Learning logic programs though divide, constrain, and conquer

Andrew Cropper

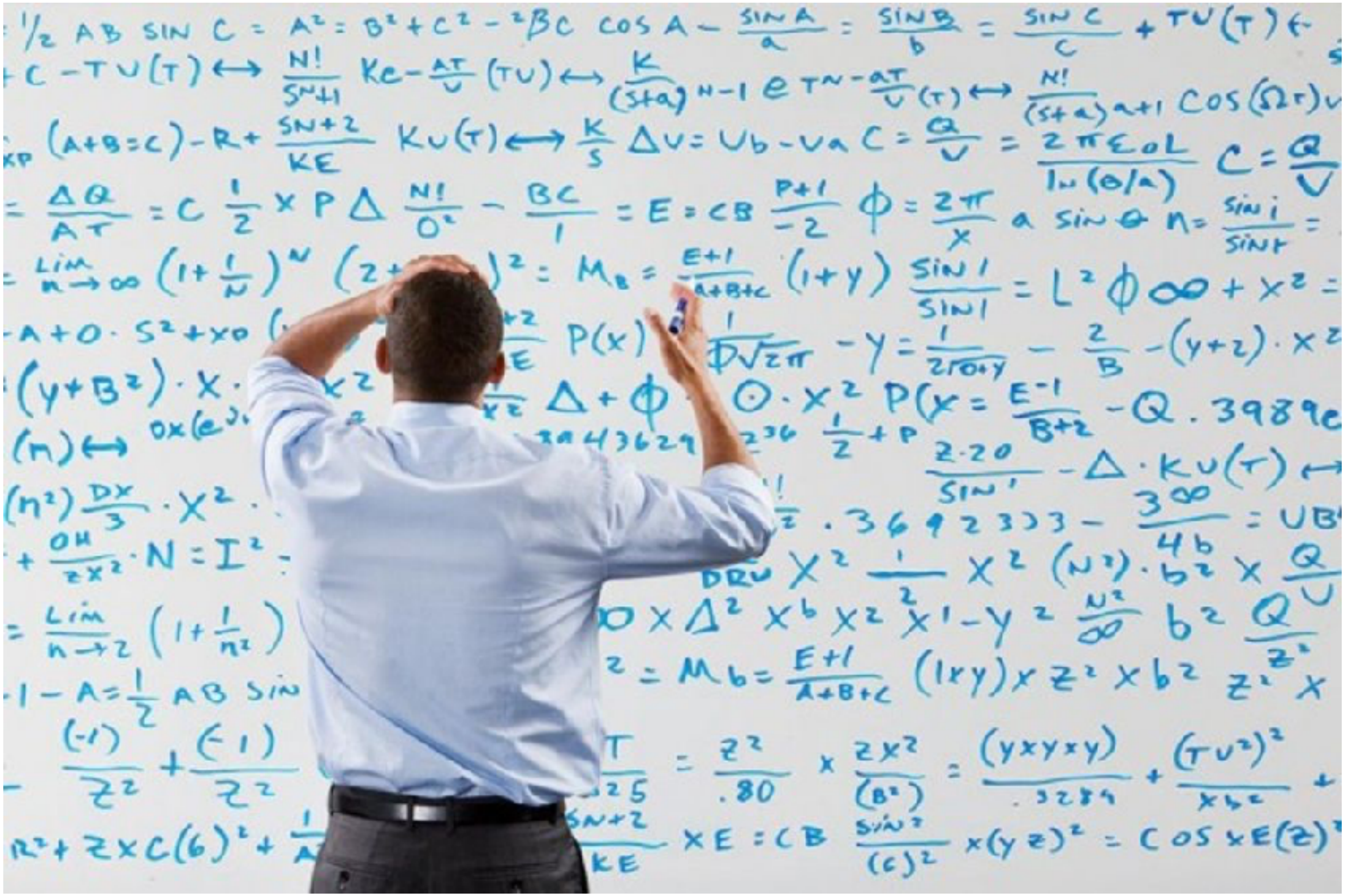
University of Oxford

https://github.com/logic-and-learning-lab/Popper

What is this talk about?

- **Simple** program induction approach
- Good performance

No technical details



Program induction

Program induction

Examples

Program induction

Examples

input	output
dog	g
sheep	p
chicken	?

Program induction

Examples

Background knowledge

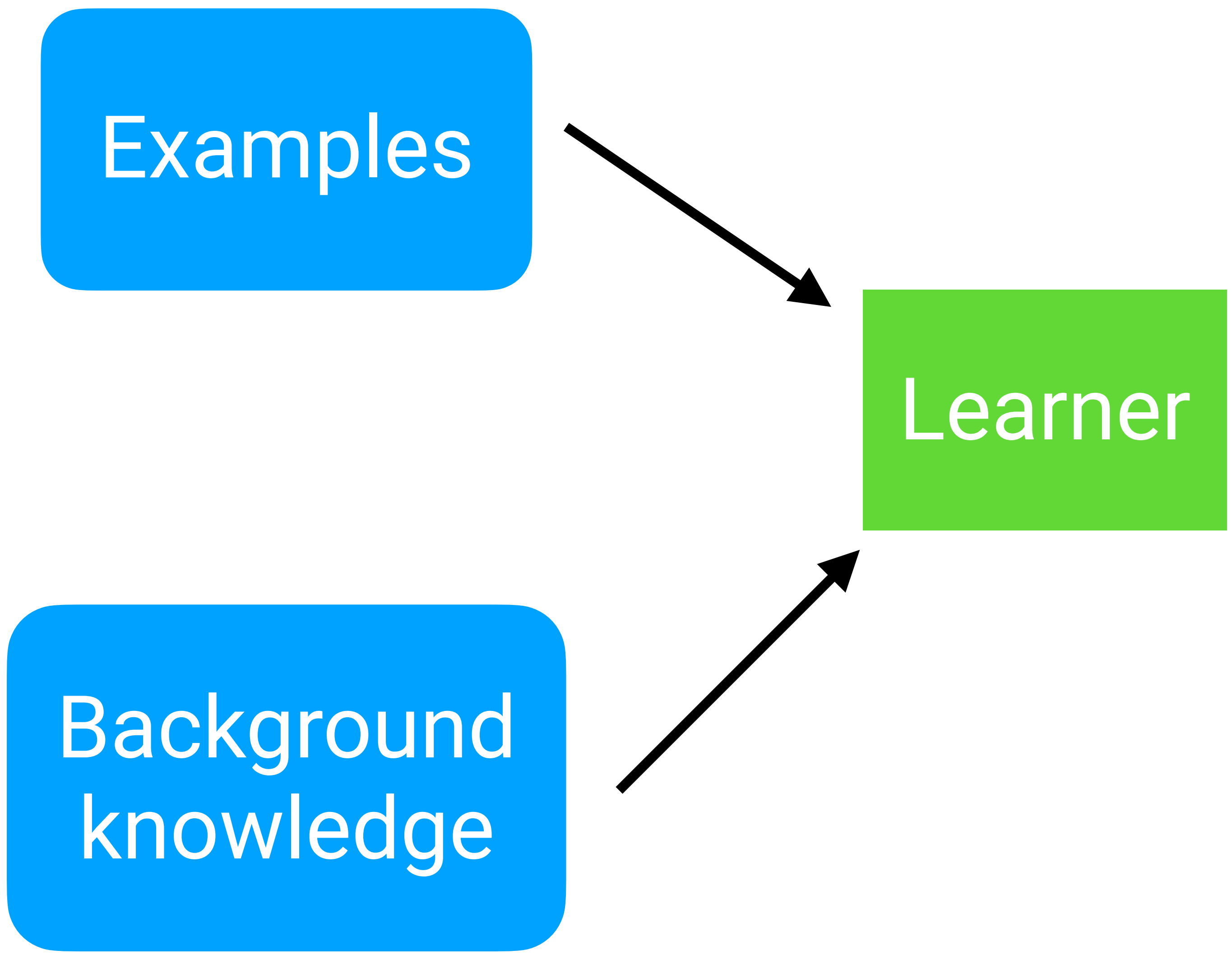
Program induction

Examples

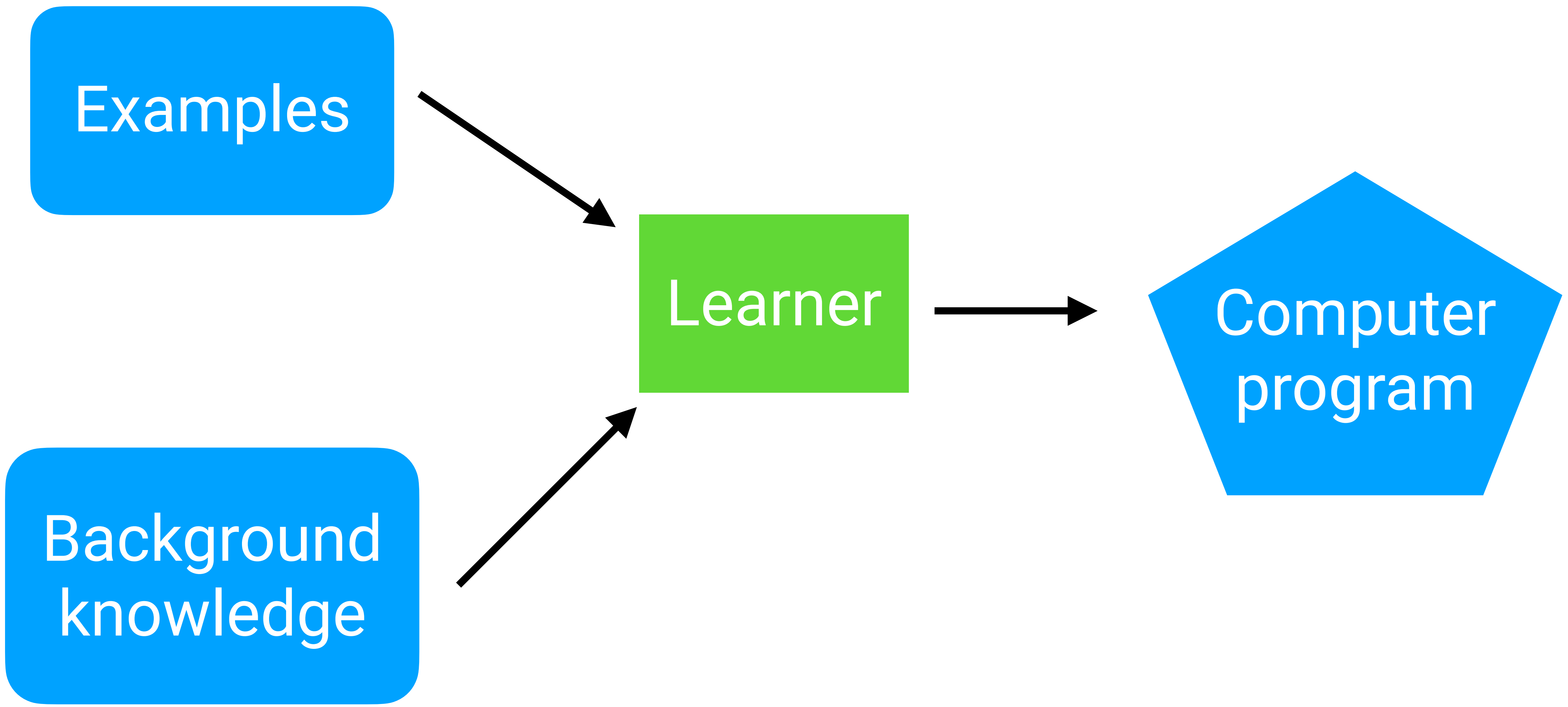
Background knowledge

```
head
tail
empty
reverse
```

Program induction



Program induction



Program induction

input	output
`dog`	`g`
`sheep`	`p`
`chicken`	**?**

Program induction

input	output
dog	g
sheep	p
chicken	**?**

```
def f(a):
    t = tail(a)
    if empty(t):
        return head(a)
    return f(t)
```

Inductive logic programming

input	output
dog	g
sheep	p
chicken	?

```
f(A,B):-tail(A,C),empty(C),head(A,B)
f(A,B):-tail(A,C),f(C,B)
```

Limitations

Large hypothesis spaces

Classical ILP

Good	Bad
Large rules	No recursion
Many rules	No predicate invention
	Overfitting

Modern ILP

Bad	**Good**
Small rules	Recursion
Few rules	Predicate invention
	Optimality

Idea

Combine old and new

Idea

Combine classical divide-and-conquer search with modern constraint-driven search

Divide, conquer, constrain (DCC)

DCC

DCC

Step 1. Learn a program for each example
Step 2. Generalise the programs

DCC

Key idea

use constraints to reduce the search complexity

DCC

Suppose we want to a program to find odd elements in a list:

```
f(A,B):-head(A,B),odd(B)
f(A,B):-head(A,B),even(B),tail(A,C),f(C,B)
```

DCC

```
e1 = f([4,3,4,6],3)
```

DCC

```
e1 = f([4,3,4,6],3)
h1 = f(A,B):-tail(A,C),head(C,B),odd(B)
```

DCC

```
e1 = f([4,3,4,6],3)
h1 = f(A,B):-tail(A,C),head(C,B),odd(B)

e2 = f([2,2,9,4,8,10],9)
```

DCC

```
e1 = f([4,3,4,6],3)
h1 = f(A,B):-tail(A,C),head(C,B),odd(B)

e2 = f([2,2,9,4,8,10],9)
h2 = f(A,B):-tail(A,C),tail(C,D),head(D,B),odd(B)
```

DCC

```
h1 = f(A,B):-tail(A,C),head(C,B),odd(B)
h2 = f(A,B):-tail(A,C),tail(C,D),head(D,B),odd(B)
```

DCC

```
h1 = f(A,B):-tail(A,C),head(C,B),odd(B)
h2 = f(A,B):-tail(A,C),tail(C,D),head(D,B),odd(B)
```

```
h3 =
  f(A,B):-tail(A,C),head(C,B),odd(B)
  f(A,B):-tail(A,C),tail(C,D),head(D,B),odd(B)
```

DCC

```
h1 = f(A,B):-tail(A,C),head(C,B),odd(B)
h2 = f(A,B):-tail(A,C),tail(C,D),head(D,B),odd(B)
h3 =
  f(A,B):-tail(A,C),head(C,B),odd(B)
  f(A,B):-tail(A,C),tail(C,D),head(D,B),odd(B)
```

Goal: find h4 s.t. |h4| < |h3|

DCC

```
h1 = f(A,B):-tail(A,C),head(C,B),odd(B)
h2 = f(A,B):-tail(A,C),tail(C,D),head(D,B),odd(B)
h3 =
  f(A,B):-tail(A,C),head(C,B),odd(B)
  f(A,B):-tail(A,C),tail(C,D),head(D,B),odd(B)
```

Conditions on h4:

- |h4| >= |h1|
- |h4| >= |h2|
- |h4| < |h3|
- |h4| is not a specialisation of h1
- |h4| is not a specialisation of h2
- |h4| is not a specialisation of h3

Optimisations

Constraints: maintain constraints during the search

Laziness: reuse existing solutions

Chunking: merge/compress examples

Does it work?

Q1. Can DCC improve learning performance?

Q2. How important are the optimisations?

Domains

- Trains (*classification*)
- Inductive general game playing
- Program synthesis

Predictive accuracies

Task	DCC	POPPER	ALEPH	METAGOL
trains1	100 ± 0	100 ± 0	100 ± 0	27 ± 0
trains2	98 ± 0	98 ± 0	100 ± 0	19 ± 0
trains3	98 ± 0	81 ± 1	100 ± 0	79 ± 0
trains4	100 ± 0	42 ± 5	39 ± 4	32 ± 0
md	99 ± 0	100 ± 0	94 ± 0	11 ± 0
buttons	98 ± 0	19 ± 0	87 ± 0	19 ± 0
rps	97 ± 0	18 ± 0	100 ± 0	18 ± 0
coins	86 ± 0	17 ± 0	17 ± 0	17 ± 0
dropk	99 ± 0	100 ± 0	52 ± 2	50 ± 0
droplast	100 ± 0	100 ± 0	50 ± 0	50 ± 0
evens	100 ± 0	100 ± 0	51 ± 0	50 ± 0
finddup	98 ± 0	98 ± 0	50 ± 0	50 ± 0
last	100 ± 0	100 ± 0	49 ± 0	55 ± 3
len	100 ± 0	100 ± 0	50 ± 0	50 ± 0
sorted	94 ± 2	96 ± 1	70 ± 1	50 ± 0
sumlist	100 ± 0	100 ± 0	50 ± 0	62 ± 4

Learning times

Task	DCC	Popper	Aleph	Metagol
trains1	8 ± 2	2 ± 0	4 ± 0.2	300 ± 0
trains2	41 ± 12	7 ± 0.9	1 ± 0.1	300 ± 0
trains3	106 ± 17	295 ± 3	35 ± 0.9	300 ± 0
trains4	268 ± 9	295 ± 2	297 ± 1	300 ± 0
md	172 ± 27	52 ± 1	3 ± 0	300 ± 0
buttons	300 ± 0	299 ± 0	86 ± 1	300 ± 0
rps	282 ± 12	285 ± 14	4 ± 0.1	0.3 ± 0
coins	291 ± 4	299 ± 0	300 ± 0	0.4 ± 0
dropk	3 ± 0.2	2 ± 0.2	3 ± 0.3	0.3 ± 0
droplast	2 ± 0.2	3 ± 0.1	300 ± 0	300 ± 0
evens	5 ± 0.4	4 ± 0.1	1 ± 0	217 ± 26
finddup	47 ± 6	13 ± 0.3	1 ± 0.1	300 ± 0
last	2 ± 0.4	2 ± 0.1	1 ± 0	270 ± 20
len	16 ± 2	5 ± 0.1	1 ± 0	300 ± 0
sorted	29 ± 3	19 ± 1	1 ± 0	288 ± 11
sumlist	18 ± 0.3	19 ± 0.6	0.6 ± 0	225 ± 29

Why does it work?

- Decompose the learning task
- Learn from failures, i.e. never repeat ourselves

Why care?

Simplicity: no metarules

Performance: good empirical results

Feature-rich:

- Recursion
- Optimal programs
- Large rules
- Many rules
- Predicate invention

Inductive general game playing

```
next(A,B):-succ(C,B),true(A,C).
next(A,B):-c_p(B),does(A,C,D),not_true(A,B),input(C,D),c_a(D).
next(A,B):-c_q(B),input(C,E),c_c(E),c_r(D),true(A,D),does(A,C,E).
next(A,B):-c_b(C),true(A,B),c_r(B),does(A,D,C),input(D,C).
next(A,B):-true(A,C),input(E,D),c_q(C),c_r(B),does(A,E,D),c_c(D).
next(A,B):-c_q(B),true(A,B),does(A,C,D),input(C,D),c_a(D).
next(A,B):-c_p(B),true(A,B),does(A,C,D),input(C,D),c_c(D).
next(A,B):-true(A,B),c_r(B),does(A,C,D),input(C,D),c_a(D).
next(A,B):-c_b(C),c_p(D),does(A,E,C),input(E,C),true(A,D),c_q(B).
next(A,B):-true(A,C),c_p(B),does(A,E,D),c_q(C),c_b(D),input(E,D).
```

Program synthesis

```
f(A,B,C):-one(B),tail(A,C).
f(A,B,C):-decrement(B,D),f(A,D,E),tail(E,C).
```

~3 seconds

Limitations + future work

Expressivity: negation as failure, higher-order

Limitations + future work

Expressivity: negation as failure, higher-order

Constants: especially numerical values

Limitations + future work

Expressivity: negation as failure, higher-order

Constants: especially numerical values

Faster: detailed failure explanation, more \`complete’ constraints

Questions?

https://github.com/logic-and-learning-lab/Popper