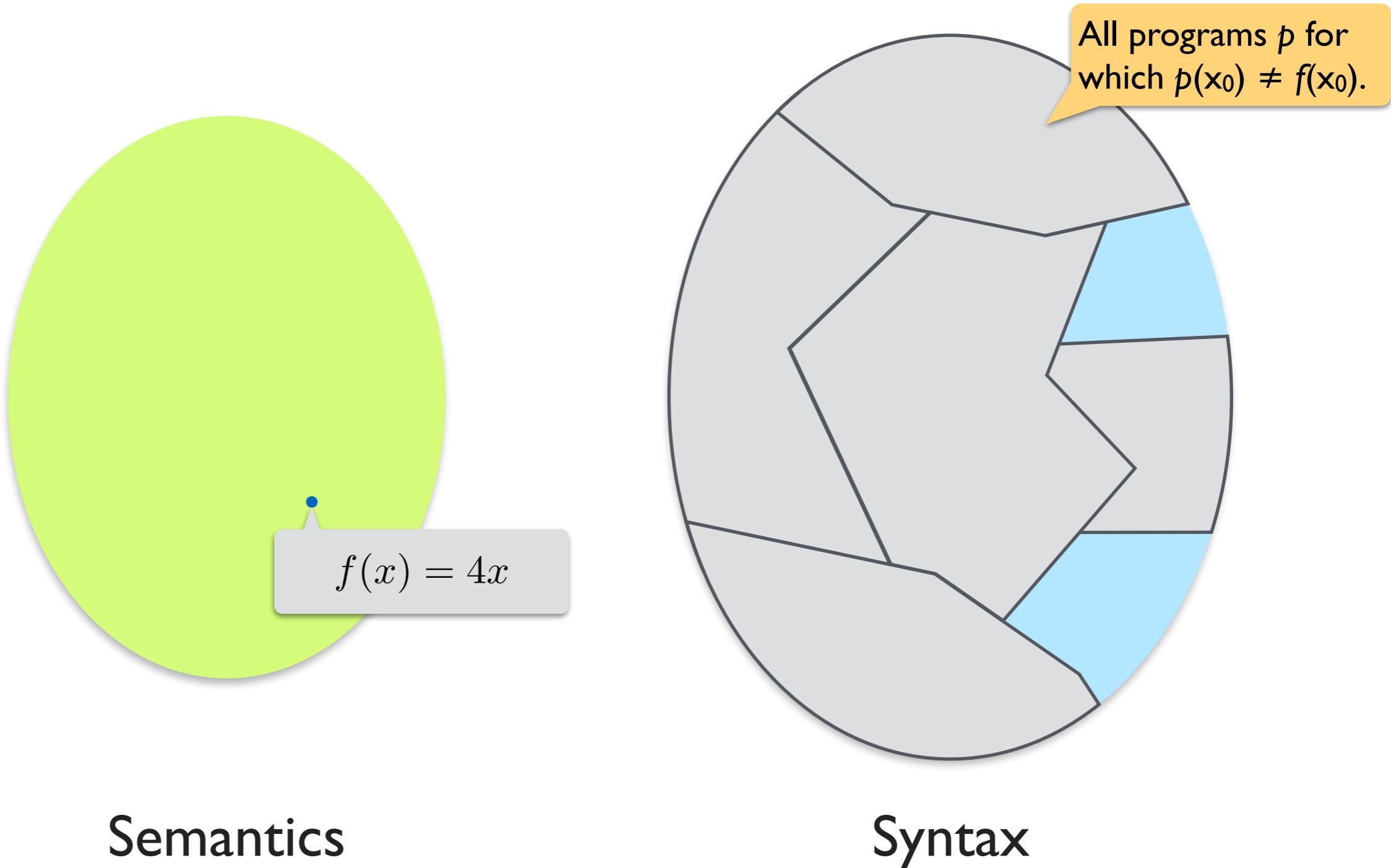
Syntax-guided synthesis: guess, check, learn

Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]



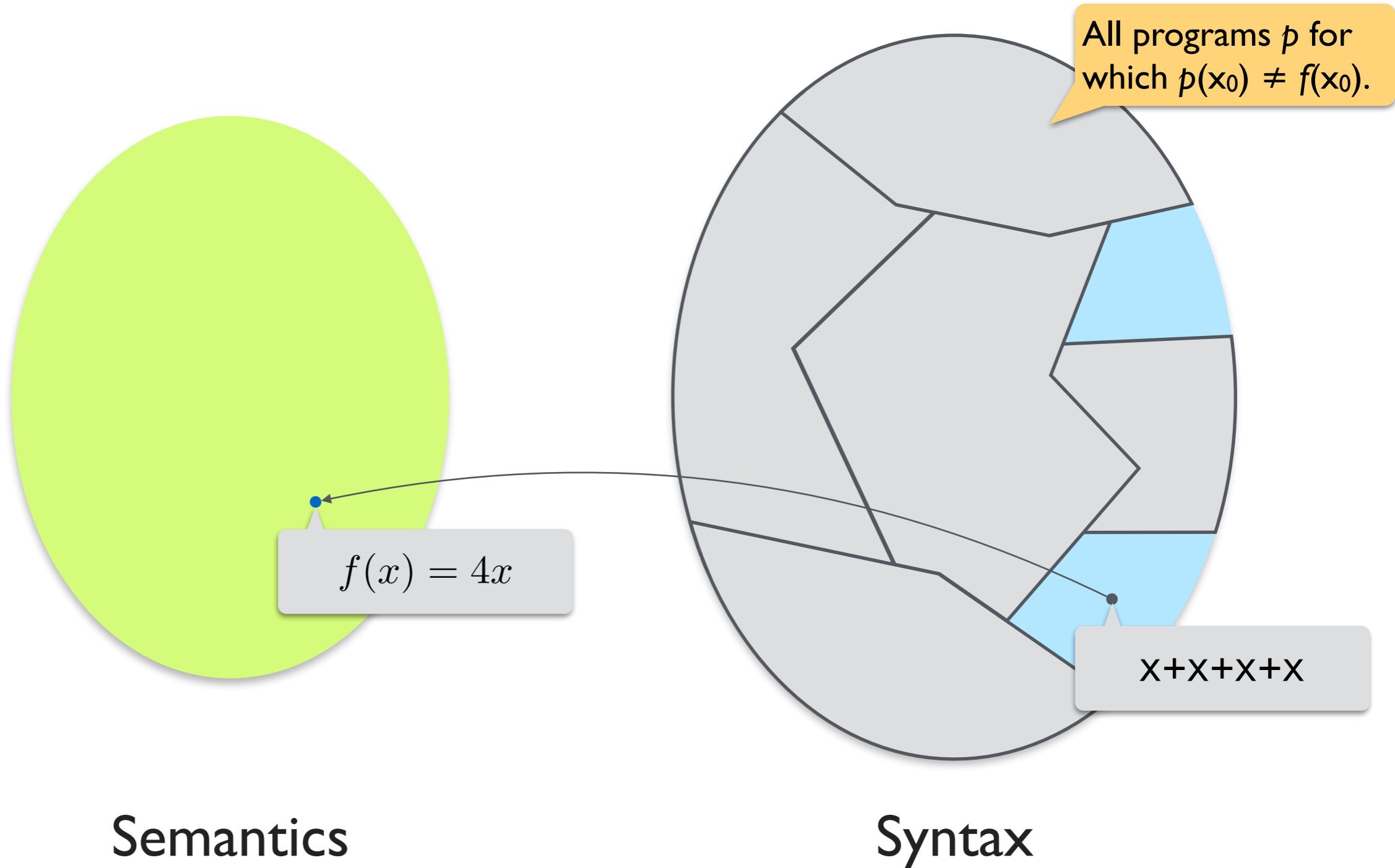
Syntax-guided synthesis: guess, check, learn

Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]



Syntax-guided synthesis: guess, check, learn

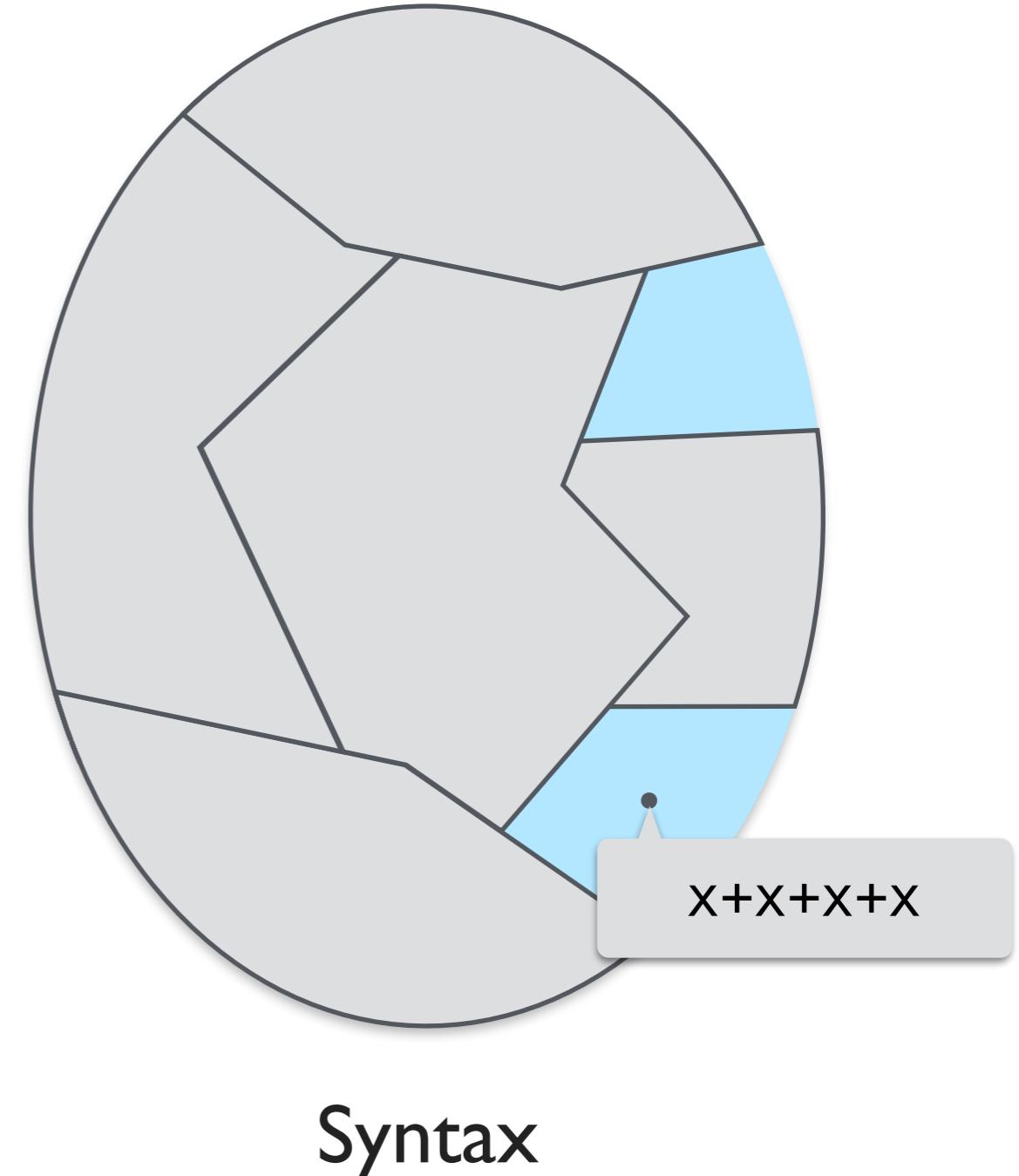
Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]



Syntax-guided synthesis: guess, check, learn

Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]

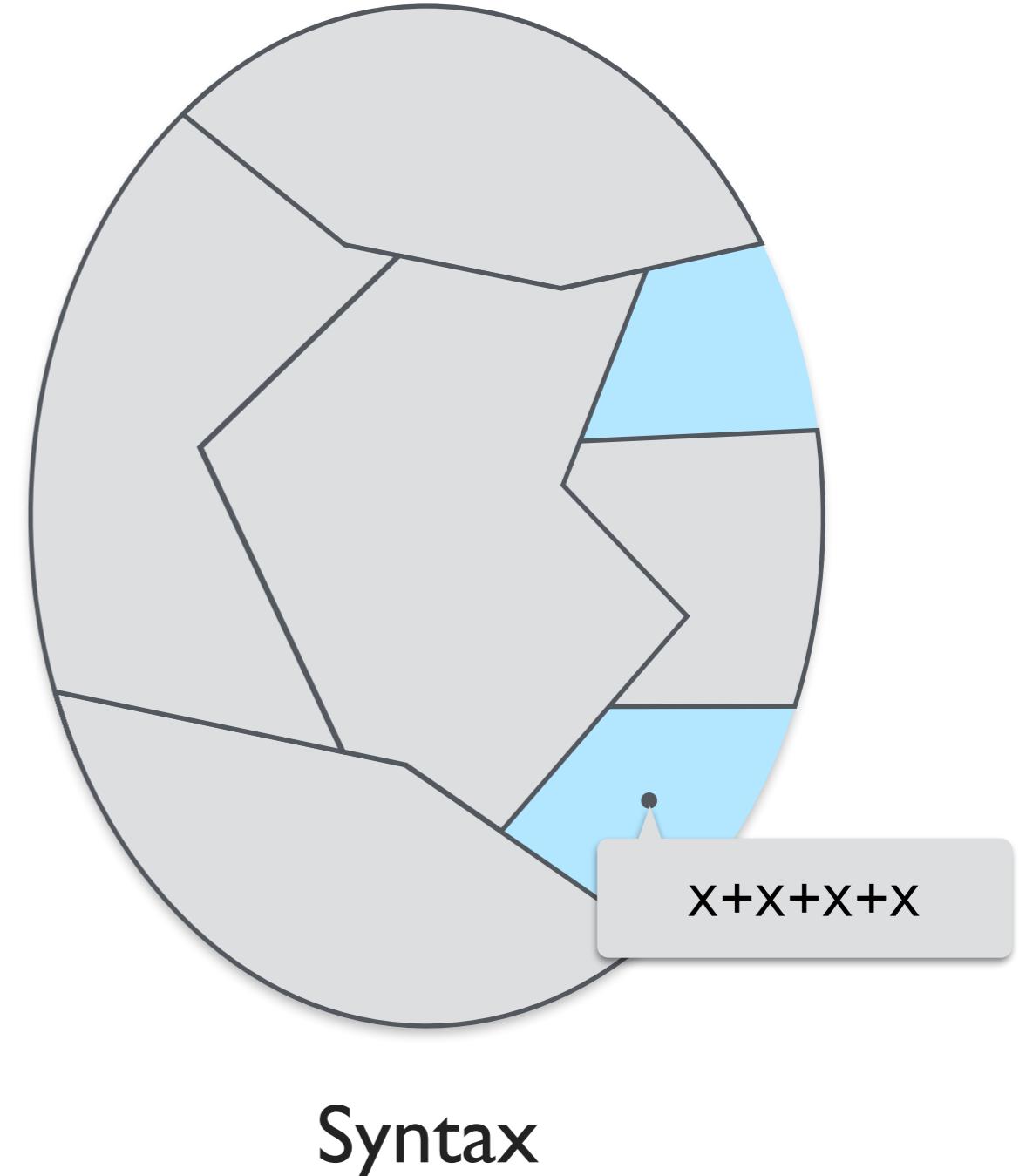
- I. Search order is critical



Syntax-guided synthesis: guess, check, learn

Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]

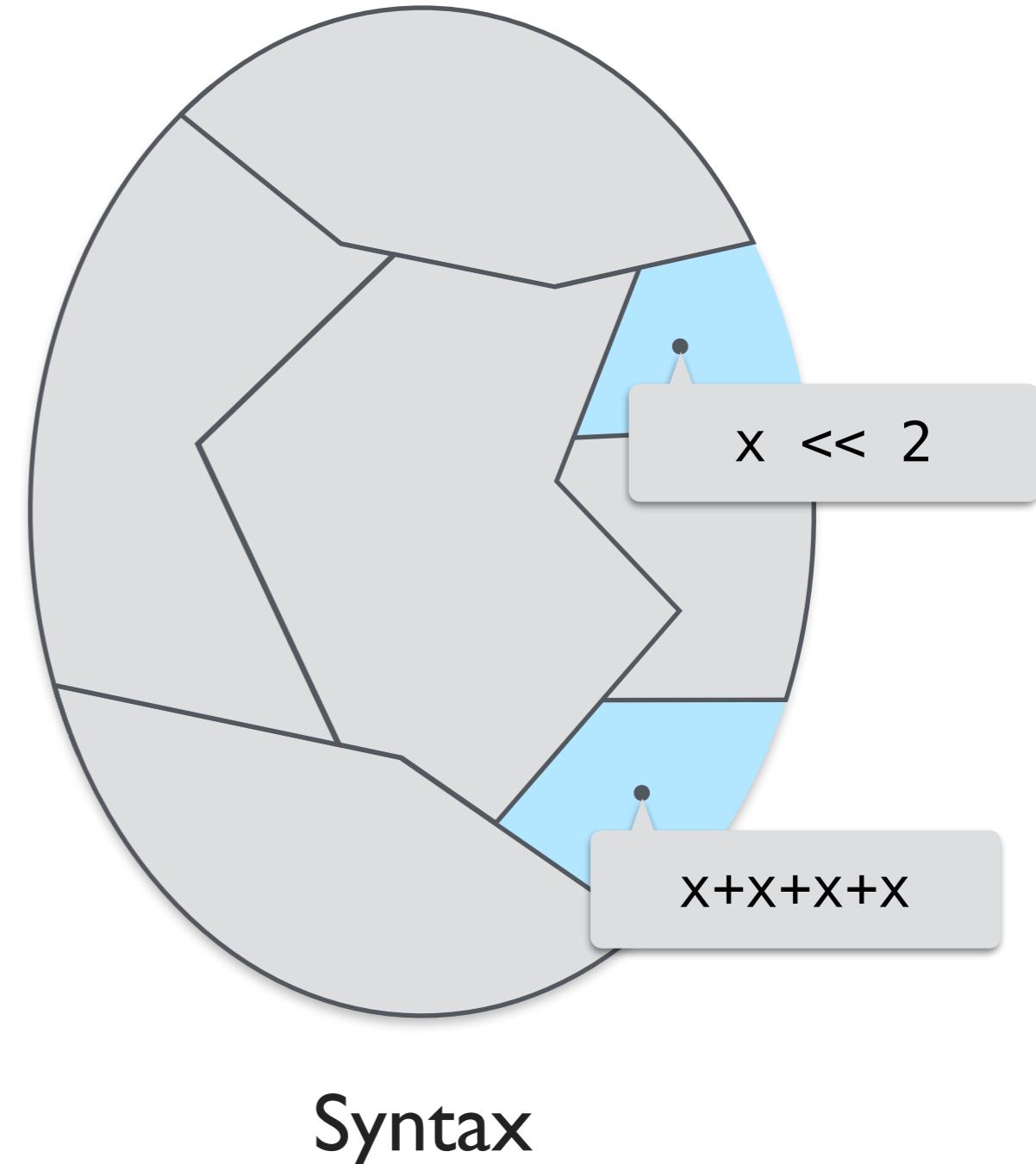
- 1. Search order is critical
- 2. Desire optimal solutions



Syntax-guided synthesis: guess, check, learn

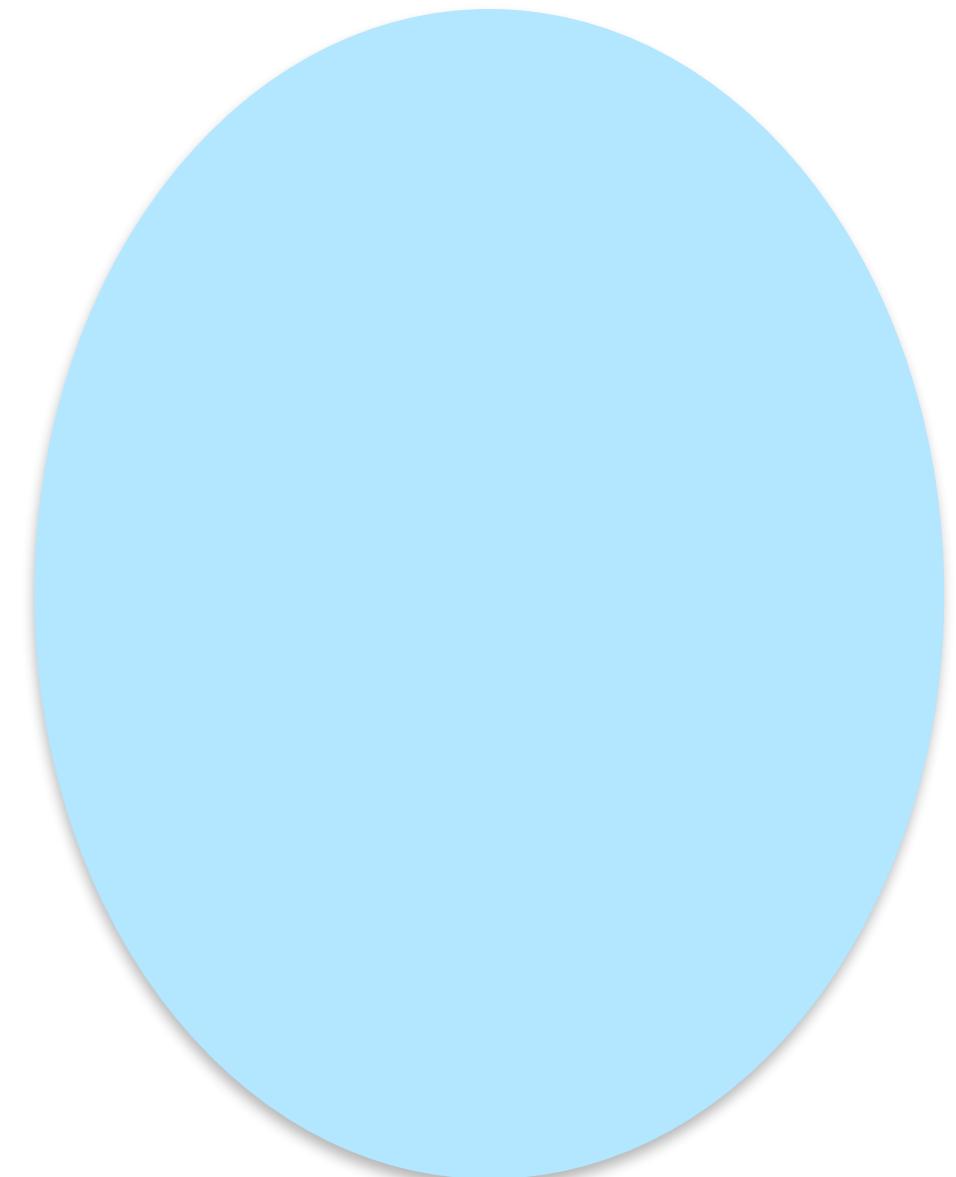
Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]

1. Search order is critical
2. Desire optimal solutions



Metasketches express structure and strategy

1. Search order is critical
2. Desire optimal solutions



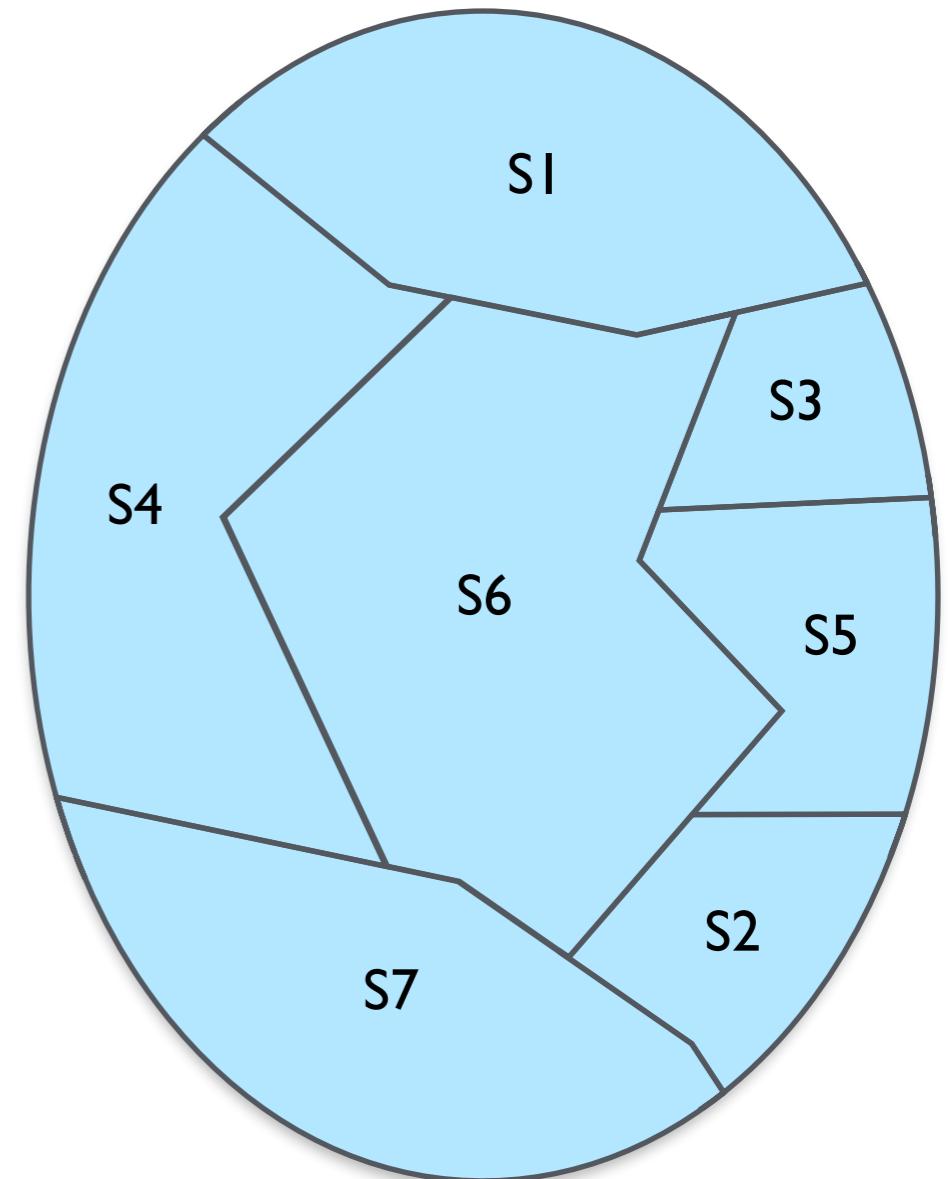
Syntax

Metasketches express structure and strategy

1. Search order is critical
2. Desire optimal solutions

A metasketch contains:

1. structured candidate space (\mathcal{S}, \leq)



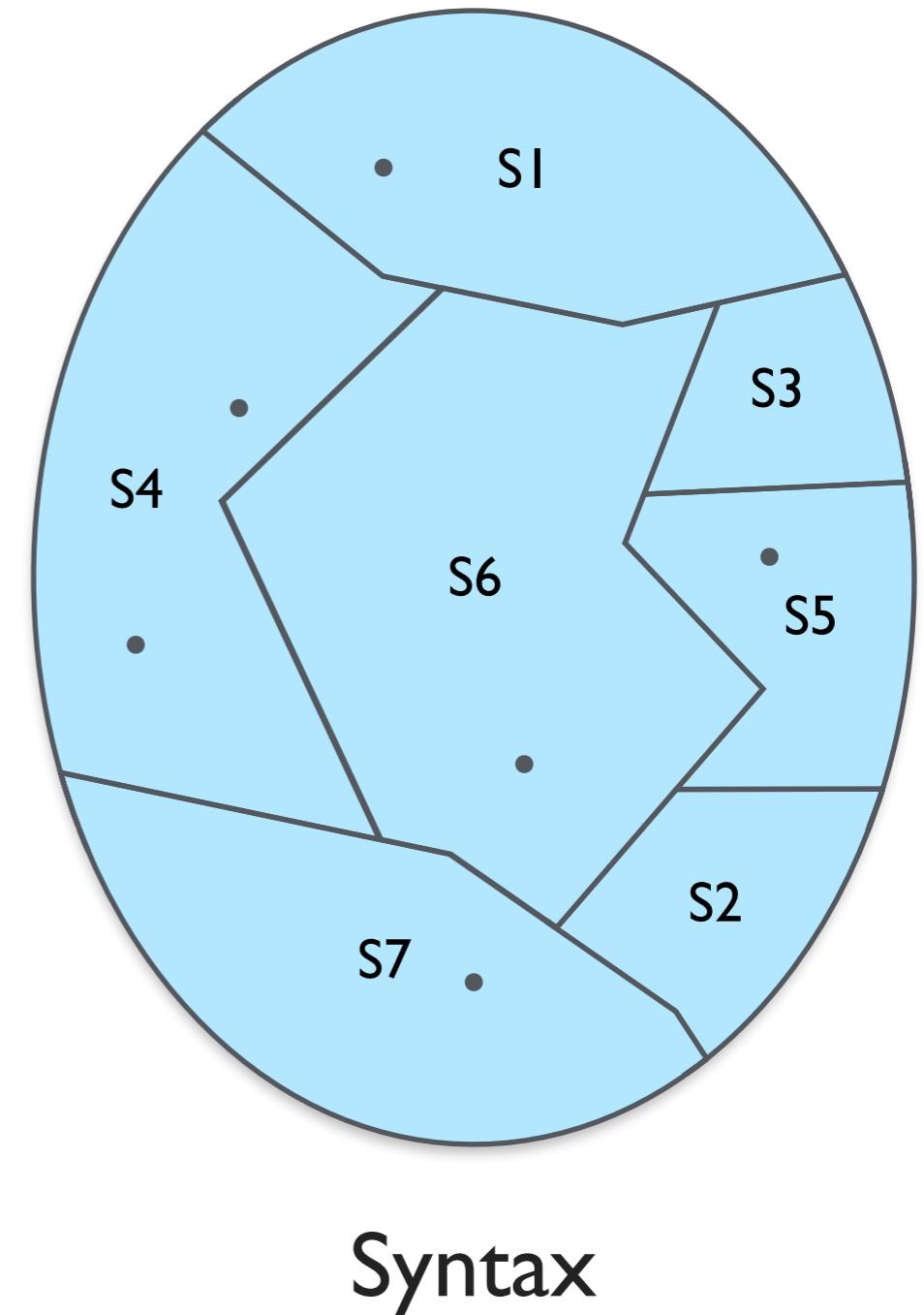
Syntax

Metasketches express structure and strategy

1. Search order is critical
2. Desire optimal solutions

A metasketch contains:

1. structured candidate space (\mathcal{S}, \leq)
2. cost function (κ)

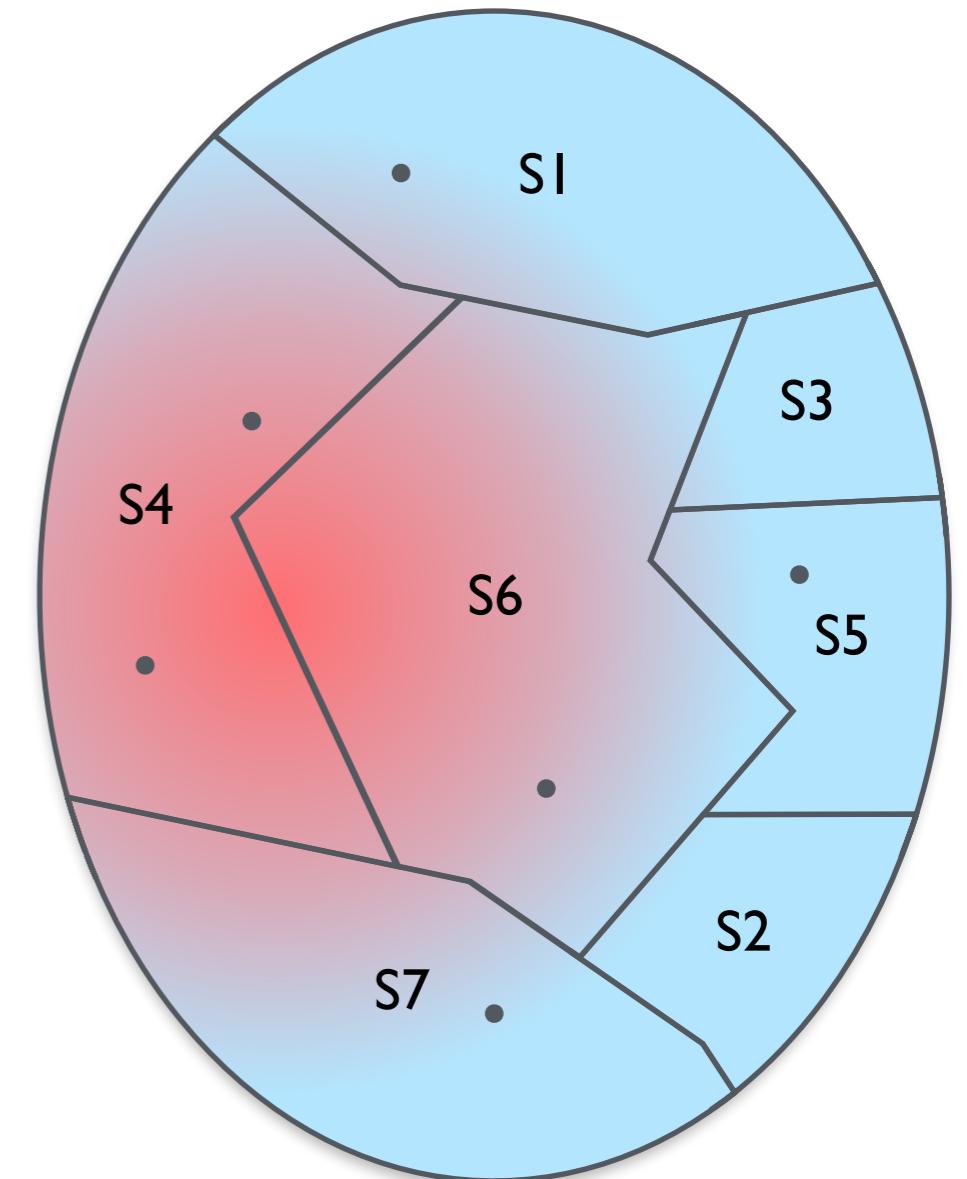


Metasketches express structure and strategy

1. Search order is critical
2. Desire optimal solutions

A metasketch contains:

1. structured candidate space (\mathcal{S}, \leq)
2. cost function (κ)
3. gradient function (g)



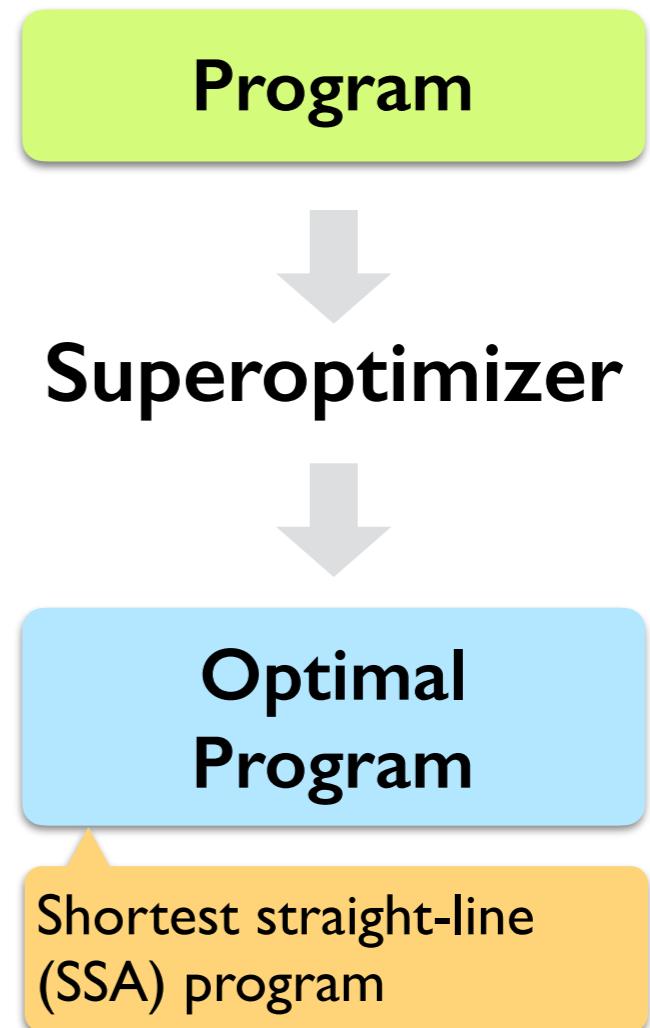
Syntax

Metasketches express structure and strategy

1. Search order is critical
2. Desire optimal solutions

A metasketch contains:

1. structured candidate space (\mathcal{S}, \leq)
2. cost function (κ)
3. gradient function (g)



Metasketches express structure and strategy

I. structured candidate space (\mathcal{S}, \leqslant)

2. cost function (κ)

3. gradient function (g)

Metasketches express structure and strategy

I. structured candidate space (\mathcal{S}, \leqslant)

A fragmentation of the candidate space, and an ordering on those fragments.

2. cost function (κ)

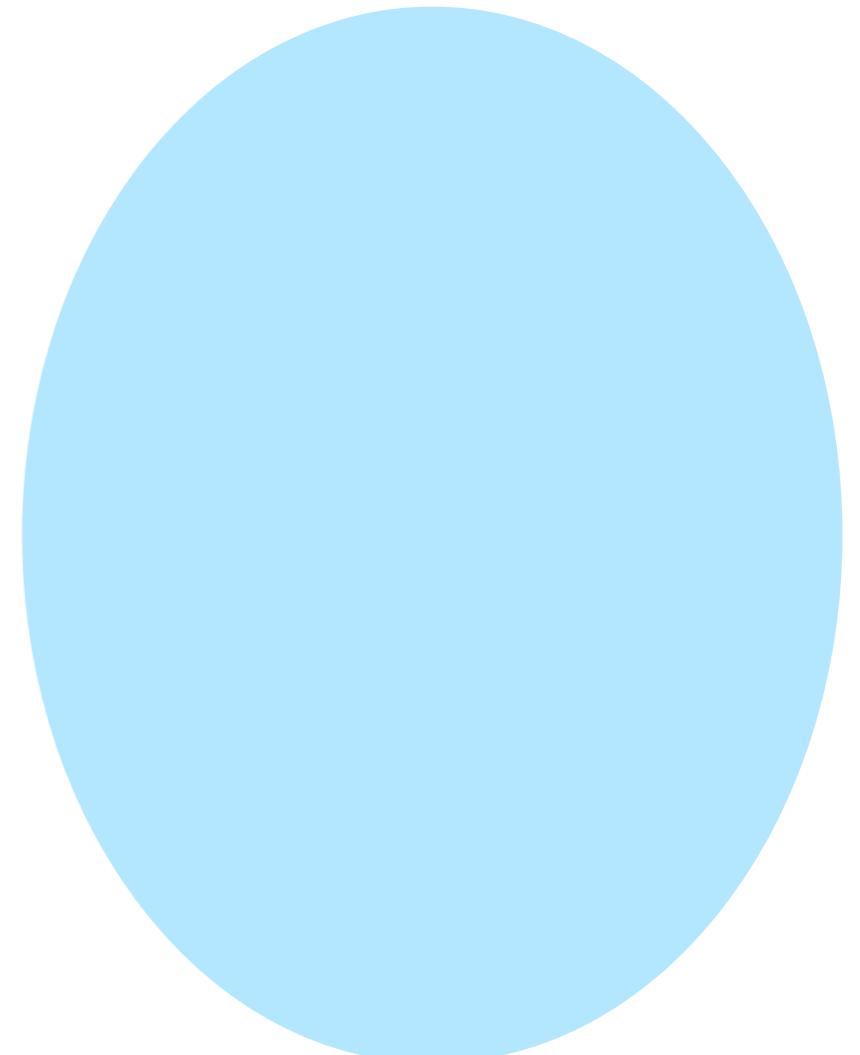
3. gradient function (g)

Metasketches express structure and strategy

1. structured candidate space (\mathcal{S}, \leq)

A fragmentation of the candidate space, and an ordering on those fragments.

\mathcal{S} = set of all SSA programs



2. cost function (κ)

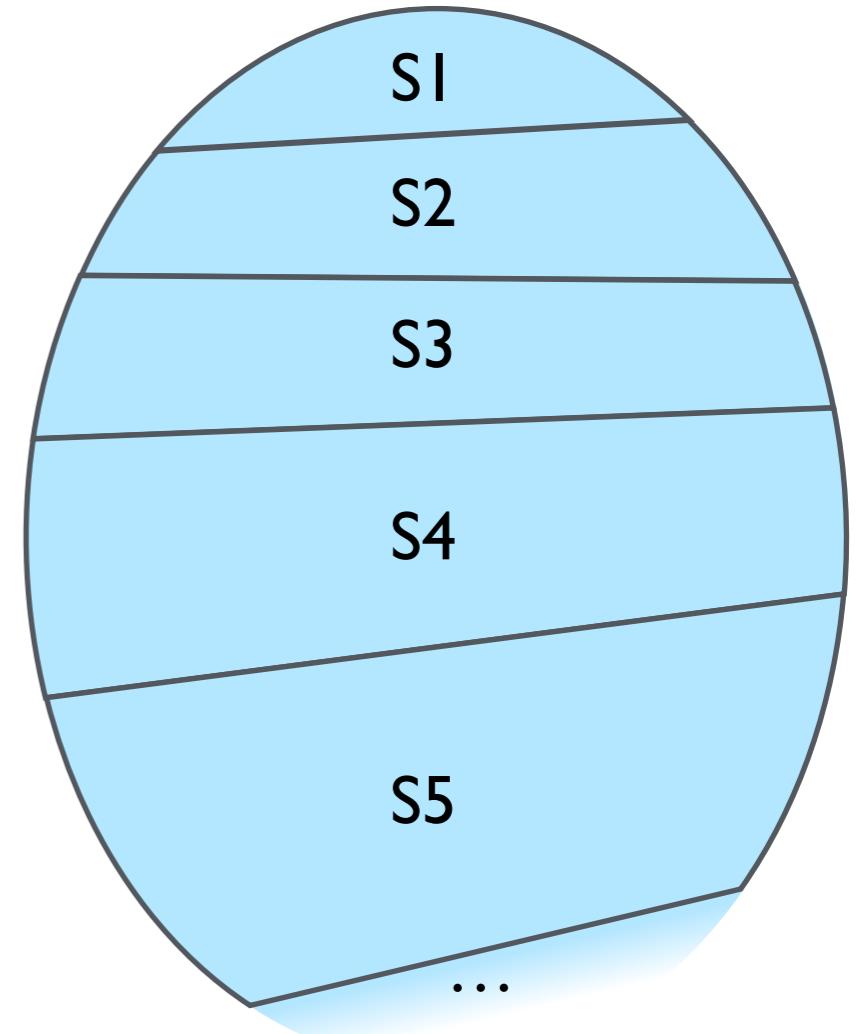
3. gradient function (g)

Metasketches express structure and strategy

I. structured candidate space (\mathcal{S} , \leq)

A fragmentation of the candidate space, and an ordering on those fragments.

\mathcal{S} = set of all SSA programs



2. cost function (κ)

3. gradient function (g)

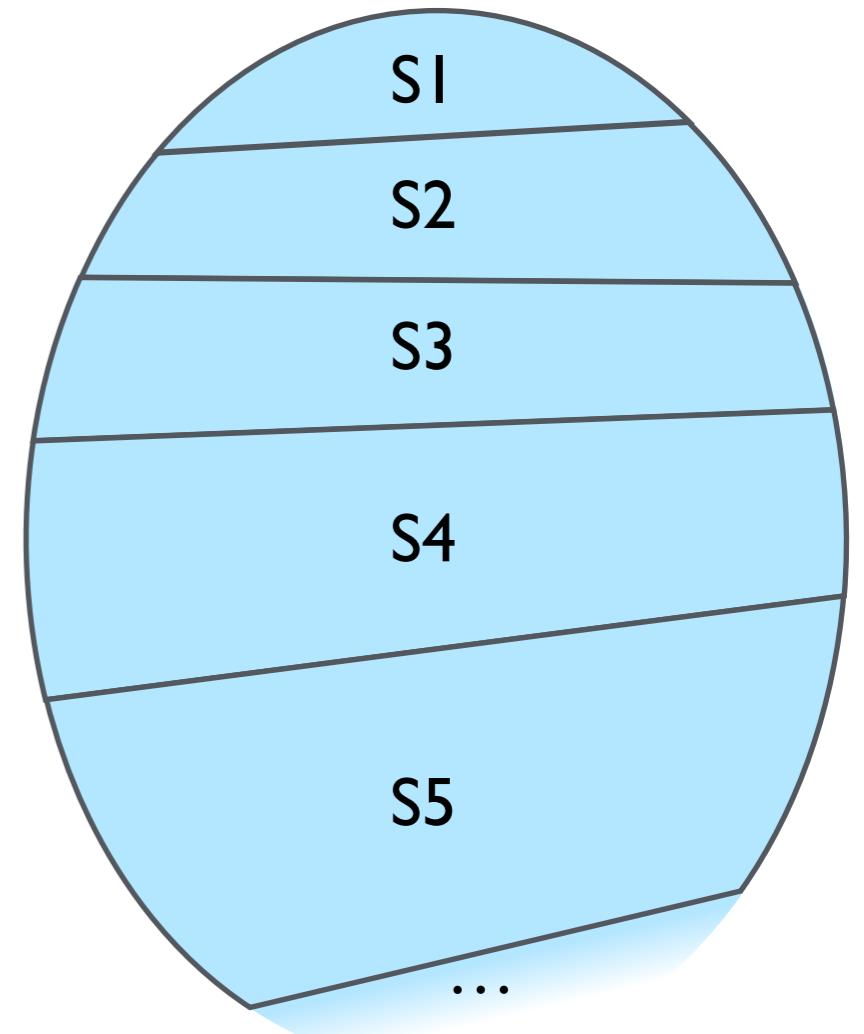
Metasketches express structure and strategy

I. structured candidate space (\mathcal{S}, \leq)

- a countable set \mathcal{S} of sketches
- a total order \leq on \mathcal{S}

A fragmentation of the candidate space, and an ordering on those fragments.

\mathcal{S} = set of all SSA programs



2. cost function (κ)

3. gradient function (g)

Metasketches express structure and strategy

I. structured candidate space (\mathcal{S}, \leq)

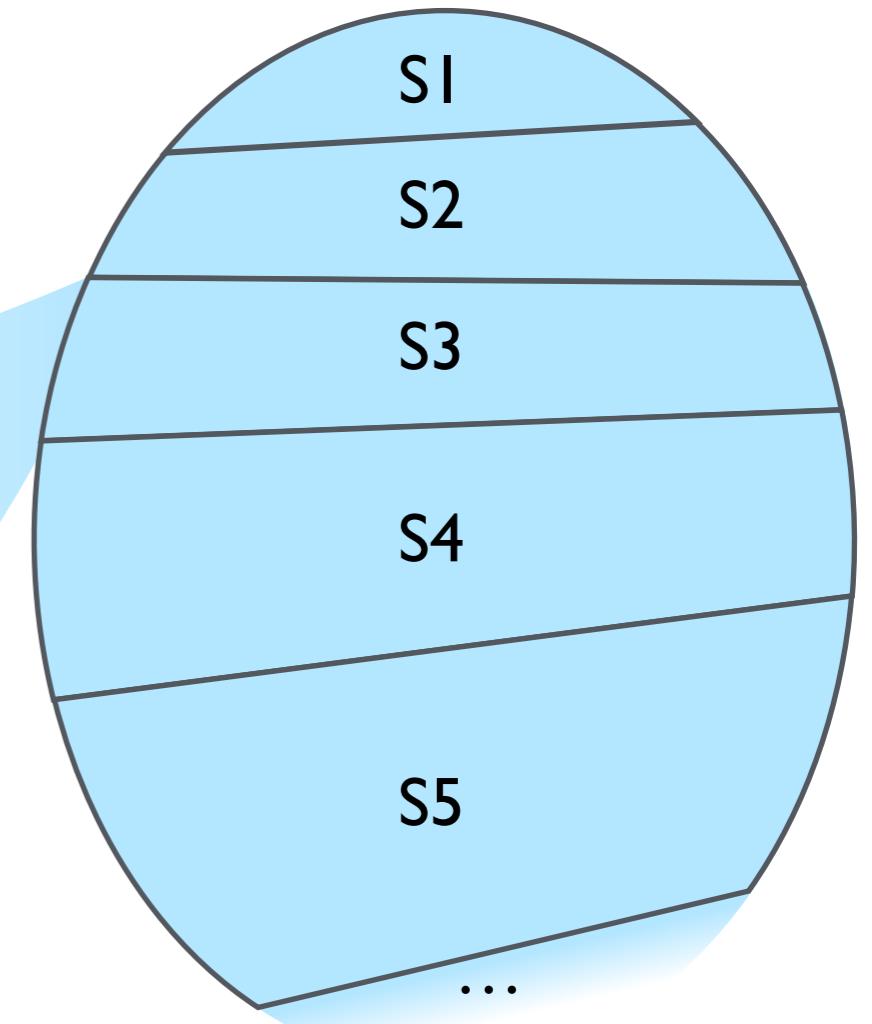
- a countable set \mathcal{S} of sketches
- a total order \leq on \mathcal{S}

A fragmentation of the candidate space, and an ordering on those fragments.

S_3 (SSA programs of length 3)

```
def f(x):
    r1 = ??op(??{x})
    r2 = ??op(??{x, r1})
    r3 = ??op(??{x, r1, r2})
    return r3
```

$\mathcal{S} = \text{set of all SSA programs}$



2. cost function (κ)

3. gradient function (g)

Metasketches express structure and strategy

I. structured candidate space (\mathcal{S} , \leq)

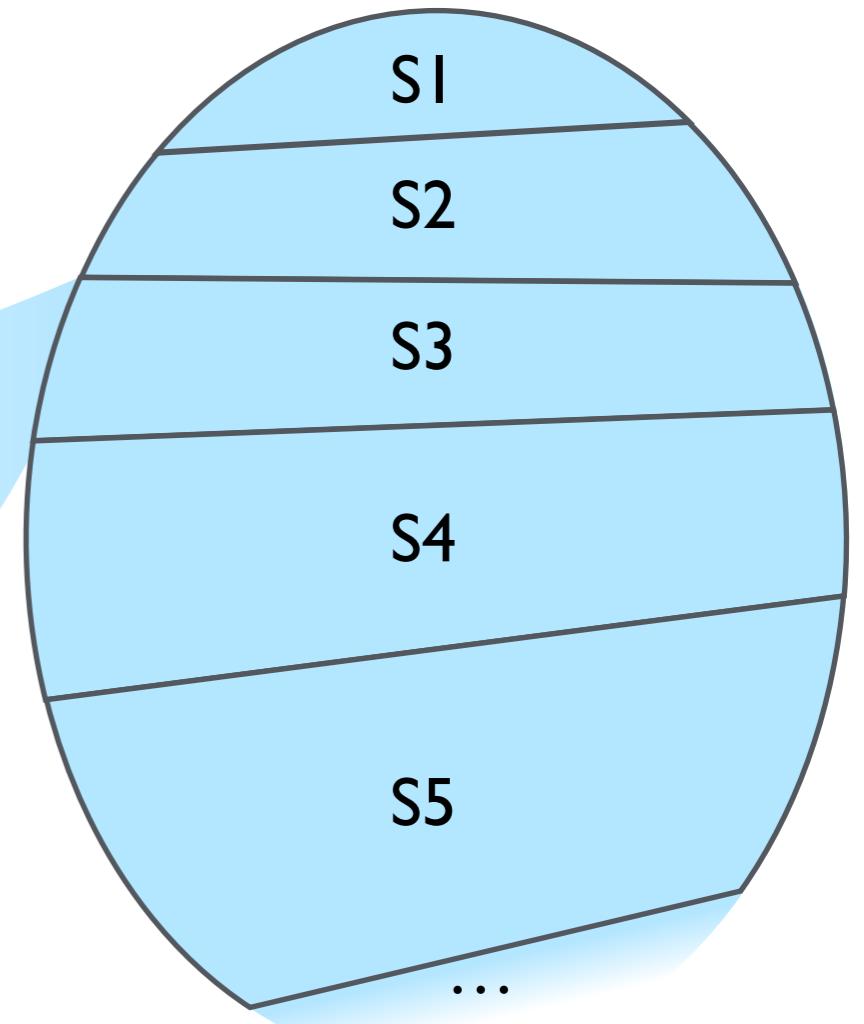
- a countable set \mathcal{S} of sketches
- a total order \leq on \mathcal{S}

A fragmentation of the candidate space, and an ordering on those fragments.

S_3 (SSA programs of length 3)

```
def f(x): +, -, <, if, ...
    r1 = ??op(??{x})
    r2 = ??op(??{x, r1})
    r3 = ??op(??{x, r1, r2})
    return r3
```

\mathcal{S} = set of all SSA programs



2. cost function (κ)

3. gradient function (g)

Metasketches express structure and strategy

I. structured candidate space (\mathcal{S} , \leq)

- a countable set \mathcal{S} of sketches
- a total order \leq on \mathcal{S}

A fragmentation of the candidate space, and an ordering on those fragments.

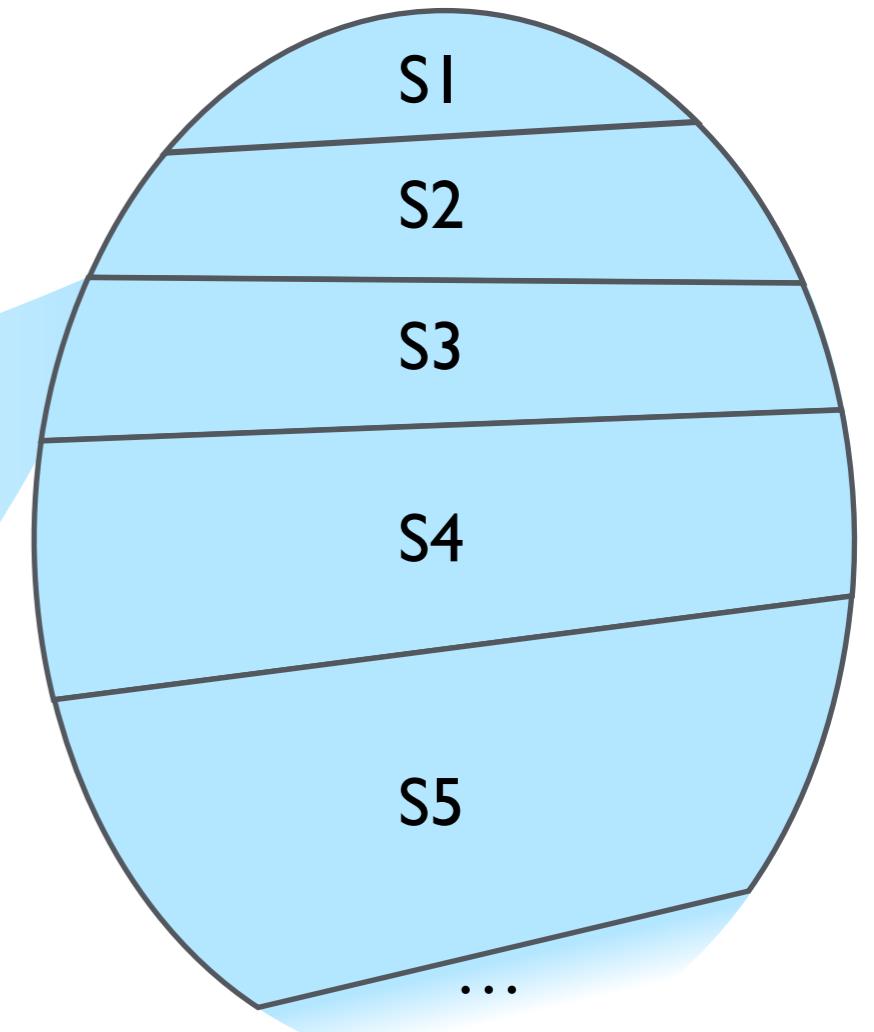
S_3 (SSA programs of length 3)

```
def f(x): +, -, <, if, ...
    r1 = ??op(??{x})
    r2 = ??op(??{x, r1})
    r3 = ??op(??{x, r1, r2})
    return r3
```

+, -, <, if, ...

Vars & constants

\mathcal{S} = set of all SSA programs



2. cost function (κ)

3. gradient function (g)

Metasketches express structure and strategy

I. structured candidate space (\mathcal{S} , \leq)

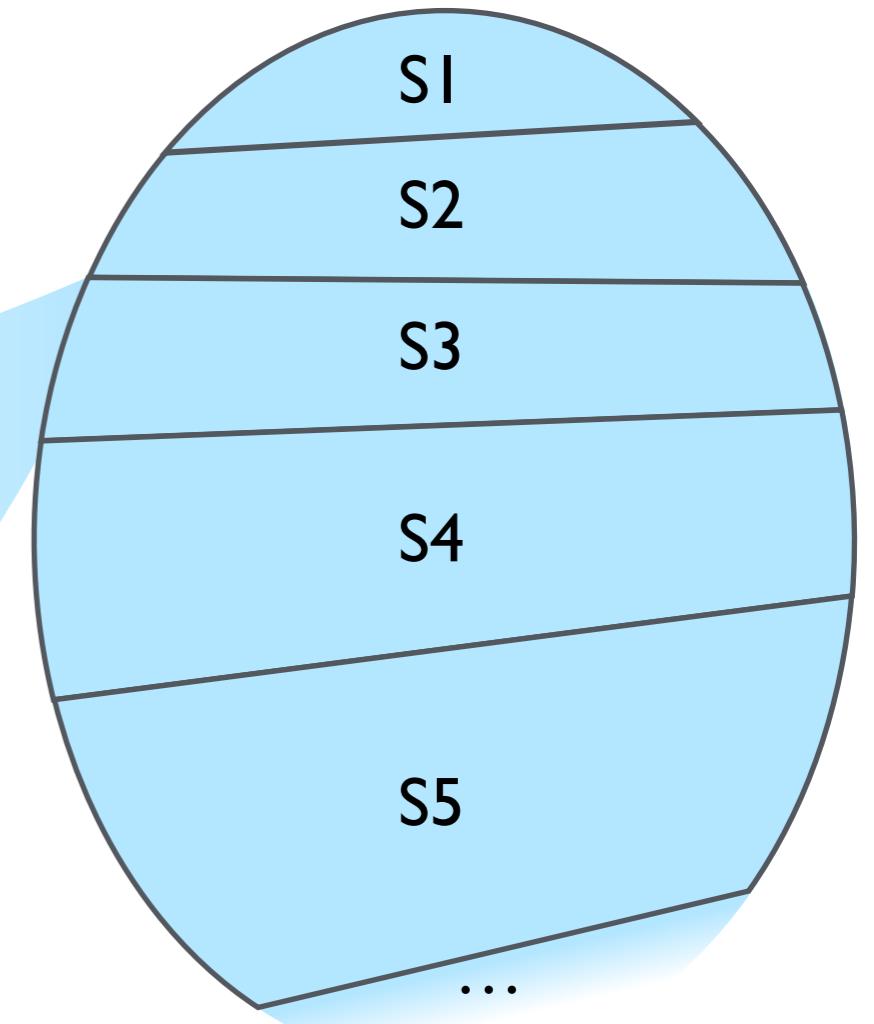
- a countable set \mathcal{S} of sketches
- a total order \leq on \mathcal{S}

A fragmentation of the candidate space, and an ordering on those fragments.

S_3 (SSA programs of length 3)

```
def f(x):
    r1 = ??op(??{x})
    r2 = ??op(??{x, r1})
    r3 = ??op(??{x, r1, r2})
    return r3
```

\mathcal{S} = set of all SSA programs



2. cost function (κ)

3. gradient function (g)

Metasketches express structure and strategy

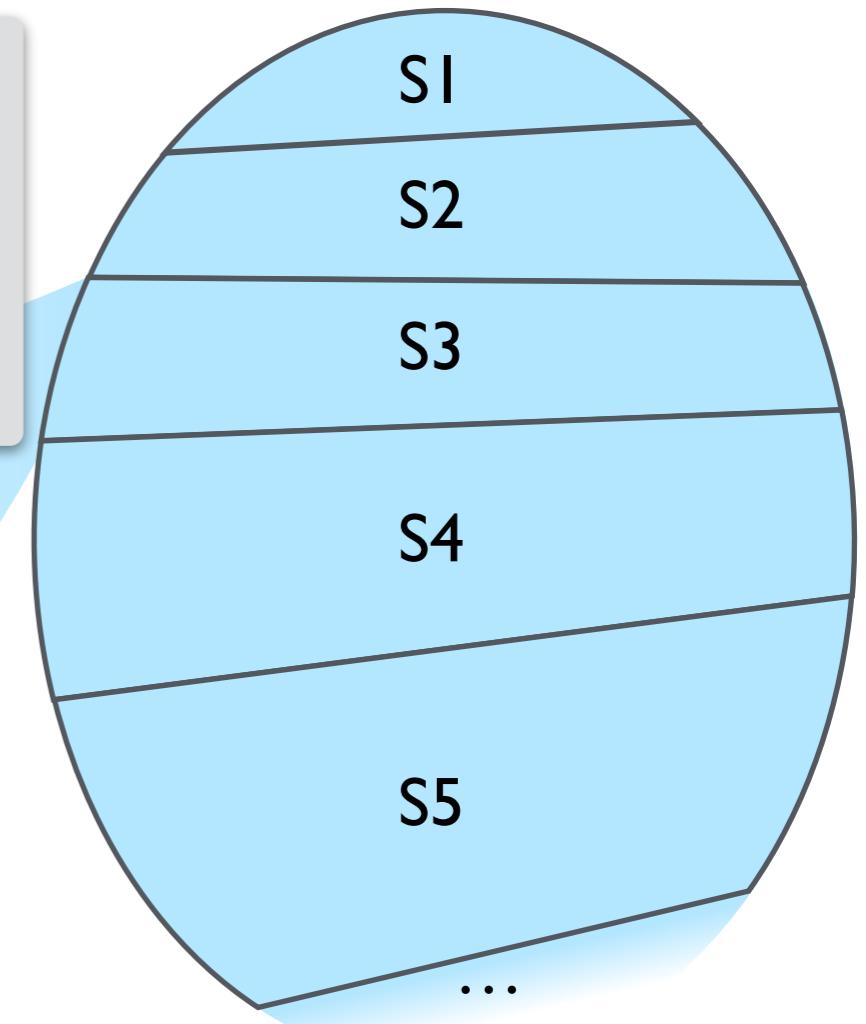
I. structured candidate space (\mathcal{S} , \leq)

- a countable set \mathcal{S} of sketches
- a total order \leq on \mathcal{S}

A fragmentation of the candidate space, and an ordering on those fragments.

Ordering expresses high-level search strategy.

\mathcal{S} = set of all SSA programs



S₃ (SSA programs of length 3)

```
def f(x):
    r1 = ??op(??{x})
    r2 = ??op(??{x, r1})
    r3 = ??op(??{x, r1, r2})
    return r3
```

2. cost function (κ)

3. gradient function (g)

Metasketches express structure and strategy

I. structured candidate space (\mathcal{S} , \leq)

- a countable set \mathcal{S} of sketches
- a total order \leq on \mathcal{S}

A fragmentation of the candidate space, and an ordering on those fragments.

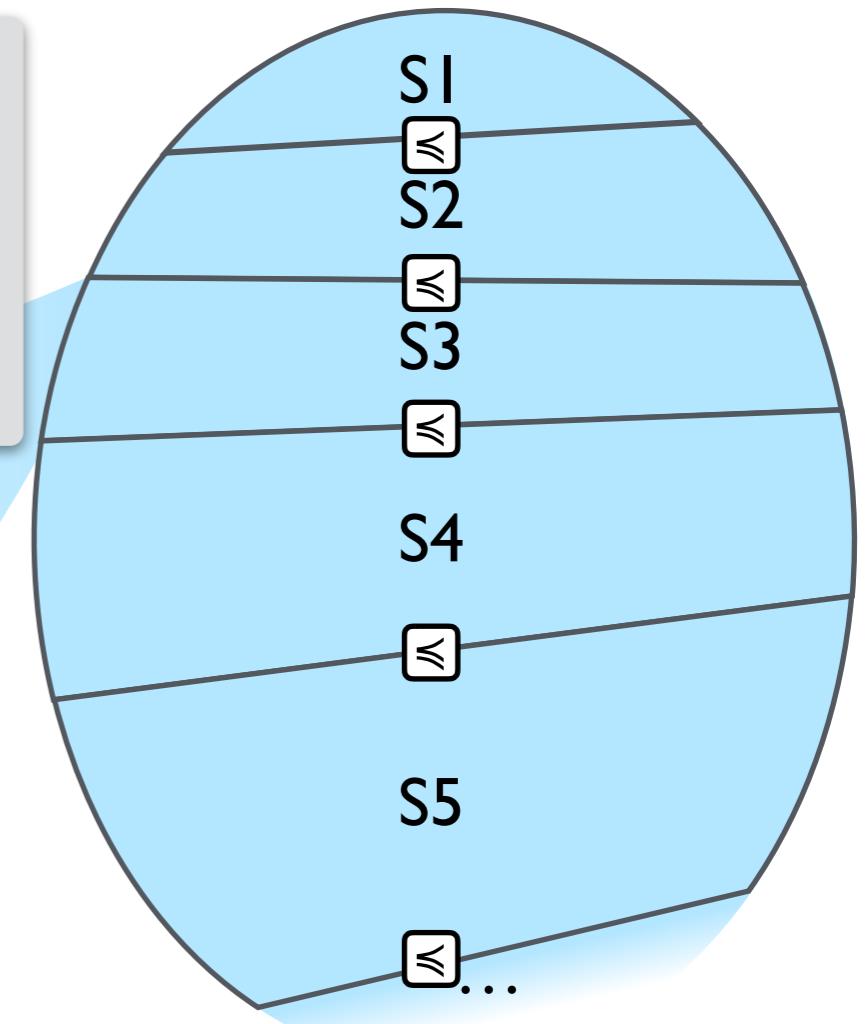
S_3 (SSA programs of length 3)

```
def f(x):
    r1 = ??op(??{x})
    r2 = ??op(??{x, r1})
    r3 = ??op(??{x, r1, r2})
    return r3
```

Ordering expresses high-level search strategy.

Here, \leq expresses iterative deepening.

\mathcal{S} = set of all SSA programs



2. cost function (κ)

3. gradient function (g)

Metasketches express structure and strategy

I. structured candidate space (\mathcal{S} , \leq)

- a countable set \mathcal{S} of sketches
- a total order \leq on \mathcal{S}

A fragmentation of the candidate space, and an ordering on those fragments.

\mathcal{S} = set of all SSA programs

```
def f(x):  
    r1 = ??op(??{x})  
    return r1
```

```
def f(x):  
    r1 = ??op(??{x})  
    r2 = ??op(??{x, r1})  
    return r2
```

```
def f(x):  
    r1 = ??op(??{x})  
    r2 = ??op(??{x, r1})  
    r3 = ??op(??{x, r1, r2})  
    return r3
```

...

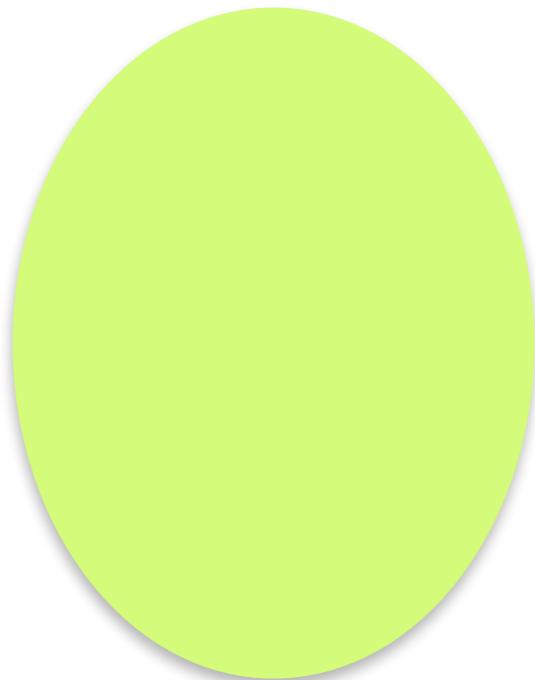
2. cost function (κ)

3. gradient function (g)

Metasketches express structure and strategy

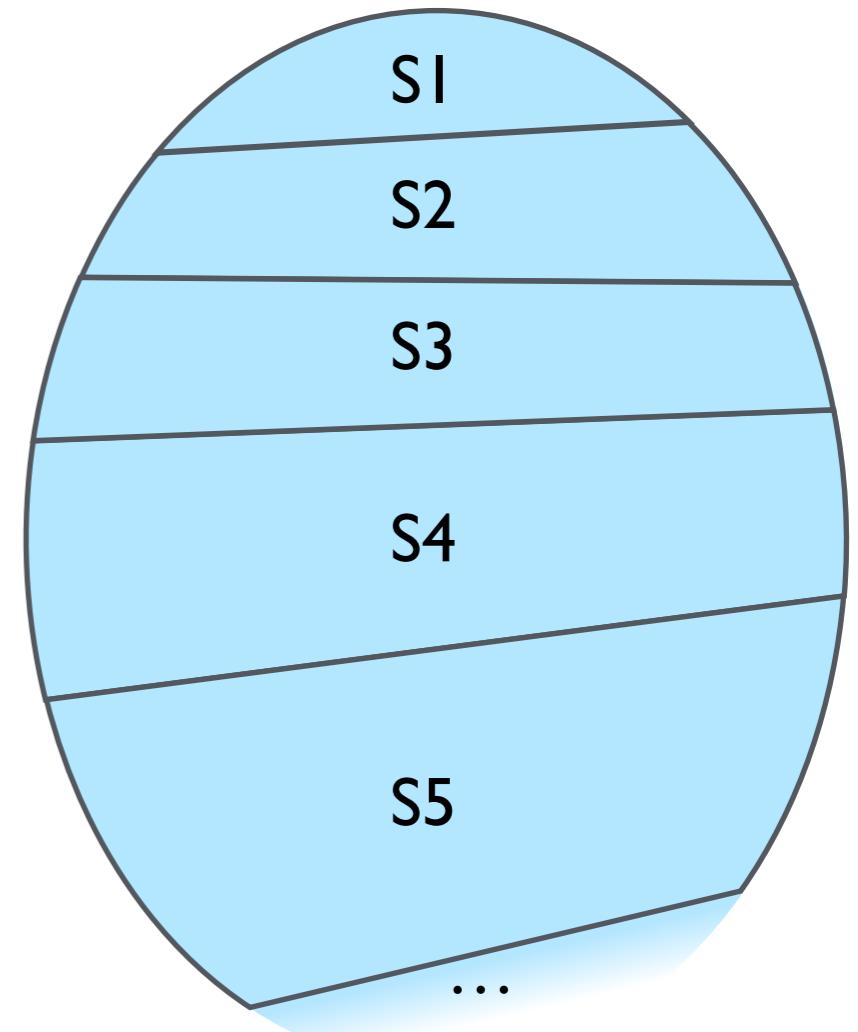
I. structured candidate space (\mathcal{S}, \leq)

- a countable set \mathcal{S} of sketches
- a total order \leq on \mathcal{S}



Semantics

\mathcal{S} = set of all SSA programs



2. cost function (κ)

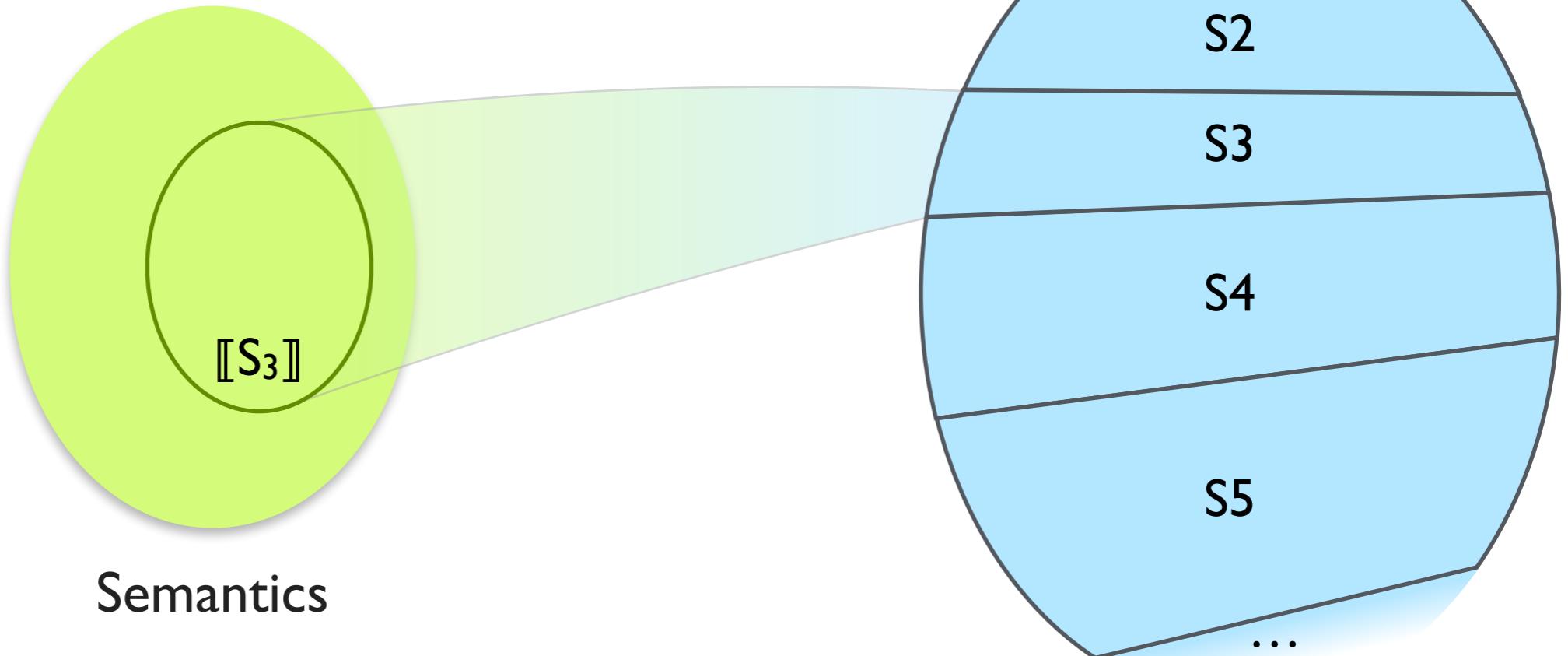
3. gradient function (g)

Metasketches express structure and strategy

I. structured candidate space (\mathcal{S}, \leq)

- a countable set \mathcal{S} of sketches
- a total order \leq on \mathcal{S}

$\mathcal{S} = \text{set of all SSA programs}$



2. cost function (κ)

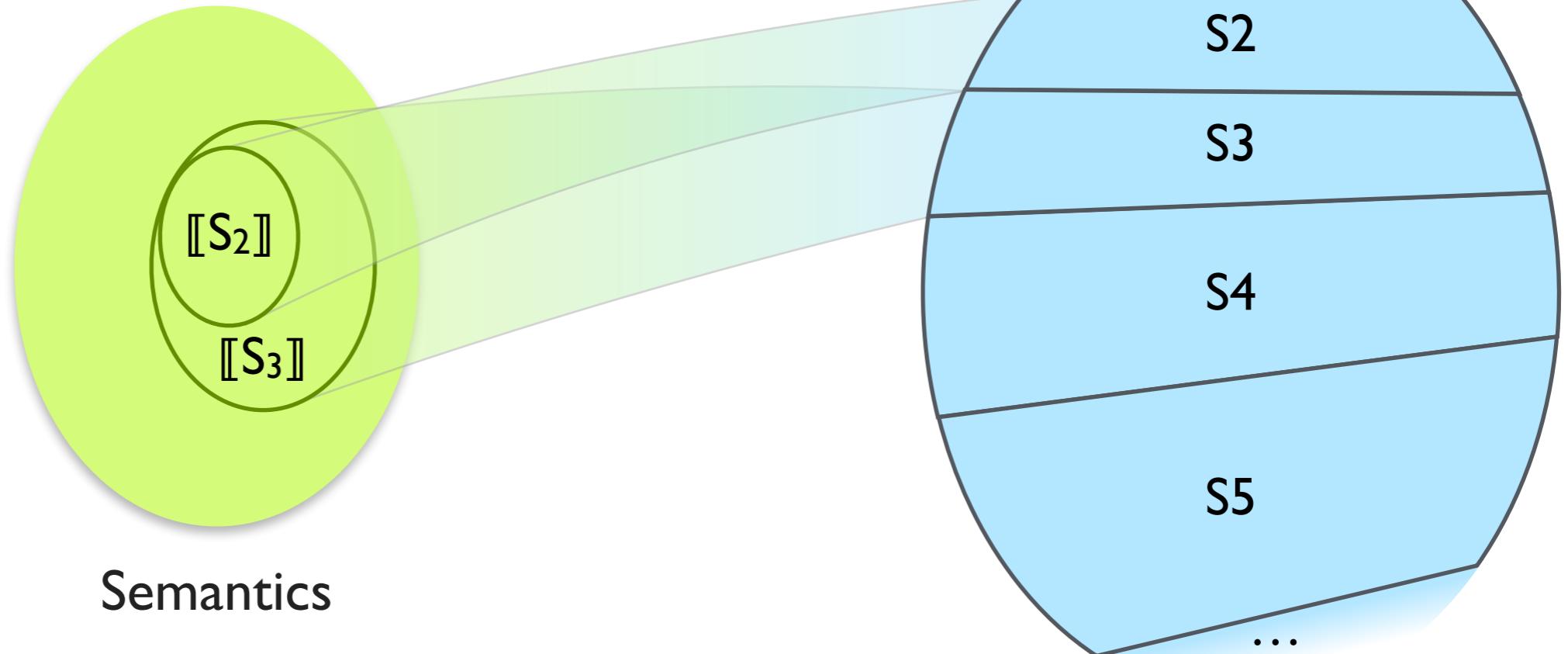
3. gradient function (g)

Metasketches express structure and strategy

I. structured candidate space (\mathcal{S}, \leq)

- a countable set \mathcal{S} of sketches
- a total order \leq on \mathcal{S}

$\mathcal{S} = \text{set of all SSA programs}$



2. cost function (κ)

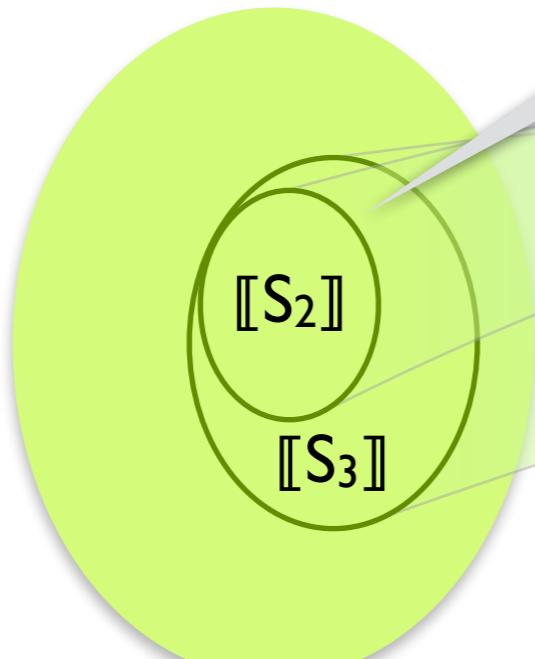
3. gradient function (g)

Metasketches express structure and strategy

I. structured candidate space (\mathcal{S}, \leq)

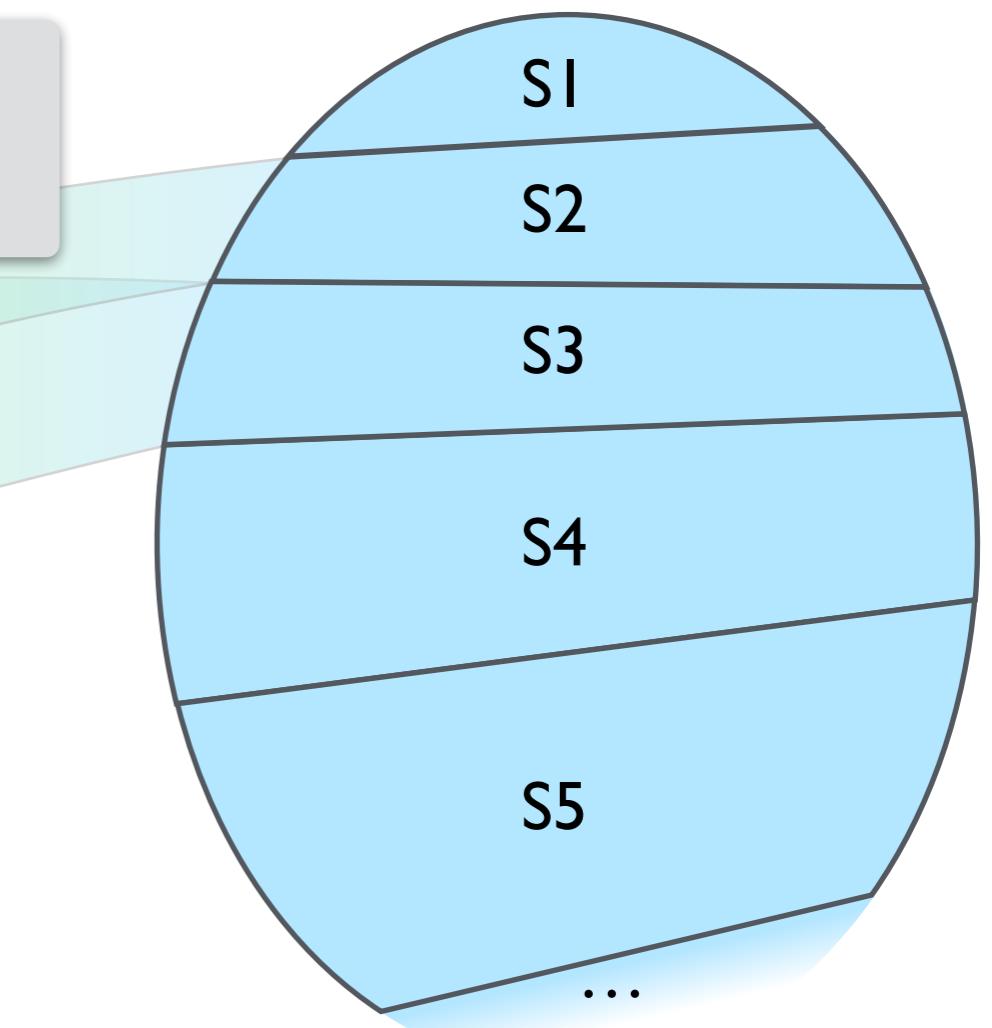
- a countable set \mathcal{S} of sketches
- a total order \leq on \mathcal{S}

$\mathcal{S} = \text{set of all SSA programs}$



Semantics

Semantic redundancy in the search space.



2. cost function (κ)

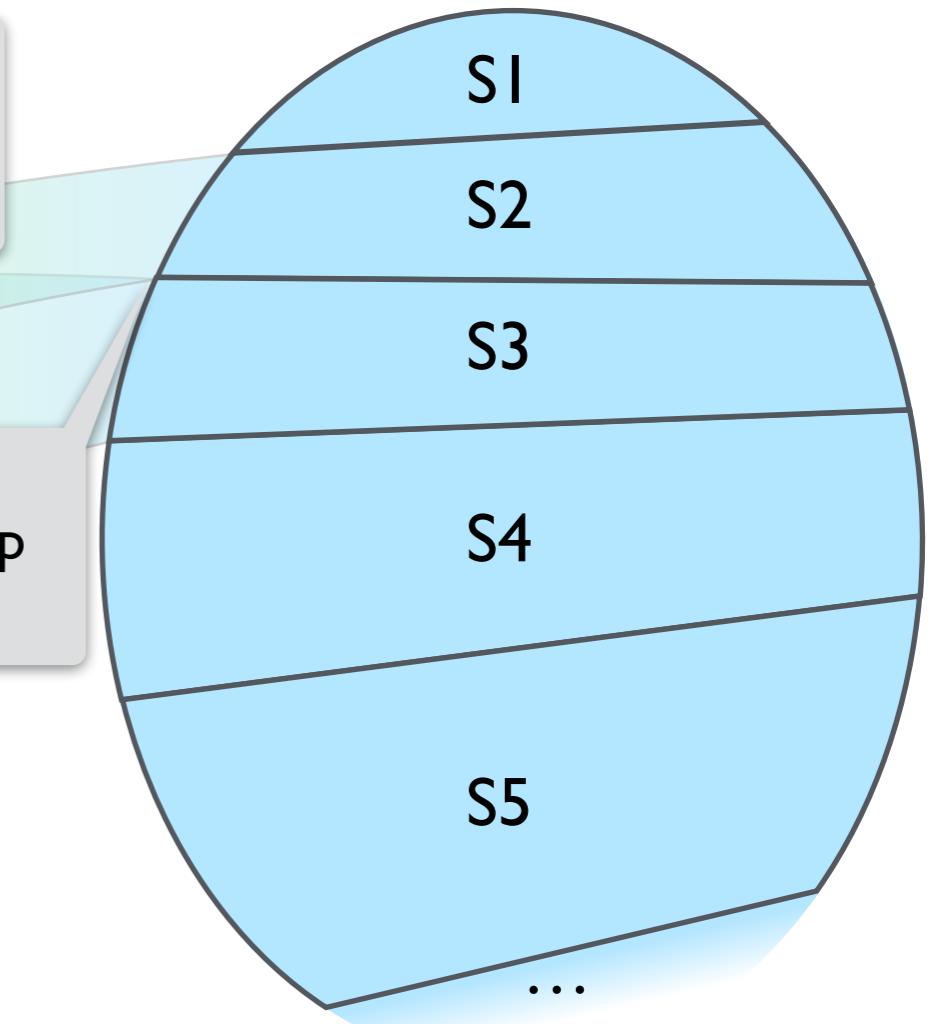
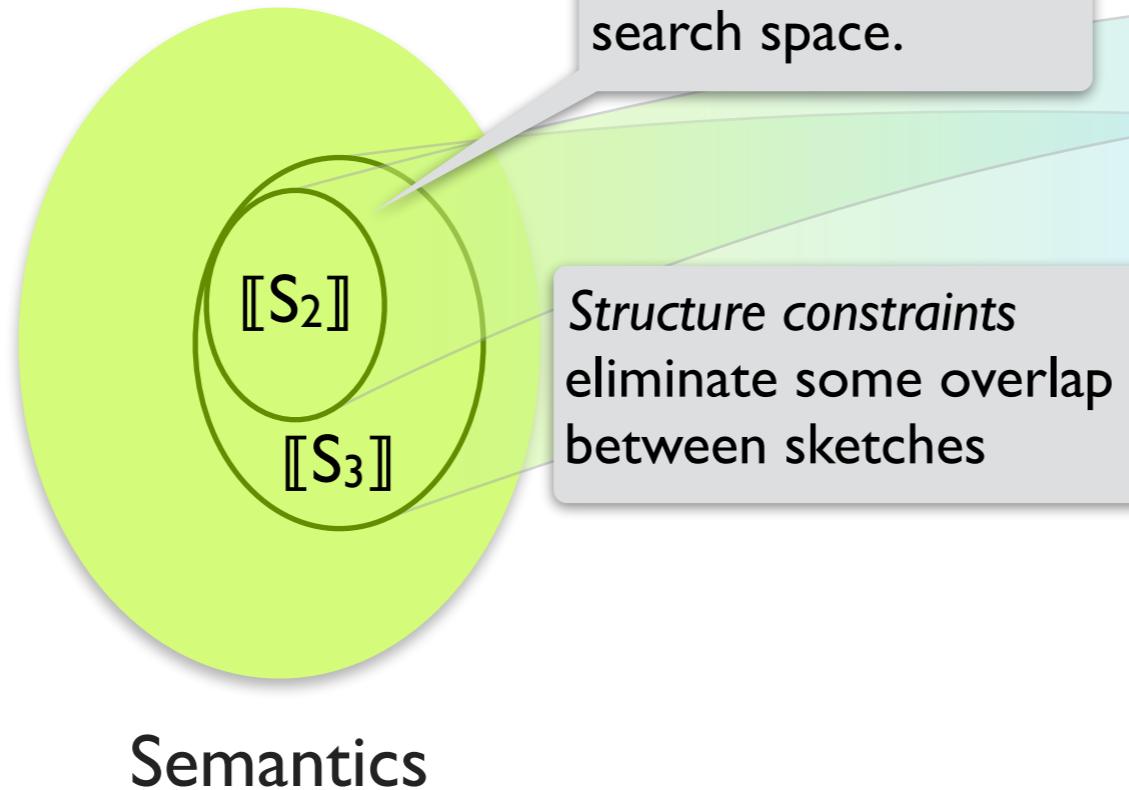
3. gradient function (g)

Metasketches express structure and strategy

I. structured candidate space (\mathcal{S} , \leq)

- a countable set \mathcal{S} of sketches
- a total order \leq on \mathcal{S}

\mathcal{S} = set of all SSA programs



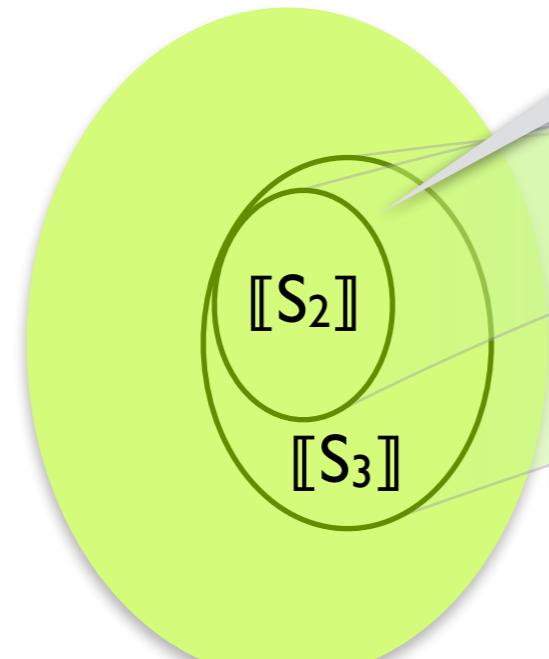
2. cost function (κ)

3. gradient function (g)

Metasketches express structure and strategy

I. structured candidate space (\mathcal{S} , \leq)

- a countable set \mathcal{S} of sketches
- a total order \leq on \mathcal{S}

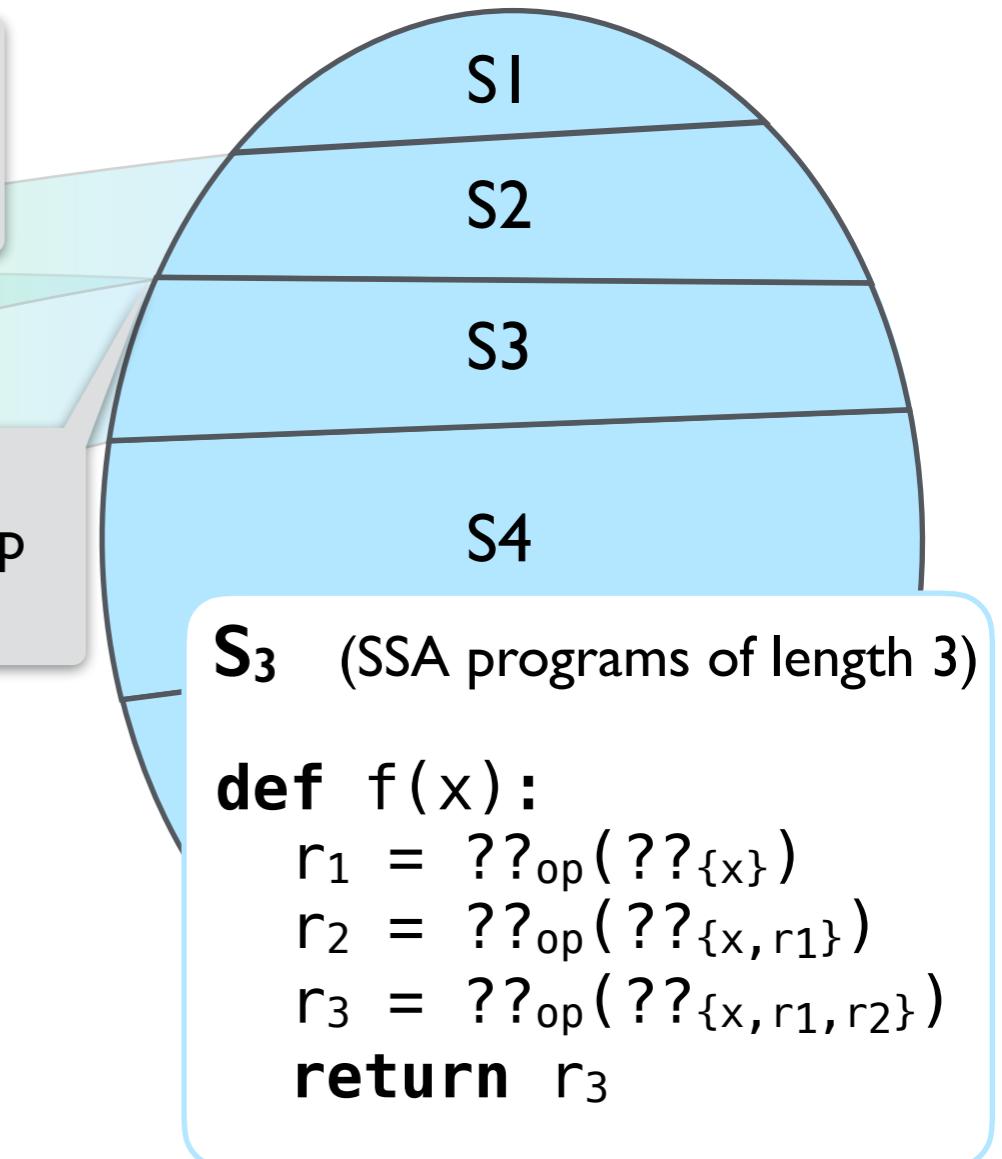


Semantics

Semantic redundancy in the search space.

Structure constraints eliminate some overlap between sketches

\mathcal{S} = set of all SSA programs



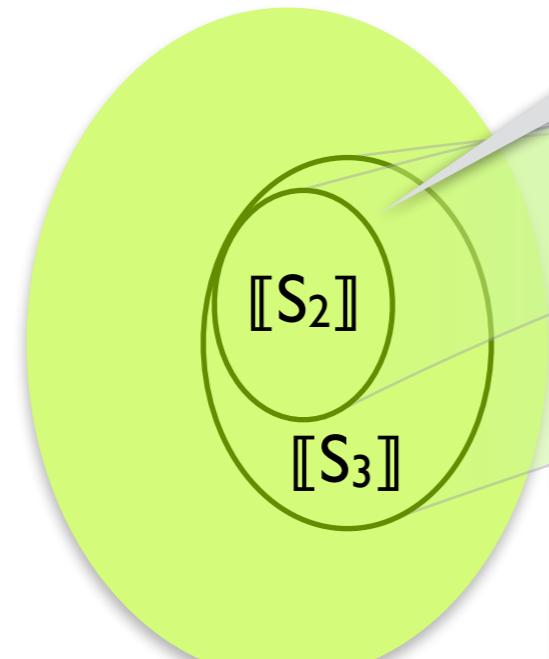
2. cost function (κ)

3. gradient function (g)

Metasketches express structure and strategy

I. structured candidate space (\mathcal{S} , \leq)

- a countable set \mathcal{S} of sketches
- a total order \leq on \mathcal{S}



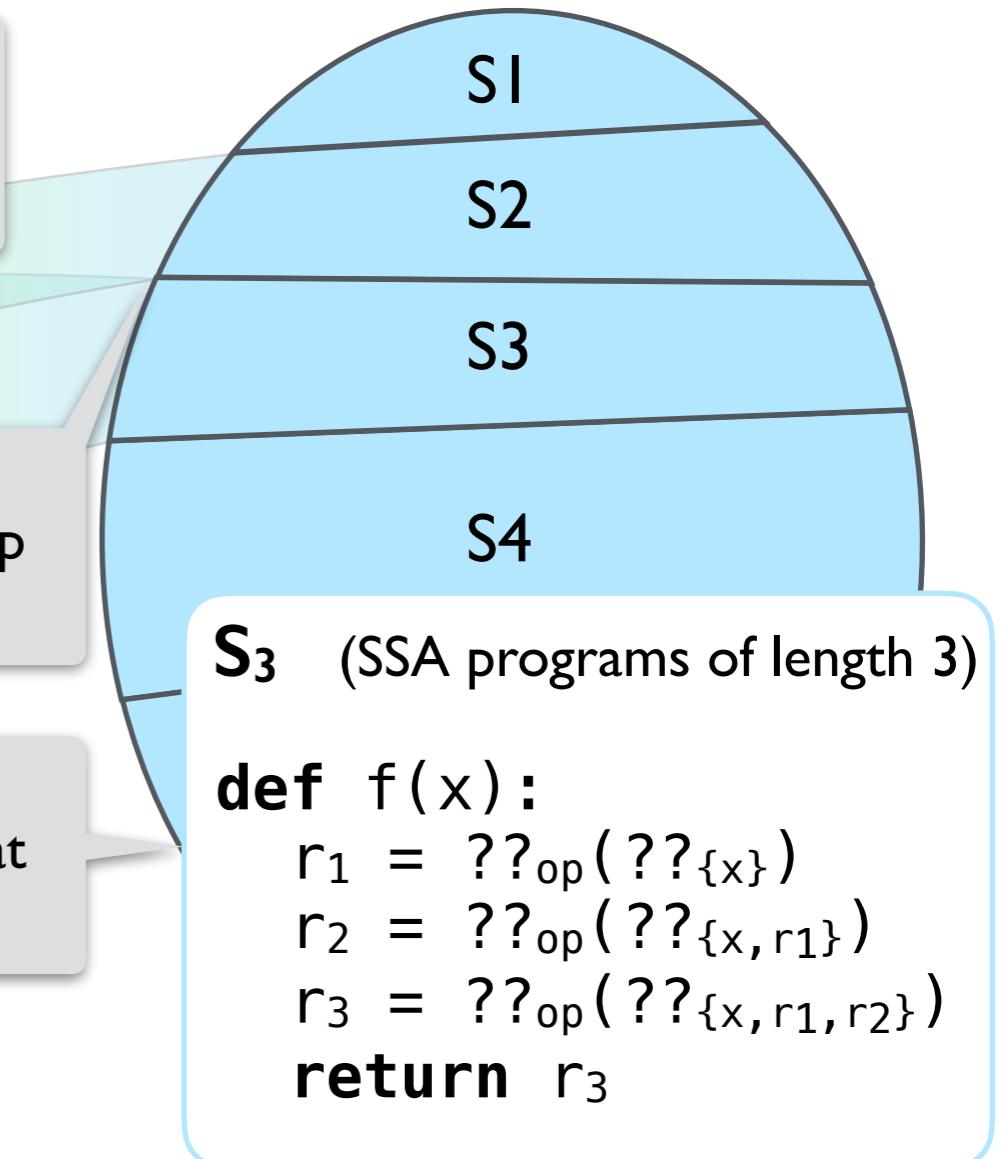
Semantics

Semantic redundancy in the search space.

Structure constraints eliminate some overlap between sketches

Eliminate dead-code redundancy: assert that each r_i is read

\mathcal{S} = set of all SSA programs



2. cost function (κ)

3. gradient function (g)

Cost functions rank candidate programs

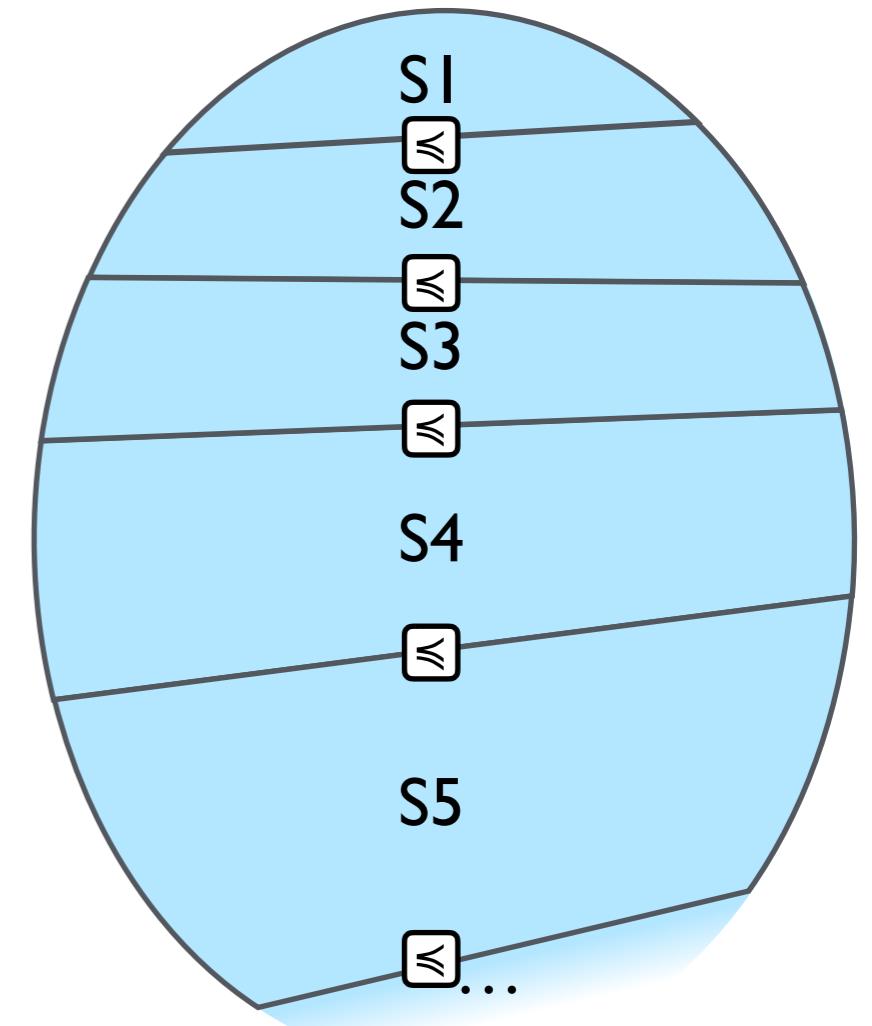
1. structured candidate space (\mathcal{S}, \leq)

2. cost function (κ)

$$\kappa : \mathcal{L} \rightarrow \mathbb{R}$$

assigns a numeric cost to each program in the language \mathcal{L}

\mathcal{S} = set of all SSA programs



3. gradient function (g)

Cost functions rank candidate programs

1. structured candidate space (\mathcal{S}, \leq)

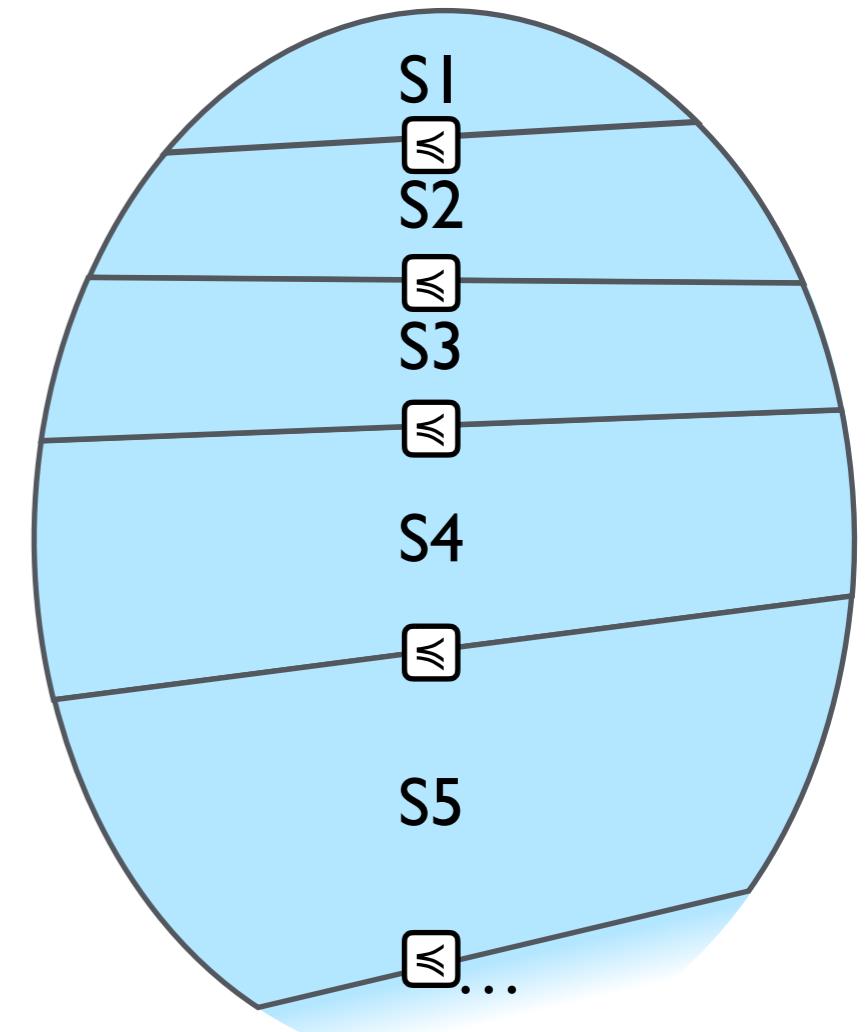
2. cost function (κ)

$$\kappa : \mathcal{L} \rightarrow \mathbb{R}$$

assigns a numeric cost to each program in the language \mathcal{L}

Cost functions can be based on both syntax and semantics (dynamic behavior)

\mathcal{S} = set of all SSA programs



3. gradient function (g)

Cost functions rank candidate programs

1. structured candidate space (\mathcal{S}, \leq)

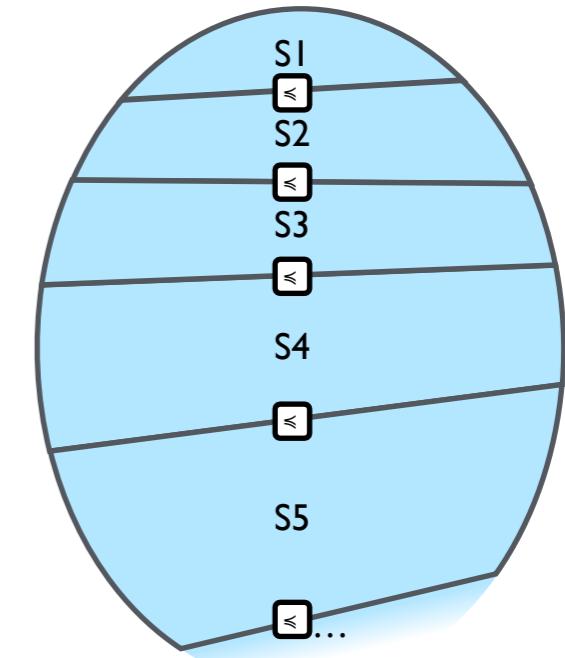
2. cost function (κ)

$$\kappa : \mathcal{L} \rightarrow \mathbb{R}$$

assigns a numeric cost to each program in the language \mathcal{L}

Cost functions can be based on both syntax and semantics (dynamic behavior)

\mathcal{S} = set of all SSA programs



$$\kappa(P) = i \quad \text{for } P \in S_i \in \mathcal{S}$$

The number of variables defined in P

3. gradient function (g)

Gradient functions provide cost structure

1. structured candidate space (\mathcal{S}, \leq)

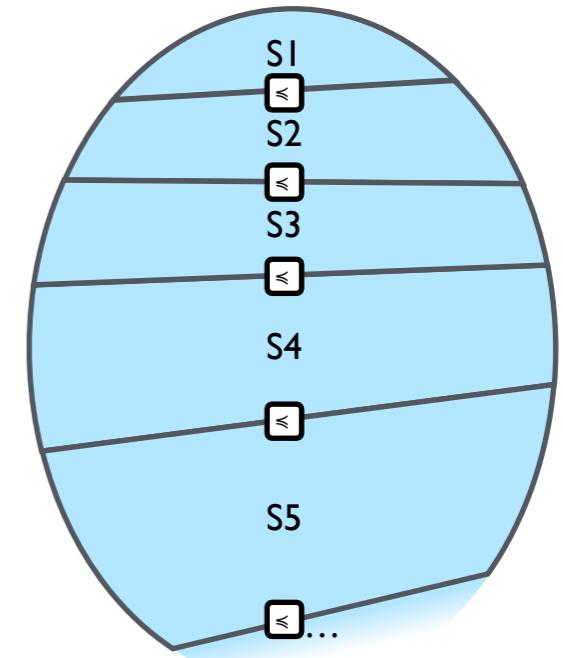
2. cost function (κ)

3. gradient function (g)

$$g : \mathbb{R} \rightarrow 2^{\mathcal{S}}$$

$g(c)$ is the set of sketches in \mathcal{S} that
may contain a solution P with $\kappa(P) < c$

\mathcal{S} = set of all SSA programs



$$\kappa(P) = i \quad \text{for } P \in S_i \in \mathcal{S}$$

Gradient functions provide cost structure

1. structured candidate space (\mathcal{S}, \leq)

2. cost function (κ)

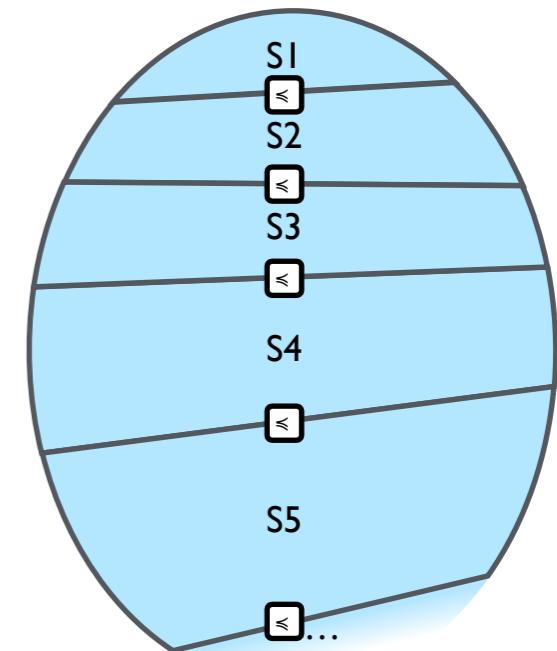
3. gradient function (g)

$$g : \mathbb{R} \rightarrow 2^{\mathcal{S}}$$

$g(c)$ is the set of sketches in \mathcal{S} that
may contain a solution P with $\kappa(P) < c$

The gradient function
overapproximates the
behavior of κ on \mathcal{S}

\mathcal{S} = set of all SSA programs



$$\kappa(P) = i \quad \text{for } P \in S_i \in \mathcal{S}$$

Gradient functions provide cost structure

1. structured candidate space (\mathcal{S}, \leq)

2. cost function (κ)

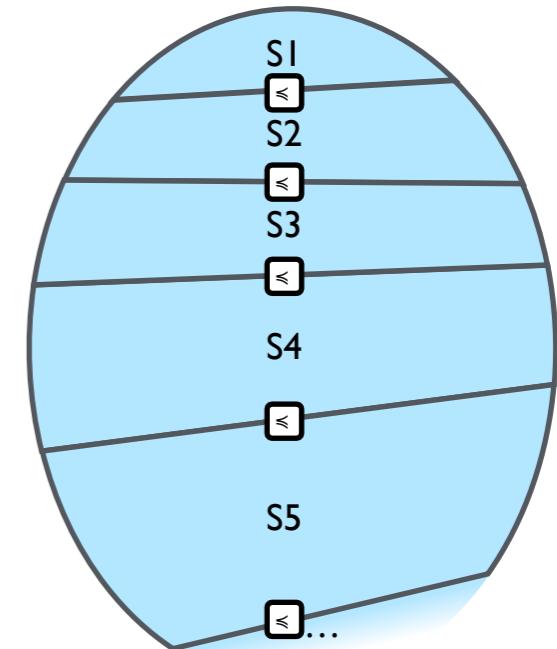
3. gradient function (g)

$$g : \mathbb{R} \rightarrow 2^{\mathcal{S}}$$

$g(c)$ is the set of sketches in \mathcal{S} that
may contain a solution P with $\kappa(P) < c$

The gradient function
overapproximates the
behavior of κ on \mathcal{S}

\mathcal{S} = set of all SSA programs



$$\kappa(P) = i \quad \text{for } P \in S_i \in \mathcal{S}$$
$$g(c) = \{ S_i \in \mathcal{S} \mid i < c \}$$

Gradient functions provide cost structure

1. structured candidate space (\mathcal{S}, \leq)

2. cost function (κ)

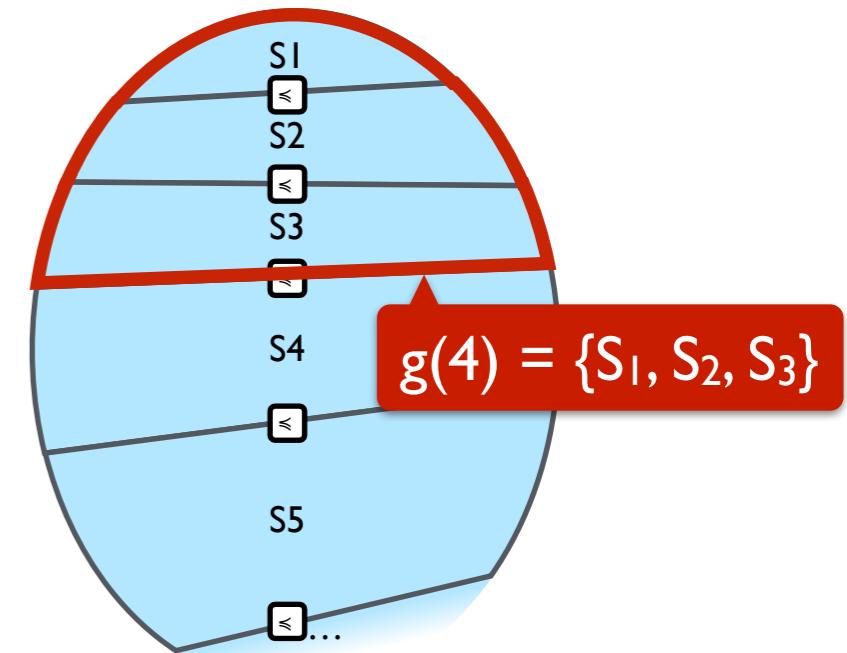
3. gradient function (g)

$$g : \mathbb{R} \rightarrow 2^{\mathcal{S}}$$

$g(c)$ is the set of sketches in \mathcal{S} that
may contain a solution P with $\kappa(P) < c$

The gradient function
overapproximates the
behavior of κ on \mathcal{S}

\mathcal{S} = set of all SSA programs



$$\kappa(P) = i \quad \text{for } P \in S_i \in \mathcal{S}$$

$$g(c) = \{ S_i \in \mathcal{S} \mid i < c \}$$

Gradient functions provide cost structure

1. structured candidate space (\mathcal{S}, \leq)

2. cost function (κ)

3. gradient function (g)

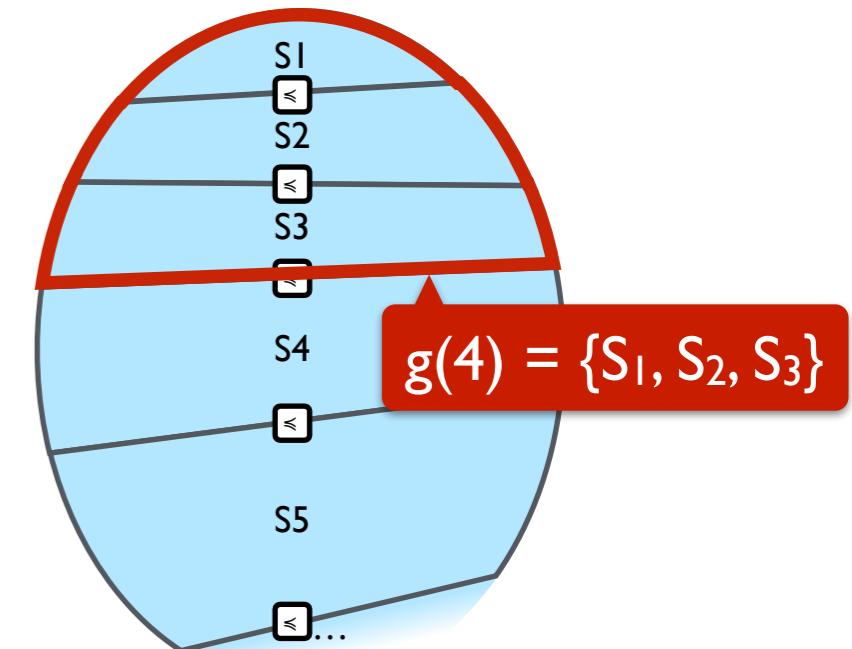
$$g : \mathbb{R} \rightarrow 2^{\mathcal{S}}$$

$g(c)$ is the set of sketches in \mathcal{S} that
may contain a solution P with $\kappa(P) < c$

The gradient function
overapproximates the
behavior of κ on \mathcal{S}

Always sound for g to
return all of \mathcal{S} if a tighter
bound is unavailable.

\mathcal{S} = set of all SSA programs



$$\kappa(P) = i \quad \text{for } P \in S_i \in \mathcal{S}$$
$$g(c) = \{ S_i \in \mathcal{S} \mid i < c \}$$

Gradient functions provide cost structure

1. structured candidate space (\mathcal{S}, \leq)

2. cost function (κ)

3. gradient function (g)

$$g : \mathbb{R} \rightarrow 2^{\mathcal{S}}$$

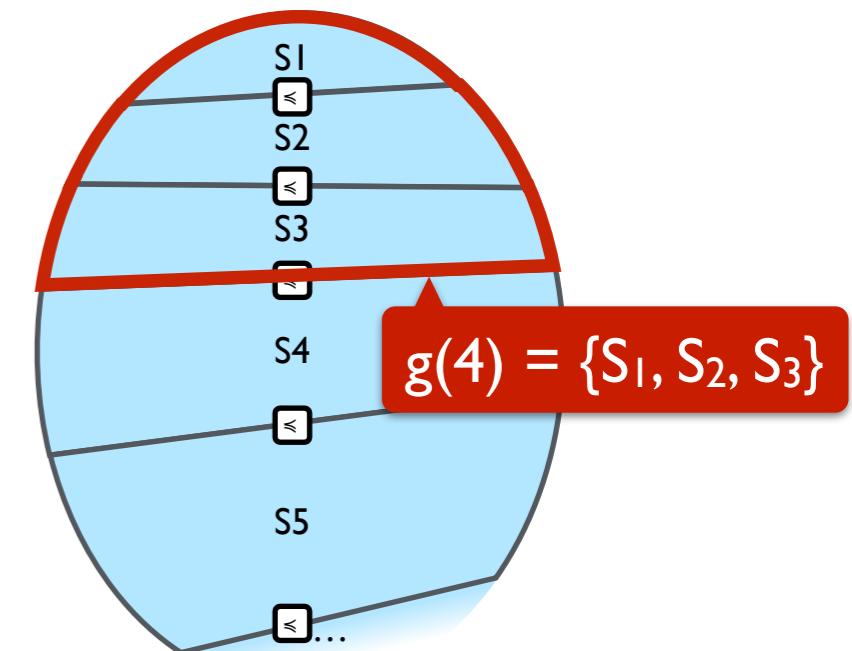
$g(c)$ is the set of sketches in \mathcal{S} that may contain a solution P with $\kappa(P) < c$

The gradient function overapproximates the behavior of κ on \mathcal{S}

Always sound for g to return all of \mathcal{S} if a tighter bound is unavailable.

$g(c)$ always being finite is sufficient (not necessary) to guarantee termination.

\mathcal{S} = set of all SSA programs



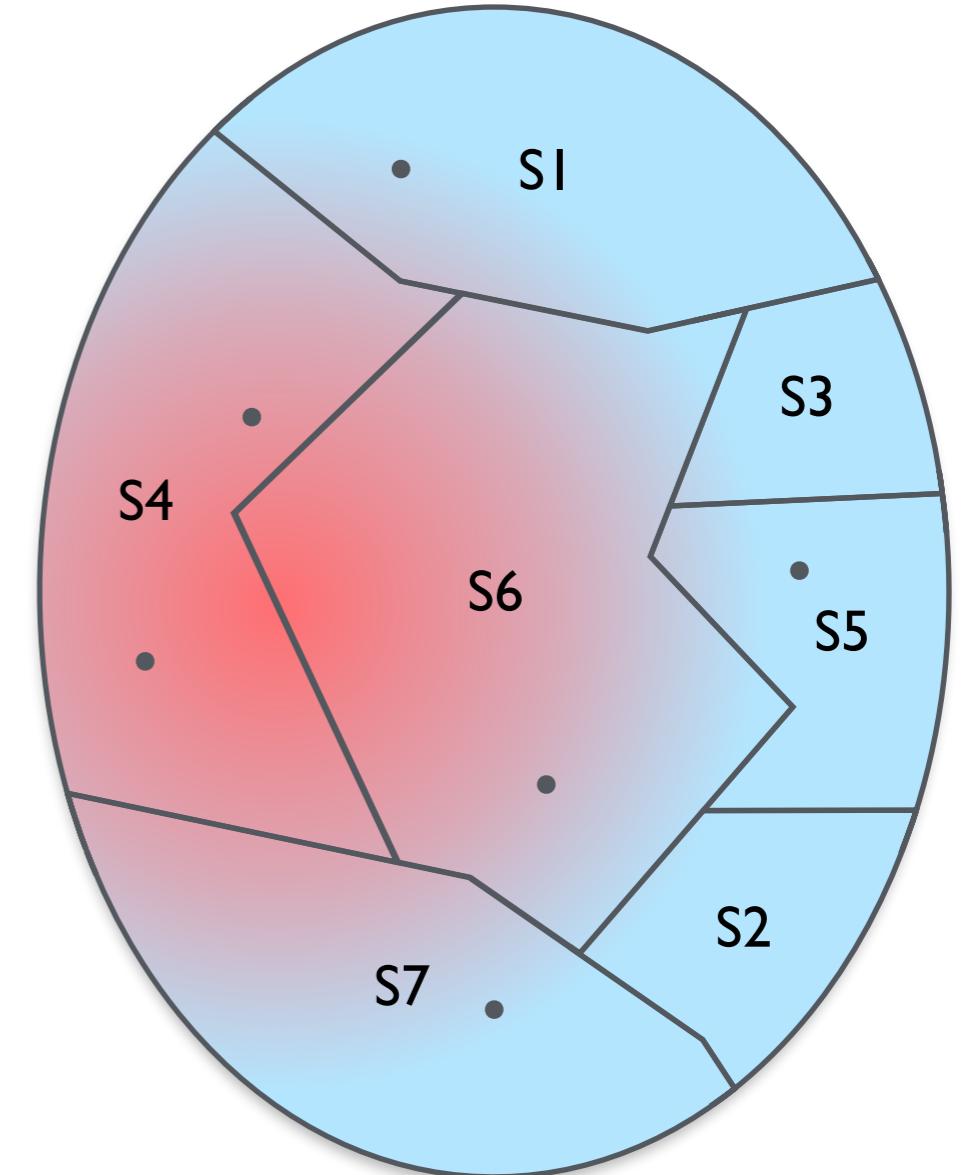
$$\kappa(P) = i \quad \text{for } P \in S_i \in \mathcal{S}$$

$$g(c) = \{ S_i \in \mathcal{S} \mid i < c \}$$

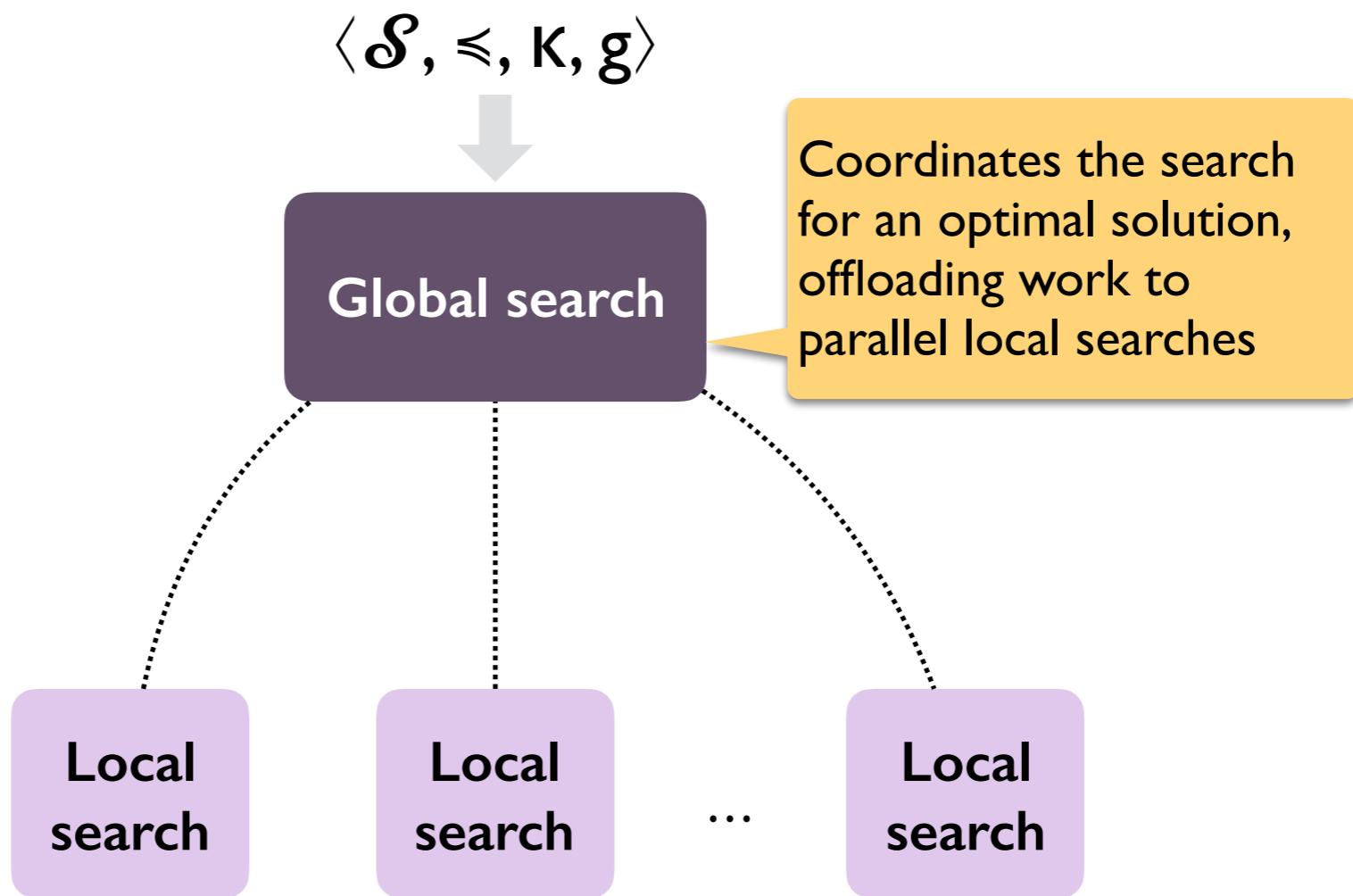
Metasketches express structure and strategy

A metasketch contains:

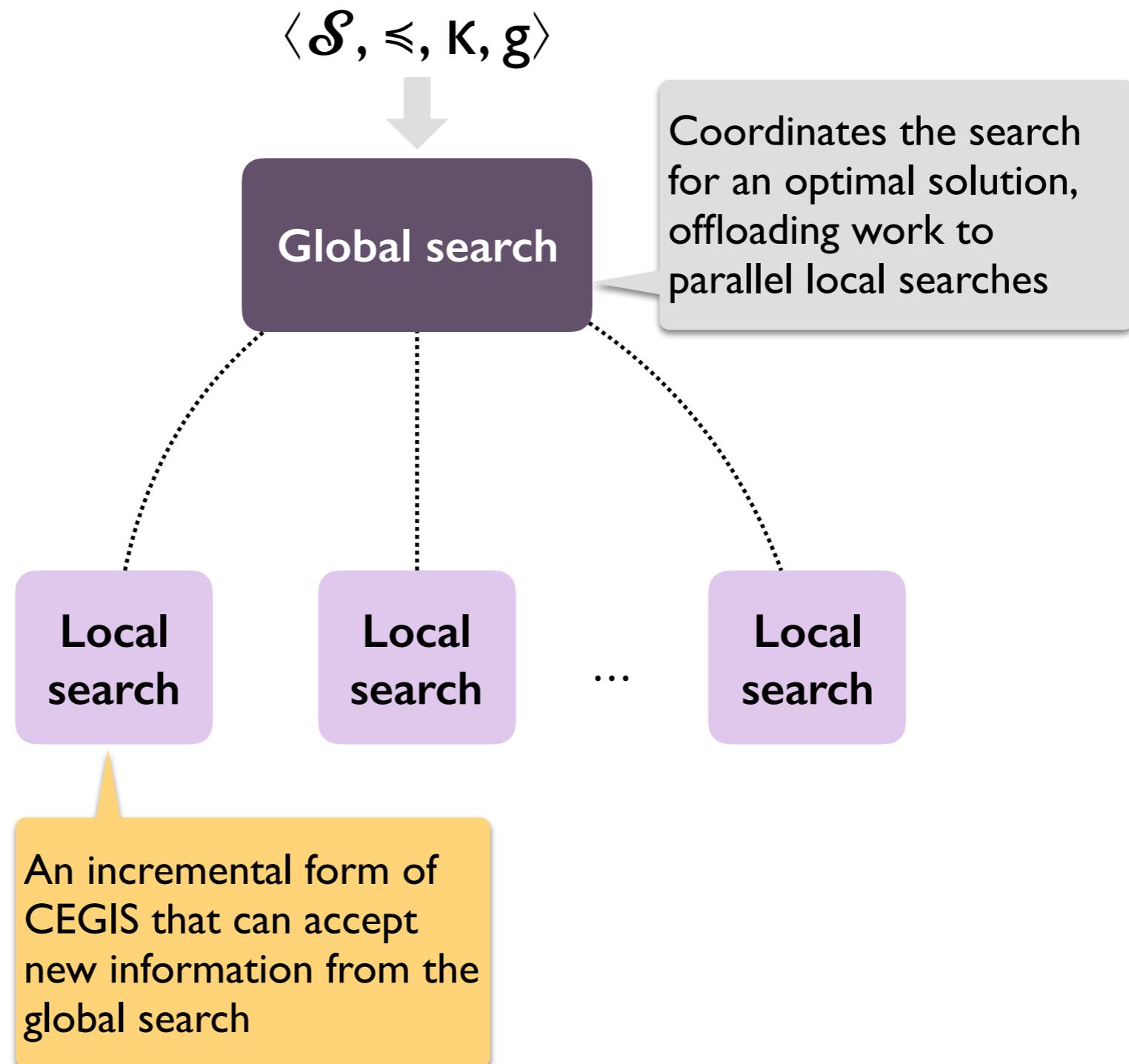
1. structured candidate space (\mathcal{S}, \leq)
2. cost function (κ)
3. gradient function (g)



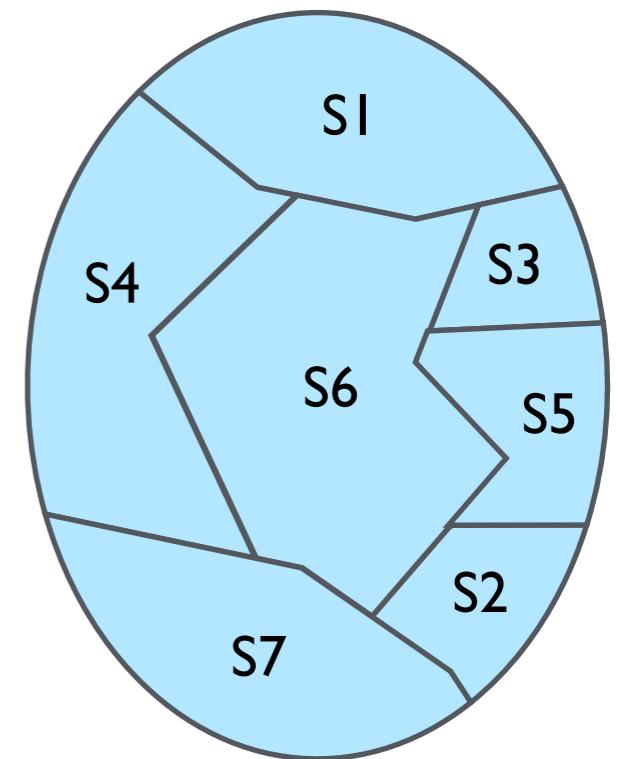
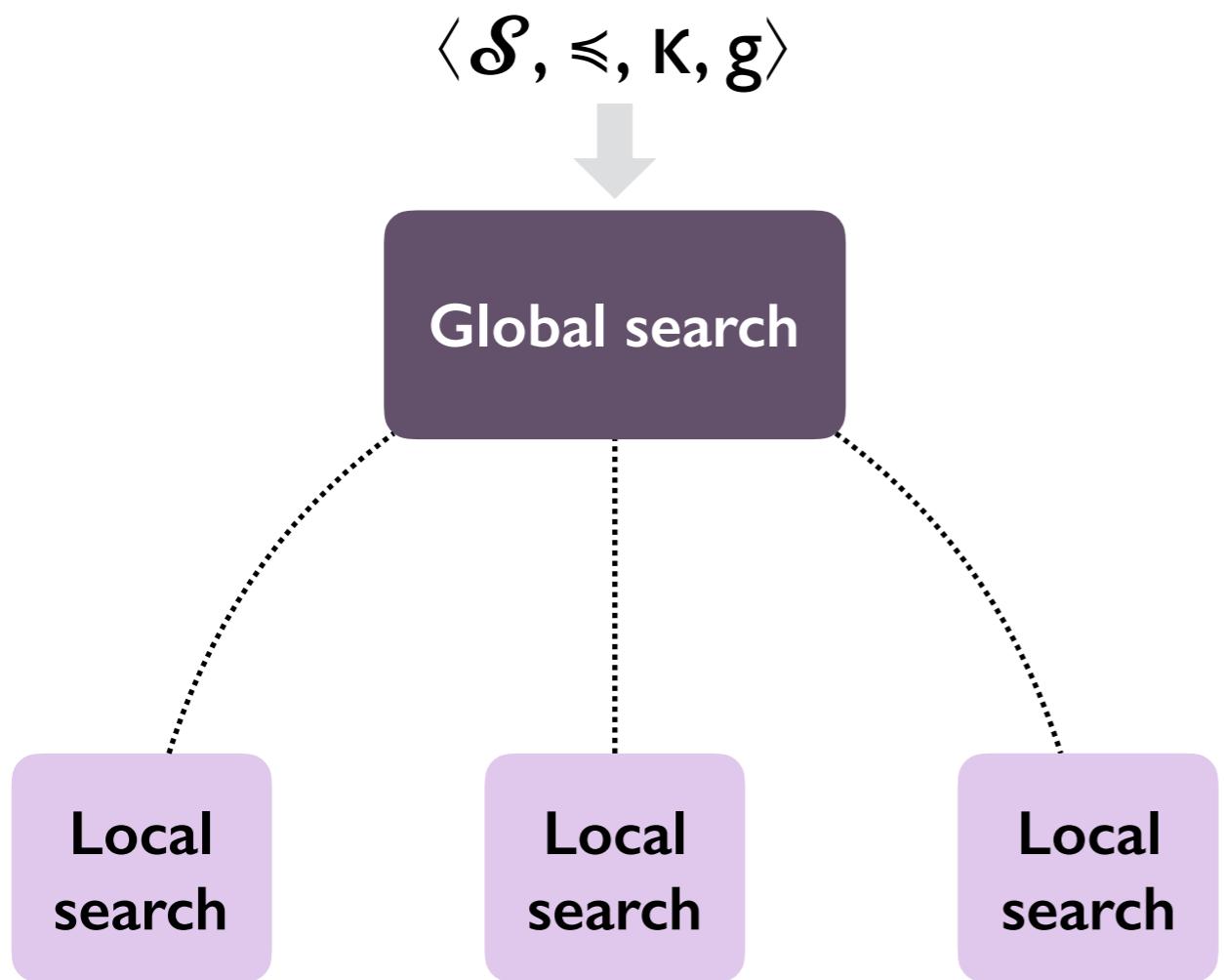
Solving with two cooperative searches



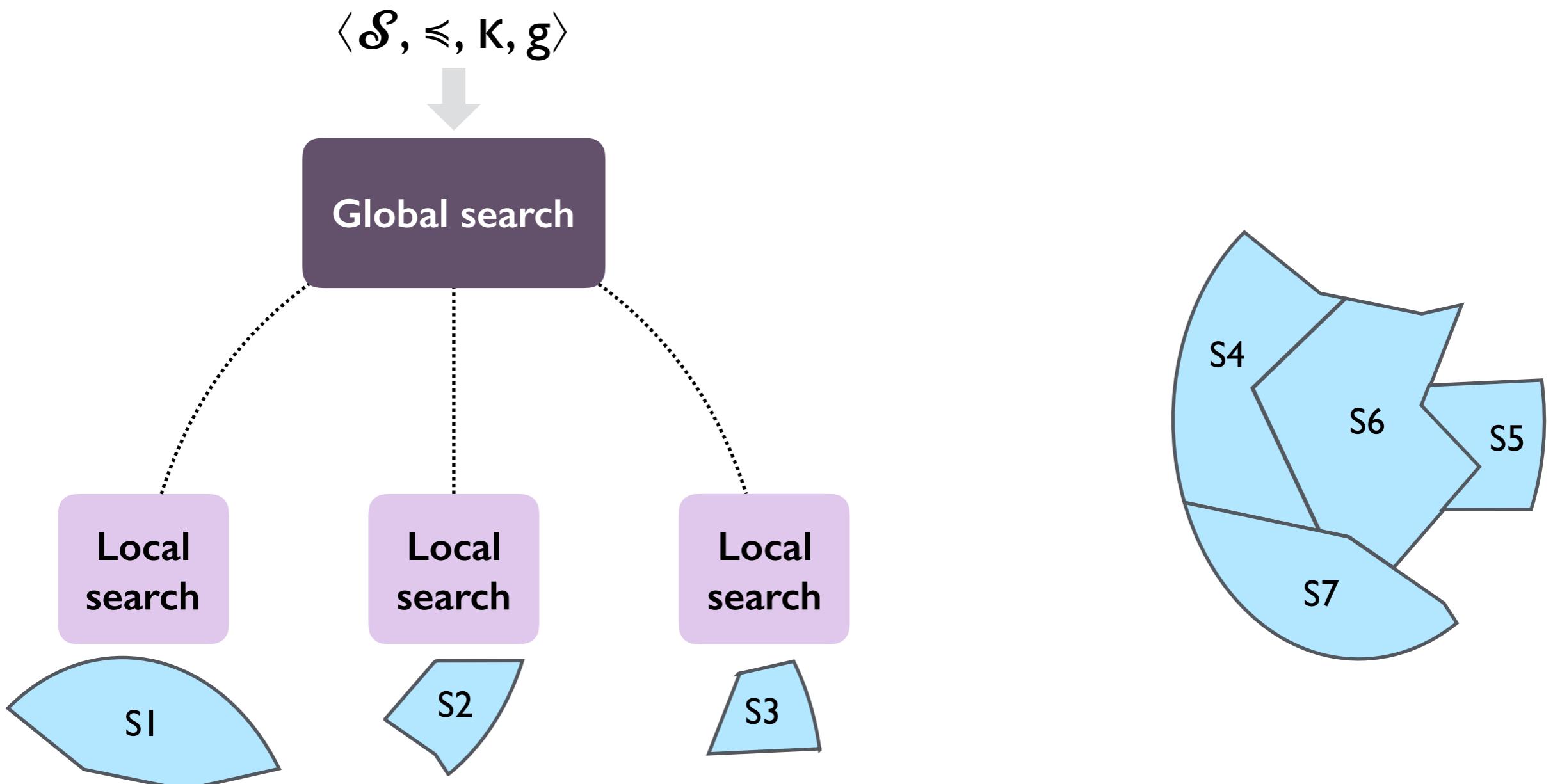
Solving with two cooperative searches



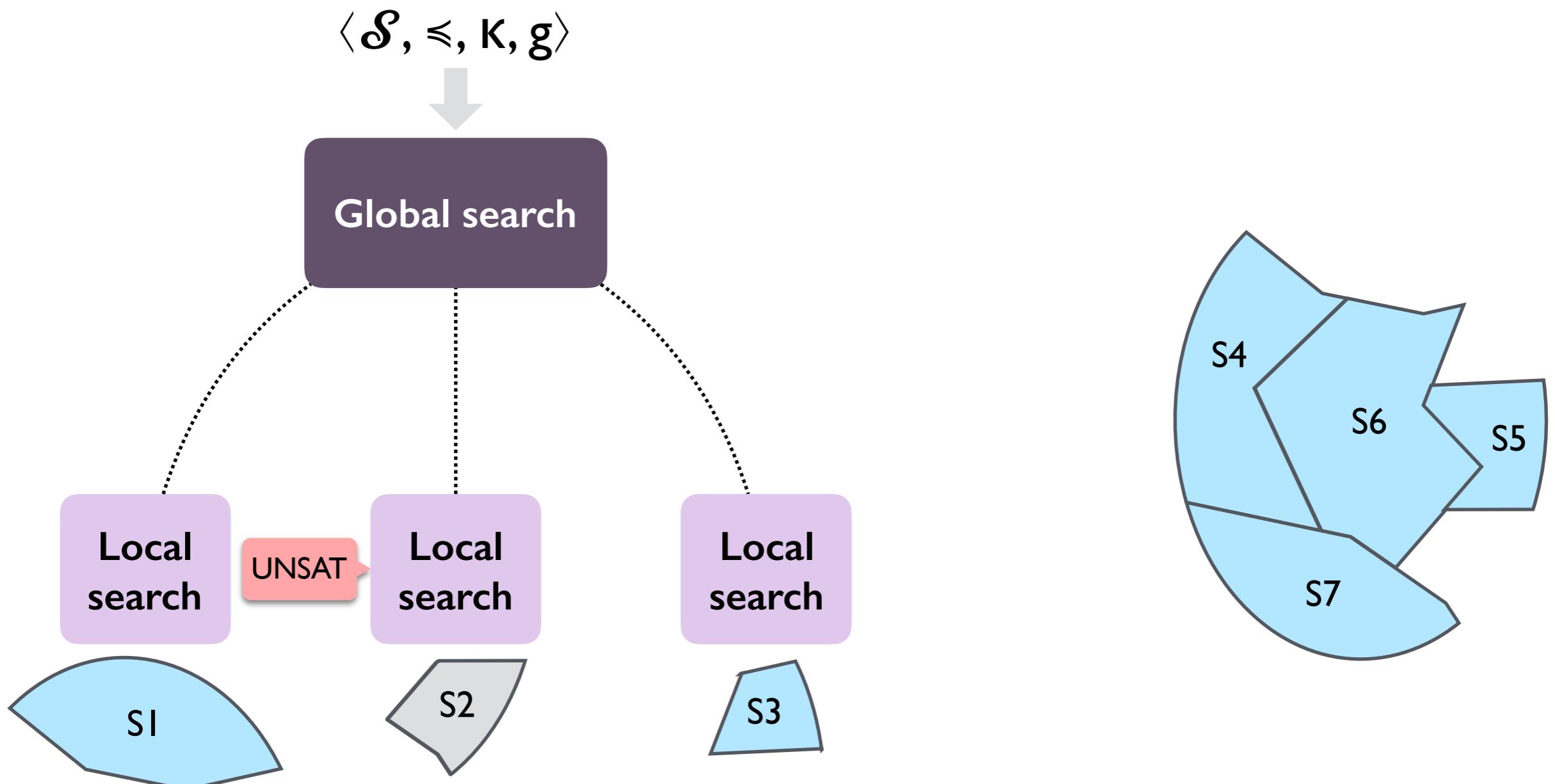
Solving with two cooperative searches



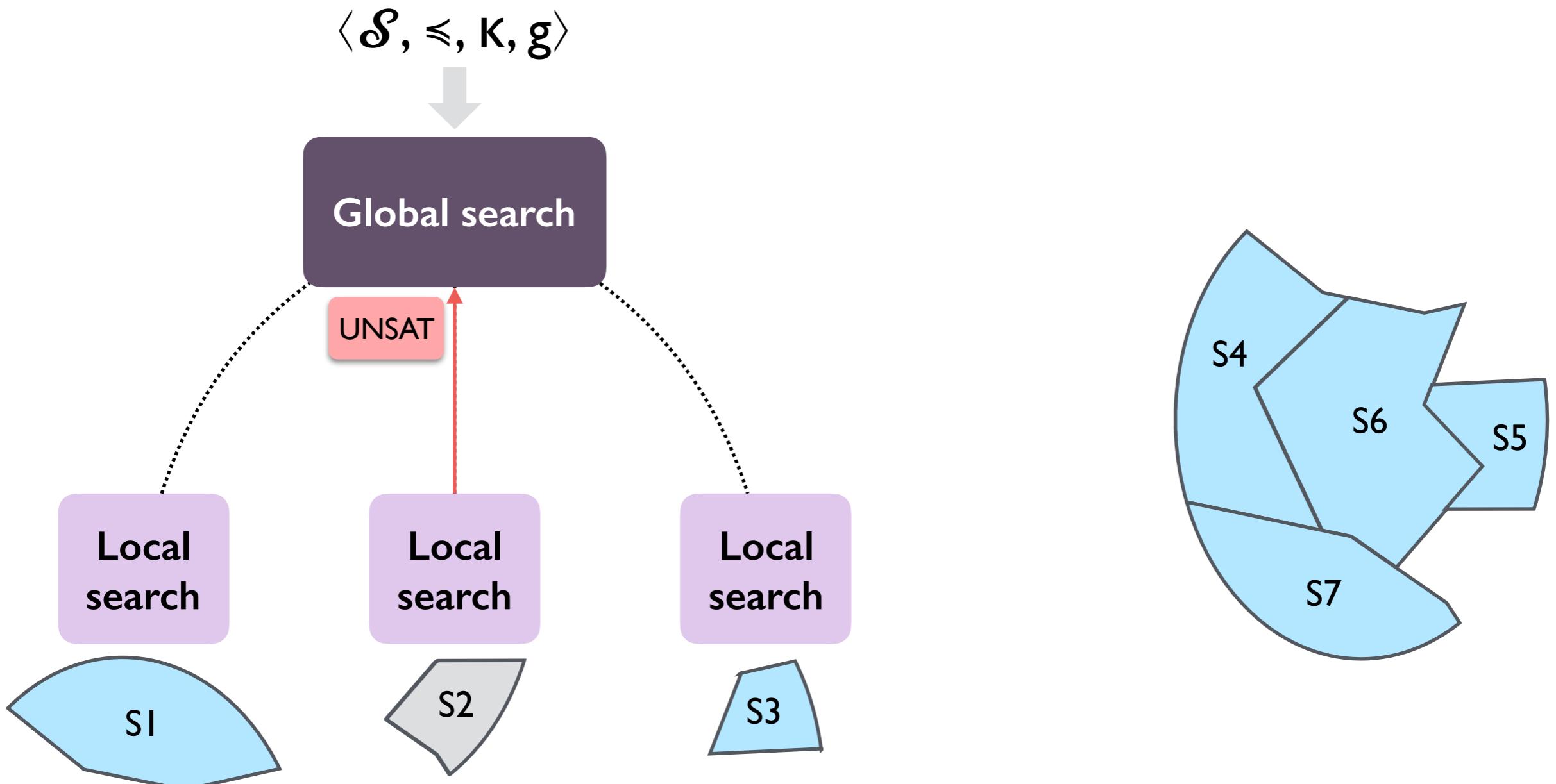
Solving with two cooperative searches



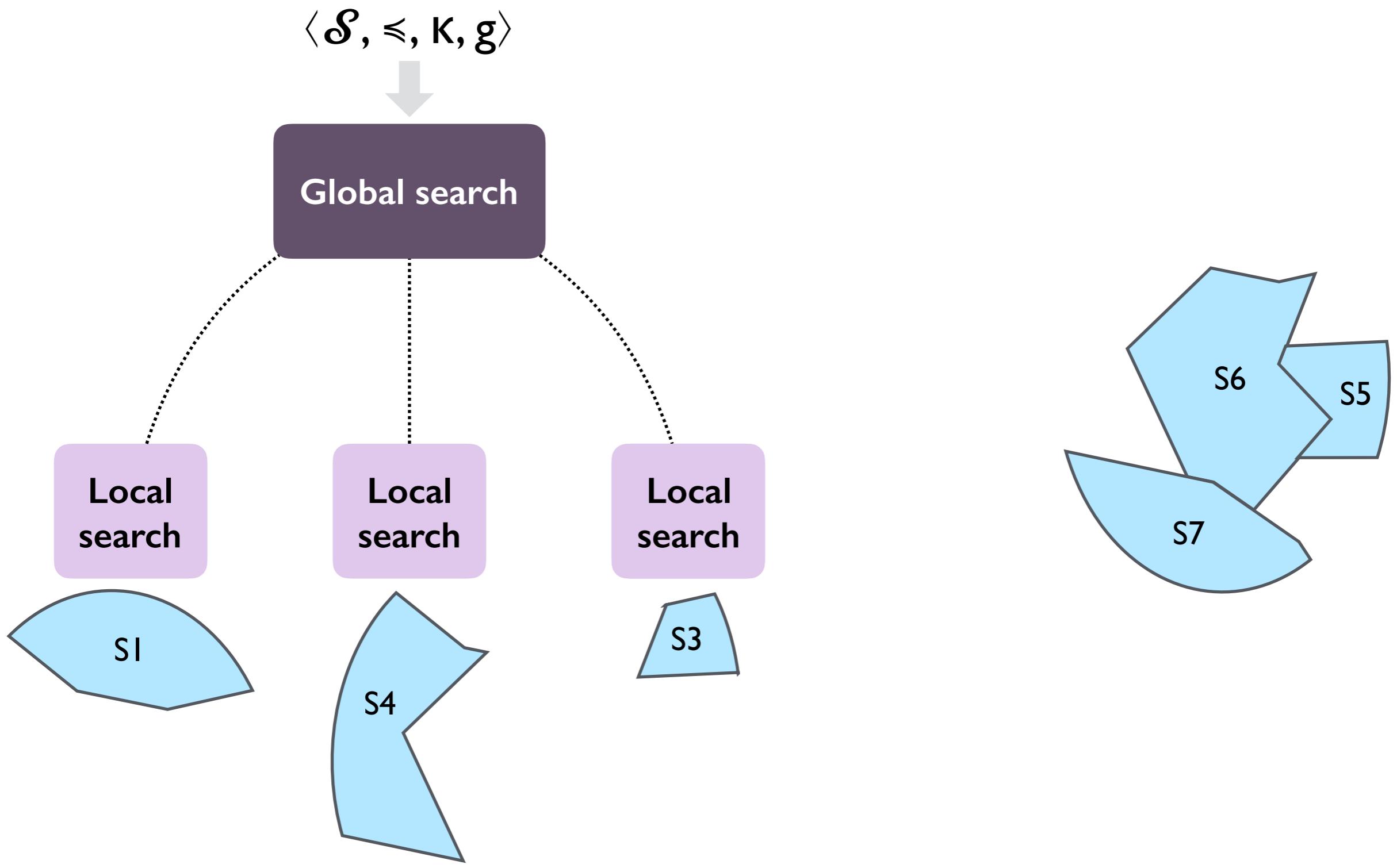
Solving with two cooperative searches



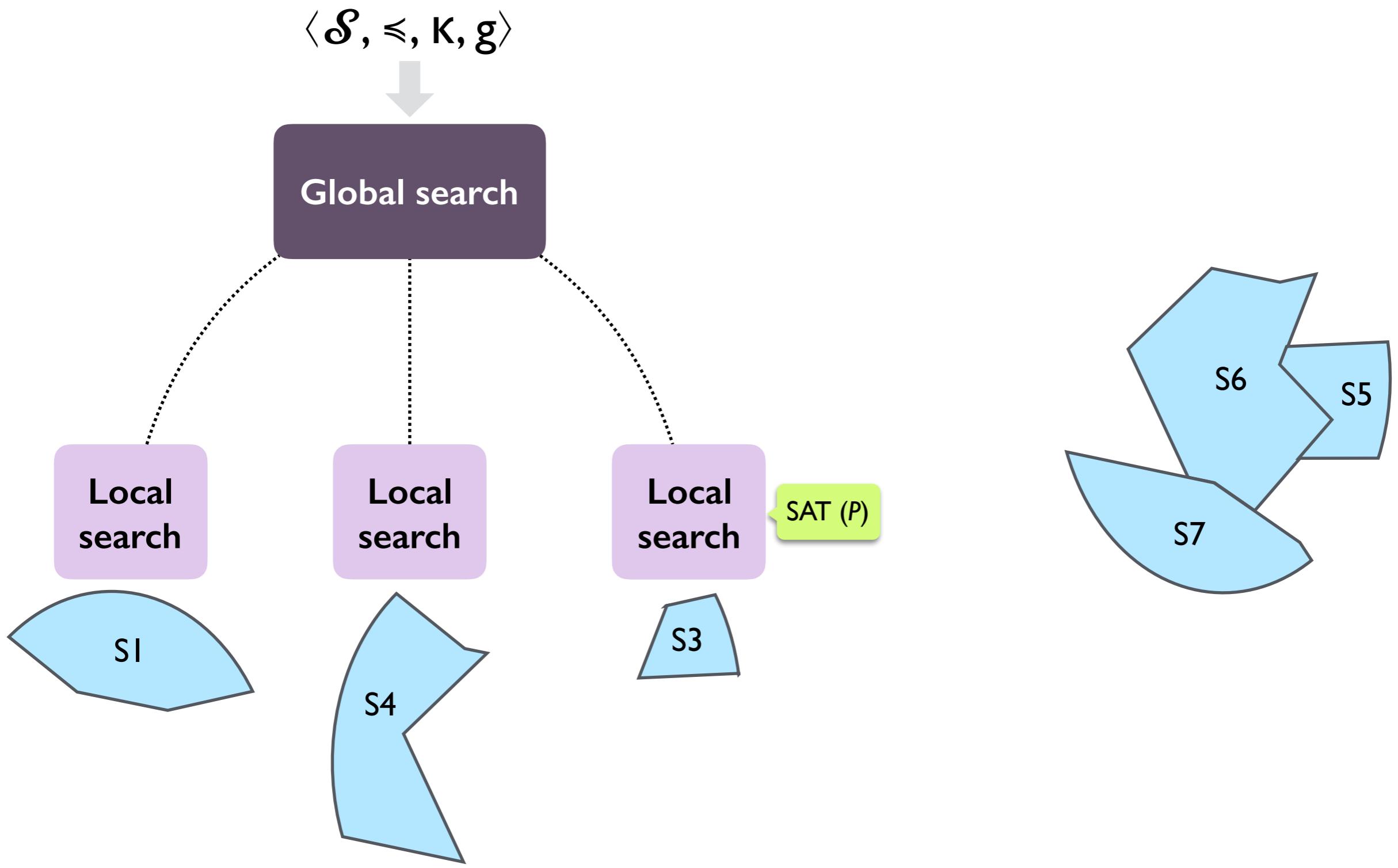
Solving with two cooperative searches



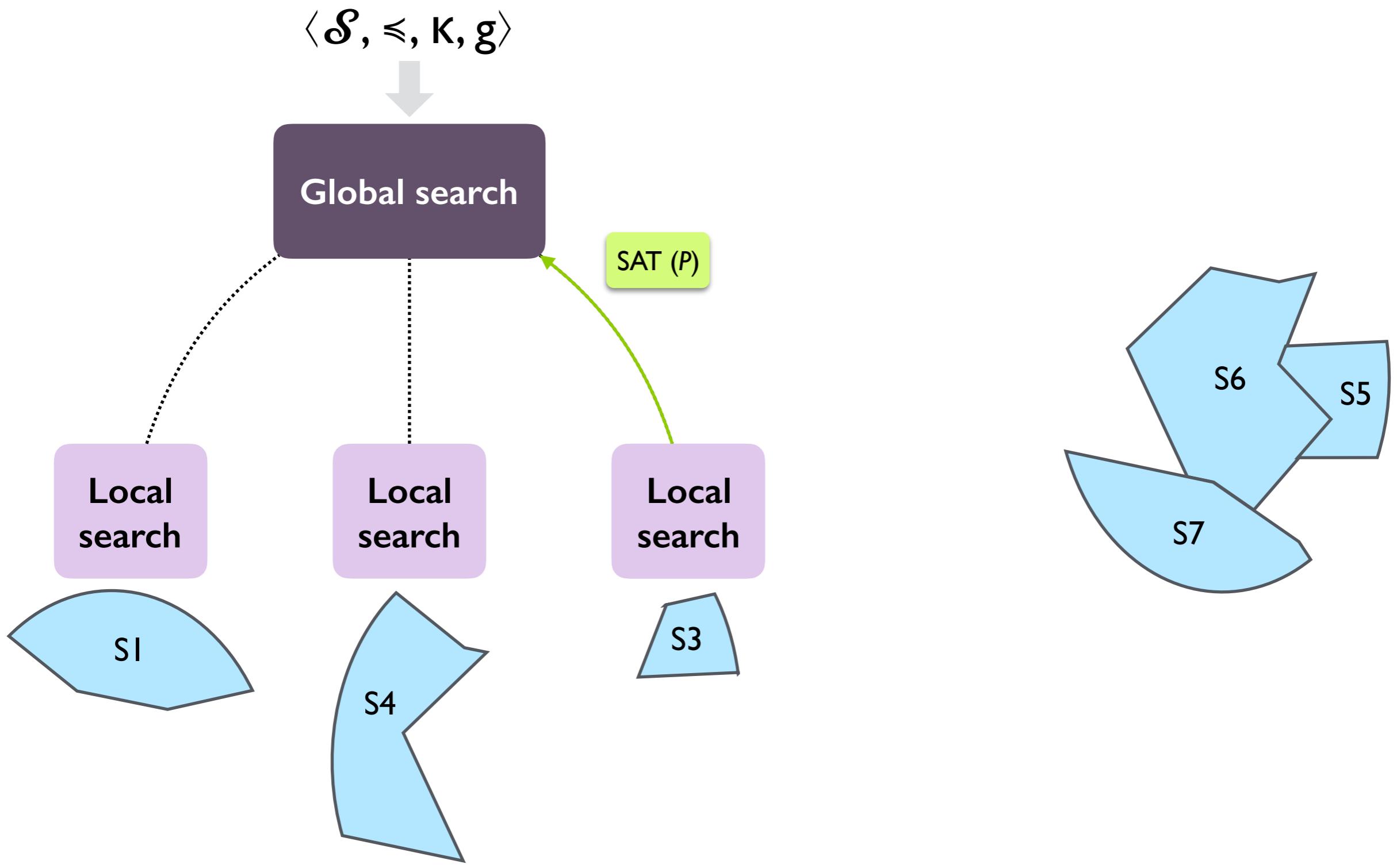
Solving with two cooperative searches



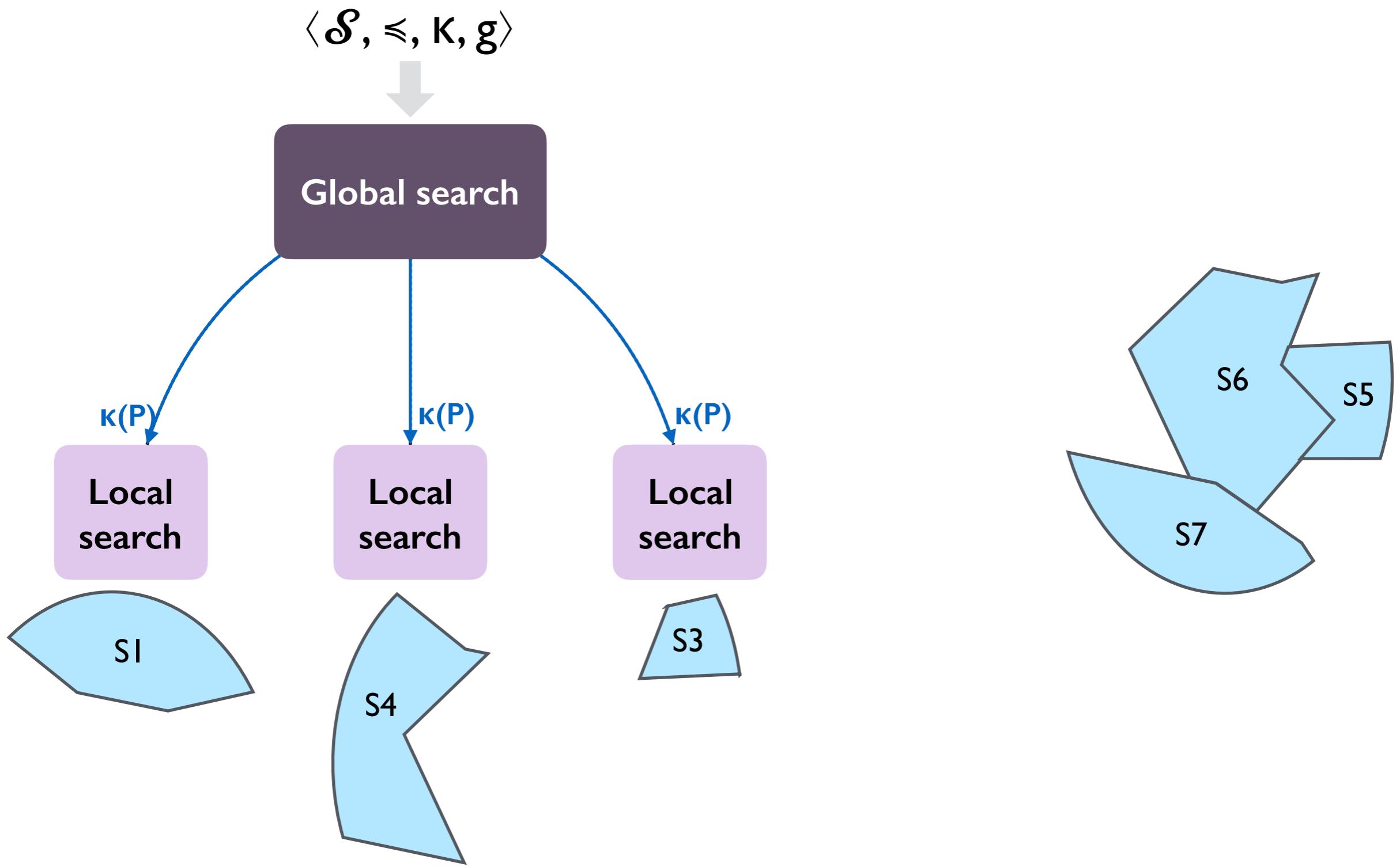
Solving with two cooperative searches



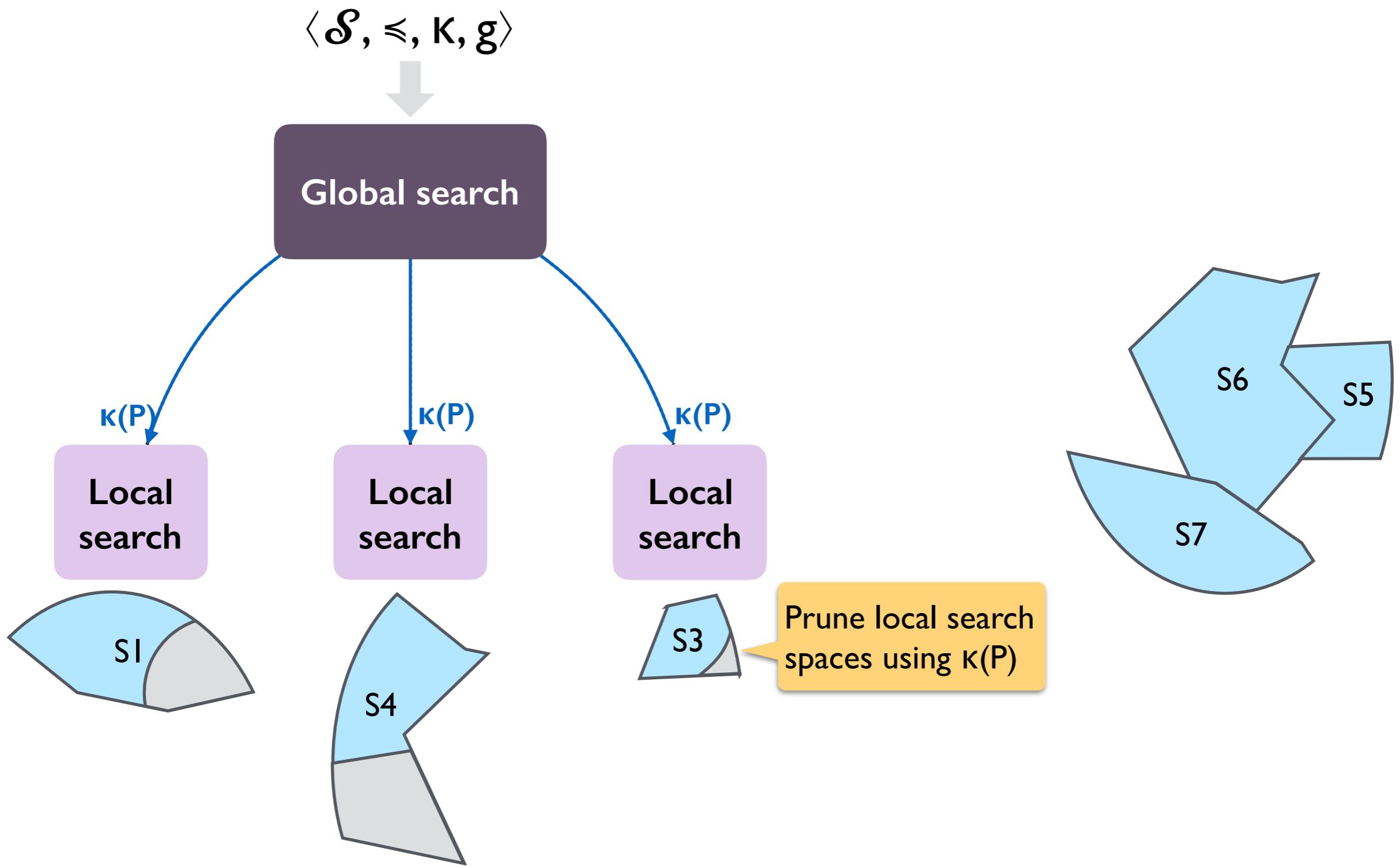
Solving with two cooperative searches



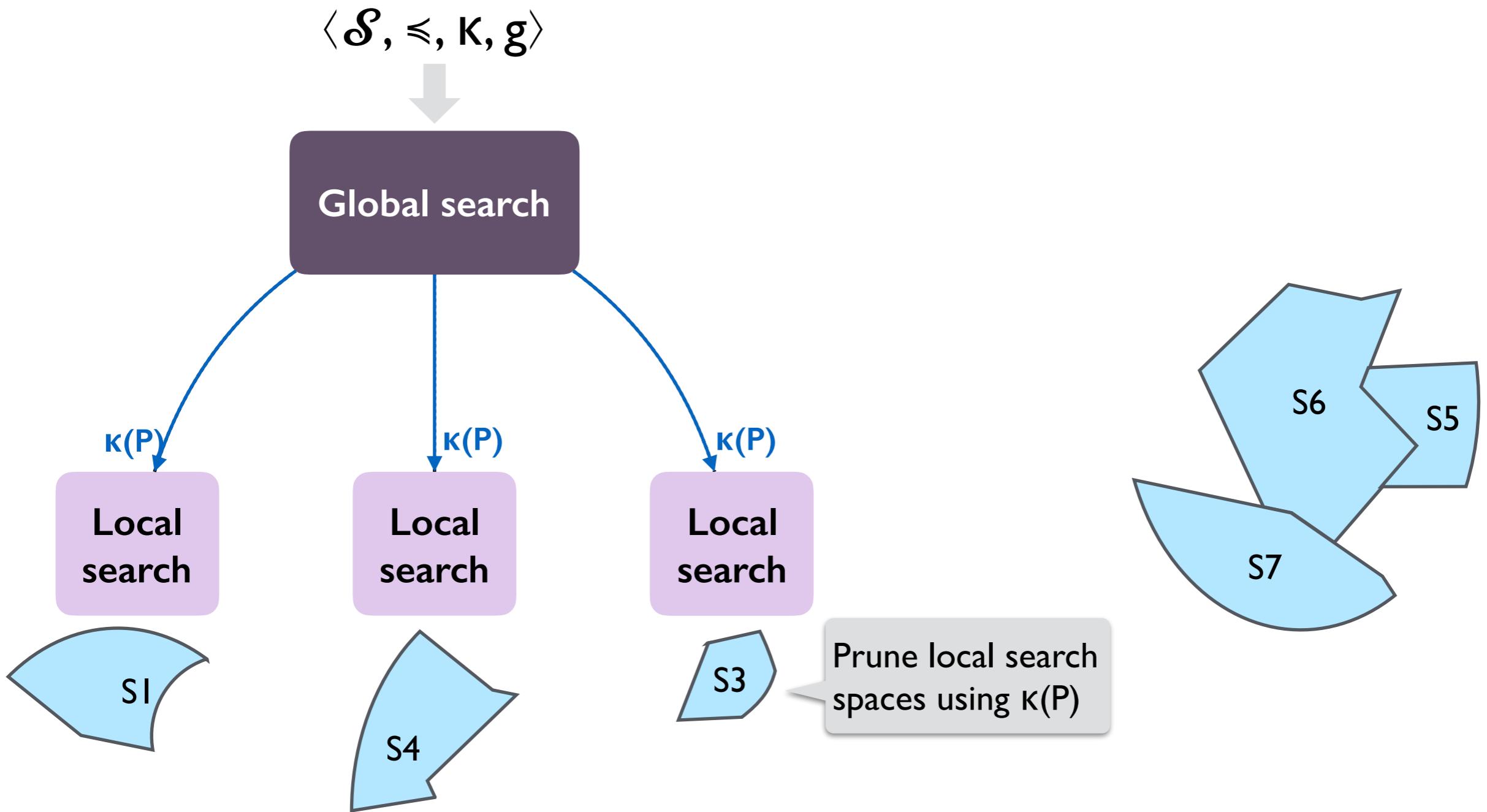
Solving with two cooperative searches



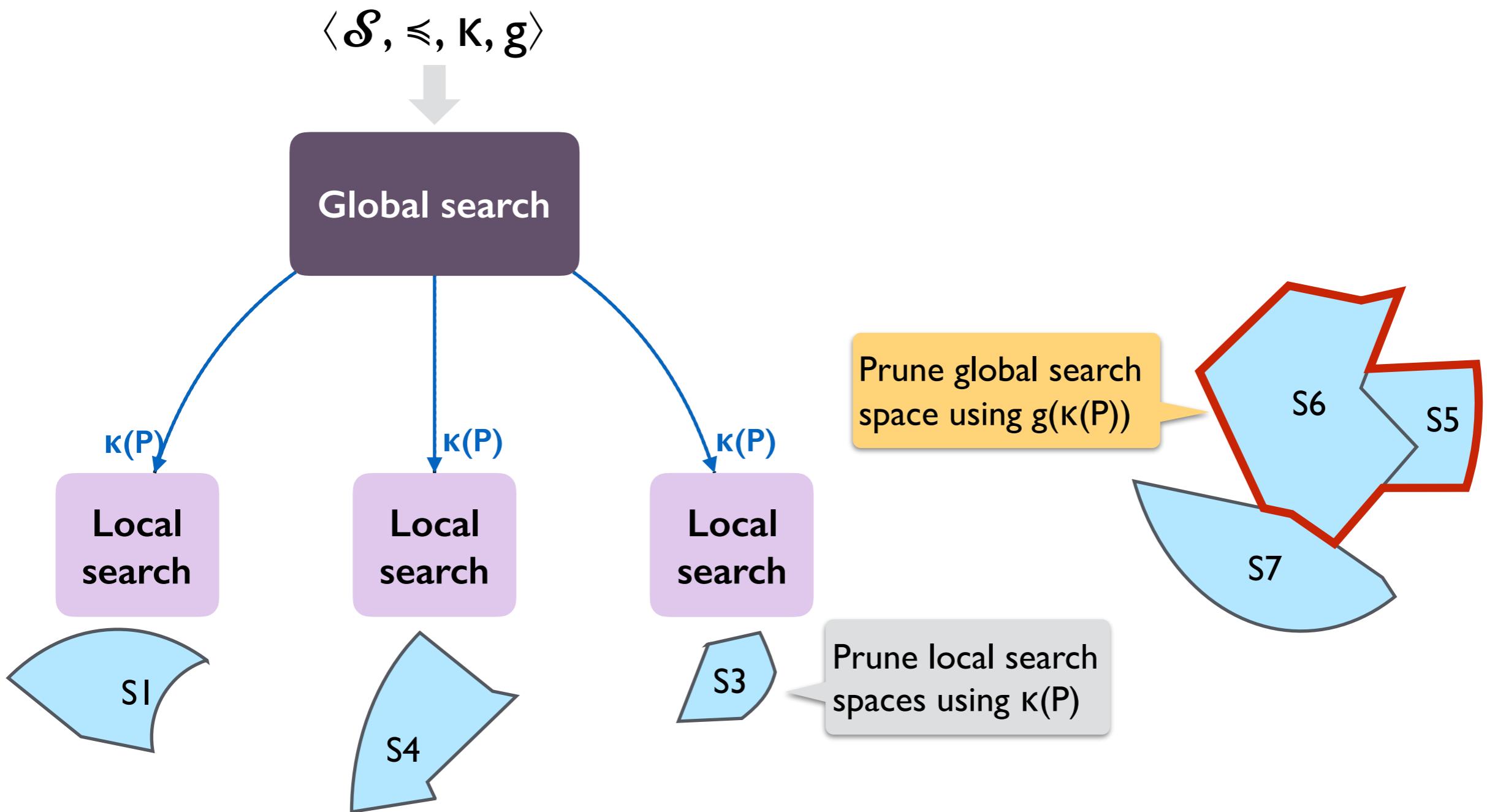
Solving with two cooperative searches



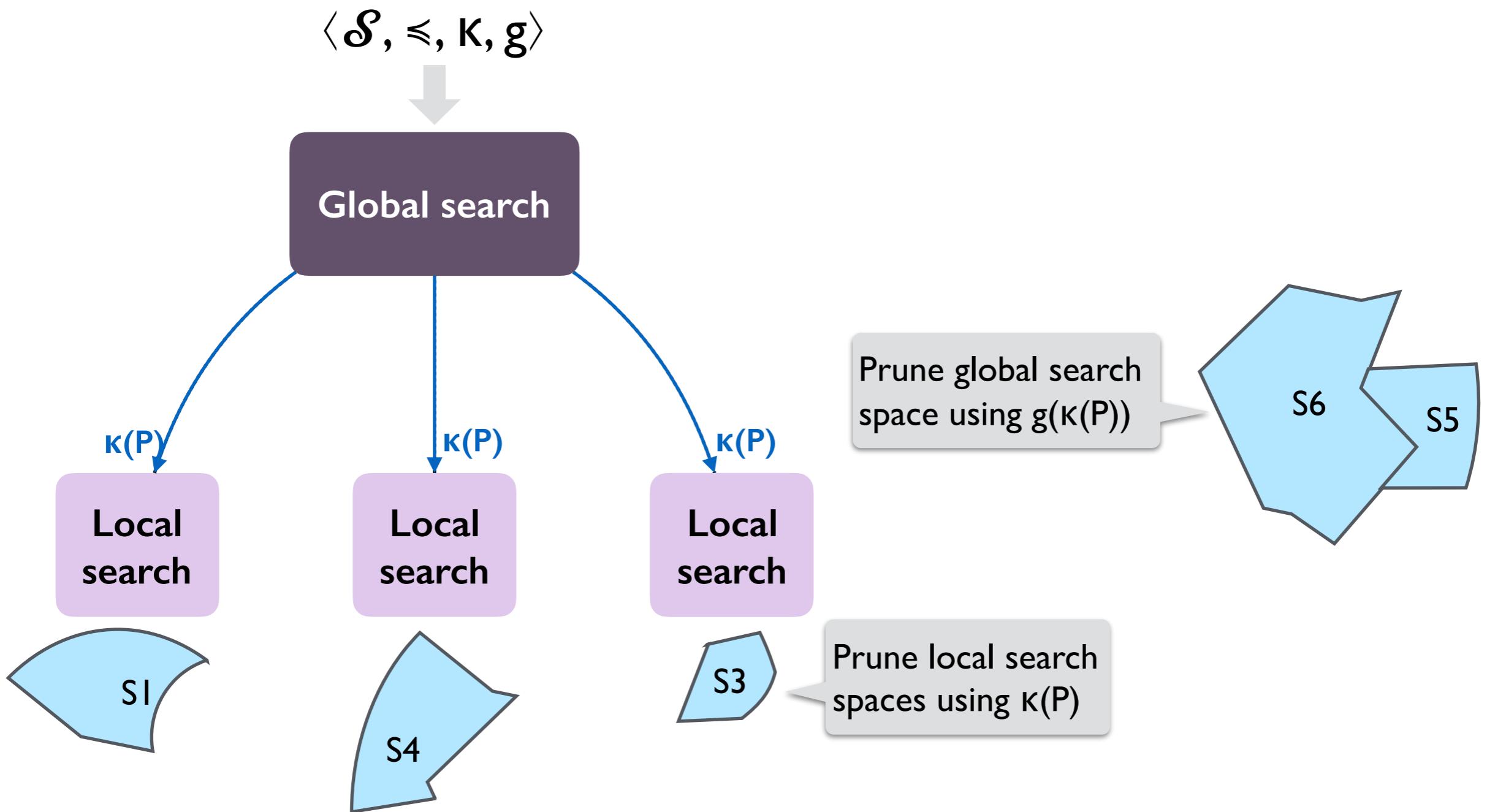
Solving with two cooperative searches



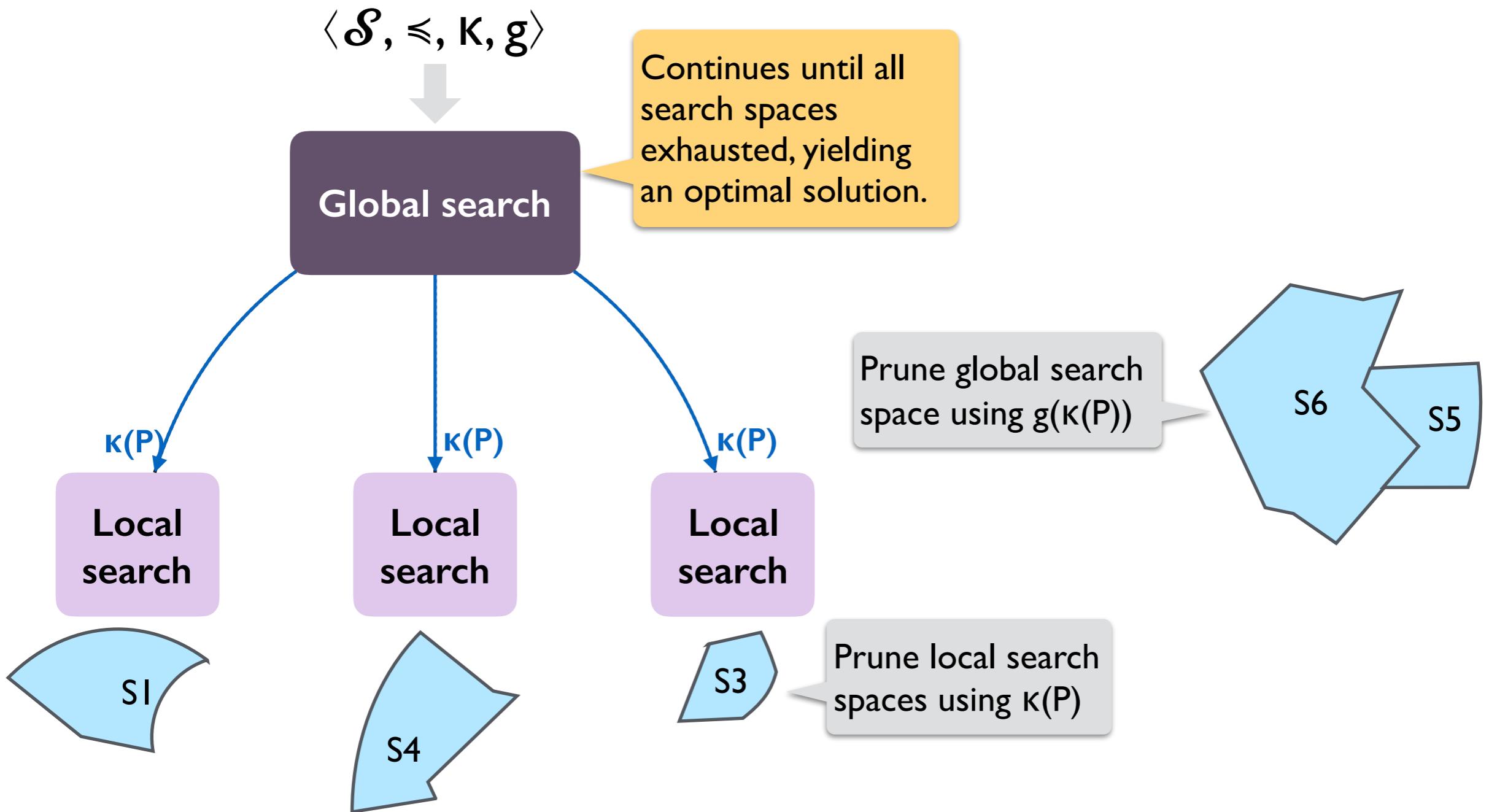
Solving with two cooperative searches



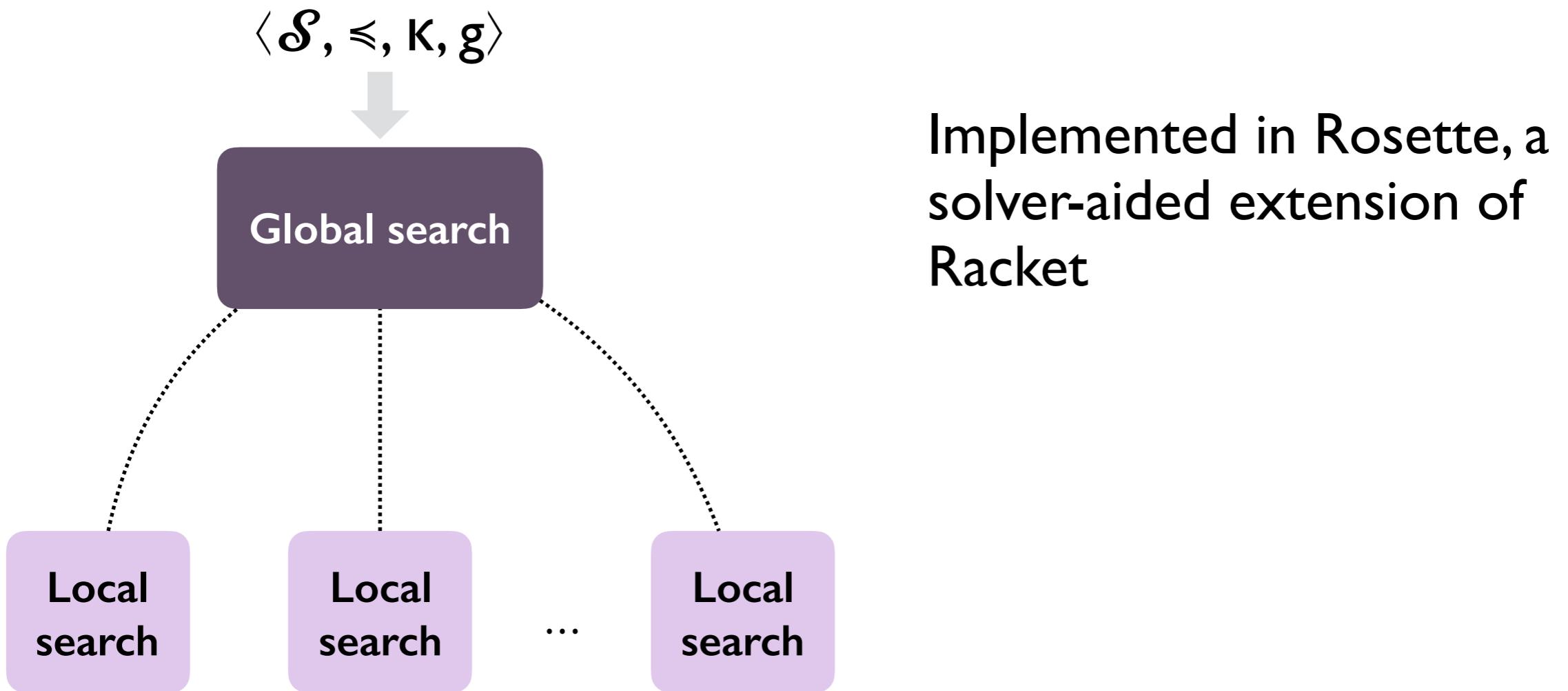
Solving with two cooperative searches



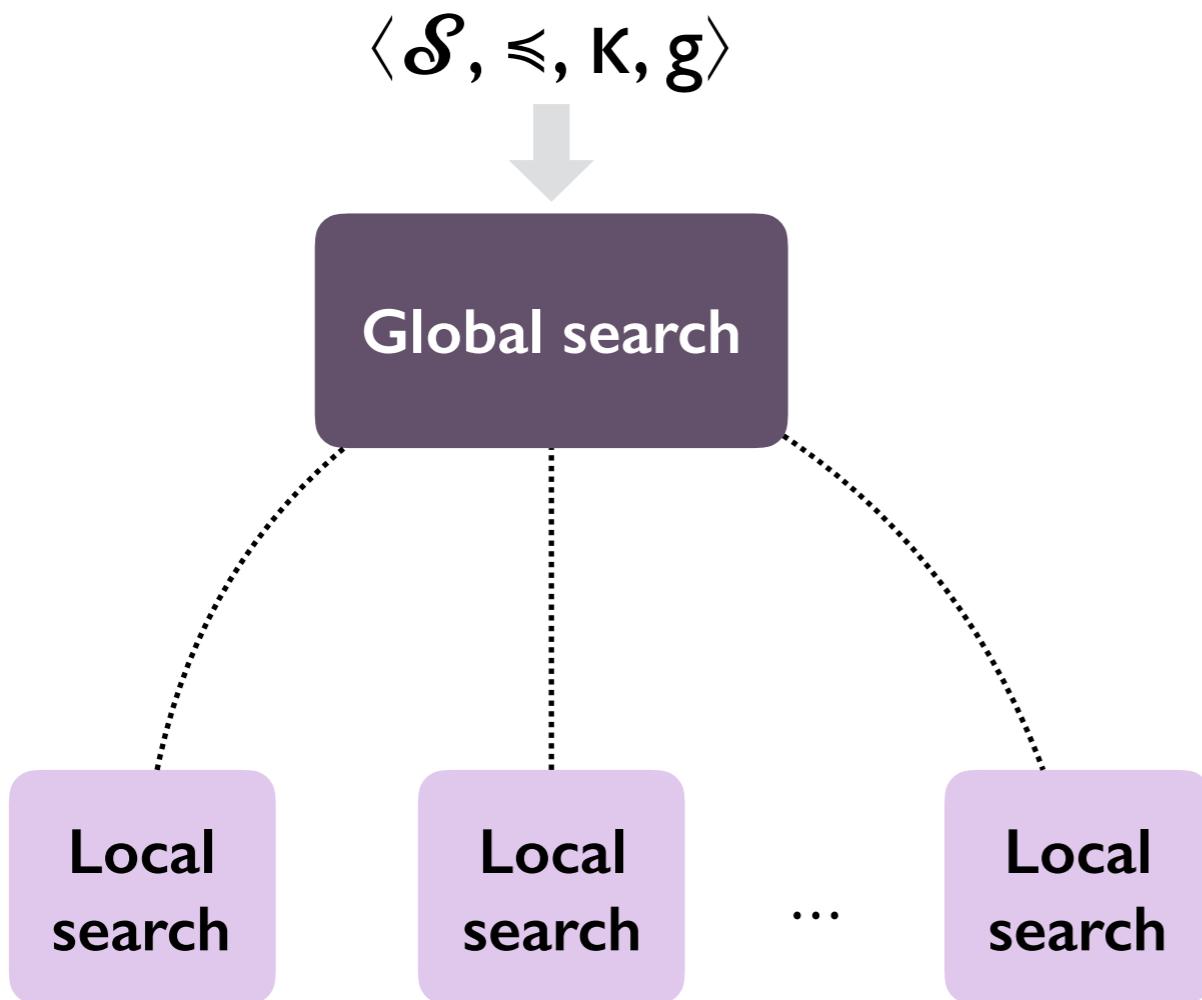
Solving with two cooperative searches



Synapse implementation



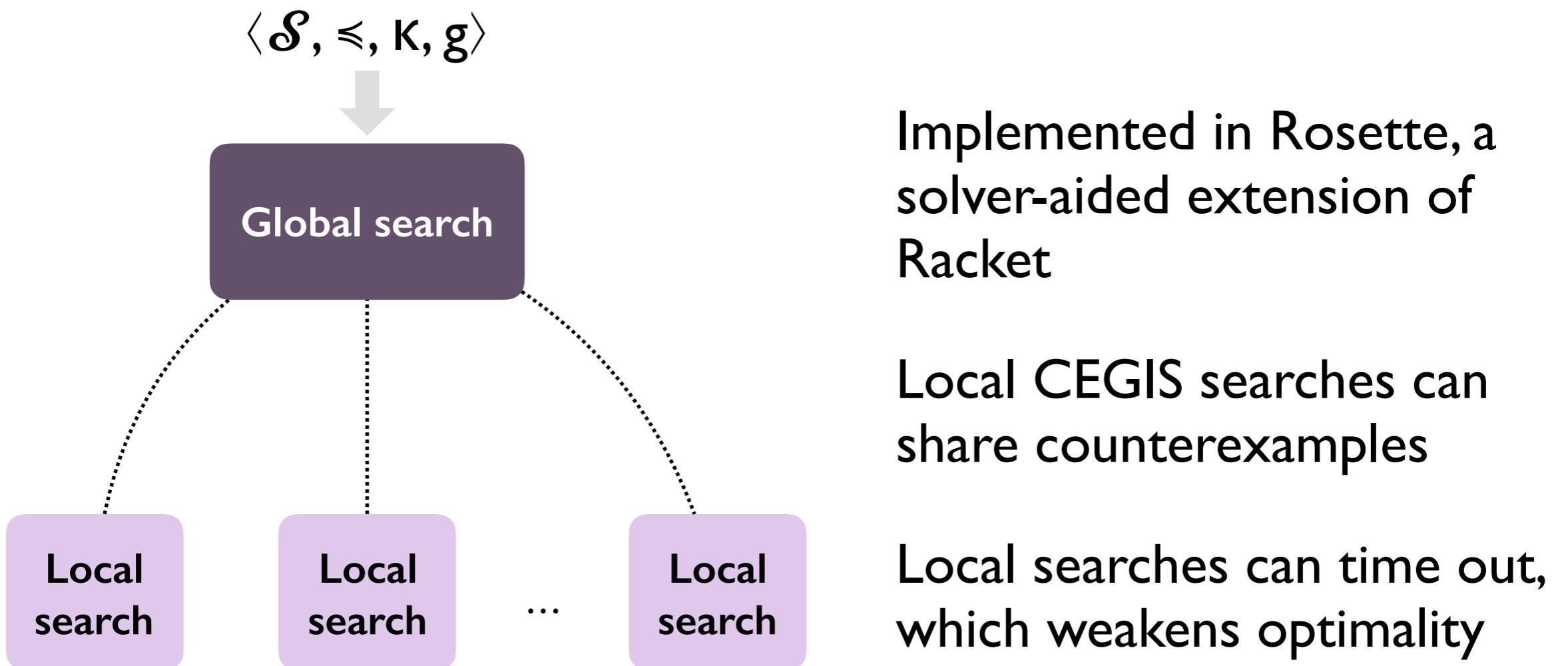
Synapse implementation



Implemented in Rosette, a solver-aided extension of Racket

Local CEGIS searches can share counterexamples

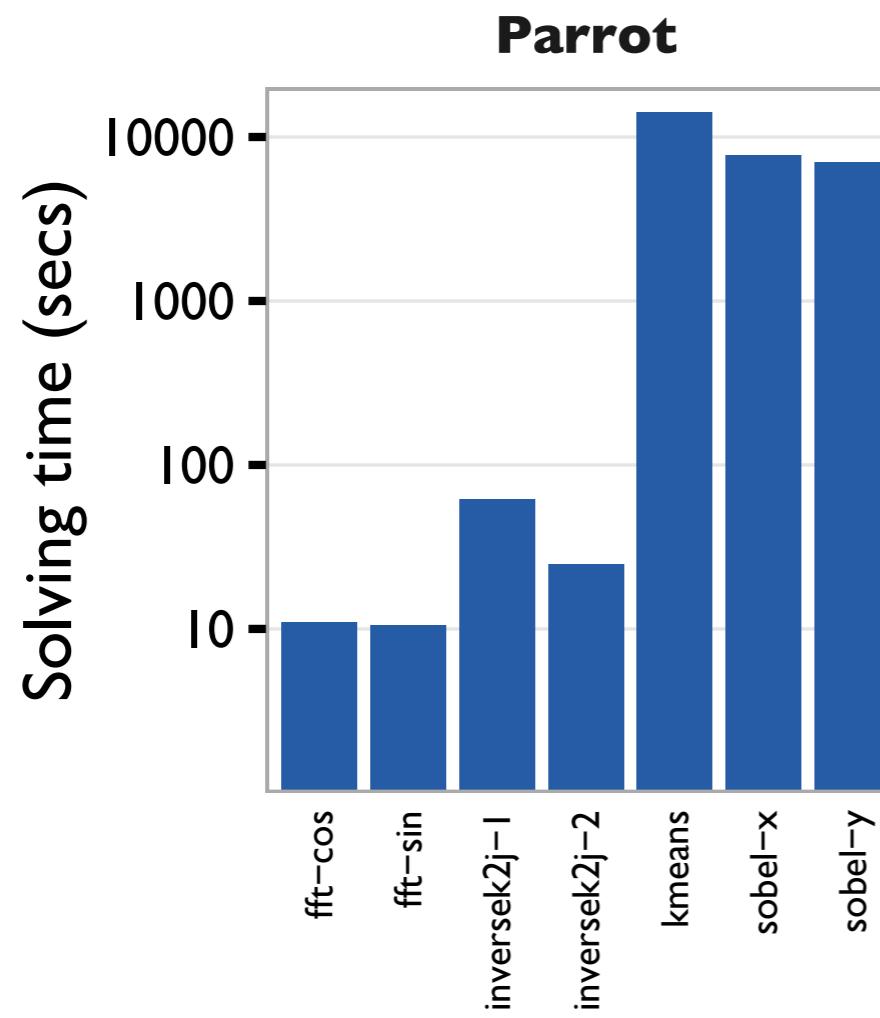
Synapse implementation



Synapse solves previously-intractable problems

Parrot benchmarks from approximate computing [Esmaelizadeh et al., 2012]

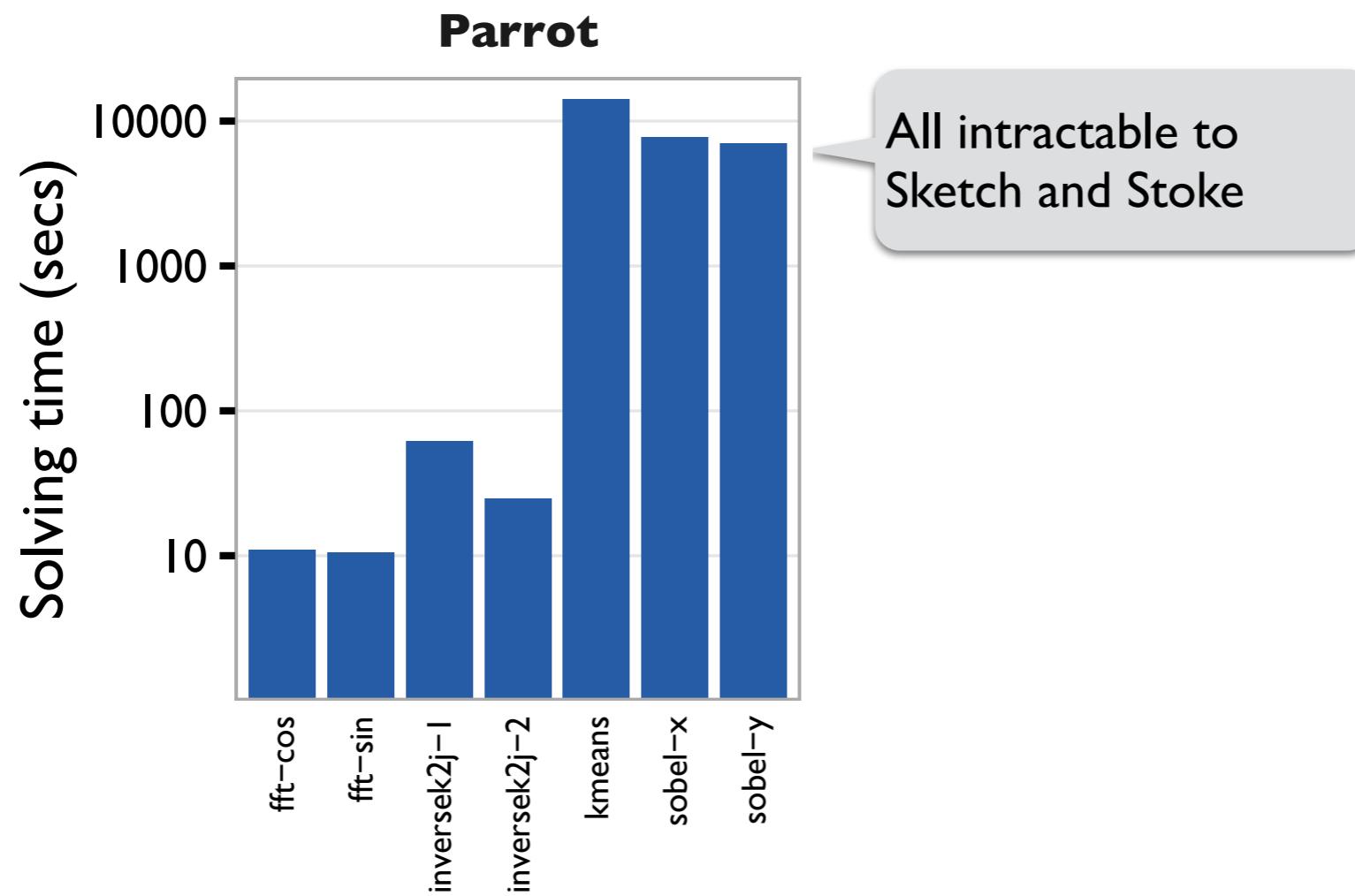
Find the most efficient approximate program within an error bound



Synapse solves previously-intractable problems

Parrot benchmarks from approximate computing [Esmaelizadeh et al., 2012]

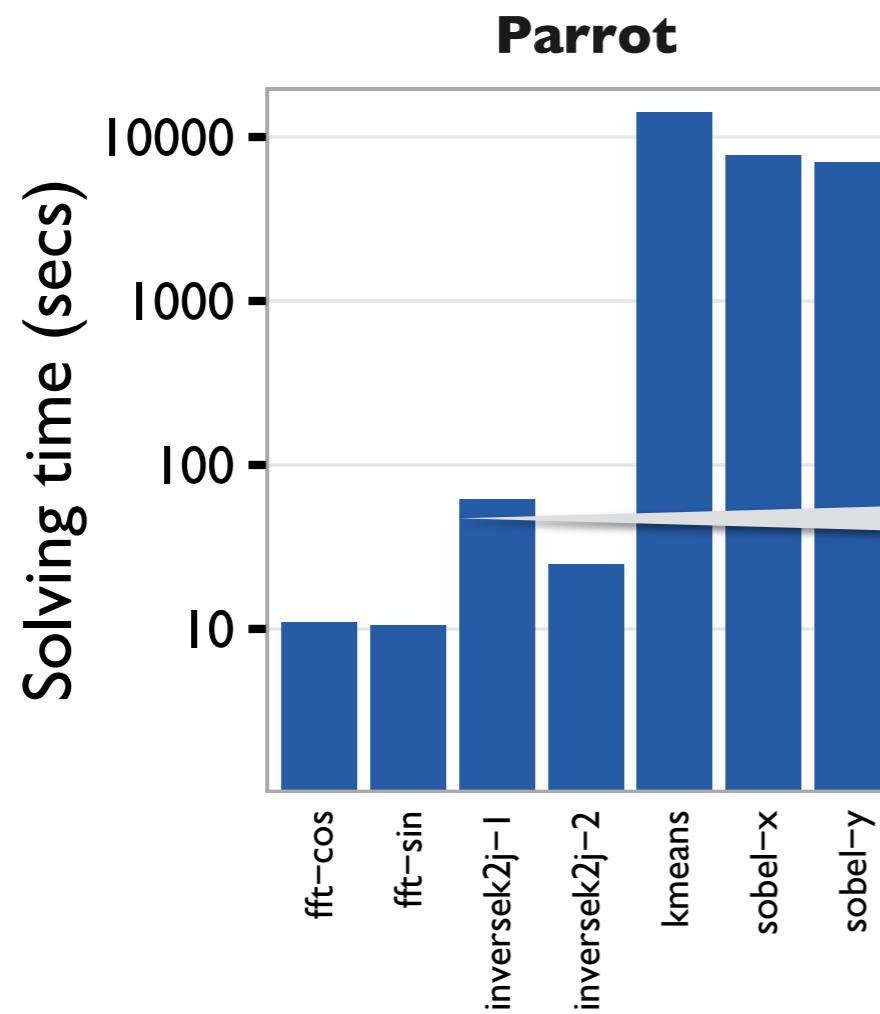
Find the most efficient approximate program within an error bound



Synapse solves previously-intractable problems

Parrot benchmarks from approximate computing [Esmaelizadeh et al., 2012]

Find the most efficient approximate program within an error bound



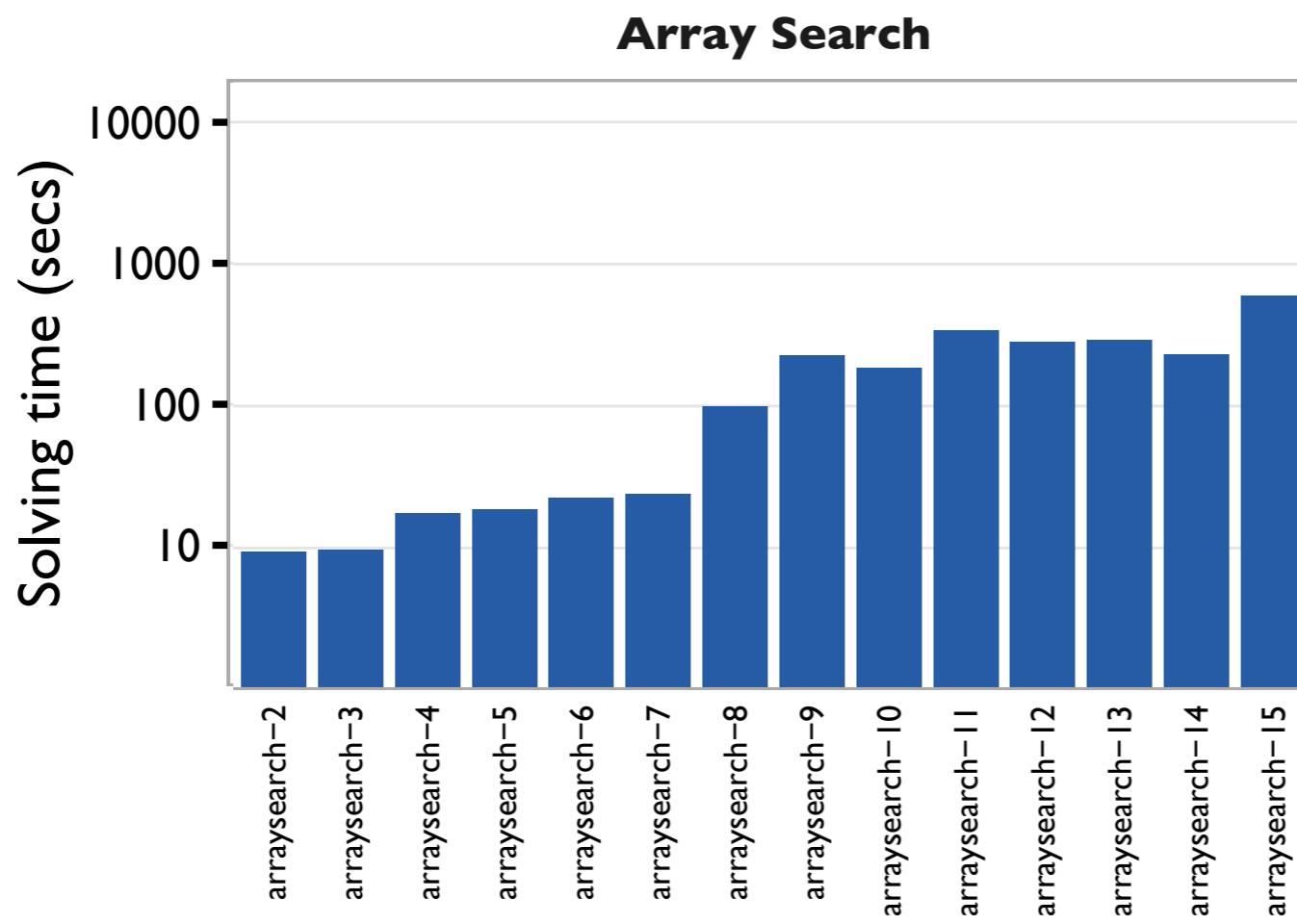
All intractable to
Sketch and Stoke

```
def inversek2j(float x, float y):  
    th2 = acos(((x*x) + (y*y) - 0.5) / 0.5)  
    th1 = asin((y * (0.5 + 0.5*cos(th2)) -  
               0.5*x*sin(th2)) / (x*x + y*y))  
    return th1
```

Synapse solves standard benchmarks optimally

Array Search benchmarks from the syntax-guided synthesis (SyGuS) competition [Alur et al., 2015]

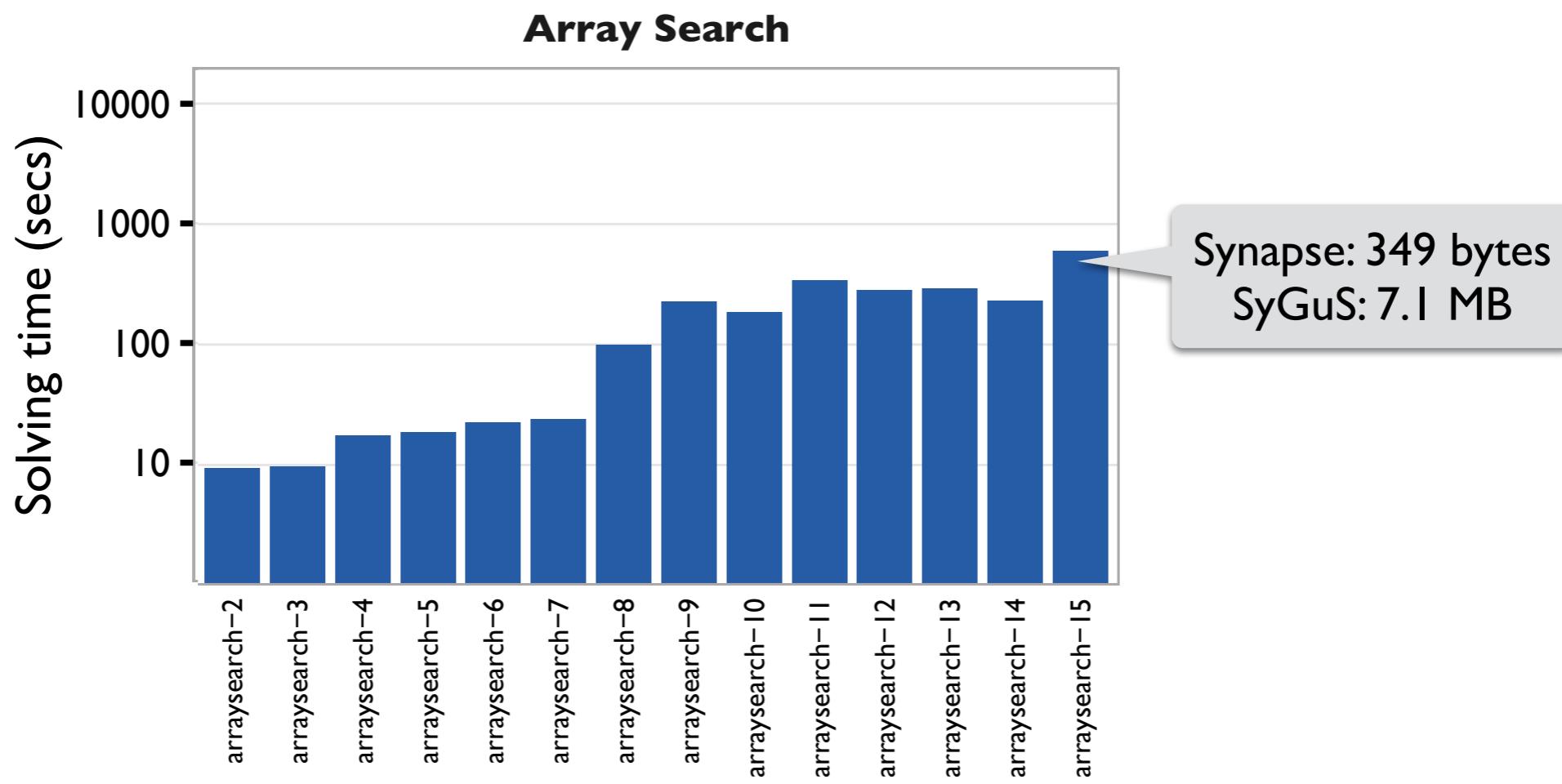
arraysearch- n : find program that searches lists of length n



Synapse solves standard benchmarks optimally

Array Search benchmarks from the syntax-guided synthesis (SyGuS) competition [Alur et al., 2015]

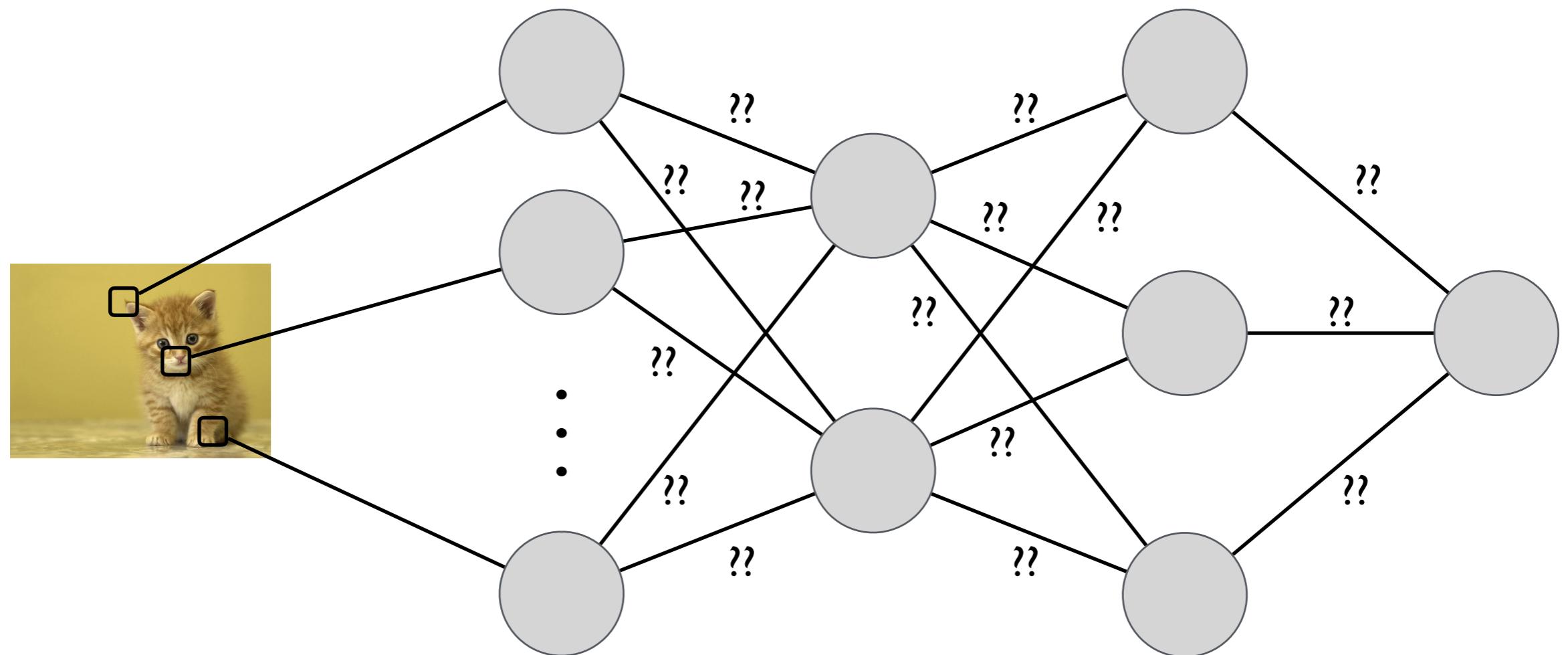
arraysearch- n : find program that searches lists of length n



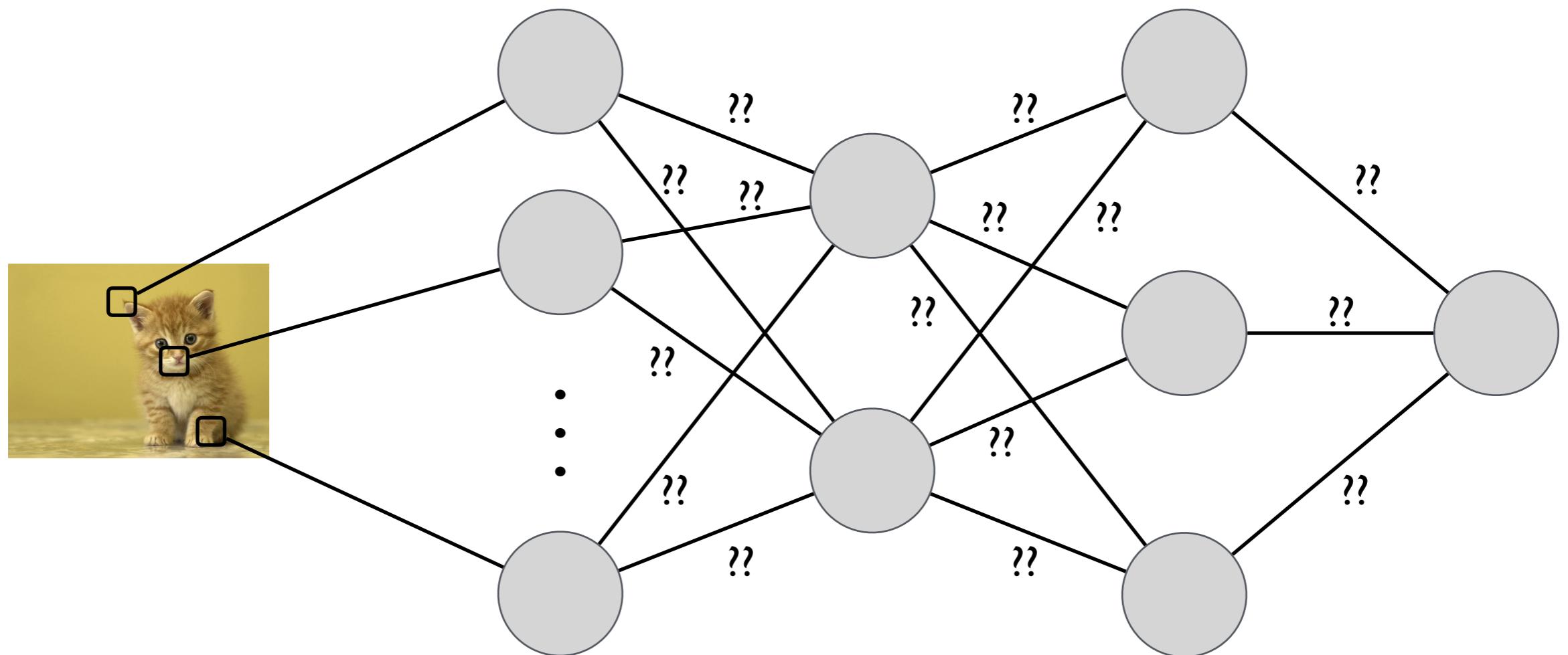
Is this a cat?



Synapse reasons about complex costs



Synapse reasons about complex costs



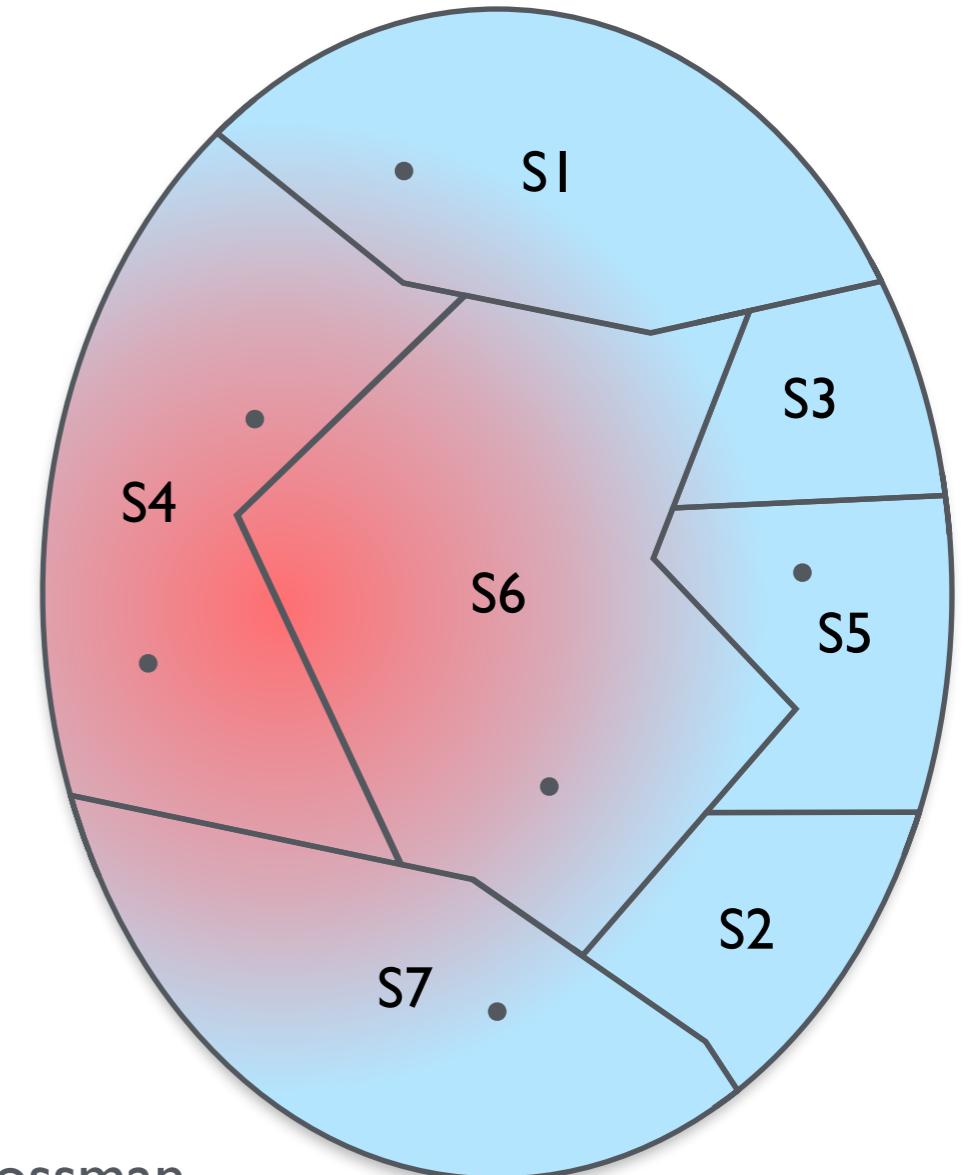
$$\kappa(P) = \sum_i |P(x_i) - y_i|$$

Classification error executes the program for each point in the training set

Metasketches express structure and strategy

A metasketch contains:

1. structured candidate space (\mathcal{S}, \leq)
2. cost function (κ)
3. gradient function (g)



James Bornholt, Emina Torlak, Luis Ceze, and Dan Grossman.
Optimizing Synthesis with Metasketches. POPL 2016.

MemSynth

James Bornholt and Emina Torlak.

Synthesizing Memory Models from Framework Sketches and Litmus Tests. PLDI 2017.

A memory model defines the reordering behaviors of a multiprocessor

Initially $X = Y = 0$

Thread 1	Thread 2
$X = 1$	$Y = 1$
print Y	print X

Can this program print two zeroes?

A memory model defines the reordering behaviors of a multiprocessor

Initially $X = Y = 0$

Thread 1

Thread 2

$X = 1$

print Y

$Y = 1$

print X

Can this program print two zeroes?

A memory model defines the reordering behaviors of a multiprocessor

Initially $X = Y = 0$

Thread 1 Thread 2

X = 1

Y = 1

print Y

print X

Can this program print two zeroes?

A memory model defines the reordering behaviors of a multiprocessor

Initially $X = Y = 0$

Thread 1 Thread 2

$X = 1$

$Y = 1$

print X

print Y

Can this program print two zeroes?

A memory model defines the reordering behaviors of a multiprocessor

Initially $X = Y = 0$

Thread 1 Thread 2

Y = 1

X = 1

print X

print Y

Can this program print two zeroes?

A memory model defines the reordering behaviors of a multiprocessor

Initially $X = Y = 0$

Thread 1

Thread 2

$Y = 1$

print X

$X = 1$

print Y

Can this program print two zeroes?



Intel® 64 and IA-32 Architectures Software Developer's Manual

VOLUME 1: Basic Architecture

1



Intel® 64 and IA-32 Architectures Software Developer's Manual

VOLUME 2B: Instruction Set Reference, A-H

2A



Intel® 64 and IA-32 Architectures Software Developer's Manual

VOLUME 2B: Instruction Set Reference, N-Z

2B



Intel® 64 and IA-32 Architectures Software Developer's Manual

VOLUME 3A: System Programming Guide, Part 1

3A



Intel® 64 and IA-32 Architectures Software Developer's Manual

VOLUME 3B: System Programming Guide, Part 2

3B



Intel® 64 and IA-32 Architectures Optimization Reference Manual



Intel® 64 and IA-32 Architectures Software Developer's Manual
VOLUME 1: Basic Architecture



Intel® 64 and IA-32 Architectures Software Developer's Manual

and modelling, and it consists of a large number of different types of models, including:
- The **agent-based approach**, which is based on the idea of individual agents interacting with each other and with their environment;
- The **cellular automata approach**, which is based on the idea of a regular grid of cells, each of which can have a different state, and which can interact with its neighbors according to a set of rules;
- The **rule-based approach**, which is based on the idea of a set of rules that define how objects should behave in different situations.
In order to make modelling easier, some approaches have been developed to reduce the complexity of the problem, such as:

- Object-oriented programming**: This approach allows you to define objects with properties and methods, and to reuse them in different parts of your program.
- Functional programming**: This approach allows you to define functions that take inputs and produce outputs, and to reuse them in different parts of your program.
- Constraint programming**: This approach allows you to define constraints that must be satisfied by the solution, and to reuse them in different parts of your program.

Identifying the location of the nearest working water or oil well and determining whether the production of oil or gas from the well is feasible. Section 4.2.2 contains detailed information on how to identify potential oil and gas wells and determine their feasibility through the use of specific databases.

4.2.1 Recovery Screening in the "Hot," "Probable," and "Inferred" Reservoirs

The reservoir screening process can be used to quickly assess a reservoir model. This allows users to quickly identify which reservoirs are likely to contain hydrocarbons and which reservoirs are likely to be nonproductive. The process involves three main steps:

1. **Initial Screening:** This step involves reviewing the reservoir model to identify any potential oil and gas wells. This includes reviewing the geological data, petrophysical data, and production history of the reservoir.
2. **Feasibility Screening:** This step involves assessing the potential for oil and gas production from the identified wells. This includes reviewing the reservoir's pressure, temperature, and flow characteristics, as well as the economic viability of the production.
3. **Final Screening:** This step involves identifying the most promising reservoirs based on the results of the previous two steps. This includes ranking the reservoirs based on their potential for oil and gas production and determining the most feasible options for development.

In this type of oil and gas exploration, it is important to use strict criteria to eliminate false positives. This ensures that only the most promising reservoirs are considered for further investigation. It also helps to avoid unnecessary costs associated with developing reservoirs that are unlikely to be productive.

4.2.2 Recovery Screening on the "Hot" and "Very Hot" Reservoir Facades

This section focuses on the identification of potential oil and gas wells located near the "Hot" and "Very Hot" reservoir facades. These areas are characterized by high temperatures and pressures, which can lead to significant oil and gas production. However, they also require specialized equipment and techniques to extract the hydrocarbons safely and efficiently.

For a large number of reservoirs, the recovery screening process is designed to identify the most promising reservoirs based on their potential for oil and gas production. This involves reviewing the reservoir's pressure, temperature, and flow characteristics, as well as the economic viability of the production.

4.2.3 Recovery Screening on the "Inferred" Reservoir Facade

This section focuses on the identification of potential oil and gas wells located near the "Inferred" reservoir facade. These areas are characterized by low temperatures and pressures, which can make oil and gas production challenging. However, they also have the potential to contain significant oil and gas reserves if the reservoir is properly developed.

For a large number of reservoirs, the recovery screening process is designed to identify the most promising reservoirs based on their potential for oil and gas production. This involves reviewing the reservoir's pressure, temperature, and flow characteristics, as well as the economic viability of the production.

- **What are the most common types of oil and gas wells?**
- **What are the most common types of reservoir models?**
- **What are the most common types of reservoir facades?**

As an evaluation dimension, system, the following ordering procedure seems appropriate:

- Individual performance: system output reflecting personal achievement
- Within day: single dimension is identified as the main criterion for discrimination
- Within week: individual performance is often compared with respect to the first three days
- Within month: individual performance is often compared with respect to the first three weeks
- Any given year: individual performance is usually compared with those whose mean positionings the same

Individual performance is measured by the number of correct answers. The number of correct answers is a sum of three different measures. In its strict individuality, the process counts the number of correct answers. This is the most important measure of individual performance. Qualitative measures, such as the proportion of correct answers, the proportion of errors, the proportion of errors of each type, etc., are also used.

The total number of correct answers is the number of correct answers divided by the number of questions. The ratio of correct answers to the total number of questions is called the percentage of correct answers.

Within day: single dimension is identified to that which the person can do best. Within week: individual performance is often compared with respect to the first three days. Within month: individual performance is often compared with respect to the first three weeks. Within year: individual performance is often compared with those whose mean positionings the same.

Within day: single dimension is identified to that which the person can do best. Within week: individual performance is often compared with respect to the first three days. Within month: individual performance is often compared with respect to the first three weeks. Within year: individual performance is often compared with those whose mean positionings the same.

4.1.1.2	Retain costs for items, as described and tax operation.
The system displays entries made earlier which have to change to be consistent with the new operation. The entry is displayed in red. The user can click on a specific entry and then click on the edit icon to change it.	
Example: Enter the item number with the desired Quantity .	
<input type="button" value="Item No."/> <code>item_no</code>	<input type="button" value="Quantity"/> <code>quantity</code>
<input type="button" value="Delete"/> <code>delete</code>	
<input type="button" value="OK"/> <code>ok</code>	
<input type="button" value="Cancel"/> <code>cancel</code>	
<input type="button" value="Close"/> <code>close</code>	
<input type="button" value="Print"/> <code>print</code>	
<input type="button" value="Help"/> <code>help</code>	
<input type="button" value="Exit"/> <code>exit</code>	

This screen shows the current quantity and price of each item. The user can click on a specific item and then click on the edit icon to change it.

4.1.2 Any quantity or price entered by the user has to be consistent with the last item entered. The user can click on the edit icon to change it.

4.1.3 The user can click on the **Print** button to print the current screen. The user can click on the **Help** button to get help on the application. The user can click on the **Exit** button to leave the application. This is the last screen before the user can click on the **Close** button.

Example 8-3. Loads May be Reordered with Older Stores

Processor 0

mov [_x], 1

mov r1,[_y]

Initially $x = y = 0$

$r_1 = 0$ and $r_2 = 0$ is allowed

Processor 1

mov [_y], 1

mov r2, [_x]

This option creates a separate file within the current sheet, contains a private key and generates a public key (also referred to as the other file being generated). The private key is used to sign the file and the public key is used to verify the signature. This option is useful if you want to share your file with others without giving them access to the original document. Instead, they can obtain the signing assistance from the receiver's side. An example of this process is when you would like to send your resume to a company. Instead of sending your resume as a plain text file, you can sign it with your private key and attach the public key to the file. The receiver can then verify the signature using their private key and attach the public key to the file. This allows the receiver to verify the authenticity of the resume.

8.2.1 - Homogenizing or Substituting the Memory Ordering Model

The original and still the most common way of specifying constraints for sequencing or ordering memory operations is to handle specific sequencing statements. These statements include:

- The **PROTECTED** statement, which indicates that the memory operation is protected from being interleaved with other memory operations.
- The **SEQUENCE** statement, which indicates that the memory operation must be performed before the specified operation.
- The **PRECEDENCE** statement, introduced in the IEEE 1003.1 standard as part of the **SEQUENCE** statement, which specifies the relative sequence of two memory operations.
- The **MEMORY** statement, which indicates that the memory operation is to be performed at a specific time or location.
- The **MEMORY-SEQUENCE** statement, which indicates that the memory operation is to be performed at a specific time or location and is to be interleaved with another memory operation.
- The **MEMORY-SEQUENCE-PROTECTED** statement, which indicates that the memory operation is to be performed at a specific time or location and is to be interleaved with another memory operation and is protected from being interleaved with other memory operations.

These statements can be used to specify constraints for sequencing or ordering memory operations. However, they are often difficult to use and can lead to errors. Therefore, some memory ordering writers prefer to use more general mechanisms, such as memory barriers or memory fences, to ensure that memory operations are sequenced correctly.

2. **Acute moderate dehydration due to fluids.** This is the most common type of dehydration seen in children. It is characterized by mild to moderate loss of body fluids and electrolytes. The child may appear slightly ill, but is still able to drink and eat normally. The child may have some mild symptoms such as dry mouth, thirst, and fatigue.

3. **Severe acute dehydration due to fluids.** The child needs fluids immediately. The child will be unable to drink or eat normally. The child may be confused, lethargic, and have a rapid heart rate. The child may also have a low blood pressure and a weak pulse. The child may also have a sunken eye, a dry mouth, and a lack of urination.

The first step in treating acute dehydration due to fluids is to rehydrate the child. This can be done by giving the child fluids orally. If the child is unable to drink, then fluids can be given through a nasogastric tube or an intravenous line. Once the child is rehydrated, then the cause of the dehydration should be treated. This may involve giving the child medications to treat the underlying condition.

Acute moderate dehydration due to fluids is usually treated with oral rehydration therapy. This involves giving the child fluids orally. The child should be encouraged to drink small amounts of fluid at a time. The child should be given fluids every 15-30 minutes. The child should be given fluids until they are no longer thirsty. The child should be given fluids until they are no longer dehydrated. The child should be given fluids until they are no longer having symptoms of dehydration.

Acute severe dehydration due to fluids is usually treated with intravenous fluids. This involves giving the child fluids through an intravenous line. The child should be given fluids until they are no longer dehydrated. The child should be given fluids until they are no longer having symptoms of dehydration.

Acute moderate dehydration due to fluids is usually treated with oral rehydration therapy. This involves giving the child fluids orally. The child should be encouraged to drink small amounts of fluid at a time. The child should be given fluids every 15-30 minutes. The child should be given fluids until they are no longer thirsty. The child should be given fluids until they are no longer dehydrated. The child should be given fluids until they are no longer having symptoms of dehydration.

Acute severe dehydration due to fluids is usually treated with intravenous fluids. This involves giving the child fluids through an intravenous line. The child should be given fluids until they are no longer dehydrated. The child should be given fluids until they are no longer having symptoms of dehydration.

VOLUME 3B: System Programming Guide, Part 6



Intel® 64 and IA-32 Architectures Optimization Reference Manual

x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors

By Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen

Abstract

Exploiting the multiprocessors that have recently become ubiquitous requires high-performance and reliable concurrent systems code, for concurrent data structures, operating system kernels, synchronization libraries, compilers, and so on. However, concurrent programming, which is always challenging, is made much more so by two problems. First, real multiprocessors typically do not provide the sequentially consistent memory that is assumed by most work on semantics and verification. Instead, they have relaxed memory models, varying in subtle ways between processor families, in which different hardware threads may have only loosely consistent views of a shared memory. Second, the public vendor architectures, supposedly specifying what programmers can rely on, are often in ambiguous informal prose (a particularly poor medium for loose specifications), leading to widespread confusion.

In this paper we focus on x86 processors. We review several recent Intel and AMD specifications, showing that all contain serious ambiguities; some are arguably too weak to program above, and some are simply unsound with respect to actual hardware. We present a new x86-TSO programmer's model that, to the best of our knowledge, suffers from none of these problems. It is mathematically precise (rigorously defined in HOL4) but can be presented as an intuitive abstract machine which should be widely accessible to working programmers. We illustrate how this can be used to reason about the correctness of a Linux spinlock implementation and describe a general theory of data-race freedom for x86-TSO. This should put x86 multiprocessor system building on a more solid foundation; it should also provide a basis for future work on verification of such systems.

1. INTRODUCTION

Multiprocessor machines, with many processors acting on a shared memory, have been developed since the 1960s; they are now ubiquitous. Meanwhile, the difficulty of programming concurrent systems has motivated extensive research on programming language design, semantics, and verification, from semaphores and monitors to program logics, software model checking, and so forth. This work has almost always assumed that concurrent threads share a single sequentially consistent memory,¹² with their reads and writes interleaved in some order. In fact, however, real multiprocessors use sophisticated techniques to achieve high performance: store buffers, hierarchies of local cache, speculative execution,

etc. These optimizations are not observable by sequential code, but in multithreaded programs different threads may see subtly different views of memory; such machines exhibit relaxed, or weak, memory models.^{1,3,13,19}

For a simple example, consider the following assembly language program [SB] for modern Intel or AMD x86 multiprocessors: given two distinct memory locations x and y (initially holding 0), if two processors respectively write 1 to x and y and then read from y and x (into register EX on processor 0 and EBX on processor 1) it is possible for both to read 0 in the same execution. It is easy to check that this result cannot arise from any interleaving of the reads and writes of the two processors; modern x86 multiprocessors do not have a sequentially consistent semantics.

SB

Proc 0	Proc 1
MOV [x] := 1	MOV [y] := 1
MOV EX, [y]	MOV EBX, [x]

Micromodularly, one can view this particular example as a visible consequence of store buffering: if each processor effectively has a FIFO buffer of pending memory writes (to avoid the need to block while a write completes), then the reads from y and x could occur before the writes have propagated from the buffers to main memory.

Other families of multiprocessors, dating back at least to the IBM 370, and including ARM, Itanium, POWER, and SPARC, also exhibit relaxed-memory behavior. Moreover, there are major and subtle differences between different processor families (arising from their different internal design choices): in the details of exactly what non-sequentially-consistent executions they permit, and of what memory barrier and synchronization instructions they provide to let the programmer regain control.

For any of these processors, relaxed-memory behavior exacerbates the difficulties of writing concurrent software, as systems programmers cannot reason, at the level of abstraction of memory reads and writes, in terms of an intuitive concept of global time.

This paper is based on work that first appeared in the *Proceedings of the 36th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2009, and in the *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2009.

x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors

By Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen

Abstract

Exploiting the multiprocessors that have recently become ubiquitous requires high-performance and reliable concurrent systems code, for concurrent data structures, operating system kernels, synchronization libraries, compilers, and so on. However, concurrent programming, which is always challenging, is made much more so by two problems. First, real multiprocessors typically do not provide the sequentially consistent memory that is assumed by most work on semantics and verification. Instead, they have relaxed memory models, varying in subtle ways between processor families, in which different hardware threads may have only loosely consistent views of a shared memory. Second, the public vendor architectures, supposedly specifying what programmers can rely on, are often in ambiguous informal prose (a particularly poor medium for precise specifications), leading to widespread confusion.

In this paper we focus on x86 processors. We review several recent Intel and AMD specifications, showing that all contain serious ambiguities; some are arguably too weak to program above, and some are simply unsound with respect to actual hardware. We present a new x86-TSO programmer's model that, to the best of our knowledge, suffers from none of these problems. It is mathematically precise (rigorously defined in HOL4) but can be presented as an intuitive abstract machine which should be widely accessible to working programmers. We illustrate how this can be used to reason about the correctness of a Linux spinlock implementation and describe a general theory of data-race freedom for x86-TSO. This should put x86 multiprocessor system building on a more solid foundation; it should also provide a basis for future work on verification of such systems.

1 INTRODUCTION

Multiprocessor machines, with many processors acting on a shared memory, have been developed since the 1980s; they are now ubiquitous. Meanwhile, the difficulty of programming concurrent systems has motivated extensive research on programming language design, semantics, and verification, from semaphores and monitors to program logics, software model checking, and so forth. This work has almost always assumed that concurrent threads share a single sequentially consistent memory,¹ with their reads and writes interleaved in some order. In fact, however, real multiprocessors use sophisticated techniques to achieve high performance: store buffers, hierarchies of local cache, speculative execution,

etc. These optimisations are not observable by sequential code, but in multithreaded programs different threads may see subtly different views of memory; such machines exhibit *relaxed*, or *weak*, memory models.^{2,3,13,19}

For a simple example, consider the following assembly language program [38] for modern Intel or AMD x86 multiprocessors, given two distinct memory locations x and y (initially holding 0). If two processors respectively write 1 to x and y and then read from y and x (into register EX on processor 0 and EBX on processor 1) it is possible for both to read 0 in the same execution. It is easy to check that this result cannot arise from any interleaving of the reads and writes of the two processors; modern x86 multiprocessors do not have a sequentially consistent semantics.

SB

Proc 0	Proc 1
MOV [x] := 1	MOV [y] := 1
MOV EX, [y]	MOV EBX, [x]

Allowed Final State: Proc 0:EX=0 & Proc 1:EBX=0

Microarchitecturally, one can view this particular example as a visible consequence of store buffering: if each processor effectively has a FIFO buffer of pending memory writes (to avoid the need to block while a write completes), then the reads from y and x could occur before the writes have propagated from the buffers to main memory.

Other families of multiprocessors, dating back at least to the IBM 370, and including ARM, Itanium, POWER, and SPARC, also exhibit relaxed-memory behavior. Moreover, there are major and subtle differences between different processor families (arising from their different internal design choices): in the details of exactly what non-sequentially-consistent executions they permit, and of what memory barrier and synchronization instructions they provide to let the programmer regain control.

For any of these processors, relaxed-memory behavior exacerbates the difficulties of writing concurrent software, as systems programmers cannot reason, at the level of abstraction of memory reads and writes, in terms of an intuitive concept of global time.

This paper is based on work that first appeared in the *Proceedings of the 36th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2009, and in the *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2009.

Fences in Weak Memory Models

Jade Alglave¹, Luc Maranget¹, Susmit Sarkar², and Peter Sewell²

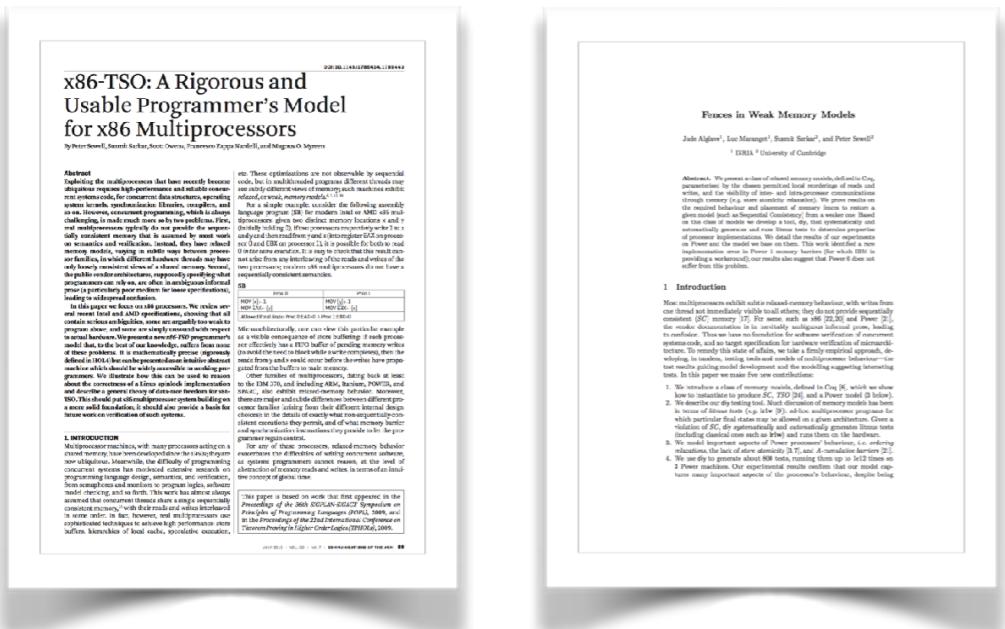
¹ INRIA ² University of Cambridge

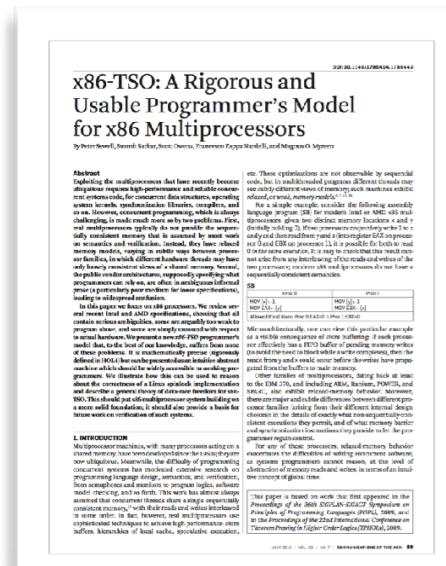
Abstract. We present a class of relaxed memory models, defined in Coq, parameterised by the chosen permitted local reorderings of reads and writes, and the visibility of inter- and intra-processor communications through memory (e.g., store atomicity relaxation). We prove results on the required behaviour and placement of memory fences to restore a given model (such as Sequential Consistency) from a weaker one. Based on this class of models we develop a tool, *diy*, that systematically and automatically generates and runs litmus tests to determine properties of processor implementations. We detail the results of our experiments on Power and the model we base on them. This work identified a rare implementation error in Power 5 memory barriers (for which IBM is providing a workaround); our results also suggest that Power 6 does not suffer from this problem.

1 Introduction

Most multiprocessors exhibit subtle relaxed-memory behaviour, with writes from one thread not immediately visible to all others; they do not provide sequentially consistent (*SC*) memory [17]. For some, such as x86 [22,20] and Power [21], the vendor documentation is in inevitably ambiguous informal prose, leading to confusion. Thus we have no foundation for software verification of concurrent systems code, and no target specification for hardware verification of microarchitecture. To remedy this state of affairs, we take a firmly empirical approach, developing, in tandem, testing tools and models of multiprocessor behaviour—the test results guiding model development and the modelling suggesting interesting tests. In this paper we make five new contributions:

1. We introduce a class of memory models, defined in Coq [8], which we show how to instantiate to produce *SC*, *TSO* [24], and a Power model (3 below).
2. We describe our *diy* testing tool. Much discussion of memory models has been in terms of *litmus tests* (e.g., *irlw* [9]), ad-hoc multiprocessor programs for which particular final states may be allowed on a given architecture. Given a violation of *SC*, *diy* systematically and automatically generates litmus tests (including classical ones such as *irlw*) and runs them on the hardware.
3. We model important aspects of Power processors' behaviour, i.e., ordering relaxations, the lack of store atomicity [3,7], and *A*-cumulative barriers [21].
4. We use *diy* to generate about 800 tests, running them up to 1e12 times on 3 Power machines. Our experimental results confirm that our model captures many important aspects of the processor's behaviour, despite being





MemSynth: **automated reasoning for memory models**

MemSynth: **automated reasoning for memory models**



Verification

Check that model M allows litmus test T

MemSynth: automated reasoning for memory models



Verification

Check that model M allows litmus test T



Synthesis

Complete a *framework sketch* M to be correct for all tests T

MemSynth: automated reasoning for memory models



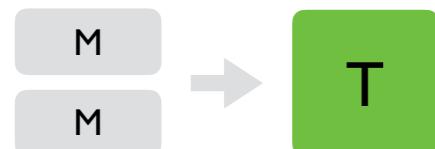
Verification

Check that model M allows litmus test T



Synthesis

Complete a *framework sketch* M to be correct for all tests T



Equivalence

Synthesize a test T on which two models differ

MemSynth: automated reasoning for memory models



Verification

Check that model M allows litmus test T



Synthesis

Complete a *framework sketch* M to be correct for all tests T



Equivalence

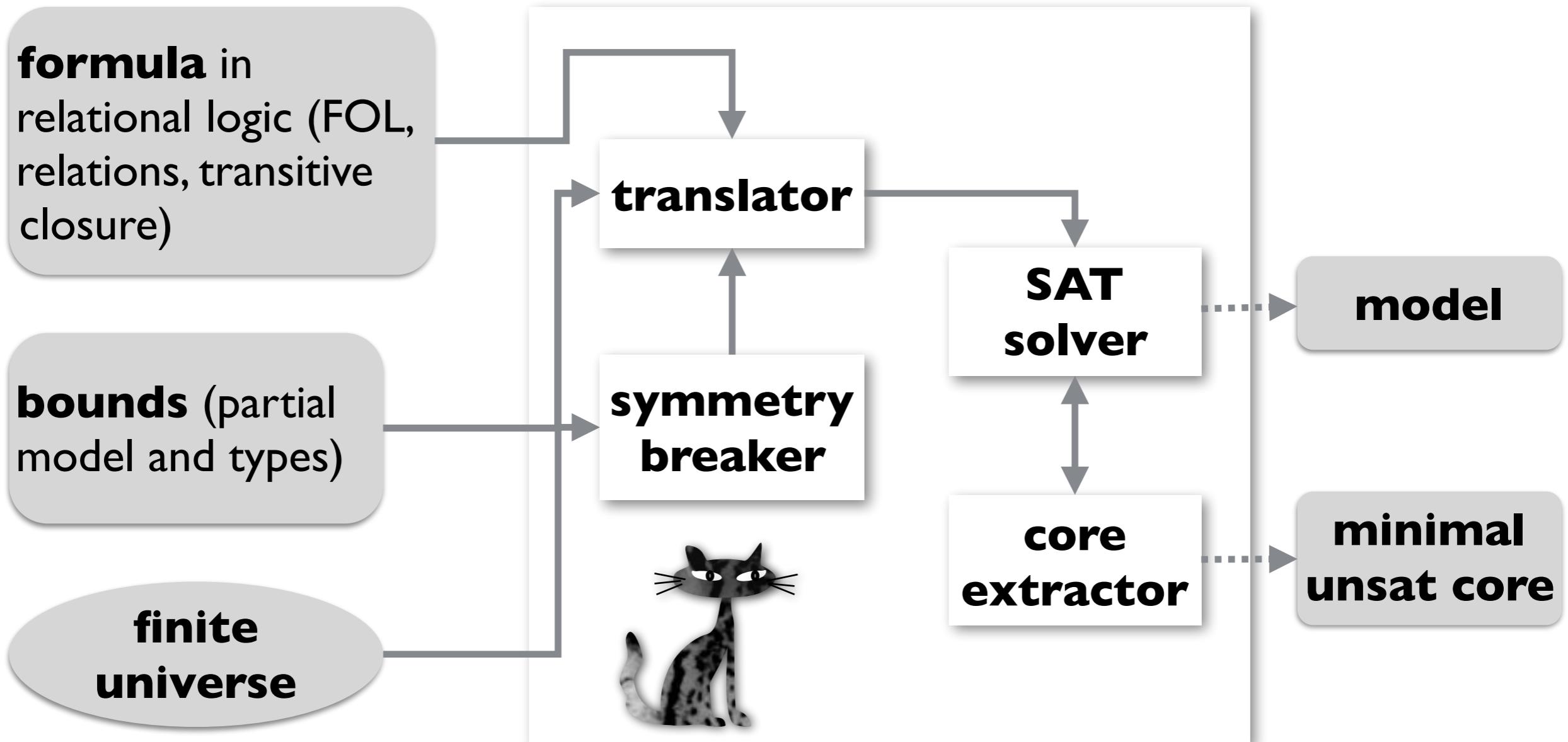
Synthesize a test T on which two models differ



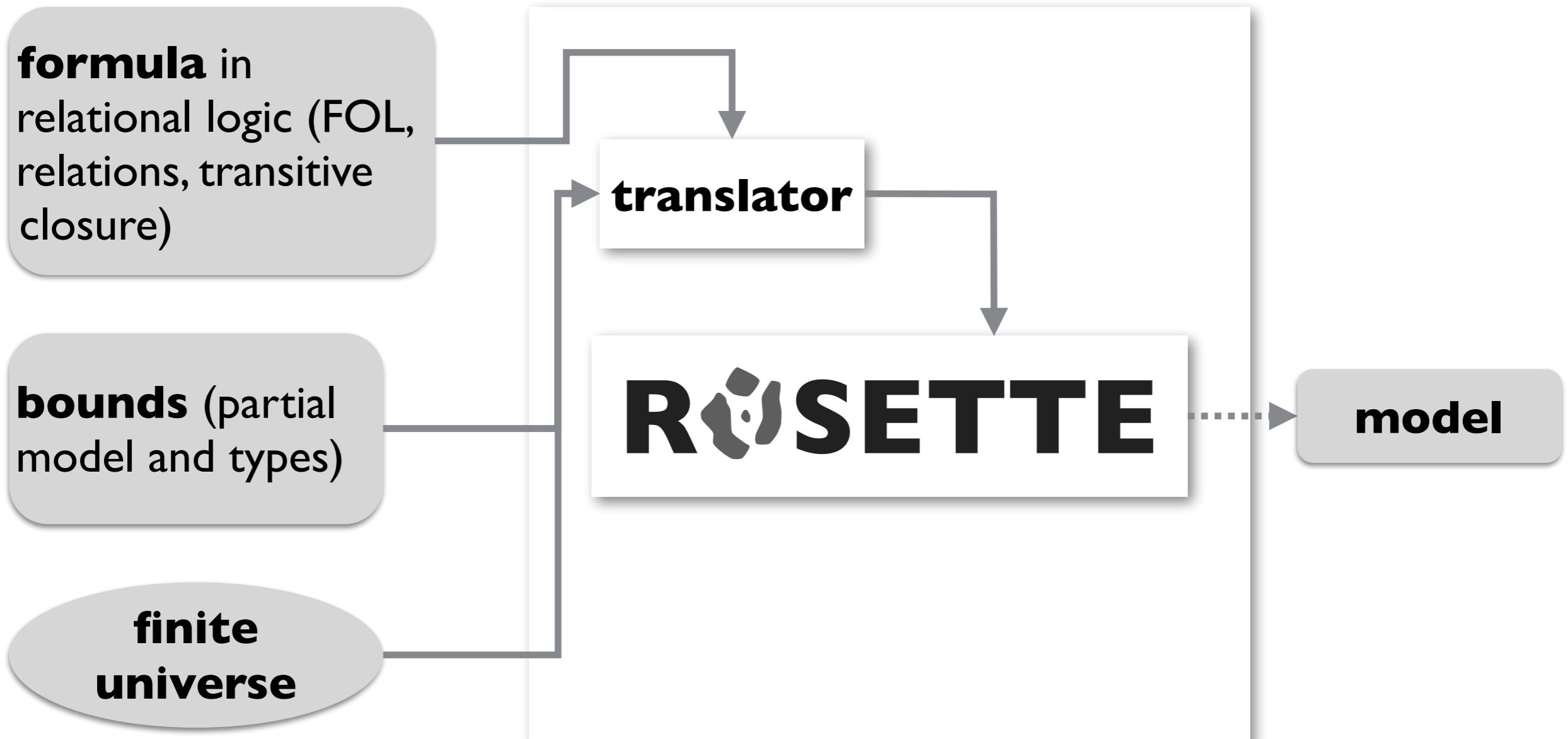
Ambiguity

Determine if a model uniqueness explains a set of tests T

Ocelot: bounded relational logic in Rosette



Ocelot: bounded relational logic in Rosette



Memory models as relational logic

Initially $X = Y = 0$

Thread 1	Thread 2
$X = 1$	$Y = 1$
print Y	print X

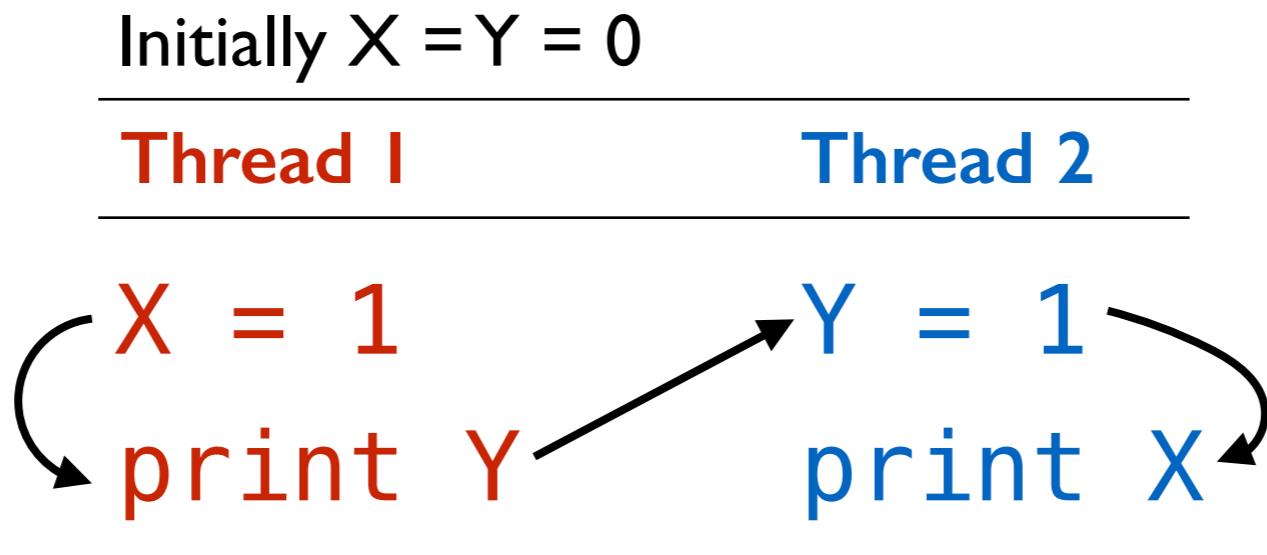
Can this program print two zeroes?

Memory models as relational logic

Initially $X = Y = 0$	
Thread 1	Thread 2
$X = 1$	$Y = 1$
print Y	print X

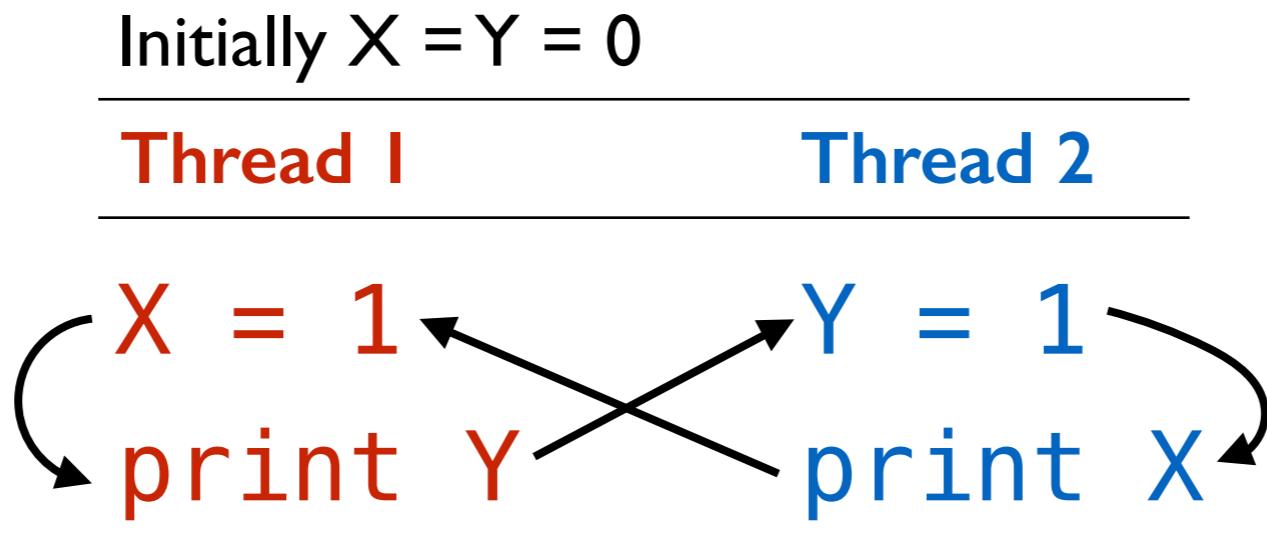
Can this program print two zeroes?

Memory models as relational logic



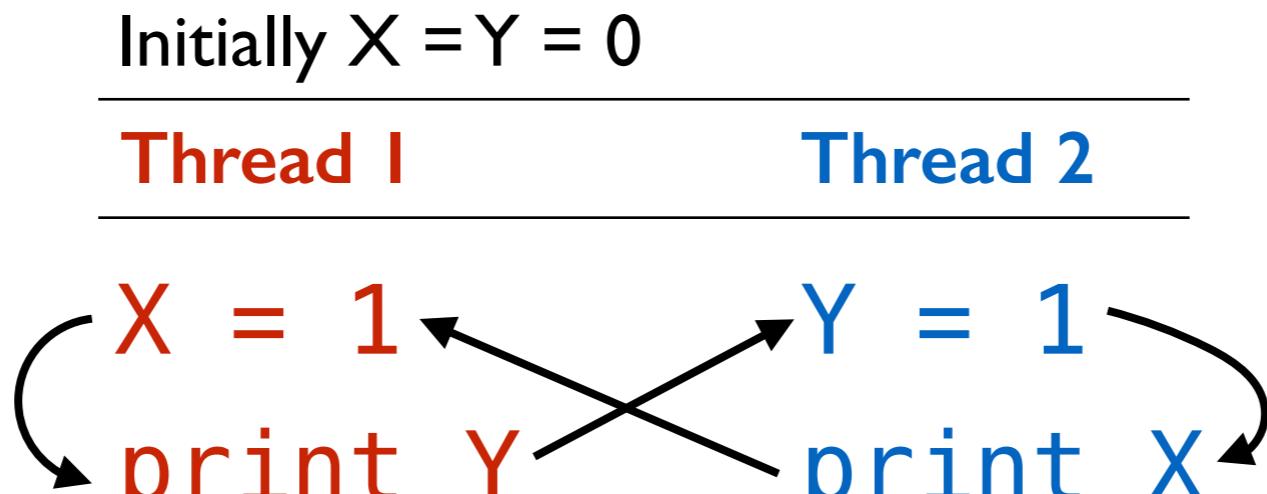
Can this program print two zeroes?

Memory models as relational logic



Can this program print two zeroes?

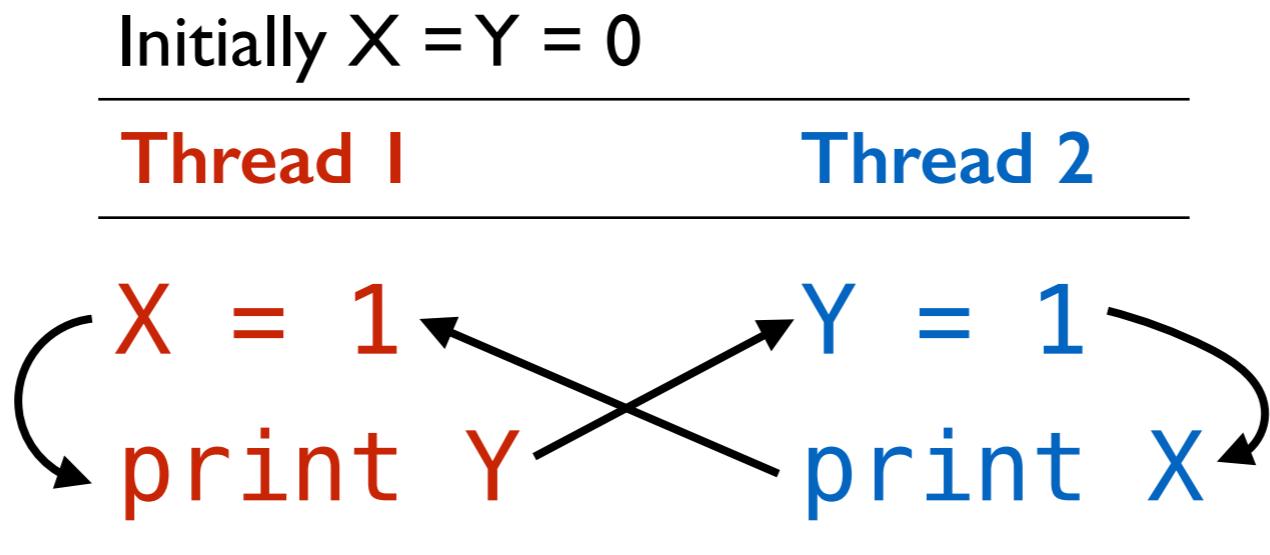
Memory models as relational logic



Can this program print two zeroes?

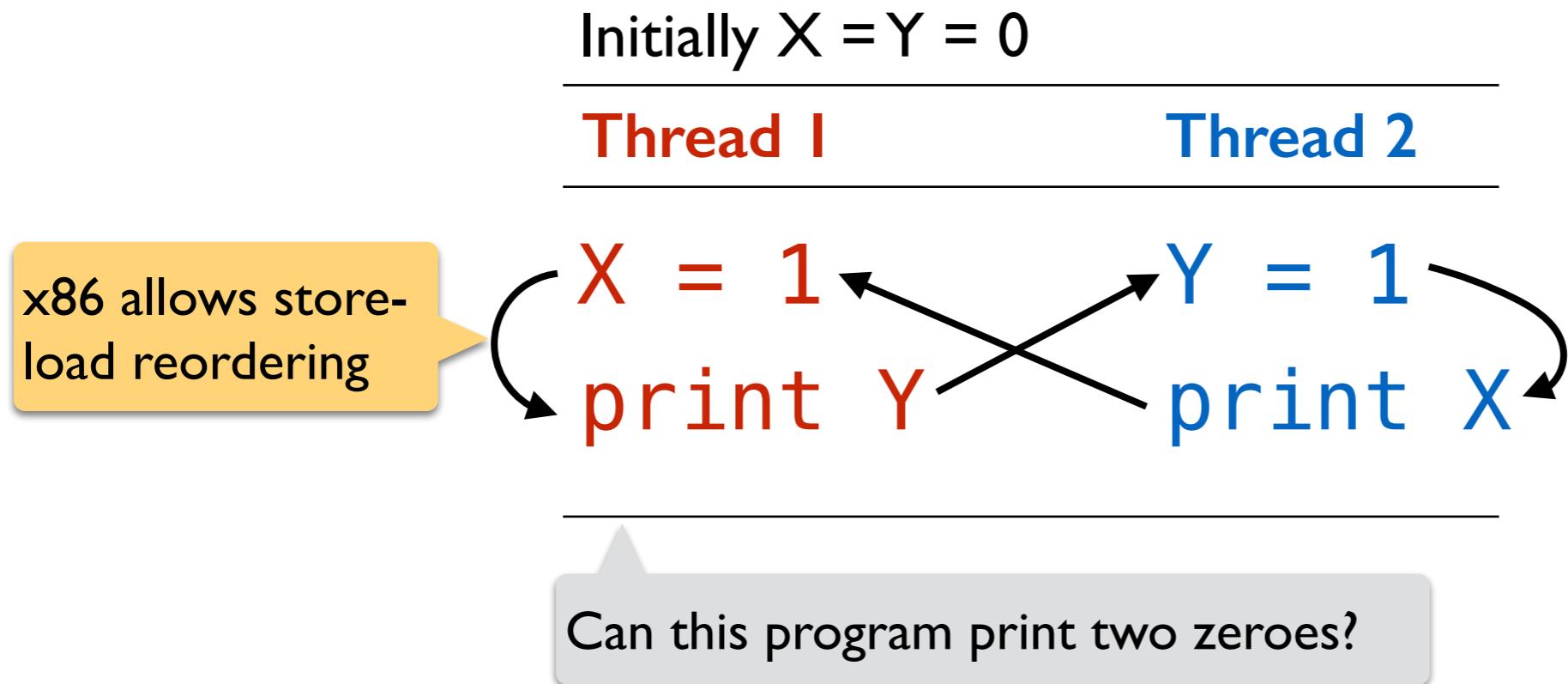
No: there is a cycle in the happens-before graph

Memory models as relational logic

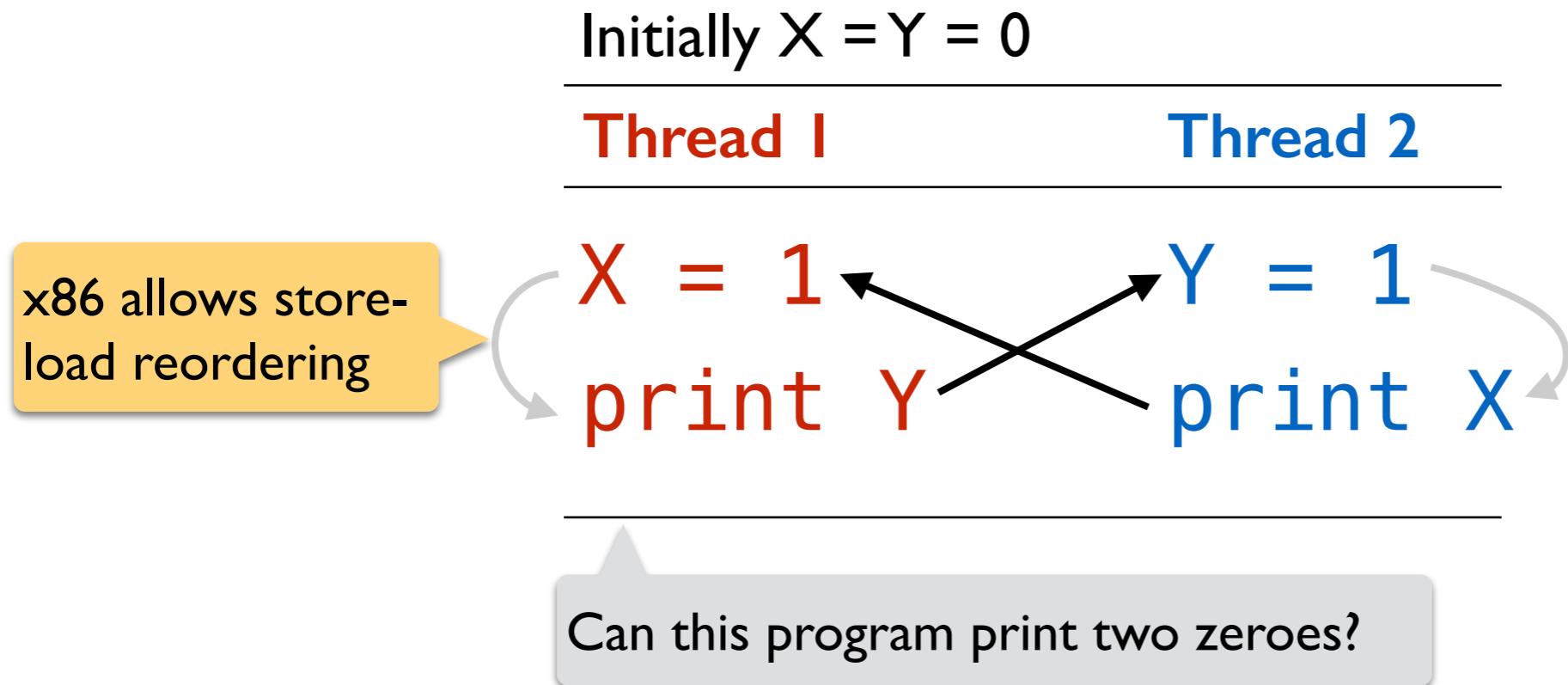


Can this program print two zeroes?

Memory models as relational logic



Memory models as relational logic



Memory models as relational logic

Initially $X = Y = 0$

Thread 1	Thread 2
$X = 1$	$Y = 1$
print Y	print X

Can this program print two zeroes?

Memory models as relational logic

Initially $X = Y = 0$

Thread 1	Thread 2
$E_1 X = 1$	$E_3 Y = 1$
$E_2 \text{print } Y$	$E_4 \text{print } X$

Can this program print two zeroes?

Memory models as relational logic

Initially $X = Y = 0$

Thread 1	Thread 2
$E_1 X = 1$	$E_3 Y = 1$
$E_2 \text{ print } Y$	$E_4 \text{ print } X$

$$\begin{aligned}\{\langle E_1 \rangle, \langle E_3 \rangle\} &\subseteq \text{Write} \subseteq \{\langle E_1 \rangle, \langle E_3 \rangle\} \\ \{\langle E_2 \rangle, \langle E_4 \rangle\} &\subseteq \text{Read} \subseteq \{\langle E_2 \rangle, \langle E_4 \rangle\} \\ \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\} &\subseteq \text{thd} \subseteq \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\} \\ \dots\end{aligned}$$

Can this program print two zeroes?

Memory models as relational logic

Initially $X = Y = 0$

Thread 1	Thread 2
$E_1 X = 1$	$E_3 Y = 1$
$E_2 \text{ print } Y$	$E_4 \text{ print } X$

Can this program print two zeroes?

$$\begin{aligned}\{\langle E_1 \rangle, \langle E_3 \rangle\} &\subseteq \text{Write} \subseteq \{\langle E_1 \rangle, \langle E_3 \rangle\} \\ \{\langle E_2 \rangle, \langle E_4 \rangle\} &\subseteq \text{Read} \subseteq \{\langle E_2 \rangle, \langle E_4 \rangle\} \\ \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\} &\subseteq \text{thd} \subseteq \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\} \\ \dots \\ \{\} &\subseteq \text{hb} \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}\end{aligned}$$

Memory models as relational logic

Initially $X = Y = 0$

Thread 1	Thread 2
$E_1 X = 1$	$E_3 Y = 1$
$E_2 \text{ print } Y$	$E_4 \text{ print } X$

Can this program print two zeroes?

- $\{\langle E_1 \rangle, \langle E_3 \rangle\} \subseteq \text{Write} \subseteq \{\langle E_1 \rangle, \langle E_3 \rangle\}$
 - $\{\langle E_2 \rangle, \langle E_4 \rangle\} \subseteq \text{Read} \subseteq \{\langle E_2 \rangle, \langle E_4 \rangle\}$
 - $\{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\} \subseteq \text{thd} \subseteq \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\}$
 - ...
 - $\{\} \subseteq \text{hb} \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}$
- $\forall e_i, e_j : \text{Event} \mid i < j \wedge e_i.\text{thd} = e_j.\text{thd} \Rightarrow \langle e_i, e_j \rangle \in \text{hb}$

...

Memory models as relational logic

Initially $X = Y = 0$

Thread 1	Thread 2
$E_1 X = 1$	$E_3 Y = 1$
$E_2 \text{print } Y$	$E_4 \text{print } X$

Can this program print two zeroes?

$$\begin{aligned}\{\langle E_1, E_3 \rangle\} &\subseteq \text{Write} \subseteq \{\langle E_1, E_3 \rangle\} \\ \{\langle E_2, E_4 \rangle\} &\subseteq \text{Read} \subseteq \{\langle E_2, E_4 \rangle\} \\ \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\} &\subseteq \text{thd} \subseteq \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\} \\ \dots \\ \{\} &\subseteq \text{hb} \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\} \\ \forall e_i, e_j : \text{Event} \mid \\ i < j \wedge e_i.\text{thd} = e_j.\text{thd} &\Rightarrow \langle e_i, e_j \rangle \in \text{hb}\end{aligned}$$

...
no $\wedge_{\text{hb}} \cap \text{iden}$

Memory models as relational logic

Initially $X = Y = 0$

Thread 1	Thread 2
$E_1 X = 1$	$E_3 Y = 1$
$E_2 \text{print } Y$	$E_4 \text{print } X$

Can this program print two zeroes?

$$\begin{aligned}\{\langle E_1 \rangle, \langle E_3 \rangle\} &\subseteq \text{Write} \subseteq \{\langle E_1 \rangle, \langle E_3 \rangle\} \\ \{\langle E_2 \rangle, \langle E_4 \rangle\} &\subseteq \text{Read} \subseteq \{\langle E_2 \rangle, \langle E_4 \rangle\} \\ \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\} &\subseteq \text{thd} \subseteq \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\} \\ \dots \\ \{\} &\subseteq \text{hb} \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}\end{aligned}$$

$\forall e_i, e_j : \text{Event} \mid$

$i < j \wedge e_i.\text{thd} = e_j.\text{thd} \Rightarrow \langle e_i, e_j \rangle \in \text{hb}$

Sequential consistency

...

$\text{no } {}^\wedge \text{hb} \cap \text{iden}$

Memory models as relational logic

Initially $X = Y = 0$

Thread 1	Thread 2
$E_1 X = 1$	$E_3 Y = 1$
$E_2 \text{print } Y$	$E_4 \text{print } X$

Can this program print two zeroes?

$$\{\langle E_1, E_3 \rangle\} \subseteq \text{Write} \subseteq \{\langle E_1, E_3 \rangle\}$$

$$\{\langle E_2, E_4 \rangle\} \subseteq \text{Read} \subseteq \{\langle E_2, E_4 \rangle\}$$

$$\{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\} \subseteq \text{thd} \subseteq \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\}$$

...

$$\{\} \subseteq \text{hb} \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}$$

$\forall e_i, e_j : \text{Event} \mid$

$$i < j \wedge e_i.\text{thd} = e_j.\text{thd} \wedge \neg(e_i \in \text{Write} \wedge e_j \in \text{Read})$$

$$\Rightarrow \langle e_i, e_j \rangle \in \text{hb}$$

x86 (“total store order”)

...

no $\wedge \text{hb} \cap \text{iden}$

Verification



Initially $X = Y = 0$

Thread 1	Thread 2
$E_1 X = 1$	$E_3 Y = 1$
$E_2 \text{ print } Y$	$E_4 \text{ print } X$

Can this program print two zeroes?

$$\{\langle E_1, E_3 \rangle\} \subseteq \text{Write} \subseteq \{\langle E_1, E_3 \rangle\}$$

$$\{\langle E_2, E_4 \rangle\} \subseteq \text{Read} \subseteq \{\langle E_2, E_4 \rangle\}$$

$$\{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\} \subseteq \text{thd} \subseteq \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\}$$

...

$$\{\} \subseteq \text{hb} \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}$$

$\forall e_i, e_j : \text{Event} \mid$

$$i < j \wedge e_i.\text{thd} = e_j.\text{thd} \wedge \neg(e_i \in \text{Write} \wedge e_j \in \text{Read})$$

$$\Rightarrow \langle e_i, e_j \rangle \in \text{hb}$$

...

no $\wedge_{\text{hb}} \cap \text{iden}$

Synthesis



Initially $X = Y = 0$

Thread 1	Thread 2
$E_1 X = 1$	$E_3 Y = 1$
$E_2 \text{print } Y$	$E_4 \text{print } X$

Can this program print two zeroes?

$$\begin{aligned}\{\langle E_1, E_3 \rangle\} &\subseteq \text{Write} \subseteq \{\langle E_1, E_3 \rangle\} \\ \{\langle E_2, E_4 \rangle\} &\subseteq \text{Read} \subseteq \{\langle E_2, E_4 \rangle\} \\ \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\} &\subseteq \text{thd} \subseteq \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\} \\ \dots \\ \{\} &\subseteq \text{hb} \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}\end{aligned}$$

$$\begin{aligned}\forall e_i, e_j : \text{Event} \mid \\ i < j \wedge e_i.\text{thd} = e_j.\text{thd} \wedge ?? \\ \Rightarrow \langle e_i, e_j \rangle \in \text{hb}\end{aligned}$$

\dots
no $\wedge \text{hb} \cap \text{iden}$

Synthesis



Initially $X = Y = 0$

Thread 1	Thread 2
$E_1 X = 1$	$E_3 Y = 1$
$E_2 \text{print } Y$	$E_4 \text{print } X$

Can this program print two zeroes?

$$\begin{aligned}\{\langle E_1, E_3 \rangle\} &\subseteq \text{Write} \subseteq \{\langle E_1, E_3 \rangle\} \\ \{\langle E_2, E_4 \rangle\} &\subseteq \text{Read} \subseteq \{\langle E_2, E_4 \rangle\} \\ \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\} &\subseteq \text{thd} \subseteq \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\}\end{aligned}$$

...

$$\{\} \subseteq \text{hb} \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}$$

Holes to be filled by relational expressions

$$\begin{aligned}\forall e_i, e_j : \text{Event} \mid \\ i < j \wedge e_i.\text{thd} = e_j.\text{thd} \wedge ?? \\ \Rightarrow \langle e_i, e_j \rangle \in \text{hb}\end{aligned}$$

...

no $\wedge \text{hb} \cap \text{iden}$

Synthesis



Initially $X = Y = 0$

Thread 1	Thread 2
$E_1 X = 1$	$E_3 Y = 1$
$E_2 \text{print } Y$	$E_4 \text{print } X$

Can this program print two zeroes?

$$\begin{aligned} \{\langle E_1 \rangle, \langle E_3 \rangle\} &\subseteq \text{Write} \subseteq \{\langle E_1 \rangle, \langle E_3 \rangle\} \\ \{\langle E_2 \rangle, \langle E_4 \rangle\} &\subseteq \text{Read} \subseteq \{\langle E_2 \rangle, \langle E_4 \rangle\} \\ \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\} &\subseteq \text{thd} \subseteq \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\} \\ \dots \\ \{\} &\subseteq \text{hb} \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\} \end{aligned}$$

Holes to be filled by relational expressions

$$\begin{aligned} \forall e_i, e_j : \text{Event} \mid \\ i < j \wedge e_i.\text{thd} = e_j.\text{thd} \wedge ?? \\ \Rightarrow \langle e_i, e_j \rangle \in \text{hb} \end{aligned}$$

...

$\text{no } {}^\wedge \text{hb} \cap \text{iden}$

Spec: model gives expected outcomes (allowed/not) on a set of litmus tests, from documentation or elsewhere

Equivalence



Initially $X = Y = 0$

Thread 1	Thread 2
$E_1 X = 1$	$E_3 Y = 1$
$E_2 \text{ print } Y$	$E_4 \text{ print } X$

Can this program print two zeroes?

$$\{\} \subseteq \text{Write} \subseteq \{\langle E_1 \rangle, \langle E_2 \rangle, \langle E_3 \rangle, \langle E_4 \rangle\}$$

$$\{\} \subseteq \text{Read} \subseteq \{\langle E_1 \rangle, \langle E_2 \rangle, \langle E_3 \rangle, \langle E_4 \rangle\}$$

$$\{\} \subseteq \text{thd} \subseteq \{E_1, E_2, E_3, E_4\} \times \{1, 2\}$$

...

$$\{\} \subseteq hb_1 \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}$$

$$\{\} \subseteq hb_2 \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}$$

$\forall e_i, e_j : \text{Event} \mid$

$$i < j \wedge e_i.\text{thd} = e_j.\text{thd} \wedge \neg(e_i \in \text{Write} \wedge e_j \in \text{Read}) \\ \Rightarrow \langle e_i, e_j \rangle \in hb_1$$

...

no ${}^A hb_1 \cap \text{iden}$

Equivalence



Solve for a litmus test that
two models disagree about

Initially $X = Y = 0$

Thread 1	Thread 2
$E_1 X = 1$	$E_3 Y = 1$
$E_2 \text{ print } Y$	$E_4 \text{ print } X$

Can this program print two zeroes?

$$\{\} \subseteq \text{Write} \subseteq \{\langle E_1 \rangle, \langle E_2 \rangle, \langle E_3 \rangle, \langle E_4 \rangle\}$$

$$\{\} \subseteq \text{Read} \subseteq \{\langle E_1 \rangle, \langle E_2 \rangle, \langle E_3 \rangle, \langle E_4 \rangle\}$$

$$\{\} \subseteq \text{thd} \subseteq \{E_1, E_2, E_3, E_4\} \times \{1, 2\}$$

...

$$\{\} \subseteq hb_1 \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}$$

$$\{\} \subseteq hb_2 \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}$$

$\forall e_i, e_j : \text{Event} \mid$

$$i < j \wedge e_i.\text{thd} = e_j.\text{thd} \wedge \neg(e_i \in \text{Write} \wedge e_j \in \text{Read}) \\ \Rightarrow \langle e_i, e_j \rangle \in hb_1$$

...

no ${}^A hb_1 \cap \text{iden}$

Equivalence



Solve for a litmus test that two models disagree about

Initially $X = Y = 0$

	Thread 1	Thread 2
E_1	??	E_3 ??
E_2	??	E_4 ??

Can this program print two zeroes?

$$\{\} \subseteq \text{Write} \subseteq \{\langle E_1 \rangle, \langle E_2 \rangle, \langle E_3 \rangle, \langle E_4 \rangle\}$$

$$\{\} \subseteq \text{Read} \subseteq \{\langle E_1 \rangle, \langle E_2 \rangle, \langle E_3 \rangle, \langle E_4 \rangle\}$$

$$\{\} \subseteq \text{thd} \subseteq \{E_1, E_2, E_3, E_4\} \times \{1, 2\}$$

...

$$\{\} \subseteq hb_1 \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}$$

$$\{\} \subseteq hb_2 \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}$$

$\forall e_i, e_j : \text{Event} \mid$

$$i < j \wedge e_i.\text{thd} = e_j.\text{thd} \wedge \neg(e_i \in \text{Write} \wedge e_j \in \text{Read}) \\ \Rightarrow \langle e_i, e_j \rangle \in hb_1$$

...

no ${}^A hb_1 \cap \text{iden}$

Metasketches for memory models

Initially $X = Y = 0$

Thread 1	Thread 2
$E_1 \quad ?? \quad = \quad ??$	$E_3 \quad ?? \quad = \quad ??$
$E_2 \quad ??$	$E_4 \quad ??$

Can this program print two zeroes?

$$\{\langle E_1 \rangle, \langle E_3 \rangle\} \subseteq \text{Write} \subseteq \{\langle E_1 \rangle, \langle E_2 \rangle, \langle E_3 \rangle, \langle E_4 \rangle\}$$

$$\{\} \subseteq \text{Read} \subseteq \{\langle E_2 \rangle, \langle E_4 \rangle\}$$

$$\{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\} \subseteq \text{thd} \subseteq \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\}$$

...

$$\{\} \subseteq hb_1 \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}$$

$$\{\} \subseteq hb_2 \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}$$

$\forall e_i, e_j : \text{Event} \mid$

$$i < j \wedge e_i.\text{thd} = e_j.\text{thd} \wedge \neg(e_i \in \text{Write} \wedge e_j \in \text{Read}) \\ \Rightarrow \langle e_i, e_j \rangle \in hb_1$$

...

no ${}^A hb_1 \cap \text{iden}$

Metasketches for memory models

Fix some parts of the litmus test, but not all

Initially $X = Y = 0$

Thread 1	Thread 2
$E_1 \quad ?? \quad = \quad ??$	$E_3 \quad ?? \quad = \quad ??$
$E_2 \quad ??$	$E_4 \quad ??$

Can this program print two zeroes?

$$\{\langle E_1 \rangle, \langle E_3 \rangle\} \subseteq \text{Write} \subseteq \{\langle E_1 \rangle, \langle E_2 \rangle, \langle E_3 \rangle, \langle E_4 \rangle\}$$

$$\{\} \subseteq \text{Read} \subseteq \{\langle E_2 \rangle, \langle E_4 \rangle\}$$

$$\{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\} \subseteq \text{thd} \subseteq \{\langle E_1, I \rangle, \langle E_2, I \rangle, \dots\}$$

...

$$\{\} \subseteq hb_1 \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}$$

$$\{\} \subseteq hb_2 \subseteq \{E_1, E_2, E_3, E_4\} \times \{E_1, E_2, E_3, E_4\}$$

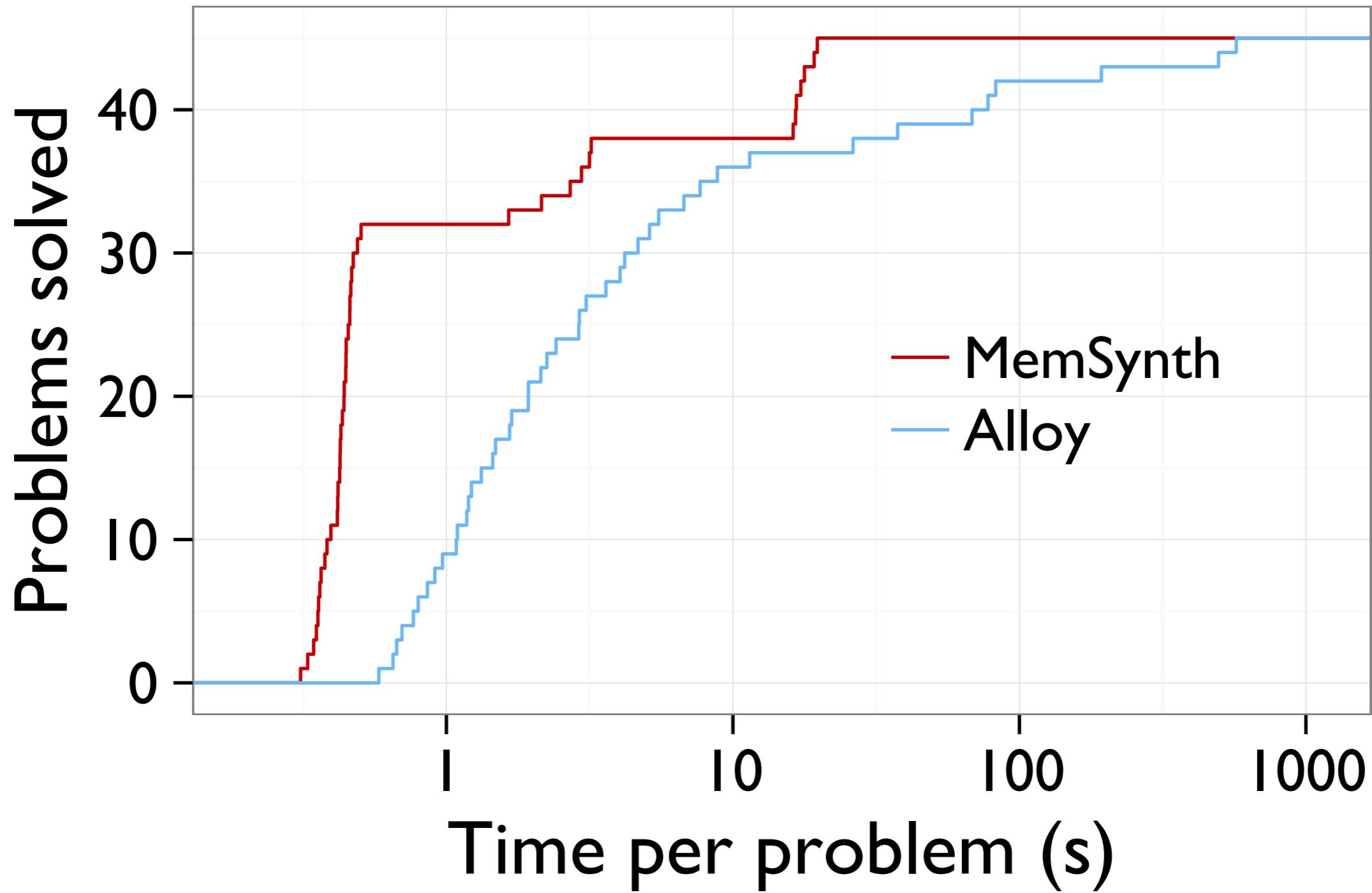
$\forall e_i, e_j : \text{Event} \mid$

$$i < j \wedge e_i.\text{thd} = e_j.\text{thd} \wedge \neg(e_i \in \text{Write} \wedge e_j \in \text{Read}) \\ \Rightarrow \langle e_i, e_j \rangle \in hb_1$$

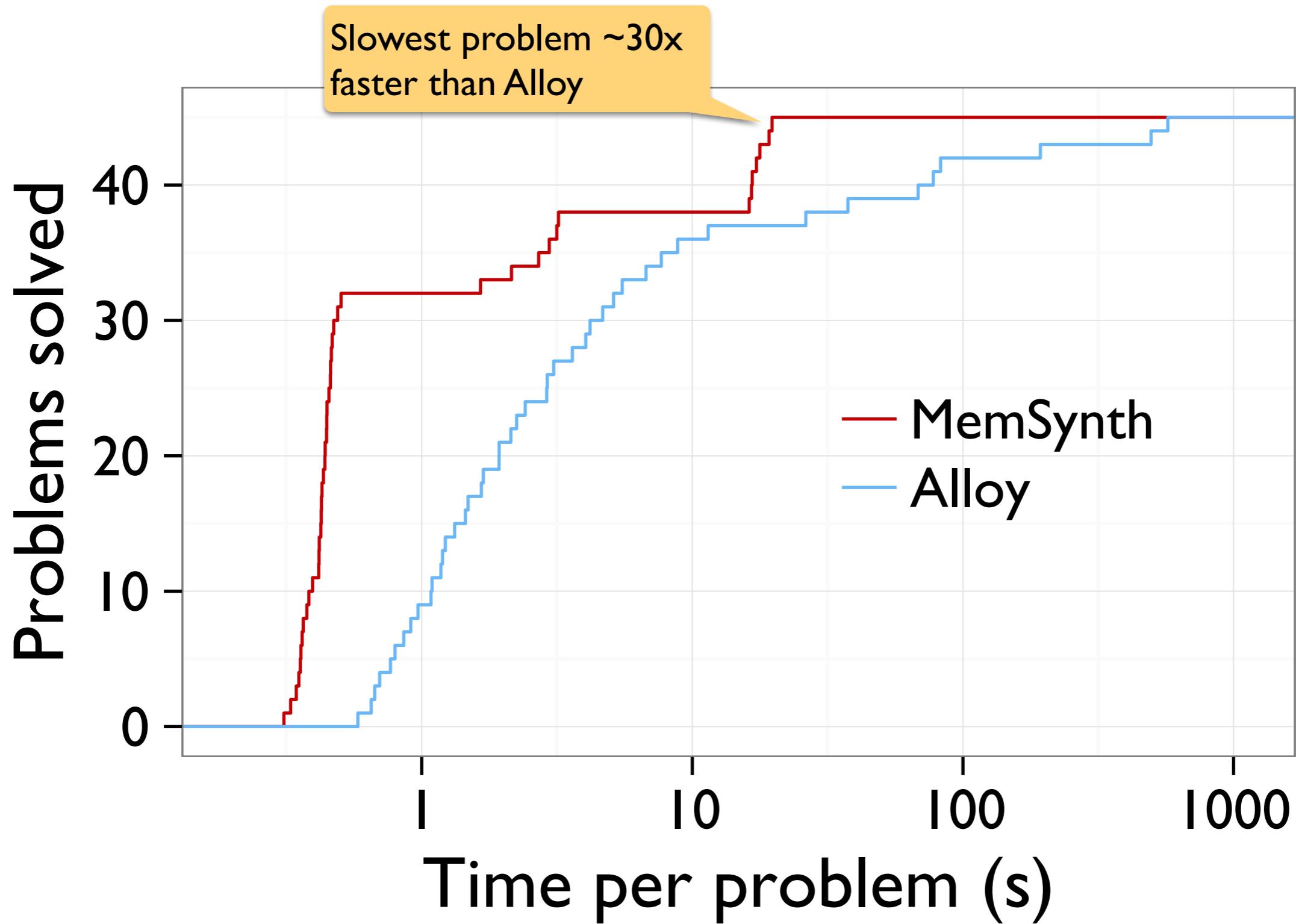
...

no ${}^A hb_1 \cap \text{iden}$

Metasketches for memory models



Metasketches for memory models



MemSynth can synthesize real models

x86

[DOI:10.1145/1789416.1789442](https://doi.org/10.1145/1789416.1789442)

x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors

By Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen

Abstract. Exploiting the multiprocessors that have recently become ubiquitous requires high-performance and reliable concurrent systems code, for concurrent data structures, operating system kernels, synchronization libraries, compilers, and so on. However, concurrent programming, which is always challenging, is made much more so by two problems. First, real multiprocessors typically do not provide the sequentially consistent memory that is assumed by most work on semantics and verification. Instead, they have relaxed memory models, varying in subtle ways between processor families, in which different hardware threads may have only loosely consistent views of a shared memory. Second, the public vendor architectures, supposedly specifying what programmers can rely on, are often in ambiguous informal prose (a particularly poor medium for loose specifications), leading to widespread confusion.

In this paper we focus on x86 processors. We review several recent Intel and AMD specifications, showing that all contain serious ambiguities; some are arguably too weak to program above, and some are simply unsound with respect to actual hardware. We present a new x86-TSO programmer's model that, to the best of our knowledge, suffers from none of these problems. It is mathematically precise (rigorously defined in HOL) but can be presented as an intuitive abstract machine which should be widely accessible to working programmers. We illustrate how this can be used to reason about the correctness of a Linux spinlock implementation and describe a general theory of data-race freedom for TSO. This should put x86 multiprocessor system building on a more solid foundation; it should also provide a basis for future work on verification of such systems.

1 INTRODUCTION

Multiprocessor machines, with many processors acting on a shared memory, have been developed since the 1980s; they are now ubiquitous. Meanwhile, the difficulty of programming concurrent systems has motivated extensive research on programming language design, semantics, and verification, from semaphores and monitors to program logics, software model checking, and so forth. This work has almost always assumed that concurrent threads share a single sequentially consistent memory,¹ with their reads and writes interleaved in some order. In fact, however, real multiprocessors use sophisticated techniques to achieve high performance: store buffers, hierarchies of local cache, speculative execution,

etc. These optimizations are not observable by sequential code, but in multithreaded programs different threads may see subtly different views of memory; such machines exhibit relaxed, or weak, memory models.^{2,3,12,19}

For a simple example, consider the following assembly language program [38] for modern Intel or AMD x86 multiprocessors given two distinct memory locations x and y (initially holding 0), if two processors respectively write 1 to x and y and then read from y and x into register EX on processor 0 and EBX on processor 1, it is possible for both to read 0 in the same execution. It is easy to check that this result cannot arise from any interleaving of the reads and writes of the two processors; modern x86 multiprocessors do not have a sequentially consistent memory.

5B

Proc 0	Proc 1
MOV [x]-1	MOV [y]-1
MOV EX-[y]	MOV EBX-[x]
Allowed Final State: Proc 0:EX=0 & Proc 1:EBX=0	

More architecturally, one can view this particular example as a visible consequence of store buffering: if each processor effectively has a FIFO buffer of pending memory writes (to avoid the need to block while a write completes), then the reads from y and x could occur before the writer have propagated from the buffers to main memory.

Other families of multiprocessors, dating back at least to the IBM 370, and including ARM, Itanium, POWER, and SPARC, also exhibit relaxed-memory behavior. Moreover, there are major and subtle differences between different processor families (arising from their different internal design choices): in the details of exactly what non-sequentially-consistent executions they permit, and of what memory barrier and synchronization instructions they provide to let the programmer regain control.

For any of these processors, relaxed memory behavior exacerbates the difficulties of writing concurrent software, as systems programmers cannot reason, at the level of abstraction of memory reads and writes, in terms of an intuitive concept of global time.

This paper is based on work that first appeared in the Proceedings of the 36th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2009, and in the Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs), 2009.

1 JFLY 2012 | VOL 33 | NO 7 | DOI:10.1145/1789416.1789442

PowerPC

Fences in Weak Memory Models

Jade Alglave¹, Luc Maranget¹, Susmit Sarkar², and Peter Sewell²

¹ INRIA ² University of Cambridge

Abstract. We present a class of relaxed memory models, defined in Coq [8], parameterised by the chosen permitted local reorderings of reads and writes, and the visibility of inter- and intra-processor communications through memory (e.g. store atomicity relaxations). We prove results on the required behaviour and placement of memory fences to restore a given model (such as Sequential Consistency) from a weaker one. Based on this class of models we develop a tool, *dy*, that systematically and automatically generates and runs litmus tests to determine properties of processor implementations. We detail the results of our experiments on Power and the model we base on them. This work identified a rare implementation error in Power 5 memory barriers (for which IBM is providing a workaround); our results also suggest that Power 6 does not suffer from this problem.

1 Introduction

Most multiprocessors exhibit subtle relaxed-memory behaviour, with writes from one thread not immediately visible to all others; they do not provide sequentially consistent (*SC*) memory [17]. For some, such as x86 [22,20] and Power [21], the vendor documentation is in inevitably ambiguous informal prose, leading to confusion. Thus we have no foundation for software verification of concurrent systems code, and no target specification for hardware verification of microarchitecture. To remedy this state of affairs, we take a firmly empirical approach, developing, in tandem, testing tools and models of multiprocessor behaviour—the test results guiding model development and the modelling suggesting interesting tests. In this paper we make five new contributions:

1. We introduce a class of memory models, defined in Coq [8], which we show how to instantiate to produce *SC*, *TSO* [24], and a Power model (3 below).
2. We describe our *dy* testing tool. Much discussion of memory models has been in terms of *litmus tests* (e.g. irw [9]), ad-hoc multiprocessor programs for which particular final states may be allowed on a given architecture. Given a violation of *SC*, *dy* systematically and automatically generates litmus tests (including classical ones such as irw) and runs them on the hardware.
3. We model important aspects of Power processors' behaviour, i.e. ordering relaxations, the lack of store atomicity [3,7], and *A*-cumulative barriers [21].
4. We use *dy* to generate about 800 tests, running them up to 1e12 times on 3 Power machines. Our experimental results confirm that our model captures many important aspects of the processor's behaviour, despite being

10 tests (Intel manual)
2 seconds

768 tests (existing work)
16 seconds

MemSynth can synthesize real models

x86

The screenshot shows the first page of a technical paper. At the top right is a DOI link: DOI:10.1145/1789436.1789442. The title 'x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors' is centered. Below the title, it says 'By Peter Sewell, Sumit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen'. The abstract begins with: 'Exploiting the multiprocessors that have recently become ubiquitous requires high-performance and reliable concurrent systems code, for concurrent data structures, operating system kernels, synchronization libraries, compilers, and so on. However, concurrent programming, which is always challenging, is made much more so by two problems. First, real multiprocessors typically do not provide the sequentially consistent memory that is assumed by most work on semantics and verification. Instead, they have relaxed memory models, varying in subtle ways between processor families, in which different hardware threads may have only loosely consistent views of a shared memory. Second, the public vendor architectures, supposedly specifying what programmers can rely on, are often in ambiguous informal prose (a particularly poor medium for loose specifications), leading to widespread confusion.' There is a table titled 'SB' showing assembly code for two processors. The table has two columns: 'Proc 0' and 'Proc 1'. Under Proc 0, there are two rows: 'MOV [x]-1' and 'MOV EAX-[y]'. Under Proc 1, there are two rows: 'MOV [y]-1' and 'MOV EAX-[z]'. Below the table is a note: 'Allocated Final State: Proc 0: DEAD00A Proc 1: FED00'. The bottom of the page contains a note: 'This paper is based on work that first appeared in the Proceedings of the 36th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2009, and in the Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs), 2009,' followed by a copyright notice: '© 2010 ACM 0001-078X/10/0100 \$15.00'.

10 tests (Intel manual)
2 seconds

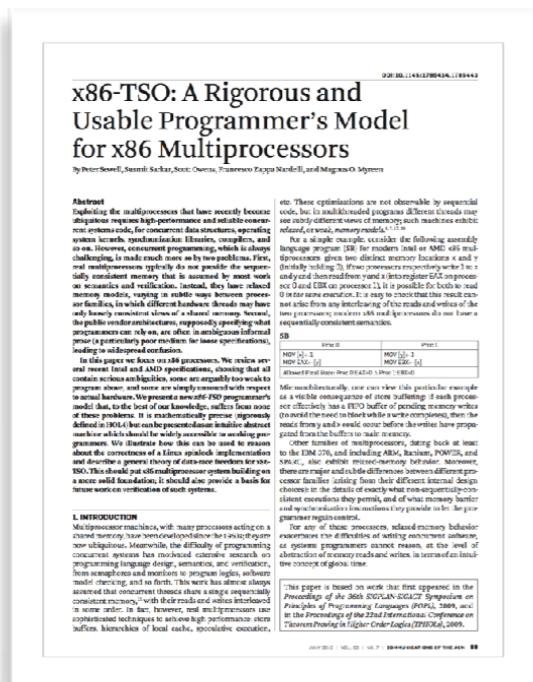
PowerPC

```
(define ppo
  (& po (& (- (- (+ po dep)
    (& (join loc (~ loc))
      (-> Write Write)))
    (& (- po dep)
      (-> Read Write)))
    (- (- po (-> Write Event))
      (- (-> Read Read)
        (& po dep))))))
(define grf
  (-> none none))
(define fences
  (+
    (^ (let ([poFpo (join (:> po Fence) po)])
        (+
          (:>
            (+ (+ rf poFpo) (join poFpo poFpo))
            (+ (join poFpo Write) (join Write poFpo)))
          (:>
            (join (+ rf poFpo) (+ rf poFpo))
            (join (+ Read Write) (+ rf poFpo)))))
        (^ (let ([poLWFpo (join (:> po Lwsyncs) po)])
            [RE+WW (+ (-> Read Event) (-> Write Write))])
          (:>
            (:>
              (+ (:> (& RE+WW poLWFpo) Write)
                (join (& RE+WW poLWFpo) rf))
              (+ (join (& RE+WW poLWFpo) Write)
                (join Write (& RE+WW poLWFpo))))
            (join
              (+ Read Write)
              (+ (<: Write rf)
                (<: Write (& RE+WW poLWFpo)))))))))))
```

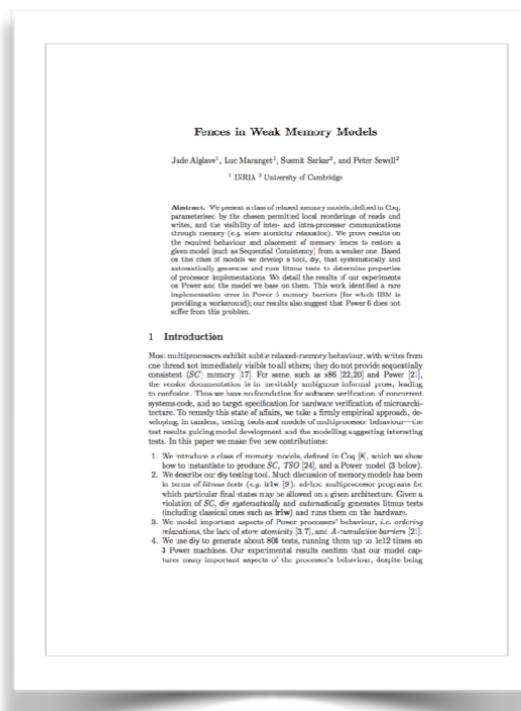
768 tests (existing work)
16 seconds

MemSynth can find ambiguities in real models

x86



PowerPC



10 tests (Intel manual)
2 seconds

3 missing litmus tests
(Intel manual identifies at least 4 different models!)

768 tests (existing work)
16 seconds

10 missing litmus tests
(existing testing identifies at least 11 different models!)

Summary

Today

- Metasketches: scalable program synthesis
- MemSynth: a metasketch-based synthesis tool

Next lecture

- Project demos!
 - 7 groups, 11 minutes per group
 - Please be on time!