Computer-Aided Reasoning for Software

CSE507

Metasketches

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

James Bornholt
bornholt@cs.washington.edu

Today

topics

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

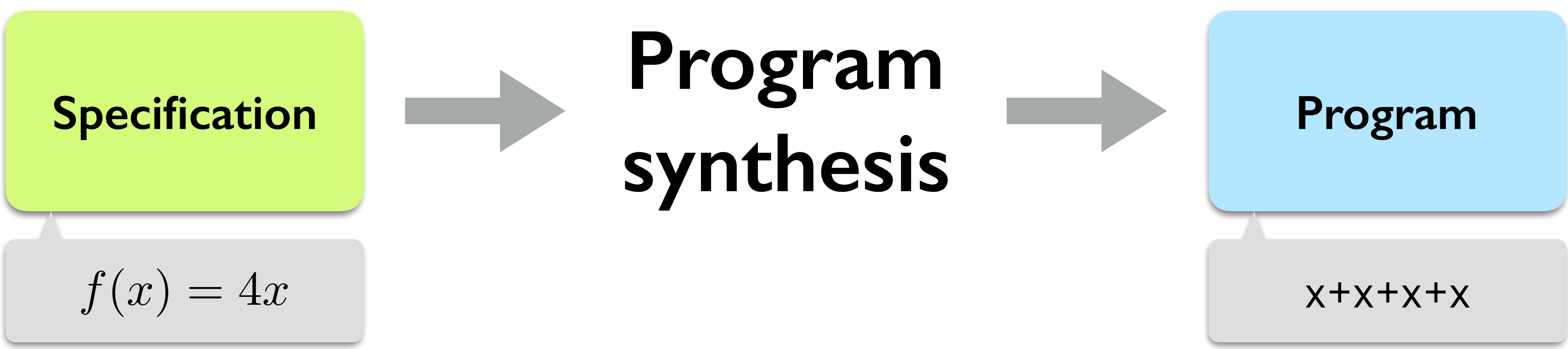
Program synthesis

Specification → **Program synthesis**


Specification
Program
synthesis
Program

Specification → **Program synthesis** → Program

$f(x) = 4x$


Specification
$f(x) = 4x$
Program
synthesis
Program
x+x+x+x

Compilation
[PLDI'14]

Data Structures
[PLDI'15]

End-user Programming
[POPL'11]

Specification → **Program synthesis** → **Program**

$f(x) = 4x$

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]

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

Specification → **Program synthesis** → **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]

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

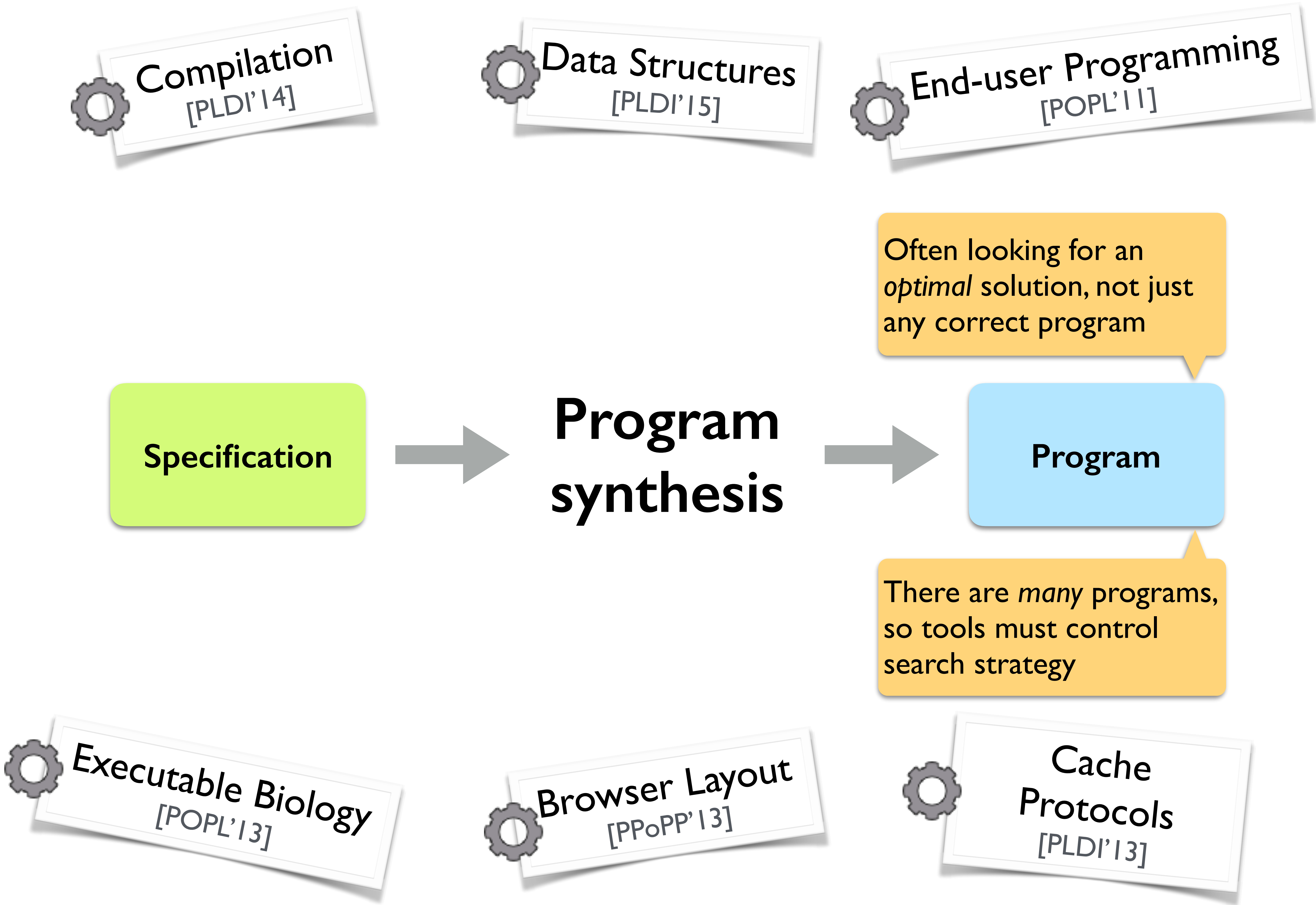
Specification → **Program synthesis** → **Program**

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

Executable Biology
[POPL'13]

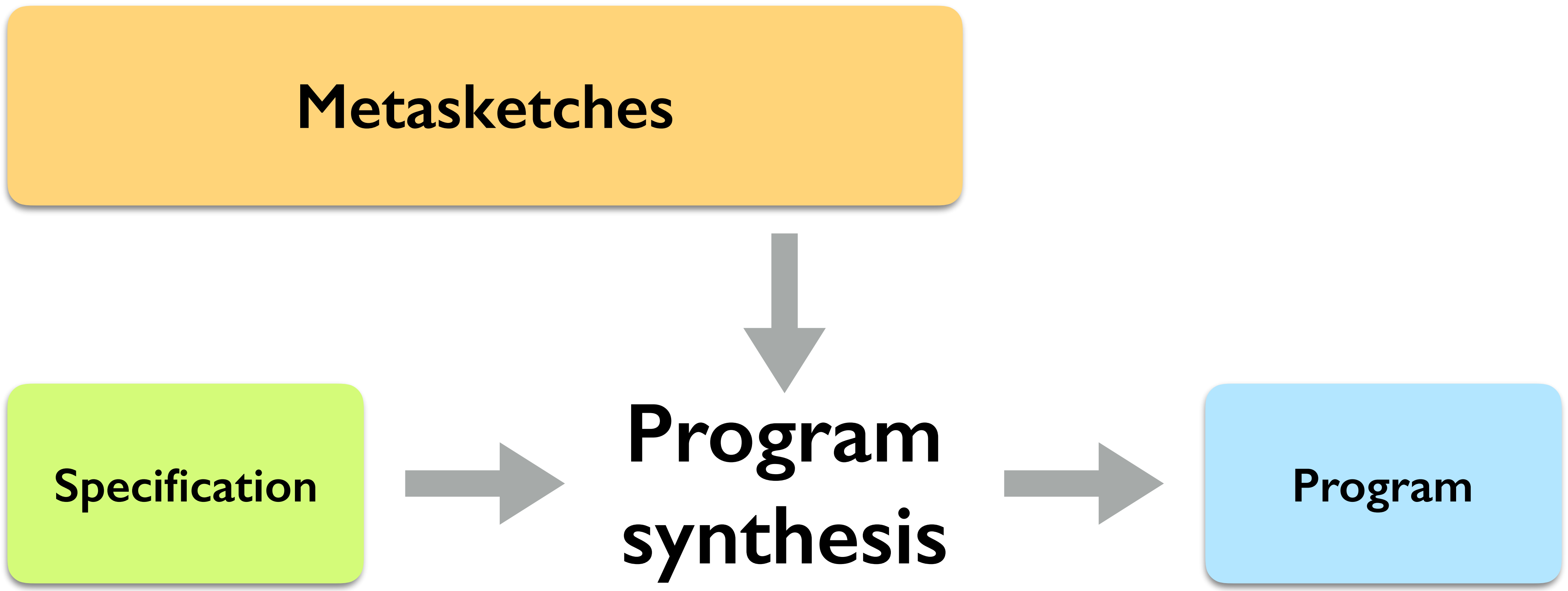
Browser Layout
[PPoPP'13]

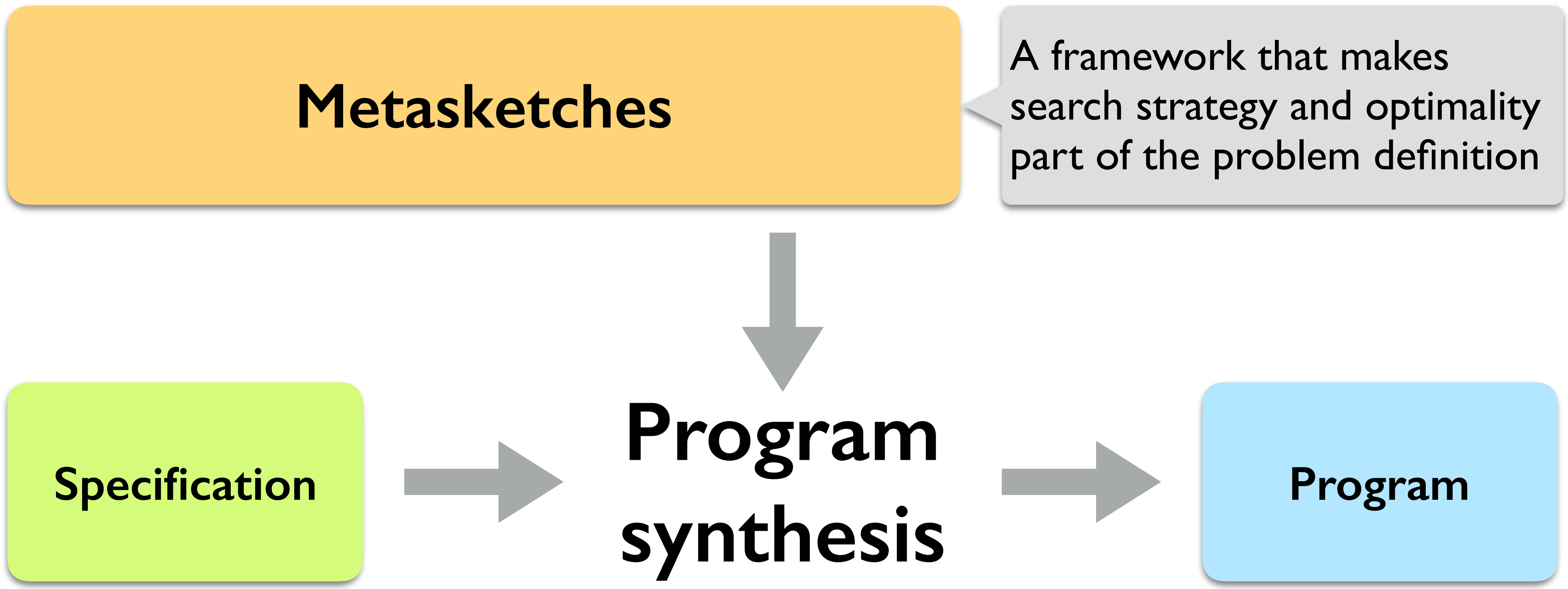
Cache Protocols
[PLDI'13]



Compilation [PLDI'14]
Data Structures [PLDI'15]
End-user Programming [POPL'11]
Often looking for an *optimal* solution, not just any correct program
Specification
Program synthesis
Program
There are *many* programs, so tools must control search strategy
Executable Biology [POPL'13]
Browser Layout [PPoPP'13]
Cache Protocols [PLDI'13]

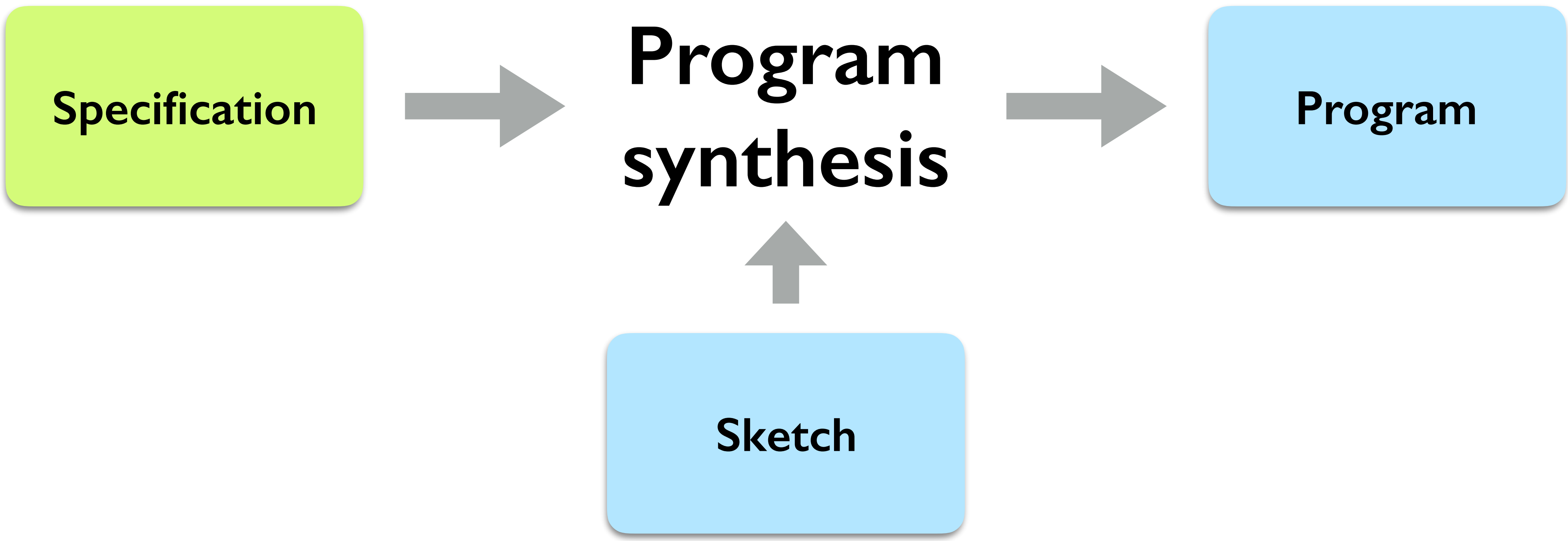

Specification
Program synthesis
Program


Metasketches
Specification
Program
synthesis
Program

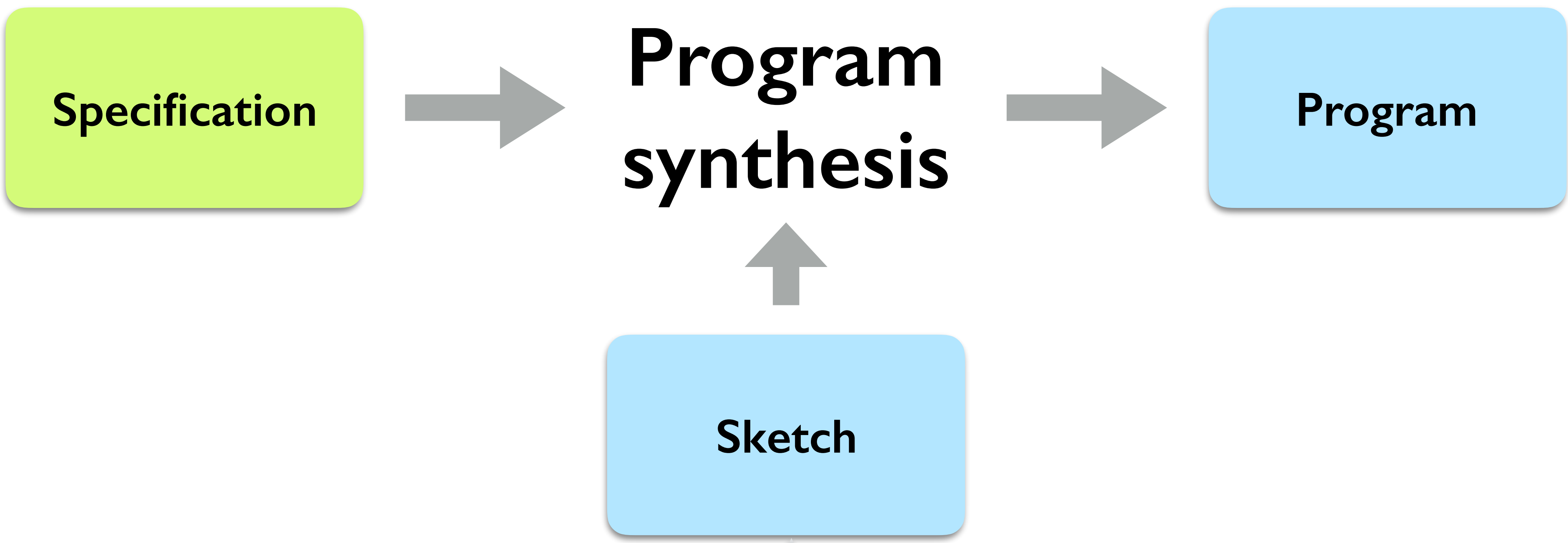

Metasketches
A framework that makes search strategy and optimality part of the problem definition
Specification
Program synthesis
Program

Syntax-guided synthesis

Syntax-guided synthesis



Syntax-guided synthesis



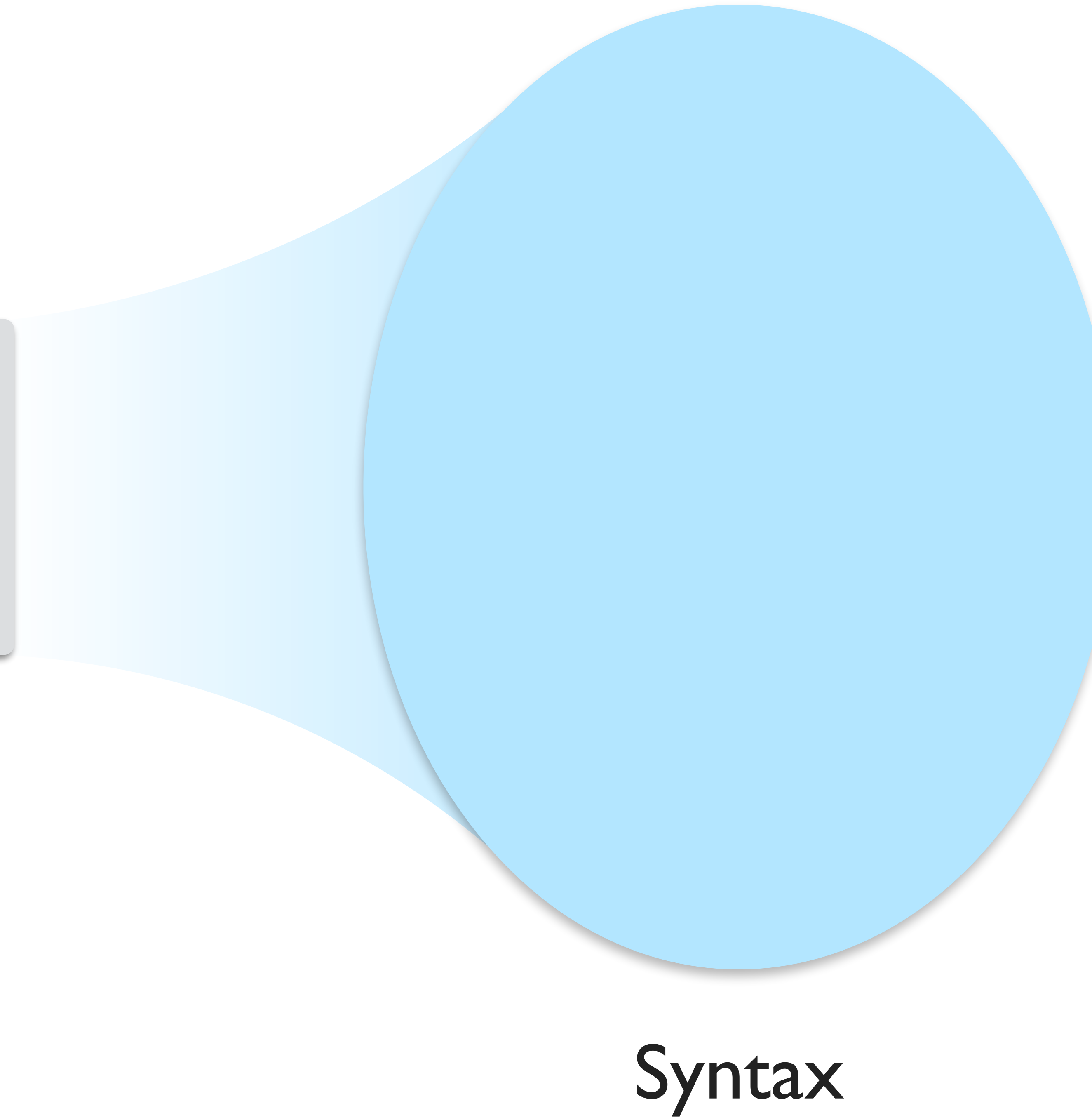

```
def f(x):
  return Expr

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
```

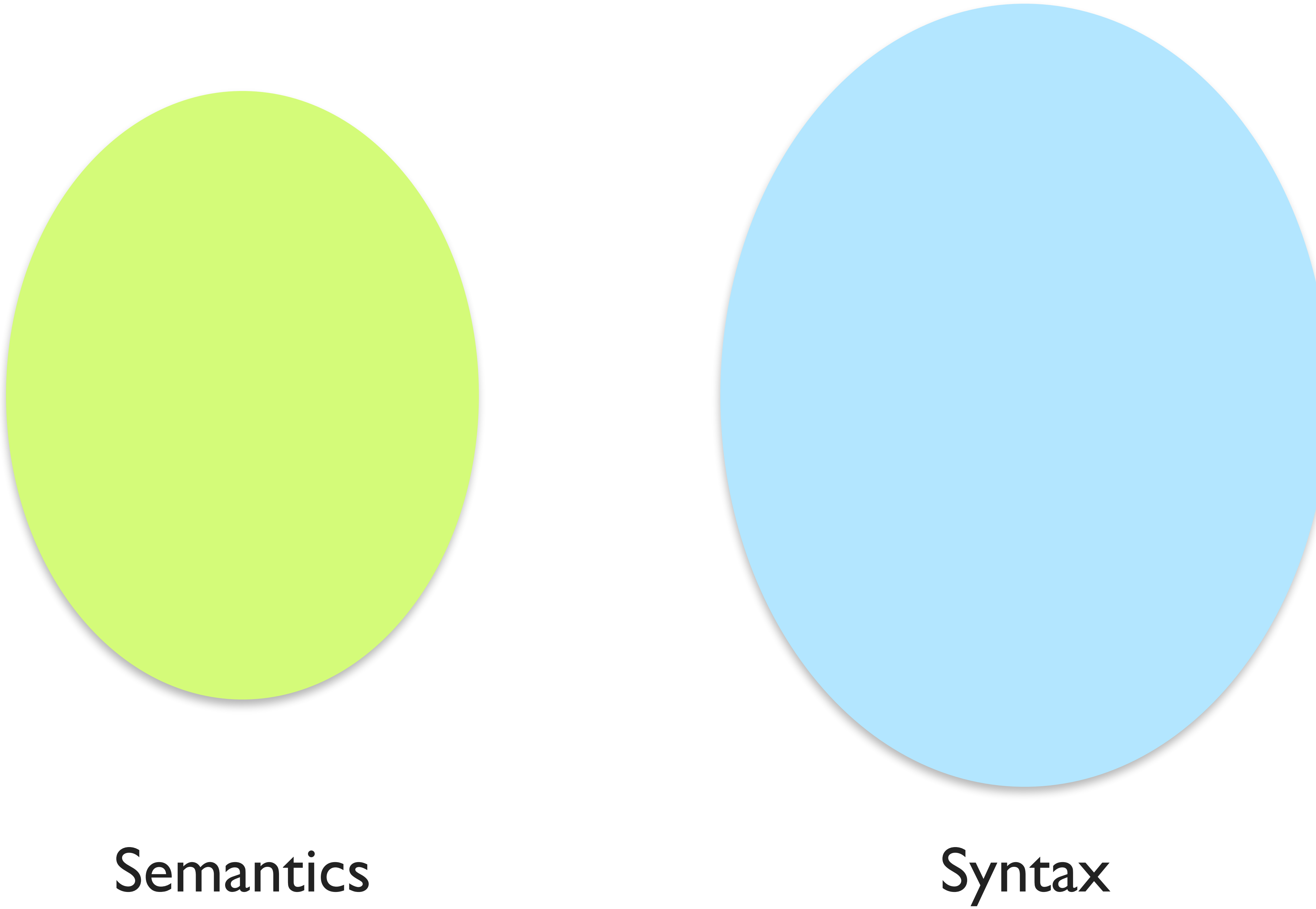
Syntax-guided synthesis: guess, check, learn

```
def f(x):
  return Expr

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



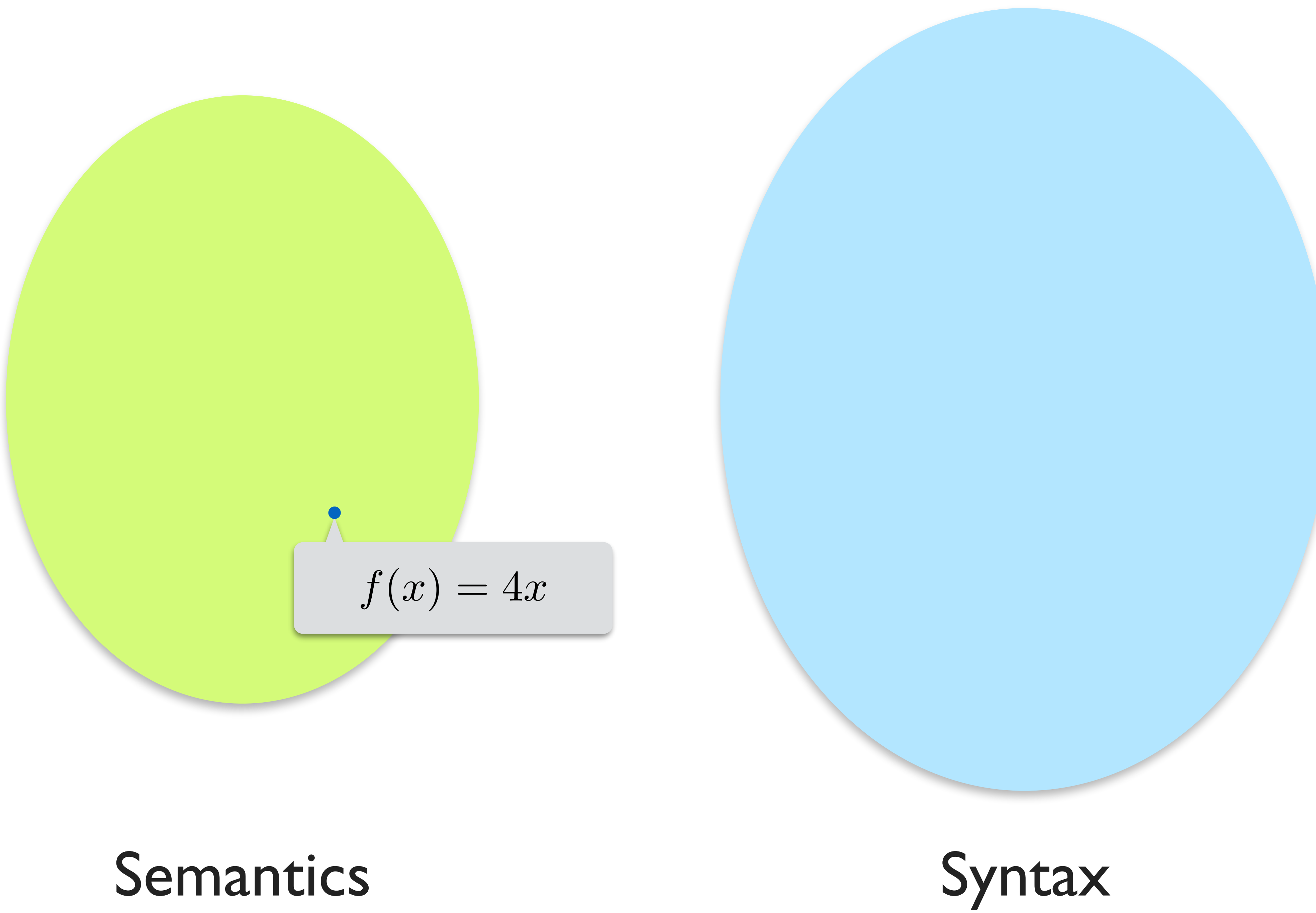
Syntax-guided synthesis: guess, check, learn

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



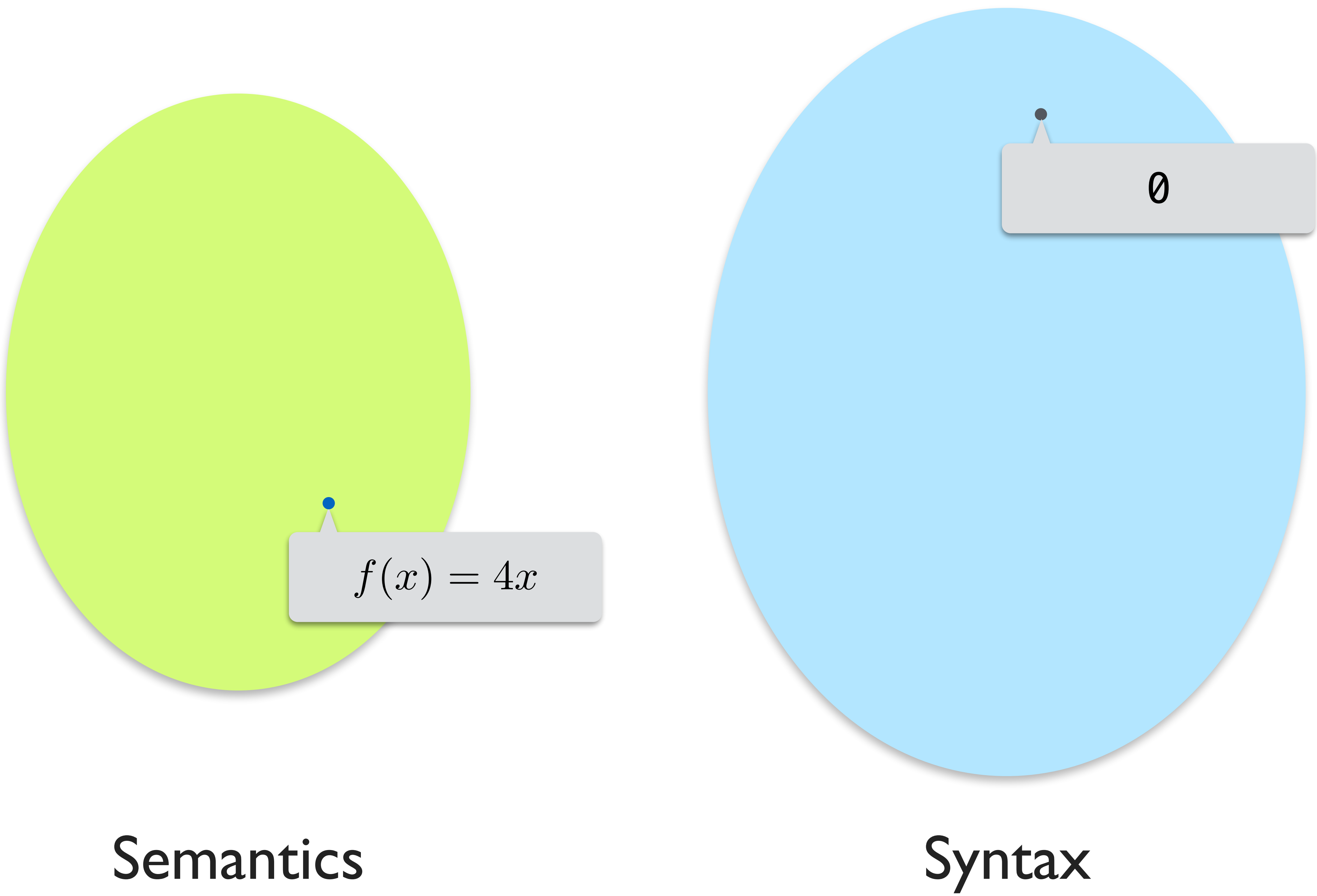
Syntax-guided synthesis: guess, check, learn

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



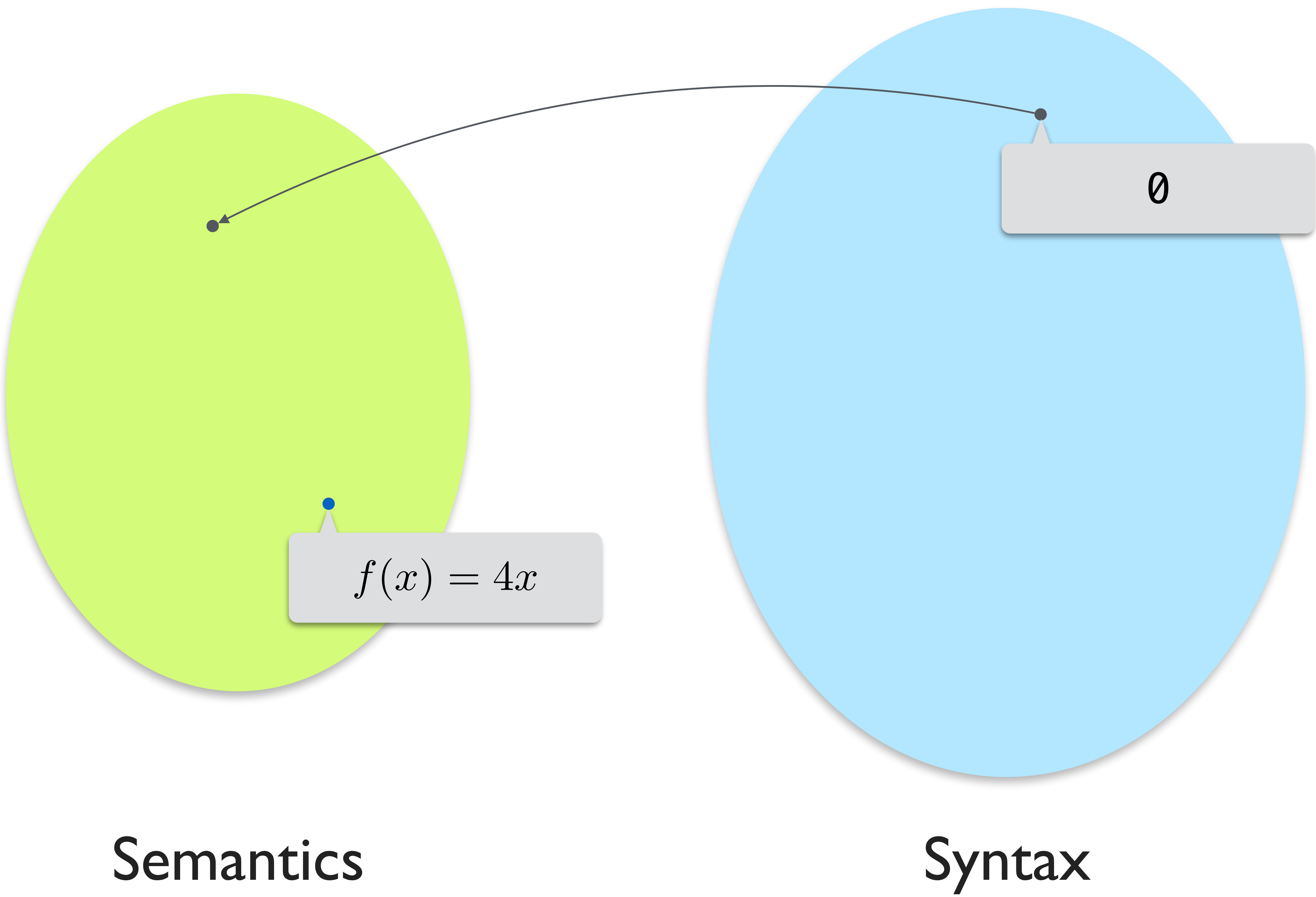
Syntax-guided synthesis: guess, check, learn

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



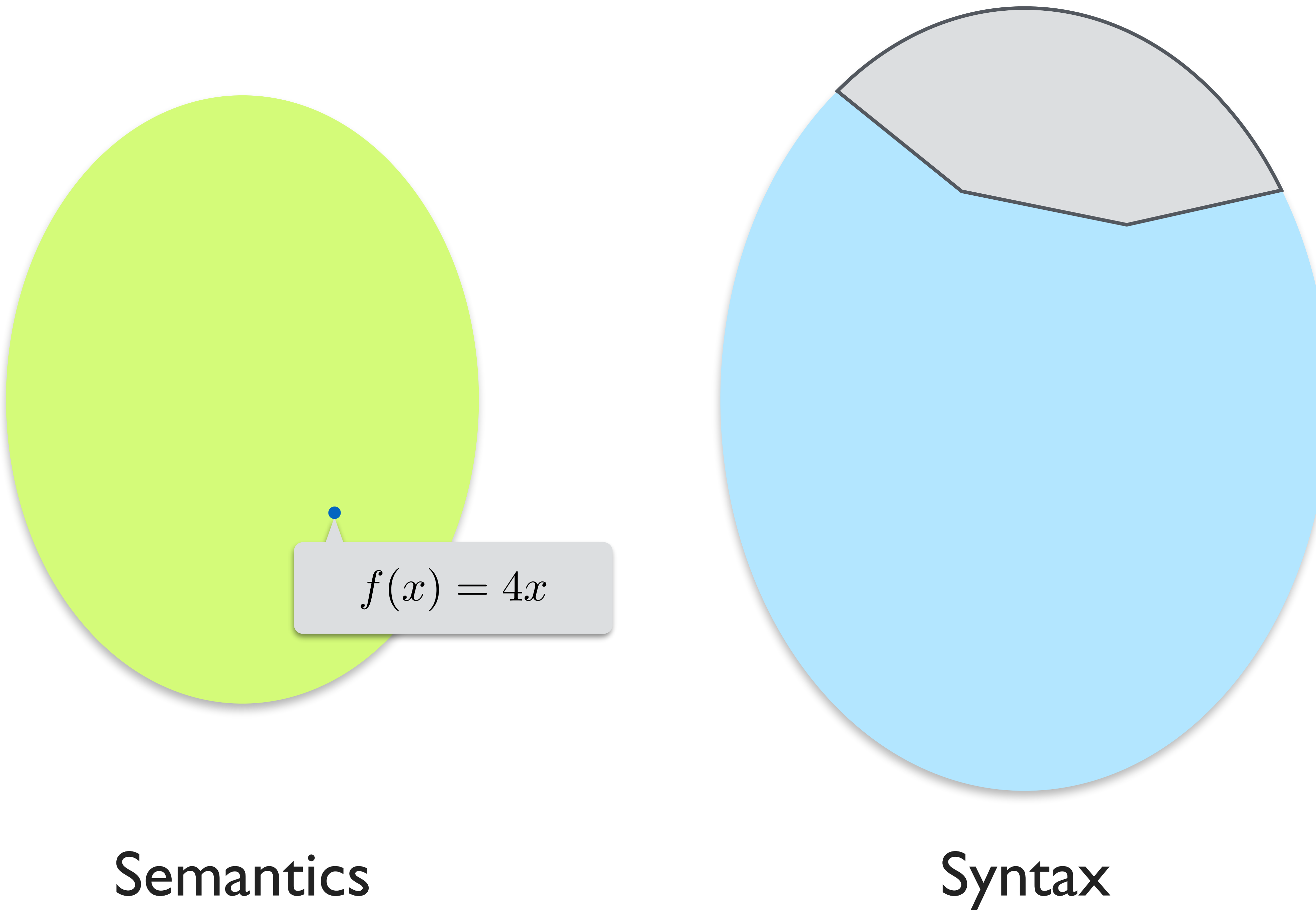
Syntax-guided synthesis: guess, check, learn

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



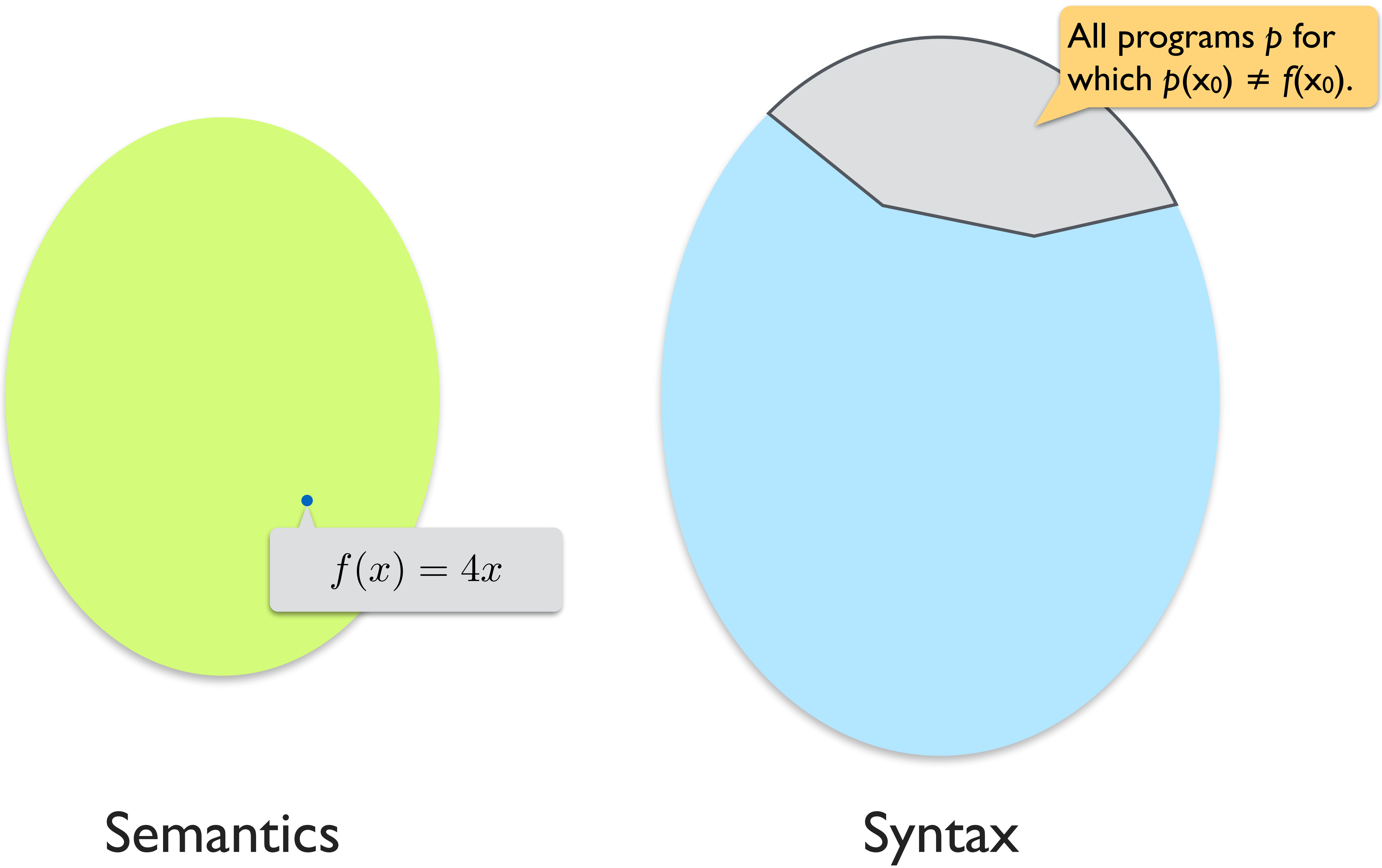
Syntax-guided synthesis: guess, check, learn

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



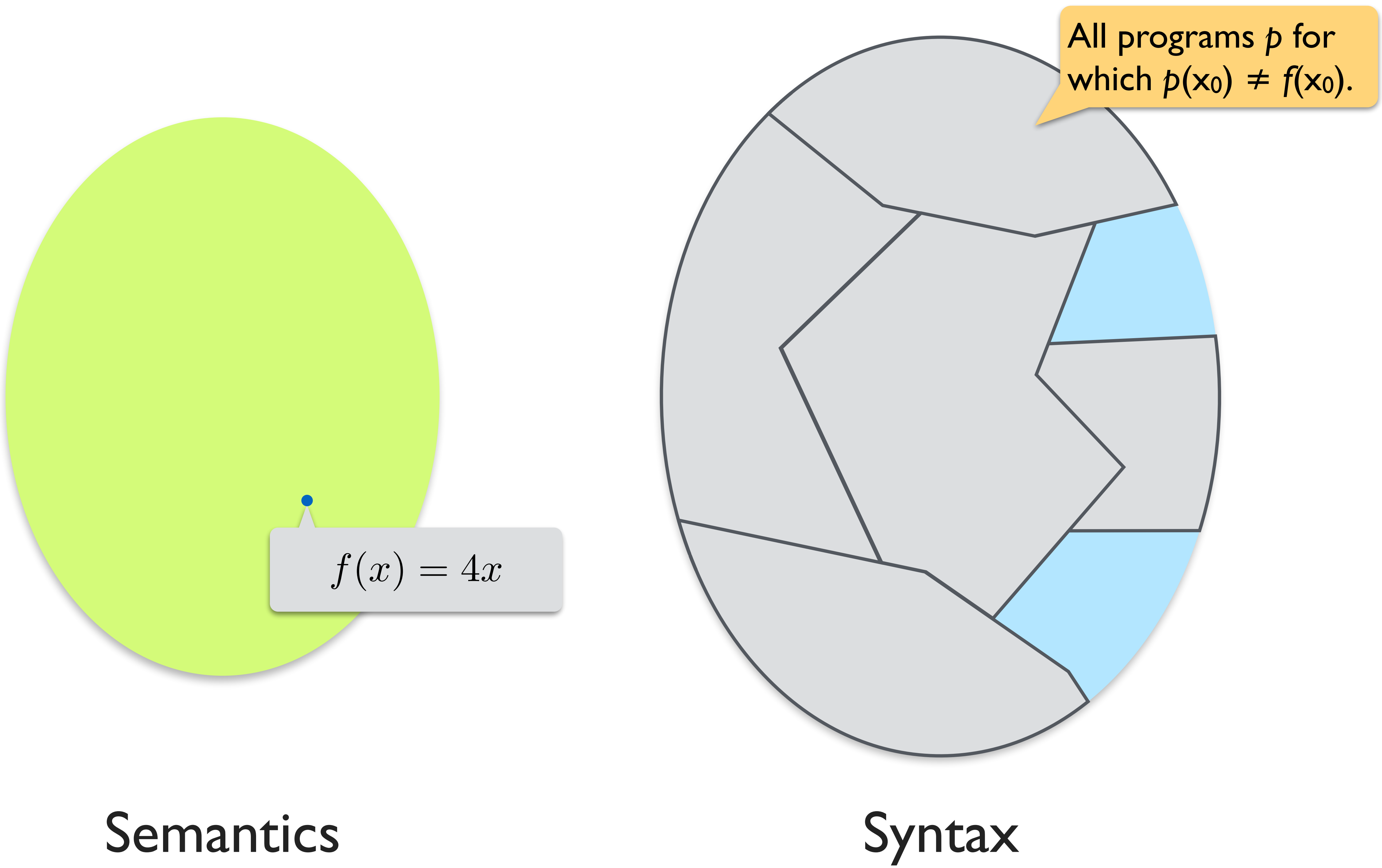
Syntax-guided synthesis: guess, check, learn

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



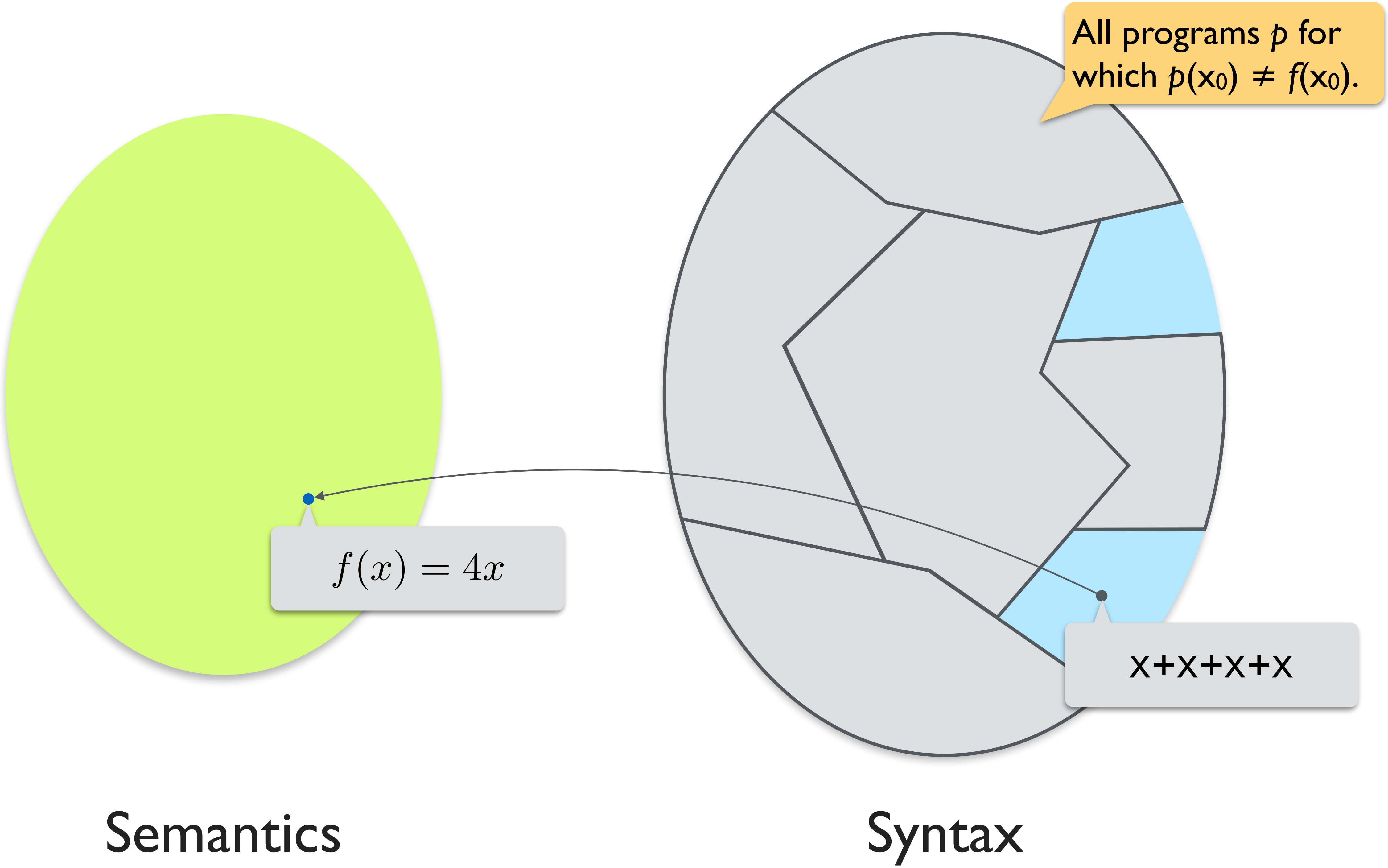
Syntax-guided synthesis: guess, check, learn

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



Syntax-guided synthesis: guess, check, learn

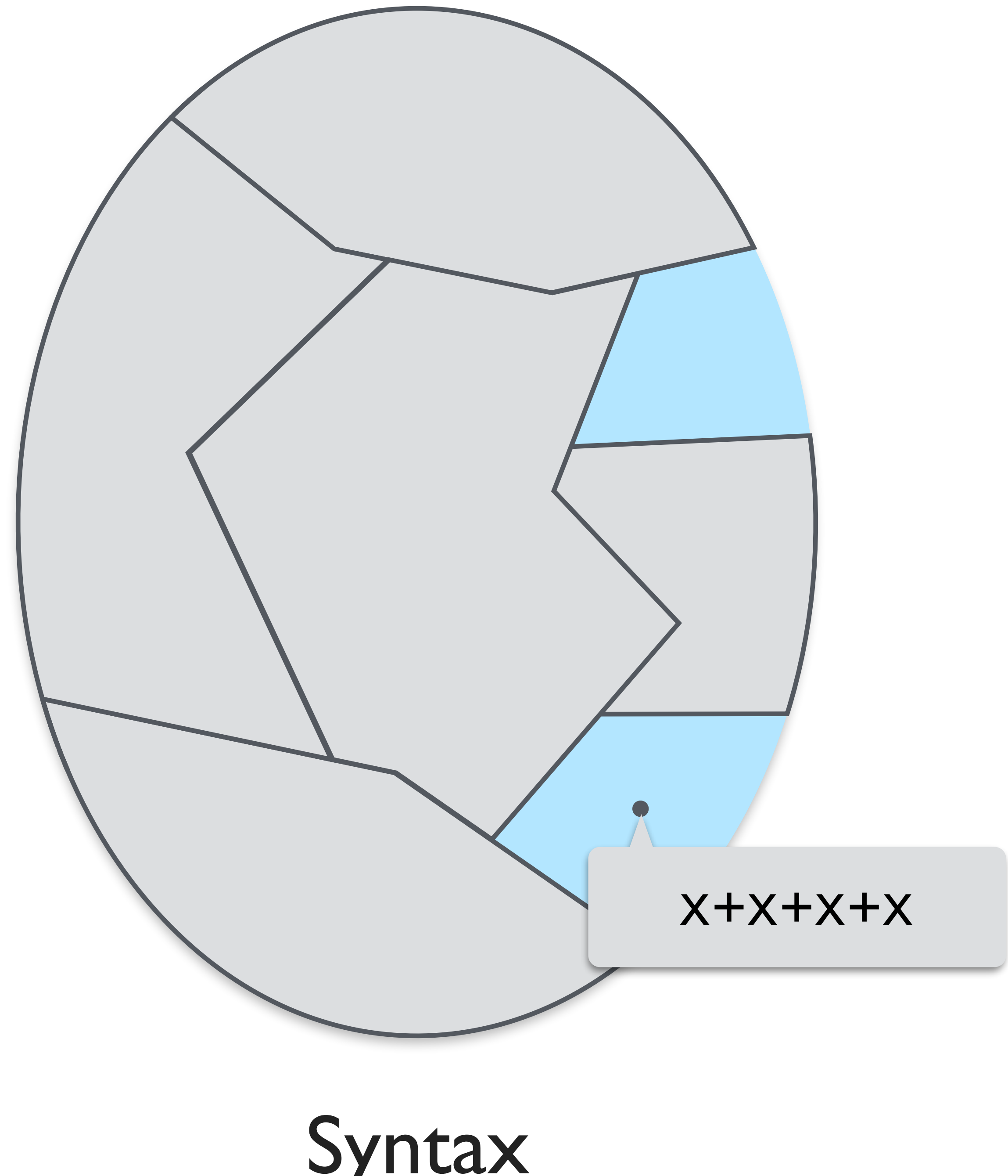
Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]



Syntax-guided synthesis: guess, check, learn

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

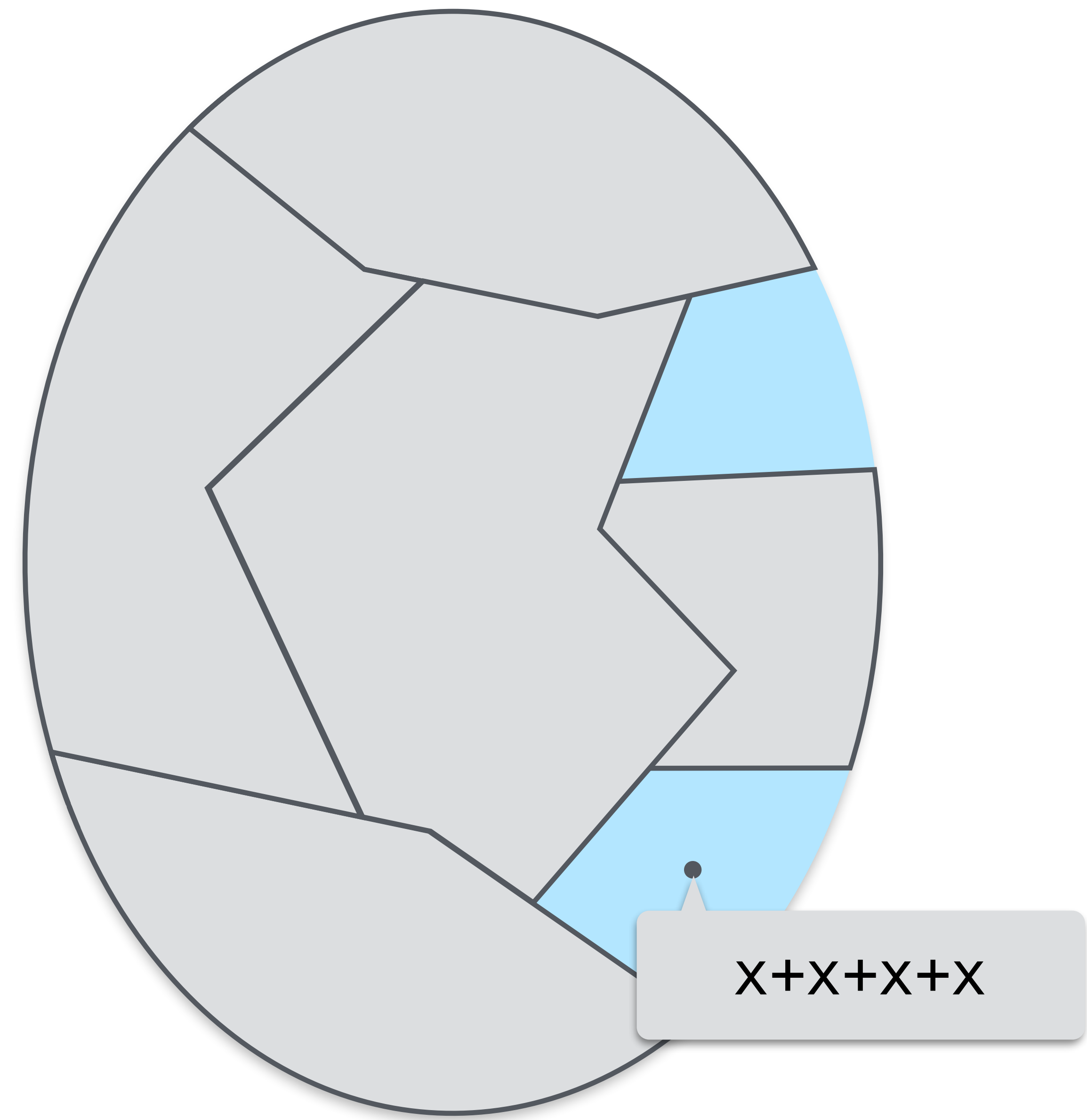
1. **Search order** is critical



Syntax-guided synthesis: guess, check, learn

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

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

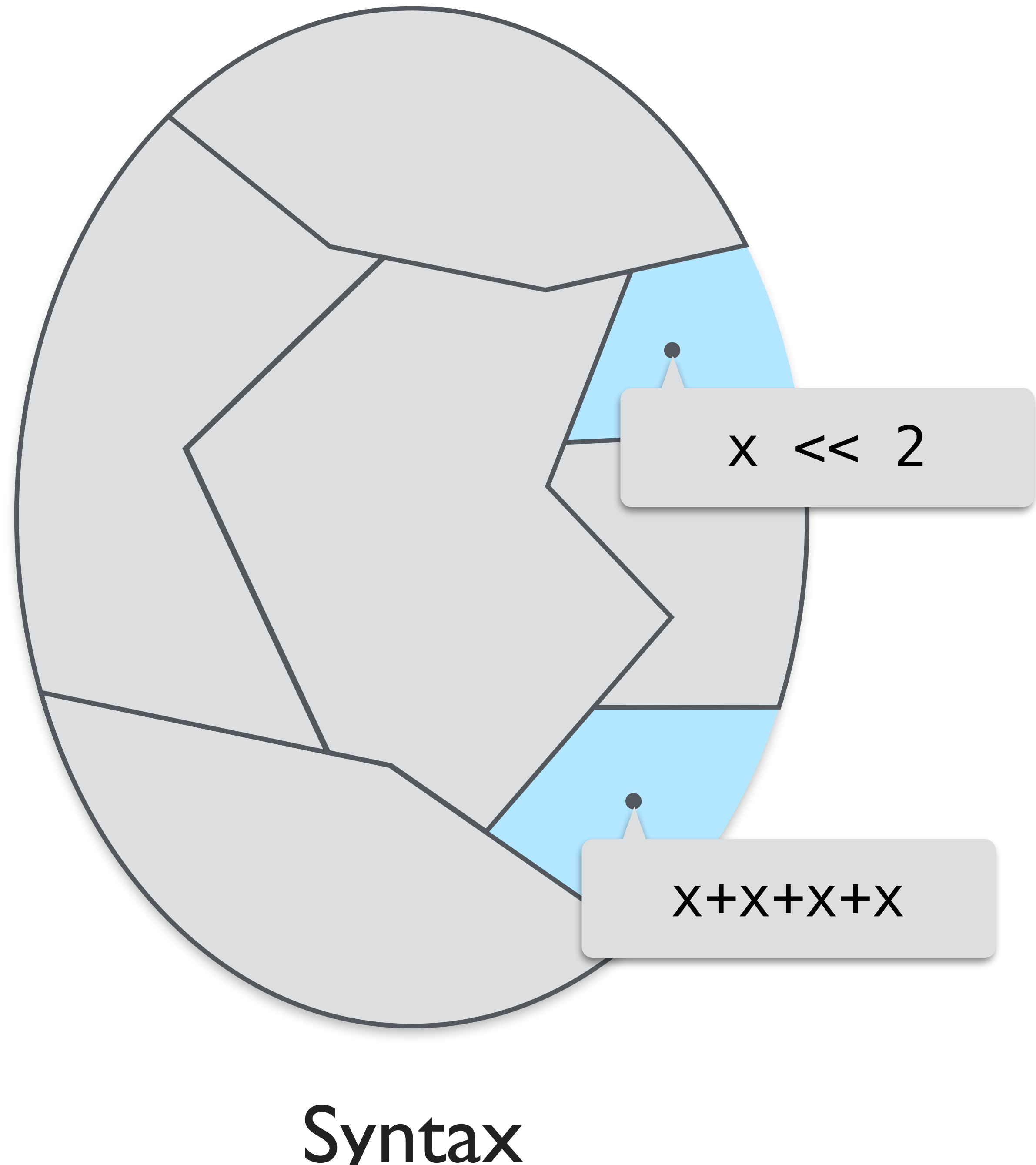


Syntax

Syntax-guided synthesis: guess, check, learn

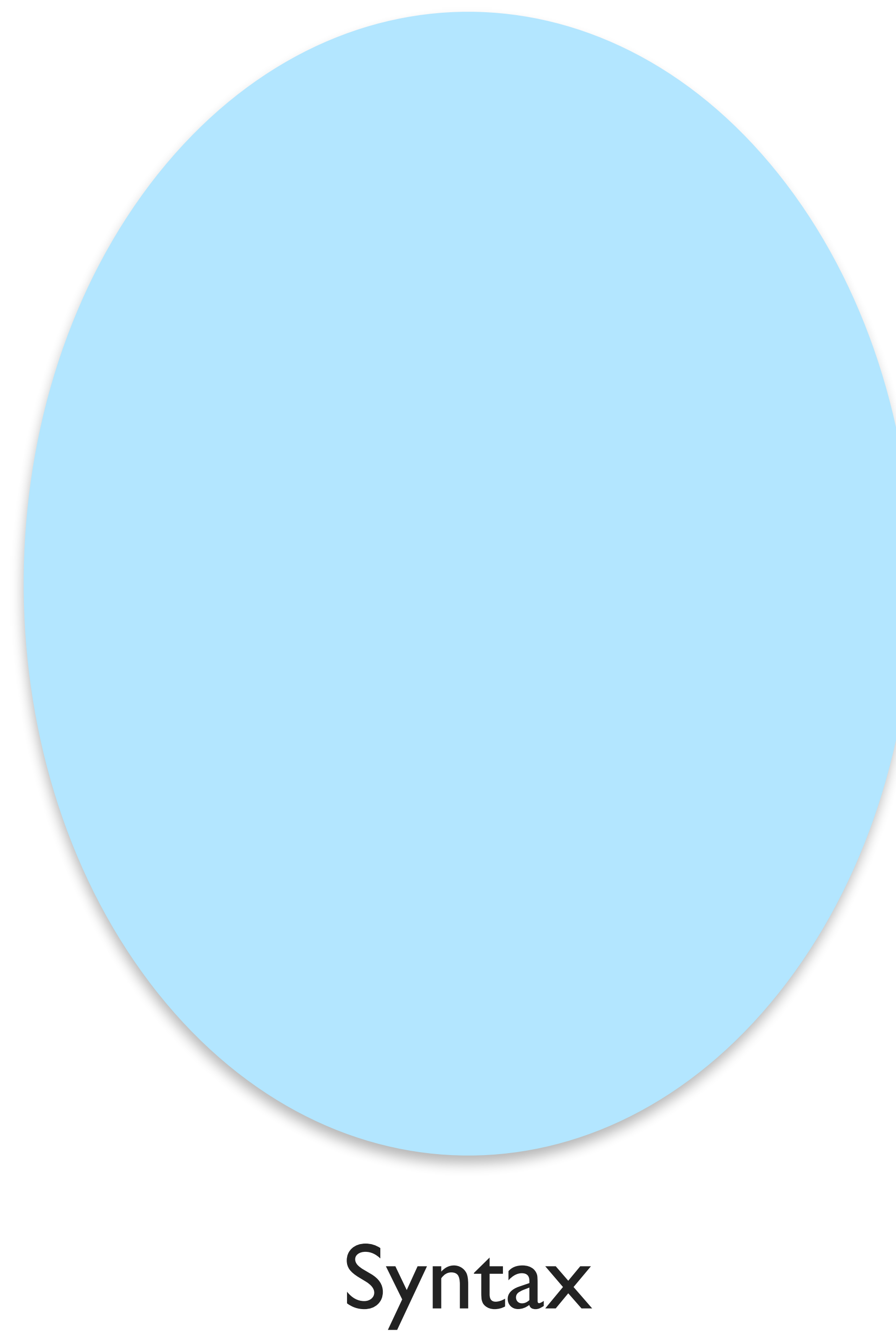
Counterexample-guided inductive synthesis [Solar-Lezama et al, 2006]

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



Metasketches express structure and strategy

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

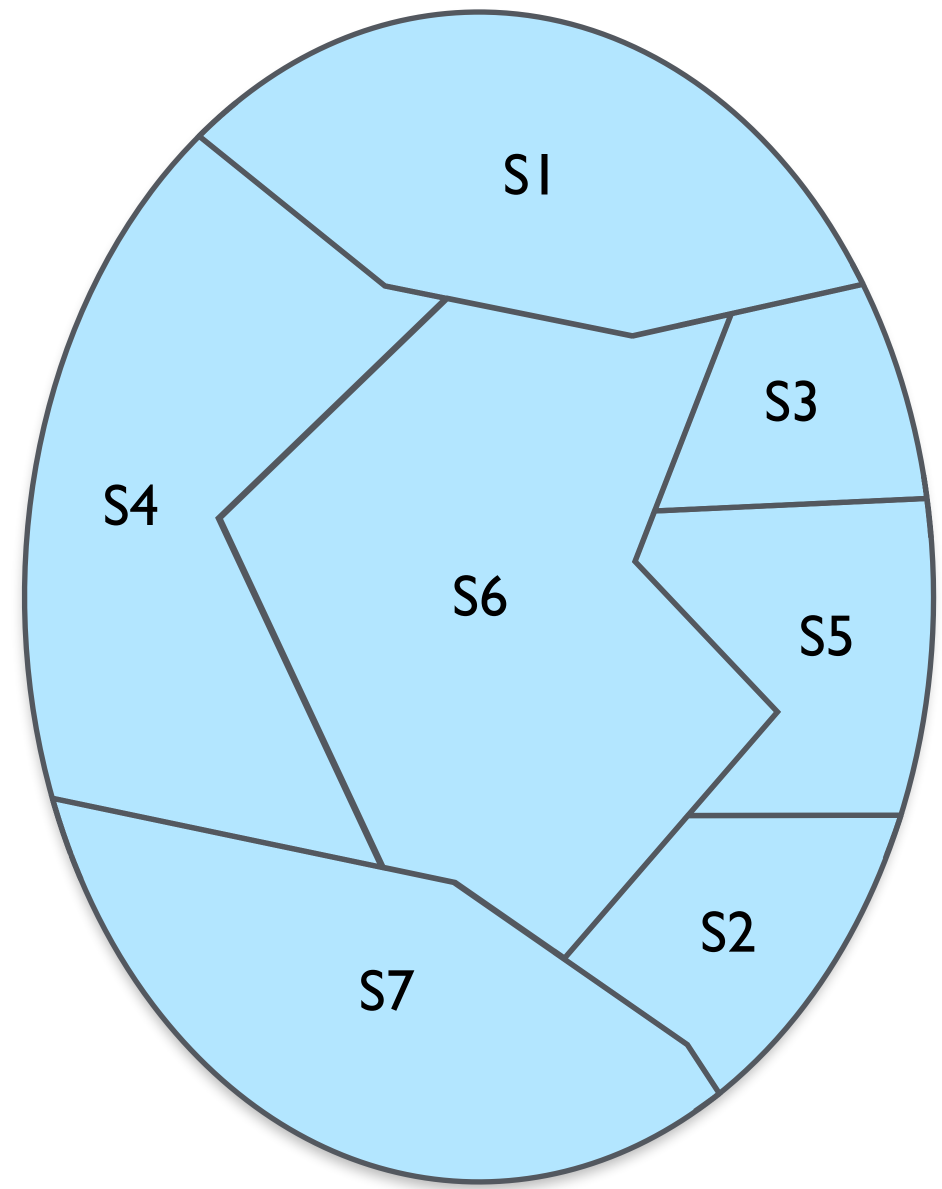


Metasketches express structure and strategy

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

A metasketch contains:

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



Syntax

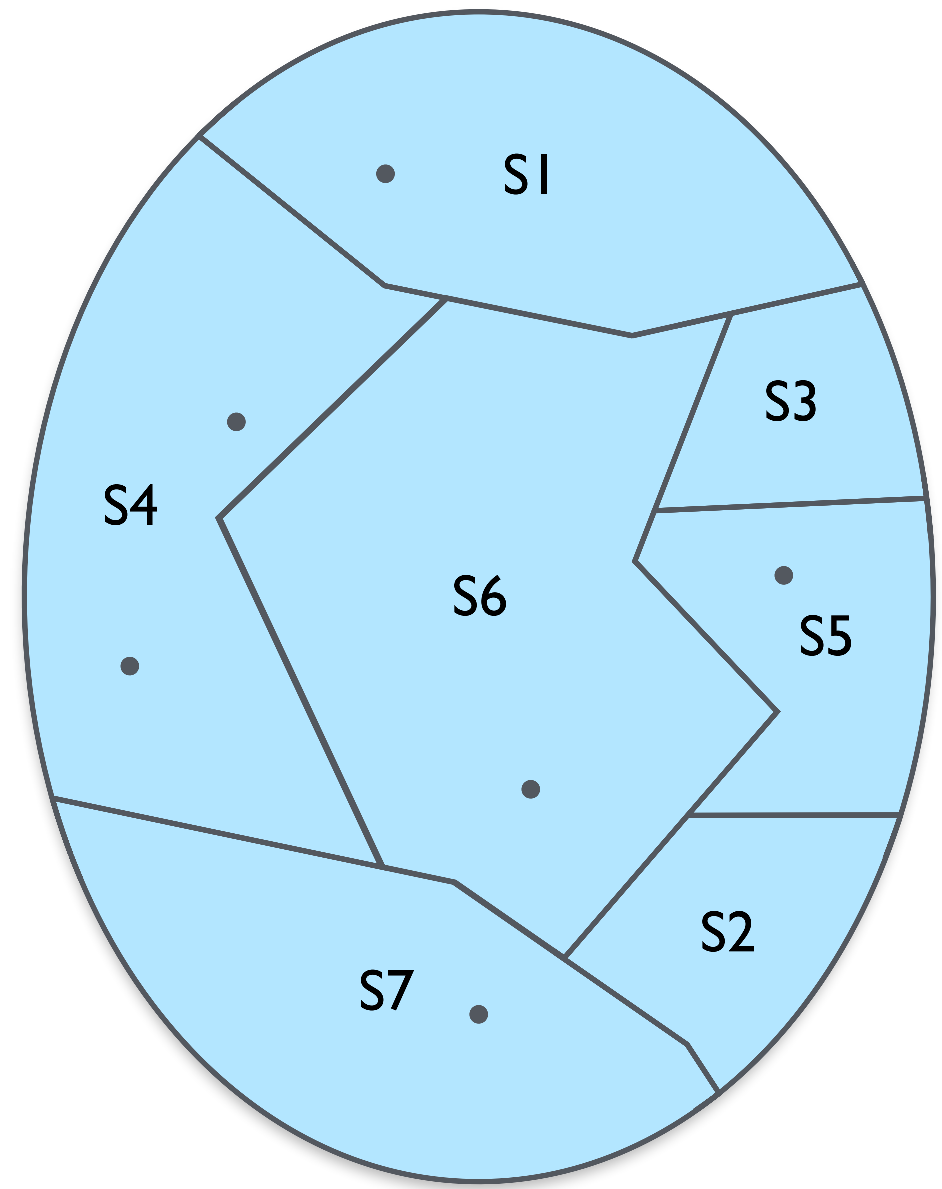
Metasketches express structure and strategy

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

A metasketch contains:

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

2. cost function (κ)



Syntax

Metasketches express structure and strategy

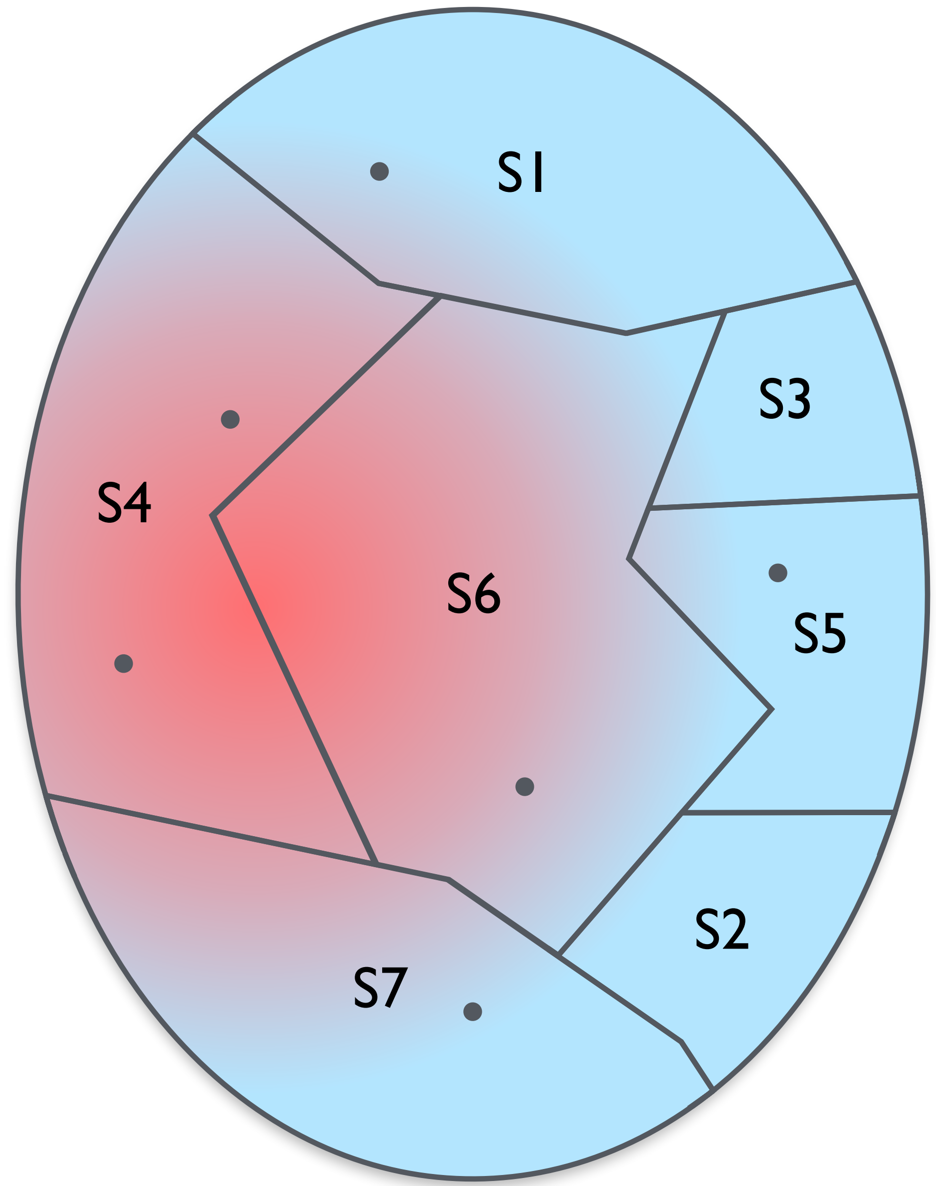
1. **Search order** is critical
2. Desire **optimal** solutions

A metasketch contains:

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

2. cost function (κ)

3. gradient function (g)



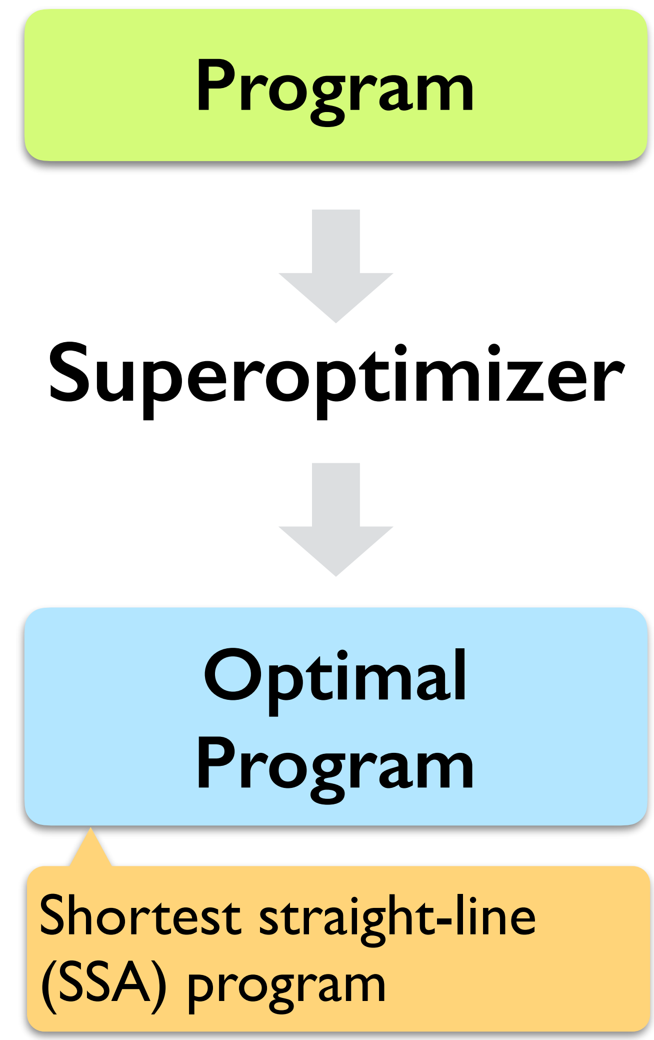
Syntax

Metasketches express structure and strategy

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

A metasketch contains:

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



Metasketches express structure and strategy

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

2. cost function (κ)

3. gradient function (g)

Metasketches express structure and strategy

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

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 ($\mathcal{S}$, $\preccurlyeq$)

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

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

2. cost function (κ)

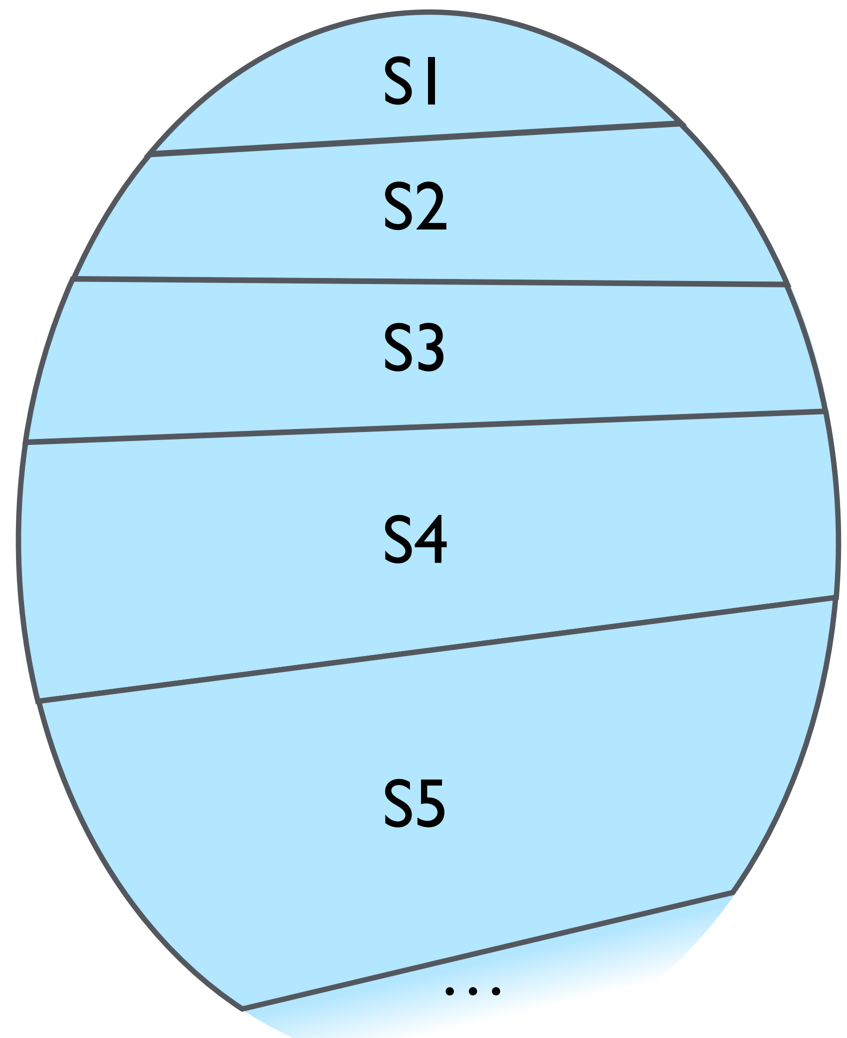
3. gradient function (g)

Metasketches express structure and strategy

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

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)

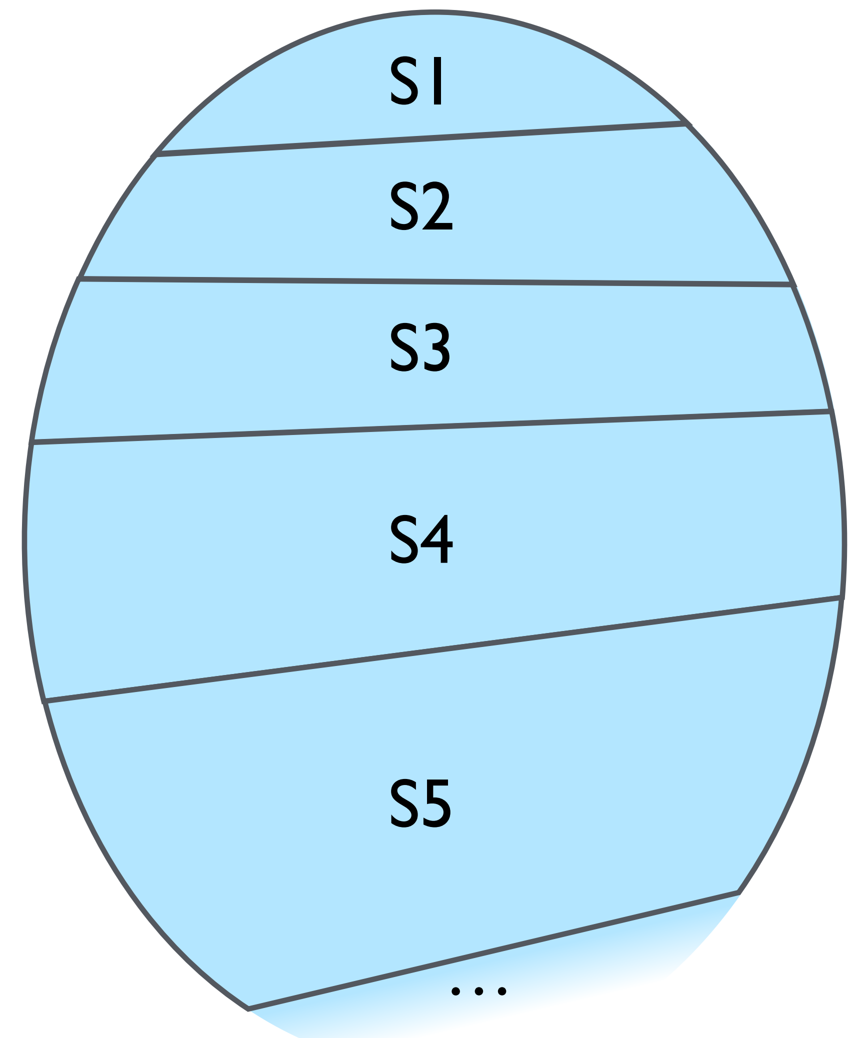
Metasketches express structure and strategy

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

- a countable set $\mathcal{S}$ of sketches
- a total order $\preccurlyeq$ 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)

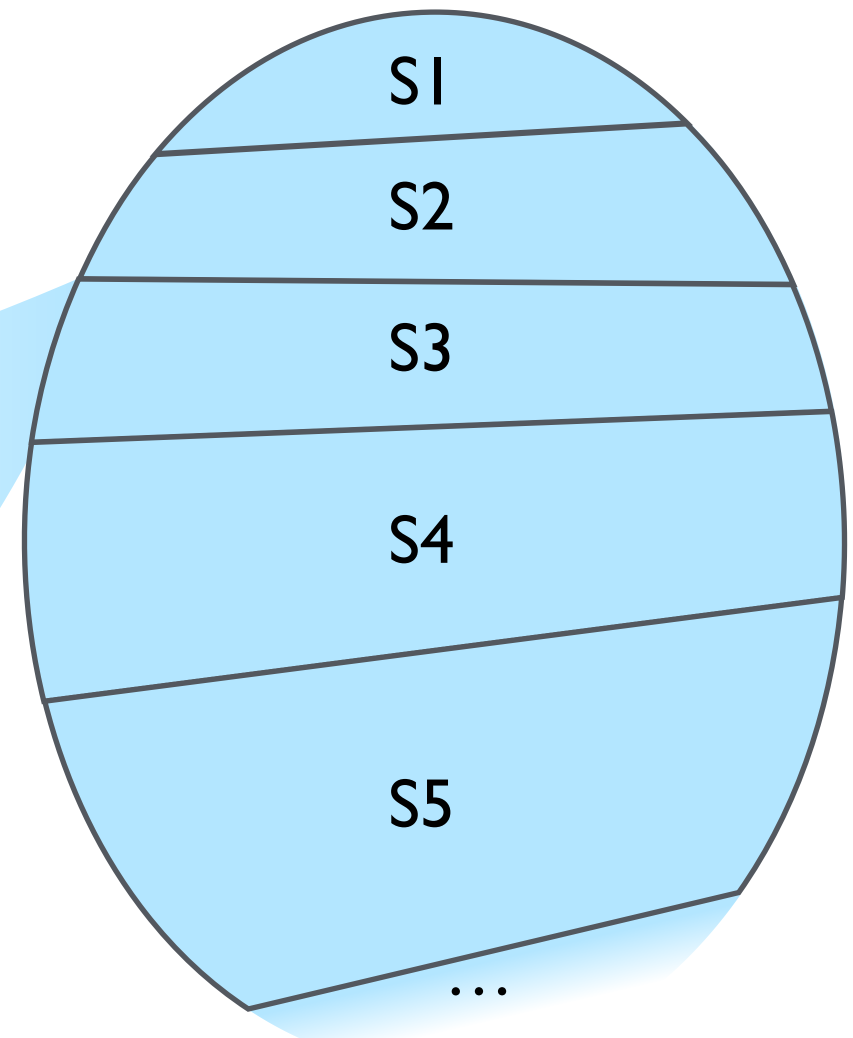
Metasketches express structure and strategy

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

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

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

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



S_3 (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

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

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

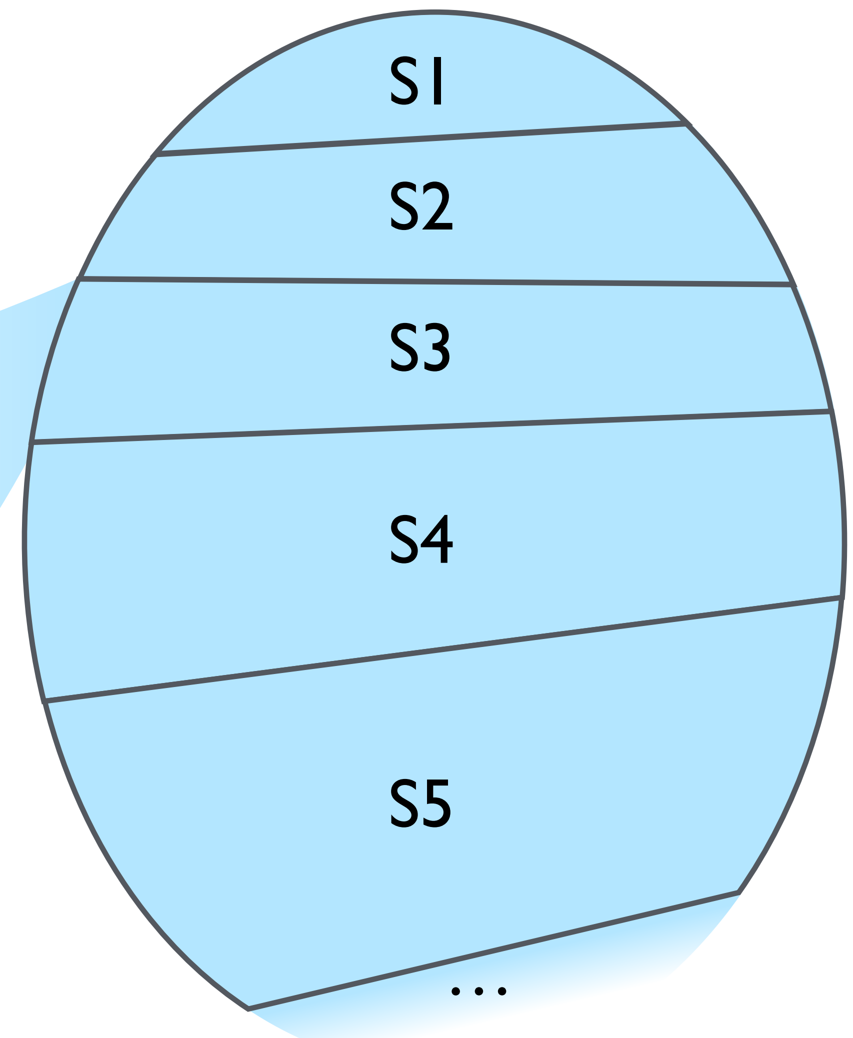
A fragmentation of the candidate space, and an ordering on those fragments.

S_3 (SSA programs of length 3)

+, -, <, if, …

```
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

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

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

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

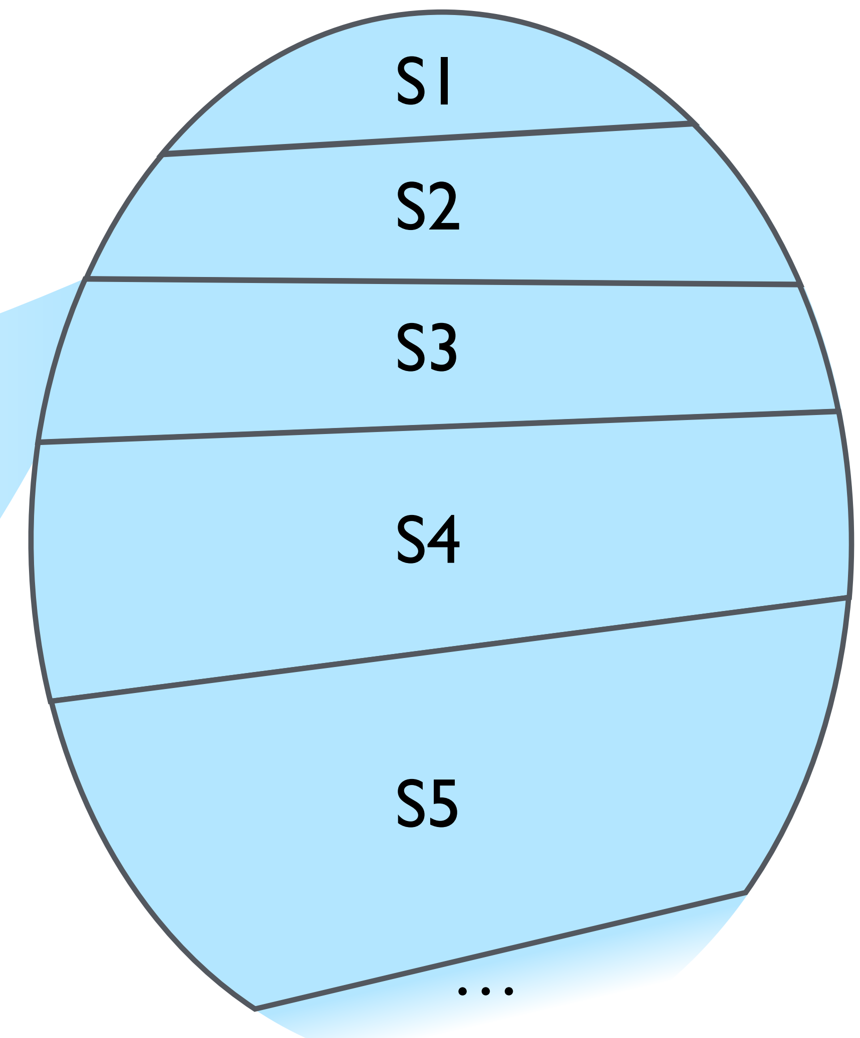
S_3 (SSA programs of length 3)

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

+, -, <, if, …

Vars & constants

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



2. cost function (κ)

3. gradient function (g)

Metasketches express structure and strategy

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

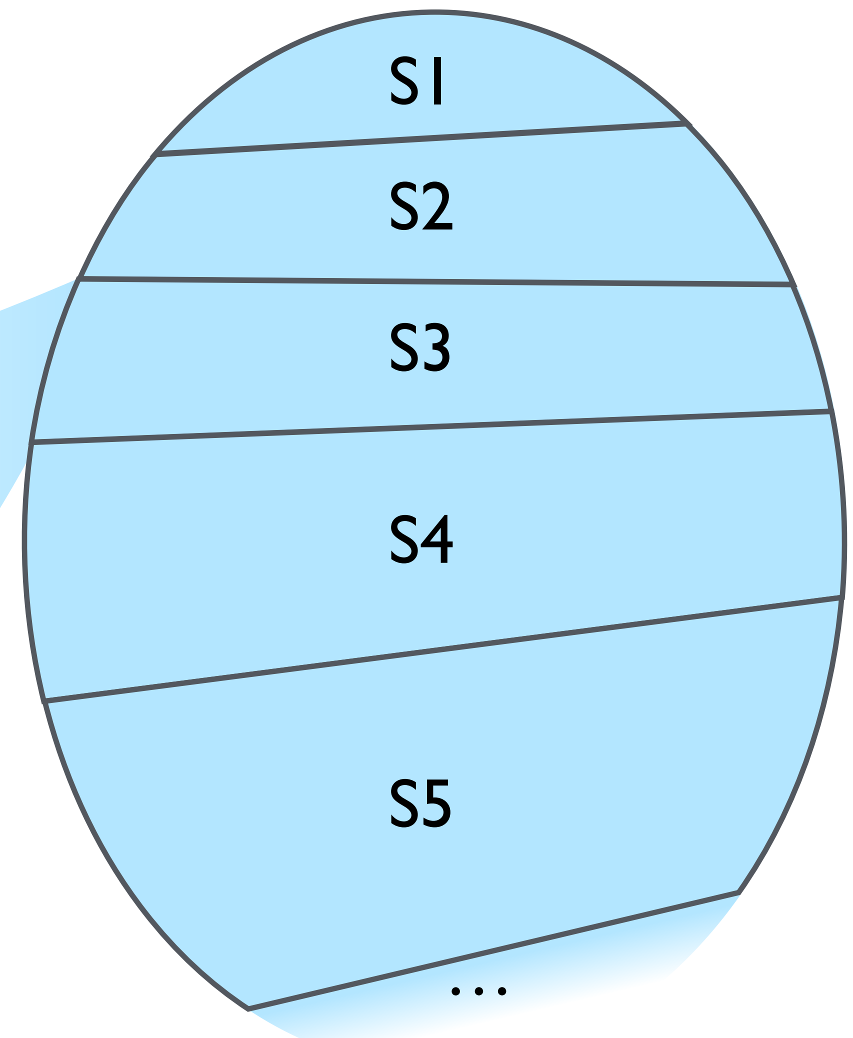
- a countable set $\mathcal{S}$ of sketches
- a total order $\preccurlyeq$ on $\mathcal{S}$

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

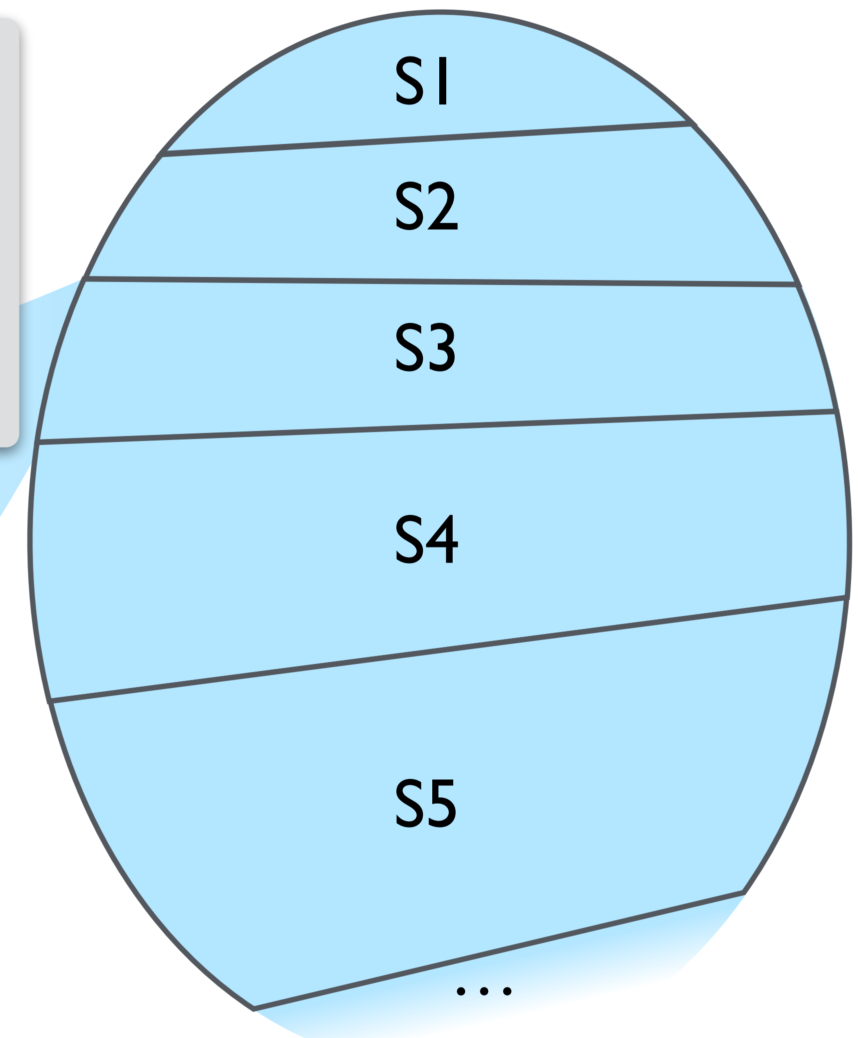
1. structured candidate space ($\mathcal{S}$, $\preccurlyeq$)

- a countable set $\mathcal{S}$ of sketches
- a total order $\preccurlyeq$ 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



S_3 (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

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

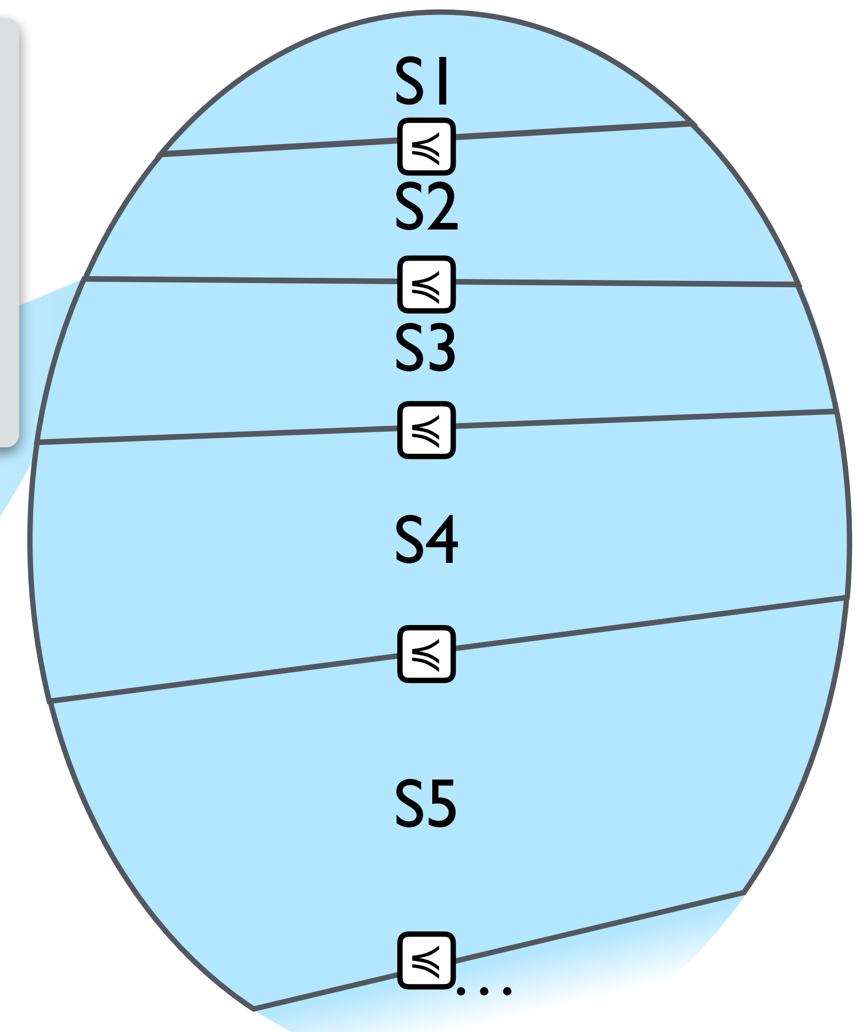
- a countable set $\mathcal{S}$ of sketches
- a total order $\preccurlyeq$ on $\mathcal{S}$

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

Ordering expresses high-level search strategy.

Here, $\preccurlyeq$ expresses iterative deepening.

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



S_3 (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

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

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

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

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

S_1
```
def f(x):
  r1 = ??op(??{x})
  return r1
```

S_2
```
def f(x):
  r1 = ??op(??{x})
  r2 = ??op(??{x,r1})
  return r2
```

S_3
```
def f(x):
  r1 = ??op(??{x})
  r2 = ??op(??{x,r1})
  r3 = ??op(??{x,r1,r2})
  return r3
```

...

Implemented as a generator that returns the next sketch in the space

2. cost function (κ)

3. gradient function (g)

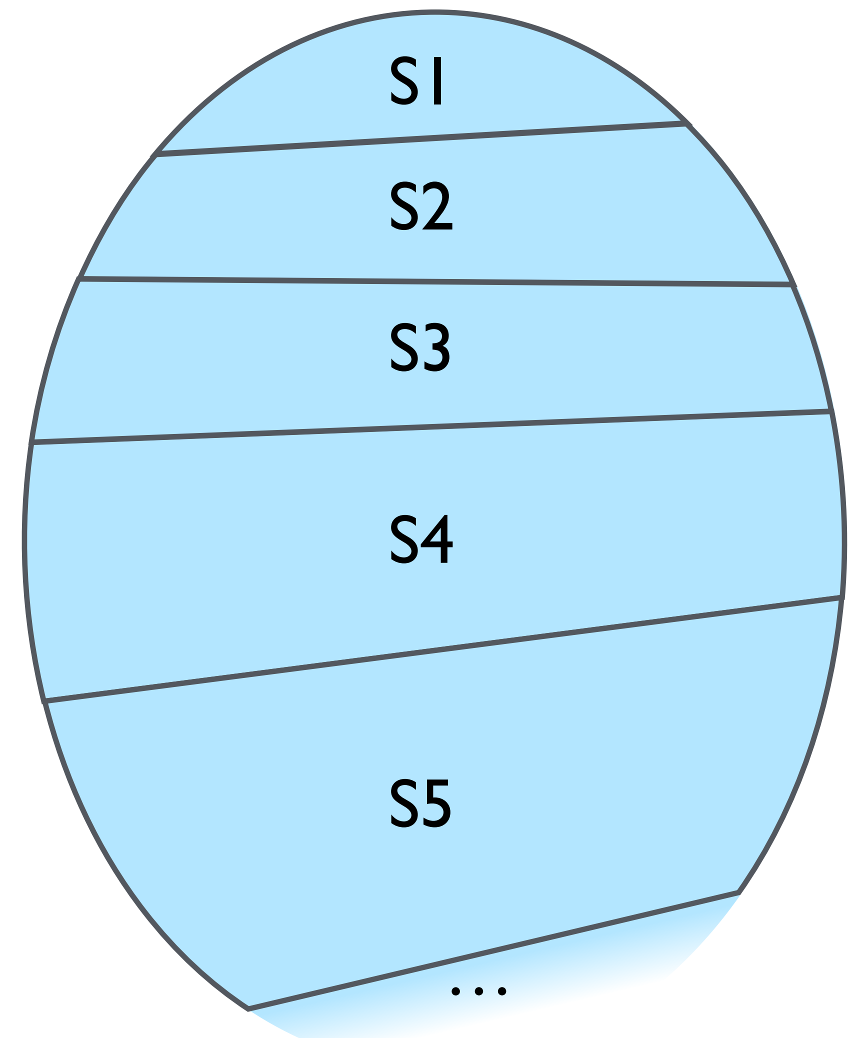
Metasketches express structure and strategy

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

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

Semantics

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



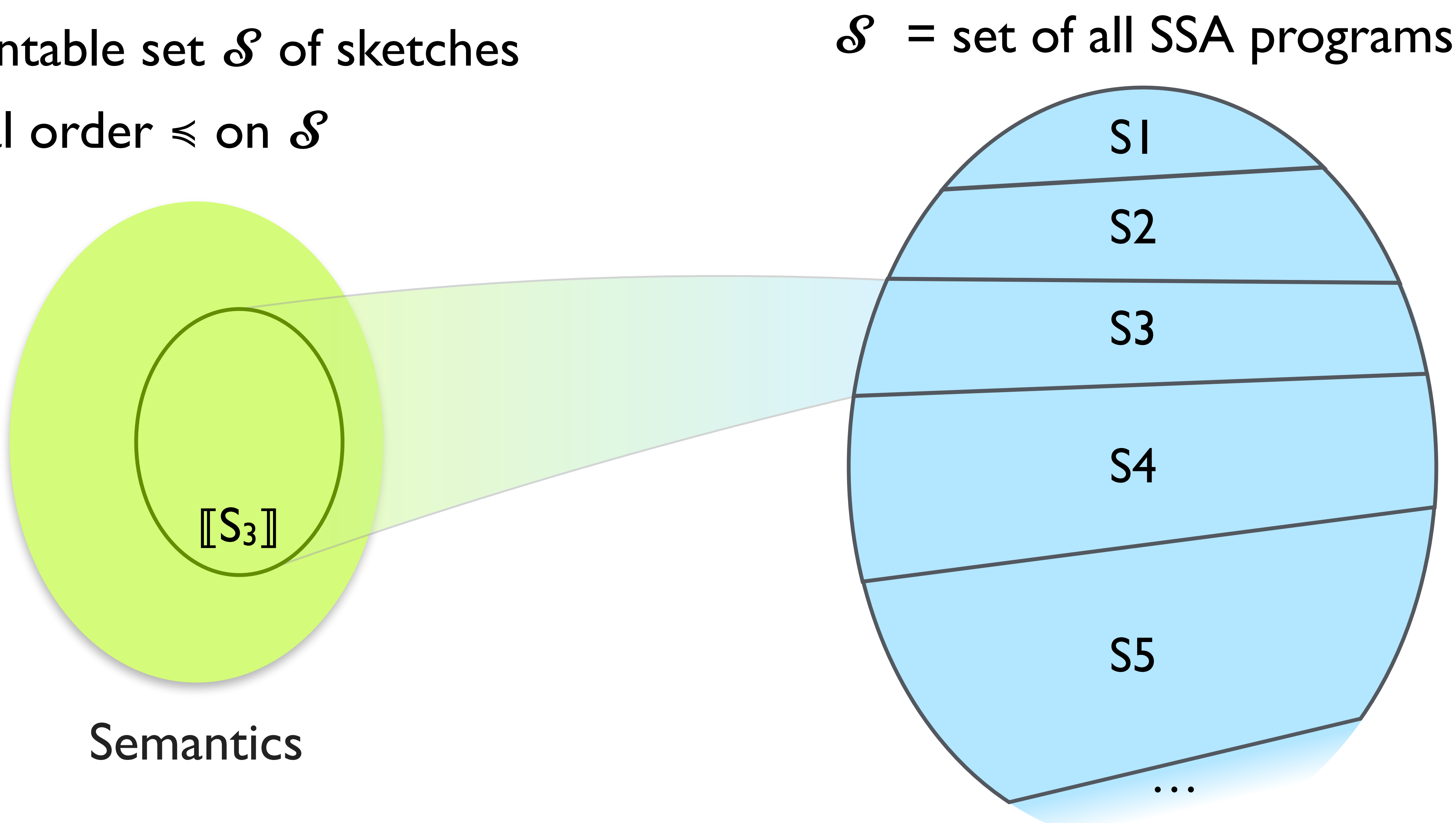
2. cost function (κ)

3. gradient function (g)

Metasketches express structure and strategy

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

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



2. cost function (κ)

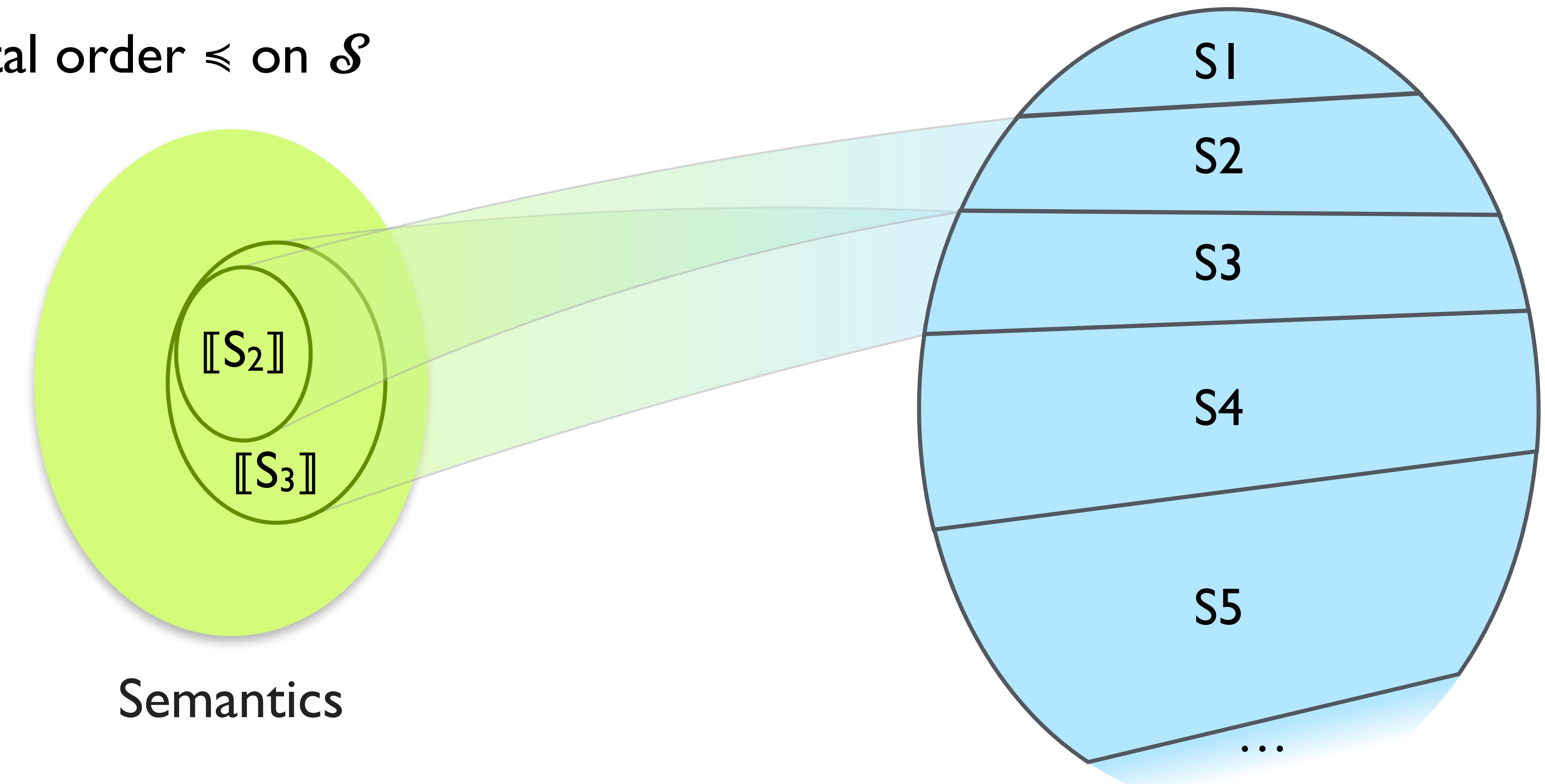
3. gradient function (g)

Metasketches express structure and strategy

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

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

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



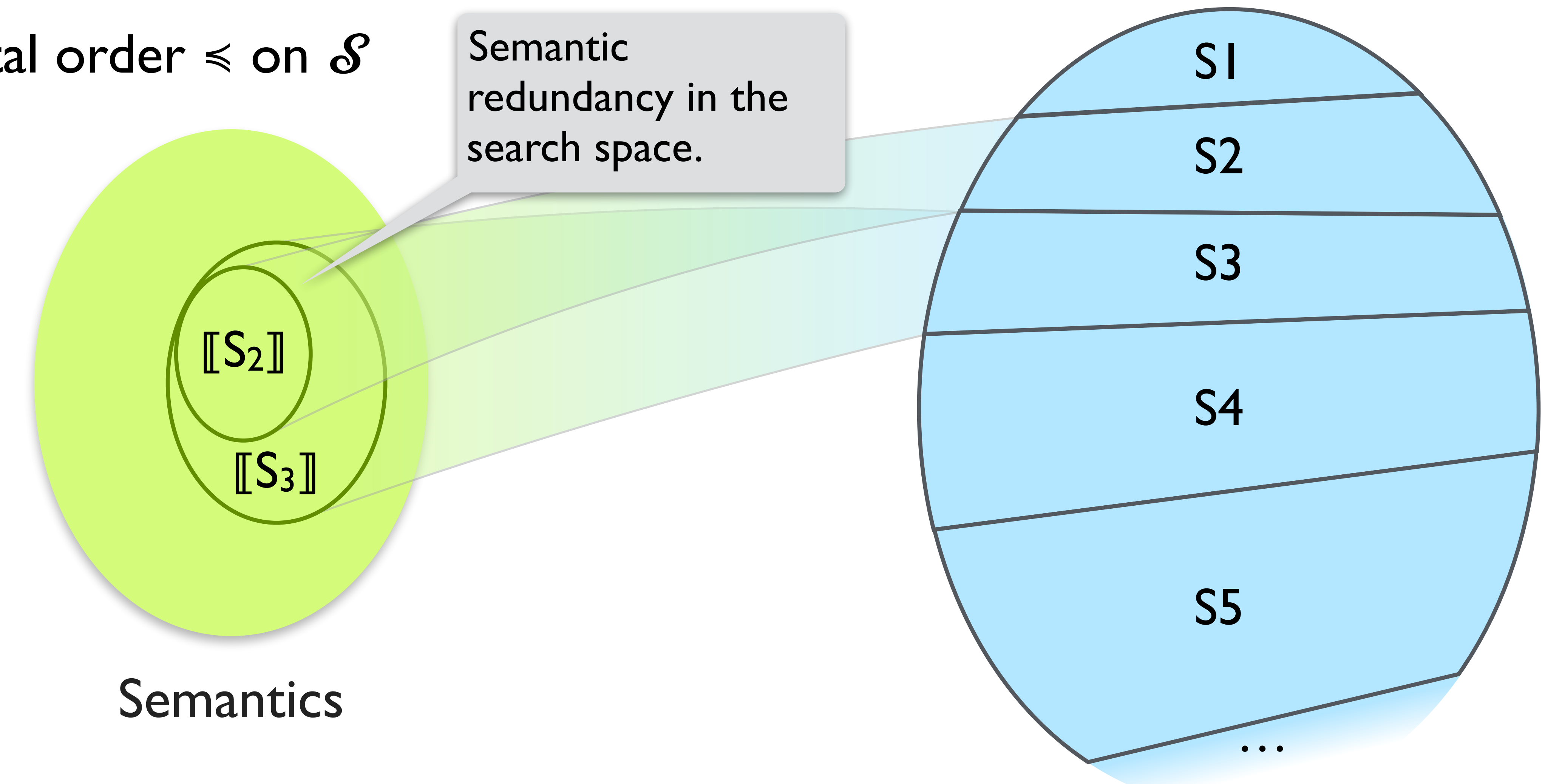
2. cost function (κ)

3. gradient function (g)

Metasketches express structure and strategy

1. **structured candidate space ($\mathcal{S}$, $\preccurlyeq$)**
   - a countable set $\mathcal{S}$ of sketches
   - a total order $\preccurlyeq$ on $\mathcal{S}$

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



2. **cost function (κ)**

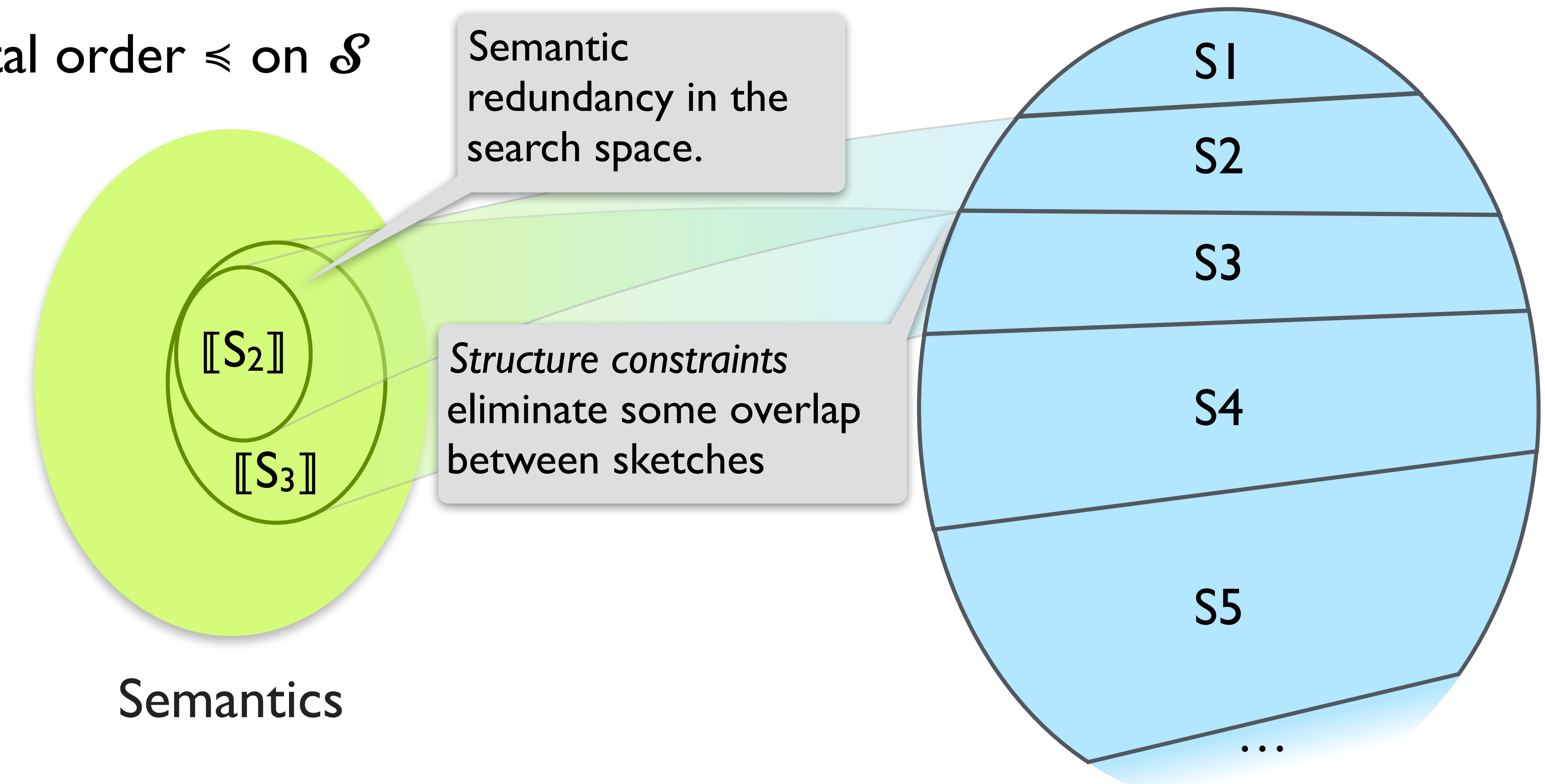
3. **gradient function (g)**

Metasketches express structure and strategy

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

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

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



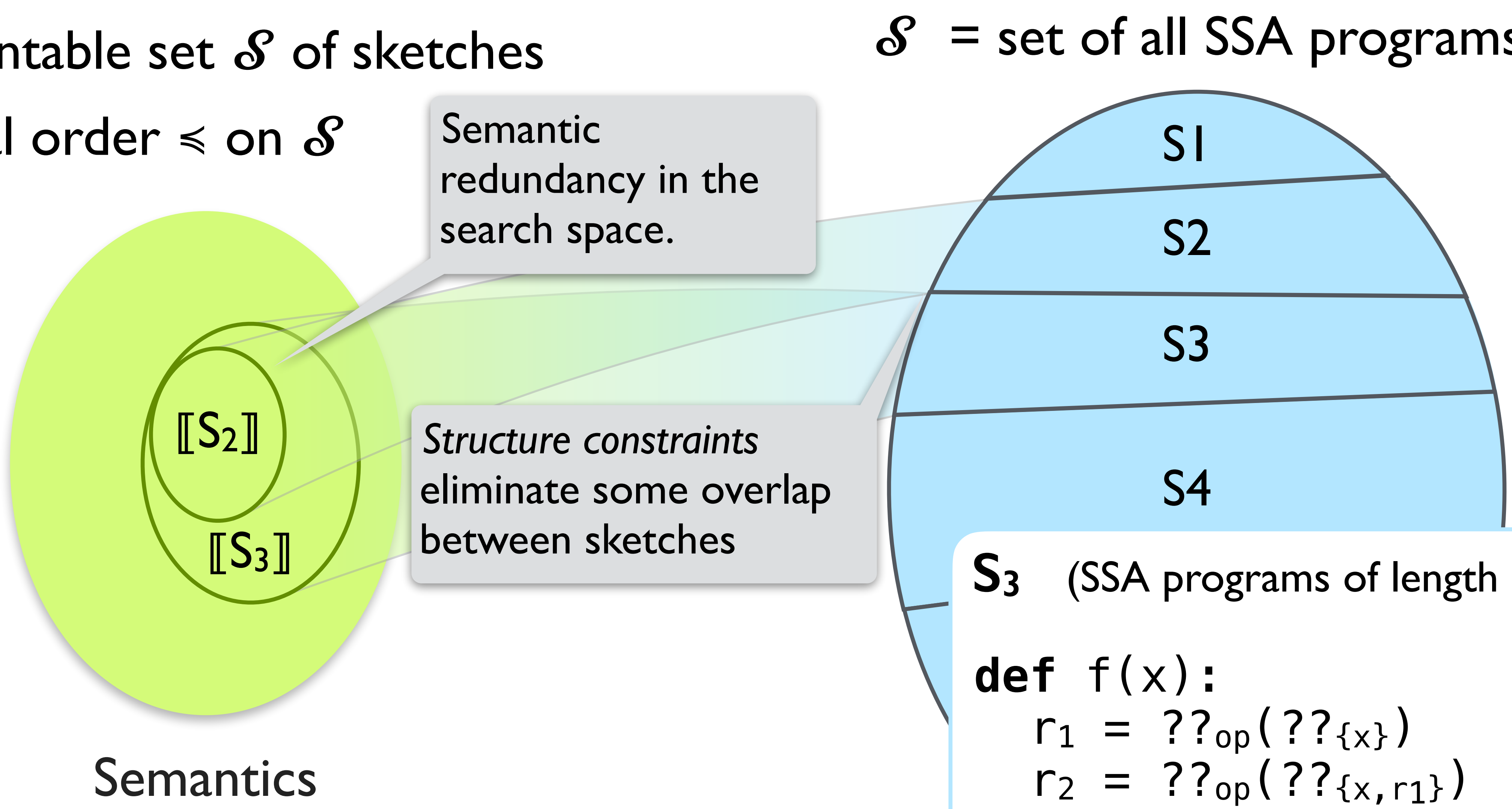
2. cost function (κ)

3. gradient function (g)

Metasketches express structure and strategy

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

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



S_3 (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)

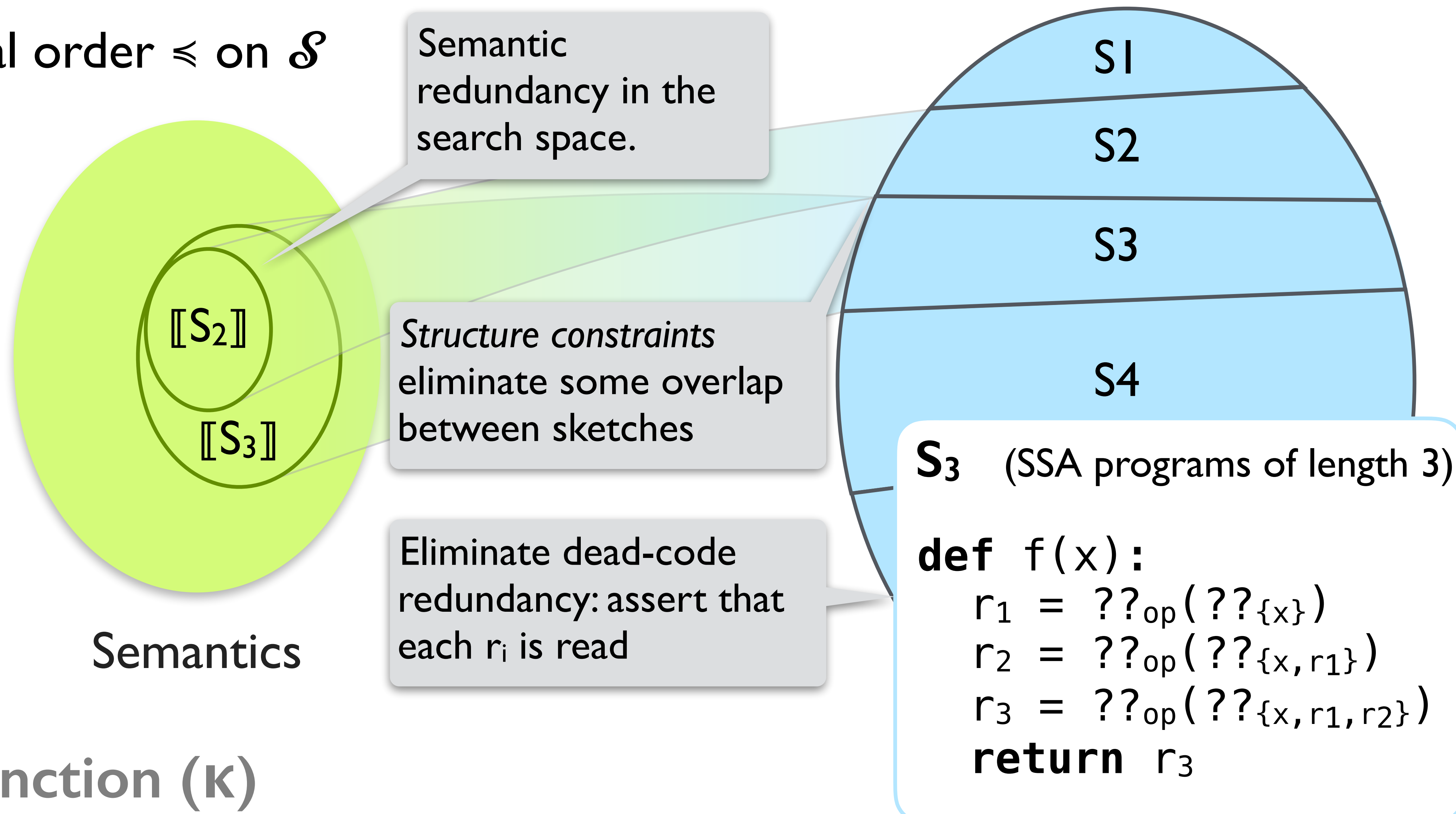
3. gradient function (g)

Metasketches express structure and strategy

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

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

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



2. cost function (κ)

3. gradient function (g)

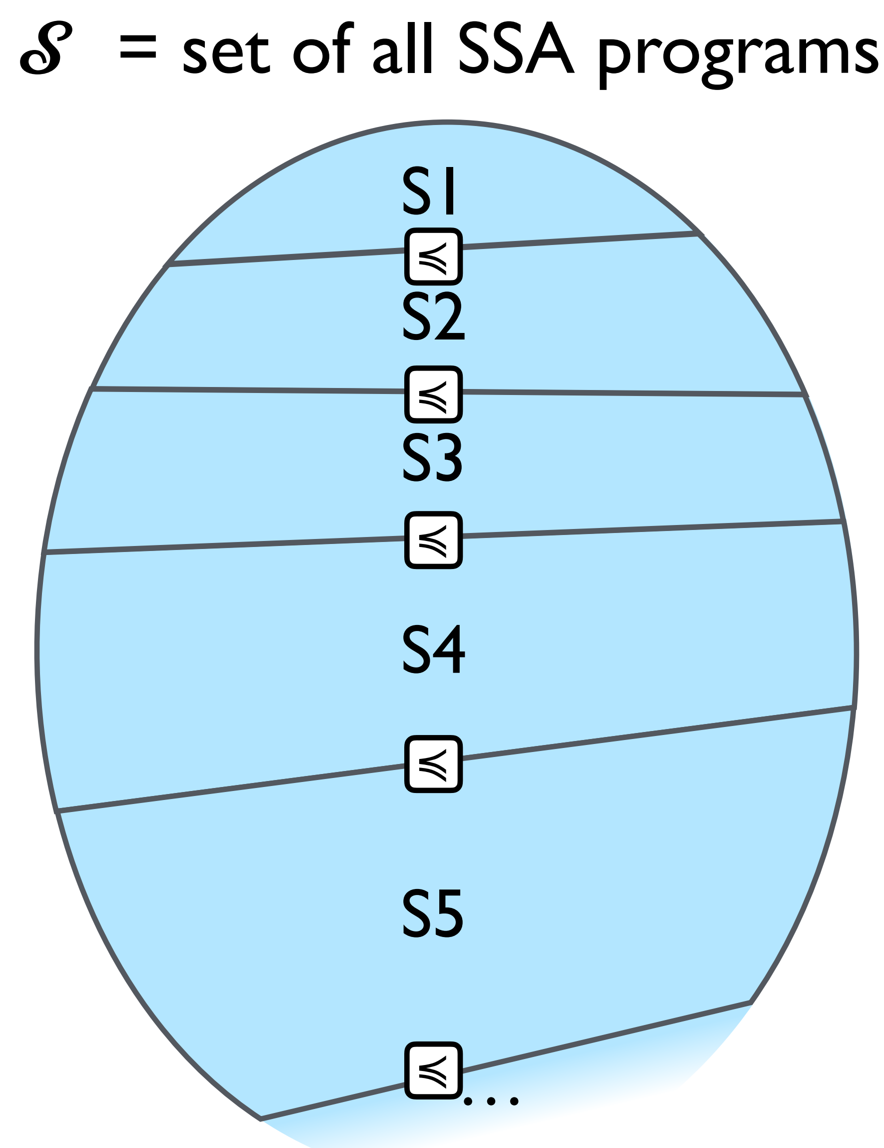
Cost functions rank candidate programs

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

2. **cost function (κ)**

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

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



3. gradient function (g)

Cost functions rank candidate programs

1. structured candidate space ($\mathscr{S}$, $\preccurlyeq$)

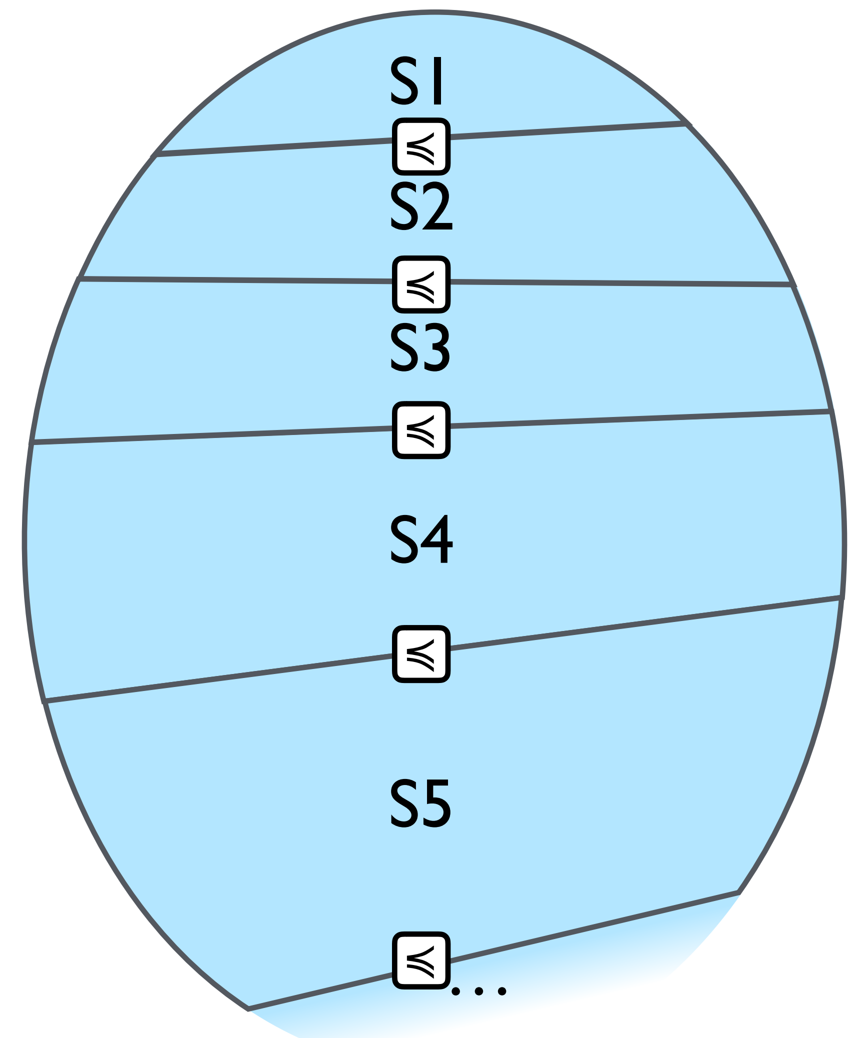
2. **cost function (κ)**

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

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

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

$\mathscr{S}$ = set of all SSA programs



3. gradient function (g)

Cost functions rank candidate programs

1. structured candidate space ($\mathscr{S}$, $\preccurlyeq$)

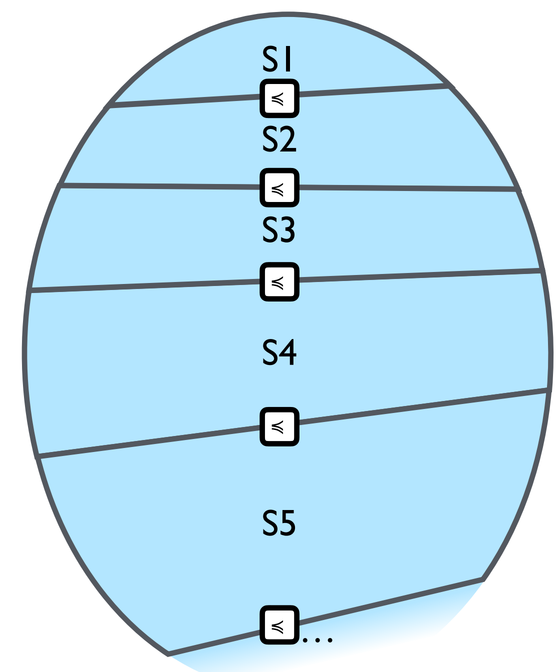
2. **cost function (κ)**

$\kappa : \mathscr{L} \to \mathbb{R}$

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

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

$\mathscr{S}$ = set of all SSA programs



$\kappa(P) = i$ for $P \in S_i \in \mathscr{S}$

The number of variables defined in P

3. gradient function (g)

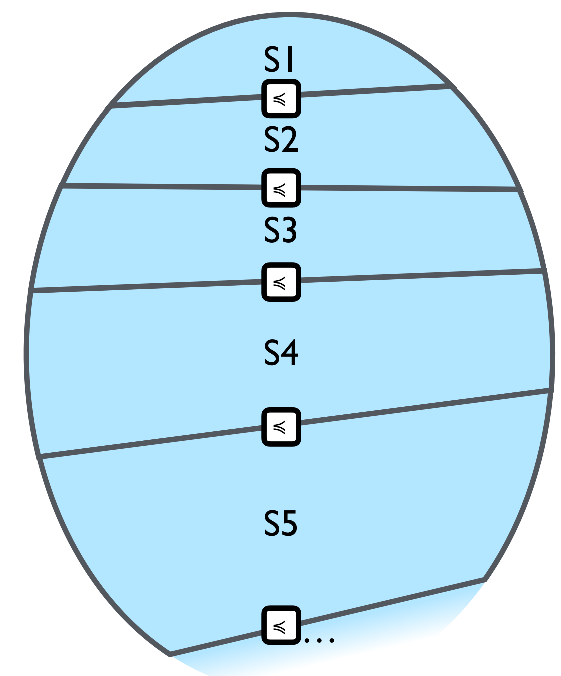
Gradient functions provide cost structure

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

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

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



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

Gradient functions provide cost structure

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

2. cost function (κ)

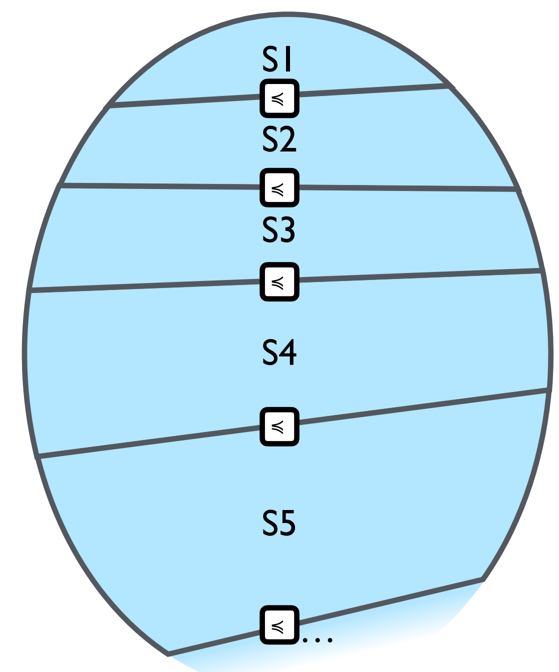
3. gradient function (g)

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

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



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

Gradient functions provide cost structure

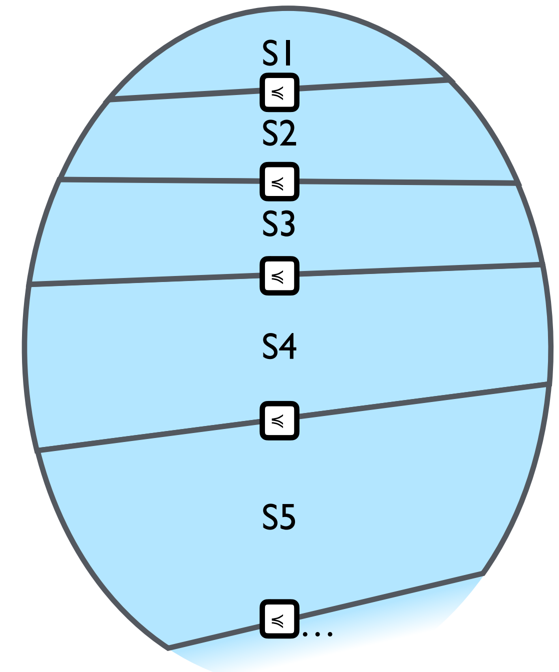
1. structured candidate space ($\mathcal{S}$, $\preccurlyeq$)
2. cost function (κ)
3. **gradient function (g)**

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

$g(c)$ 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



$\kappa(P) = i$ 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 ($\mathcal{S}$, $\preccurlyeq$)

2. cost function (κ)

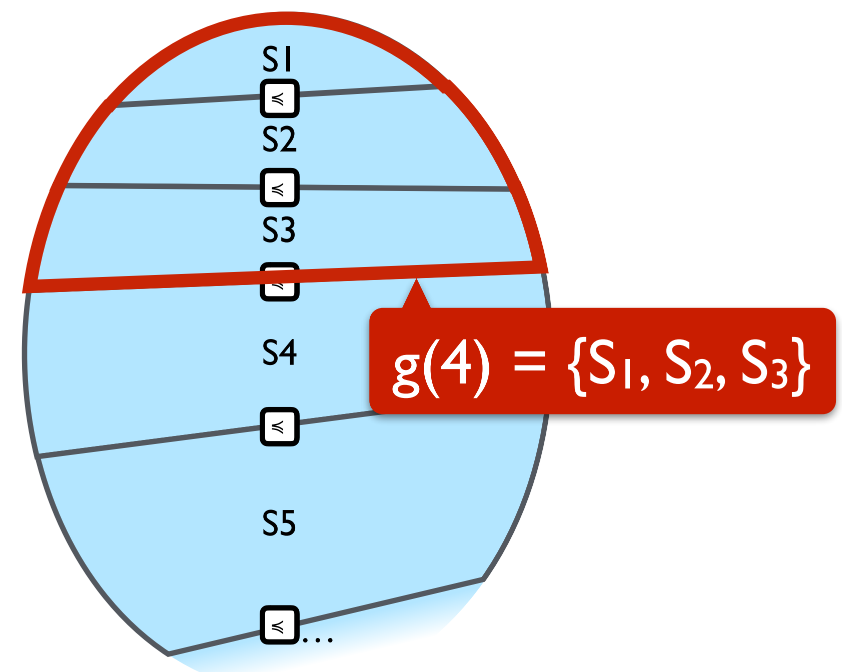
3. **gradient function (g)**

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

   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



$\kappa(P) = i$ 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 ($\mathscr{S}$, $\preccurlyeq$)
2. cost function (κ)
3. **gradient function (g)**

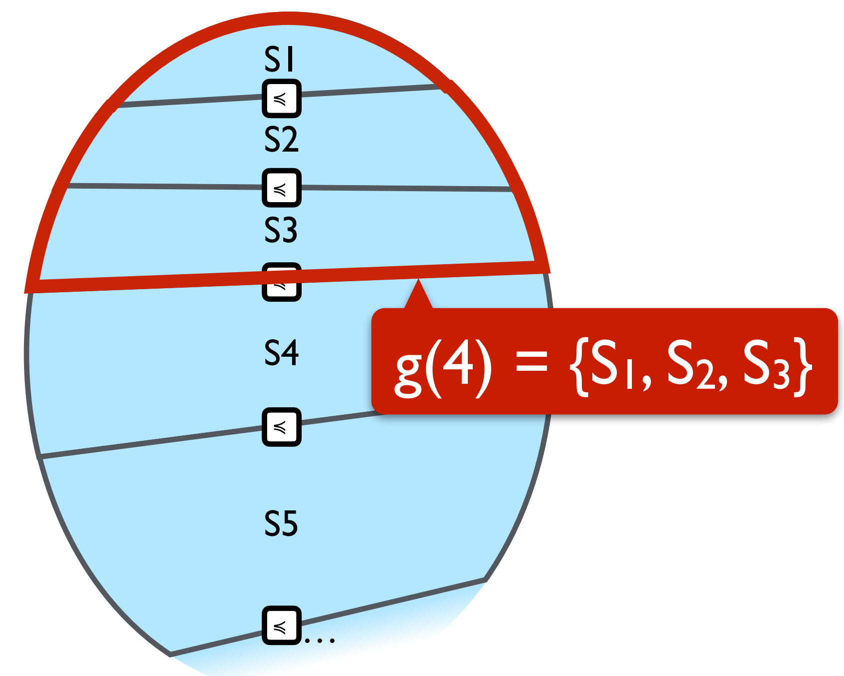
$g : \mathbb{R} \rightarrow 2^{\mathscr{S}}$

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

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

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

$\mathscr{S}$ = set of all SSA programs



$\kappa(P) = i$ for $P \in S_i \in \mathscr{S}$

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

Gradient functions provide cost structure

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

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

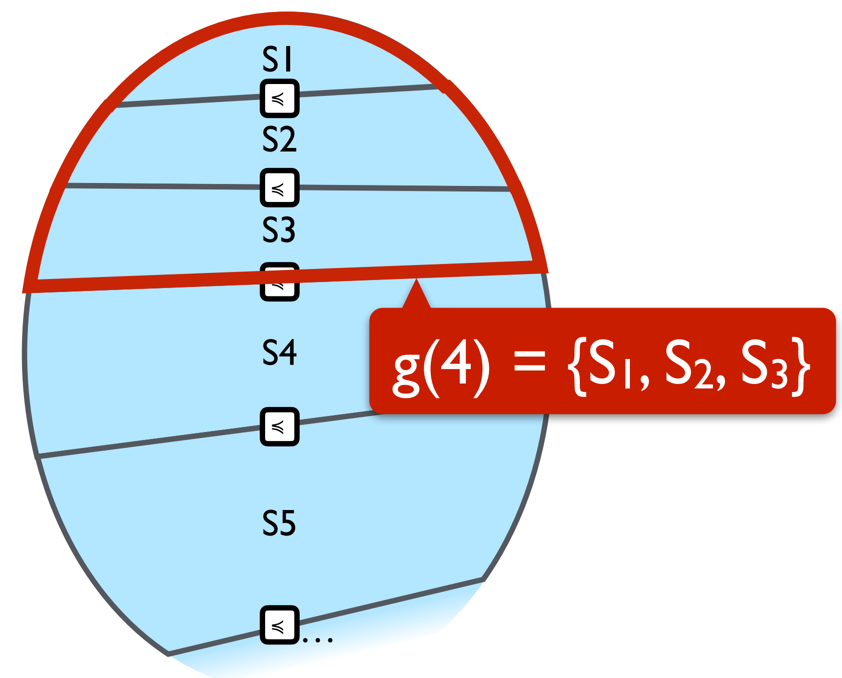
$g(c)$ is the set of sketches in $\mathcal{S}$ that *may* contain a solution P with $\kappa(P) < c$

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



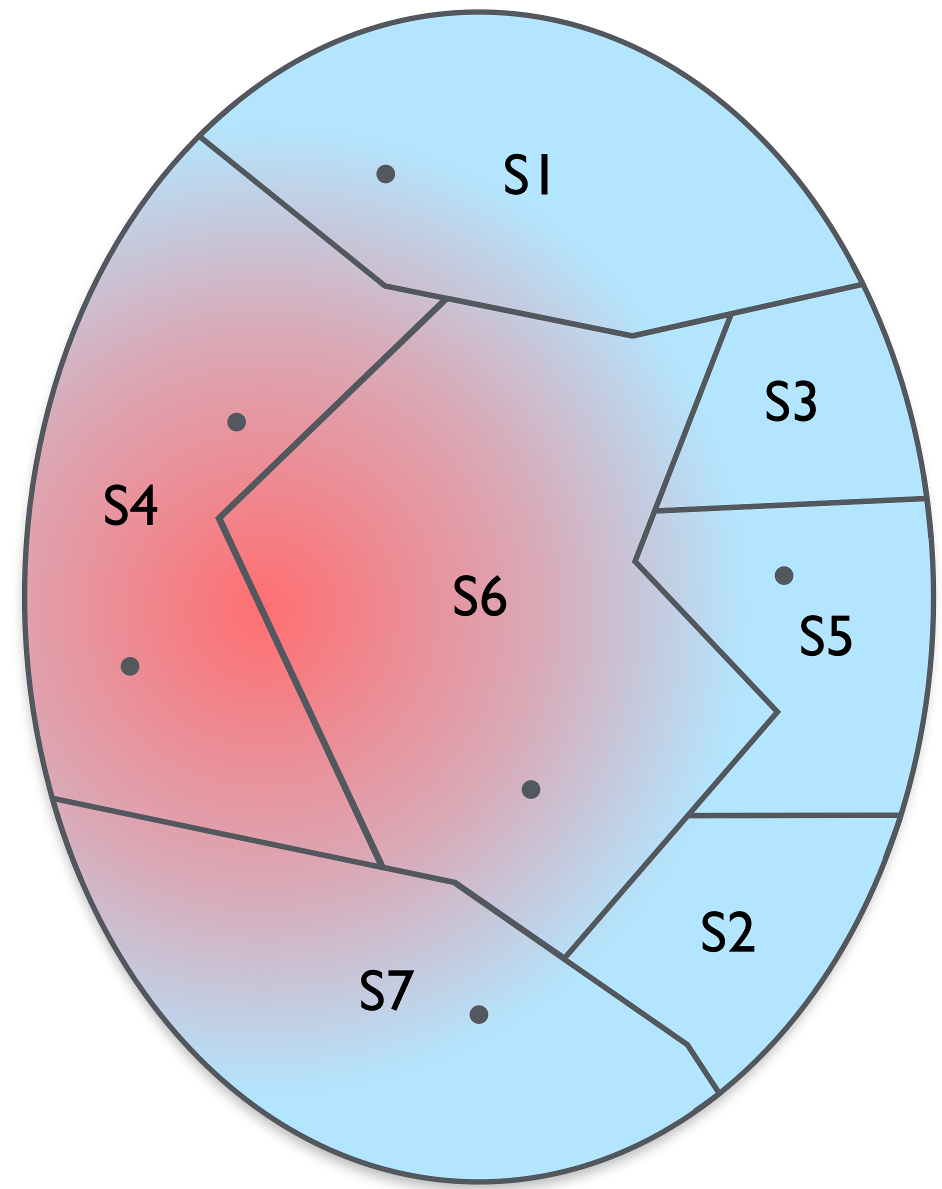
$\kappa(P) = i$ for $P \in S_i \in \mathcal{S}$

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

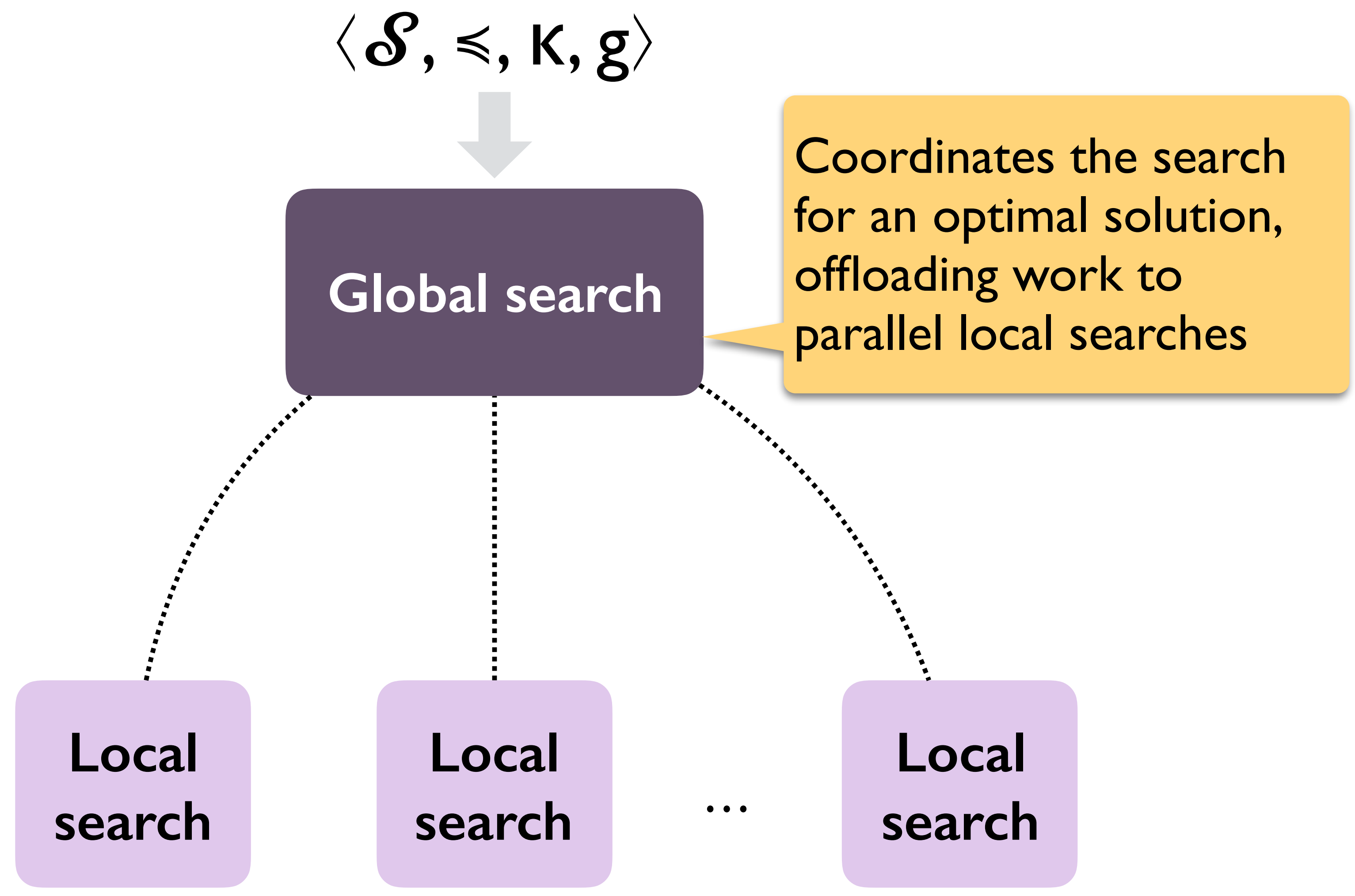
Metasketches express structure and strategy

<u>**A metasketch contains:**</u>

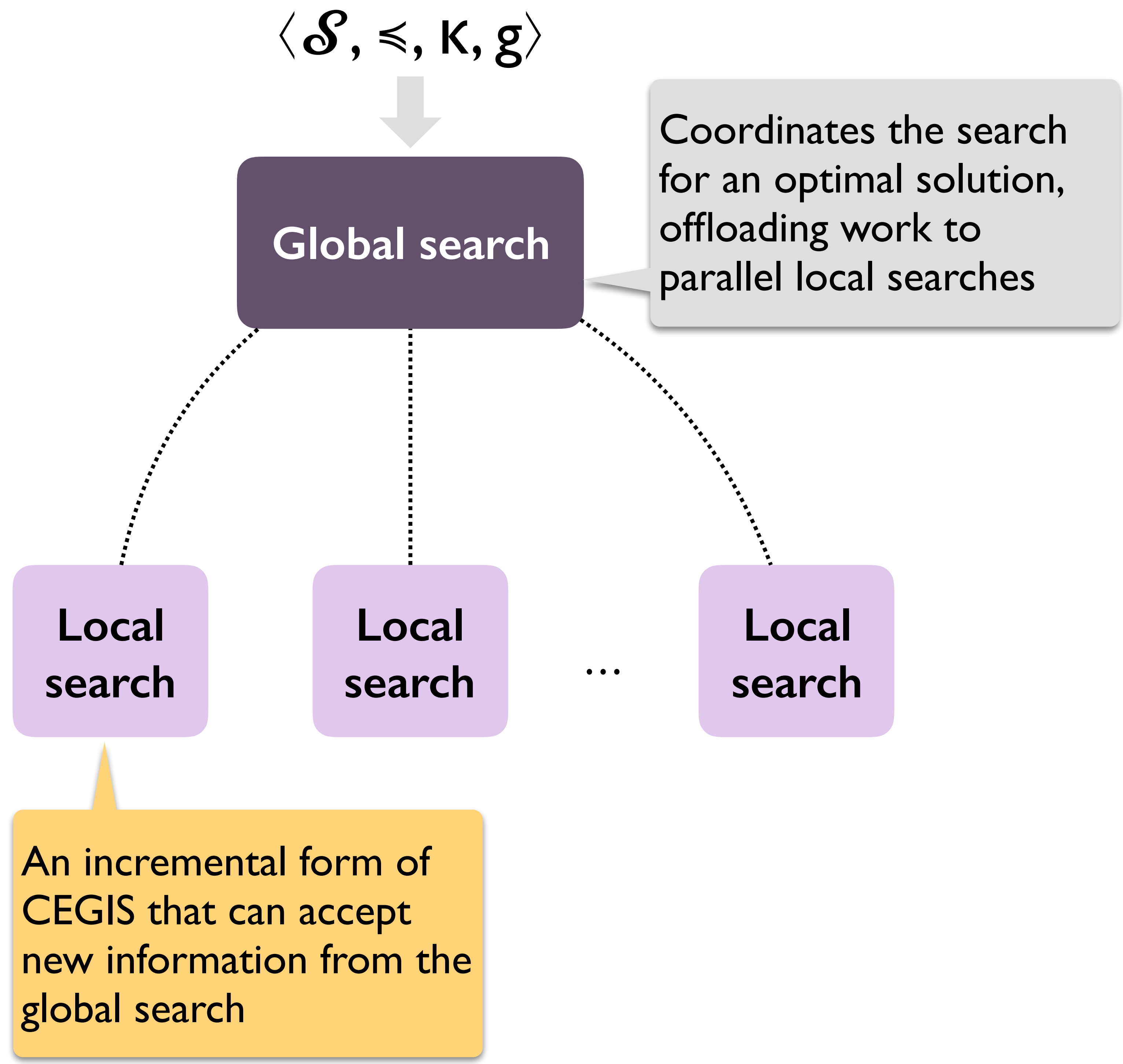
1. structured candidate space $(\mathcal{S}, \preccurlyeq)$
2. cost function (κ)
3. gradient function (g)



Solving with two cooperative searches

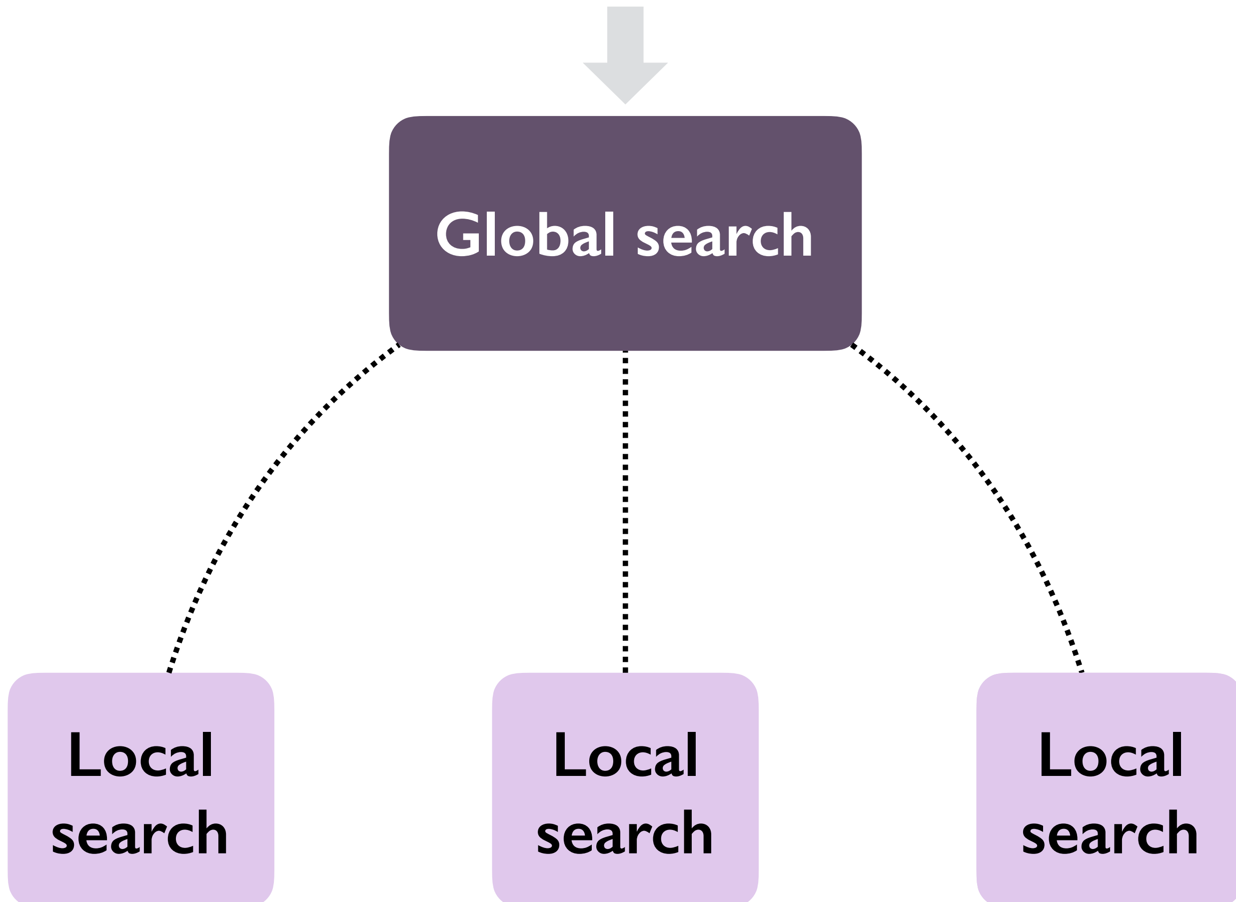


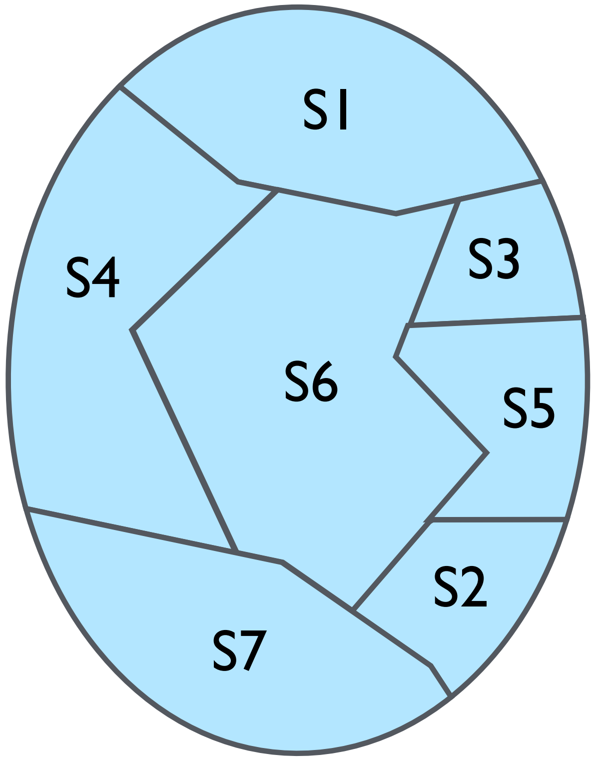
Solving with two cooperative searches



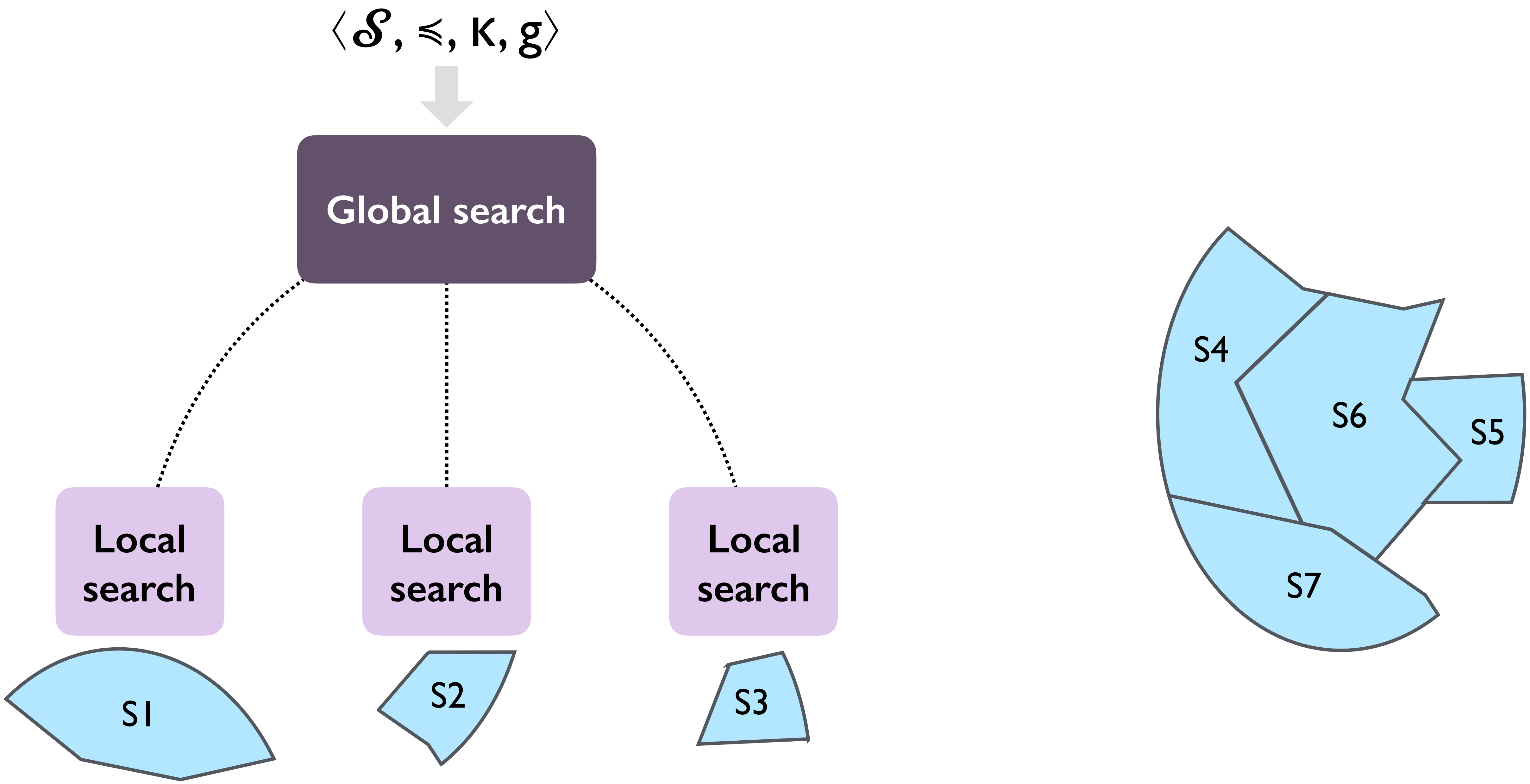
Solving with two cooperative searches

$\langle \mathcal{S}, \preccurlyeq, \kappa, g \rangle$

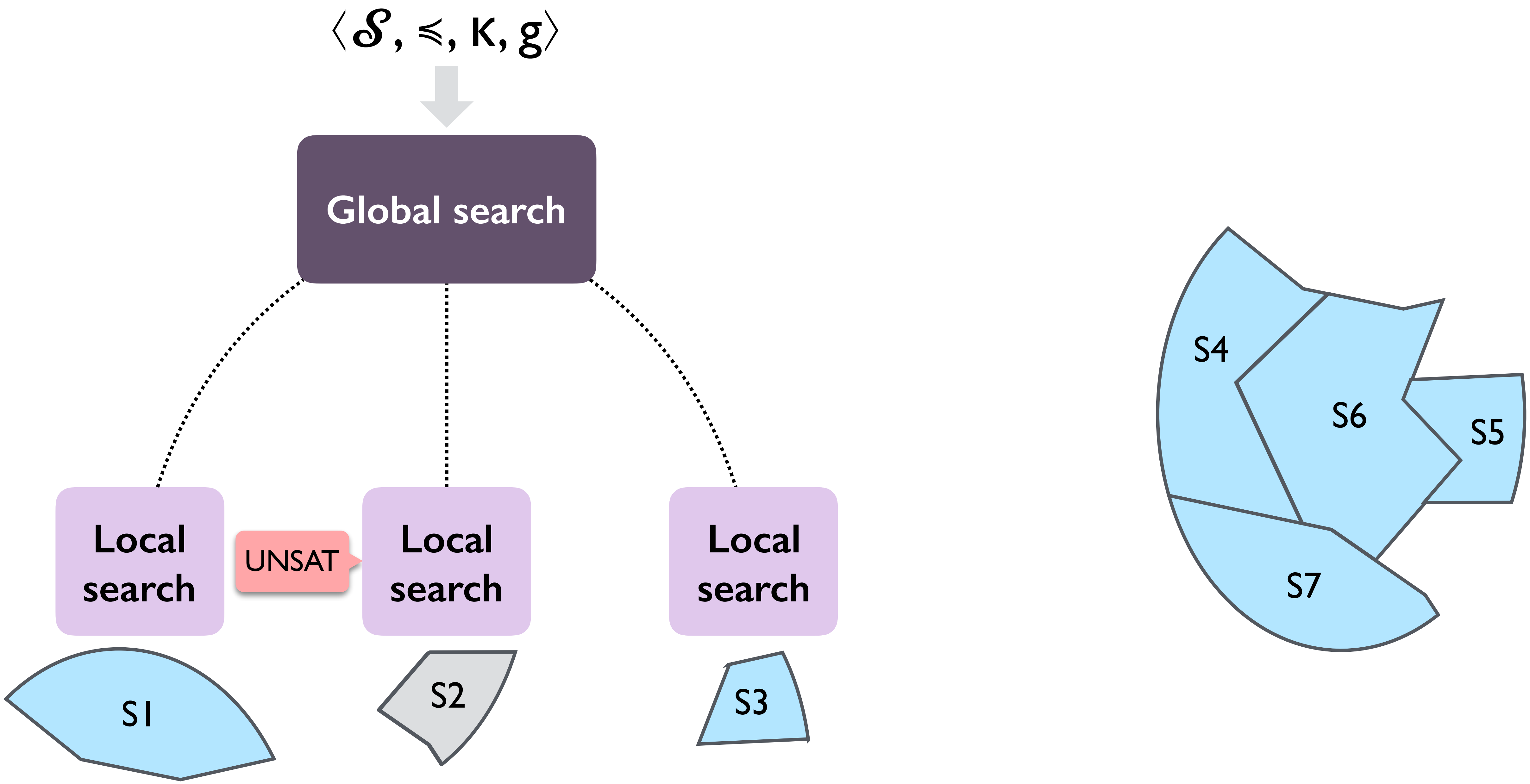




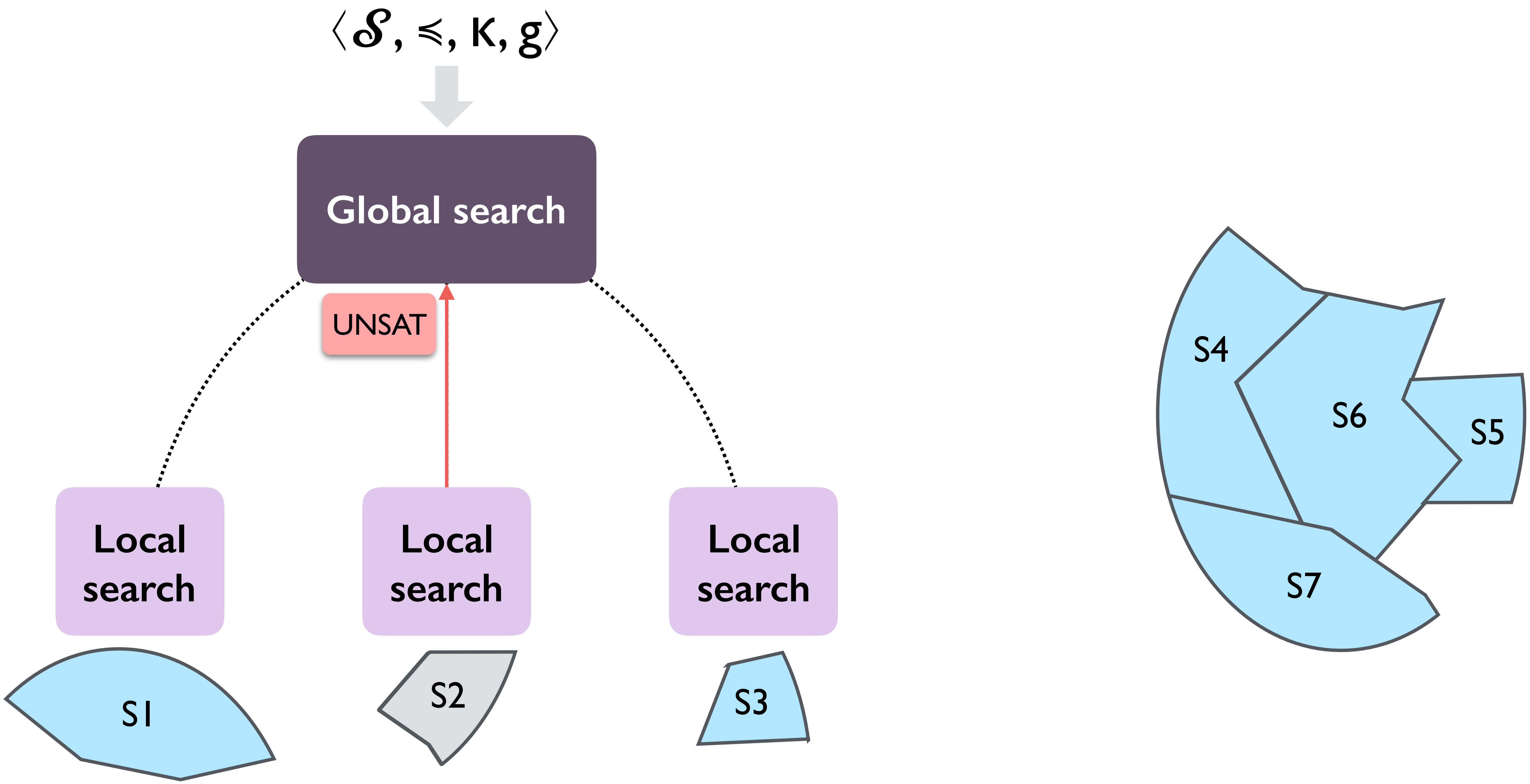
Solving with two cooperative searches



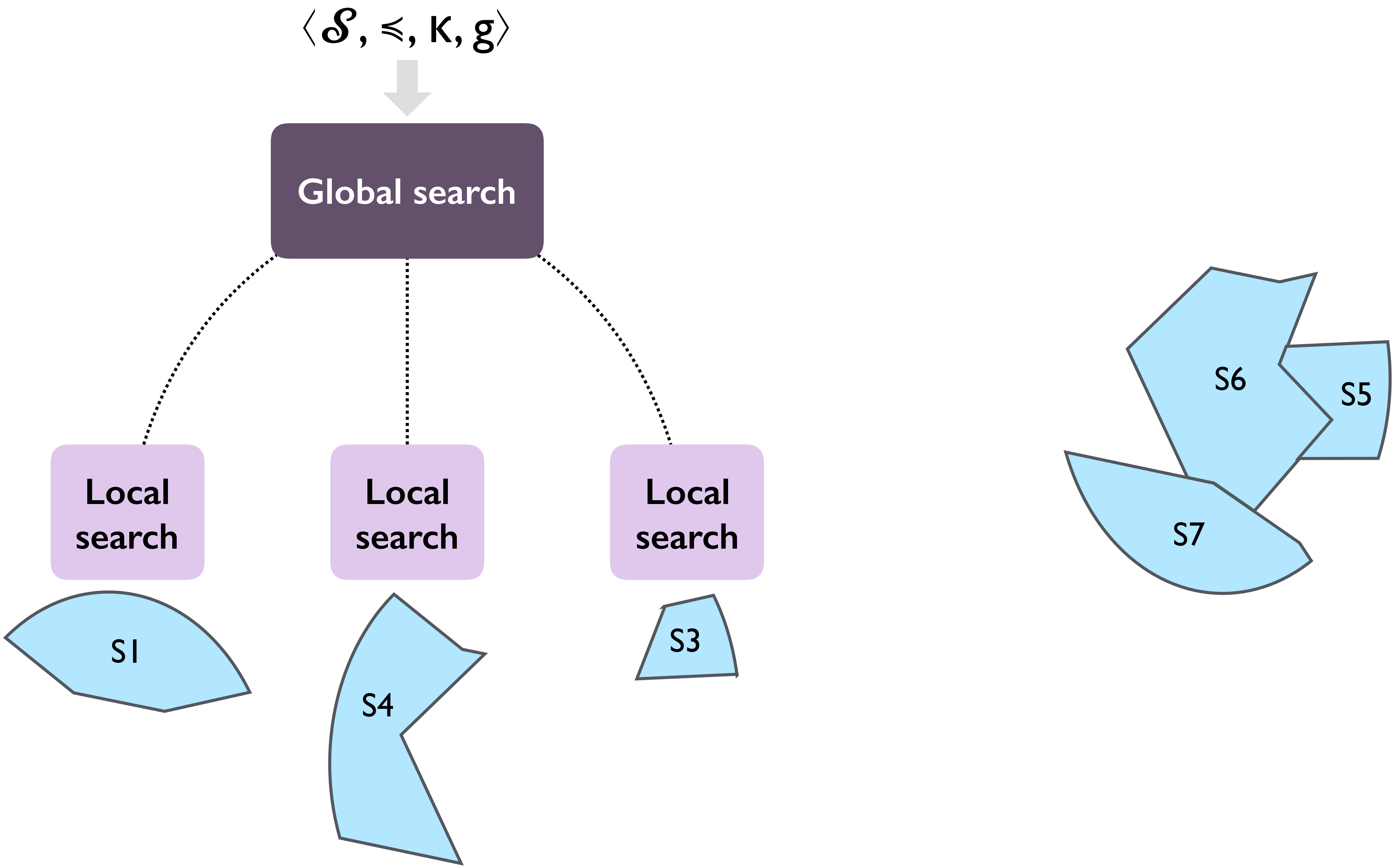
Solving with two cooperative searches



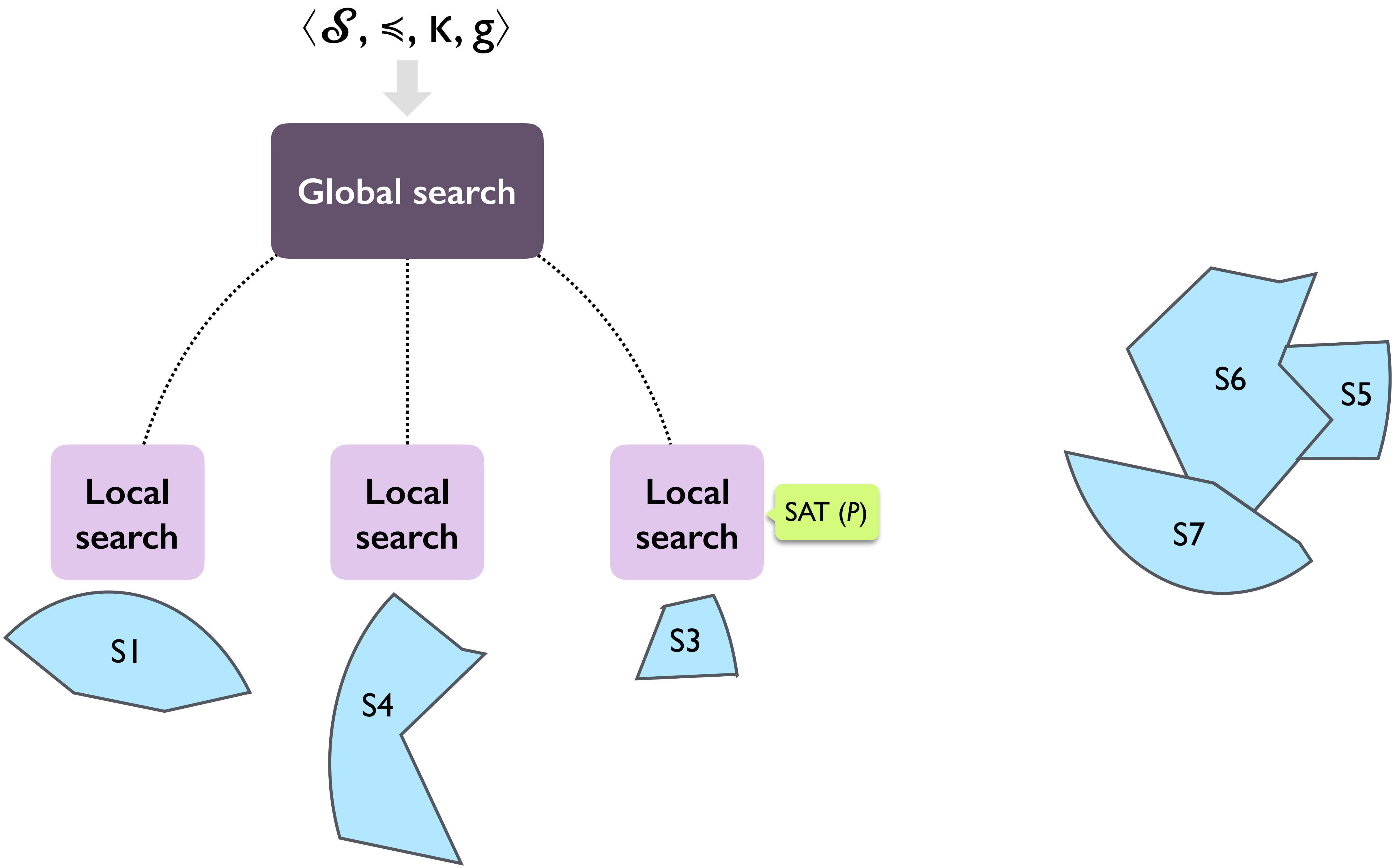
Solving with two cooperative searches



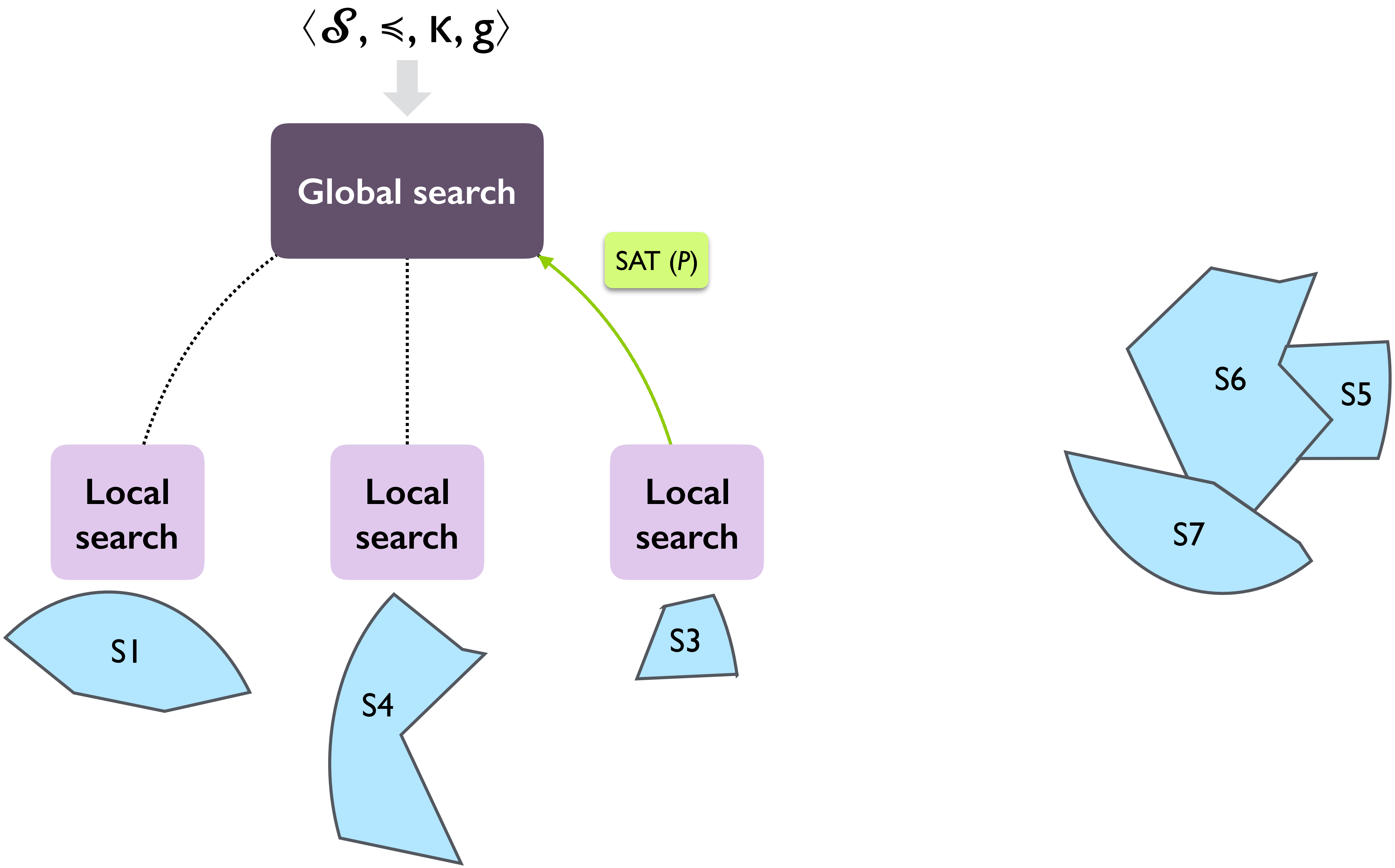
Solving with two cooperative searches



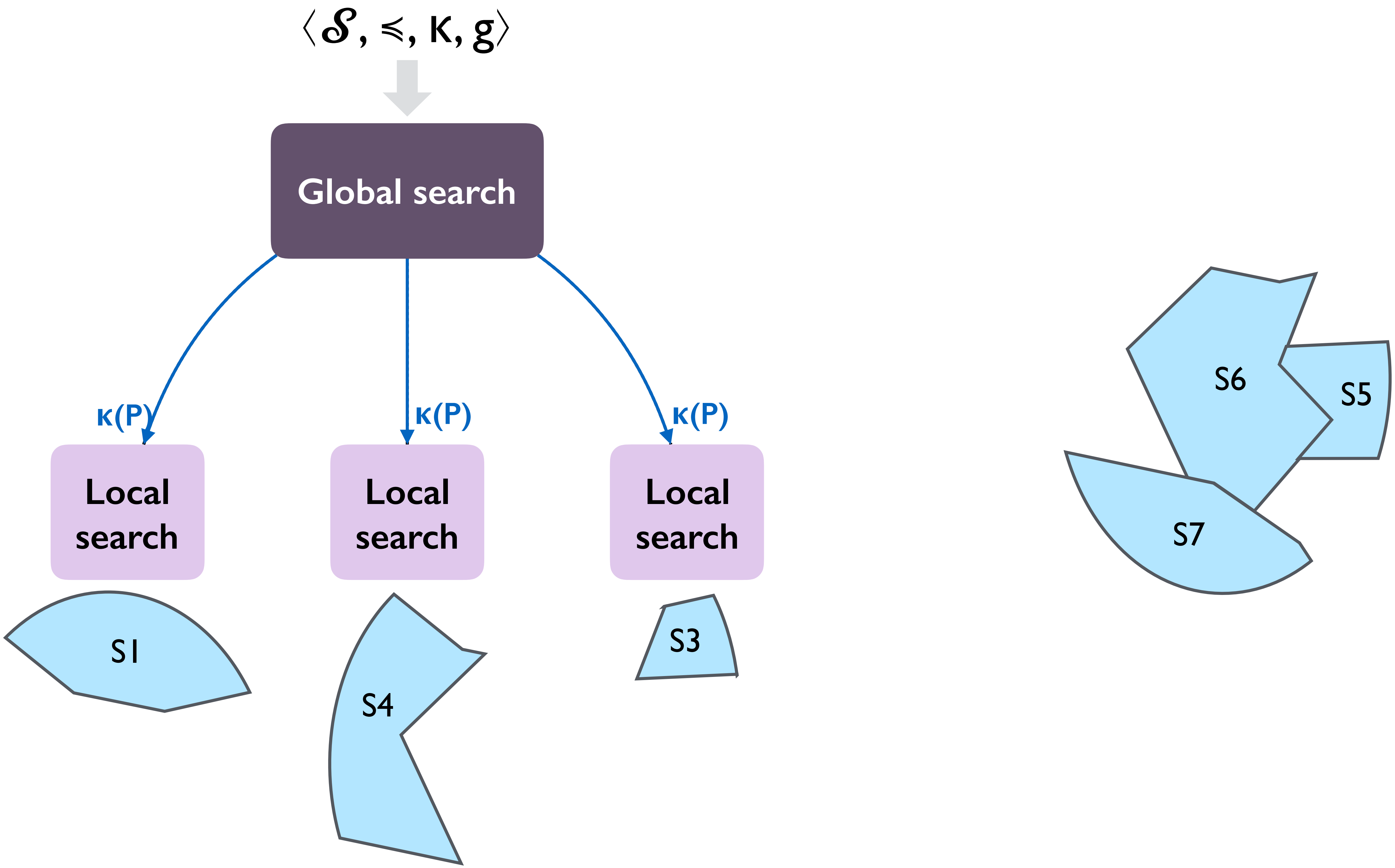
Solving with two cooperative searches



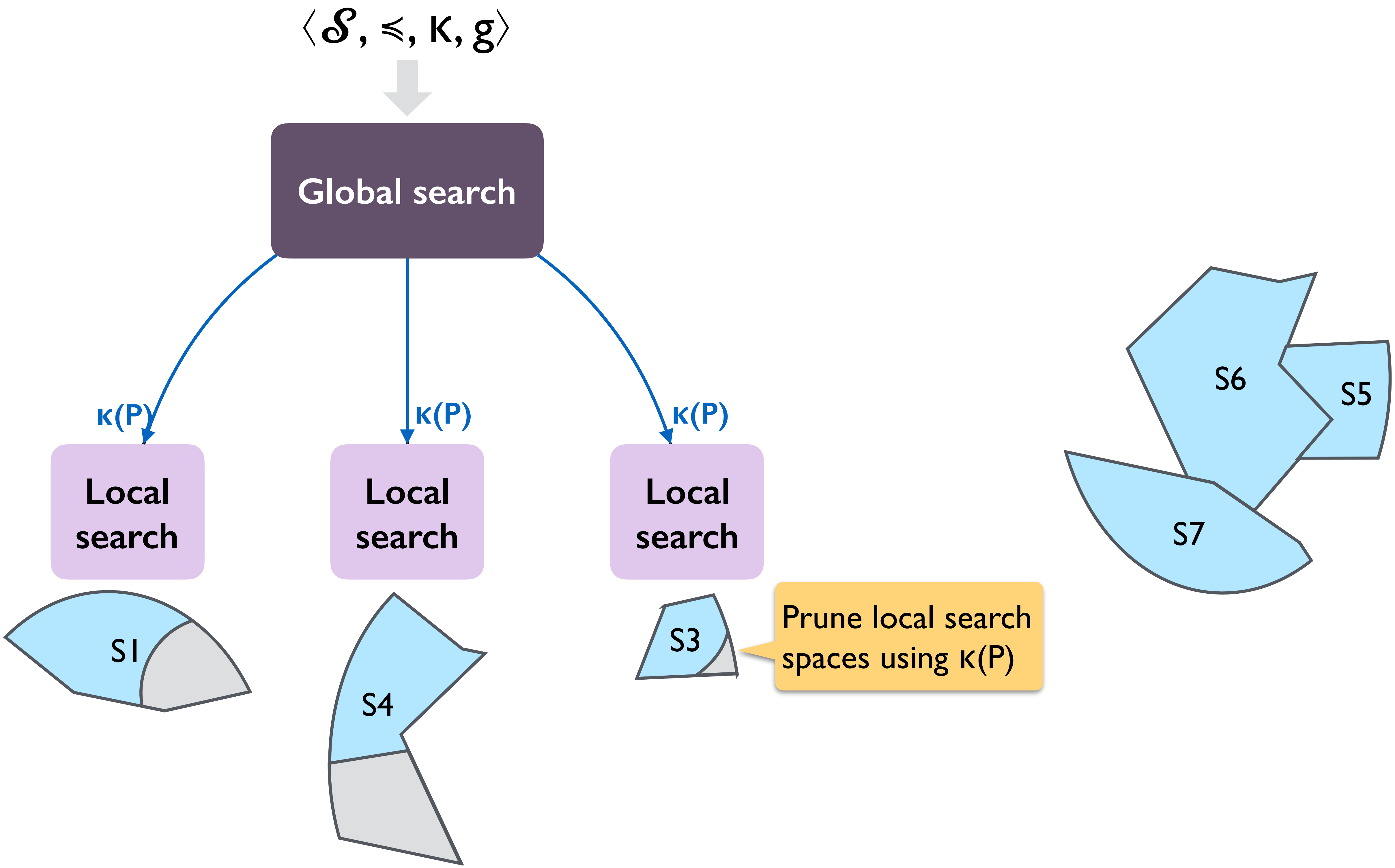
Solving with two cooperative searches



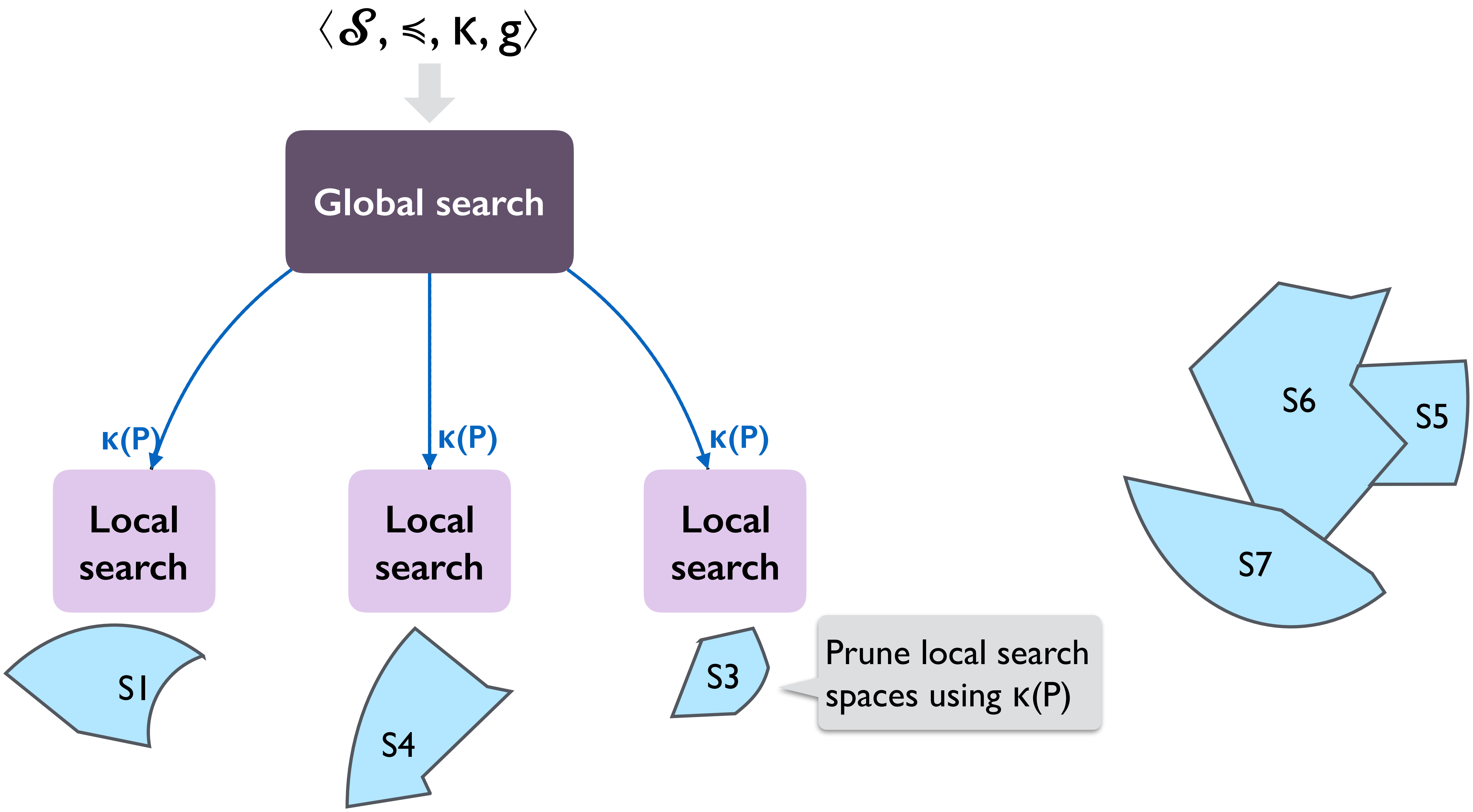
Solving with two cooperative searches



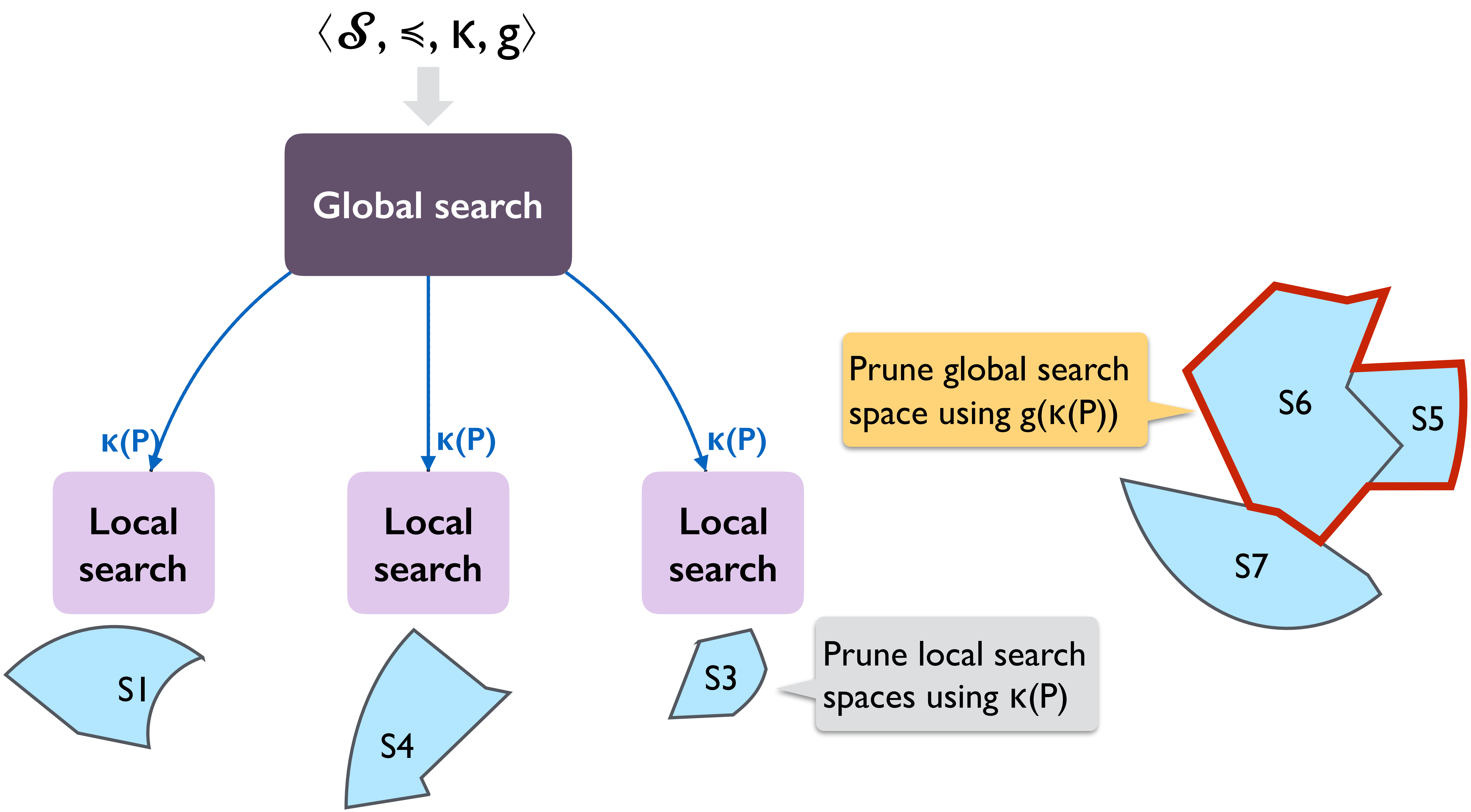
Solving with two cooperative searches



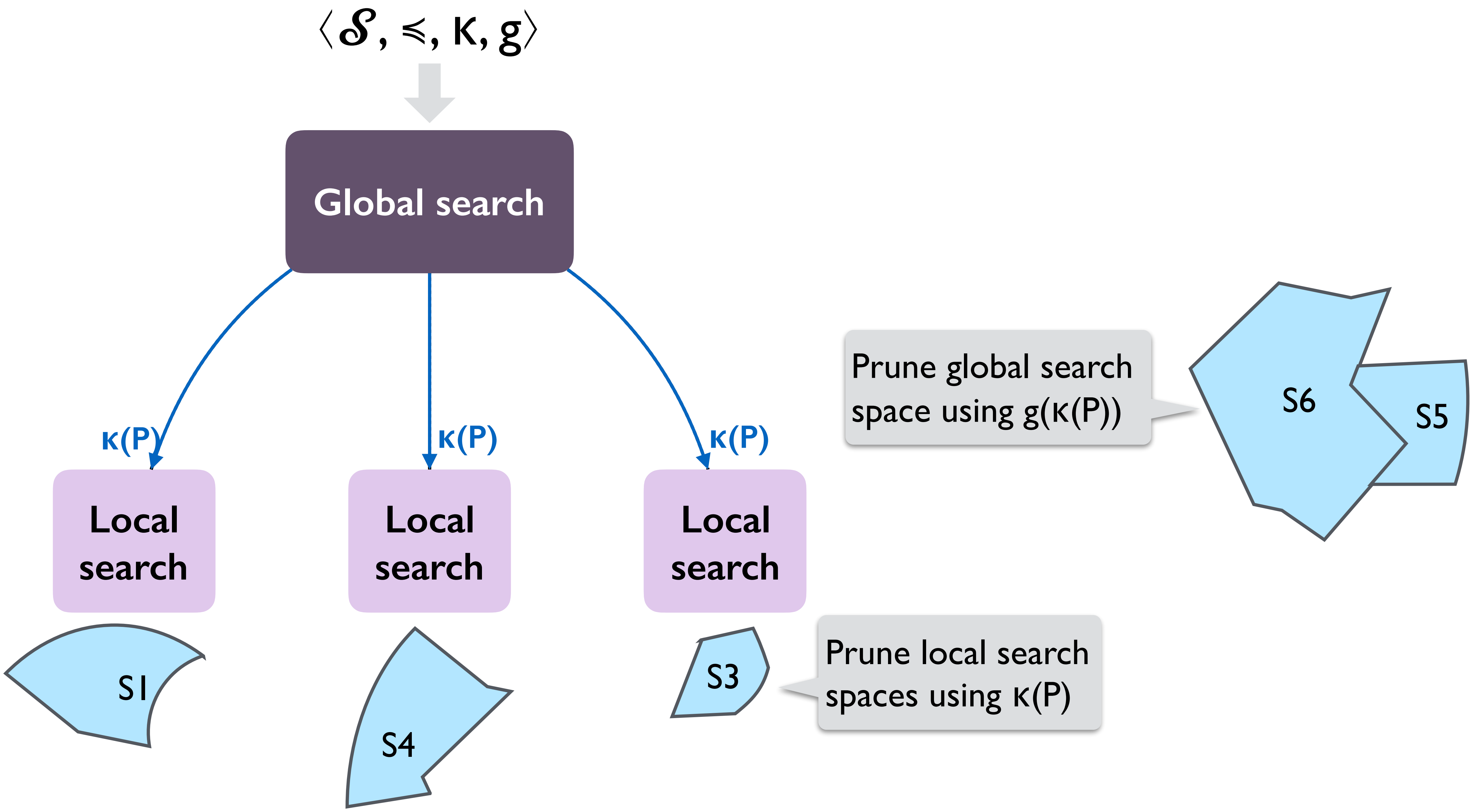
Solving with two cooperative searches



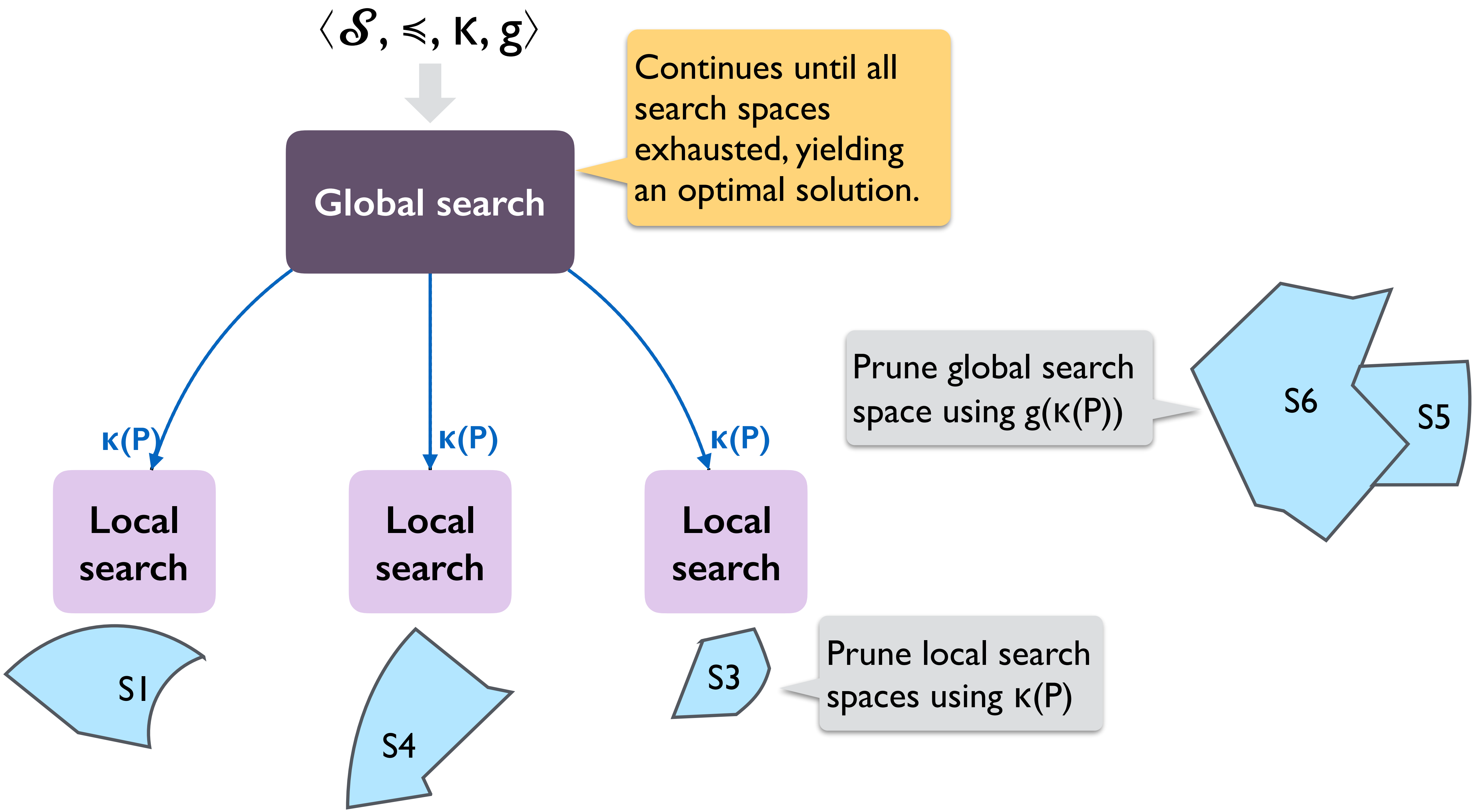
Solving with two cooperative searches



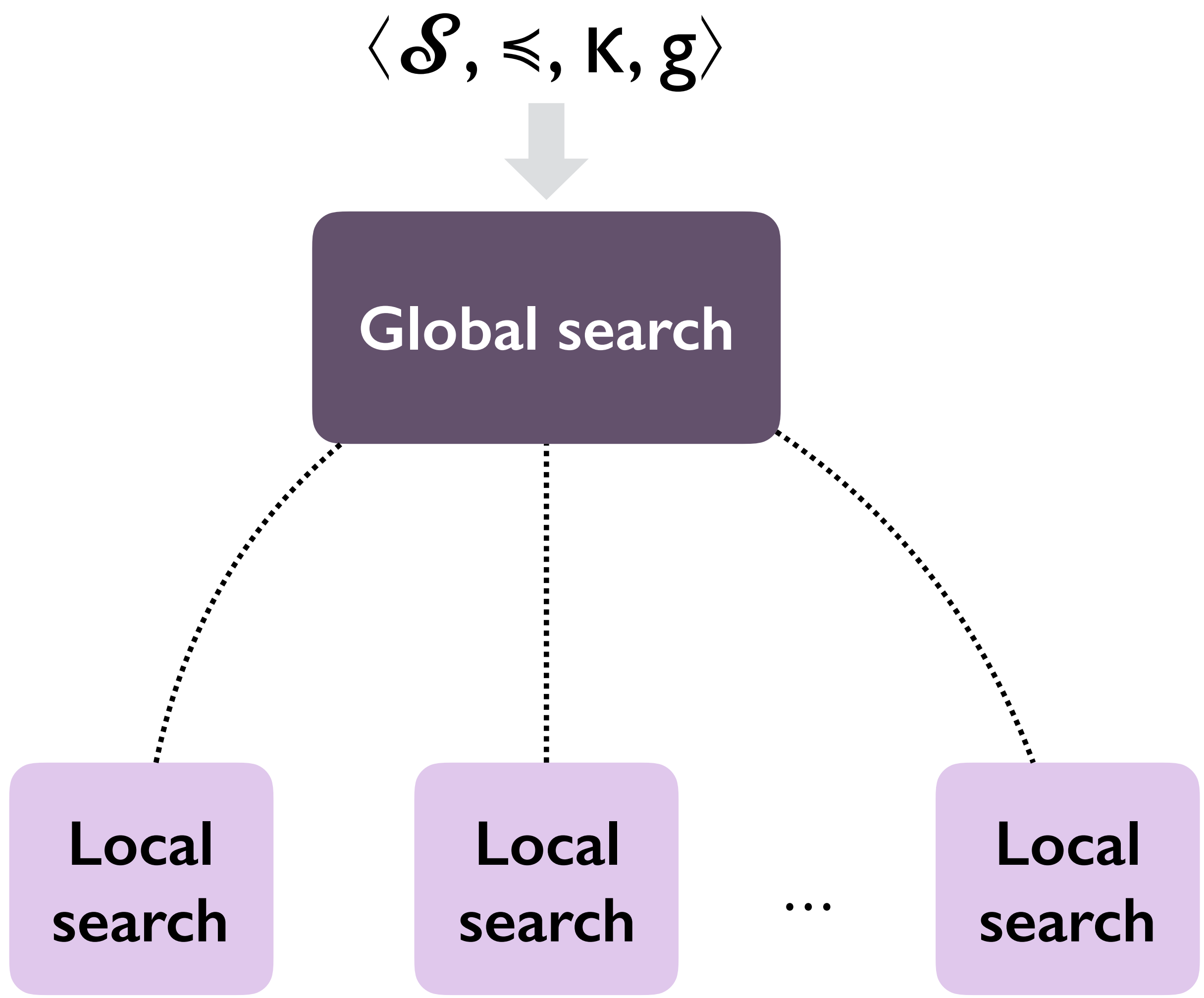
Solving with two cooperative searches



Solving with two cooperative searches

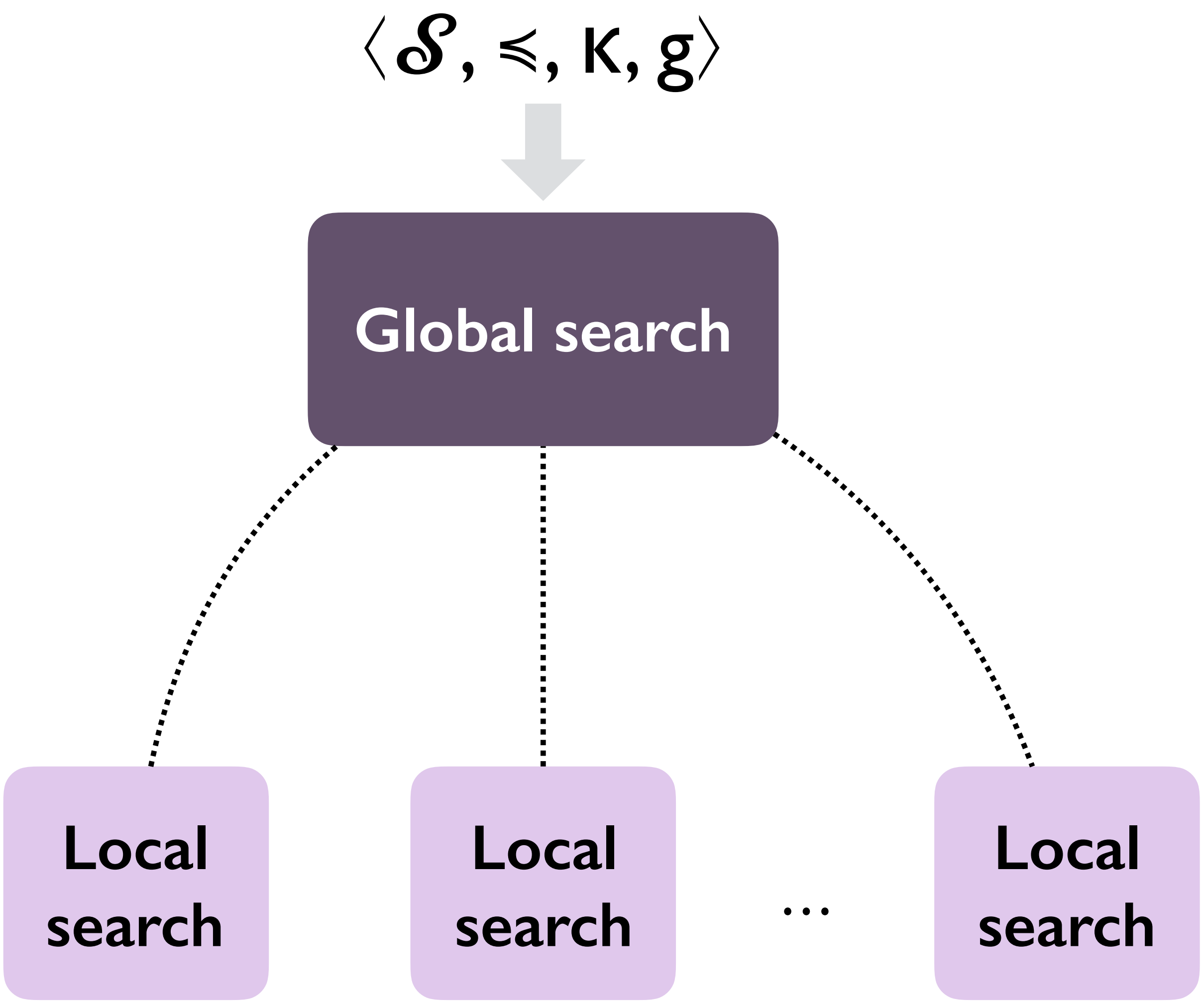


Synapse implementation



Implemented in Rosette, a solver-aided extension of Racket

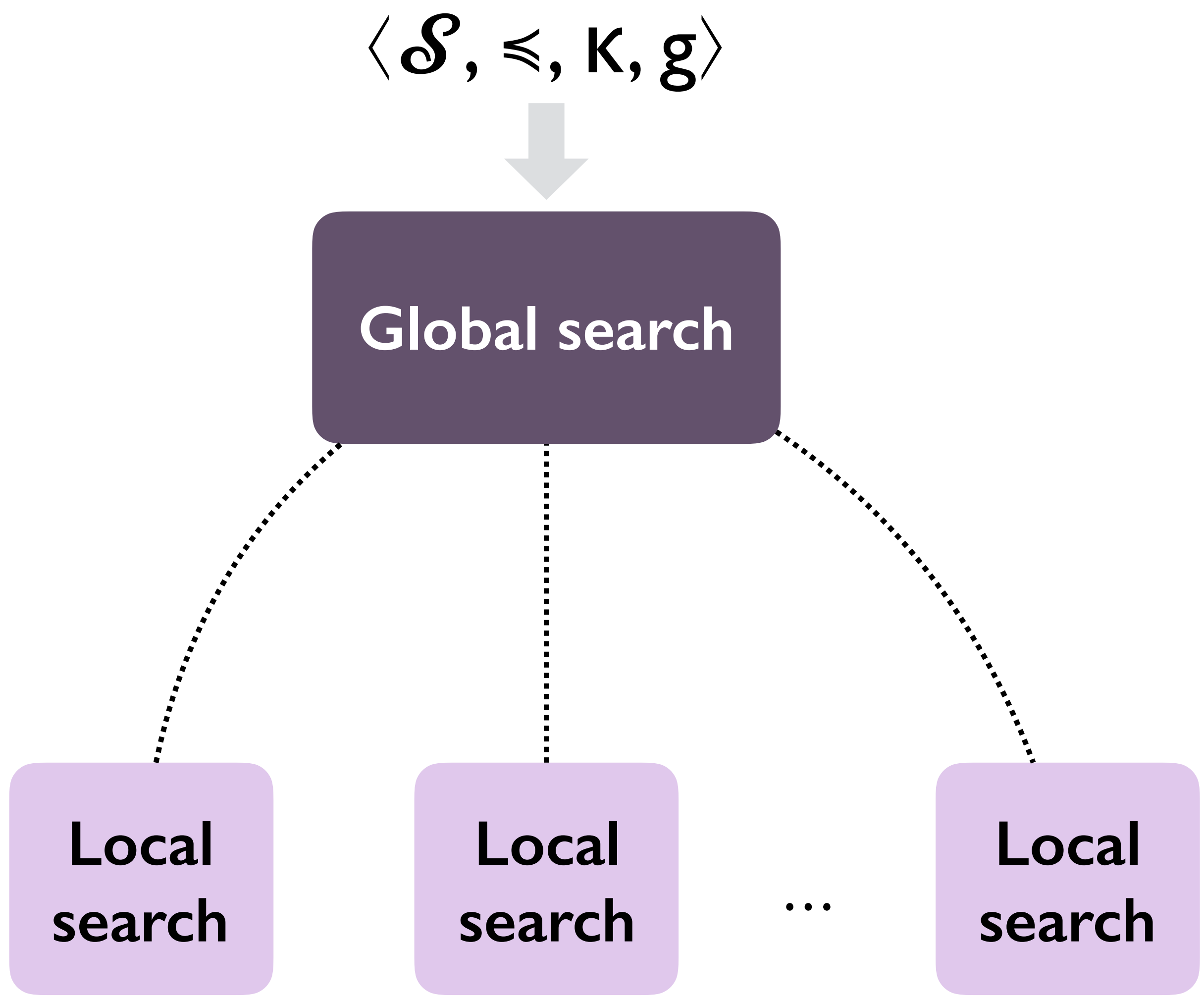
Synapse implementation



Implemented in Rosette, a solver-aided extension of Racket

Local CEGIS searches can share counterexamples

Synapse implementation



Implemented in Rosette, a solver-aided extension of Racket

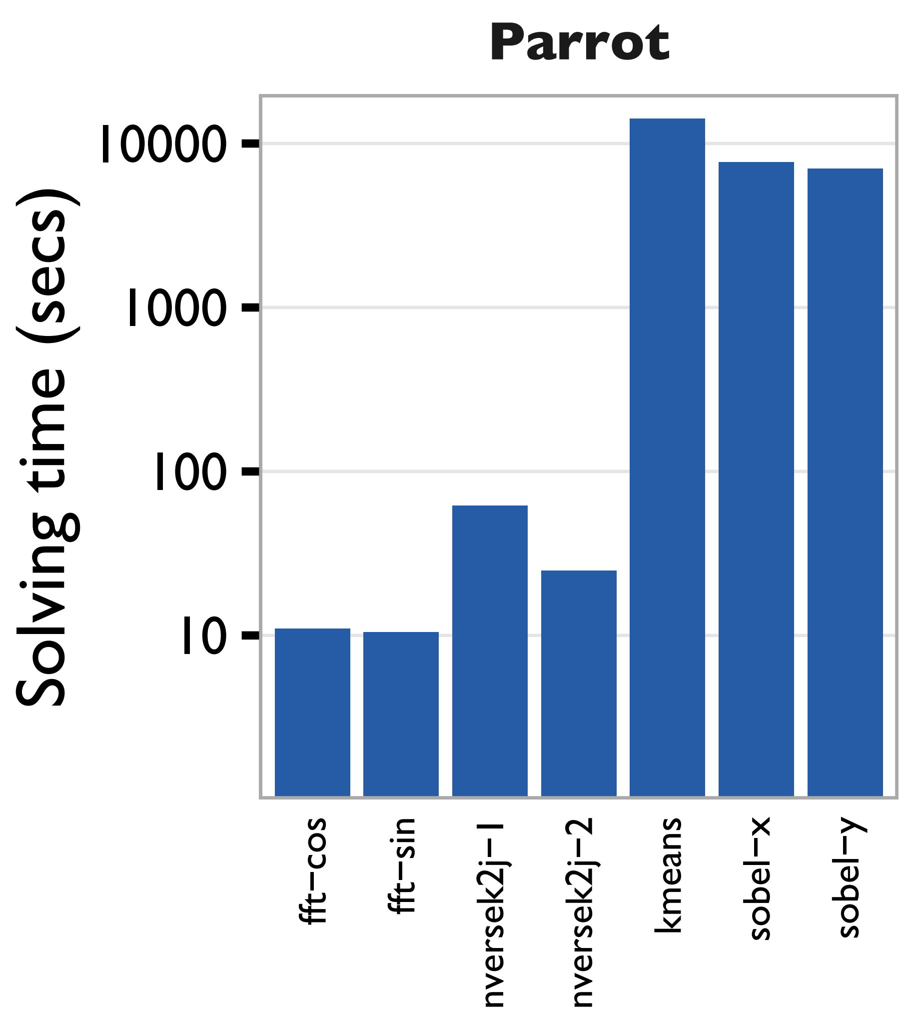
Local CEGIS searches can share counterexamples

Local searches can time out, which weakens optimality

Synapse solves previously-intractable problems

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

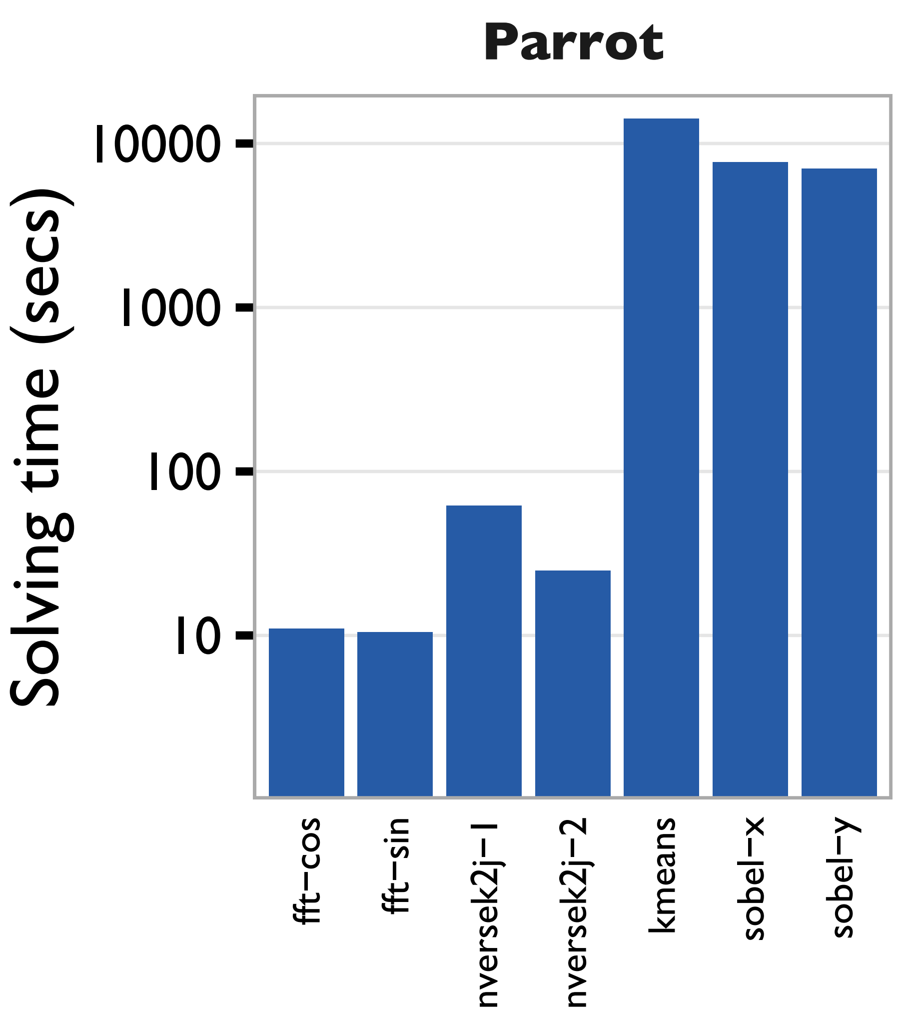
Find the most efficient approximate program within an error bound



Synapse solves previously-intractable problems

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

Find the most efficient approximate program within an error bound

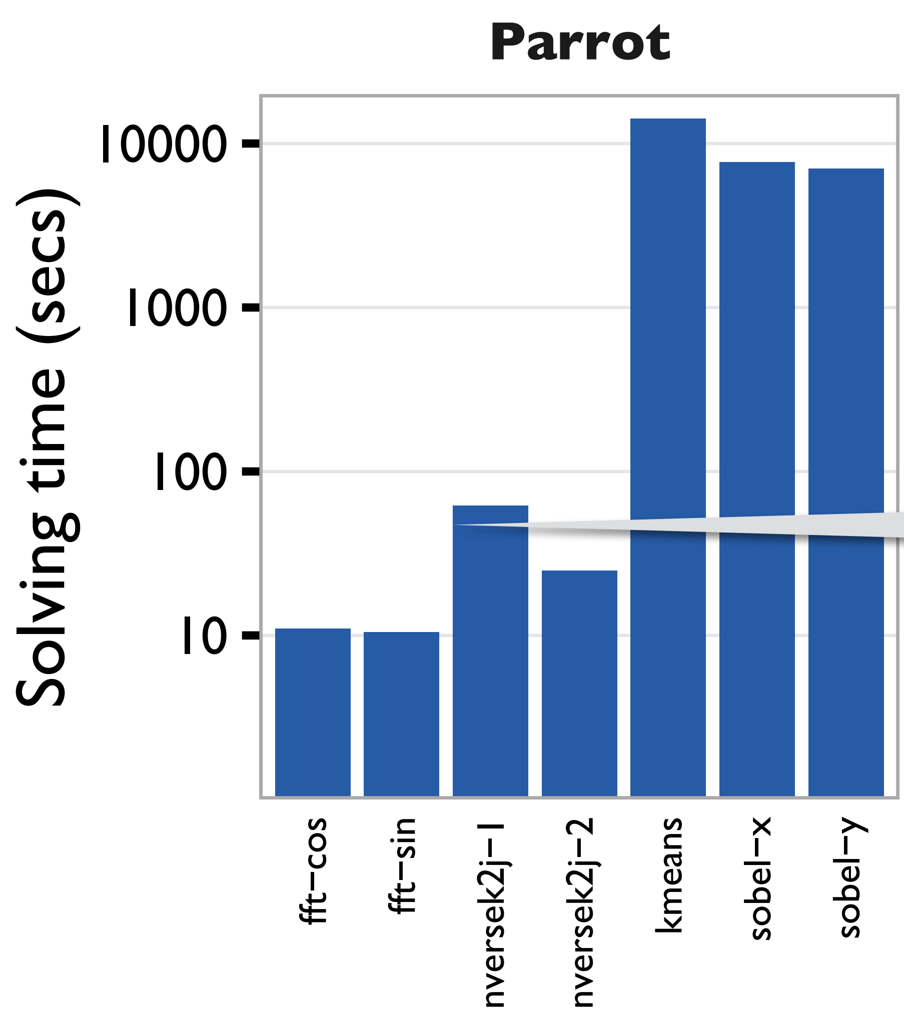


All intractable to Sketch and Stoke

Synapse solves previously-intractable problems

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

Find the most efficient approximate program within an error bound



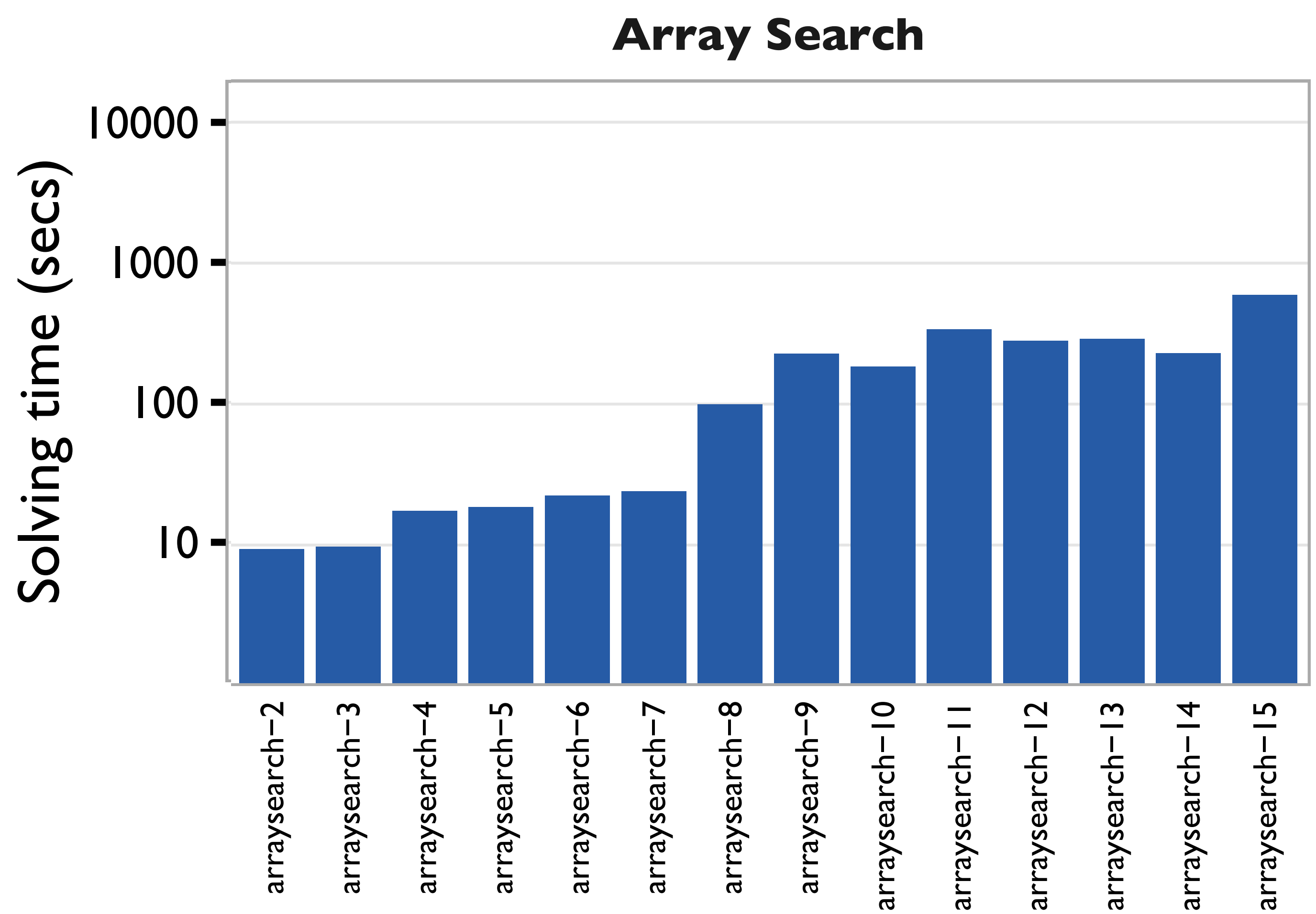
All intractable to Sketch and Stoke

```
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]

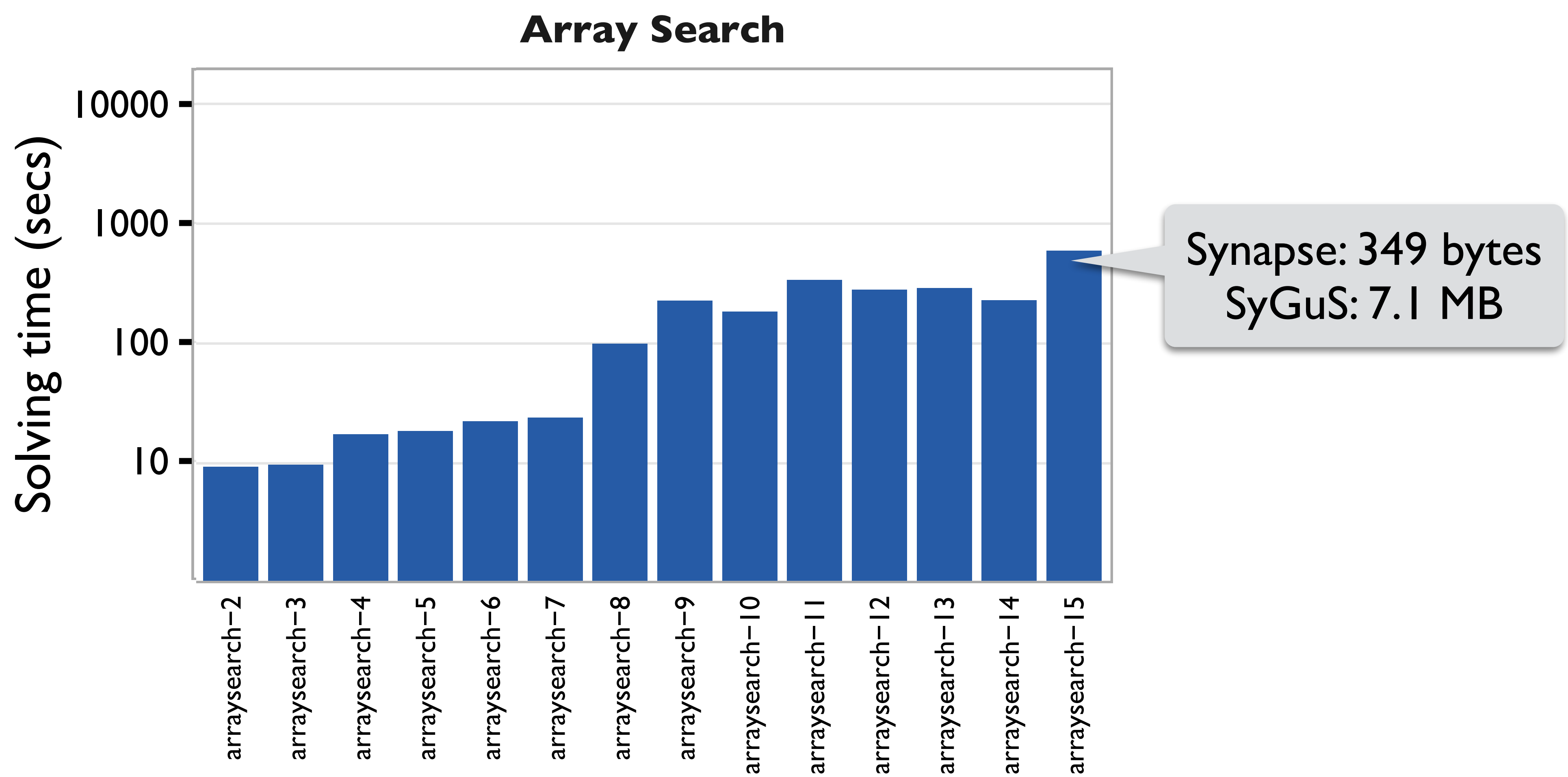
arraysearch-*n*: find program that searches lists of length *n*



Synapse solves standard benchmarks optimally

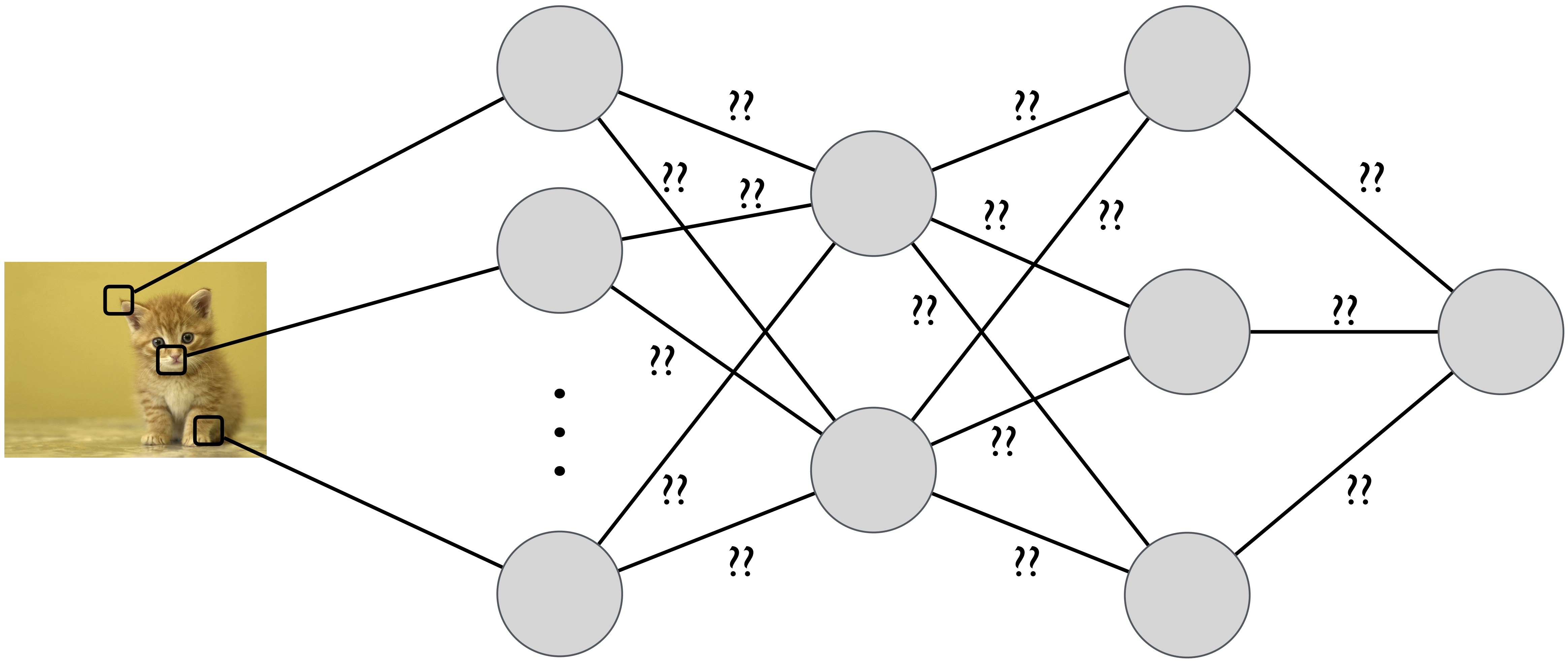
Array Search benchmarks from the syntax-guided synthesis (SyGuS) competition [Alur et al., 2015]

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

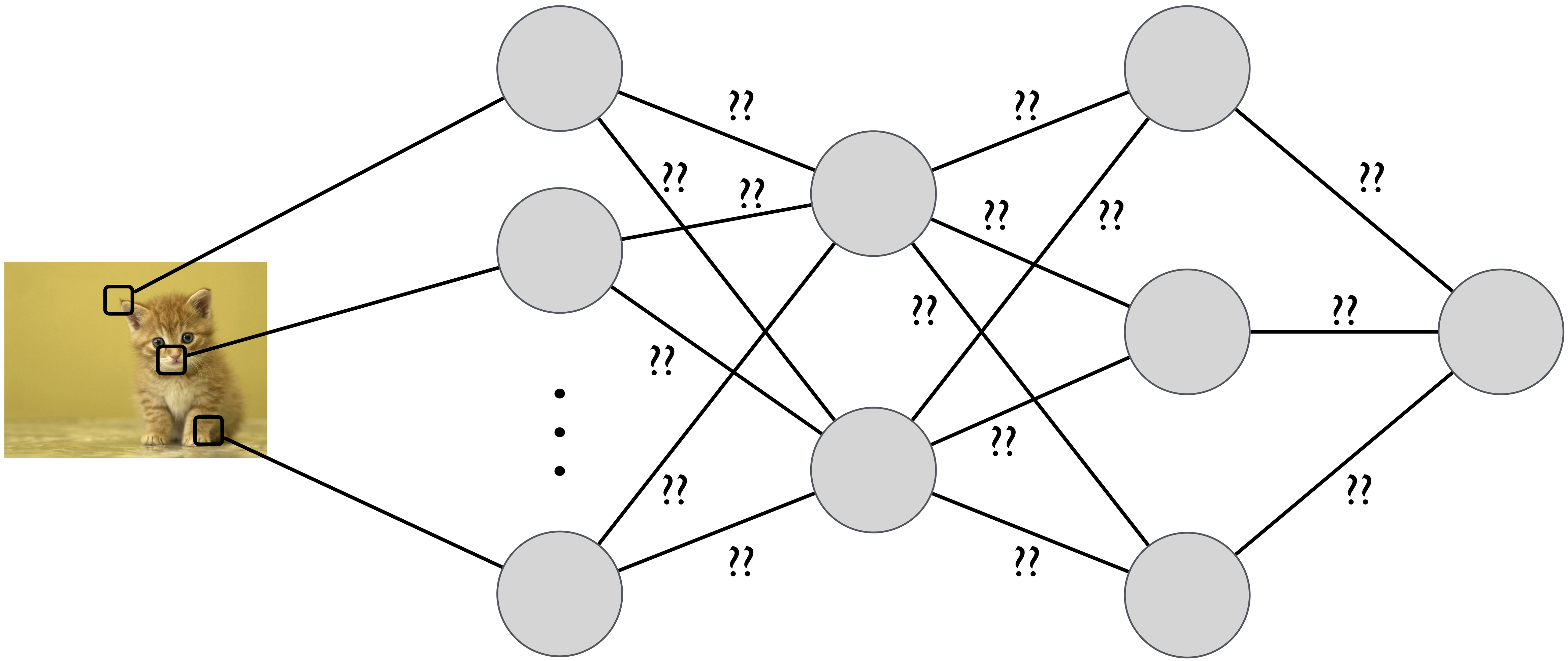


Is this a cat?

Synapse reasons about complex costs



Synapse reasons about complex costs



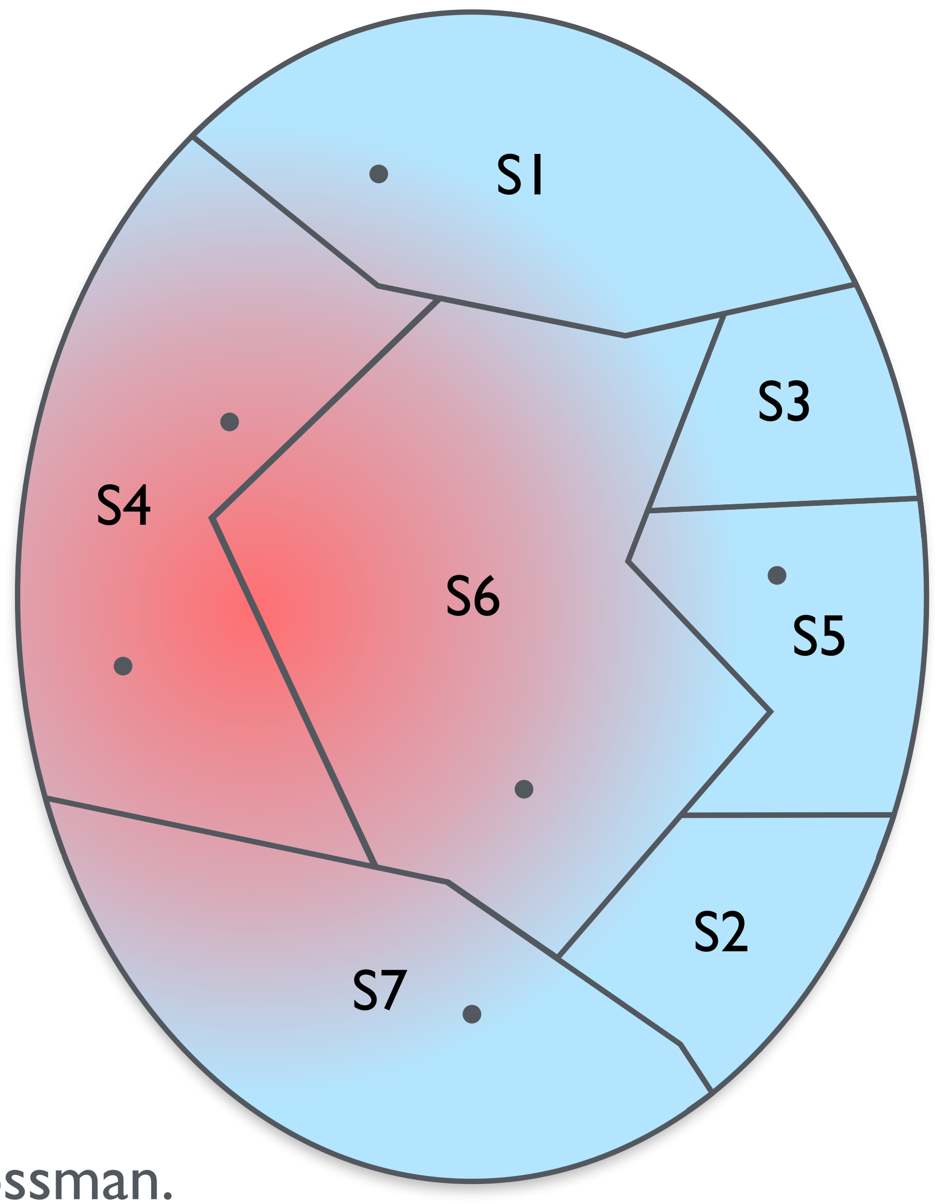
$$\kappa(P) = \sum_i |P(x_i) - y_i|$$

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

Metasketches express structure and strategy

<u>A metasketch contains:</u>

1. structured candidate space ($\mathcal{S}$, $\preccurlyeq$)
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?

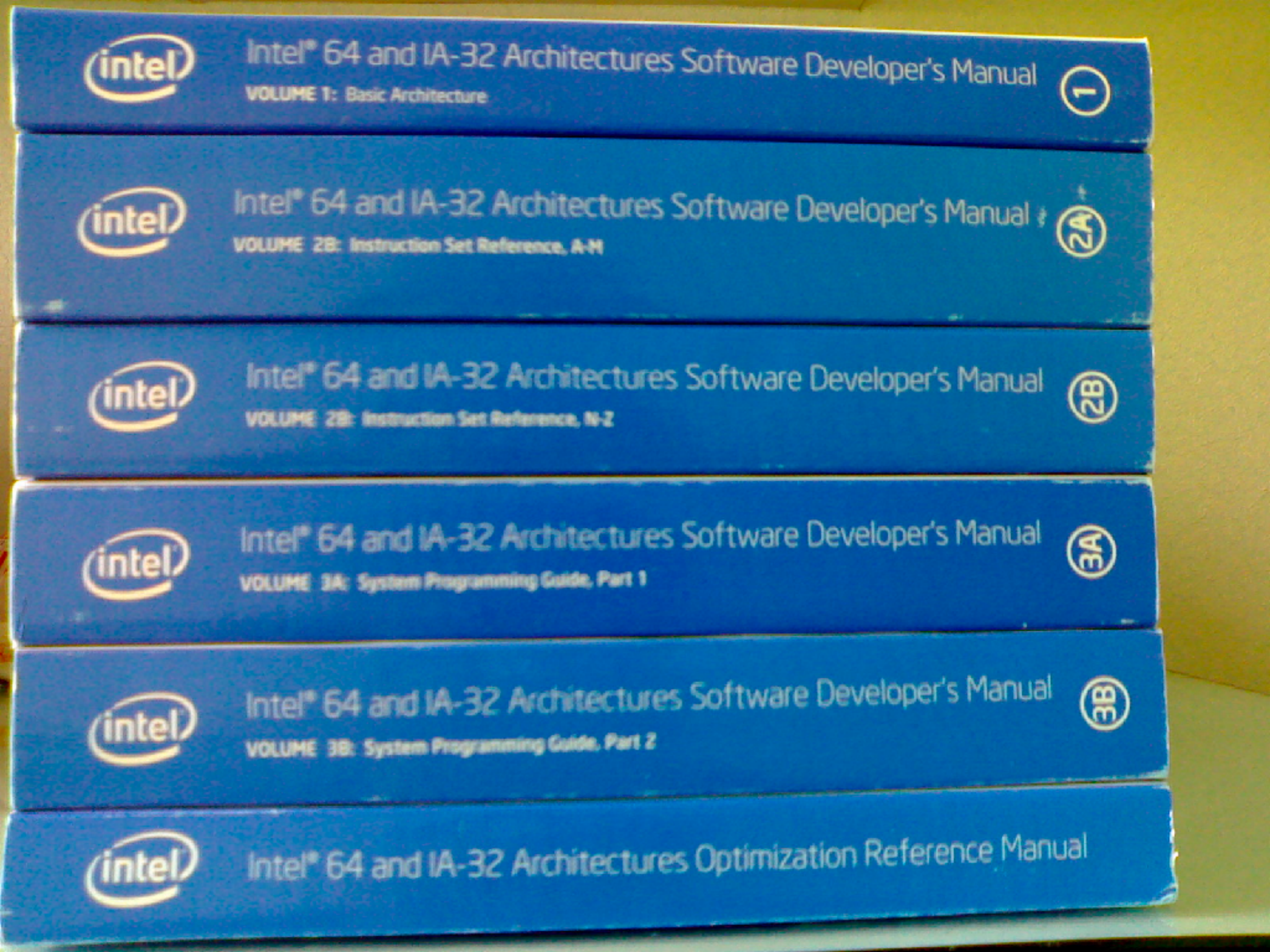
A memory model defines the reordering behaviors of a multiprocessor

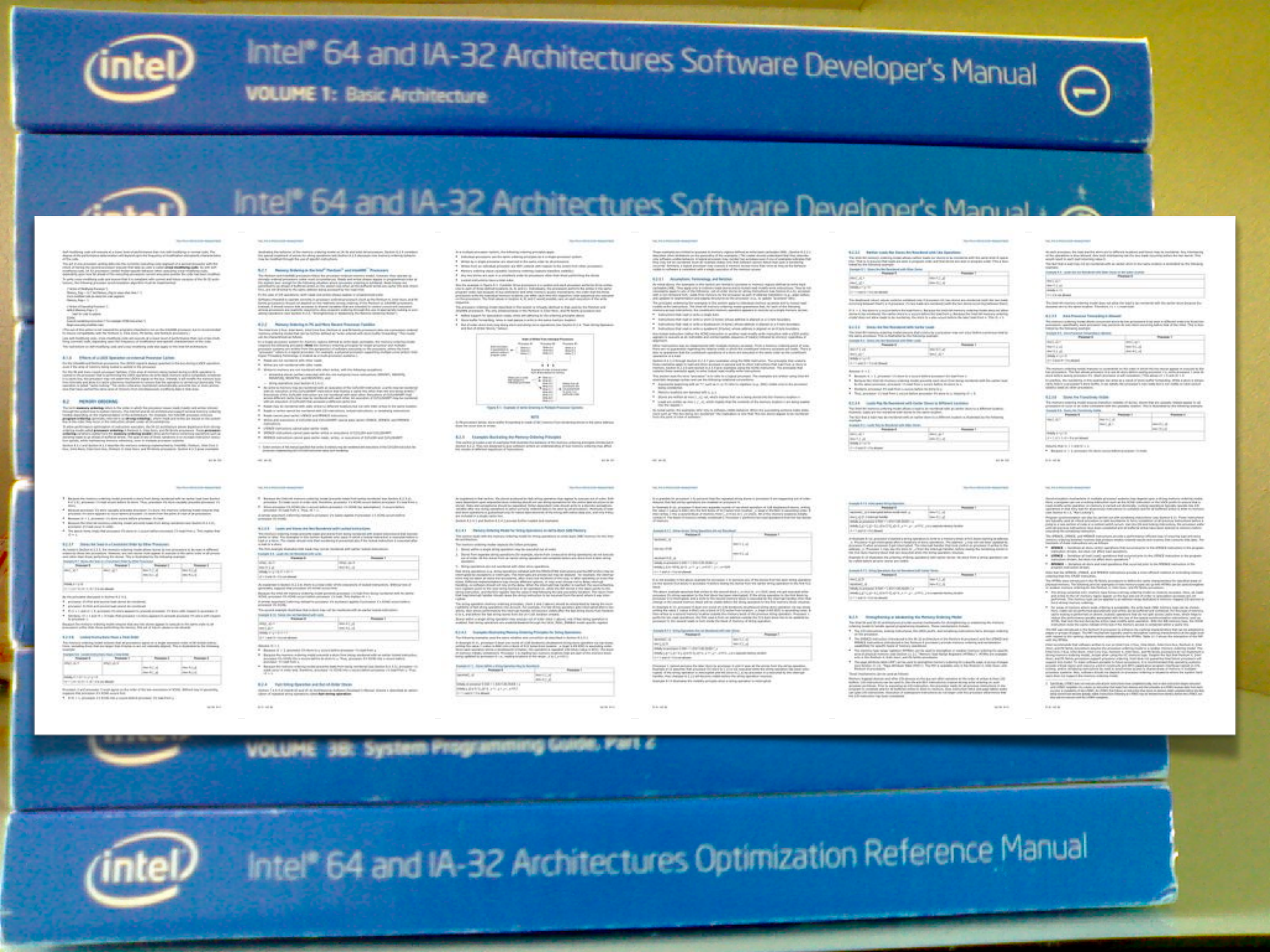
Initially X = Y = 0

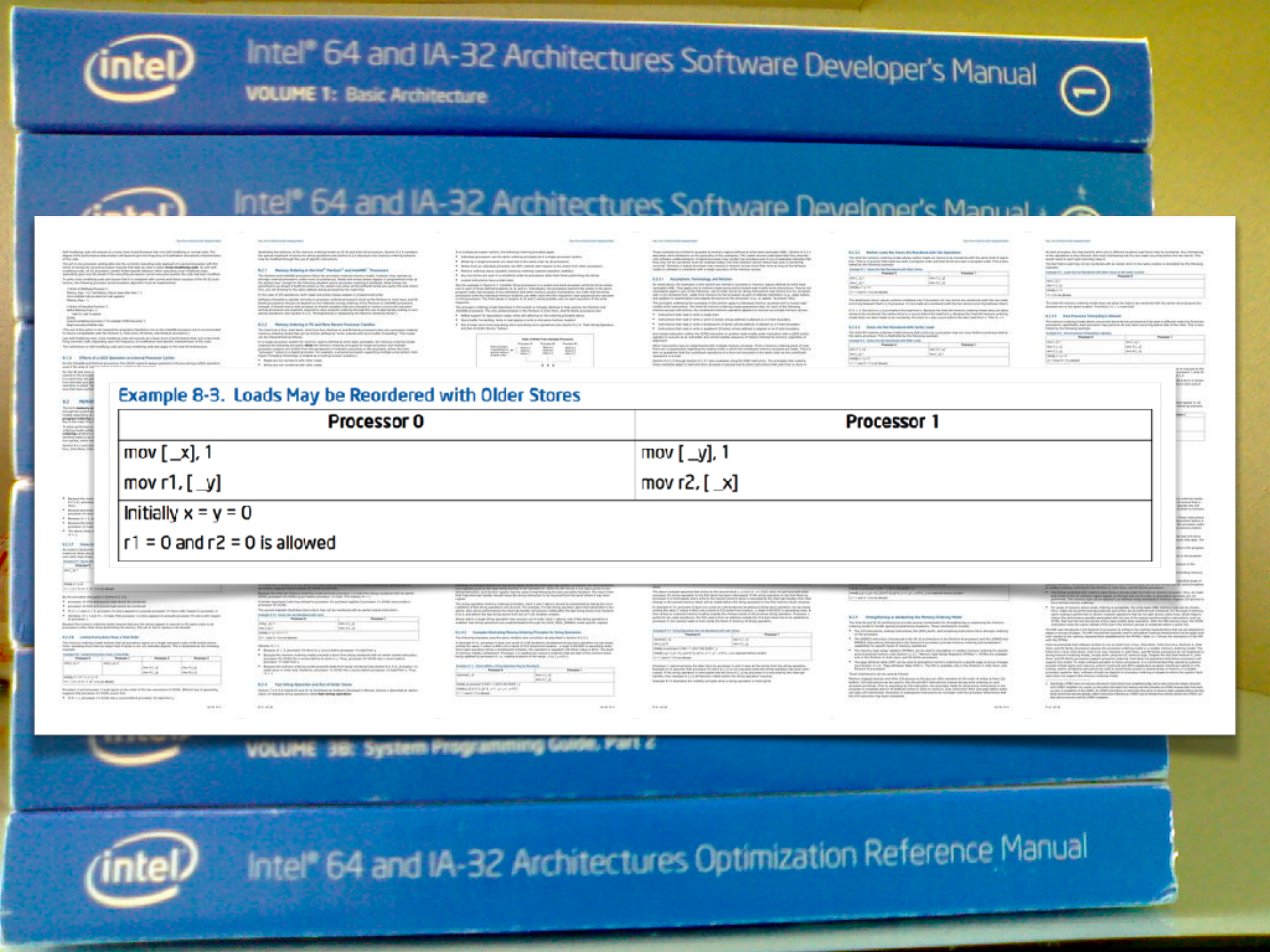
Thread 1	Thread 2

```
Y = 1
print X
X = 1
print Y
```

Can this program print two zeroes?


intel
Intel® 64 and IA-32 Architectures Software Developer's Manual
VOLUME 1: Basic Architecture
1
intel
Intel® 64 and IA-32 Architectures Software Developer's Manual
VOLUME 2B: Instruction Set Reference, A-M
2A
intel
Intel® 64 and IA-32 Architectures Software Developer's Manual
VOLUME 2B: Instruction Set Reference, N-Z
2B
intel
Intel® 64 and IA-32 Architectures Software Developer's Manual
VOLUME 3A: System Programming Guide, Part 1
3A
intel
Intel® 64 and IA-32 Architectures Software Developer's Manual
VOLUME 3B: System Programming Guide, Part 2
3B
intel
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
VOLUME 3B: System Programming Guide, Part 2
Intel® 64 and IA-32 Architectures Optimization Reference Manual



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

Processor 0	Processor 1
mov [_x], 1 mov r1, [_y]	mov [_y], 1 mov r2, [_x]
Initially x = y = 0 r1 = 0 and r2 = 0 is allowed	

DOI:10.1145/1785414.1785443

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,[21] 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*.[6,7,17,19]

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 EAX 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

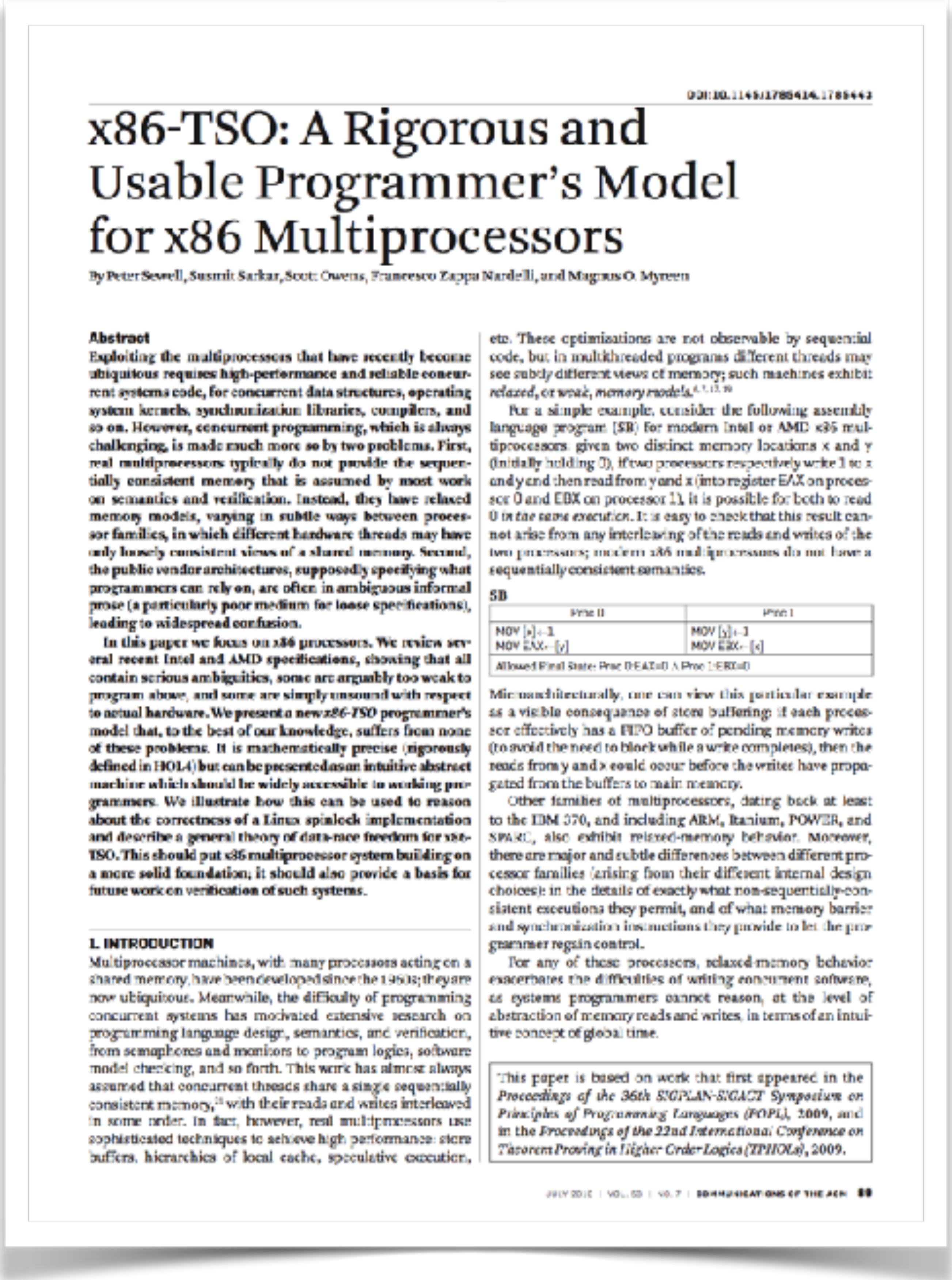
Proc 0	Proc 1
MOV [x]←1 MOV EAX←[y]	MOV [y]←1 MOV EBX←[x]
Allowed Final State: Proc 0:EAX=0 ∧ Proc 1:EBX=0	

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.



DOI:10.1145/1785414.1785443

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,[21] 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*.[6,7,17,19]

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 EAX 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 EAX←[y]	MOV [y]←1 MOV EBX←[x]
Allowed Final State: Proc 0:EAX=0 ∧ Proc 1:EBX=0	

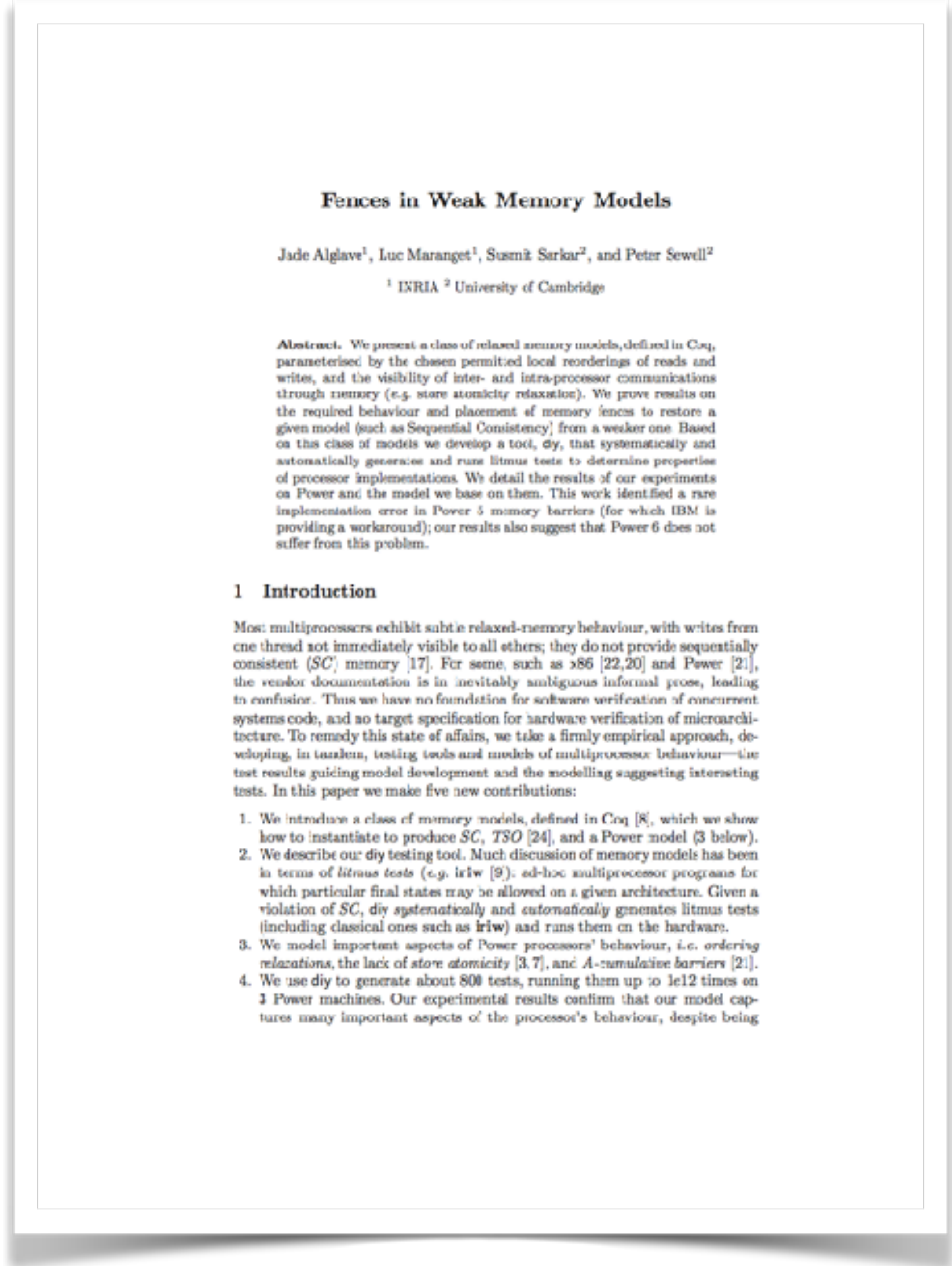
Microarchitecturally, one can view this particular example as a visible consequence of store buffering: if each processor effectively has a FIFO buffer of pending memory writes (to avoid the need to block while a write completes), then the reads from y and x could occur before the writes have propagated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

JULY 2010 | VOL. 53 | NO. 7 | COMMUNICATIONS OF THE ACM 89



Fences in Weak Memory Models

Jade Alglave[1], Luc Maranget[1], Susmit Sarkar[2], and Peter Sewell[2]

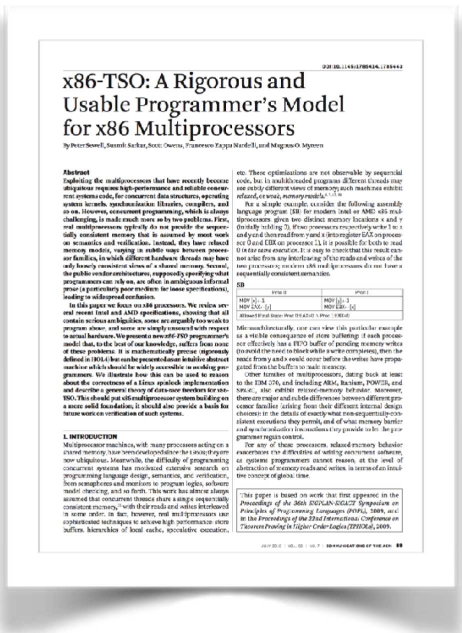
[1] INRIA [2] University of Cambridge

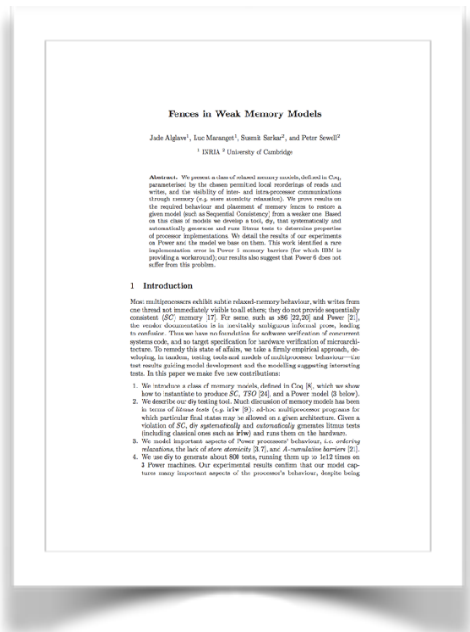
Abstract. We present a class of relaxed memory models, defined in Coq, parameterised by the chosen permitted local reorderings of reads and writes, and the visibility of inter- and intra-processor communications through memory (e.g. store atomicity relaxation). We prove results on the required behaviour 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.

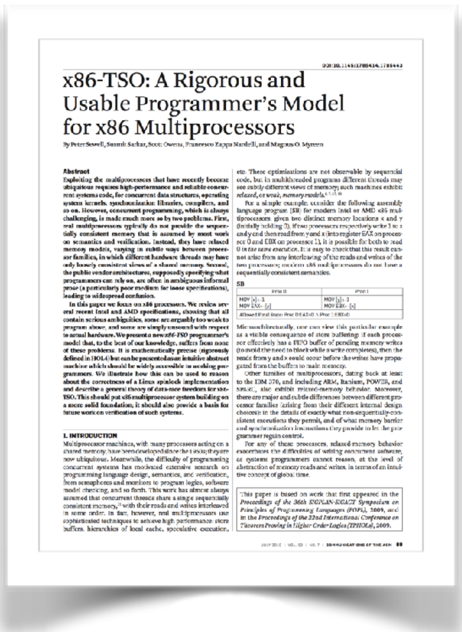
1 Introduction

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. iriw [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 iriw) 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

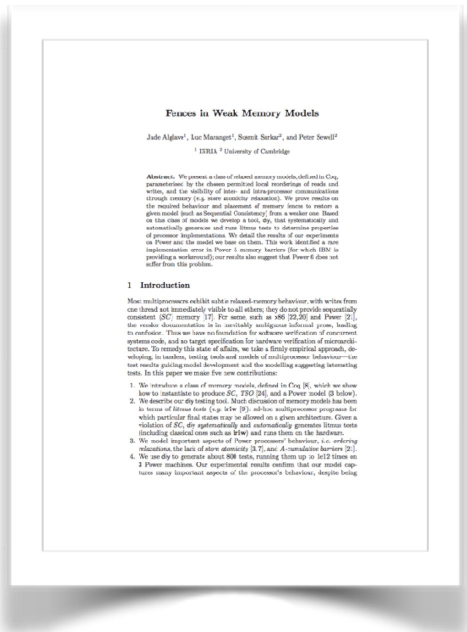

x86-TSO: A Rigorous and
Usable Programmer's Model
for x86 Multiprocessors
By Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen
Abstract
1. INTRODUCTION


Fences in Weak Memory Models
Jade Alglave[1], Luc Maranget[1], Susmit Sarkar[2], and Peter Sewell[2]
[1] INRIA [2] University of Cambridge
Abstract.
1 Introduction



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



Fences in Weak Memory Models

Jade Alglave[1], Luc Maranget[1], Susmit Sarkar[2], and Peter Sewell[2]

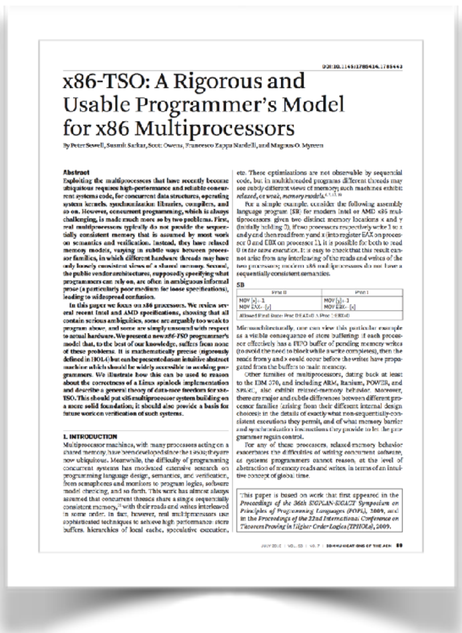
[1] INRIA [2] University of Cambridge

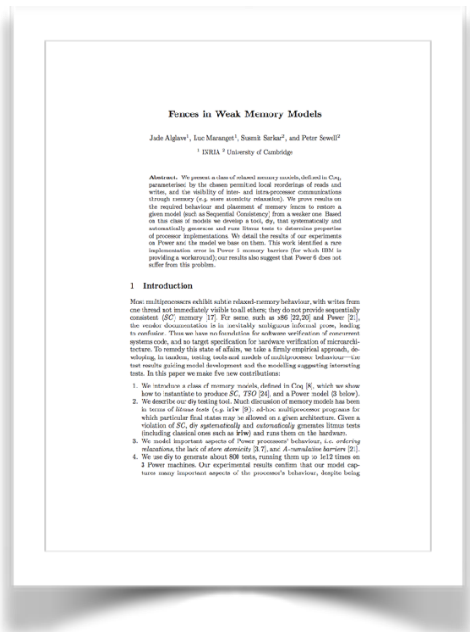
ARM

RISC-V

Trippel et al.
ASPLOS'17

…





ARM

RISC-V

Trippel et al.
ASPLOS'17

...

GPUs

Alglave et al.
ASPLOS'15

Accelerators

...

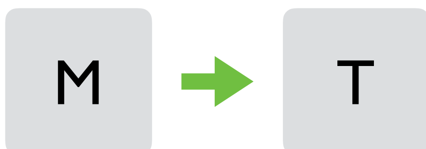
MemSynth:
automated reasoning for memory models

MemSynth: automated reasoning for memory models

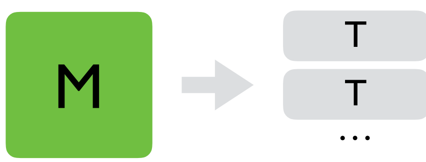
M → T

Verification
Check that model M allows litmus test T

MemSynth:
automated reasoning for memory models

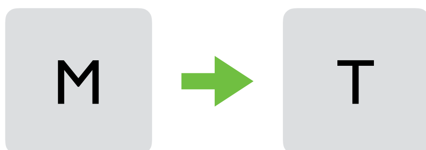


Verification
Check that model M allows litmus test T

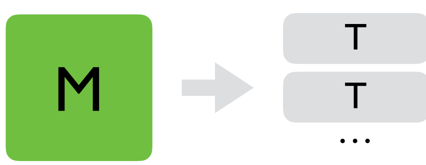


Synthesis
Complete a *framework sketch* M to be correct for all tests T

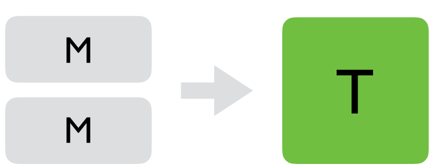
MemSynth:
automated reasoning for memory models



Verification
Check that model M allows litmus test T

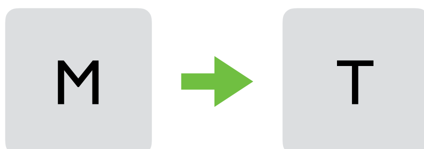


Synthesis
Complete a *framework sketch* M to be correct for all tests T



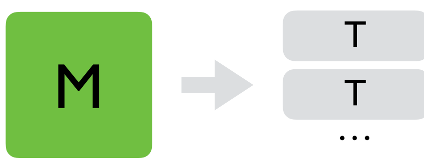
Equivalence
Synthesize a test T on which two models differ

MemSynth:
automated reasoning for memory models



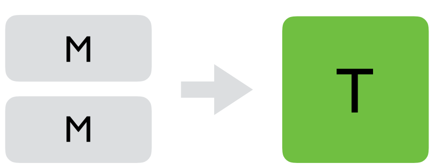
Verification

Check that model M allows litmus test T



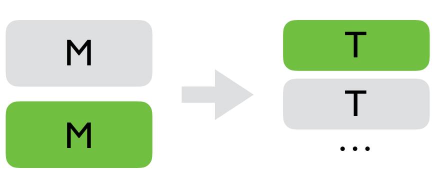
Synthesis

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



Equivalence

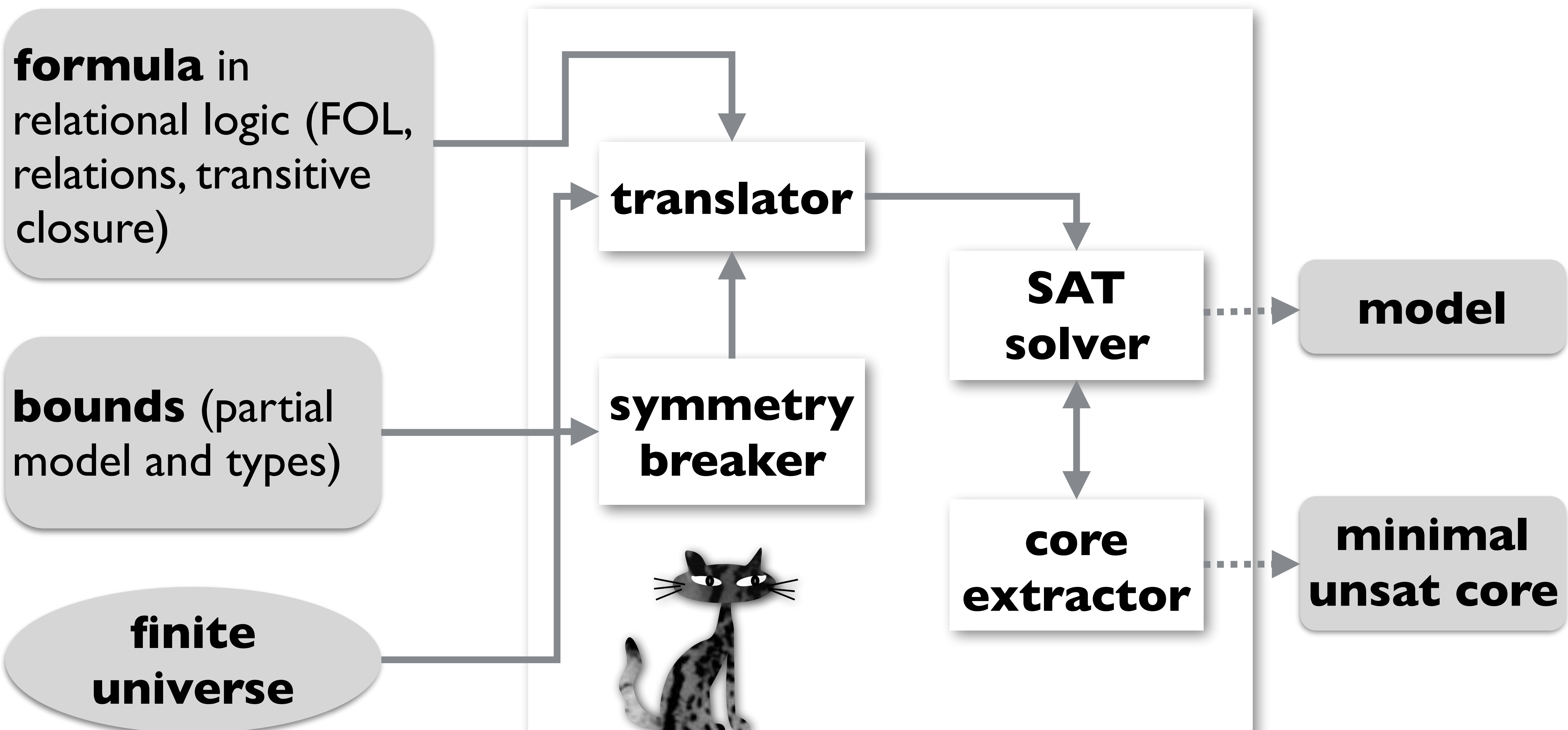
Synthesize a test T on which two models differ



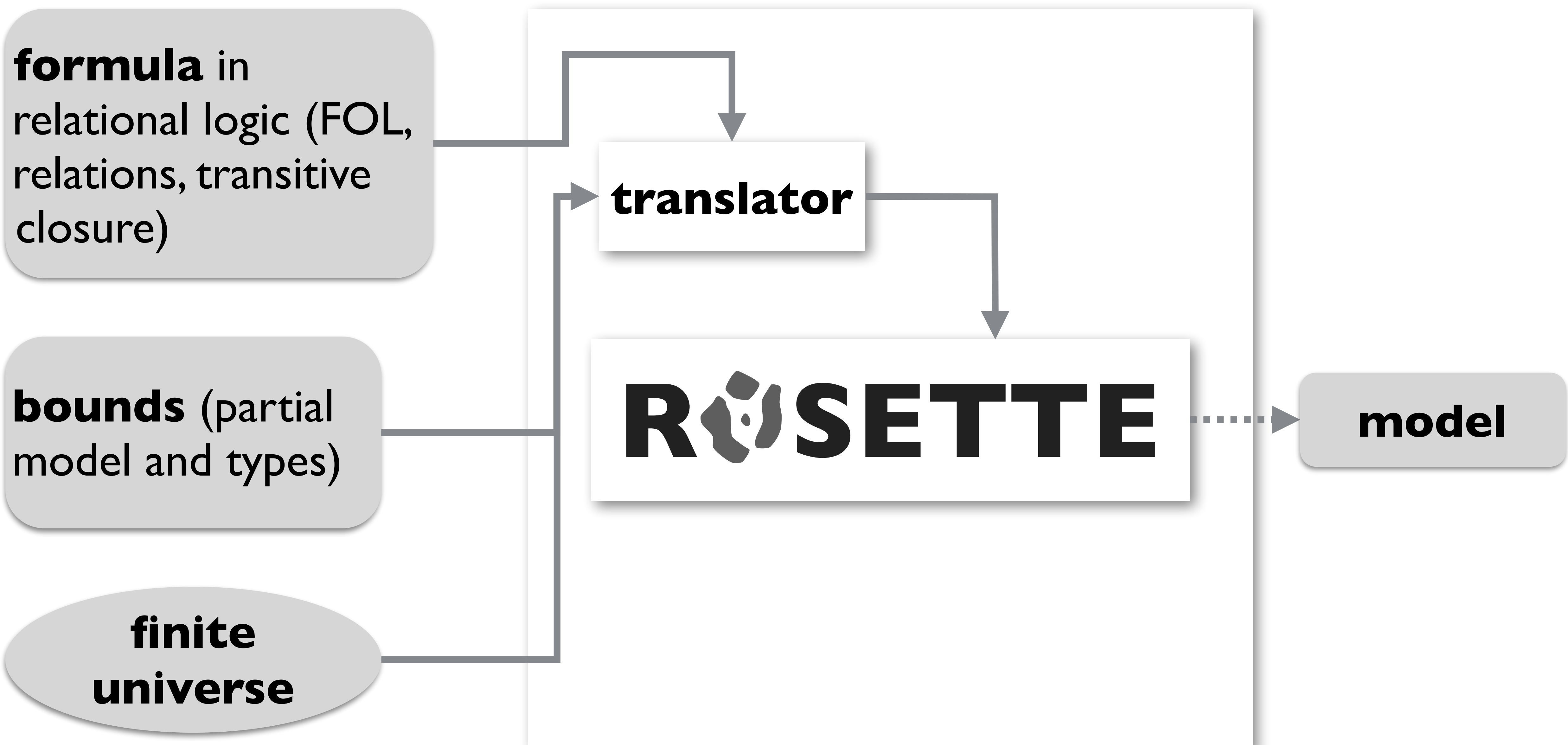
Ambiguity

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

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

Initially X = Y = 0

Thread 1	Thread 2
`X = 1`	`Y = 1`
`print Y`	`print X`

Can this program print two zeroes?

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

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`

x86 allows store-load reordering

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`

x86 allows store-load reordering

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

Initially X = Y = 0

Thread 1		Thread 2
E_1 `X = 1`		E_3 `Y = 1`
E_2 `print Y`		E_4 `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 `print Y`	E_4 `print X`

Can this program print two zeroes?

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

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

$\{\langle \mathbf{E_1}, \mathbf{1}\rangle, \langle \mathbf{E_2}, \mathbf{1}\rangle, \ldots\} \subseteq \text{thd} \subseteq \{\langle \mathbf{E_1}, \mathbf{1}\rangle, \langle \mathbf{E_2}, \mathbf{1}\rangle, \ldots\}$

…

Memory models as relational logic

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

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

$$\{\langle \mathbf{E_1}, \mathbf{1} \rangle, \langle \mathbf{E_2}, \mathbf{1} \rangle, \ldots\} \subseteq \text{thd} \subseteq \{\langle \mathbf{E_1}, \mathbf{1} \rangle, \langle \mathbf{E_2}, \mathbf{1} \rangle, \ldots\}$$

…

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

Initially X = Y = 0

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

Can this program print two zeroes?

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

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

$\{\langle \mathbf{E_1}, \mathbf{1} \rangle, \langle \mathbf{E_2}, \mathbf{1} \rangle, \ldots\} \subseteq \text{thd} \subseteq \{\langle \mathbf{E_1}, \mathbf{1} \rangle, \langle \mathbf{E_2}, \mathbf{1} \rangle, \ldots\}$

…

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

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

…

Memory models as relational logic

Initially X = Y = 0

	Thread 1		Thread 2
E_1	`X = 1`	E_3	`Y = 1`
E_2	`print Y`	E_4	`print X`

Can this program print two zeroes?

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

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

$\{\langle \mathbf{E_1}, \mathbf{1} \rangle, \langle \mathbf{E_2}, \mathbf{1} \rangle, \ldots\} \subseteq \text{thd} \subseteq \{\langle \mathbf{E_1}, \mathbf{1} \rangle, \langle \mathbf{E_2}, \mathbf{1} \rangle, \ldots\}$

…

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

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

…

no ^hb ∩ iden

Memory models as relational logic

Initially X = Y = 0

Thread 1	Thread 2
E_1 `X = 1`	E_3 `Y = 1`
E_2 `print Y`	E_4 `print X`

Can this program print two zeroes?

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

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

$\{\langle \mathbf{E_1}, \mathbf{1}\rangle, \langle \mathbf{E_2}, \mathbf{1}\rangle, \ldots\} \subseteq \text{thd} \subseteq \{\langle \mathbf{E_1}, \mathbf{1}\rangle, \langle \mathbf{E_2}, \mathbf{1}\rangle, \ldots\}$

…

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

Sequential consistency

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

…

no ^hb ∩ iden

Memory models as relational logic

Initially X = Y = 0

Thread 1	**Thread 2**
E1 `X = 1`	**E3** `Y = 1`
E2 `print Y`	**E4** `print X`

Can this program print two zeroes?

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

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

$\{\langle \mathbf{E_1}, \mathbf{1}\rangle, \langle \mathbf{E_2}, \mathbf{1}\rangle, \ldots\} \subseteq \text{thd} \subseteq \{\langle \mathbf{E_1}, \mathbf{1}\rangle, \langle \mathbf{E_2}, \mathbf{1}\rangle, \ldots\}$

…

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

x86 (“total store order”)

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

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

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

…

no ^hb ∩ iden

Verification

M → T

Initially X = Y = 0

Thread 1	Thread 2
E1 `X = 1`	E3 `Y = 1`
E2 `print Y`	E4 `print X`

Can this program print two zeroes?

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

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

$\{\langle \mathbf{E_1}, \mathbf{1} \rangle, \langle \mathbf{E_2}, \mathbf{1} \rangle, \ldots\} \subseteq \text{thd} \subseteq \{\langle \mathbf{E_1}, \mathbf{1} \rangle, \langle \mathbf{E_2}, \mathbf{1} \rangle, \ldots\}$

…

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

$\forall\, e_i, e_j : \text{Event} \mid$
$\quad i < j \wedge e_i.\text{thd} = e_j.\text{thd} \wedge \neg(e_i \in \text{Write} \wedge e_j \in \text{Read})$
$\qquad \Rightarrow \langle e_i, e_j \rangle \in \text{hb}$

…

no ^hb ∩ iden

Synthesis

M → T T …

Initially X = Y = 0

	Thread 1		Thread 2
E_1	`X = 1`	E_3	`Y = 1`
E_2	`print Y`	E_4	`print X`

Can this program print two zeroes?

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

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

$\{\langle \mathbf{E_1}, \mathbf{1} \rangle, \langle \mathbf{E_2}, \mathbf{1} \rangle, \ldots\} \subseteq \text{thd} \subseteq \{\langle \mathbf{E_1}, \mathbf{1} \rangle, \langle \mathbf{E_2}, \mathbf{1} \rangle, \ldots\}$

…

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

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

…

no ^hb ∩ iden

Synthesis

M → T T …

Initially X = Y = 0

	Thread 1		Thread 2
E_1	`X = 1`	E_3	`Y = 1`
E_2	`print Y`	E_4	`print X`

Can this program print two zeroes?

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

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

$\{\langle \mathbf{E_1}, \mathbf{1}\rangle, \langle \mathbf{E_2}, \mathbf{1}\rangle, \ldots\} \subseteq \text{thd} \subseteq \{\langle \mathbf{E_1}, \mathbf{1}\rangle, \langle \mathbf{E_2}, \mathbf{1}\rangle, \ldots\}$

…

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

Holes to be filled by relational expressions

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

…

no ^hb ∩ iden

Synthesis

M → T T …

Initially X = Y = 0

Thread 1	Thread 2
E1 `X = 1`	E3 `Y = 1`
E2 `print Y`	E4 `print X`

Can this program print two zeroes?

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

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

$\{\langle \mathbf{E_1}, \mathbf{1}\rangle,\langle \mathbf{E_2}, \mathbf{1}\rangle,\ldots\} \subseteq \text{thd} \subseteq \{\langle \mathbf{E_1}, \mathbf{1}\rangle,\langle \mathbf{E_2}, \mathbf{1}\rangle,\ldots\}$

…

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

Holes to be filled by relational expressions

$\forall\ e_i, e_j : \text{Event}\ |$
$\quad i < j \wedge e_i.\text{thd} = e_j.\text{thd} \wedge \mathbf{??}$
$\quad\quad \Rightarrow \langle e_i, e_j\rangle \in \text{hb}$

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

…

no ^hb ∩ iden

Equivalence

M
M
→ T

Initially X = Y = 0

Thread 1		Thread 2	
E1	`X = 1`	E3	`Y = 1`
E2	`print Y`	E4	`print X`

Can this program print two zeroes?

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

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

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

…

$\{\} \subseteq \text{hb}_1 \subseteq \{\mathbf{E_1}, \mathbf{E_2}, \mathbf{E_3}, \mathbf{E_4}\} \times \{\mathbf{E_1}, \mathbf{E_2}, \mathbf{E_3}, \mathbf{E_4}\}$

$\{\} \subseteq \text{hb}_2 \subseteq \{\mathbf{E_1}, \mathbf{E_2}, \mathbf{E_3}, \mathbf{E_4}\} \times \{\mathbf{E_1}, \mathbf{E_2}, \mathbf{E_3}, \mathbf{E_4}\}$

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

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

$\qquad \Rightarrow \langle e_i, e_j \rangle \in \text{hb}_1$

…

no $\hat{}\text{hb}_1 \cap$ iden

Equivalence

M

M

T

Solve for a litmus test that two models disagree about

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

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

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

…

$\{\} \subseteq \text{hb}_1 \subseteq \{\mathbf{E_1}, \mathbf{E_2}, \mathbf{E_3}, \mathbf{E_4}\} \times \{\mathbf{E_1}, \mathbf{E_2}, \mathbf{E_3}, \mathbf{E_4}\}$

$\{\} \subseteq \text{hb}_2 \subseteq \{\mathbf{E_1}, \mathbf{E_2}, \mathbf{E_3}, \mathbf{E_4}\} \times \{\mathbf{E_1}, \mathbf{E_2}, \mathbf{E_3}, \mathbf{E_4}\}$

Initially X = Y = 0

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

Can this program print two zeroes?

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

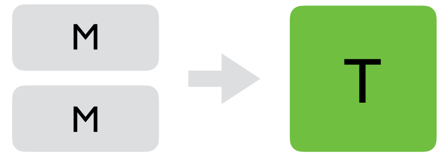
$\quad i < j \wedge e_i.\text{thd} = e_j.\text{thd} \wedge \neg(e_i \in \text{Write} \wedge e_j \in \text{Read})$

$\qquad \Rightarrow \langle e_i, e_j \rangle \in \text{hb}_1$

…

no $\hat{}\text{hb}_1 \cap \text{iden}$

Equivalence



Solve for a litmus test that two models disagree about

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

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

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

…

$\{\} \subseteq \text{hb}_1 \subseteq \{\mathbf{E_1}, \mathbf{E_2}, \mathbf{E_3}, \mathbf{E_4}\} \times \{\mathbf{E_1}, \mathbf{E_2}, \mathbf{E_3}, \mathbf{E_4}\}$

$\{\} \subseteq \text{hb}_2 \subseteq \{\mathbf{E_1}, \mathbf{E_2}, \mathbf{E_3}, \mathbf{E_4}\} \times \{\mathbf{E_1}, \mathbf{E_2}, \mathbf{E_3}, \mathbf{E_4}\}$

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

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

$\qquad \Rightarrow \langle e_i, e_j \rangle \in \text{hb}_1$

…

no $\hat{}\text{hb}_1 \cap \text{iden}$

Initially X = Y = 0

	Thread 1		Thread 2
$\mathbf{E_1}$	??	$\mathbf{E_3}$	??
$\mathbf{E_2}$	??	$\mathbf{E_4}$	??

Can this program print two zeroes?

Metasketches for memory models

Initially X = Y = 0

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

Can this program print two zeroes?

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

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

$\{\langle \mathbf{E_1}, \mathbf{1}\rangle, \langle \mathbf{E_2}, \mathbf{1}\rangle, \ldots\} \subseteq \text{thd} \subseteq \{\langle \mathbf{E_1}, \mathbf{1}\rangle, \langle \mathbf{E_2}, \mathbf{1}\rangle, \ldots\}$

…

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

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

$\forall\ e_i, e_j : \text{Event} \mid$
$\quad i < j \land e_i.\text{thd} = e_j.\text{thd} \land \neg(e_i \in \text{Write} \land e_j \in \text{Read})$
$\quad\quad \Rightarrow \langle e_i, e_j\rangle \in hb_1$

…

no $\hat{}hb_1 \cap \text{iden}$

Metasketches for memory models

Fix *some* parts of the litmus test, but not all

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

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

$\{\langle \mathbf{E_1}, \mathbf{1} \rangle, \langle \mathbf{E_2}, \mathbf{1} \rangle, \ldots\} \subseteq \text{thd} \subseteq \{\langle \mathbf{E_1}, \mathbf{1} \rangle, \langle \mathbf{E_2}, \mathbf{1} \rangle, \ldots\}$

…

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

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

Initially X = Y = 0			
	Thread 1		**Thread 2**
$\mathbf{E_1}$	?? = ??	$\mathbf{E_3}$	?? = ??
$\mathbf{E_2}$	??	$\mathbf{E_4}$	??

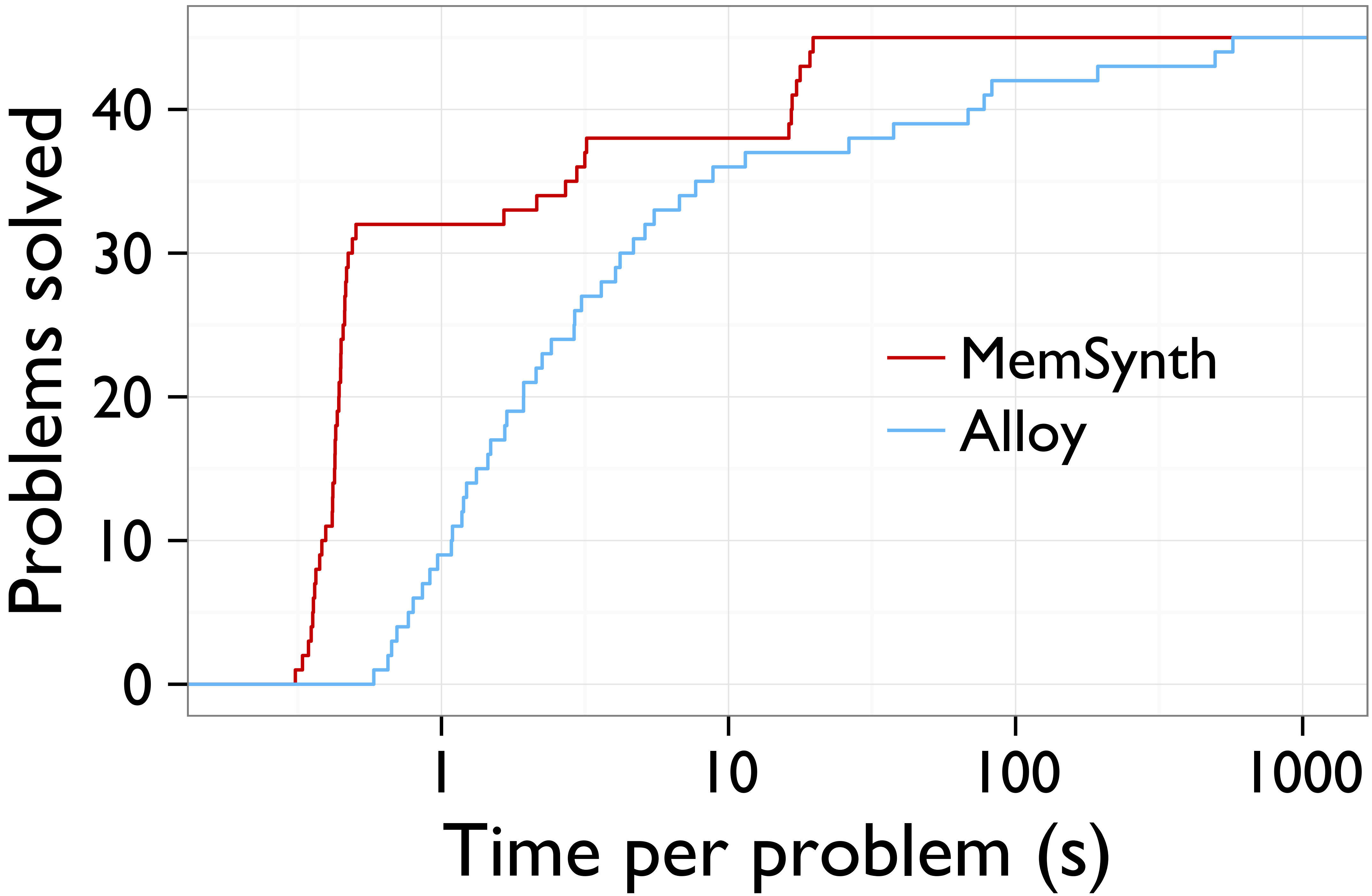
$\forall\ e_i, e_j : \text{Event} \mid$
$\quad i < j \wedge e_i.\text{thd} = e_j.\text{thd} \wedge \neg(e_i \in \text{Write} \wedge e_j \in \text{Read})$
$\qquad \Rightarrow \langle e_i, e_j \rangle \in hb_1$

Can this program print two zeroes?

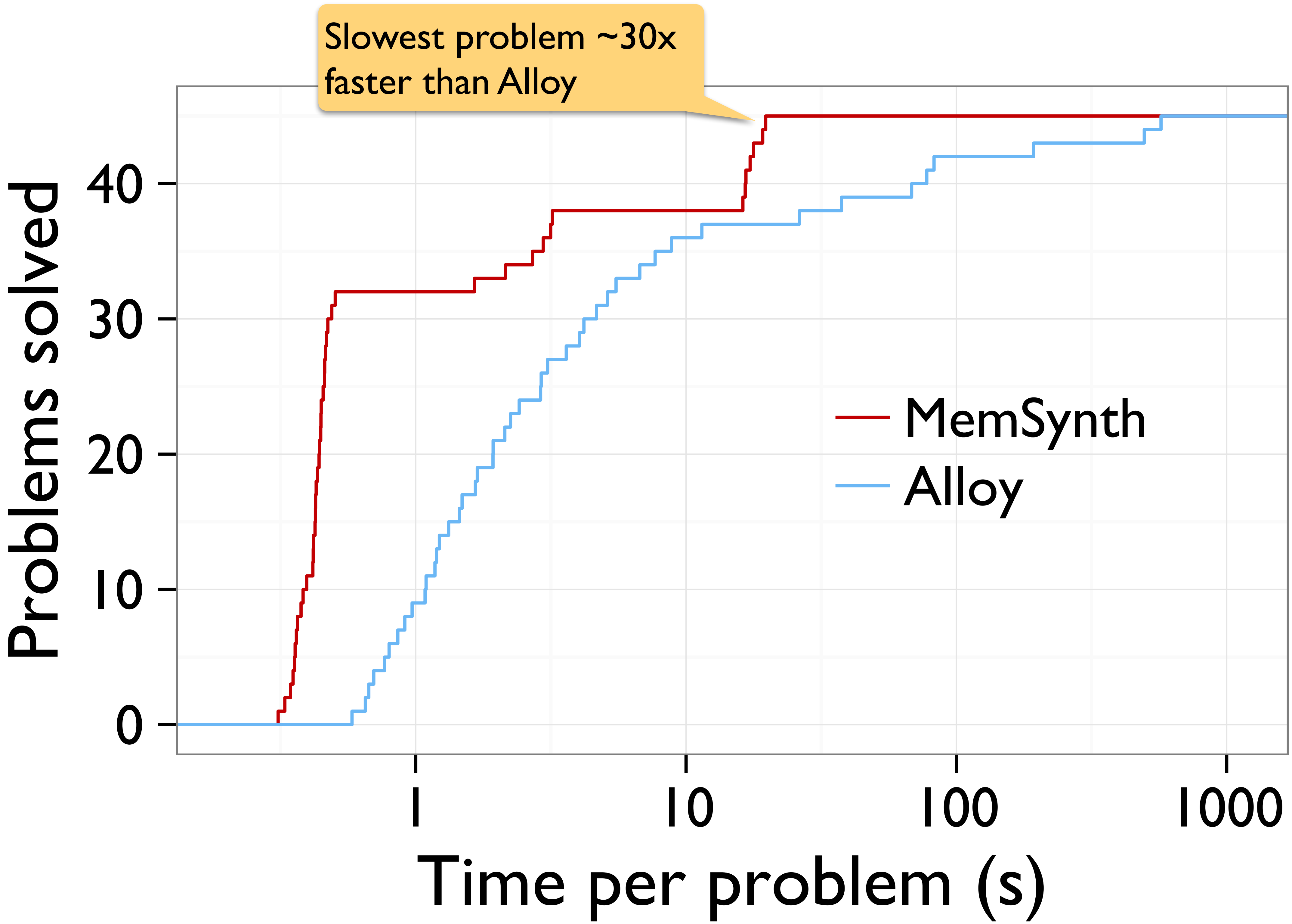
…

no $\hat{}hb_1 \cap$ iden

Metasketches for memory models

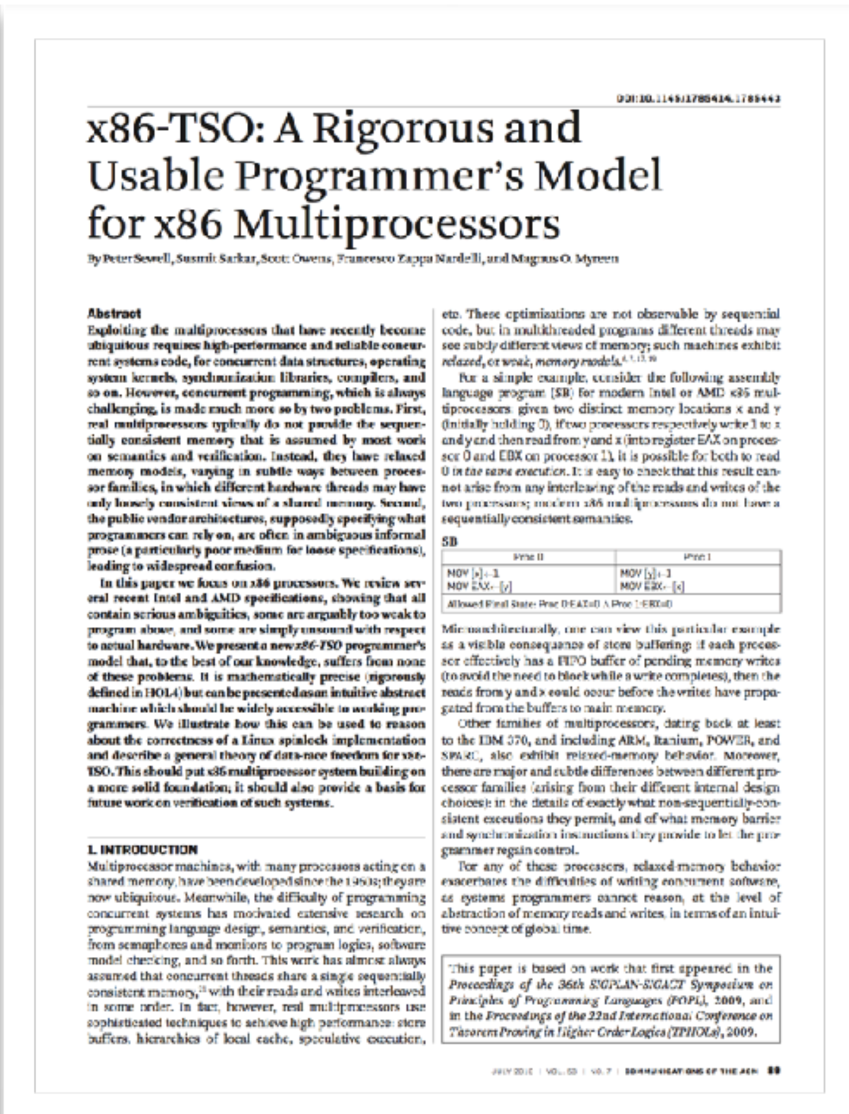


Metasketches for memory models



MemSynth can synthesize real models

x86



DOI:10.1145/1785414.1785443

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,[21] 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*.

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 EAX 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 EAX←[y]	MOV [y]←1 MOV EBX←[x]
Allowed Final State: Proc 0:EAX=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.

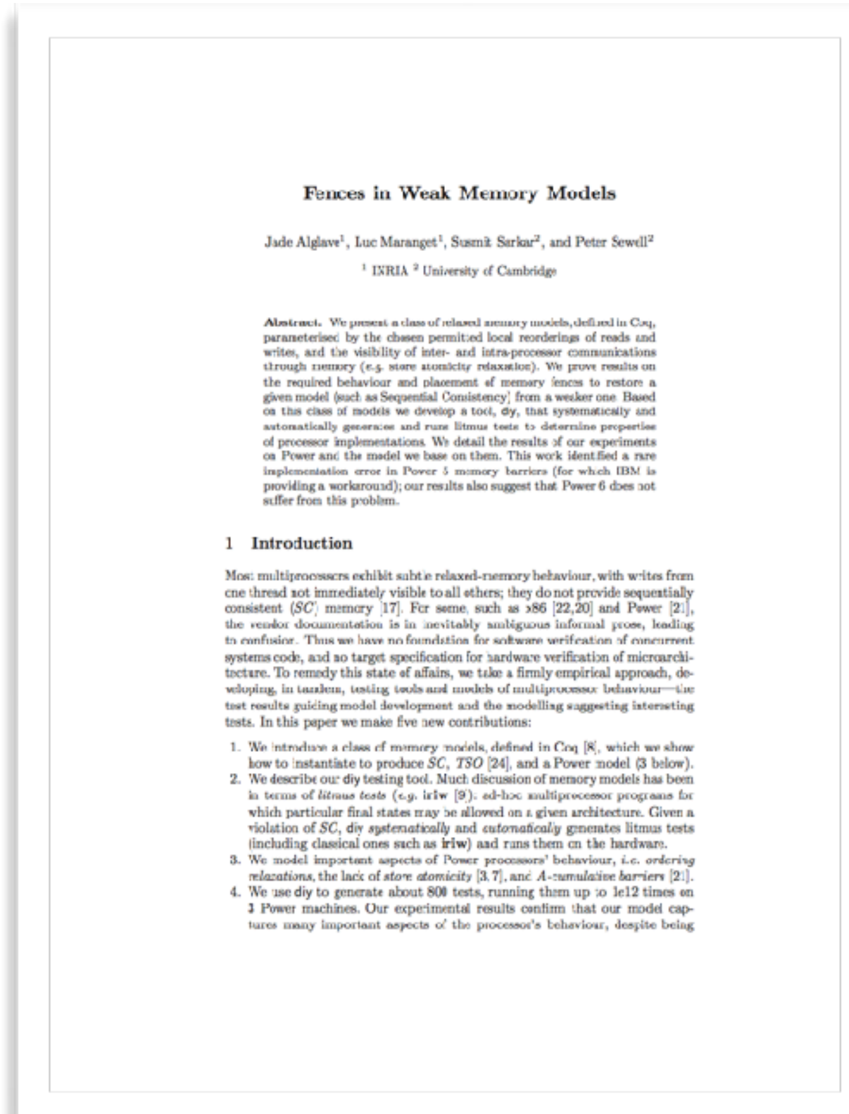
For any of these processors, relaxed-memory behavior exacerbates the difficulties of writing concurrent software, as systems programmers cannot reason, at the level of abstraction of memory reads and writes, in terms of an intuitive concept of global time.

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.

JULY 2010 | VOL. 53 | NO. 7 | COMMUNICATIONS OF THE ACM 89

10 tests (Intel manual)
2 seconds

PowerPC



Fences in Weak Memory Models

Jade Alglave[1], Luc Maranget[1], Susmit Sarkar[2], and Peter Sewell[2]

[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

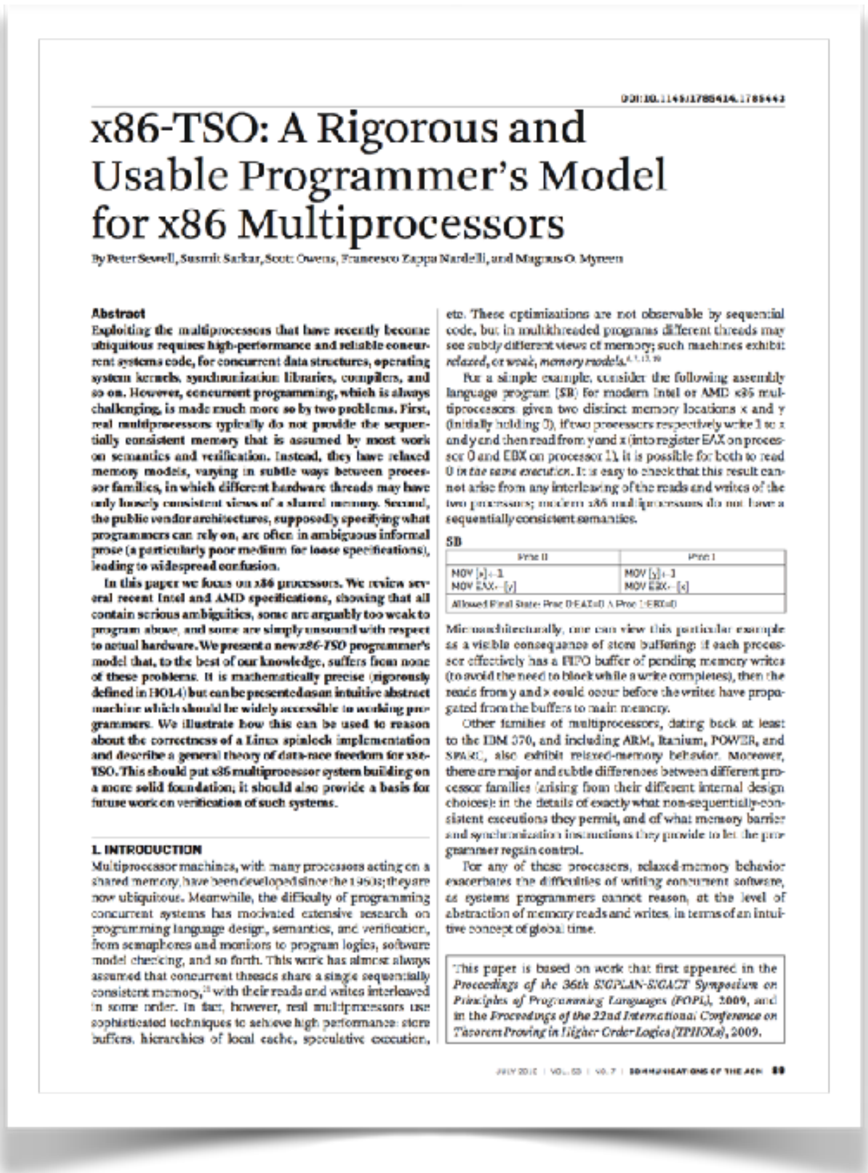
Most multiprocessors exhibit subtle relaxed-memory behaviour, with writes from one thread not immediately visible to all others; they do not provide sequentially consistent (*SC*) memory [17]. For some, such as x86 [22,20] and Power [21], the vendor documentation is in inevitably ambiguous informal prose, 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. iriw [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 **iriw**) 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

768 tests (existing work)
16 seconds

MemSynth can synthesize real models

x86



DOI:10.1145/1785414.1785443

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,[21] 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*.

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 EAX 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 EAX←[y]	MOV [y]←1 MOV EBX←[x]
Allowed Final State: Proc 0:EAX=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.

JULY 2010 | VOL. 53 | NO. 7 | COMMUNICATIONS OF THE ACM 89

10 tests (Intel manual)
2 seconds

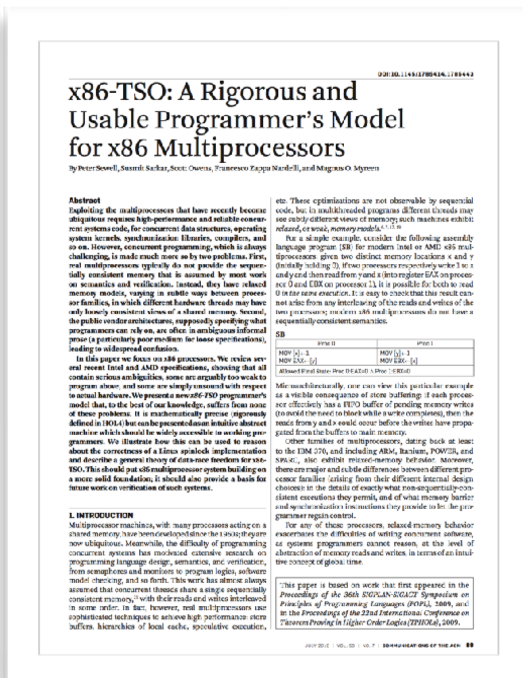
PowerPC

```
(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


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

10 tests (Intel manual)
2 seconds

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

PowerPC


Fences in Weak Memory Models

Jade Alglave[1], Luc Maranget[1], Susmit Sarkar[2], and Peter Sewell[2]

[1] INRIA [2] University of Cambridge

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!