Inducing logic programs by learning from failures

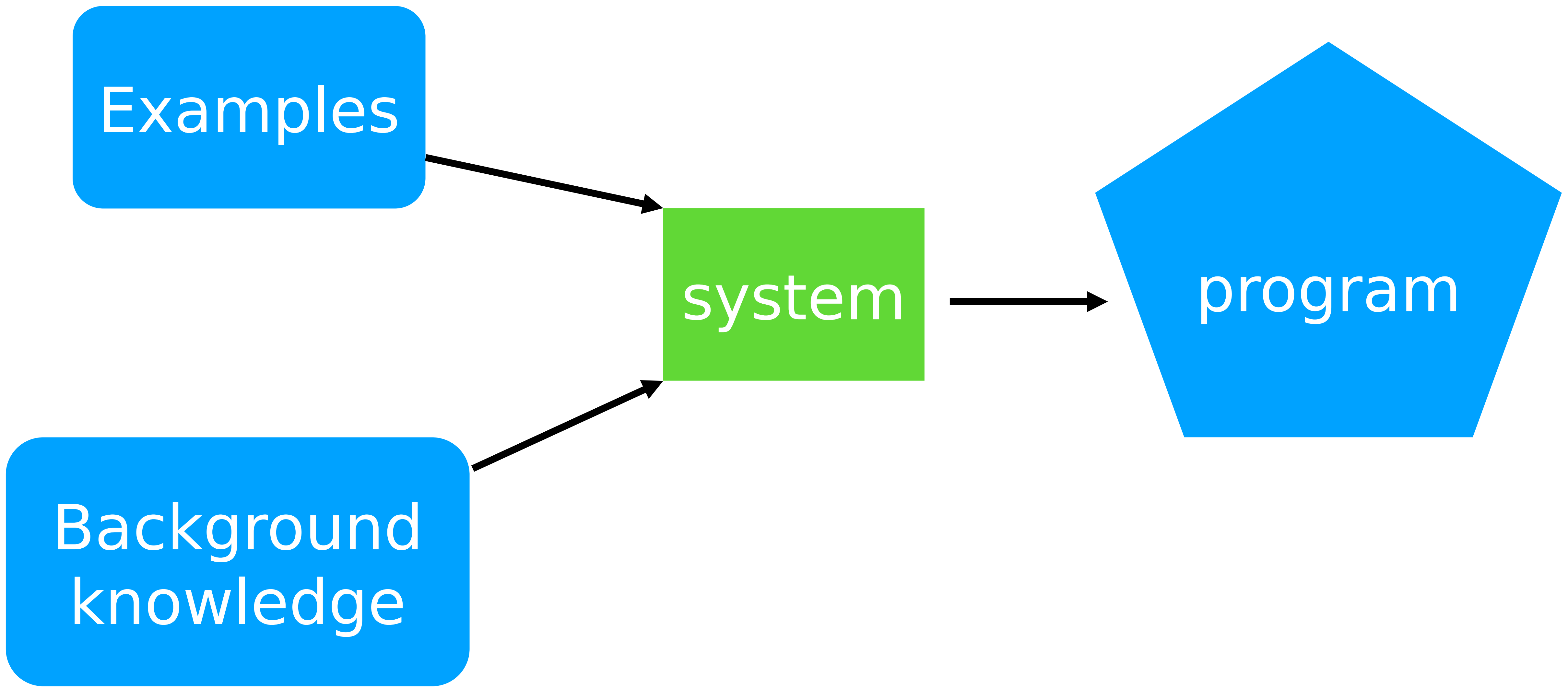
(Popper)

Rolf Morel & Andrew Cropper

What is this talk about?

- Basics of Inductive Logic Programming
- A new **very simple** ILP approach
- A new system: Popper
  - Same features as existing systems
  - Outperforms these systems
- Much scope for novel extensions

Program induction



Inductive Logic Programming (ILP)

A form of ML using
logic programs to represent:

- Examples (i.e. training/test data)
- background knowledge
- hypotheses

Examples

input	output
dog	g
sheep	p
chicken	n

representation
last(dog,g)
last(sheep,p)
last(chicken,n)

Background Knowledge (BK)

```
head([H|_],H).
tail([_|T],T).
empty(A):- A=[].
double(A,B):- B is A+A.
```

Hypotheses

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

Motivation

Limitations of current systems:

- Difficulty with recursion
  classical ILP, such as *Progol, Aleph*

Motivation

Limitations of current systems:

- Difficulty with recursion
  classical ILP, such as *Progol, Aleph*

- Difficulty with infinite domains
  modern systems, such as *ILASP, HEXMIL*

Motivation

Limitations of current systems:

- Difficulty with recursion
  classical ILP, such as *Progol, Aleph*
- Difficulty with infinite domains
  modern systems, such as *ILASP, HEXMIL*
- Difficulty with large hypothesis spaces
  ILASP, HEXMIL, δILP precompute entire rule space

Motivation

Limitations of current systems:

- Difficulty with recursion
  classical ILP, such as *Progol, Aleph*
- Difficulty with infinite domains
  modern systems, such as *ILASP, HEXMIL*
- Difficulty with large hypothesis spaces
  ILASP, HEXMIL, δILP precompute entire rule space
- Needing program templates (known as metarules)
  Metagol, HEXMIL, δILP

Learning from failures

Automating Karl Popper’s falsifiability:

1. Form a hypothesis
2. Empirically evaluate hypothesis
3. If hypothesis fails, **determine why**
4. Use the **explanation** to rule out other hypotheses

Learning from failures

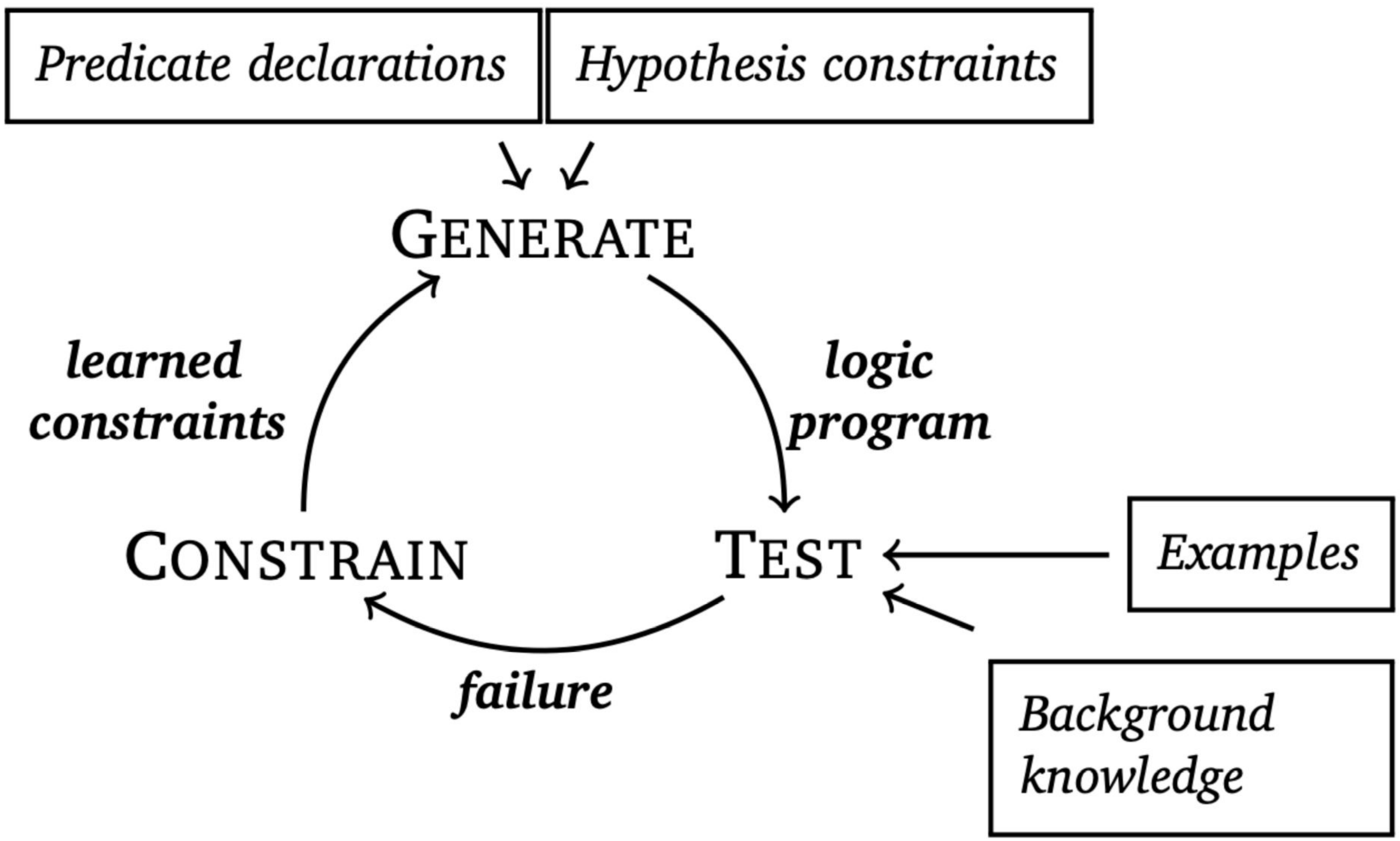
Automating Karl Popper’s falsifiability:

1. Form a hypothesis
   Generate a program
2. Empirically evaluate hypothesis
   Test program on training examples
3. If hypothesis fails, determine why
   Determine kind of program failure
4. Use the explanation to rule out other hypotheses
   Never generate programs with the same failure

Learning from failures

1. Generate
2. Test
3. Constrain

Learning from failures



input	output
laura	a
penelope	e
emma	m
james	e

input	output
laura	a
penelope	e
emma	m
james	e

$$E^{+} = \left\{ \begin{array}{l} \texttt{last([l,a,u,r,a],a).} \\ \texttt{last([p,e,n,e,l,o,p,e],e).} \end{array} \right\} \qquad E^{-} = \left\{ \begin{array}{l} \texttt{last([e,m,m,a],m).} \\ \texttt{last([j,a,m,e,s],e).} \end{array} \right\}$$

$$
\mathscr{H}_1 = \left\{
\begin{array}{l}
h_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\} \\
h_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\} \\
h_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\} \\
h_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\} \\
h_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\
h_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\
h_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\
h_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array}
\right\}
$$

Hypothesis space is much larger (and can be infinite)

$$h_1 = \{\ \texttt{last(A,B):- head(A,B).}\ \}$$

$$h_1 = \{ \texttt{last(A,B):- head(A,B).} \}$$

input	output	entailed
laura	a	**no**
penelope	e	**no**
emma	m	no
james	e	no

$$h_1 = \{\ \texttt{last(A,B):- head(A,B).}\ \}$$

input	output	entailed
laura	a	**no**
penelope	e	**no**
emma	m	no
james	e	no

H1 is too specific

Prune specialisations

$$\mathscr{H}_1 = \left\{ \begin{array}{l}
\mathsf{h}_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\} \\
\mathsf{h}_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\} \\
\mathsf{h}_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\} \\
\mathsf{h}_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\} \\
\mathsf{h}_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\
\mathsf{h}_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\
\mathsf{h}_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\
\mathsf{h}_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array} \right\}$$

Prune specialisations

$$
\mathcal{H}_1 = \left\{
\begin{array}{l}
\cancel{h_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\}} \\
h_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\} \\
h_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\} \\
h_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\} \\
h_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\
h_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\
h_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\
h_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array}
\right\}
$$

Prune specialisations

$\mathcal{H}_1 =$

- ~~$h_1 =$ { `last(A,B):- head(A,B).` }~~
- ~~$h_2 =$ { `last(A,B):- head(A,B),empty(A).` }~~
- $h_3 =$ { `last(A,B):- head(A,B),reverse(A,C),head(C,B).` }
- $h_4 =$ { `last(A,B):- tail(A,C),head(C,B).` }
- $h_5 =$ { `last(A,B):- reverse(A,C),head(C,B).` }
- $h_6 =$ { `last(A,B):- tail(A,C),head(C,B).` `last(A,B):- reverse(A,C),head(C,B).` }
- $h_7 =$ { `last(A,B):- tail(A,C),head(C,B).` `last(A,B):- tail(A,C),tail(C,D),head(D,B).` }
- $h_8 =$ { `last(A,B):- reverse(A,C),tail(C,D),head(D,B).` `last(A,B):- tail(A,C),reverse(C,D),head(D,B).` }

Prune specialisations

$$
\mathscr{H}_1 = \left\{
\begin{array}{l}
\sout{\mathsf{h}_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\}} \\
\sout{\mathsf{h}_2 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\}} \\
\sout{\mathsf{h}_3 = \left\{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \right\}} \\
\mathsf{h}_4 = \left\{ \texttt{last(A,B):- tail(A,C),head(C,B).} \right\} \\
\mathsf{h}_5 = \left\{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \right\} \\
\mathsf{h}_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\
\mathsf{h}_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\
\mathsf{h}_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\}
\end{array}
\right\}
$$

$$h_4 = \{\texttt{last(A,B):- tail(A,C),head(C,B).}\}$$

$$h_4 = \{\texttt{last(A,B):- tail(A,C),head(C,B).}\}$$

input	output	entailed
laura	a	yes
penelope	e	yes
emma	m	**yes**
james	e	no

$$h_4 = \{ \texttt{last(A,B):- tail(A,C),head(C,B).} \}$$

input	output	entailed
laura	a	yes
penelope	e	yes
emma	m	**yes**
james	e	no

H4 is too general

Prune generalisations

$\mathscr{H}_1 =$

- ~~$h_1 = \{$ `last(A,B):- head(A,B).` $\}$~~
- ~~$h_2 = \{$ `last(A,B):- head(A,B),empty(A).` $\}$~~
- ~~$h_3 = \{$ `last(A,B):- head(A,B),reverse(A,C),head(C,B).` $\}$~~
- $h_4 = \{$ `last(A,B):- tail(A,C),head(C,B).` $\}$
- $h_5 = \{$ `last(A,B):- reverse(A,C),head(C,B).` $\}$
- $h_6 = \{$ `last(A,B):- tail(A,C),head(C,B).` `last(A,B):- reverse(A,C),head(C,B).` $\}$
- $h_7 = \{$ `last(A,B):- tail(A,C),head(C,B).` `last(A,B):- tail(A,C),tail(C,D),head(D,B).` $\}$
- $h_8 = \{$ `last(A,B):- reverse(A,C),tail(C,D),head(D,B).` `last(A,B):- tail(A,C),reverse(C,D),head(D,B).` $\}$

Prune generalisations

$\mathcal{H}_1 =$

- ~~h_1 = { last(A,B):- head(A,B). }~~
- ~~h_2 = { last(A,B):- head(A,B),empty(A). }~~
- ~~h_3 = { last(A,B):- head(A,B),reverse(A,C),head(C,B). }~~
- ~~h_4 = { last(A,B):- tail(A,C),head(C,B). }~~
- h_5 = { last(A,B):- reverse(A,C),head(C,B). }
- h_6 = { last(A,B):- tail(A,C),head(C,B). last(A,B):- reverse(A,C),head(C,B). }
- h_7 = { last(A,B):- tail(A,C),head(C,B). last(A,B):- tail(A,C),tail(C,D),head(D,B). }
- h_8 = { last(A,B):- reverse(A,C),tail(C,D),head(D,B). last(A,B):- tail(A,C),reverse(C,D),head(D,B). }

Prune generalisations

$\mathcal{H}_1 = \{$

~~$h_1 = \{$ last(A,B):- head(A,B). $\}$~~

~~$h_2 = \{$ last(A,B):- head(A,B),empty(A). $\}$~~

~~$h_3 = \{$ last(A,B):- head(A,B),reverse(A,C),head(C,B). $\}$~~

~~$h_4 = \{$ last(A,B):- tail(A,C),head(C,B). $\}$~~

$h_5 = \{$ last(A,B):- reverse(A,C),head(C,B). $\}$

~~$h_6 = \{$ last(A,B):- tail(A,C),head(C,B).~~
~~last(A,B):- reverse(A,C),head(C,B). $\}$~~

$h_7 = \{$ last(A,B):- tail(A,C),head(C,B).
last(A,B):- tail(A,C),tail(C,D),head(D,B). $\}$

$h_8 = \{$ last(A,B):- reverse(A,C),tail(C,D),head(D,B).
last(A,B):- tail(A,C),reverse(C,D),head(D,B). $\}$

$\}$

Prune generalisations

$\mathscr{H}_1 =$

- ~~$h_1 =$ { last(A,B):- head(A,B). }~~
- ~~$h_2 =$ { last(A,B):- head(A,B),empty(A). }~~
- ~~$h_3 =$ { last(A,B):- head(A,B),reverse(A,C),head(C,B). }~~
- ~~$h_4 =$ { last(A,B):- tail(A,C),head(C,B). }~~
- $h_5 =$ { last(A,B):- reverse(A,C),head(C,B). }
- $h_6 =$ { ~~last(A,B):- tail(A,C),head(C,B).~~ ~~last(A,B):- reverse(A,C),head(C,B).~~ }
- $h_7 =$ { ~~last(A,B):- tail(A,C),head(C,B).~~ ~~last(A,B):- tail(A,C),tail(C,D),head(D,B).~~ }
- $h_8 =$ { last(A,B):- reverse(A,C),tail(C,D),head(D,B). last(A,B):- tail(A,C),reverse(C,D),head(D,B). }

$h_5 = \{$ `last(A,B):- reverse(A,C),head(C,B).` $\}$

$h_5 = \{$ `last(A,B):- reverse(A,C),head(C,B).` $\}$

input	output	entailed
laura	a	yes
penelope	e	yes
emma	m	no
james	e	no

H5 does not fail, so return it

Hypothesis constraints

- Generalisation
- Specialisation
- *Redundancy*

Constraints are **sound:**
they do not prune (*optimal*) solutions

Key ideas

1. Refine the hypothesis space through learned hypothesis constraints

2. Decompose the learning problem (i.e. do not just throw the whole problem to a SAT solver)

Learning from failures

Advantages	Disadvantages
• Optimality • Completeness • Recursion • Infinite domains • Fast • **Simple**	• Noise • ~~Predicate invention~~

Popper

1. Generate (ASP program)
2. Test (Prolog)
3. Constrain (ASP constraints)

Generate

Meta-level ASP program, i.e. models are programs

```
% possible clauses
allowed_clause(0..N-1):- max_clauses(N).

% variables
var(0..N-1):- max_vars(N).

% clauses with a head literal
clause(Clause):- head_literal(Clause,_,_,_).

%% head literals
0 {head_literal(Clause,P,A,Vars): head_pred(P,A), vars(A,Vars)} 1:-
    allowed_clause(Clause).

%% body literals
1 {body_literal(Clause,P,A,Vars): body_pred(P,A), vars(A,Vars)} N:-
    clause(Clause), max_body(N).

% variable combinations
vars(1,(Var1,)):- var(Var1).
vars(2,(Var1,Var2)):- var(Var1),var(Var2).
vars(3,(Var1,Var2,Var3)):- var(Var1),var(Var2),var(Var3).
```

Declarative!

Generate

Adding constraints eliminates models and thus programs

```
recursive:- recursive(Clause).

recursive(Clause):- head_literal(Clause,P,A,_), body_literal(Clause,P,A,_).

has_base:- clause(Clause), not recursive(Clause).

% need multiple clauses for recursion
:- recursive(_), not clause(1).

% prevent recursion without a basecase
:- recursive, not has_base.
```

Hard-coded intuitive constraints are important, but they could be learned

Generate

```
head_var(Clause,Var):- head_literal(Clause,_,_,Vars), var_member(Var,Vars).

body_var(Clause,Var):- body_literal(Clause,_,_,Vars), var_member(Var,Vars).

% prevent singleton variables
:- clause_var(Clause,Var), #count{P,Vars: var_in_literal(Clause,P,Vars,Var)} == 1.

% head vars must appear in the body
:- head_var(Clause,Var), not body_var(Clause,Var).

%% type matching
:- var_in_literal(Clause,P,Vars1,Var),var_in_literal(Clause,Q,Vars2,Var),
    var_pos(Var,Vars1,Pos1),var_pos(Var,Vars2,Pos2),
    type(P,Pos1,Type1),type(Q,Pos2,Type2),
    Type1 != Type2.
```

Generate

Domain specific declarative bias:
user-provided hypothesis constraints

```
:- body_literal(Cl,p,2,_),
   body_literal(Cl,q,2,_).
```

Test using Prolog

1. Fast
2. Infinite domains
3. Complex data structures

Could use a Datalog engine, or an ASP solver, or something else

Constrain

```
h = { last(A,B):- head(A,B). }
```

```
:-
  head_literal(C0,last,2,(C0V0,C0V1)),
  body_literal(C0,head,2,(C0V0,C0V1)),
  C0V0 != C0V1,clause_size(C0,1).
```

The above is a generalisation constraint

Popper algorithm

Algorithm 1 Popper

```
def popper(e+, e-, bk, declarations, constraints, max_literals):
  num_literals = 1
  while num_literals ≤ max_literals:
    program = generate(declarations, constraints, num_literals)
    if program == 'space_exhausted':
      num_literals += 1
      continue
    outcome = test(e+, e-, bk, program)
    if outcome == ('all_positive', 'none_negative')
      return program
    constraints += learn_constraints(program, outcome)
  return {}
```

Uses Clingo’s multi-shot solving to remember state

Popper

	Progol	Metagol	ILASP	∂ILP	Popper
Hypotheses	Normal	Definite	ASP	Datalog	Definite
Language bias	Modes	Metarules	Modes	Templates	Declarations
Predicate invention	No	**Yes**	Partly	Partly	No
Noise handling	**Yes**	No	**Yes**	**Yes**	No
Recursion	Partly	**Yes**	**Yes**	**Yes**	**Yes**
Optimality	No	**Yes**	**Yes**	**Yes**	**Yes**
Infinite domains	**Yes**	**Yes**	No	No	**Yes**
Hypothesis constraints	No	No	No	No	**Yes**

Does it work?

Q1. Can constraints improve learning performance, i.e. does it outperform pure enumeration?

Q2. Can Popper outperform SOTA ILP systems?

Buttons

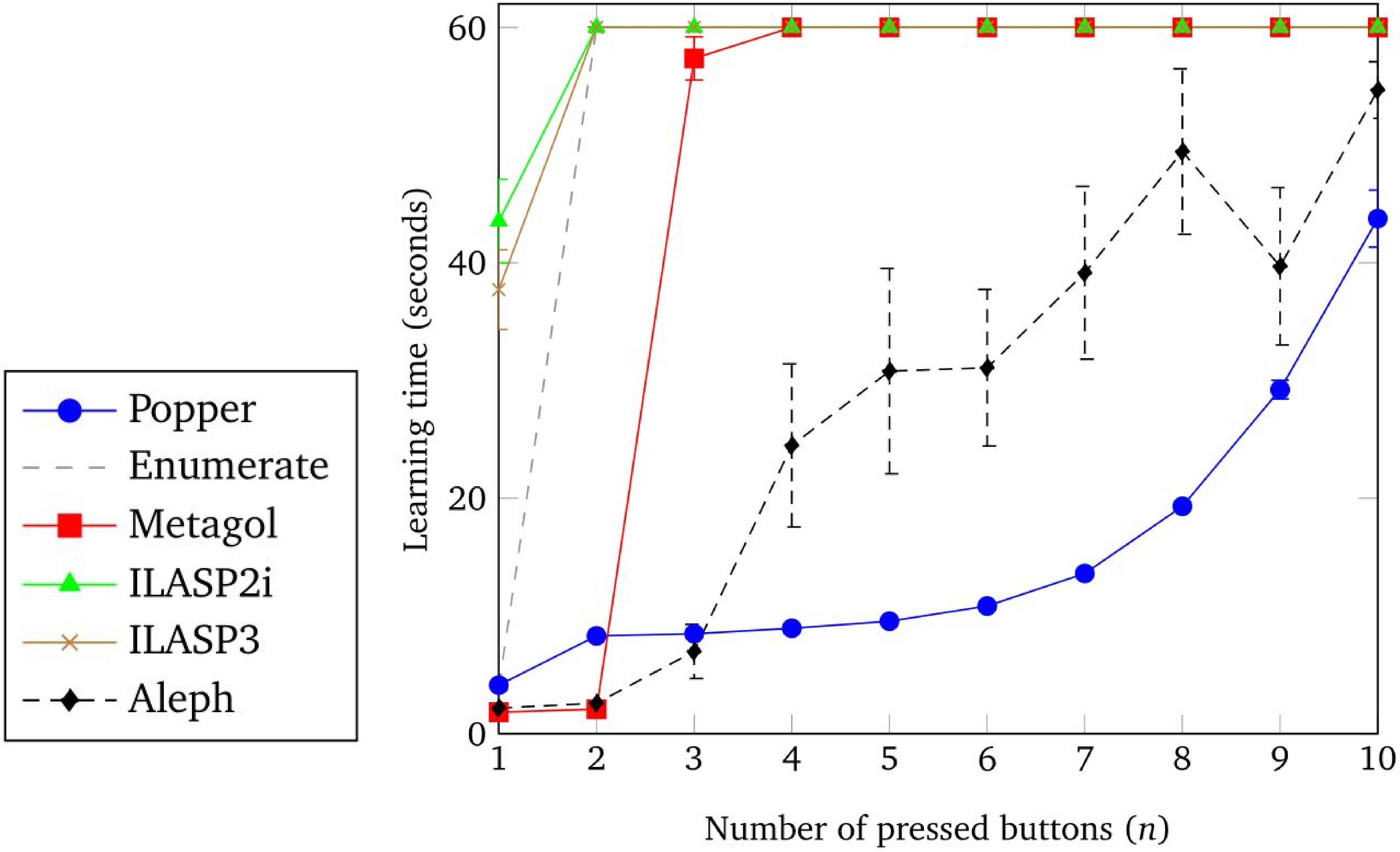
Purposely simple experiment to test the claims

Given *p* buttons, learn which *n* need to be pressed

```
win(A):- button6(A),button4(A),button7(A)
```

Hypothesis space for p = 200 **contains about** 10^{16} **programs**

p = 200 **Buttons**



(x-axis is program size)

Programming puzzles

Name	Description	Example solution
addhead	Prepend the head three times	addhead(A,B):—head(A,C),cons(C,A,D),cons(C,D,E),cons(C,E,B).
dropk	Drop the first k elements	dropk(A,B,C):—one(B),tail(A,C). dropk(A,B,C):—tail(A,D),decrement(B,E),dropk(D,E,B).
droplast	Drop the last element	droplast(A,B):—tail(A,B), tail (B,C),empty(C). droplast(A,B):—tail(A,C),droplast(C,D),head(A,E),cons(E,D,B).
evens	Check all elements are even	evens(A):—empty(A). evens(A):—even(A),tail(A,C),evens(C).
finddup	Find duplicate elements	finddup(A,B):—head(A,B),tail(A,C),member(B,C). finddup(A,B):—tail(A,C),finddup(C,B).
last	Last element	last (A,B):—tail(A,C),empty(C),head(A,B). last (A,B):—tail(A,C), last (C,B).
len	Calculate list length	len(A,B):—empty(A),zero(B). len(A,B):—tail(A,C),len(C,D),succ(D,B).
member	Member of a list	member(A,B):—head(A,B). member(A,B):—tail(A,C),member(C,B).
sorted	Check list is sorted	sorted(A):—empty(A). sorted(A):—head(A,B),tail(A,C),head(C,D),geq(D,B),sorted(C).
threesame	First three elements are identical	threesame(A):—head(A,B),tail(A,C),head(C,B),tail(C,D),head(D,B).

Programming puzzles (Accuracy)

Name	Popper	Enumerate	Metagol	Aleph
addhead	**100** ± 0	**100** ± 0	n/a	90 ± 10
dropk	**100** ± 0	50 ± 0	n/a	50 ± 0
droplast	**100** ± 0	50 ± 0	n/a	50 ± 0
evens	**100** ± 0	**100** ± 0	50 ± 0	50 ± 0
finddup	98 ± 0	50 ± 0	**100** ± 0	50 ± 0
last	**100** ± 0	50 ± 0	**100** ± 0	50 ± 0
len	**100** ± 0	50 ± 0	50 ± 0	50 ± 0
member	**100** ± 0	**100** ± 0	**100** ± 0	50 ± 0
sorted	**100** ± 0	50 ± 0	50 ± 0	68 ± 2
threesame	**99** ± 0	**99** ± 0	**99** ± 0	**99** ± 0

Programming puzzles (Learning times)

Name	Popper	Enumerate	Metagol	Aleph
addhead	**0.5** ± 0	2 ± 0	n/a	103 ± 49
dropk	**0.8** ± 0	300 ± 0	n/a	3 ± 0.2
droplast	**3** ± 0.1	300 ± 0	n/a	300 ± 0
evens	4 ± 0.1	159 ± 0.1	300 ± 0	**1** ± 0
finddup	36 ± 2	300 ± 0	2 ± 0.5	**1.0** ± 0.1
last	2 ± 0.1	300 ± 0	**0.7** ± 0.2	1 ± 0.1
len	12 ± 0.3	300 ± 0	300 ± 0	**1** ± 0
member	0.4 ± 0.1	7 ± 0	**0.3** ± 0	0.9 ± 0.1
sorted	23 ± 1	300 ± 0	300 ± 0	**0.8** ± 0
threesame	**0.2** ± 0.1	0.4 ± 0.2	0.9 ± 0.3	0.5 ± 0

Future work

- Sub-programs as failure explanation
- Completeness of constraints
- Parallel Popper
- More “expressive” hypotheses (e.g. ASP)

Conclusions

Simplicity: LFF is a simple form of ILP

Performance:
Popper can outperform S.O.T.A. approaches.

Feature rich:
Popper supports recursion, infinite domains, and learning optimal programs.

Paper: ***Learning programs by learning from failures***.
Cropper and Morel. Machine Learning, 2021.