GPS / Program Induction

December 17, 2019

Human program induction everywhere

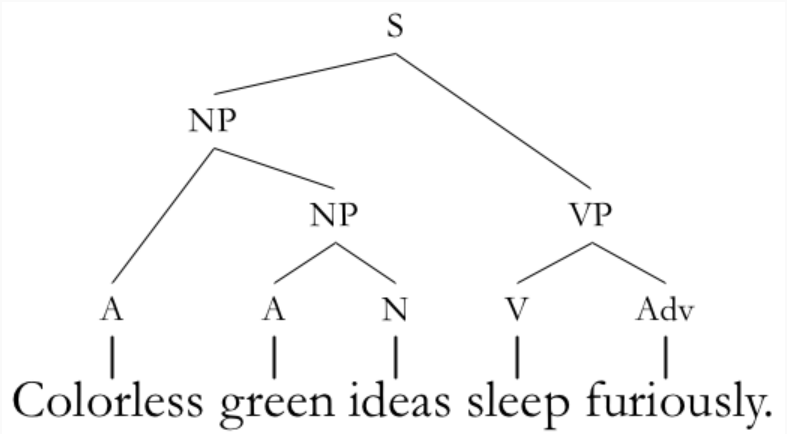
```
(MEMBER
 (LAMBDA (X L)
  (COND  ((NULL L) NIL)
         ((EQ X (FIRST L)) T)
         (T (MEMBER X (REST L))))))
```

Allen, Anatomy of Lisp, 1975

Human program induction everywhere

```
(MEMBER
 (LAMBDA (X L)
  (COND  ((NULL L) NIL)
         ((EQ X (FIRST L)) T)
         (T (MEMBER X (REST L))))))
```

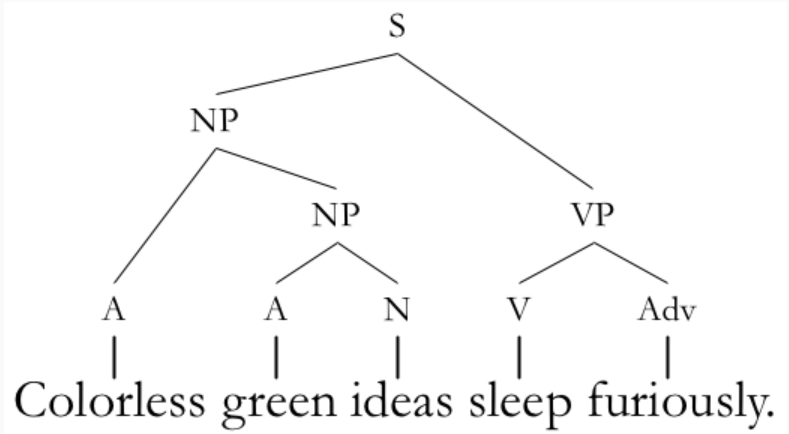
Allen, Anatomy of Lisp, 1975



Human program induction everywhere

```
(MEMBER
 (LAMBDA (X L)
  (COND  ((NULL L) NIL)
         ((EQ X (FIRST L)) T)
         (T (MEMBER X (REST L))))))
```

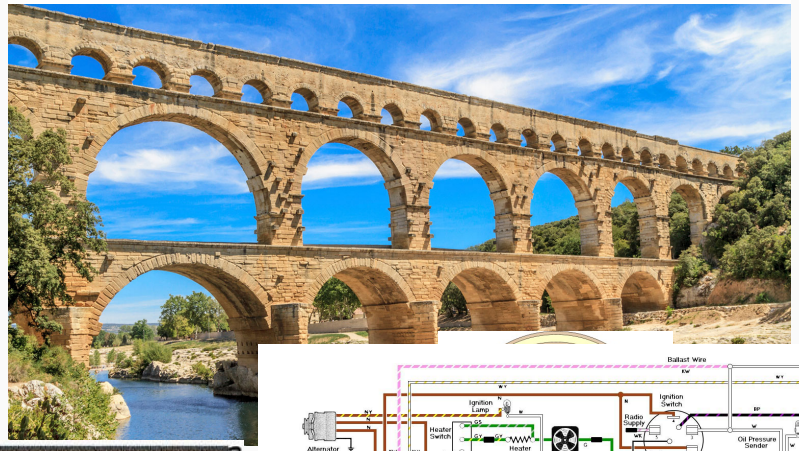
Allen, Anatomy of Lisp, 1975

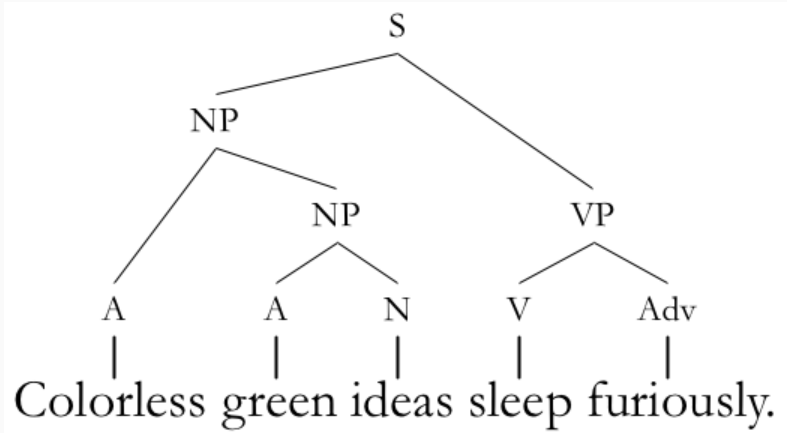


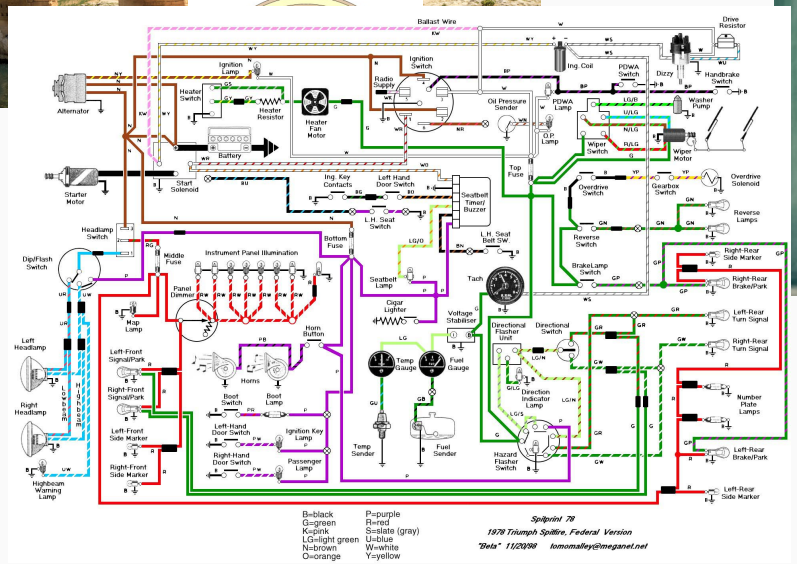
Human program induction everywhere

```
(MEMBER
 (LAMBDA (X L)
  (COND  ((NULL L) NIL)
         ((EQ X (FIRST L)) T)
         (T (MEMBER X (REST L))))))
```

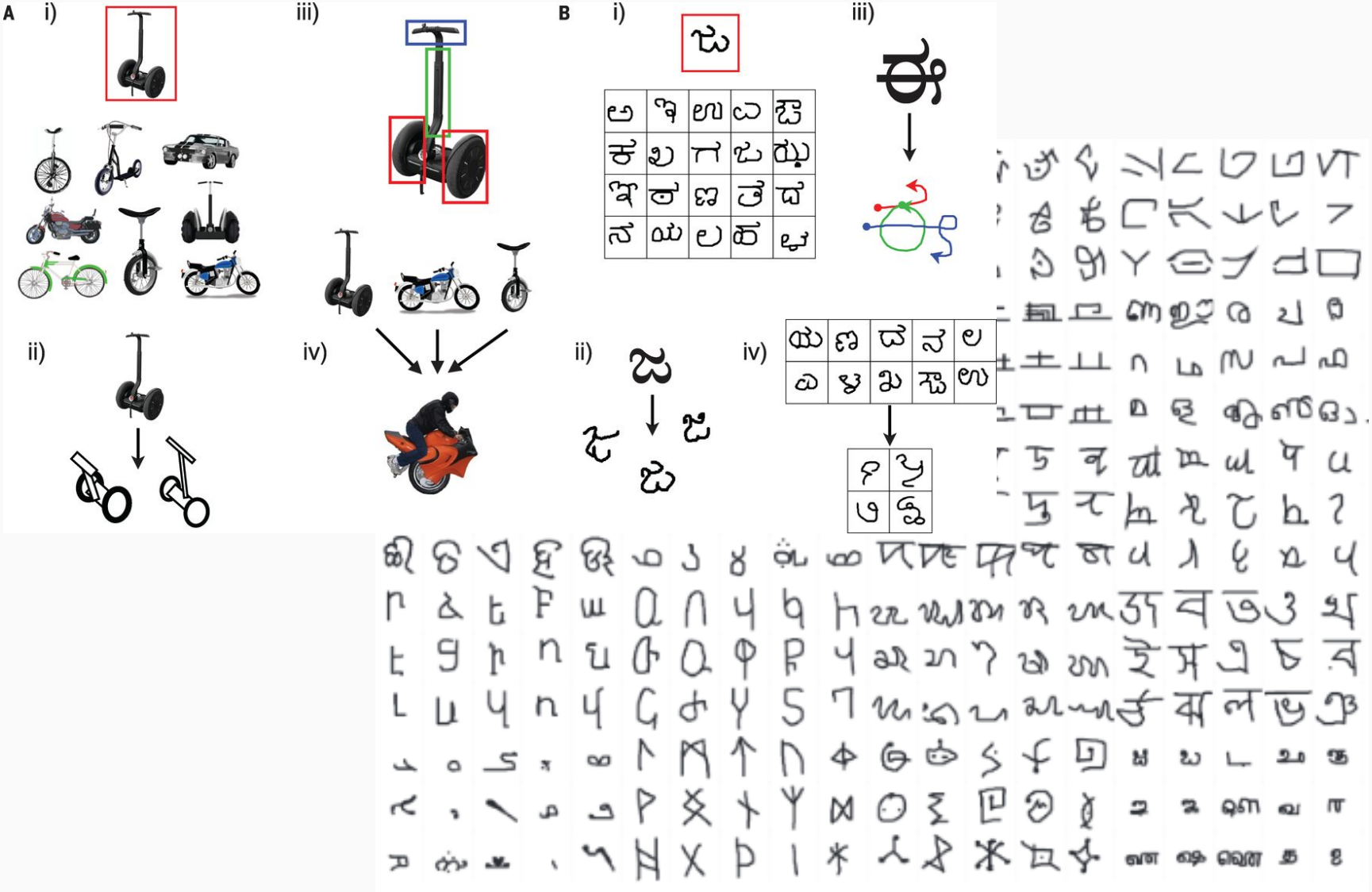
Allen, Anatomy of Lisp, 1975







Human program induction everywhere



Lake et al 2015

Exploration-Compression

Dechter et al. 2013

What if we don't have the right programming language for solving our problems?

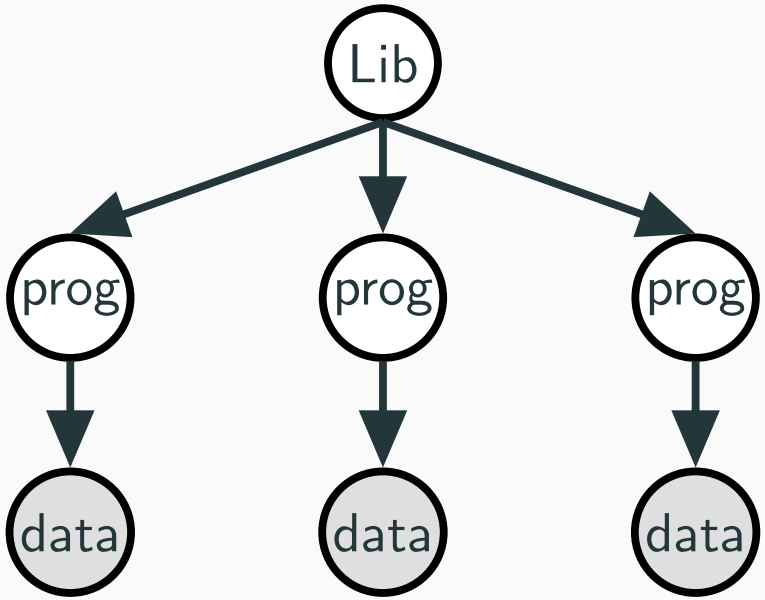
Exploration-Compression

Dechter et al. 2013

What if we don't have the right programming language for solving our problems?

Idea: Iteratively grow the language, alternating problem-solving and language-growing. Grow the language by "compressing out" reused program components, and cache those components in a learned library

Exploration-Compression as Bayesian inference

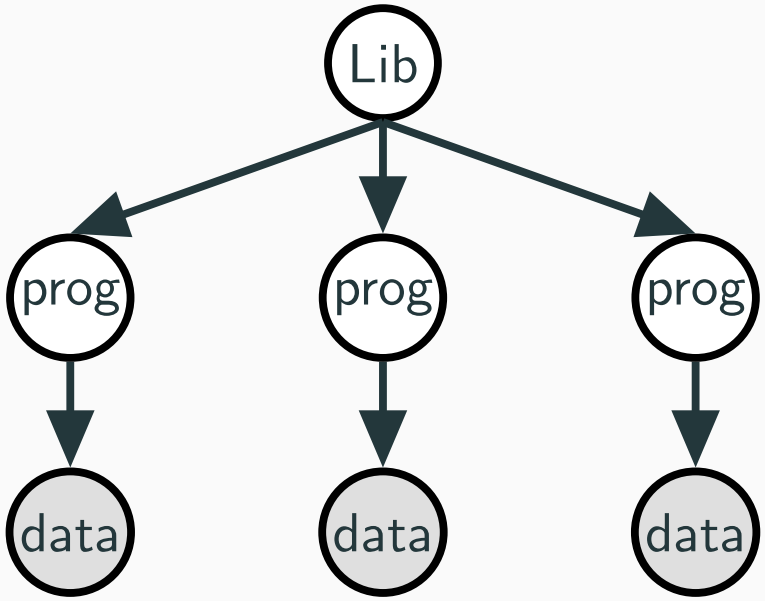


Explore: Enumerate programs built from library (Lib). Keep program if it solves a task (explains an observed datum)

Compress: Update library by incorporating reused subtrees of programs found during Explore. maximize:

$$\mathbb{P}\left[\text{Lib}\right] \prod_{\text{task}} \sum_{\substack{\text{prog solving task} \\ \text{prog found during Explore}}} \mathbb{P}\left[\text{prog}|\text{Lib}\right]$$

Exploration-Compression as Bayesian inference



Explore: Enumerate programs built from library (Lib). Keep program if it solves a task (explains an observed datum)

Compress: Update library by incorporating reused subtrees of programs found during Explore. *minimize*:

$$\underbrace{-\log \mathbb{P}\left[\text{Lib}\right]}_{\text{size of library}} + \sum_{\text{task}} \underbrace{-\log \max_{\substack{\text{prog solving task} \\ \text{prog found during Explore}}} \mathbb{P}\left[\text{prog}|\text{Lib}\right]}_{\text{size of program solving task wrt library}}$$

Compression in action

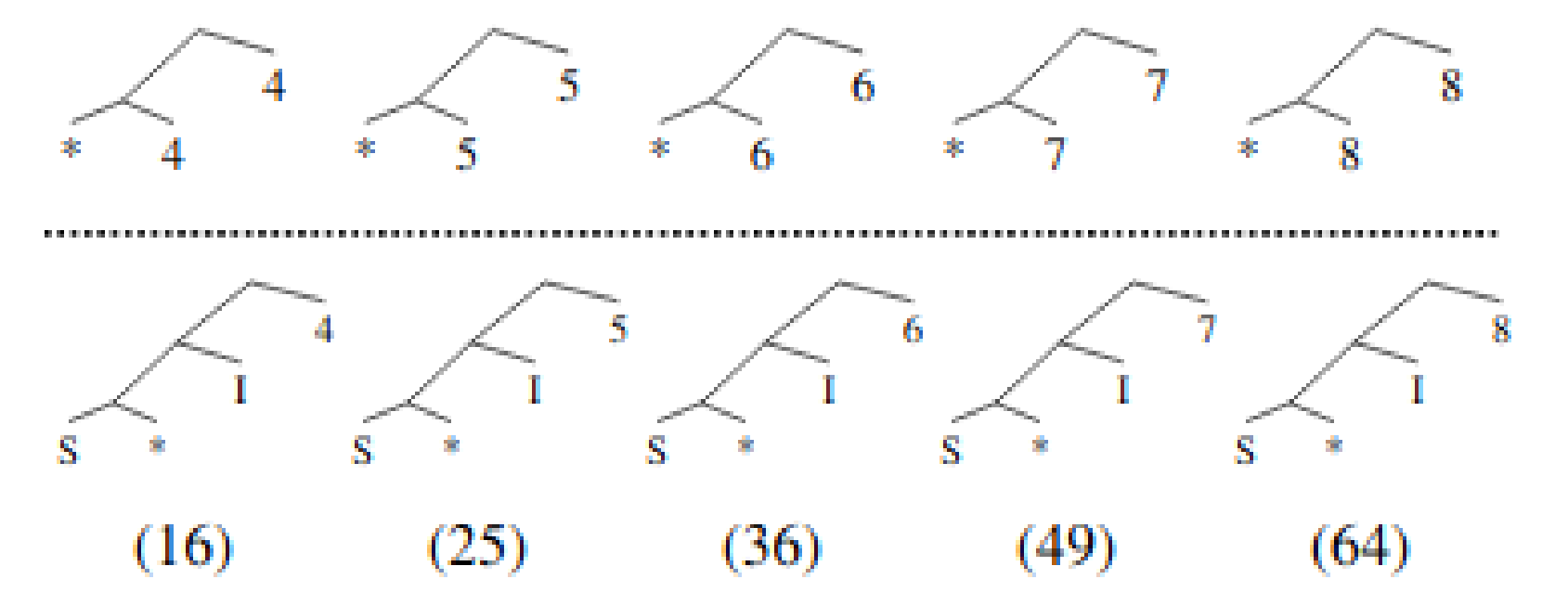
Suppose you need to build programs that calculate 16, 25, 36, 49, 64

Intuitively, here we should invent a new function called “square”

Compression in action

Suppose you need to build programs that calculate 16, 25, 36, 49, 64

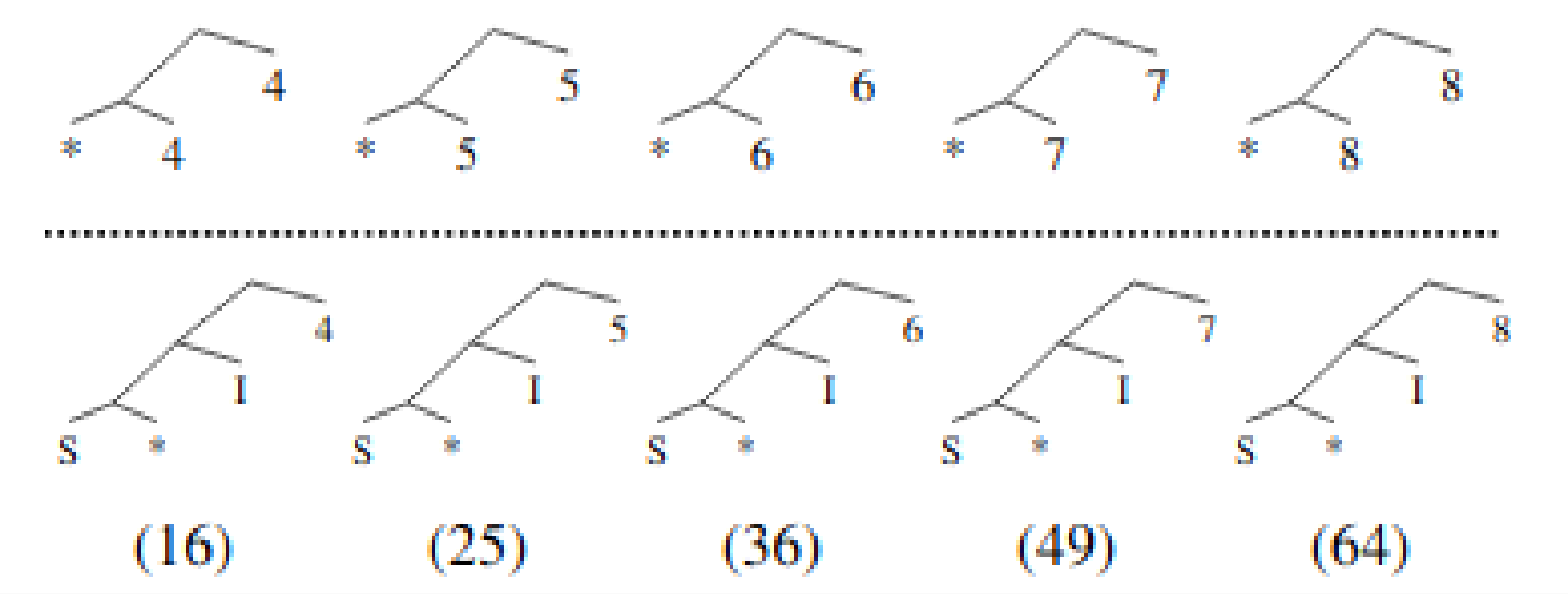
Intuitively, here we should invent a new function called “square”



Compression in action

Suppose you need to build programs that calculate 16, 25, 36, 49, 64

Intuitively, here we should invent a new function called "square"



symbols w/o square (top): $3 \times 5 = 15$

symbols w/ square (bottom): $3 + 2 \times 5 = 12$

Bootstrap learning dynamics

At first, almost nothing is solvable. System bootstraps itself by solving easier problems first, reusing motifs discovered in their solutions to progress toward harder problems

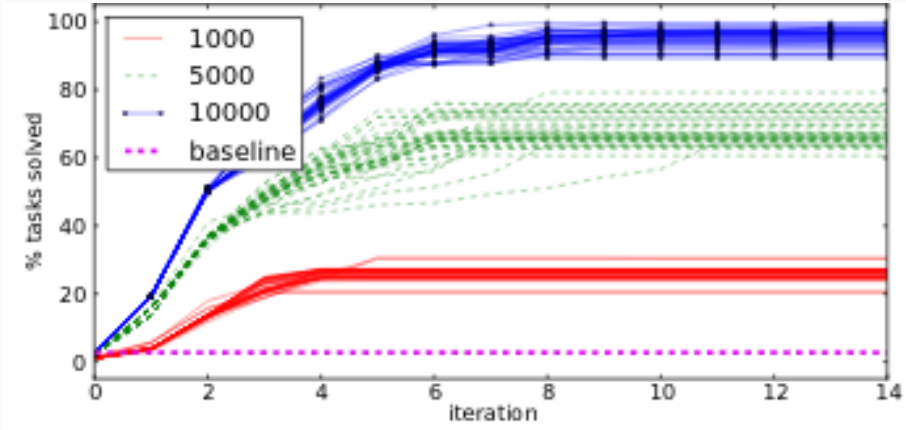


Figure 2: Learning curves as a function of frontier size. As frontier size is increased, curves plateau closer to 100% performance. A baseline search over 150000 expressions only hits 3% of the tasks (dashed pink line).

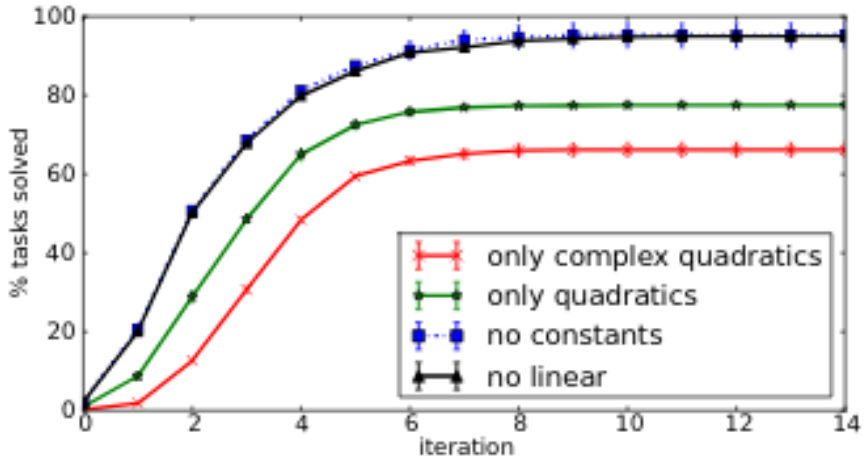
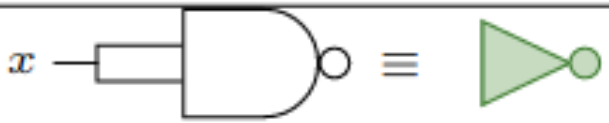
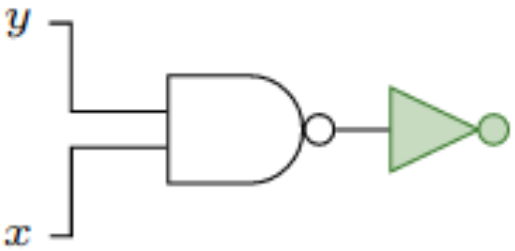
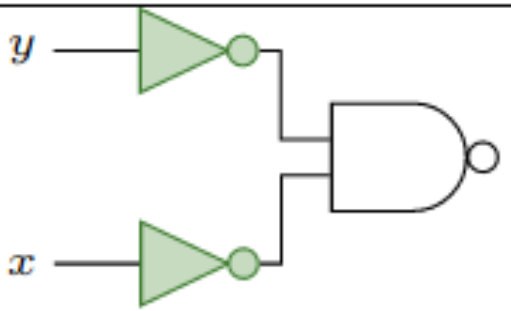
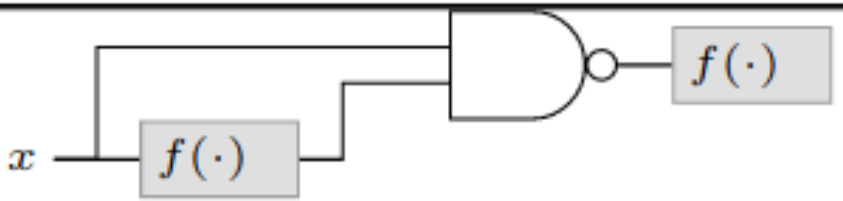
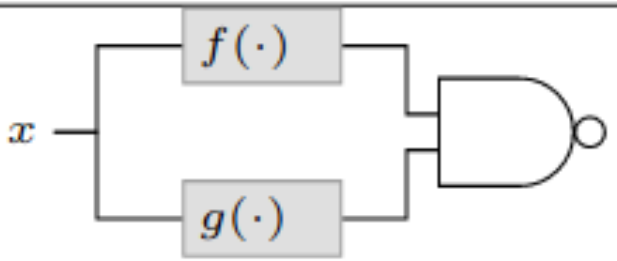


Figure 3: How do different task sets affect learning curves? Learning curves at a frontier size of 10000 for different task sets.

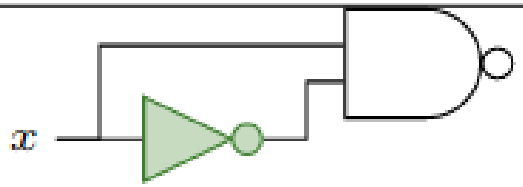
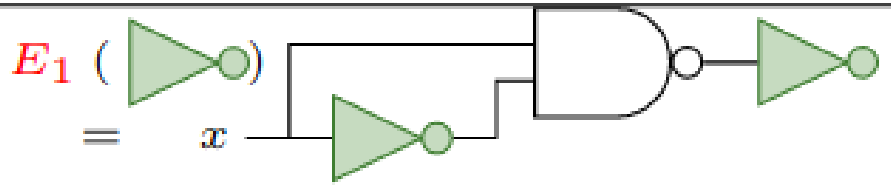
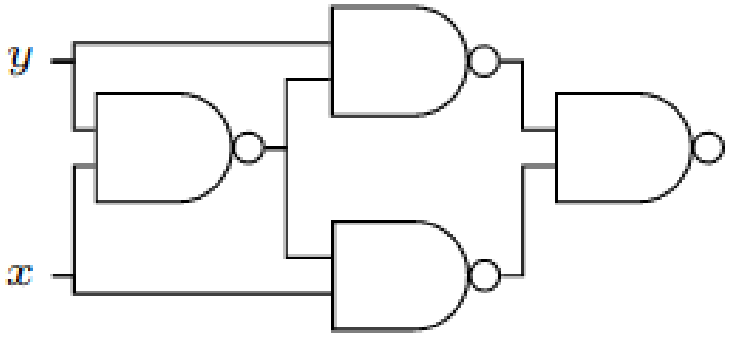
Hinges on:

- sufficient search effort (frontier size, left)
- a graded mixture of difficulty (right)

Learning interpretable concepts in the library

func.	CL expression	schematic
NOT	(S NAND I)	
AND	(C B NAND) (B (S NAND I)) → (C B NAND) (B NOT)	
OR	((B (C (B (S NAND) NAND))) (S NAND I) → ((B (C (B (S NAND) NAND))) NOT	
E_1	S B (S NAND)	
E_2	(B (C B NAND) S)	

Learning interpretable concepts in the library

TRUE	(S NAND) (S NAND I) → (S NAND) NOT	
FALSE	(S B (S NAND)) (S NAND I) → E_1 NOT	
XOR	((S (B C (((B (C B NAND)) S) (((B (C B NAND)) S) (B C ((B (B NAND)) NAND))))))) I) → ((S (B C (E_2 (E_2 (B C ((B (B NAND)) NAND))))))) I)	

Growing domain-specific knowledge

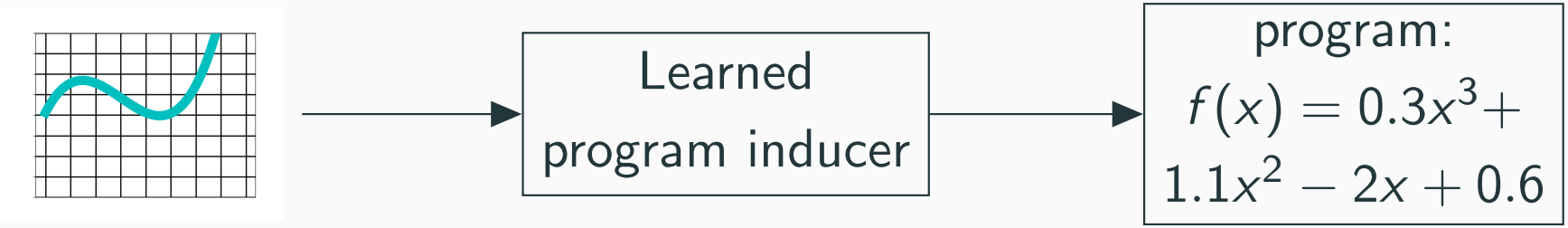
Goal: acquire domain-specific knowledge needed to induce a class of programs

- Library of concepts (declarative knowledge; domain specific language; generative model over programs)
- Inference strategy (procedural knowledge; synthesis algorithm)

Growing domain-specific knowledge

Goal: acquire domain-specific knowledge needed to induce a class of programs

- Library of concepts (declarative knowledge; domain specific language; generative model over programs)
- Inference strategy (procedural knowledge; synthesis algorithm)



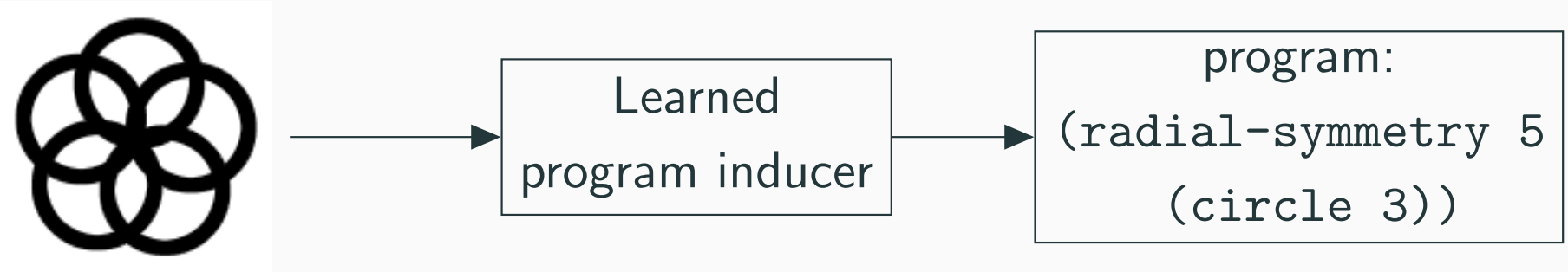
Concepts: x^3, $\alpha x + \beta$, etc

Inference strategy: neurosymbolic search for programs

Growing domain-specific knowledge

Goal: acquire domain-specific knowledge needed to induce a class of programs

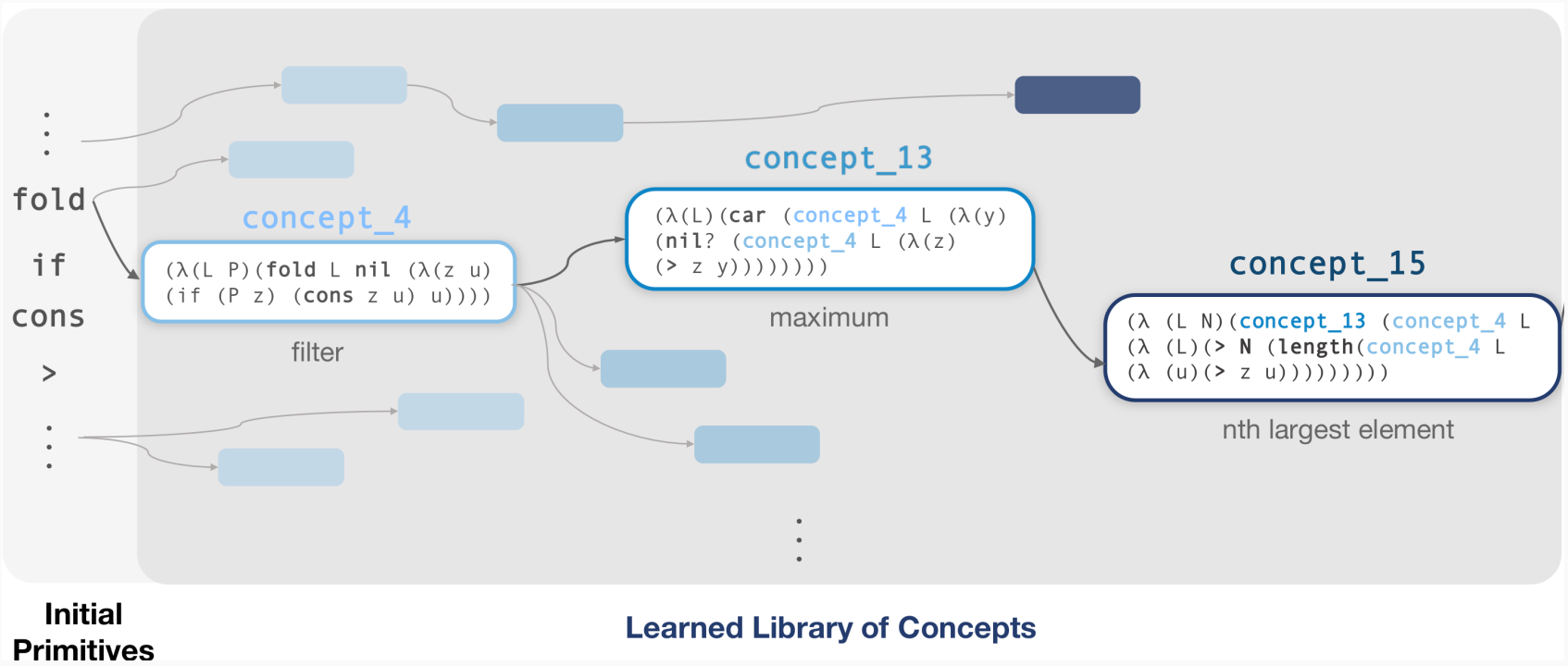
- Library of concepts (declarative knowledge; domain specific language; generative model over programs)
- Inference strategy (procedural knowledge; synthesis algorithm)



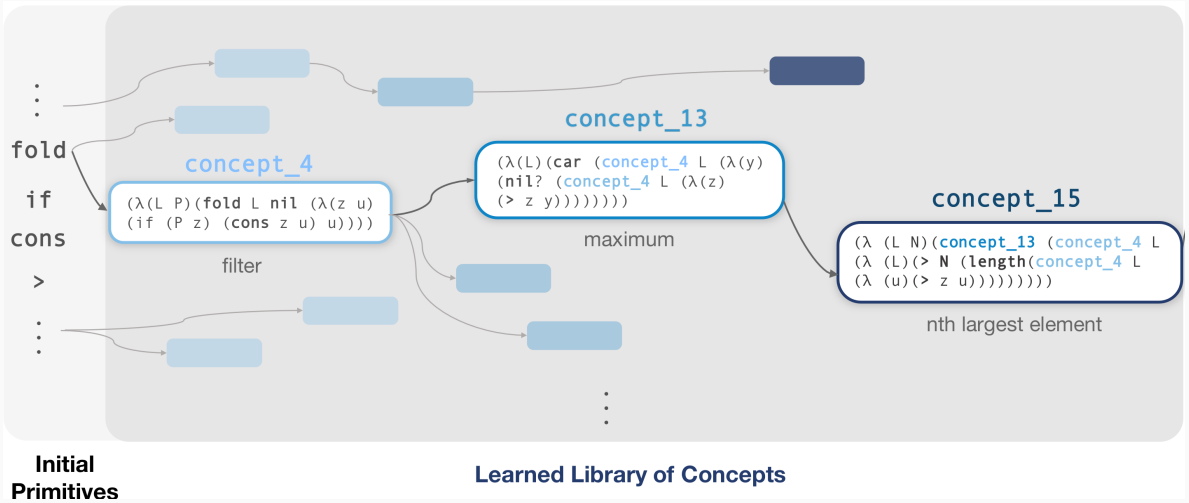
Concepts: `circle`, `radial-symmetry`, etc

Inference strategy: neurosymbolic search for programs

Library learning



Library learning



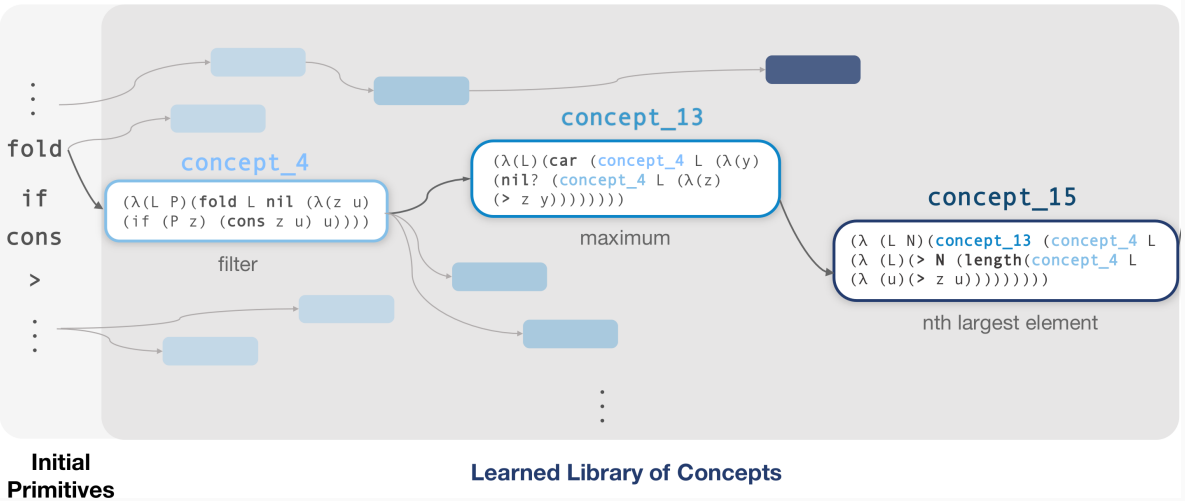
Problem: sort list

Solution:

```
(map (λ (n) (concept_15 L (+ 1 n))) (range (length L))))
```

get nth largest element where n = 1, 2, 3 ….length of list

Library learning



Problem: sort list

Solution:

```
(map (λ (n) (concept_15 L (+ 1 n))) (range (length L))))
```

get nth largest element where n = 1, 2, 3 ….length of list

Solution in initial primitives:

```
(λ (x) (map (λ (y) (car (fold (fold x nil (λ (z u) (if (gt?
(+ y 1) (length (fold x nil (λ (v w) (if (gt? z v) (cons v
w) w))))) (cons z u) u))) nil (λ (a b) (if (nil? (fold
(fold x nil (λ (c d) (if (gt? (+ y 1) (length (fold x nil
(λ (e f) (if (gt? c e) (cons e f) f))))) (cons c d) d)))
nil (λ (g h) (if (gt? g a) (cons g h) h)))) (cons a b)
b))))) (range (length x))))
```

Discovered Problem Solutions

DreamCoder

- **Wake:** Solve problems by writing programs
- **Sleep:** Improve DSL and neural recognition model:
  - **Abstraction sleep:** Improve library
  - **Dream sleep:** Improve neural inference model
- Combines ideas from Wake-Sleep & Exploration-Compression

DreamCoder

- **Wake:** Solve problems by writing programs
- **Sleep:** Improve DSL and neural recognition model:
  - **Abstraction sleep:** Improve library
  - **Dream sleep:** Improve neural inference model
- Combines ideas from Wake-Sleep & Exploration-Compression

List Processing

Sum List

```
[1 2 3] → 6
[4 6 8 1] → 17
```

Double

```
[1 2 3 4] → [2 4 6 8]
[6 5 1] → [12 10 2]
```

Check Evens

```
[0 2 3] → [T T F]
[2 4 9 6] → [T T F T]
```

Text Editing

Abbreviate

```
Allen Newell → A.N.
Herb Simon → H.S.
```

Drop Last Characters

```
jabberwocky → jabberw
copycat → cop
```

Extract

```
see spot(run) → run
a (bee) see → bee
```

Regexes

Phone Numbers

```
(555) 867-5309
(650) 555-2368
```

Currency

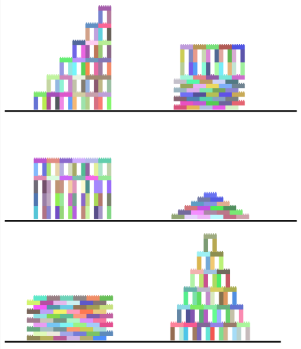
```
$100.25
$4.50
```

Dates

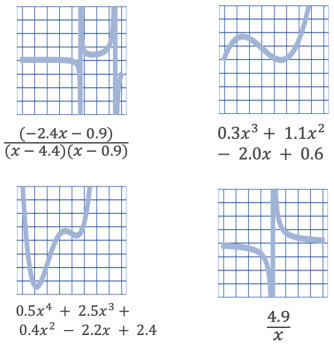
```
Y1775/0704
Y2000/0101
```

LOGO Graphics

Block Towers



Symbolic Regression



Recursive Programming

Filter

```
[■■■■■] → [■■]
[■■■■■■] → [■■■■]
[■■■■] → [■■■]
```

Index List

```
0, [■■■■■■] → ■
1, [■■■■■■] → ■
1, [■■■■■■■] → ■
```

Length

```
[■■■■] → 4
[■■■■■■] → 6
[■■■] → 3
```

Every Other

```
[■■■■] → [■■]
[■■■■■] → [■■■]
[■■■■■■] → [■■■]
```

Physics

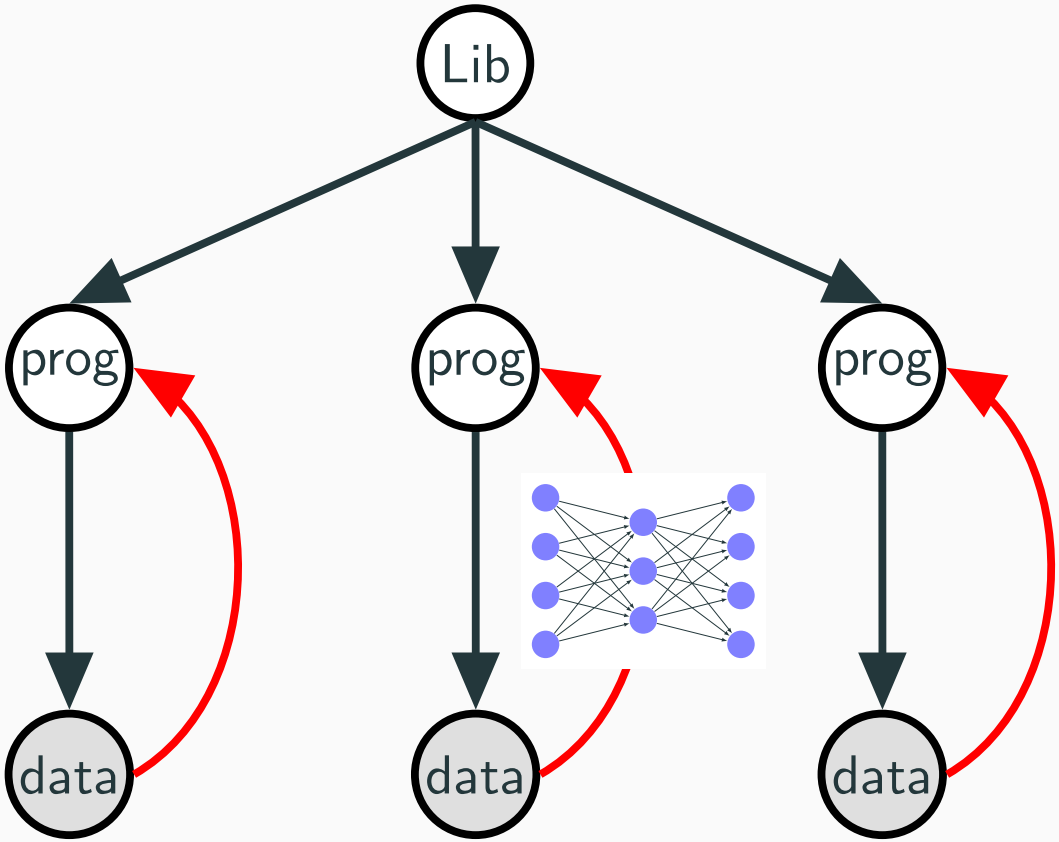
$$KE = \frac{1}{2}m|\vec{v}|^2$$

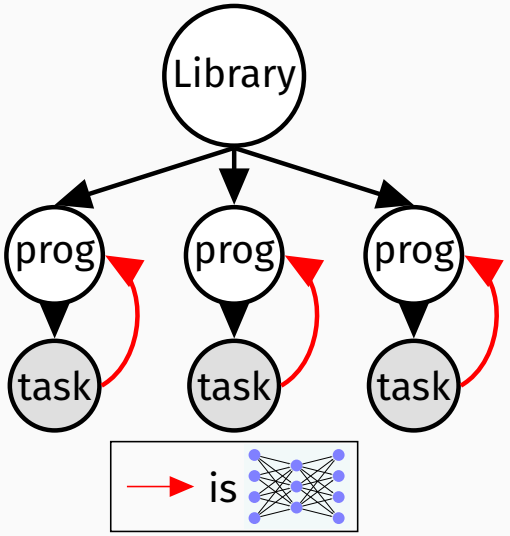
$$\vec{a} = \frac{1}{m}\sum_i \vec{F}_i$$

$$\vec{F} \propto \frac{q_1 q_2}{|\vec{r_1} - \vec{r_2}|^2}\widehat{r_1 - r_2}$$

$$R_{total} = \left(\Sigma_i \frac{1}{R_i}\right)^{-1}$$

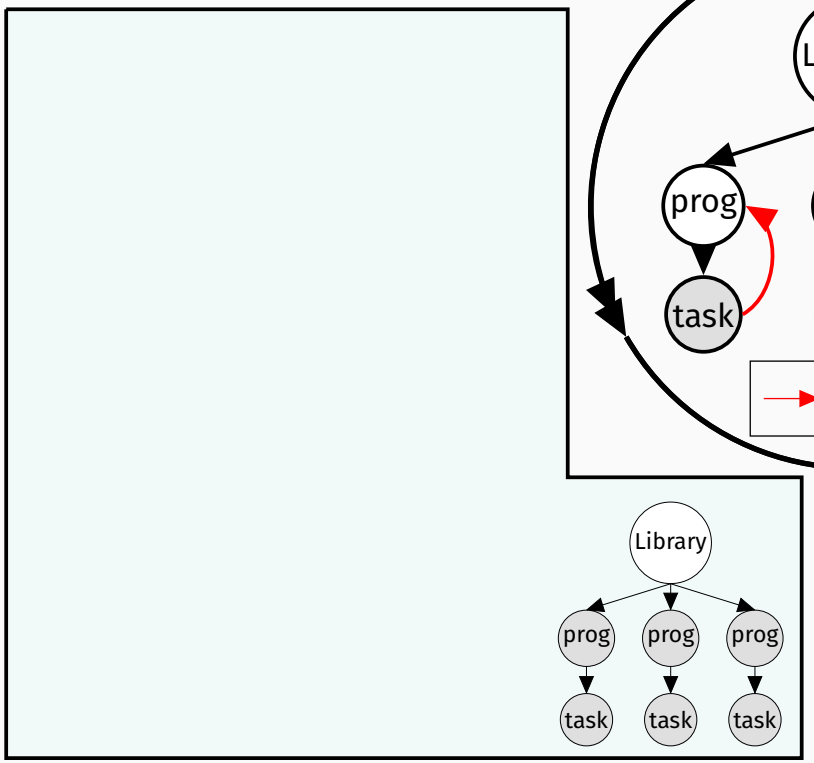
Library learning as amortized Bayesian inference



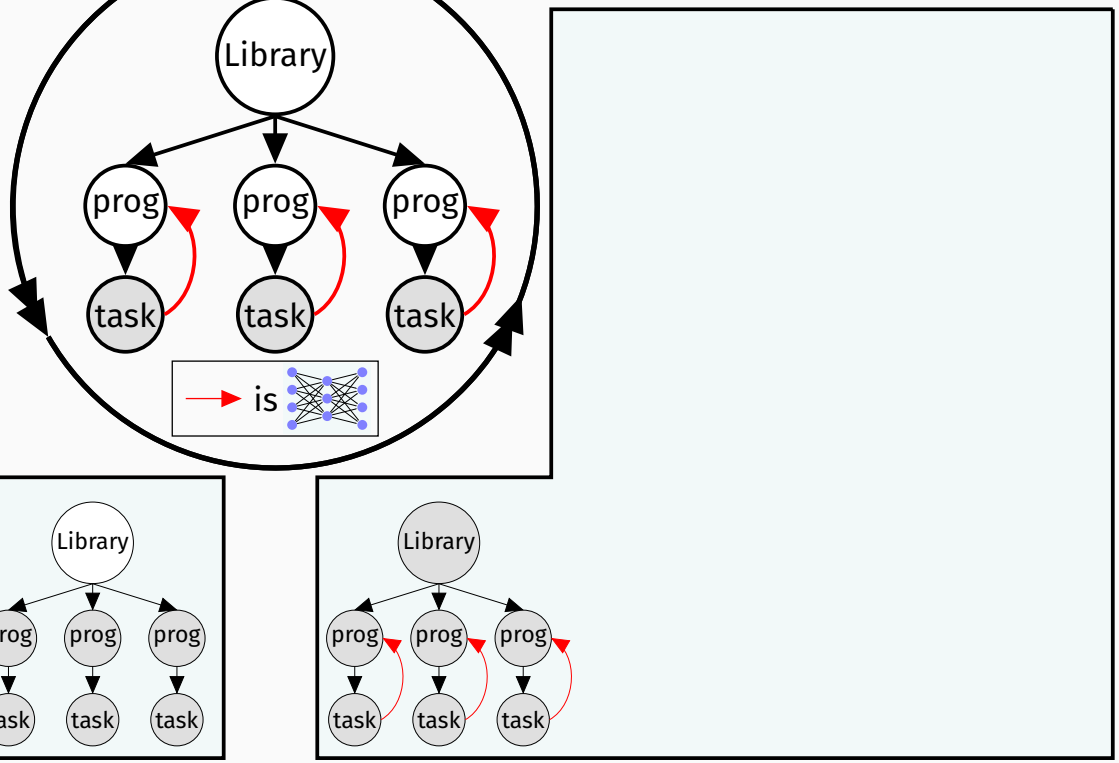

Library
prog
prog
prog
task
task
task
is

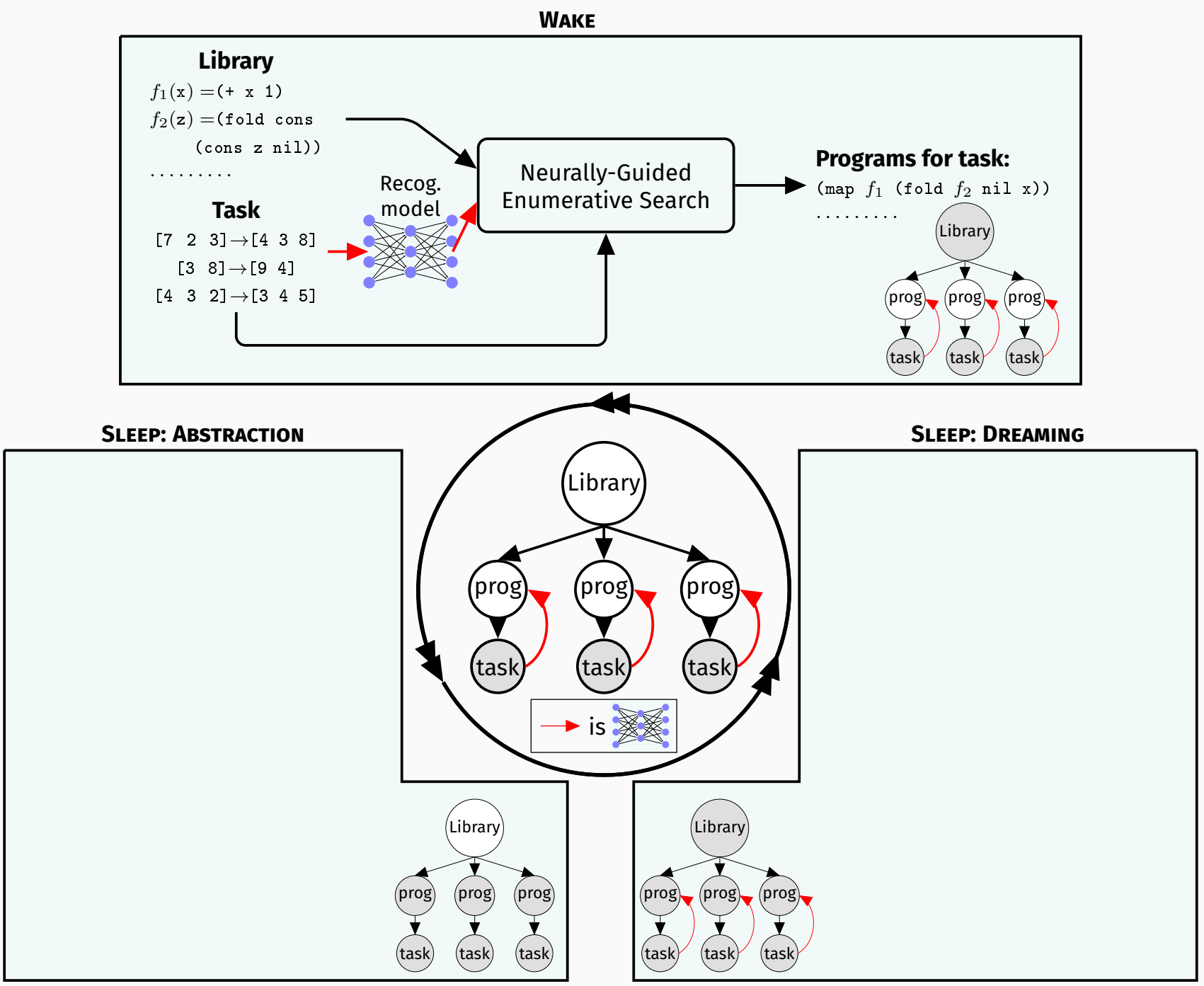
Wake

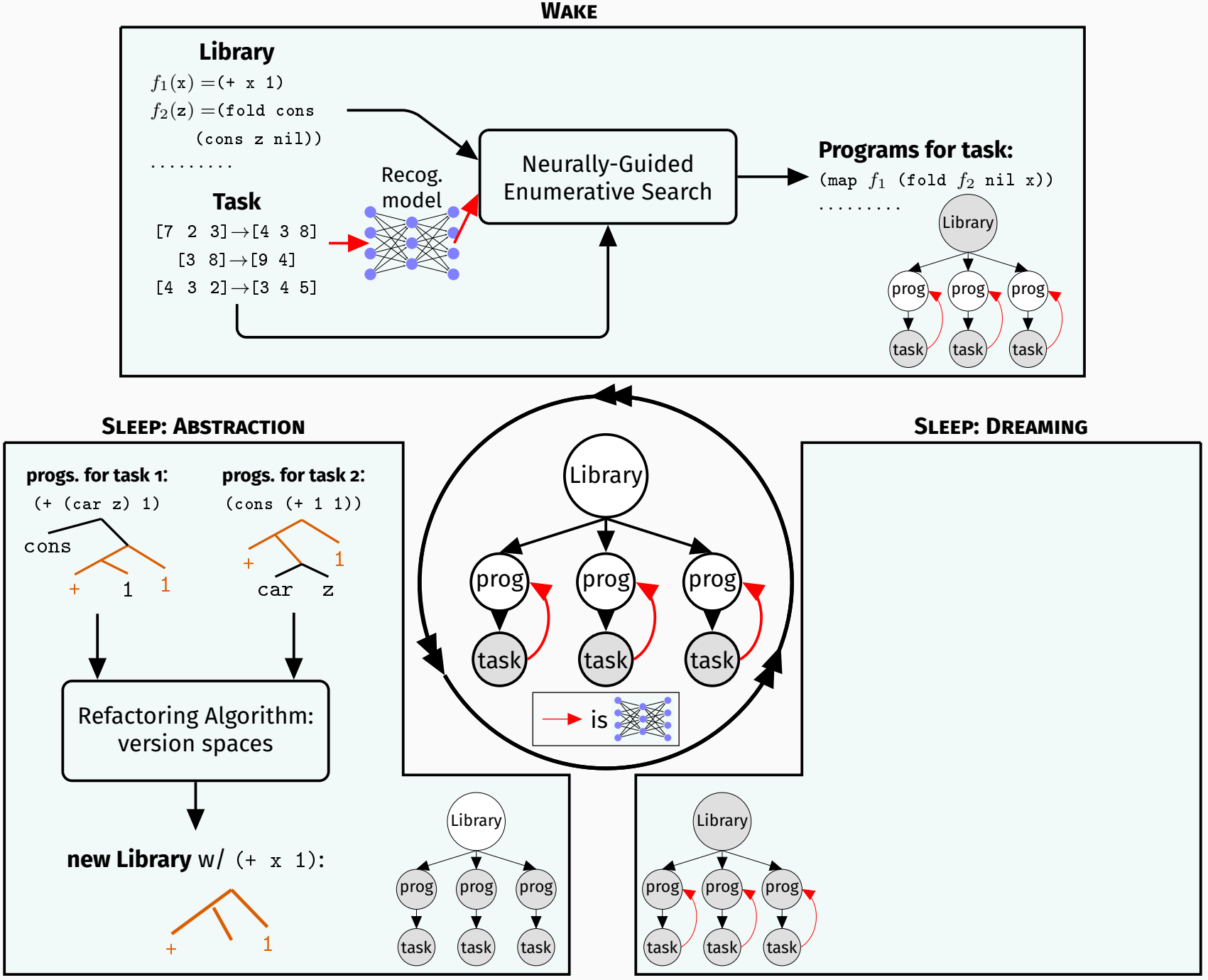
Sleep: Abstraction

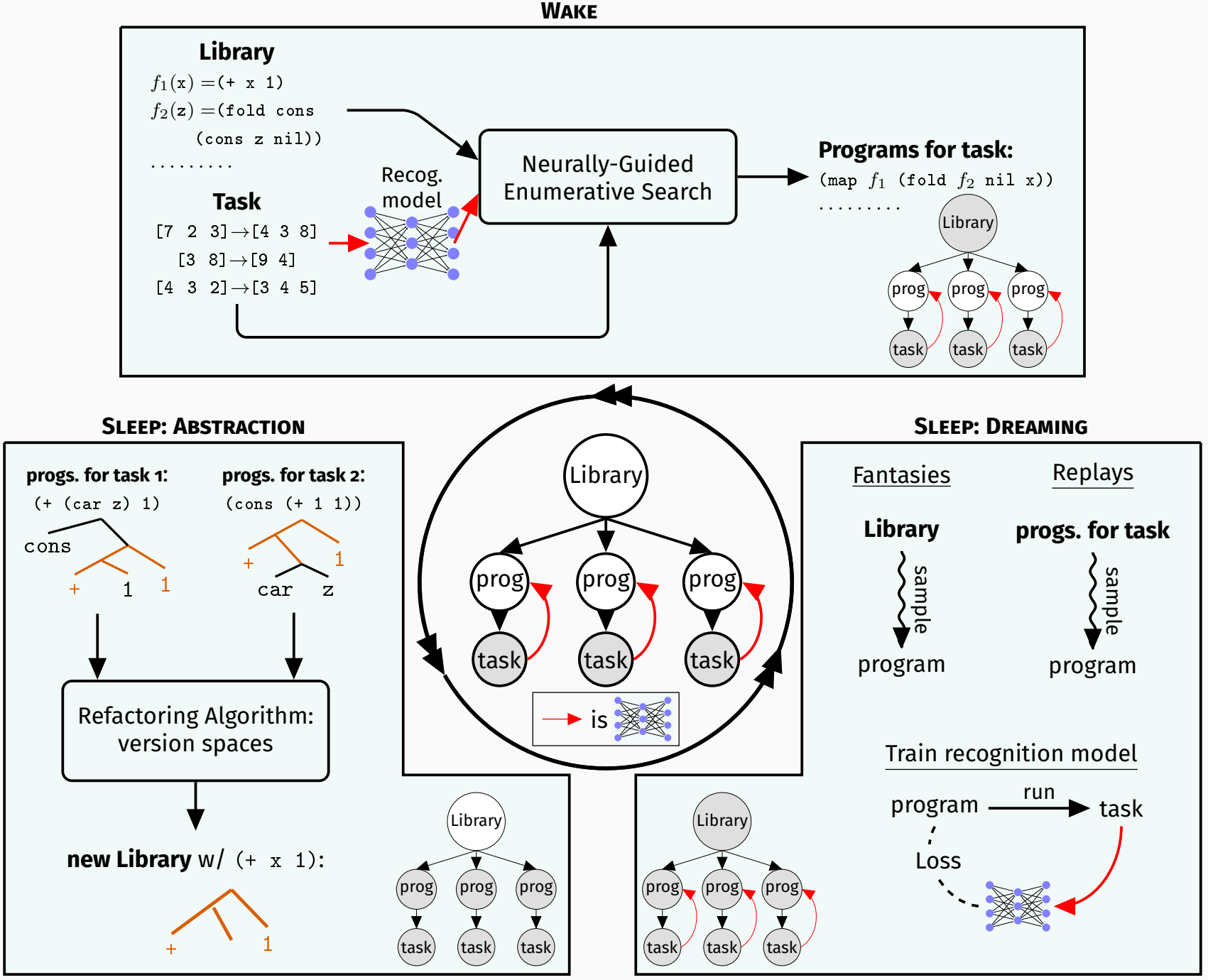


Sleep: Dreaming

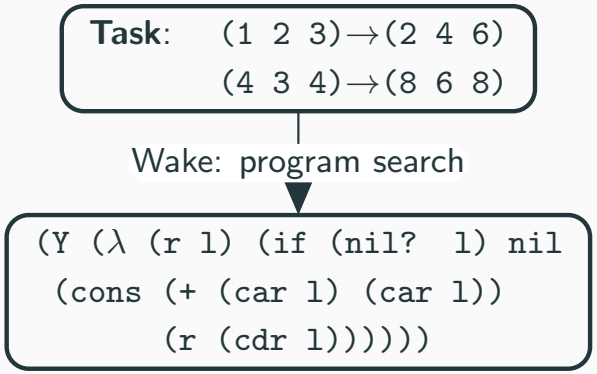


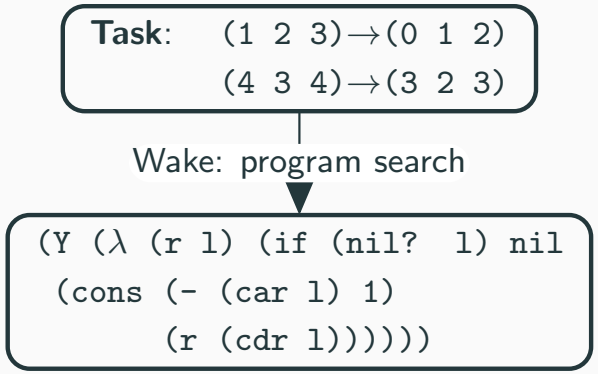

WAKE
Library
f_1(x) =(+ x 1)
f_2(z) =(fold cons
(cons z nil))
.....
Task
[7 2 3]→[4 3 8]
[3 8]→[9 4]
[4 3 2]→[3 4 5]
Recog. model
Neurally-Guided Enumerative Search
Programs for task:
(map f_1 (fold f_2 nil x))
.....
Library
prog
task
SLEEP: ABSTRACTION
SLEEP: DREAMING
Library
prog
task
is
Library
prog
task
Library
prog
task


WAKE
Library
f_1(x) =(+ x 1)
f_2(z) =(fold cons
(cons z nil))
.....
Task
[7 2 3]→[4 3 8]
[3 8]→[9 4]
[4 3 2]→[3 4 5]
Recog. model
Neurally-Guided Enumerative Search
Programs for task:
(map f_1 (fold f_2 nil x))
.....
Library
prog
task
SLEEP: ABSTRACTION
progs. for task 1:
(+ (car z) 1)
progs. for task 2:
(cons (+ 1 1))
cons
car z
Refactoring Algorithm: version spaces
new Library w/ (+ x 1):
SLEEP: DREAMING
is

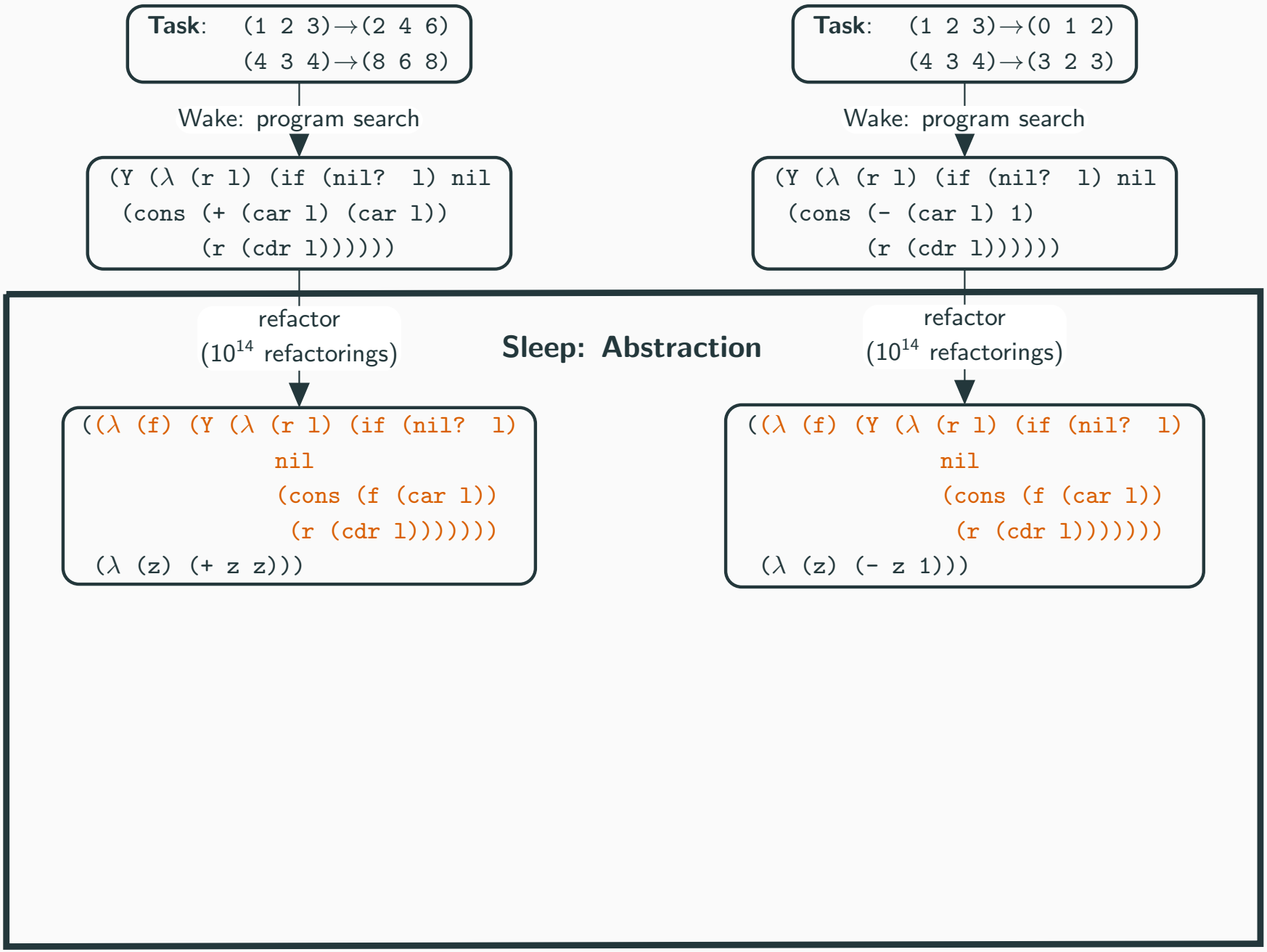

WAKE
Library
f_1(x) =(+ x 1)
f_2(z) =(fold cons
(cons z nil))
.....
Task
[7 2 3]→[4 3 8]
[3 8]→[9 4]
[4 3 2]→[3 4 5]
Recog. model
Neurally-Guided Enumerative Search
Programs for task:
(map f_1 (fold f_2 nil x))
.....
Library
prog
task
SLEEP: ABSTRACTION
progs. for task 1:
(+ (car z) 1)
cons
progs. for task 2:
(cons (+ 1 1))
car z
Refactoring Algorithm: version spaces
new Library w/ (+ x 1):
Library
is
SLEEP: DREAMING
Fantasies
Replays
Library
progs. for task
sample
program
Train recognition model
program
run
task
Loss

Abstraction Sleep: Growing the library via refactoring

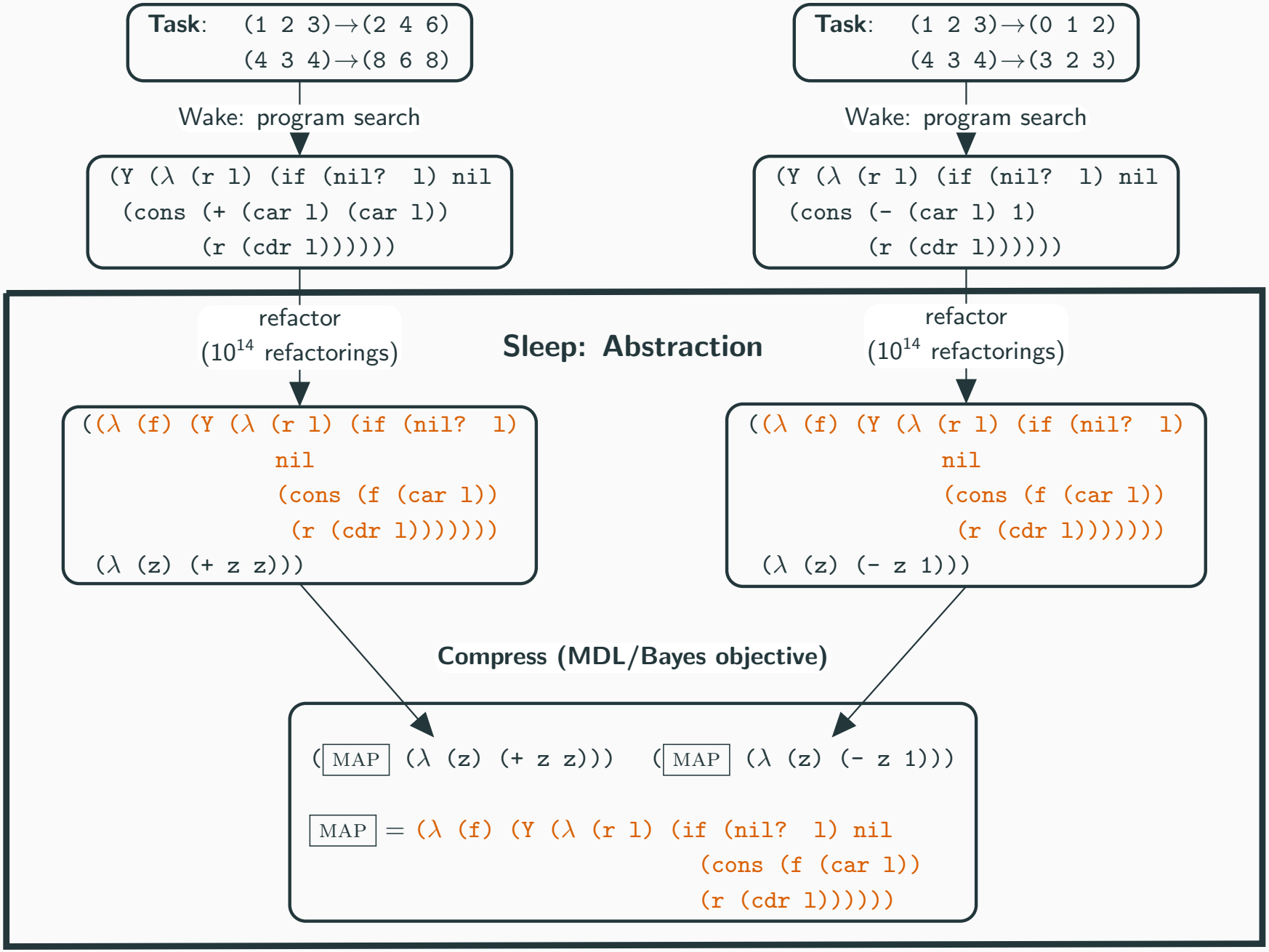




Abstraction Sleep: Growing the library via refactoring



Abstraction Sleep: Growing the library via refactoring



Version space algebra for refactoring

$$
\begin{aligned}
\text{Expr} \rightarrow\ & \text{var} \\
\mid\ & \lambda\text{var.Expr} \\
\mid\ & (\text{Expr Expr}) \\
\mid\ & \text{primitive}
\end{aligned}
$$

Version space algebra for refactoring

$$
\begin{array}{lll}
\text{Expr} \to \text{var} & \text{VS} \to \text{var} & \\
\quad \mid \lambda\text{var.Expr} & \quad \mid \lambda\text{var.VS} & \\
\quad \mid (\text{Expr Expr}) & \quad \mid (\text{VS VS}) & \\
\quad \mid \text{primitive} & \quad \mid \text{primitive} & \\
 & \quad \mid \text{VS} \uplus \text{VS} & \textit{nondeterministic choice} \\
 & \quad \mid \Lambda & \textit{choose any expression} \\
 & \quad \mid \varnothing & \textit{choose no expression}
\end{array}
$$

Version space algebra for refactoring

$$
\begin{array}{lll}
\text{Expr} \to \text{var} & \text{VS} \to \text{var} & \\
\quad \mid \lambda\text{var}.\text{Expr} & \quad \mid \lambda\text{var}.\text{VS} & \\
\quad \mid (\text{Expr Expr}) & \quad \mid (\text{VS VS}) & \\
\quad \mid \text{primitive} & \quad \mid \text{primitive} & \\
 & \quad \mid \text{VS} \uplus \text{VS} & \textit{nondeterministic choice} \\
 & \quad \mid \Lambda & \textit{choose any expression} \\
 & \quad \mid \varnothing & \textit{choose no expression}
\end{array}
$$

what version spaces mean:

$$
\begin{array}{ll}
[\![\text{var}]\!] = \{\text{var}\} & [\![v_1 \uplus v_2]\!] = \{e : v \in \{v_1, v_2\}, e \in [\![v]\!]\} \\
[\![\lambda x.v]\!] = \{\lambda x.e : e \in [\![v]\!]\} & [\![(v_1\, v_2)]\!] = \{(e_1\, e_2) : e_1 \in [\![v_1]\!], e_2 \in [\![v_2]\!]\} \\
[\![\varnothing]\!] = \varnothing & [\![\Lambda]\!] = \Lambda
\end{array}
$$

Using version spaces

$$
\begin{aligned}
\mathrm{VS} \rightarrow\ & \mathrm{var} \mid \lambda \mathrm{var}.\mathrm{VS} \mid (\mathrm{VS}\ \mathrm{VS}) \mid \mathrm{primitive} \\
\mid\ & \mathrm{VS} \uplus \mathrm{VS} && \textit{nondeterministic choice} \\
\mid\ & \Lambda && \textit{choose any expression} \\
\mid\ & \varnothing && \textit{choose no expression}
\end{aligned}
$$

Using version spaces

$$\begin{aligned}
\text{VS} \rightarrow\ & \text{var} \mid \lambda\text{var.VS} \mid (\text{VS VS}) \mid \text{primitive} \\
\mid\ & \text{VS} \uplus \text{VS} && \textit{nondeterministic choice} \\
\mid\ & \Lambda && \textit{choose any expression} \\
\mid\ & \varnothing && \textit{choose no expression}
\end{aligned}$$

exploit the fact that $e \in [\![v]\!]$ can be efficiently computed:

$$\textsc{refactor}(v|\text{Lib}) = \begin{cases} e, \text{ if } e \in \text{Lib and } e \in [\![v]\!] \\ \textsc{refactor}'(v|\text{Lib}), \text{ otherwise.} \end{cases}$$

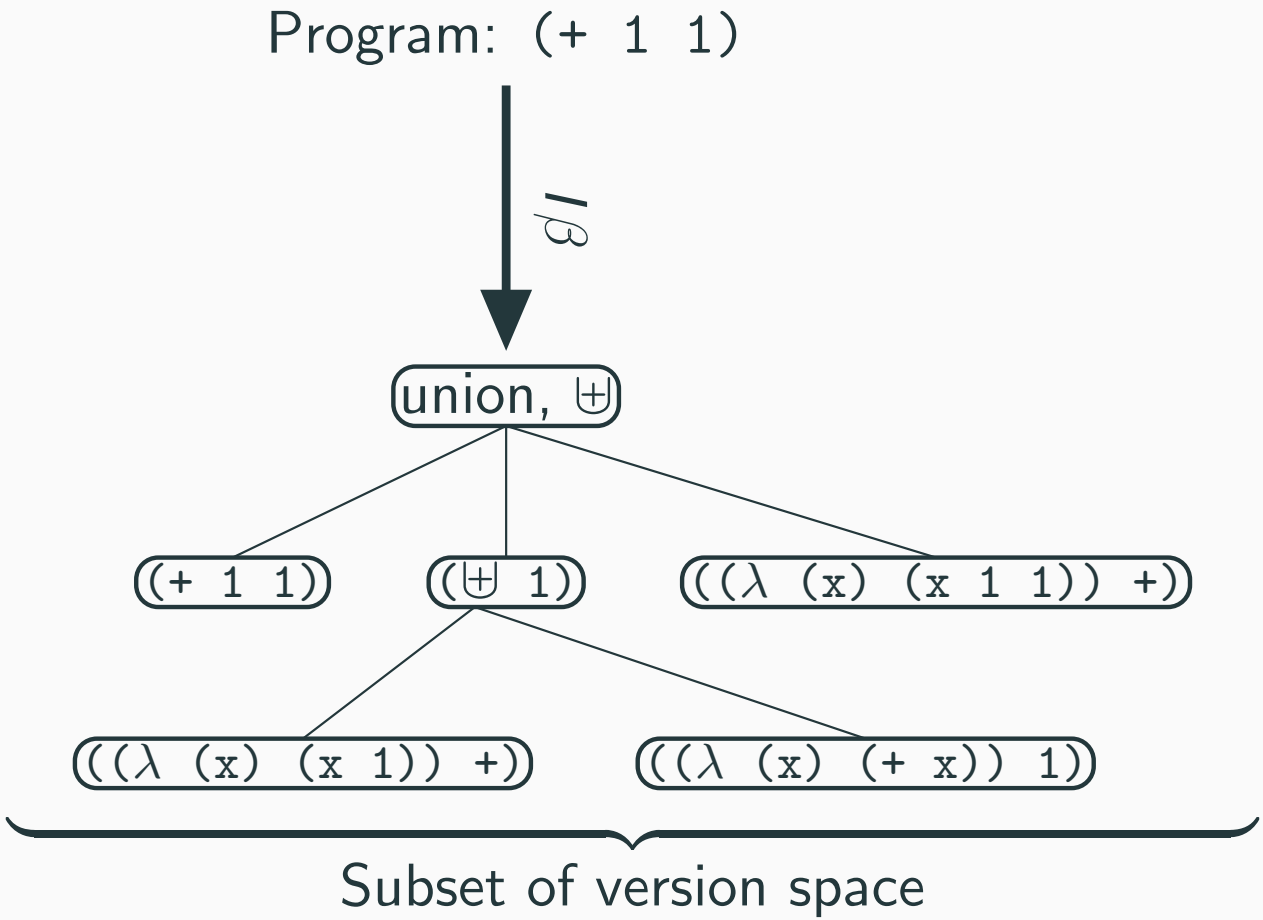
$$\textsc{refactor}'(v|\text{Lib}) = v, \text{ if } v \text{ is a primitive or variable}$$

$$\textsc{refactor}'(\lambda x.b|\text{Lib}) = \lambda x.\textsc{refactor}(b|\text{Lib})$$

$$\textsc{refactor}'(v_1\, v_2|\text{Lib}) = (\textsc{refactor}(v_1|\text{Lib})\ \ \textsc{refactor}(v_2|\text{Lib}))$$

Using version spaces

$$\begin{aligned} \text{VS} \rightarrow\ & \text{var} \mid \lambda\text{var.VS} \mid (\text{VS VS}) \mid \text{primitive} \\ \mid\ & \text{VS} \uplus \text{VS} && \textit{nondeterministic choice} \\ \mid\ & \Lambda && \textit{choose any expression} \\ \mid\ & \varnothing && \textit{choose no expression} \end{aligned}$$



Using version spaces

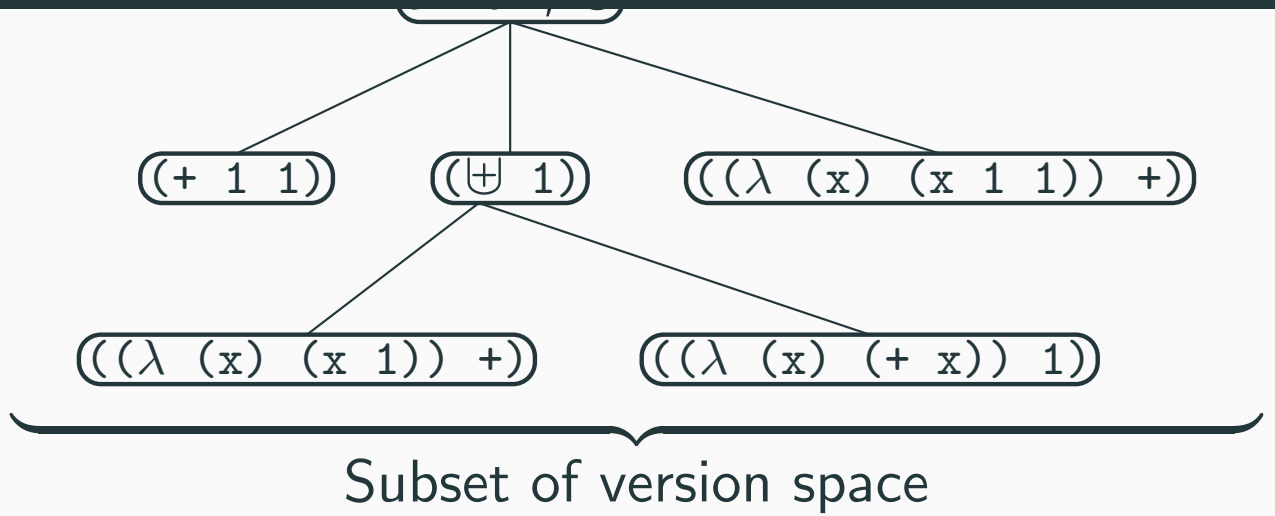
$$\begin{aligned} \text{VS} \rightarrow\ & \text{var} \mid \lambda\text{var.VS} \mid (\text{VS VS}) \mid \text{primitive} \\ \mid\ & \text{VS} \uplus \text{VS} && \textit{nondeterministic choice} \\ \mid\ & \Lambda && \textit{choose any expression} \\ \mid\ & \varnothing && \textit{choose no expression} \end{aligned}$$

completeness: $I\beta$ gets all the refactorings

let $v_2 = I\beta(v_1)$ and $e_1 \in [\![v_1]\!]$. for any $e_2 \longrightarrow e_1$ then $e_2 \in [\![v_2]\!]$

consistency: $I\beta$ only gets valid refactorings

let $v_2 = I\beta(v_1)$ and $e_2 \in [\![v_2]\!]$. then there is a $e_1 \in [\![v_1]\!]$ where $e_2 \longrightarrow e_1$



DreamCoder Domains

List Processing

Sum List

```
[1 2 3] → 6
[4 6 8 1] → 17
```

Double

```
[1 2 3 4] → [2 4 6 8]
[6 5 1] → [12 10 2]
```

Check Evens

```
[0 2 3] → [T T F]
[2 4 9 6] → [T T F T]
```

Text Editing

Abbreviate

```
Allen Newell → A.N.
Herb Simon → H.S.
```

Drop Last Characters

```
jabberwocky → jabberw
copycat → cop
```

Extract

```
see spot(run) → run
a (bee) see → bee
```

Regexes

Phone Numbers

```
(555) 867-5309
(650) 555-2368
```

Currency

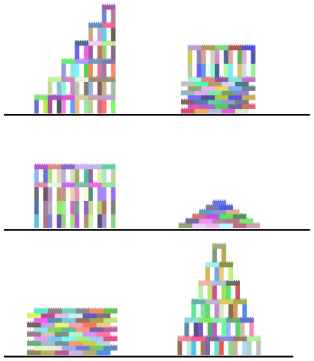
```
$100.25
$4.50
```

Dates

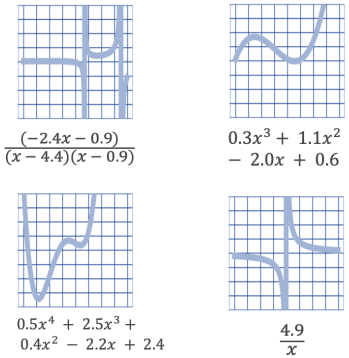
```
Y1775/0704
Y2000/0101
```

LOGO Graphics

Block Towers

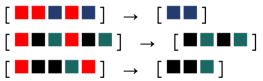


Symbolic Regression



Recursive Programming

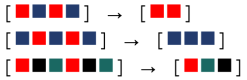
Filter



Index List

Length

Every Other



Physics

$$KE = \frac{1}{2}m|\vec{v}|^2$$

$$\vec{a} = \frac{1}{m}\sum_i \vec{F}_i$$

$$\vec{F} \propto \frac{q_1 q_2}{|\vec{r_1} - \vec{r_2}|^2}\widehat{r_1 - r_2}$$

$$R_{total} = \left(\Sigma_i \frac{1}{R_i}\right)^{-1}$$

DreamCoder Domains

List Processing

Sum List

[1 2 3] → 6
[4 6 8 1] → 17

Double

[1 2 3 4] → [2 4 6 8]
[6 5 1] → [12 10 2]

Check Evens

[0 2 3] → [T T F]
[2 4 9 6] → [T T F T]

Text Editing

Abbreviate

Allen Newell → A.N.
Herb Simon → H.S.

Drop Last Characters

jabberwocky → jabberw
copycat → cop

Extract

see spot(run) → run
a (bee) see → bee

Regexes

Phone Numbers

(555) 867-5309
(650) 555-2368

Currency

$100.25
$4.50

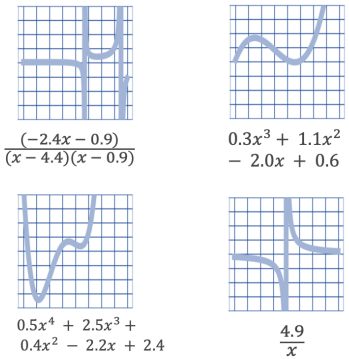
Dates

Y1775/0704
Y2000/0101

LOGO Graphics

Block Towers

Symbolic Regression



Recursive Programming

Filter

[ ] → [ ]
[ ] → [ ]
[ ] → [ ]

Index List

0, [ ] →
1, [ ] →
1, [ ] →

Length

[ ] → 4
[ ] → 6
[ ] → 3

Every Other

[ ] → [ ]
[ ] → [ ]
[ ] → [ ]

Physics

$$KE = \frac{1}{2}m|\vec{v}|^2$$

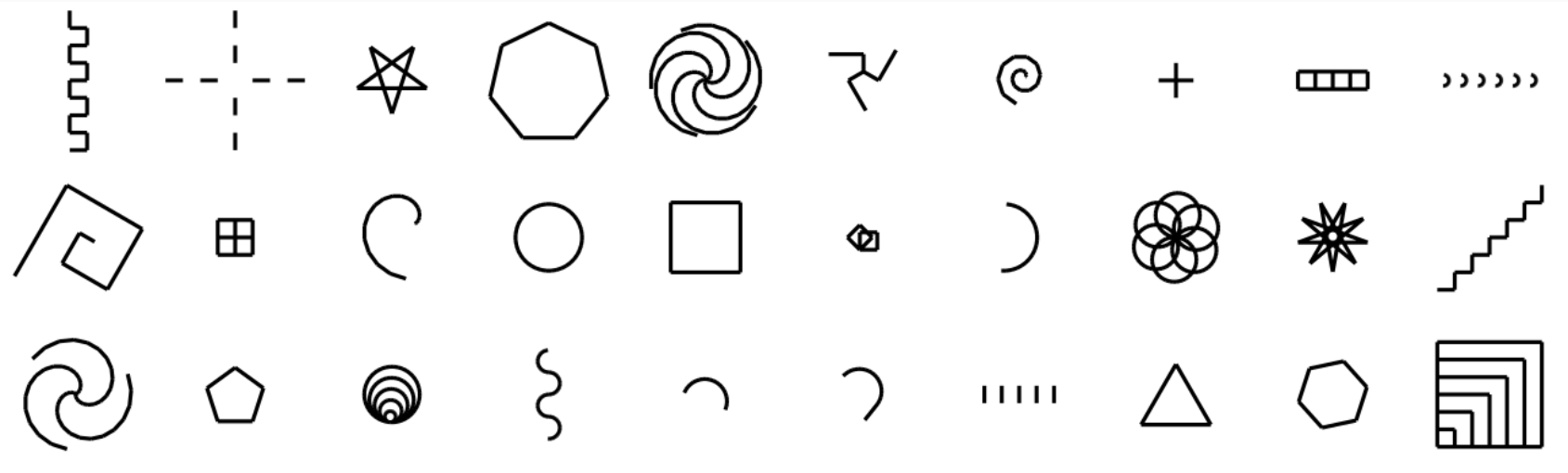
$$\vec{a} = \frac{1}{m}\sum_i \vec{F}_i$$

$$\vec{F} \propto \frac{q_1 q_2}{|\vec{r_1} - \vec{r_2}|^2}\widehat{r_1 - r_2}$$

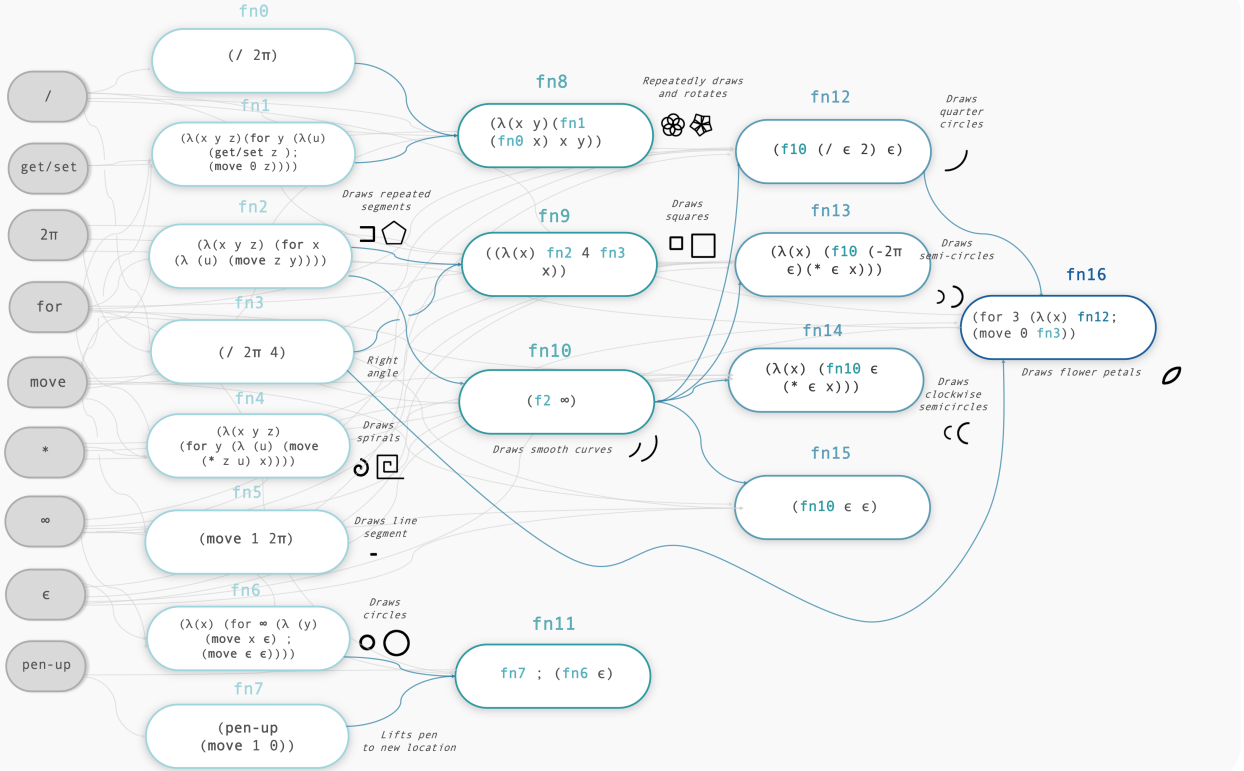
$$R_{total} = \left(\Sigma_i \frac{1}{R_i}\right)^{-1}$$

LOGO Graphics

30 out of 160 tasks



LOGO Graphics – learning interpretable library of concepts



```
(fn8 5 (fn4 (* ϵ 2) ∞ ϵ))
```

```
(for 2 (λ (x)
(fn14 2); (fn13 2)))
```

```
(for 7 (λ (x) (fn9 x)))
```

```
(fn10 (* (fn0 7) 3) 3)
```

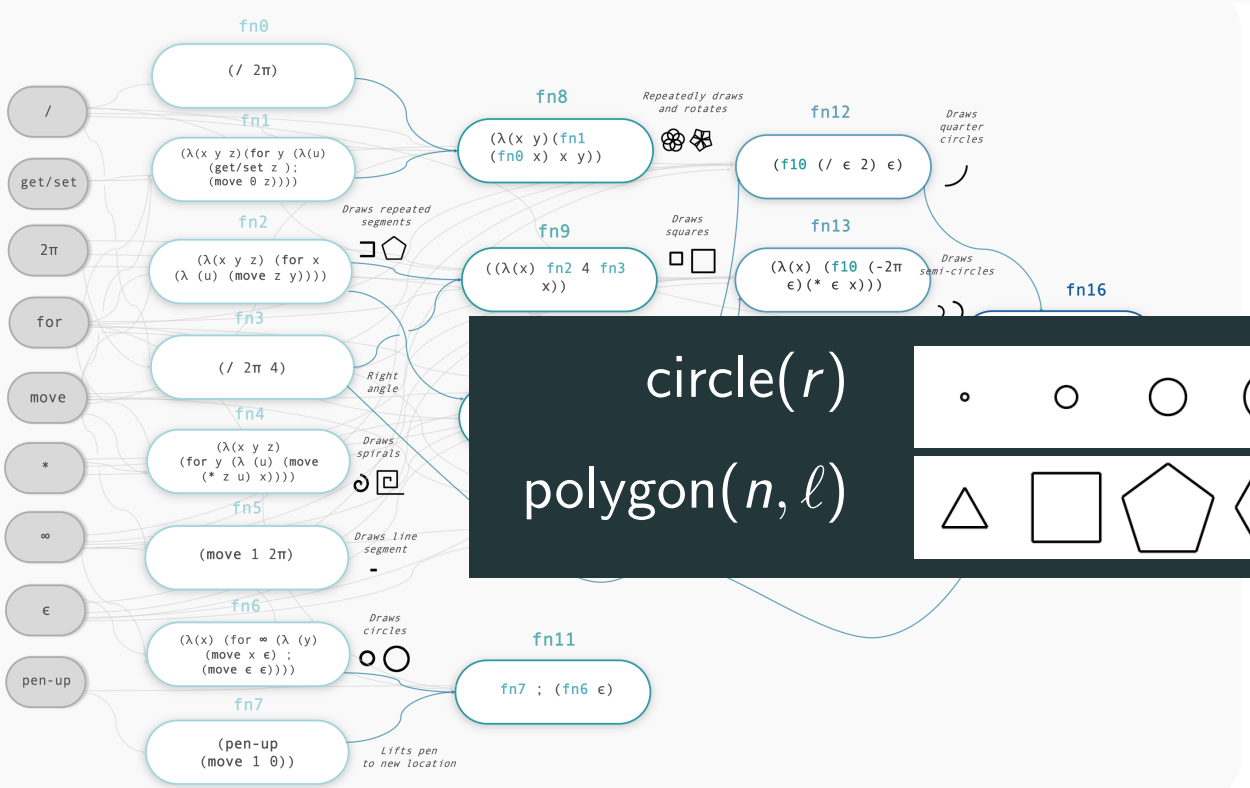
```
(fn8  6 (fn7 ; fn5 ;
fn7 ; fn5))
```

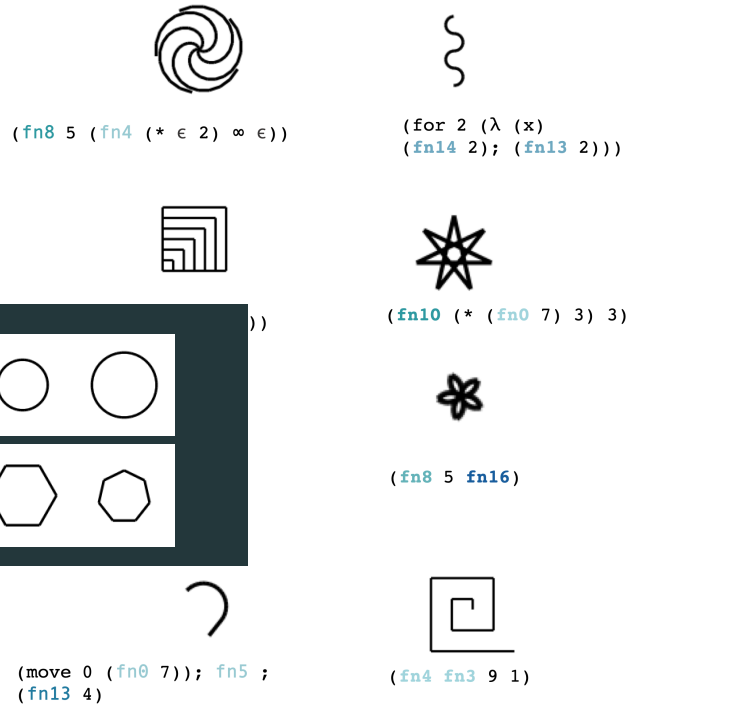
```
(fn8 5 fn16)
```

```
(move 0 (fn0 7)); fn5 ;
(fn13 4)
```

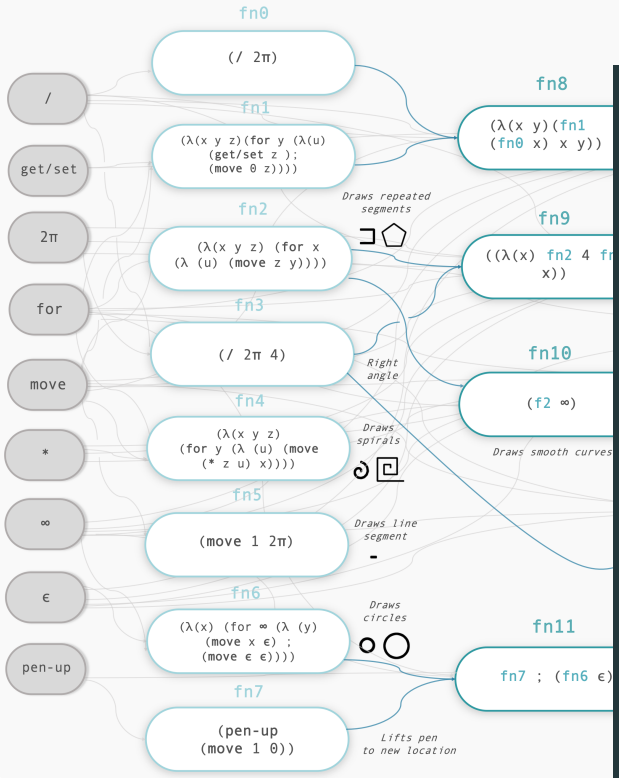
```
(fn4 fn3 9 1)
```

LOGO Graphics – learning interpretable library of concepts

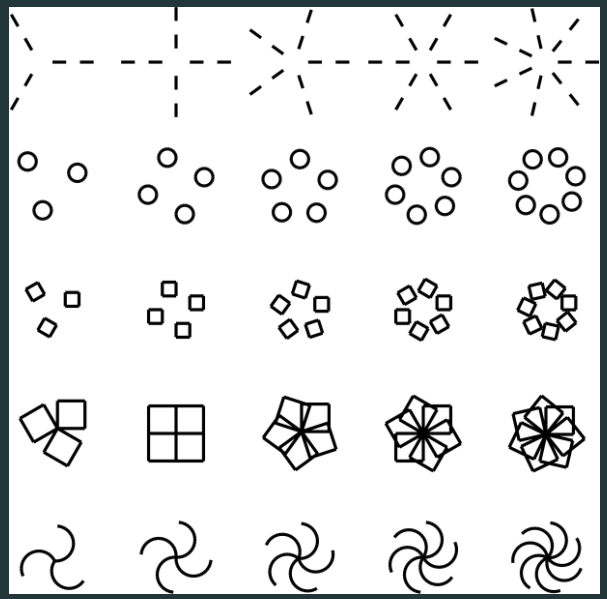


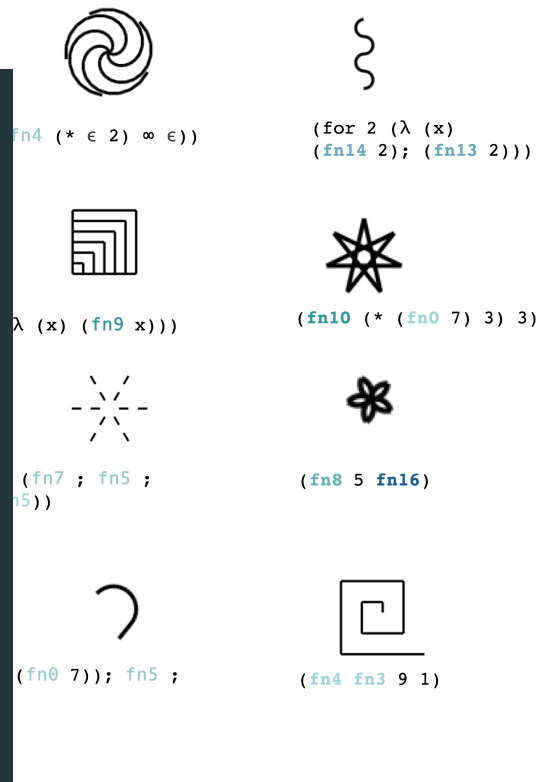


LOGO Graphics – learning interpretable library of concepts



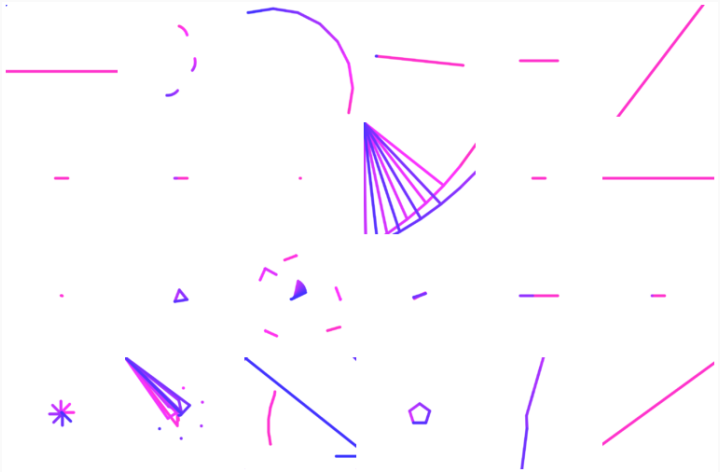
radial symmetry(n, body)



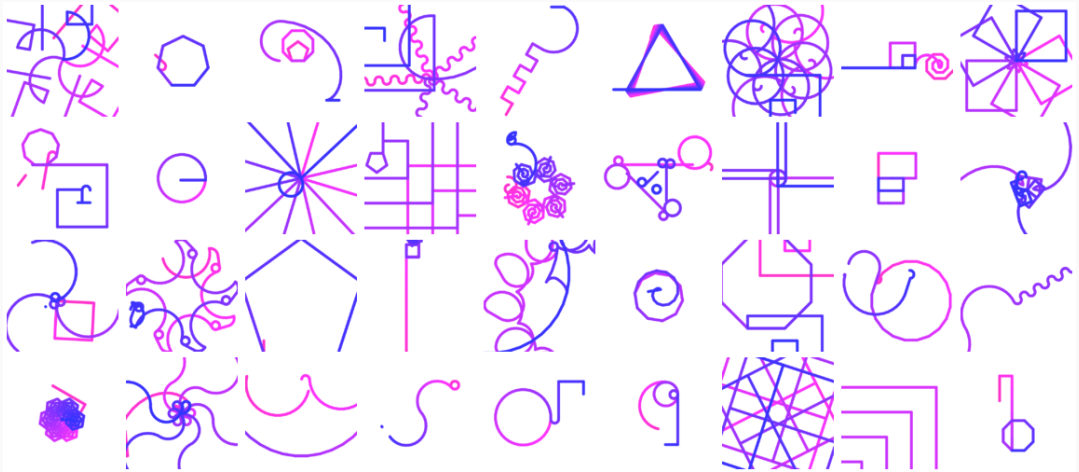


what does DreamCoder dream of?

before learning

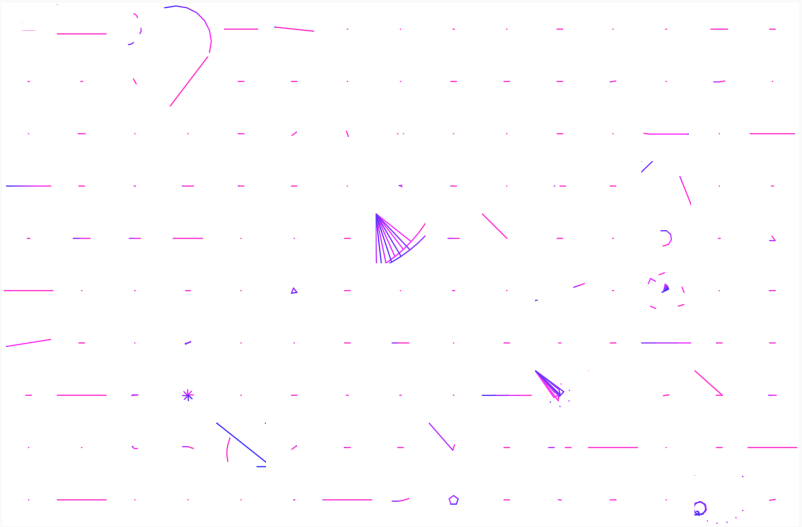


after learning

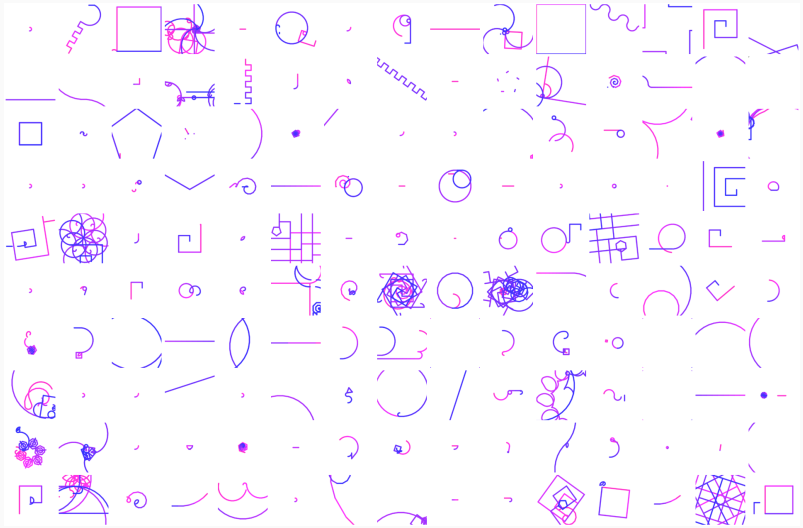


what does DreamCoder dream of?

before learning

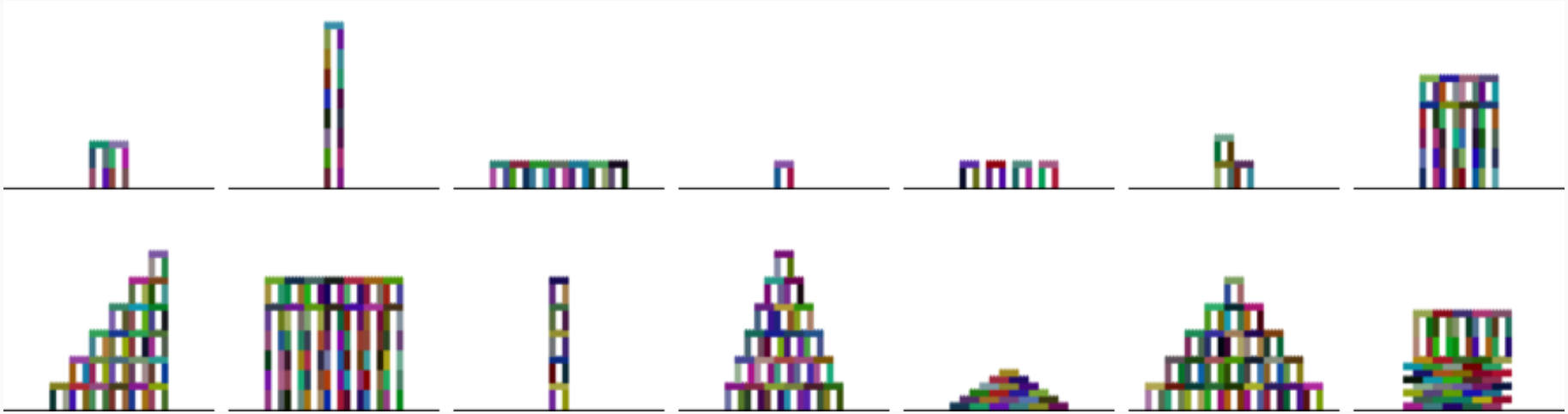


after learning



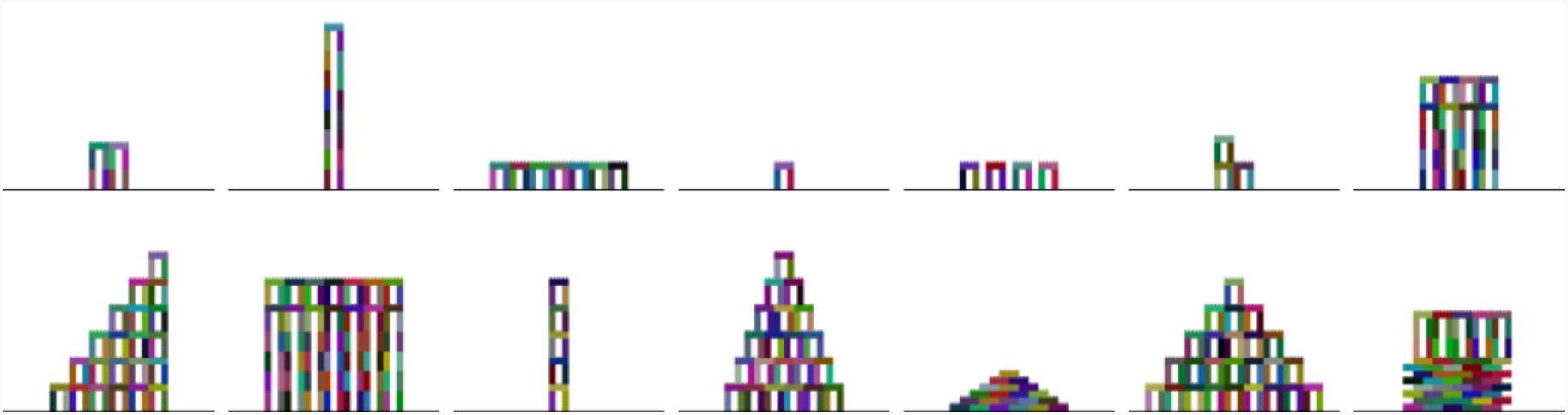
Planning to build towers

example tasks (112 total)



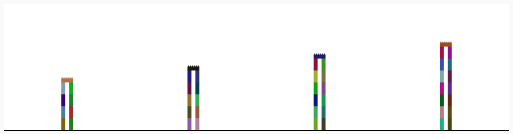
Planning to build towers

example tasks (112 total)

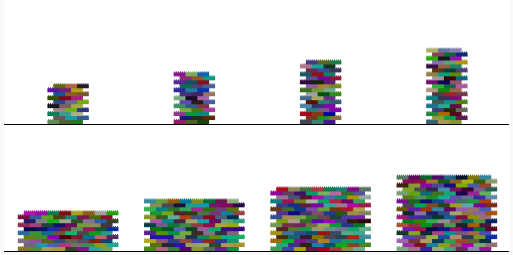


learned library routines ($\approx$ 20 total)

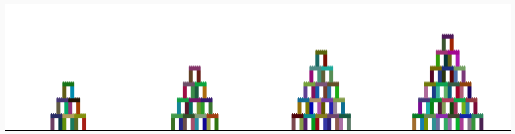
arch(h)



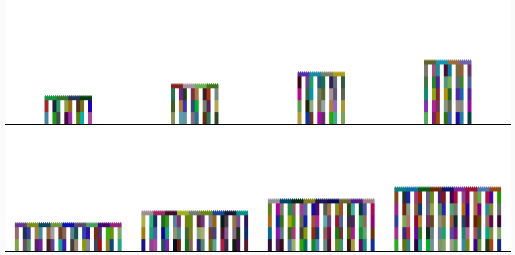
wall(w, h)



pyramid(h)

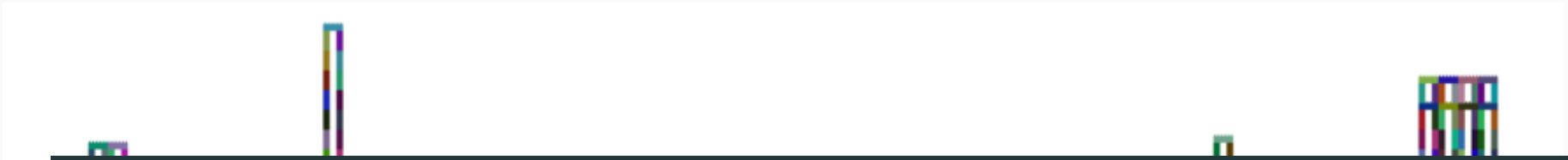


bridge(w, h)



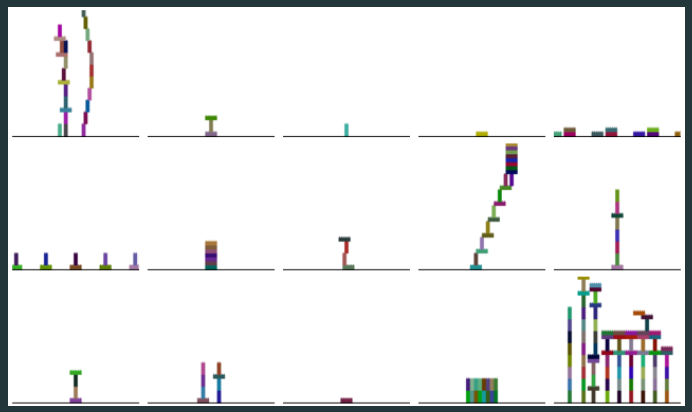
Planning to build towers

example tasks (112 total)

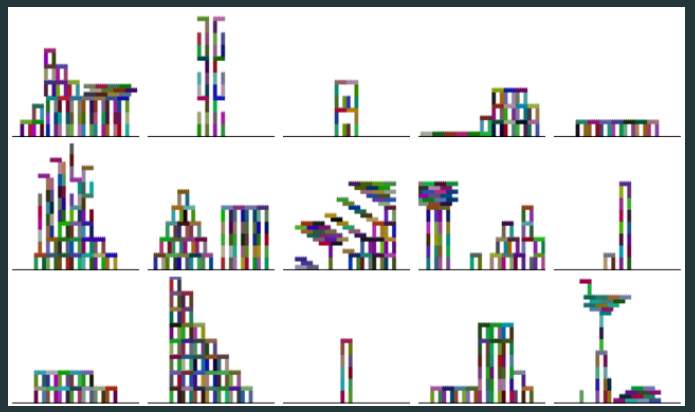


lea

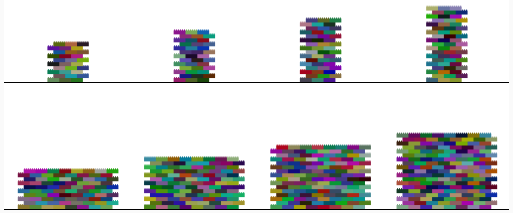
dreams before learning



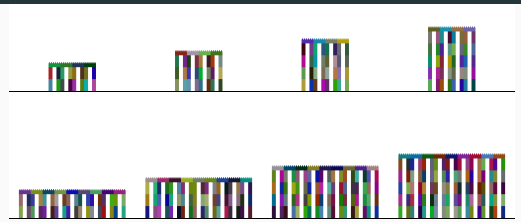
dreams after learning



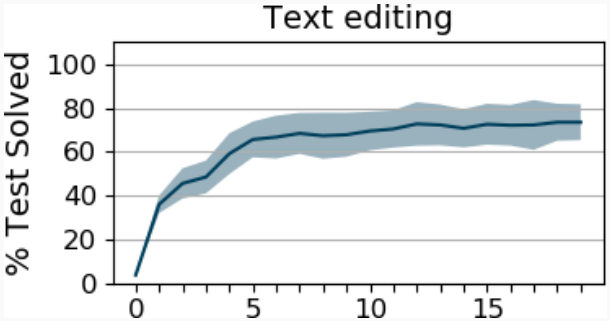
wall(w, h)

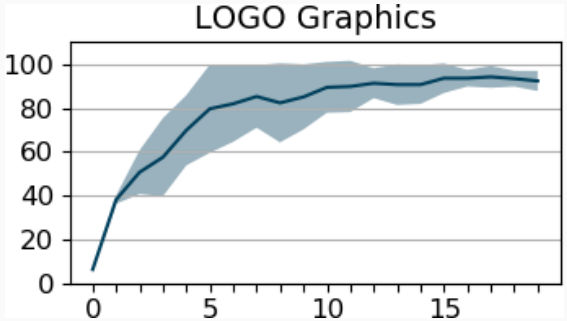


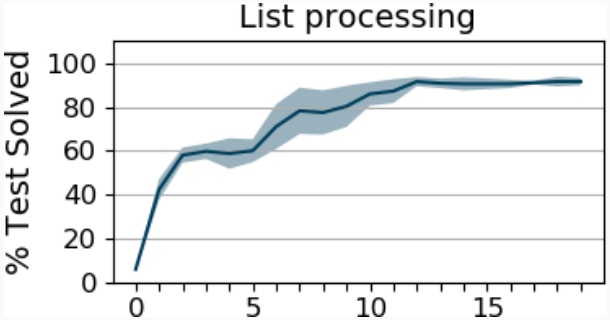
bridge(w, h)

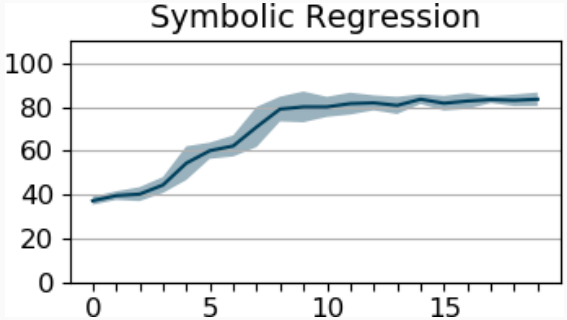


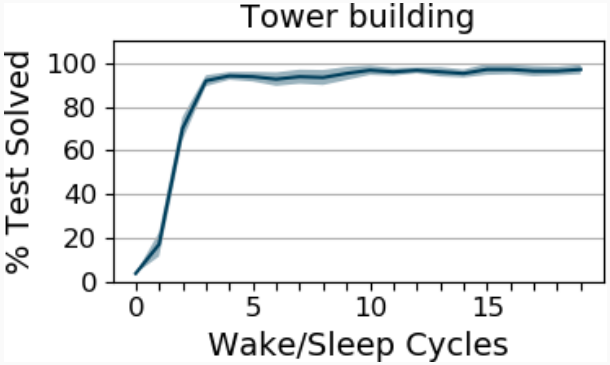
synergy between dreaming and library learning

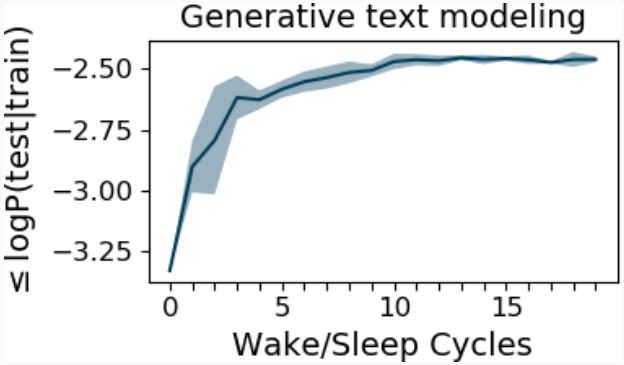


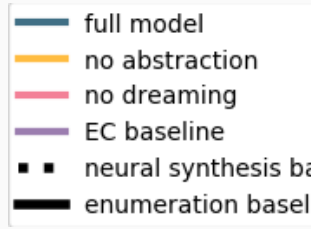




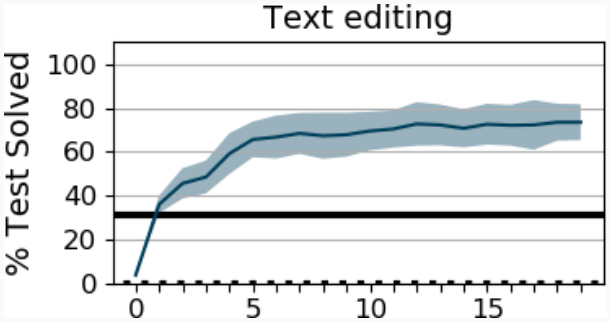


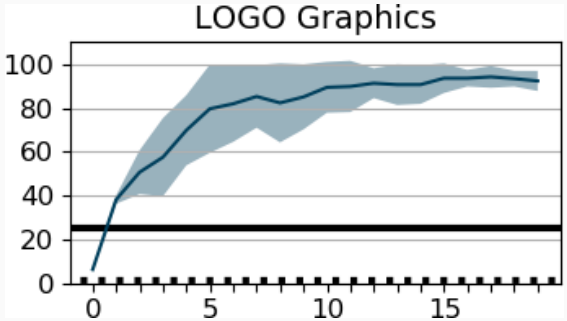


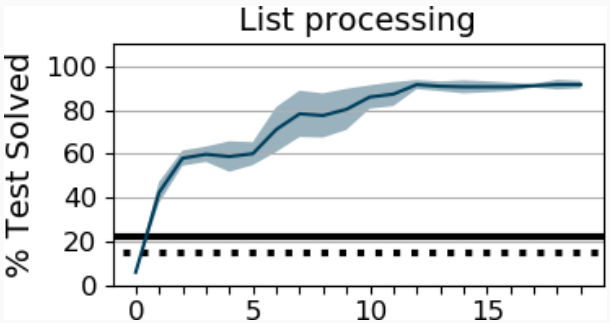


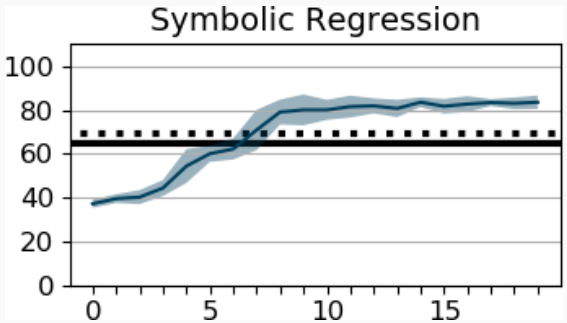


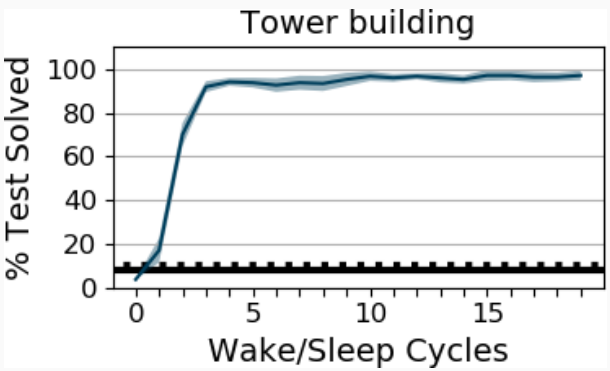
synergy between dreaming and library learning

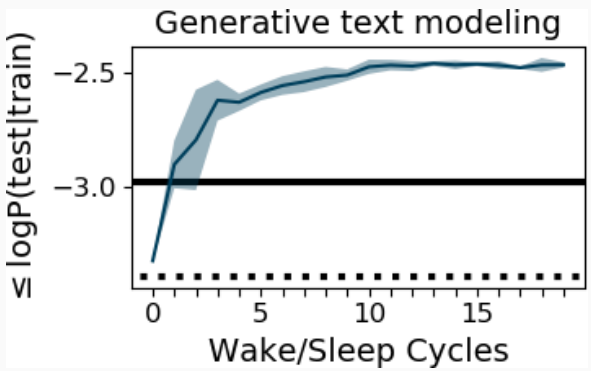


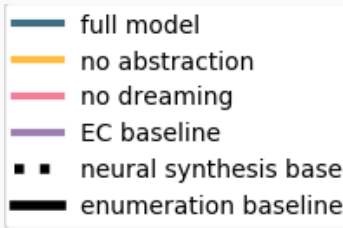




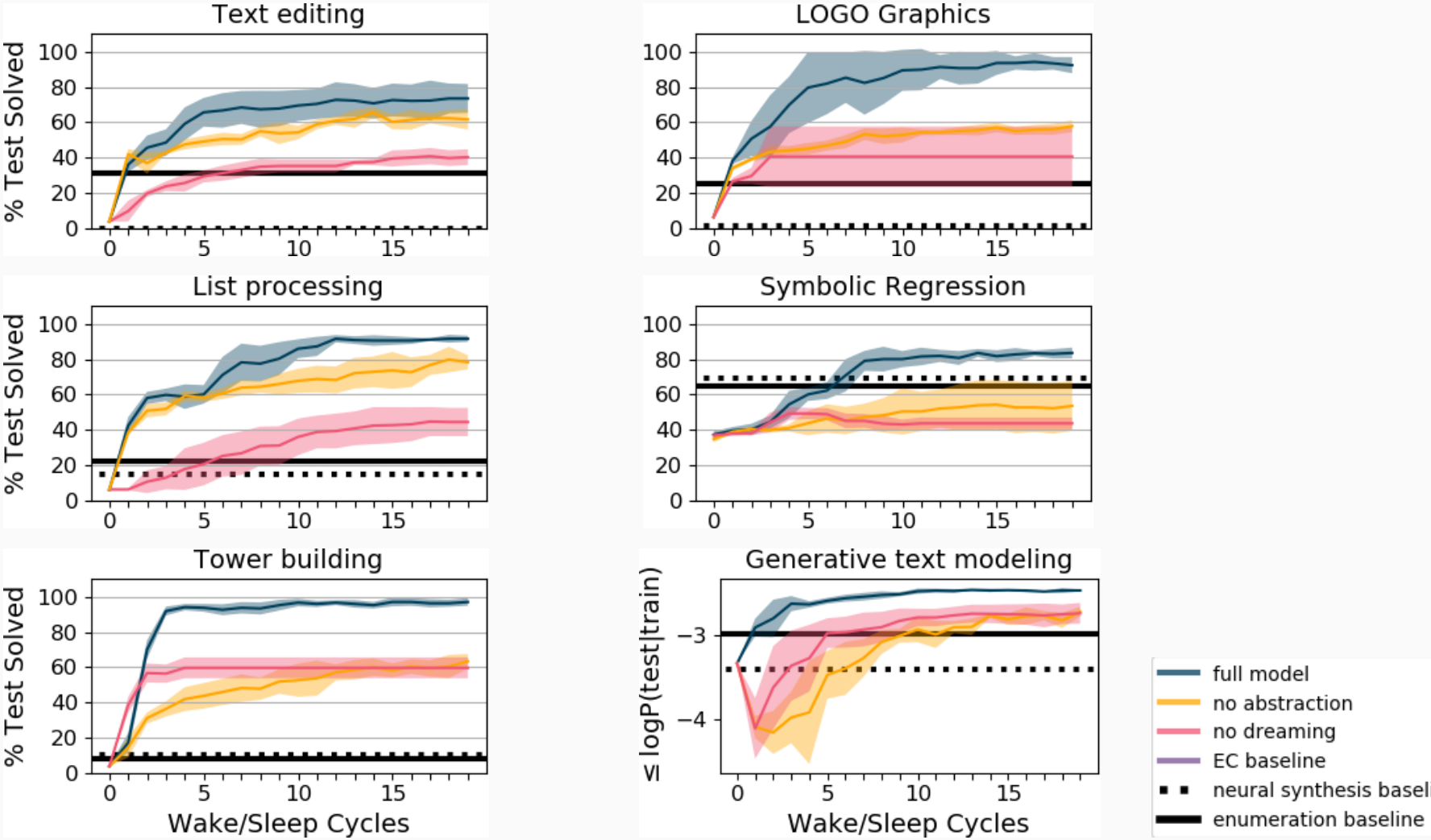




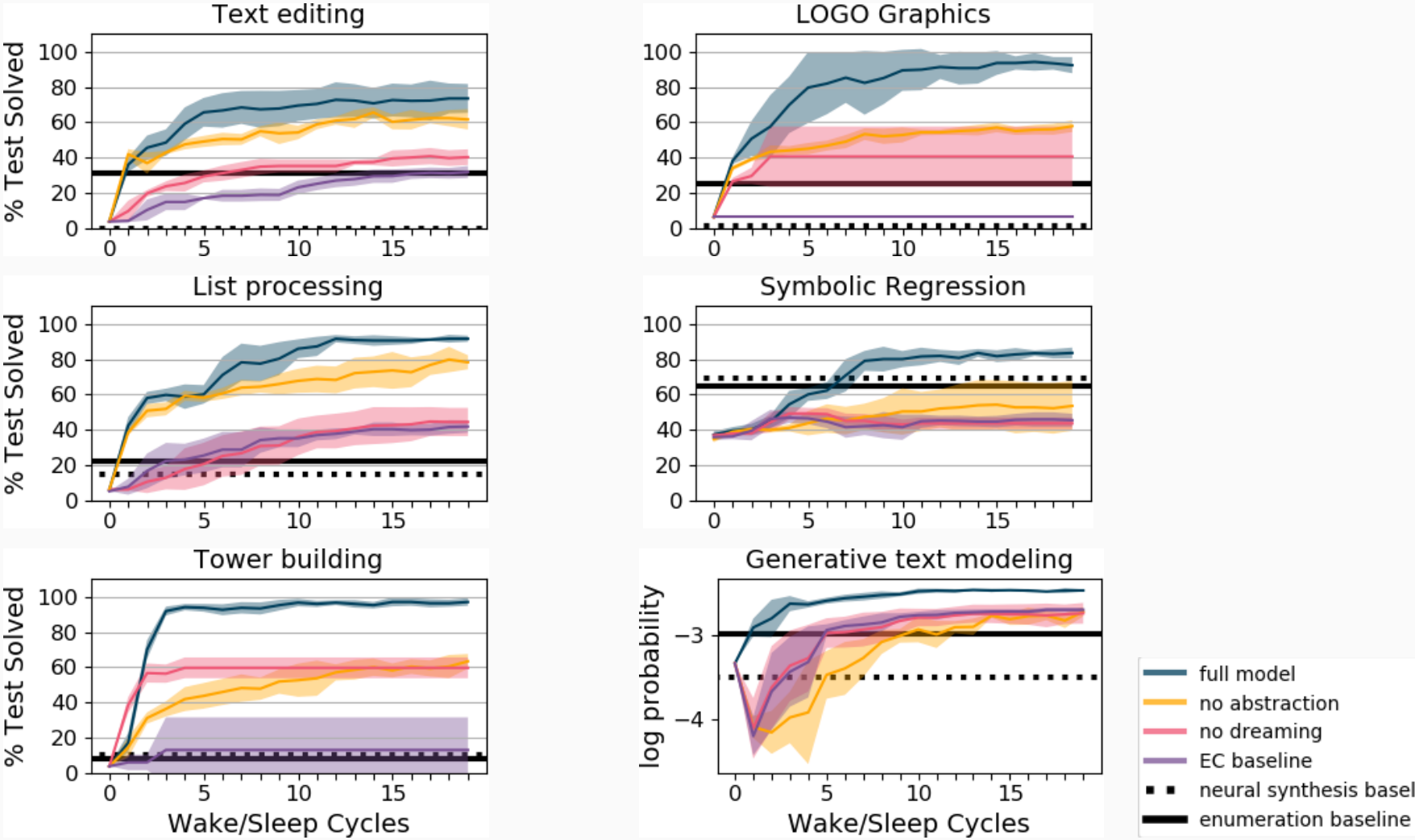




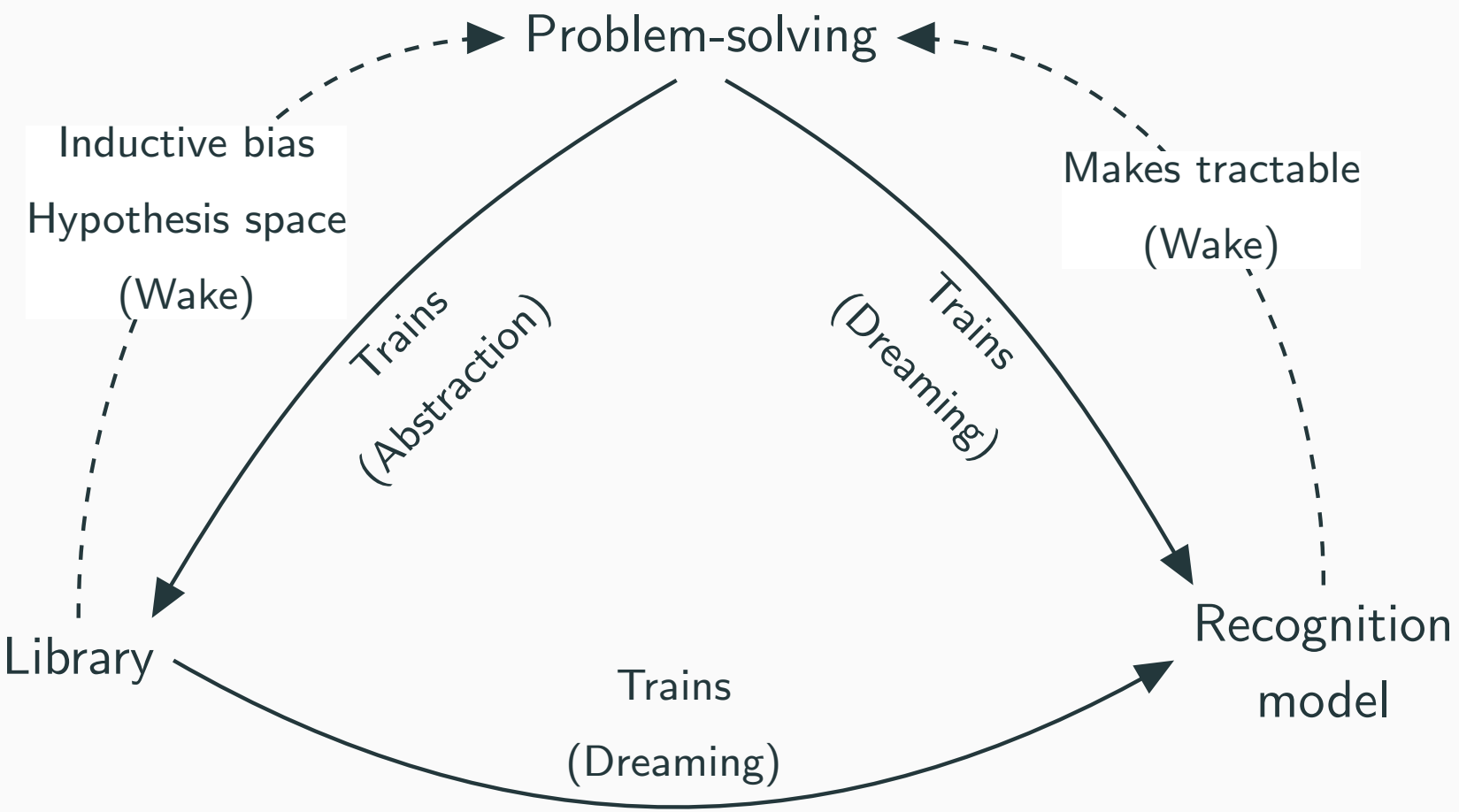
synergy between dreaming and library learning



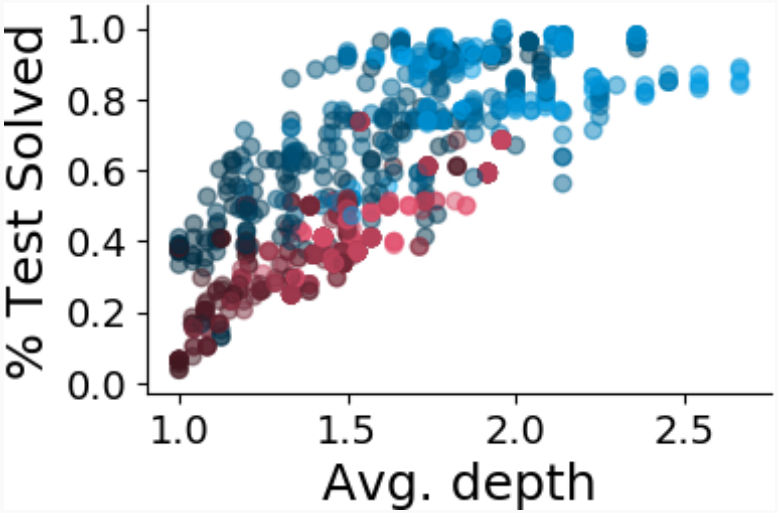
synergy between dreaming and library learning

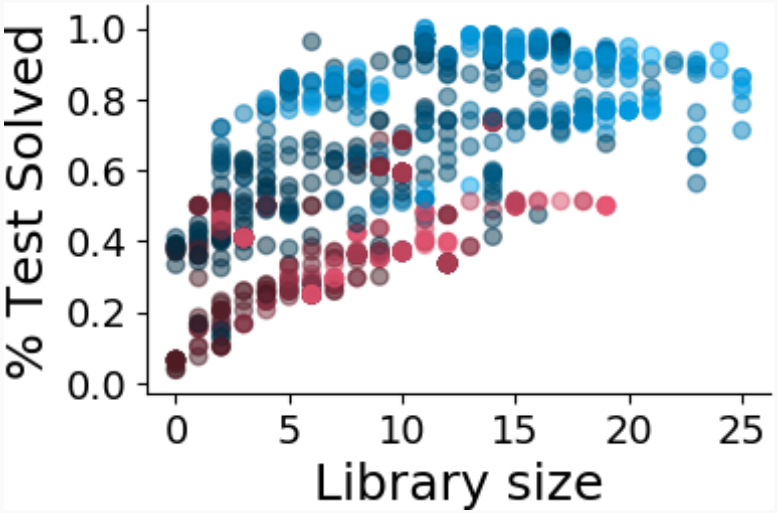


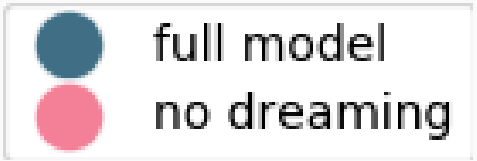
synergy between dreaming and library learning



Evidence for dreaming bootstrapping better libraries







Dark→Light: Early in learning→Later in learning

Making your own problems

DreamCoder dreams randomly – but humans construct their own problems in decidedly nonrandom ways

What makes a good problem?

Making your own problems

DreamCoder dreams randomly – but humans construct their own problems in decidedly nonrandom ways

What makes a good problem?

Idea: Pareto optimality, a good problem should jointly optimize multiple objectives that are in tension

Making your own problems

DreamCoder dreams randomly – but humans construct their own problems in decidedly nonrandom ways

What makes a good problem?

Idea: Pareto optimality, a good problem should jointly optimize multiple objectives that are in tension

"Surprisingly stable for how tall it is"

"Surprisingly fast for how small it is"

"Surprisingly prickly for how sticky it is"

"Surprisingly simple for how complex it seems at first glance"

Problems with surprisingly simple solutions

Pareto frontier of:

$$(\underbrace{\text{size of program}}_{\text{make it small}}, \underbrace{\text{probability of program under neural network}}_{\text{make it unlikely}})$$

Problems with surprisingly simple solutions

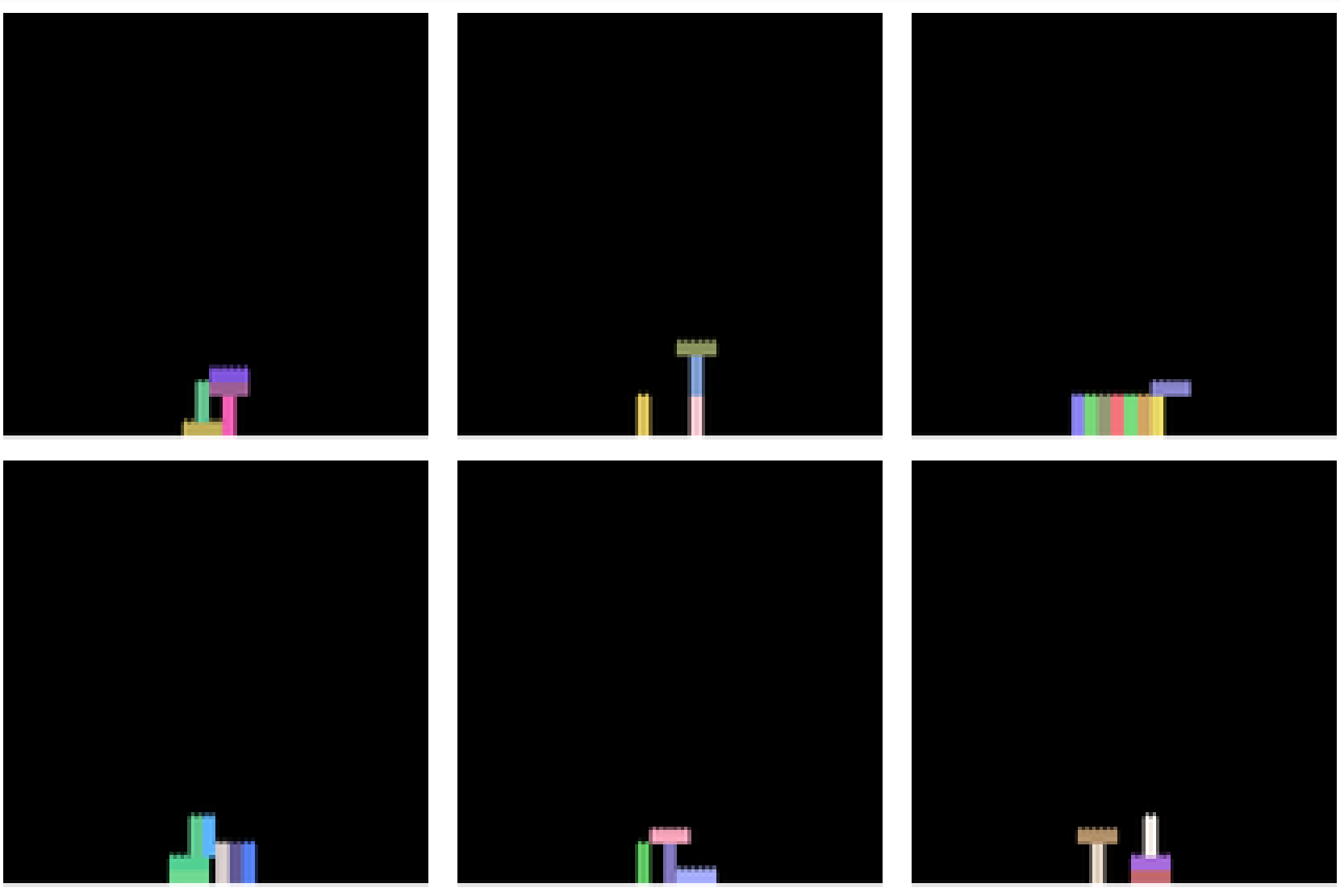
Pareto frontier of:

(size of program, probability of program under neural network)

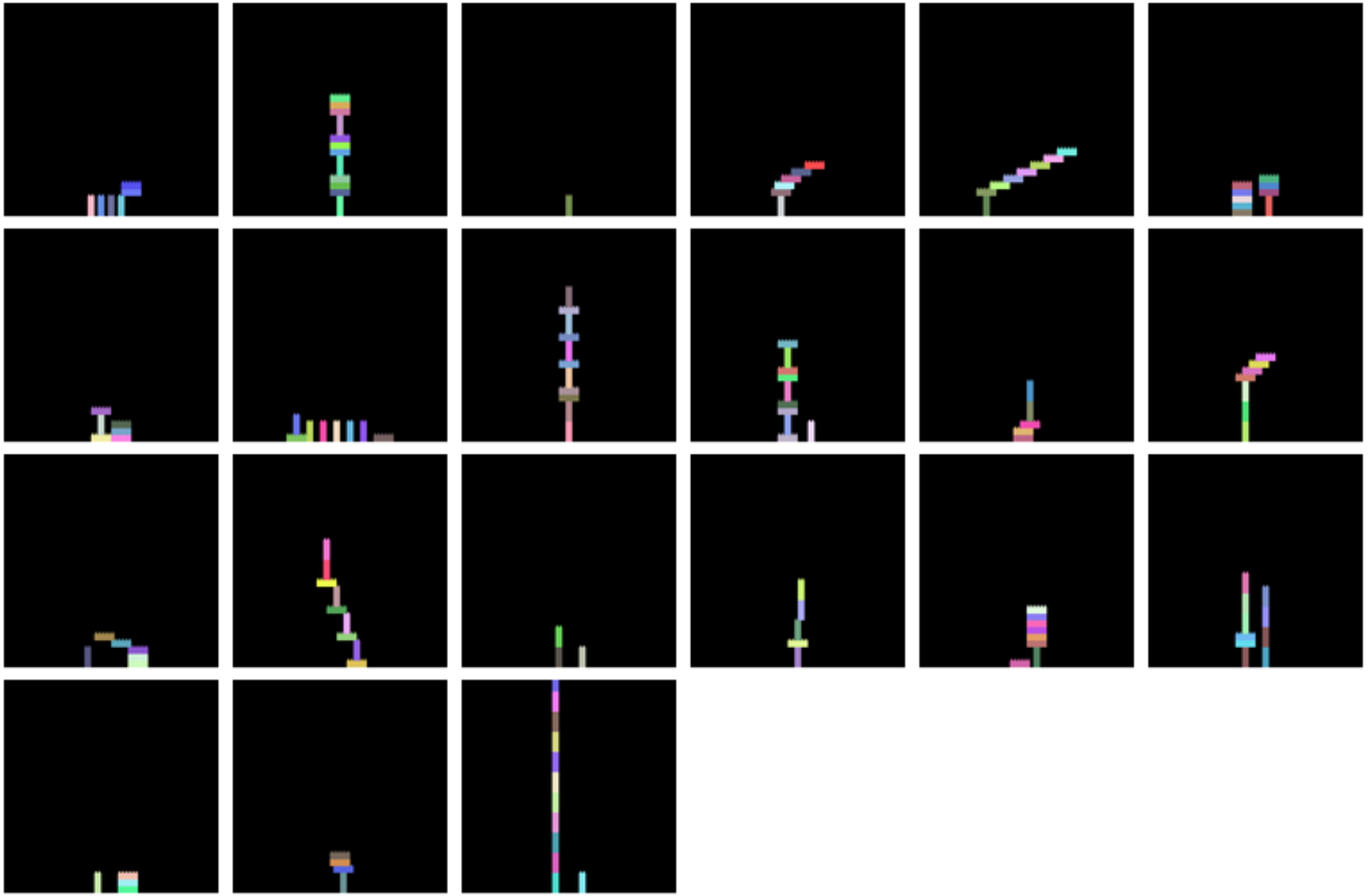
make it small — make it unlikely

intuition: geometric art might be like this

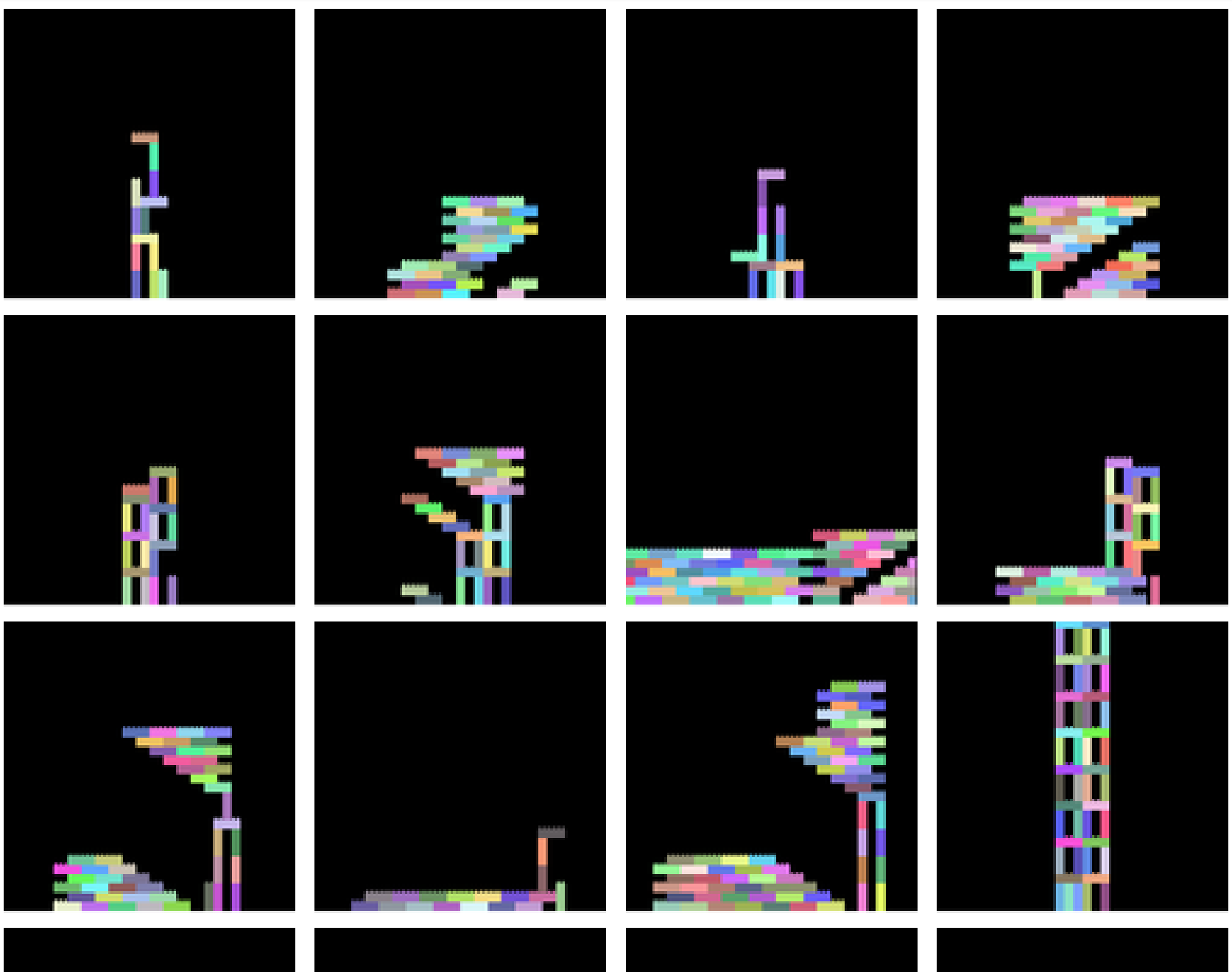
Having DreamCoder make up its own towers



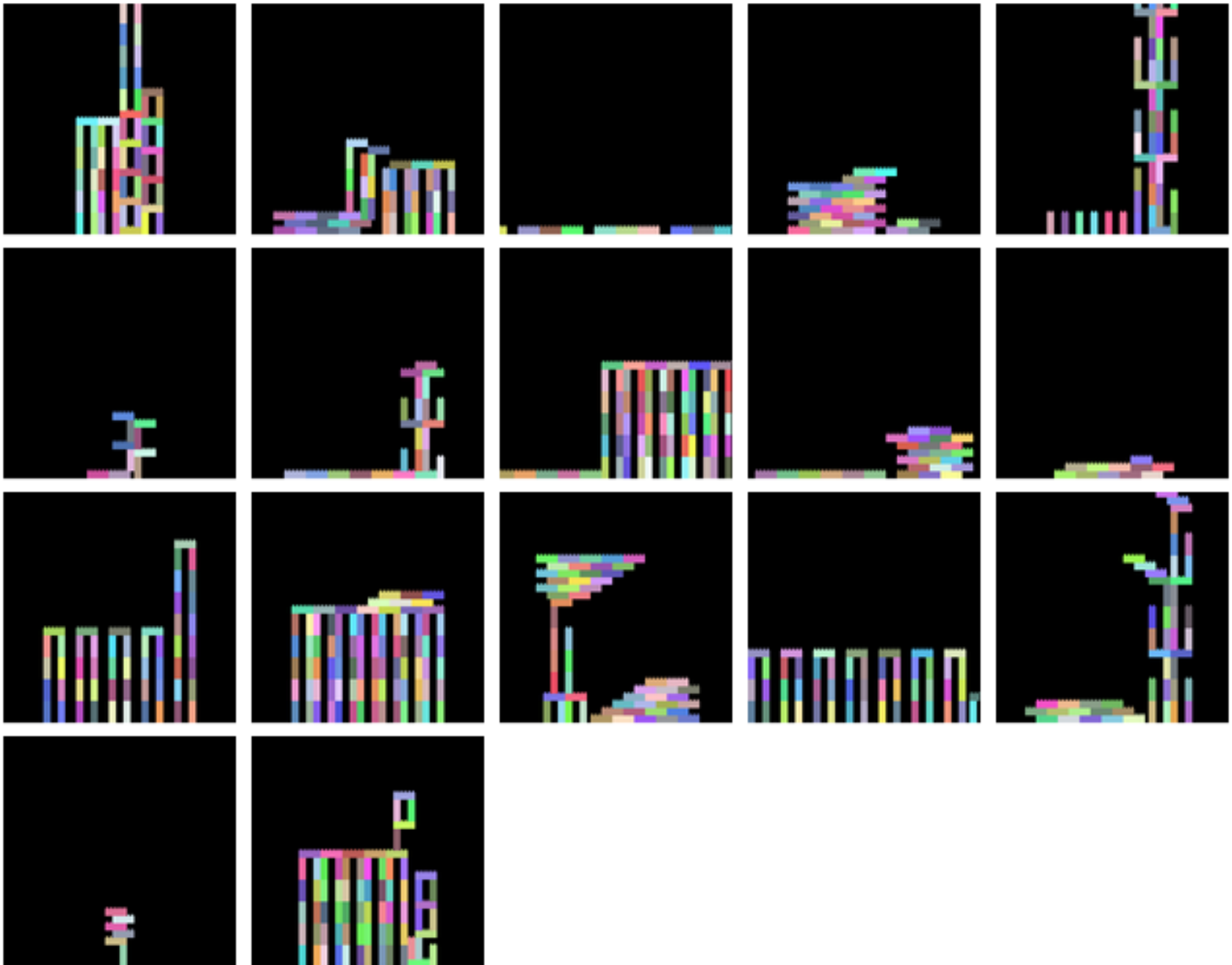
Having DreamCoder make up its own towers



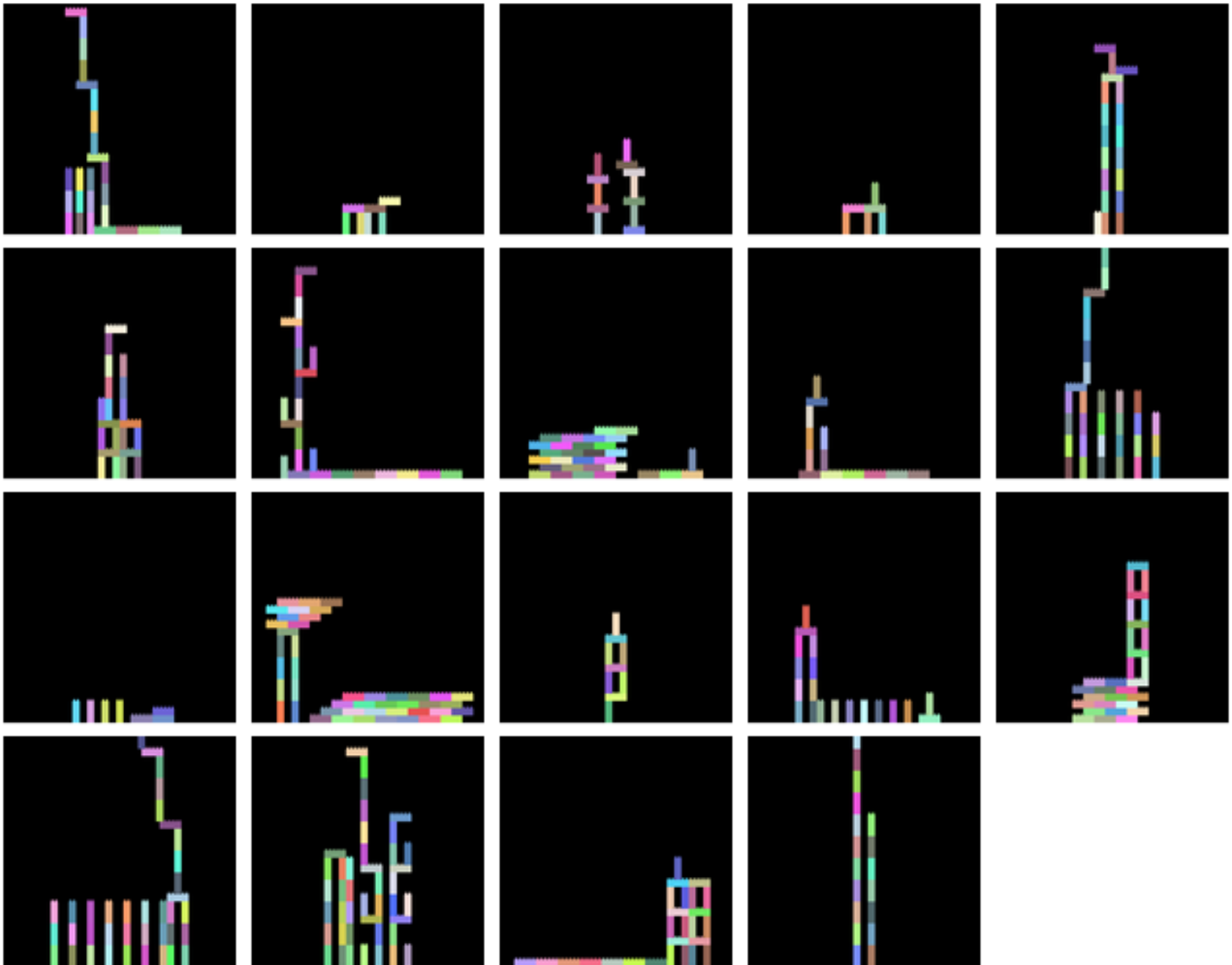
Having DreamCoder make up its own towers



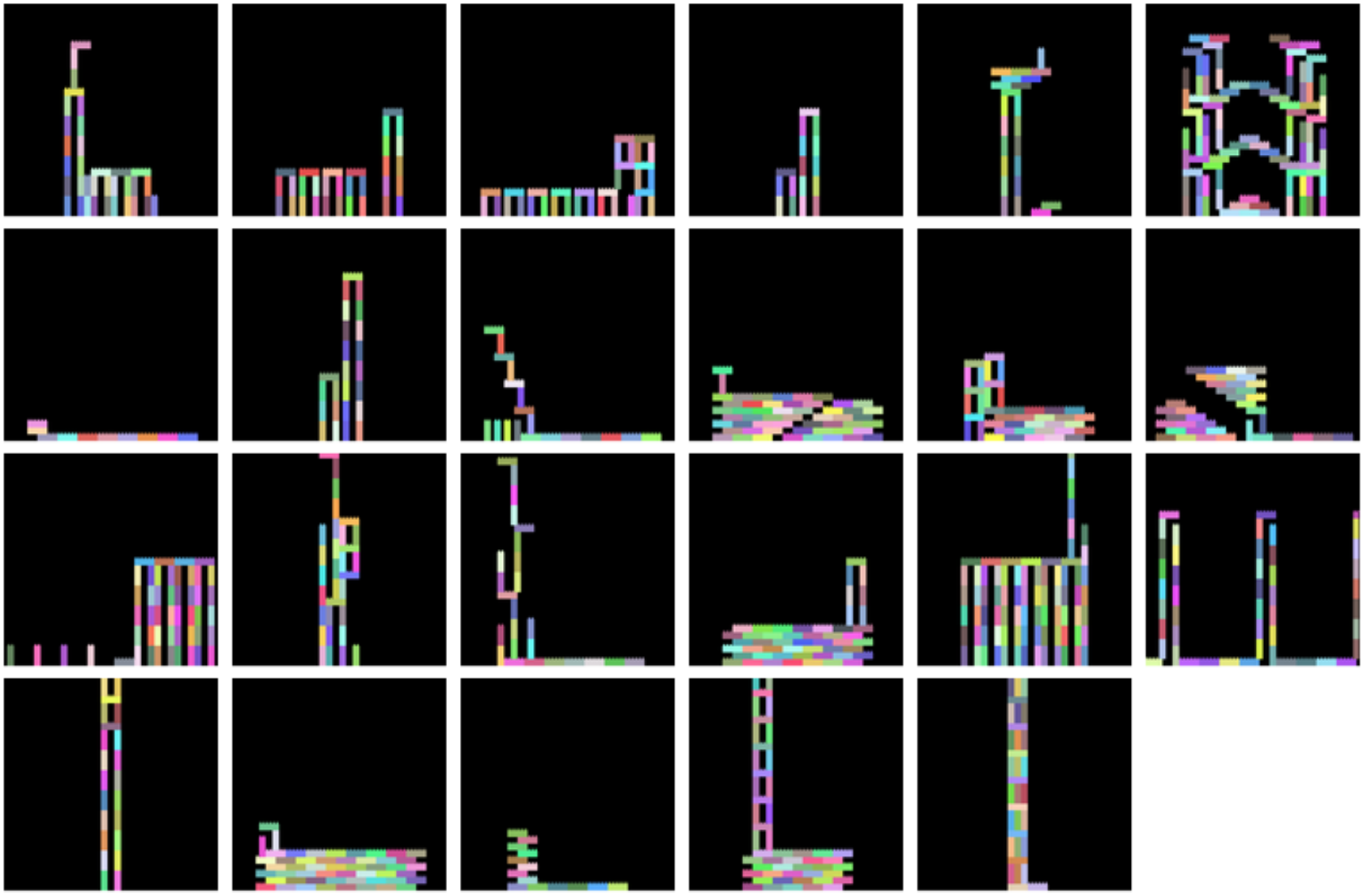
Having DreamCoder make up its own towers



Having DreamCoder make up its own towers

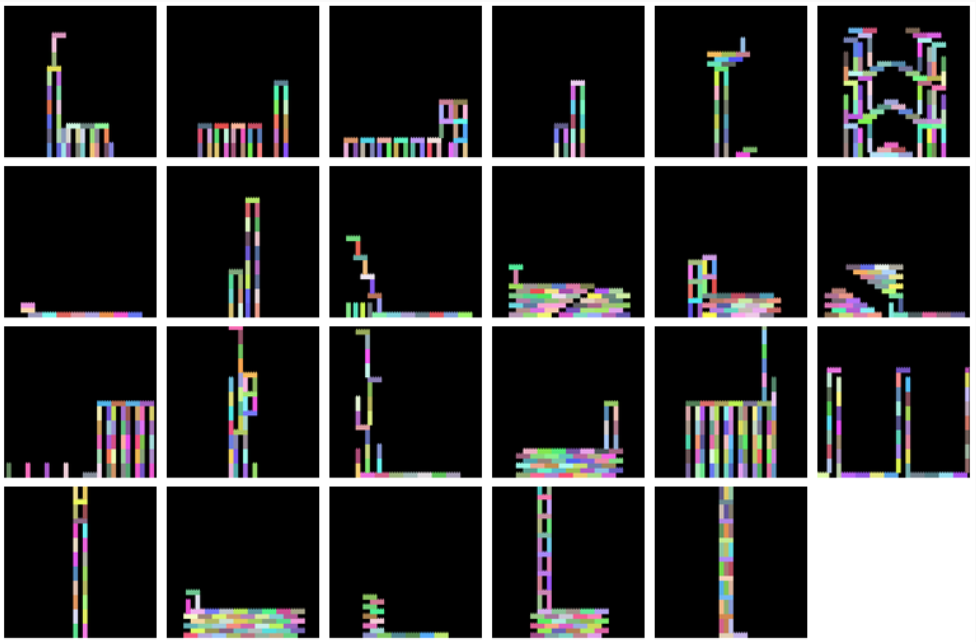


Having DreamCoder make up its own towers

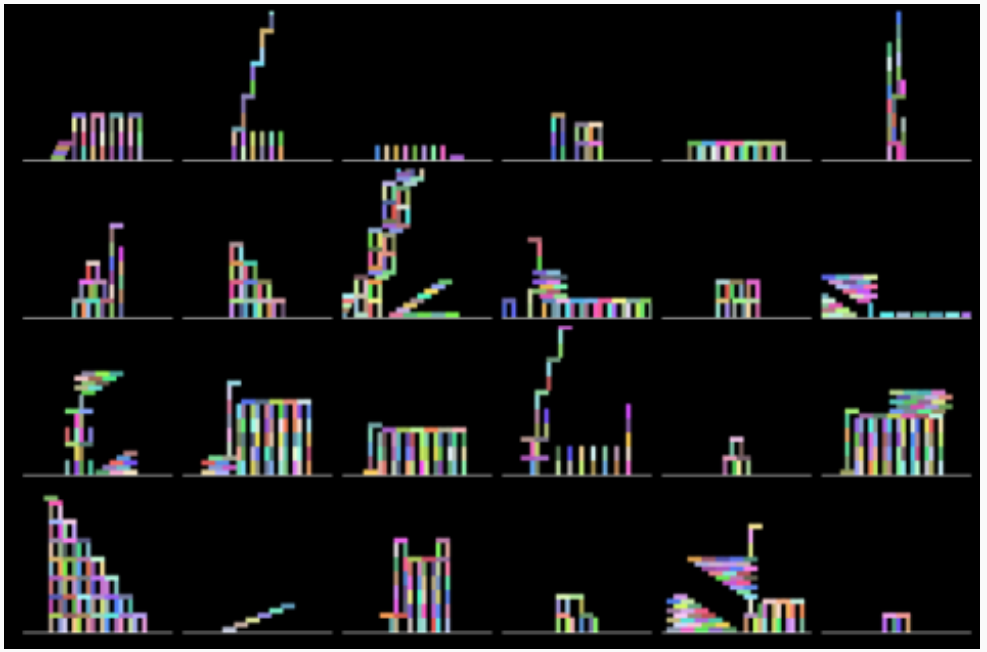


Comparison to dreams

"playful" generation



"dream" generation



the Schmidhuber universe

POWERPLAY

Idea: bootstrap a general problem solver "out of thin air"

POWERPLAY

Idea: bootstrap a general problem solver "out of thin air"

POWERPLAY Algorithm:

1. start with a problem-solver (neural net, program, ...)
2. initialize a set of problems. starts out empty

POWERPLAY

Idea: bootstrap a general problem solver "out of thin air"

POWERPLAY Algorithm:

1. start with a problem-solver (neural net, program, ...)

2. initialize a set of problems. starts out empty

3. search for a (problem, problem-solver') pair such that:

(i) the current problem-solver cannot solve the problem

(ii) BUT, problem-solver' *can* solve the problem

(iii) AND, problem-solver' solves everything else in the set of problems

POWERPLAY

Idea: bootstrap a general problem solver "out of thin air"

POWERPLAY Algorithm:

1. start with a problem-solver (neural net, program, ...)

2. initialize a set of problems. starts out empty

3. search for a (problem, problem-solver') pair such that:

(i) the current problem-solver cannot solve the problem

(ii) BUT, problem-solver' *can* solve the problem

(iii) AND, problem-solver' solves everything else in the set of problems

4. update the problem-solver to problem-solver'; add the problem to the set of problems; go back to step (3)

POWERPLAY

Idea: bootstrap a general problem solver "out of thin air"

POWERPLAY Algorithm:

1. start with a problem-solver (neural net, program, ...)

2. initialize a set of problems. starts out empty

Important point:
a "new problem" can also be just solving old problems more efficiently

(i) the current problem-solver cannot solve the problem

(ii) BUT, problem-solver' *can* solve the problem

(iii) AND, problem-solver' solves everything else in the set of problems

4. update the problem-solver to problem-solver'; add the problem to the set of problems; go back to step (3)

POWERPLAY on a recurrent network

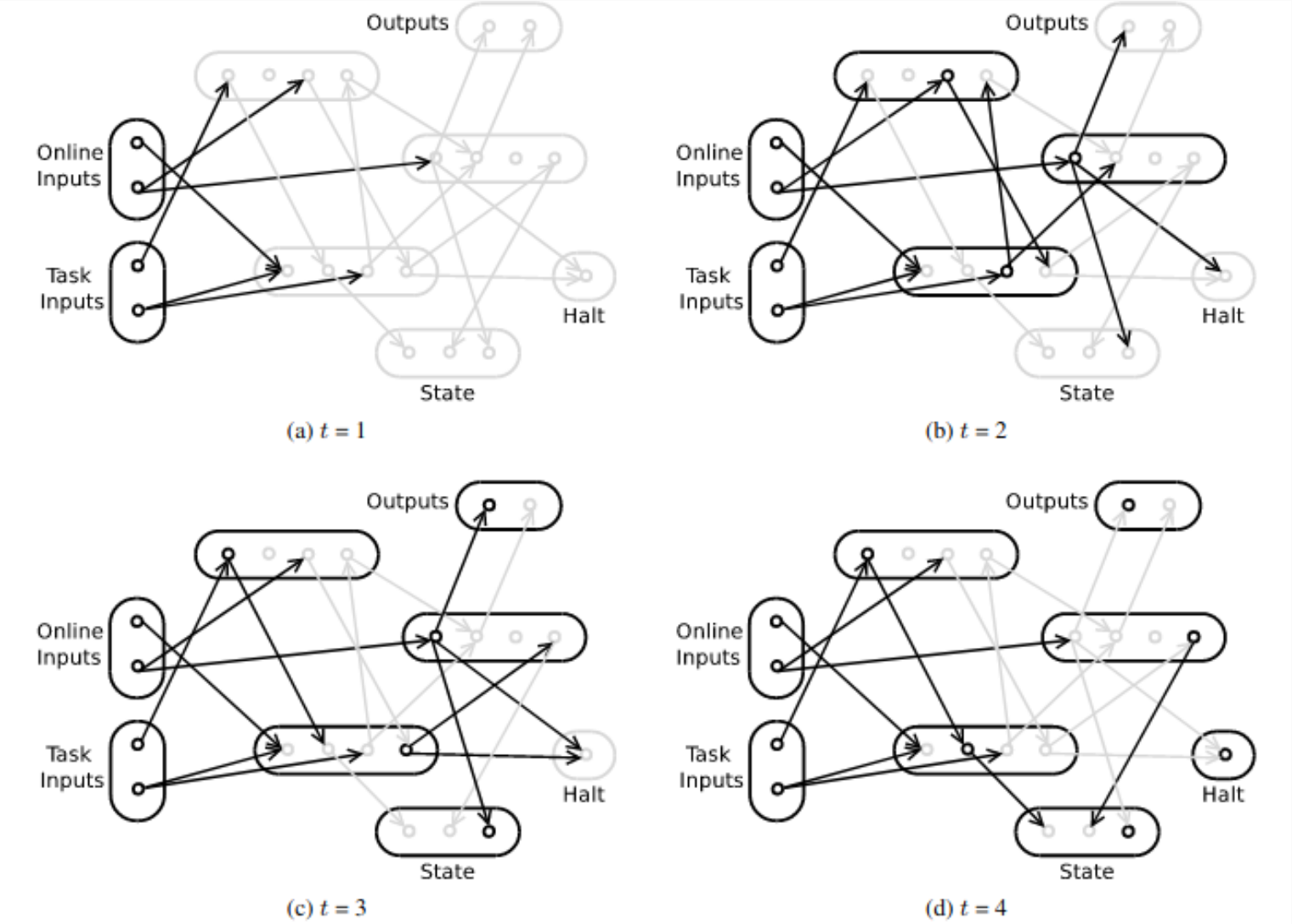
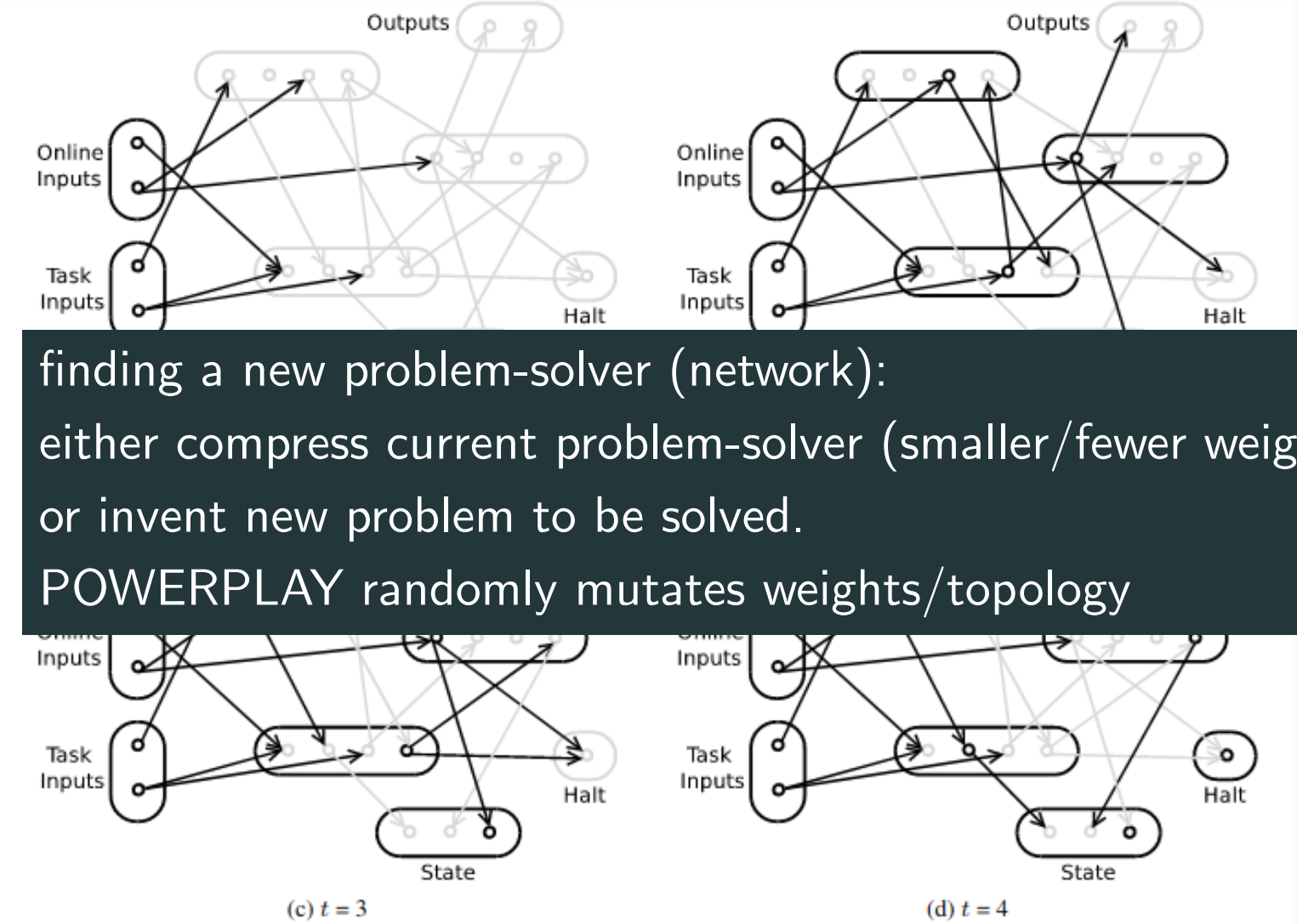


Figure 2: SLIM RNN activation scheme. At various time steps, active/winning neurons and their outgoing connections are highlighted. At each step, at most one neuron per WITAS can become active and propagate activations through its outgoing connections.

POWERPLAY on a recurrent network



finding a new problem-solver (network):
either compress current problem-solver (smaller/fewer weights),
or invent new problem to be solved.
POWERPLAY randomly mutates weights/topology

Figure 2: SLIM RNN activation scheme. At various time steps, active/winning neurons and their outgoing connections are highlighted. At each step, at most one neuron per WITAS can become active and propagate activations through its outgoing connections.

POWERPLAY demo

Learn to move around a "fovea." 24-dimensional input, 8 outputs (cardinal direction+step size)

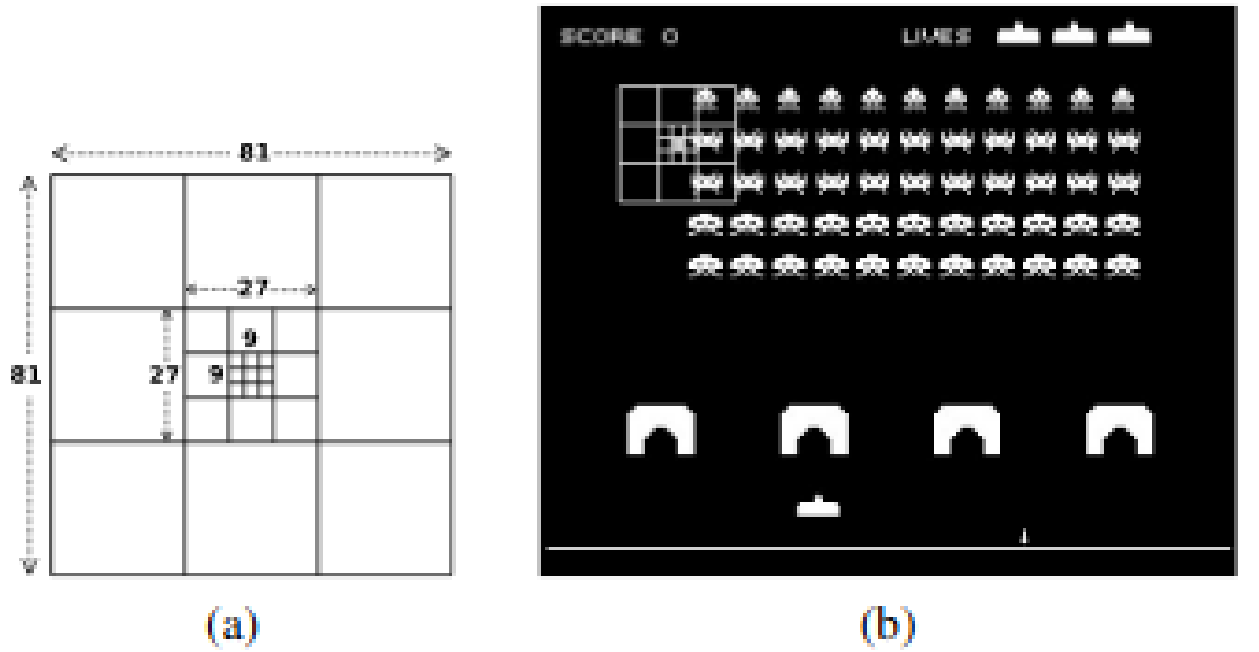


Fig. 3. (a) Fovea design. Pixel intensities over each square are averaged to produce a real valued input. The smallest squares in the center are of size 3x3. (b) The RNN controls the fovea movement over a static image, such as a still from the Space Invaders game.

Sparsification (+ spontaneous modularization)

Typically $\approx 50\%$ of network weights used for each task

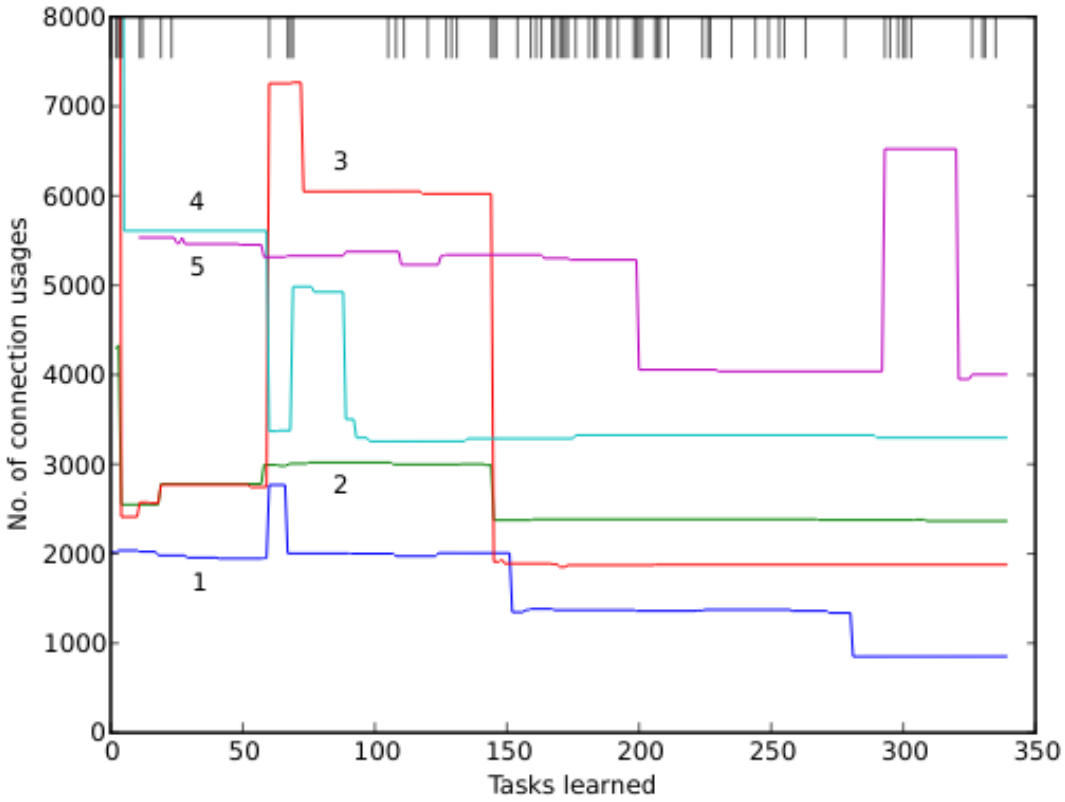


Figure 4: For the first five self-invented non-compression tasks, we plot the number of connection usages per task. In this run, solutions to 340 self-generated tasks were learned. 67 of them were non-compression tasks (marked by small black lines at the top); the rest resulted in successful compressions of the SLIM RNN's weight matrix. Over time, previously learned skills tend to require less and less computational resources, i.e., the SLIM RNN-based solver learns to speed up its solutions to previous self-invented tasks. Although some plot lines occasionally go up, this is compensated for by a decrease of connection usages for dozens of other tasks (not shown here to prevent clutter).

Words of caution

"It may not be advisable to let a general variant of POWERPLAY loose in an uncontrolled situation,e.g., on a multi-computer network on the internet, possibly with access to control of physical devices, and the potential to acquire additional computational and physical resources... This type of artificial curiosity/creativity, however, may conflict with human intentions on occasion. On the other hand, unchecked curiosity may sometimes also be harmful or fatal to the learning system itself – curiosity can kill the cat."

(from the conclusion of the POWERPLAY paper)