Journal of Artificial Intelligence Research 1 (1993) 1-15 Submitted 6/91; published 9/91

Inductive Logic Programming At 30: A New Introduction

Andrew Cropper ANDREW.CROPPER@CS.OX.AC.UK
University of Oxford

Sebastijan Dumančić S.DUMANCIC@TUDELFT.NL
TU Delft

Abstract

Inductive logic programming (ILP) is a form of machine learning. The goal of ILP is to induce a hypothesis (a set of logical rules) that generalises training examples. As ILP turns 30, we provide a new introduction to the field. We introduce the necessary logical notation and the main learning settings; describe the building blocks of an ILP system; compare several systems on several dimensions; describe four systems (Aleph, TILDE, ASPAL, and Metagol); highlight key application areas; and, finally, summarise current limitations and directions for future research.

1. Introduction

A remarkable feat of human intelligence is the ability to learn knowledge. A key form of learning is *induction*: the process of forming general rules (hypotheses) from specific observations (examples). For instance, suppose you draw 10 red balls out of a bag, then you might induce a hypothesis (a rule) that all the balls in the bag are red. Having induced this hypothesis, you can predict the colour of the next ball out of the bag.

Machine learning (ML) automates induction. ML induces a hypothesis (also called a *model*) that generalises training examples (observations). For instance, given labelled images of cats and dogs, the goal of ML is to induce a hypothesis that predicts whether an unlabelled image is a cat or a dog. Inductive logic programming (ILP) (Muggleton, 1991) is a form of ML. As with other forms of ML, the goal of ILP is to induce a hypothesis that generalises training examples. However, whereas most forms of ML use tables[1] to represent data (examples and hypotheses), ILP uses *logic programs* (sets of logical rules). Moreover, whereas most forms of ML learn functions, ILP learns relations. We illustrate ILP using four scenarios.

1.1 Scenario 1: Concept Learning

Suppose we want to predict whether someone is happy. To do so, we ask four people (*alice*, *bob*, *claire*, and *dave*) whether they are happy. We also ask for additional information, specifically their job, their company, and whether they like lego. Many ML approaches, such as a decision tree or neural network learner, would represent this data as a table, such as Table 1. Using standard ML terminology, each row represents a training example, the first three columns (*name*, *job*, and *enjoys lego*) represent *features*, and the final column (*happy*) represents the *label* or *classification*. Given this table as input, the goal is to induce a hypothesis that generalises the training examples. For instance, a neural network learner would learn a table of numbers that weight the importance of the features (or hidden features in a multi-layer network). We can then use the hypothesis to predict labels for unseen examples.

[1] Table-based learning is *attribute-value* learning. See De Raedt (2008) for an overview of the hierarchy of representations. Note that not *all* other forms of ML use tables. For instance, Rocktäschel and Riedel (2017) use embeddings.

Name	**Job**	**Enjoys lego**	**Happy**
alice	lego builder	yes	yes
bob	lego builder	no	no
claire	estate agent	yes	no
dave	estate agent	no	no

Table 1: A table representation of a ML task.

Rather than represent data as tables, ILP represents data as logic programs, sets of logical rules. The main building block of a logic program is an *atom*. An atom is of the form $p(x_1, \ldots, x_n)$, where p is a *predicate* symbol of arity n (takes n arguments) and each x_i is a *term*. A logic program uses atoms to represent data. For instance, we can represent that *alice* enjoys lego as the atom `enjoys_lego(alice)` and that *bob* is a lego builder as `lego_builder(bob)`.

An ILP task is formed of three sets (B, E^+, E^-). The set B is *background knowledge* (BK). BK is similar to features but can contain relations and information indirectly associated with the examples. We can represent the data in Table 1 as the set B:

$$
\mathrm{B} = \left\{ \begin{array}{l}
\texttt{lego\_builder(alice).} \\
\texttt{lego\_builder(bob).} \\
\texttt{estate\_agent(claire).} \\
\texttt{estate\_agent(dave).} \\
\texttt{enjoys\_lego(alice).} \\
\texttt{enjoys\_lego(claire).}
\end{array} \right\}
$$

ILP usually follows the *closed world assumption* (Reiter, 1977), so if anything is not explicitly true we assume it is false. With this assumption, we do not need to explicitly state that `enjoys_lego(bob)` and `enjoys_lego(dave)` are false.

The sets E^+ and E^- represent positive and negative examples respectively. We can represent the examples in Table 1 as:

$$
\mathrm{E}^+ = \left\{ \begin{array}{l} \texttt{happy(alice).} \end{array} \right\} \mathrm{E}^- = \left\{ \begin{array}{l}
\texttt{happy(bob).} \\
\texttt{happy(claire).} \\
\texttt{happy(dave).}
\end{array} \right\}
$$

Given these sets, the goal of ILP is to induce a hypothesis that with the BK logically entails as many positive and as few negative examples as possible. A hypothesis (H) in ILP is a set of logical rules, such as:

$$
\mathrm{H} = \left\{ \begin{array}{l} \forall A.\ \texttt{lego\_builder(A)} \land \texttt{enjoys\_lego(A)} \longrightarrow \texttt{happy(A)} \end{array} \right\}
$$

This hypothesis contains one rule that says that for all `A`, if `A` is a lego builder (`lego_builder(A)`) and enjoys lego (`enjoys_lego(A)`), then `A` is happy (`happy(A)`). Having induced a rule, we can deduce knowledge from it. For instance, this rule says if `lego_builder(alice)` and `enjoys_lego(alice)` are true then `happy(alice)` must also be true.

The above rule is written in a standard first-order logic notation. We usually write logic programs in reverse implication form:

$$
\texttt{head:-}\ \texttt{body}_1, \texttt{body}_2, \ldots, \texttt{body}_n
$$

⋮

Zahálka, J., & Zelezný, F. (2011). An experimental test of occam's razor in classification. *Machine Learning*, *82*(3), 475–481.

Zelle, J. M., & Mooney, R. J. (1995). Comparative results on using inductive logic programming for corpus-based parser construction. In Wermter, S., Riloff, E., & Scheler, G. (Eds.), *Connectionist, Statistical, and Symbolic Approaches to Learning for Natural Language Processing*, Vol. 1040 of *Lecture Notes in Computer Science*, pp. 355–369. Springer.

Zelle, J. M., & Mooney, R. J. (1996). Learning to parse database queries using inductive logic programming. In Clancey, W. J., & Weld, D. S. (Eds.), *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 2*, pp. 1050–1055. AAAI Press / The MIT Press.

Zeng, Q., Patel, J. M., & Page, D. (2014). Quickfoil: Scalable inductive logic programming. *Proc. VLDB Endow.*, *8*(3), 197–208.

A rule in this form states that the *head* atom is true when every $body_i$ atom is true. A comma denotes conjunction. In logic programming, every variable is assumed to be universally quantified, so we drop quantifiers. We also flip the direction of the implication symbol $\longrightarrow$ to $\longleftarrow$ and often replace it with `:-` because it is easier to use when writing computer programs. Therefore, in logic programming notation, the above hypothesis is:

$$\mathtt{H} = \left\{ \texttt{happy(A):- lego\_builder(A),enjoys\_lego(A).} \right\}$$

Logic programs are *declarative* which means that the order of atoms in a rule does not change its semantics[2]. For instance, the above hypothesis is semantically identical to this one:

$$\mathtt{H} = \left\{ \texttt{happy(A):- enjoys\_lego(A),lego\_builder(A).} \right\}$$

1.2 Scenario 2: Data Curation

Suppose we want to learn a string transformation programs from *input* $\mapsto$ *output* examples, such as program that returns the last character of a string:

Input	**Output**
machine	e
learning	g
algorithm	m

Many forms of ML would represent these examples as a table, such as using a *one-hot-encoding* technique[3]. By contrast, ILP represents these examples as atoms, such as:

$$\mathtt{E}^{+} = \left\{ \begin{array}{l} \texttt{last([m,a,c,h,i,n,e], e).} \\ \texttt{last([l,e,a,r,n,i,n,g], g).} \\ \texttt{last([a,l,g,o,r,i,t,m], m).} \end{array} \right\}$$

The symbol `last` is the *target predicate* that we want to learn (the relation to generalise). The first argument of each atom represents an input list and the second argument represents an output value. To induce a hypothesis for these examples, we need to provide suitable BK, such as common list operations:

Name	**Description**	**Example**
empty(A)	A is an empty list	`empty([]).`
head(A,B)	B is the head of the list A	`head([c,a,t],c).`
tail(A,B)	B is the tail of the list A	`tail([c,a,t],[a,t]).`

Given the aforementioned examples and BK with the above list operations, the goal is to search for a hypothesis that generalises the examples. At a high level, an ILP system builds a hypothesis by combining information from the BK and examples. The set of all possible hypotheses is called the *hypothesis*

[2] This statement about the declarative nature of logic programs is imprecise. The order of rules often matters in practice, such as in Prolog. We discuss this issue in detail in Section 2.3.

[3] Using the simplest binary one-hot-encoding approach, we would have a feature for every letter and an example would have the value 1 for that feature if the letter appears in the example; otherwise, the value will be 0.

space. In other words, the goal of an ILP system is to search the hypothesis space for a hypothesis that generalises the examples.

In this data curation scenario, an ILP system could induce the hypothesis:

$$H = \left\{ \begin{array}{ll} \texttt{last(A,B):-} & \texttt{head(A,B),tail(A,C),empty(C).} \\ \texttt{last(A,B):-} & \texttt{tail(A,C),last(C,B).} \end{array} \right\}$$

This hypothesis contains two rules. The first rule says that B is the last element of A when B is the head of A and the tail of A is empty. The second rule says that B is the last element of A when B is the last element of the tail of A.

1.3 Scenario 3: Program Synthesis

Suppose we have the following positive and negative examples, again represented as atoms, where the first argument is an unsorted list and the second argument is a sorted list:

$$E^+ = \left\{ \begin{array}{l} \texttt{sort([2,1],[1,2]).} \\ \texttt{sort([5,3,1],[1,3,5]).} \end{array} \right\} E^- = \left\{ \begin{array}{l} \texttt{sort([2,1],[2,1]).} \\ \texttt{sort([1,3,1],[1,1,1]).} \end{array} \right\}$$

Also suppose that as BK we have the same `empty`, head, and `tail` relations from the string transformation scenario and two additional relations:

Name	**Description**	**Example**
partition(Pivot,A,L,R)	L is a sublist of A containing elements less than or equal to Pivot and R is a sublist of A containing elements greater than the pivot	`pivot(3,[4,1,5,2],[1,2],[4,5]).`
append(A,B,C)	true when C is the concatenation of A and B	`append([a,b,c],[d,e],[a,b,c,d,e]).`

Given these sets, an ILP system could induce the hypothesis:

$$H = \left\{ \begin{array}{ll} \texttt{sort(A,B):-} & \texttt{empty(A),empty(B).} \\ \texttt{sort(A,B):-} & \texttt{head(A,Pivot),partition(Pivot,A,L1,R1),} \\ & \texttt{sort(L1,L2),sort(R1,R2),append(L2,R2,B).} \end{array} \right\}$$

This hypothesis corresponds to the *quicksort* algorithm (Hoare, 1961) and generalises to lists of arbitrary length and elements not seen in the training examples. This scenario shows that ILP is a form of *program synthesis* (Shapiro, 1983), where the goal is to automatically build executable programs.

1.4 Scenario 4: Scientific Discovery

As Srinivasan et al. (1994) state, *"There is more to scientific theory formulation than data fitting. To be acceptable, a theory must be understandable and open to critical analysis"*. For this reason, ILP has been widely used for scientific discovery. For instance, King et al. (1992) use ILP to model structure-activity relationships for drug design. In this work, an ILP system takes as input positive and negative examples and BK. The positive examples are paired examples of greater activity. For instance, the positive example `great(d20,di5)` states that *drug 20* has higher activity than *drug 15*. Negative examples are examples of drug pairings with lower activity. The BK contains information about the chemical structures of drugs

Struyf, J., Davis, J., & Jr., C. D. P. (2006). An efficient approximation to lookahead in relational learners. In Fürnkranz, J., Scheffer, T., & Spiliopoulou, M. (Eds.), *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*, Vol. 4212 of *Lecture Notes in Computer Science*, pp. 775–782. Springer.

Summers, P. D. (1977). A methodology for LISP program construction from examples. *J. ACM*, *24*(1), 161–175.

Tamaddoni-Nezhad, A., Bohan, D., Raybould, A., & Muggleton, S. (2014). Towards machine learning of predictive models from ecological data. In Davis, J., & Ramon, J. (Eds.), *Inductive Logic Programming - 24th International Conference, ILP 2014, Nancy, France, September 14-16, 2014, Revised Selected Papers*, Vol. 9046 of *Lecture Notes in Computer Science*, pp. 154–167. Springer.

Tamaddoni-Nezhad, A., Chaleil, R., Kakas, A. C., & Muggleton, S. (2006). Application of abductive ILP to learning metabolic network inhibition from temporal data. *Machine Learning*, *64*(1-3), 209–230.

Tamaki, H., & Sato, T. (1984). Unfold/fold transformation of logic programs. In Tärnlund, S. (Ed.), *Proceedings of the Second International Logic Programming Conference, Uppsala University, Uppsala, Sweden, July 2-6, 1984*, pp. 127–138. Uppsala University.

Tärnlund, S. (1977). Horn clause computability. *BIT*, *17*(2), 215–226.

Torrey, L., & Shavlik, J. (2009). Transfer learning. *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, *1*, 242.

Torrey, L., Shavlik, J. W., Walker, T., & Maclin, R. (2007). Relational macros for transfer in reinforcement learning. In Blockeel, H., Ramon, J., Shavlik, J. W., & Tadepalli, P. (Eds.), *Inductive Logic Programming, 17th International Conference, ILP 2007, Corvallis, OR, USA, June 19-21, 2007, Revised Selected Papers*, Vol. 4894 of *Lecture Notes in Computer Science*, pp. 254–268. Springer.

Tourret, S., & Cropper, A. (2019). Sld-resolution reduction of second-order horn fragments. In Calimeri, F., Leone, N., & Manna, M. (Eds.), *Logics in Artificial Intelligence - 16th European Conference, JELIA 2019, Rende, Italy, May 7-11, 2019, Proceedings*, Vol. 11468 of *Lecture Notes in Computer Science*, pp. 259–276. Springer.

Turcotte, M., Muggleton, S., & Sternberg, M. J. E. (2001). The effect of relational background knowledge on learning of protein three-dimensional fold signatures. *Machine Learning*, *43*(1/2), 81–95.

Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, *59*(236), 433–460.

Vera, S. (1975). Induction of concepts in the predicate calculus. In *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, September 3-8, 1975*, pp. 281–287.

Wang, W. Y., Mazaitis, K., & Cohen, W. W. (2014). Structure learning via parameter learning. In Li, J., Wang, X. S., Garofalakis, M. N., Soboroff, I., Suel, T., & Wang, M. (Eds.), *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014*, pp. 1199–1208. ACM.

Wielemaker, J., Schrijvers, T., Triska, M., & Lager, T. (2012). Swi-prolog. *Theory Pract. Log. Program.*, *12*(1-2), 67–96.

Wirth, N. (1985). *Algorithms and data structures*. Prentice Hall.

Wrobel, S. (1996). First-order theory refinement. In *Advances in Inductive Logic Programming*, pp. 14–33.

Conference, ILP 2018, Ferrara, Italy, September 2-4, 2018, Proceedings, Vol. 11105 of *Lecture Notes in Computer Science*, pp. 141–156. Springer.

Silver, D. L., Yang, Q., & Li, L. (2013). Lifelong machine learning systems: Beyond learning algorithms. In *Lifelong Machine Learning, Papers from the 2013 AAAI Spring Symposium, Palo Alto, California, USA, March 25-27, 2013*, Vol. SS-13-05 of *AAAI Technical Report*. AAAI.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, *529*(7587), 484.

Sivaraman, A., Zhang, T., den Broeck, G. V., & Kim, M. (2019). Active inductive logic programming for code search. In Atlee, J. M., Bultan, T., & Whittle, J. (Eds.), *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pp. 292–303. IEEE / ACM.

Solomonoff, R. J. (1964a). A formal theory of inductive inference. part I. *Information and Control*, *7*(1), 1–22.

Solomonoff, R. J. (1964b). A formal theory of inductive inference. part II. *Information and Control*, *7*(2), 224–254.

Sourek, G., Aschenbrenner, V., Zelezný, F., Schockaert, S., & Kuzelka, O. (2018). Lifted relational neural networks: Efficient learning of latent relational structures. *J. Artif. Intell. Res.*, *62*, 69–100.

Srinivasan, A., Muggleton, S., King, R., & Sternberg, M. (1994). Mutagenesis: ILP experiments in a non-determinate biological domain. In Wrobel, S. (Ed.), *Proceedings of the Fourth International Inductive Logic Programming Workshop*. Gesellschaft fur Mathematik und Datenverarbeitung MBH. GMD-Studien Nr 237.

Srinivasan, A. (2001). The ALEPH manual. *Machine Learning at the Computing Laboratory, Oxford University*.

Srinivasan, A., King, R. D., & Bain, M. (2003). An empirical study of the use of relevance information in inductive logic programming. *J. Machine Learning Res.*, *4*, 369–383.

Srinivasan, A., King, R. D., Muggleton, S., & Sternberg, M. J. E. (1997). Carcinogenesis predictions using ILP. In Lavrac, N., & Dzeroski, S. (Eds.), *Inductive Logic Programming, 7th International Workshop, ILP-97, Prague, Czech Republic, September 17-20, 1997, Proceedings*, Vol. 1297 of *Lecture Notes in Computer Science*, pp. 273–287. Springer.

Srinivasan, A., Muggleton, S., Sternberg, M. J. E., & King, R. D. (1996). Theories for mutagenicity: A study in first-order and feature-based induction. *Artif. Intell.*, *85*(1-2), 277–299.

Srinivasan, A., Muggleton, S. H., & King, R. D. (1995). Comparing the use of background knowledge by inductive logic programming systems. In *Proceedings of the 5th International Workshop on Inductive Logic Programming*, pp. 199–230. Department of Computer Science, Katholieke Universiteit Leuven.

Srinivasan, A., Page, D., Camacho, R., & King, R. D. (2006). Quantitative pharmacophore models with inductive logic programming. *Machine Learning*, *64*(1-3), 65–90.

Stahl, I. (1995). The appropriateness of predicate invention as bias shift operation in ILP. *Machine Learning*, *20*(1-2), 95–117.

Sterling, L., & Shapiro, E. Y. (1994). *The art of Prolog: advanced programming techniques*. MIT press.

and the properties of *substituents*[4]. For instance, the atom struc(d35,no2,nhcoch3,h) states that *drug 35* has *no2* substituted at position *3*, *nhcoch3* substituted at position 4, and no substitution at position 5, and the atom flex(no2,3) states that *no2* has flexibility 3. Given such examples and BK, the ILP system Golem (Muggleton & Feng, 1990) induces multiple rules to explain the examples, including this one:

```
great(A,B):- struc(A,C,D,E),struc(B,F,h,h),h_donor(C,hdon0),
              polarisable(C,polari1),flex(F,G),flex(C,H),
              great_flex(G,H),great6_flex(G).
```

This rule says that Drug A is better than drug B if drug B has no substitutions at positions 4 and 5, and drug B at position 3 has flexibility >6, and drug A at position 3 has polarisability = 1, and drug A at position 3 has hydrogen donor = 0, and drug A at position 3 is less flexible than drug B at position 3.

As this scenario illustrates, ILP can learn human-readable hypotheses. The interpretability of such rules is crucial to allow domain experts to gain insight.

1.5 Why ILP?

Most ML approaches rely on statistical inference. By contrast, ILP relies on logical inference and uses techniques from *automated reasoning* and *knowledge representation*. Table 2 shows a simplified comparison between ILP and statistical ML approaches. We briefly discuss these differences.

	Statistical ML	**ILP**
Examples	Many	Few
Data	Tables	Logic programs
Hypotheses	Propositional/functions	First-/higher-order relations
Explainability	Difficult	Possible
Knowledge transfer	Difficult	Easy

Table 2: A simplified comparison between ILP and statistical ML approaches based on the table by Gulwani et al. (2015).

Examples. Many forms of ML are notorious for their inability to generalise from small numbers of training examples, notably deep learning (Marcus, 2018; Chollet, 2019; Bengio et al., 2019). As Evans and Grefenstette (2018) point out, if we train a neural system to add numbers with 10 digits, it can generalise to numbers with 20 digits, but when tested on numbers with 100 digits, the predictive accuracy drastically decreases (Reed & de Freitas, 2016; Kaiser & Sutskever, 2016). By contrast, ILP can induce hypotheses from small numbers of examples, often from a single example (Lin et al., 2014; Muggleton et al., 2018). This data efficiency is important when we only have small amounts of training data. For instance, Gulwani (2011) applies techniques similar to ILP to induce programs from user-provided examples in Microsoft Excel to solve string transformation problems, where it is infeasible to ask a user for thousands of examples. This data efficiency has made ILP attractive in many real-world applications, especially in drug design, where large numbers of examples are not always easy to obtain.

[4] An atom or group other than hydrogen on a molecule.

Data. Using logic programs to represent data allows ILP to learn with complex relational information and for easy integration of expert knowledge. For instance, if learning causal relations in causal networks, a user can encode constraints about the network (Inoue et al., 2013). If learning to recognise events, a user could provide the axioms of the event calculus (Katzouris et al., 2015). Relational BK allows us to succinctly represent infinite relations. For instance, it is trivial to define a summation relation over the infinite set of natural numbers (`add(A,B,C):- C = A+B`). By contrast, tabled-based ML approaches are mostly restricted to finite data and cannot represent this information. For instance, it is impossible to provide a decision tree learner (Quinlan, 1986, 1993) this infinite relation because it would require an infinite feature table. Even if we restricted ourselves to a finite set of n natural numbers, a table-based approach would still need n^3 features to represent the complete summation relation.

Hypotheses. Because they are closely related to relational databases, logic programs naturally support relational data such as graphs. Because of the expressivity of logic programs, ILP can learn complex relational theories, such as cellular automata (Inoue et al., 2014; Evans et al., 2021), event calculus theories (Katzouris et al., 2015), and Petri nets (Bain & Srinivasan, 2018), and various forms of non-monotonic programs (Bain & Muggleton, 1991; Inoue & Kudoh, 1997; Sakama, 2001). Because of the symbolic nature of logic programs, ILP can reason about hypotheses, which allows it to learn *optimal* programs, such as minimal time-complexity programs (Cropper & Muggleton, 2019) and secure access control policies (Law et al., 2020). Moreover, because induced hypotheses have the same language as the BK, they can be stored in the BK, making transfer learning trivial (Lin et al., 2014).

Explainability. Because of logic's similarity to natural language, logic programs can be easily read by humans, which is crucial for explainable AI[5]. Because of this interpretability, ILP has long been used for *scientific discovery*[6] (King et al., 1992; Srinivasan et al., 1996, 1997, 2006; Kaalia et al., 2016). For instance, the *Robot Scientist* (King et al., 2004) is a system that uses ILP to generate hypotheses to explain data and can also automatically devise experiments to test the hypotheses, physically run the experiments, interpret the results, and then repeat the cycle. Whilst researching yeast-based functional genomics, the Robot Scientist became the first machine to independently discover new scientific knowledge (King et al., 2009).

Knowledge transfer. Most ML algorithms are single-task learners and cannot reuse learned knowledge. For instance, although AlphaGo (Silver et al., 2016) has super-human Go ability, it cannot reuse this knowledge to play other games, nor the same game with a slightly different board. By contrast, because of its symbolic representation, ILP naturally supports lifelong and transfer learning (Torrey et al., 2007; Cropper, 2019), which is considered essential for human-like AI (Lake et al., 2016). For instance, when inducing solutions to a set of string transformation tasks, such as those in Scenario 2, Lin et al. (2014) show that an ILP system can automatically identify easier problems to solve, learn programs for them, and then *reuse* the learned programs to help learn programs for more difficult problems. Moreover, they show that this knowledge transfer approach leads to a hierarchy of reusable programs, where each program builds on simpler programs.

[5] Muggleton et al. (2018) (also explored by Ai et al. (2021)) evaluate the comprehensibility of ILP hypotheses using Michie's (1988) notion of *ultra-strong ML*, where a learned hypothesis is expected to not only be accurate but to also demonstrably improve the performance of a human when provided with the learned hypothesis.

[6] Muggleton (1999b) provides a (slightly outdated) summary of scientific discovery using ILP.

Rocktäschel, T., & Riedel, S. (2017). End-to-end differentiable proving. In Guyon, I., von Luxburg, U., Bengio, S., Wallach, H. M., Fergus, R., Vishwanathan, S. V. N., & Garnett, R. (Eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pp. 3788–3800.

Russell, S. (2019). *Human compatible: Artificial intelligence and the problem of control*. Penguin.

Sakama, C. (2001). Nonmonotonic inductive logic programming. In Eiter, T., Faber, W., & Truszczynski, M. (Eds.), *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17-19, 2001, Proceedings*, Vol. 2173 of *Lecture Notes in Computer Science*, pp. 62–80. Springer.

Sakama, C., & Inoue, K. (2009). Brave induction: a logical framework for learning from incomplete information. *Machine Learning*, *76*(1), 3–35.

Sammut, C. (1981). Concept learning by experiment. In Hayes, P. J. (Ed.), *Proceedings of the 7th International Joint Conference on Artificial Intelligence, IJCAI '81, Vancouver, BC, Canada, August 24-28, 1981*, pp. 104–105. William Kaufmann.

Sammut, C. (1993). The origins of inductive logic programming: A prehistoric tale. In *Proceedings of the 3rd international workshop on inductive logic programming*, pp. 127–147. J. Stefan Institute.

Sammut, C., Sheh, R., Haber, A., & Wicaksono, H. (2015). The robot engineer. In Inoue, K., Ohwada, H., & Yamamoto, A. (Eds.), *Late Breaking Papers of the 25th International Conference on Inductive Logic Programming, Kyoto University, Kyoto, Japan, August 20th to 22nd, 2015*, Vol. 1636 of *CEUR Workshop Proceedings*, pp. 101–106. CEUR-WS.org.

Sato, T. (1995). A statistical learning method for logic programs with distribution semantics. In Sterling, L. (Ed.), *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*, pp. 715–729. MIT Press.

Sato, T., & Kameya, Y. (2001). Parameter learning of logic programs for symbolic-statistical modeling. *J. Artif. Intell. Res.*, *15*, 391–454.

Schaffer, C. (1993). Overfitting avoidance as bias. *Machine Learning*, *10*, 153–178.

Schüller, P., & Benz, M. (2018). Best-effort inductive logic programming via fine-grained cost-based hypothesis generation - the inspire system at the inductive logic programming competition. *Machine Learning.*, *107*(7), 1141–1169.

Shapiro, E. Y. (1983). *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA.

Si, X., Lee, W., Zhang, R., Albarghouthi, A., Koutris, P., & Naik, M. (2018). Syntax-guided synthesis of Datalog programs. In Leavens, G. T., Garcia, A., & Pasareanu, C. S. (Eds.), *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pp. 515–527. ACM.

Si, X., Raghothaman, M., Heo, K., & Naik, M. (2019). Synthesizing datalog programs using numerical relaxation. In Kraus, S. (Ed.), *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pp. 6117–6124. ijcai.org.

Siebers, M., & Schmid, U. (2018). Was the year 2000 a leap year? step-wise narrowing theories with metagol. In Riguzzi, F., Bellodi, E., & Zese, R. (Eds.), *Inductive Logic Programming - 28th International*

Otero, R. P. (2001). Induction of stable models. In Rouveirol, C., & Sebag, M. (Eds.), *Inductive Logic Programming, 11th International Conference, ILP 2001, Strasbourg, France, September 9-11, 2001, Proceedings*, Vol. 2157 of *Lecture Notes in Computer Science*, pp. 193–205. Springer.

Page, D., & Srinivasan, A. (2003). ILP: A short look back and a longer look forward. *J. Machine Learning Res.*, *4*, 415–430.

Picado, J., Termehchy, A., Fern, A., & Pathak, S. (2017). Towards automatically setting language bias in relational learning. In Schelter, S., & Zadeh, R. (Eds.), *Proceedings of the 1st Workshop on Data Management for End-to-End Machine Learning, DEEM@SIGMOD 2017, Chicago, IL, USA, May 14, 2017*, pp. 3:1–3:4. ACM.

Plotkin, G. (1971). *Automatic Methods of Inductive Inference*. Ph.D. thesis, Edinburgh University.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, *1*(1), 81–106.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, *5*, 239–266.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

Raedt, L. D., Dumancic, S., Manhaeve, R., & Marra, G. (2020). From statistical relational to neuro-symbolic artificial intelligence. In Bessiere, C. (Ed.), *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pp. 4943–4950. ijcai.org.

Raghothaman, M., Mendelson, J., Zhao, D., Naik, M., & Scholz, B. (2020). Provenance-guided synthesis of datalog programs. *Proc. ACM Program. Lang.*, *4*(POPL), 62:1–62:27.

Ray, O. (2009). Nonmonotonic abductive inductive learning. *J. Applied Logic*, *7*(3), 329–340.

Reed, S. E., & de Freitas, N. (2016). Neural programmer-interpreters. In Bengio, Y., & LeCun, Y. (Eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

Reiter, R. (1977). On closed world data bases. In Gallaire, H., & Minker, J. (Eds.), *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977*, Advances in Data Base Theory, pp. 55–76, New York. Plemum Press.

Ribeiro, T., Folschette, M., Magnin, M., & Inoue, K. (2020). Learning any semantics for dynamical systems represented by logic programs. working paper or preprint.

Ribeiro, T., & Inoue, K. (2014). Learning prime implicant conditions from interpretation transition. In Davis, J., & Ramon, J. (Eds.), *Inductive Logic Programming - 24th International Conference, ILP 2014, Nancy, France, September 14-16, 2014, Revised Selected Papers*, Vol. 9046 of *Lecture Notes in Computer Science*, pp. 108–125. Springer.

Ribeiro, T., Magnin, M., Inoue, K., & Sakama, C. (2015). Learning multi-valued biological models with delayed influence from time-series observations. In Li, T., Kurgan, L. A., Palade, V., Goebel, R., Holzinger, A., Verspoor, K., & Wani, M. A. (Eds.), *14th IEEE International Conference on Machine Learning and Applications, ICMLA 2015, Miami, FL, USA, December 9-11, 2015*, pp. 25–31. IEEE.

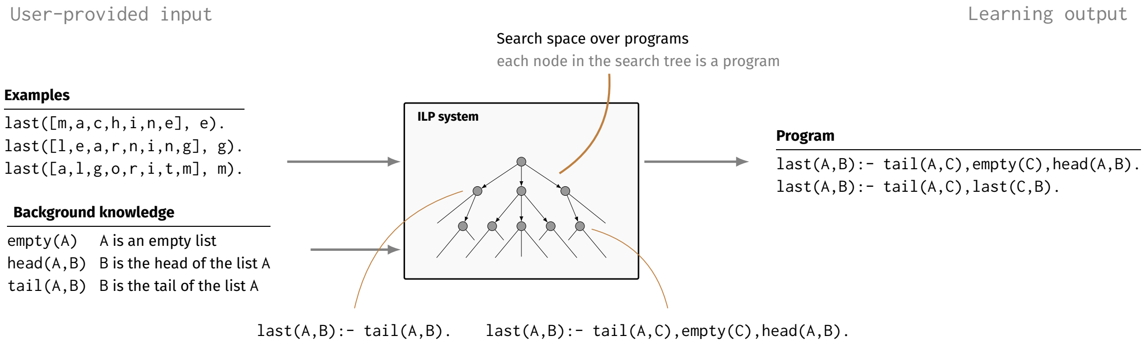
Richards, B. L., & Mooney, R. J. (1995). Automated refinement of first-order horn-clause domain theories. *Machine Learning.*, *19*(2), 95–131.

Richardson, M., & Domingos, P. M. (2006). Markov logic networks. *Machine Learning*, *62*(1-2), 107–136.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *J. ACM*, *12*(1), 23–41.

1.6 How Does ILP Work?

Building an ILP system (Figure 1) requires making several choices or assumptions. Understanding these assumptions is key to understanding ILP. We discuss these assumptions in Section 4 but briefly summarise them now.



```
last([m,a,c,h,i,n,e], e).
last([l,e,a,r,n,i,n,g], g).
last([a,l,g,o,r,i,t,m], m).
```

```
empty(A)    A is an empty list
head(A,B)   B is the head of the list A
tail(A,B)   B is the tail of the list A
```

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

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

Figure 1: An ILP system learns programs from examples and BK. The system learns by searching a space of possible programs which are constructed from BK.

Learning setting. The central choice is how to represent examples. The examples in the scenarios in this section include boolean concepts (*lego_builder*) and input-output examples (string transformation and sorting). Although boolean concepts and input-output examples are common representations, there are other representations, such as interpretations (Blockeel & De Raedt, 1998) and transitions (Inoue et al., 2014). The representation determines the learning setting which in turn defines what it means for a program to solve the ILP problem.

Representation language. ILP represents data as logic programs. There are, however, many logic programming languages, each with strengths and weaknesses. For instance, Prolog is a Turing-complete logic programming language. Datalog is a syntactical subset of Prolog that sacrifices features (such as data structures) and expressivity (it is not Turing-complete) to gain efficiency and decidability. Some languages support non-monotonic reasoning, such as answer set programming (Gebser et al., 2012). Choosing a suitable representation language is crucial in determining which problems a system can solve.

Defining the hypothesis space. The fundamental ILP problem is to search the hypothesis space for a suitable hypothesis. The hypothesis space contains all possible programs that can be built in the chosen representation language. Unrestricted, the hypothesis space is infinite, so it is vital to restrict it to make the search feasible. As with all ML techniques, ILP restricts the hypothesis space by enforcing an *inductive bias* (Mitchell, 1997). A *language bias* enforces restrictions on hypotheses, such as how many variables or relations can be in a hypothesis. Choosing an appropriate language bias is necessary for efficient learning and is a major challenge.

Search method. Having defined the hypothesis space, the problem is to efficiently search it. The traditional way to categorise approaches is whether they use a *top-down* or *bottom-up* search, where *generality* orders the search space[7]. Top-down approaches (Quinlan, 1990; Blockeel & De Raedt, 1998; Bratko,

[7] A hypothesis h_1 is more general than h_2 if h_1 entails at least as many examples as h_2. A hypothesis h_1 is more specific than h_2 if h_1 entails fewer examples then h_2.

1999; Muggleton et al., 2008; Ribeiro & Inoue, 2014) start with an overly general hypothesis and try to specialise it. Bottom-up approaches (Muggleton, 1987; Muggleton & Buntine, 1988; Muggleton & Feng, 1990; Inoue et al., 2014) start with an overly specific hypothesis and try to generalise it. Some approaches combine the two (Muggleton, 1995; Srinivasan, 2001; Cropper, 2022). A third approach has recently emerged called *meta-level* ILP (Inoue et al., 2013; Muggleton et al., 2015; Inoue, 2016; Law et al., 2020; Cropper & Morel, 2021). This approach represents an ILP problem as a *meta-level* logic program, i.e. a program that reasons about programs. Meta-level approaches often delegate the search for a hypothesis to an off-the-shelf solver (Corapi et al., 2011; Muggleton et al., 2014; Law et al., 2014; Kaminski et al., 2018; Evans et al., 2021; Cropper & Morel, 2021) after which the meta-level solution is translated back to a standard solution for the ILP problem.

1.7 A Brief History

That ILP is a form of ML surprises many researchers who only associate ML with statistical techniques. However, if we follow Mitchell's (1997) definition of ML[8] then ILP is no different from other ML approaches: it improves given more examples. The confusion seems to come from ILP's use of logic as a representation for learning. However, as Domingos (2015) points out, there are generally five areas of ML: symbolists, connectionists, Bayesian, analogisers, and evolutionists. ILP is in the symbolic learning category.

Turing can be seen as one of the first symbolist, as he proposed using a logical representation to build thinking machines (Turing, 1950; Muggleton, 1994a). McCarthy (1959) made the first comprehensive proposal for the use of logic in AI with his *advice seeker*. Much work on using logic for ML soon followed. Recognising the limitations of table-based representations, Banerji (1964) proposed using predicate logic as a representation language for learning. Michalski's (1969) work on the AQ algorithm, which induces rules using a set covering algorithm, has greatly influenced many ILP systems. Plotkin's (1971) work on subsumption and least general generalisation has influenced nearly all of ILP, especially theory. Other notable work includes Vera (1975) on induction algorithms for predicate calculus and Sammut's (1981) MARVIN system, one of the first to learn executable programs. Shapiro's (1983) work on inducing Prolog programs made major contributions to ILP, including the concepts of backtracking and refinement operators. Quinlan's (1990) FOIL system is one of the most well-known ILP systems and is a natural extension of ID3 (Quinlan, 1986) from the propositional setting to the first-order setting. Other notable contributions include *inverse resolution* (Muggleton & Buntine, 1988), which was also one of the earliest approaches at predicate invention. ILP as a field was founded by Muggleton (1991), who stated that it lies at the intersection of ML and knowledge representation.

1.8 Contributions

There are several excellent ILP survey papers (Sammut, 1993; Muggleton & De Raedt, 1994; Muggleton, 1999a; Page & Srinivasan, 2003; Muggleton et al., 2012) and books (Nienhuys-Cheng & Wolf, 1997; De Raedt, 2008). In this paper, we want to provide a new introduction to the field aimed at a general AI reader interested in symbolic learning. We differ from existing surveys by including, and mostly focusing on, recent developments (Cropper et al., 2020a), such as new methods for learning recursive programs, predicate invention, and meta-level search. Although we cover work on inducing Datalog and answer set programs, we mostly focus on approaches that induce definite programs, and in particular Prolog

[8] According to Mitchell's (1997), an algorithm is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Muggleton, S., & De Raedt, L. (1994). Inductive logic programming: Theory and methods. *J. Log. Program.*, *19/20*, 629–679.

Muggleton, S., De Raedt, L., Poole, D., Bratko, I., Flach, P. A., Inoue, K., & Srinivasan, A. (2012). ILP turns 20 - biography and future challenges. *Machine Learning*, *86*(1), 3–23.

Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. In *Algorithmic Learning Theory, First International Workshop, ALT '90, Tokyo, Japan, October 8-10, 1990, Proceedings*, pp. 368–381.

Muggleton, S., & Firth, J. (2001). Relational rule induction with cp rogol 4.4: A tutorial introduction. In *Relational data mining*, pp. 160–188. Springer.

Muggleton, S., Paes, A., Costa, V. S., & Zaverucha, G. (2009). Chess revision: Acquiring the rules of chess variants through FOL theory revision from examples. In De Raedt, L. (Ed.), *Inductive Logic Programming, 19th International Conference, ILP 2009, Leuven, Belgium, July 02-04, 2009. Revised Papers*, Vol. 5989 of *Lecture Notes in Computer Science*, pp. 123–130. Springer.

Muggleton, S., Santos, J. C. A., & Tamaddoni-Nezhad, A. (2008). Toplog: ILP using a logic program declarative bias. In de la Banda, M. G., & Pontelli, E. (Eds.), *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, Vol. 5366 of *Lecture Notes in Computer Science*, pp. 687–692. Springer.

Muggleton, S., Santos, J. C. A., & Tamaddoni-Nezhad, A. (2009). Progolem: A system based on relative minimal generalisation. In De Raedt, L. (Ed.), *Inductive Logic Programming, 19th International Conference, ILP 2009, Leuven, Belgium, July 02-04, 2009. Revised Papers*, Vol. 5989 of *Lecture Notes in Computer Science*, pp. 131–148. Springer.

Muggleton, S., & Tamaddoni-Nezhad, A. (2008). QG/GA: a stochastic search for progol. *Machine Learning*, *70*(2-3), 121–133.

Muggleton, S. H., Lin, D., Chen, J., & Tamaddoni-Nezhad, A. (2013). Metabayes: Bayesian meta-interpretative learning using higher-order stochastic refinement. In *Inductive Logic Programming - 23rd International Conference, ILP 2013, Rio de Janeiro, Brazil, August 28-30, 2013, Revised Selected Papers*, pp. 1–17.

Muggleton, S. H., Lin, D., Pahlavi, N., & Tamaddoni-Nezhad, A. (2014). Meta-interpretive learning: application to grammatical inference. *Machine Learning*, *94*(1), 25–49.

Muggleton, S. H., Lin, D., & Tamaddoni-Nezhad, A. (2011). Mc-toplog: Complete multi-clause learning guided by a top theory. In Muggleton, S., Tamaddoni-Nezhad, A., & Lisi, F. A. (Eds.), *Inductive Logic Programming - 21st International Conference, ILP 2011, Windsor Great Park, UK, July 31 - August 3, 2011, Revised Selected Papers*, Vol. 7207 of *Lecture Notes in Computer Science*, pp. 238–254. Springer.

Muggleton, S. H., Lin, D., & Tamaddoni-Nezhad, A. (2015). Meta-interpretive learning of higher-order dyadic Datalog: predicate invention revisited. *Machine Learning*, *100*(1), 49–73.

Muggleton, S. H., Schmid, U., Zeller, C., Tamaddoni-Nezhad, A., & Besold, T. R. (2018). Ultra-strong machine learning: comprehensibility of programs learned with ILP. *Machine Learning*, *107*(7), 1119–1140.

Nienhuys-Cheng, S.-H., & Wolf, R. d. (1997). *Foundations of Inductive Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Osera, P., & Zdancewic, S. (2015). Type-and-example-directed program synthesis. In Grove, D., & Blackburn, S. (Eds.), *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pp. 619–630. ACM.

Martínez, D., Ribeiro, T., Inoue, K., Alenyà, G., & Torras, C. (2015). Learning probabilistic action models from interpretation transitions. In Vos, M. D., Eiter, T., Lierler, Y., & Toni, F. (Eds.), *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), Cork, Ireland, August 31 - September 4, 2015*, Vol. 1433 of *CEUR Workshop Proceedings*. CEUR-WS.org.

McCarthy, J. (1959). Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pp. 75–91, London. Her Majesty's Stationary Office.

McCreath, E., & Sharma, A. (1995). Extraction of meta-knowledge to restrict the hypothesis space for ilp systems. In *Eighth Australian Joint Conference on Artificial Intelligence*, pp. 75–82.

Michalski, R. S. (1969). On the quasi-minimal solution of the general covering problem..

Michie, D. (1988). Machine learning in the next five years. In Sleeman, D. H. (Ed.), *Proceedings of the Third European Working Session on Learning, EWSL 1988, Turing Institute, Glasgow, UK, October 3-5, 1988*, pp. 107–122. Pitman Publishing.

Mitchell, T. M. (1997). *Machine learning*. McGraw Hill series in computer science. McGraw-Hill.

Mooney, R. J. (1999). Learning for semantic interpretation: Scaling up without dumbing down. In Cussens, J., & Dzeroski, S. (Eds.), *Learning Language in Logic*, Vol. 1925 of *Lecture Notes in Computer Science*, pp. 57–66. Springer.

Mooney, R. J., & Califf, M. E. (1995). Induction of first-order decision lists: Results on learning the past tense of english verbs. *J. Artif. Intell. Res.*, *3*, 1–24.

Morales, E. M. (1996). Learning playing strategies in chess. *Computational Intelligence*, *12*, 65–87.

Morel, R., Cropper, A., & Ong, C. L. (2019). Typed meta-interpretive learning of logic programs. In Calimeri, F., Leone, N., & Manna, M. (Eds.), *Logics in Artificial Intelligence - 16th European Conference, JELIA 2019, Rende, Italy, May 7-11, 2019, Proceedings*, Vol. 11468 of *Lecture Notes in Computer Science*, pp. 198–213. Springer.

Muggleton, S. (1987). Duce, an oracle-based approach to constructive induction. In McDermott, J. P. (Ed.), *Proceedings of the 10th International Joint Conference on Artificial Intelligence. Milan, Italy, August 23-28, 1987*, pp. 287–292. Morgan Kaufmann.

Muggleton, S. (1991). Inductive logic programming. *New Generation Computing*, *8*(4), 295–318.

Muggleton, S. (1994a). Logic and learning: Turing's legacy. In *Machine Intelligence 13*, pp. 37–56.

Muggleton, S. (1994b). Predicate invention and utilization. *J. Exp. Theor. Artif. Intell.*, *6*(1), 121–130.

Muggleton, S. (1995). Inverse entailment and progol. *New Generation Comput.*, *13*(3&4), 245–286.

Muggleton, S. (1999a). Inductive logic programming: Issues, results and the challenge of learning language in logic. *Artif. Intell.*, *114*(1-2), 283–296.

Muggleton, S. (1999b). Scientific knowledge discovery using inductive logic programming. *Commun. ACM*, *42*(11), 42–46.

Muggleton, S., & Buntine, W. L. (1988). Machine invention of first order predicates by inverting resolution. In Laird, J. E. (Ed.), *Machine Learning, Proceedings of the Fifth International Conference on Machine Learning, Ann Arbor, Michigan, USA, June 12-14, 1988*, pp. 339–352. Morgan Kaufmann.

Muggleton, S., Dai, W., Sammut, C., Tamaddoni-Nezhad, A., Wen, J., & Zhou, Z. (2018). Meta-interpretive learning from noisy images. *Machine Learning*, *107*(7), 1097–1118.

programs. We do not detail work that combines neural networks with ILP, to which there are already suitable survey papers (d'Avila Garcez et al., 2019; Raedt et al., 2020).

The rest of the paper is organised as follows:

- We describe necessary logic programming notation (Section 2).
- We define the standard ILP learning settings (Section 3).
- We describe the basic assumptions required to build an ILP system (Section 4).
- We compare many ILP systems and describe the features they support (Section 5).
- We describe four ILP systems in detail (Aleph, TILDE, ASPAL, and Metagol) (Section 6).
- We summarise some of the key application areas of ILP (Section 7).
- We briefly survey related work (Section 8).
- We conclude by outlining the main current limitations of ILP and suggesting directions for future research (Section 9)

2. Logic Programming

ILP uses logic programs (Kowalski, 1974) to represent BK, examples, and hypotheses. A logic program is fundamentally different from an imperative program (e.g. C, Java, Python) and very different from a functional program (e.g. Haskell, OCaml). Imperative programming views a program as a sequence of step-by-step instructions where computation is the process of executing the instructions. By contrast, logic programming views a program as a logical theory (a set of logical rules) where computation is various forms of deduction over the theory, such as searching for a proof, refutation, or a model of it. Another major difference is that a logic program is *declarative* (Lloyd, 1994) because it allows a user to state *what* a program should do, rather than *how* it should work. This declarative nature means that the order of rules in a logic program does not (usually) matter.

In the rest of this section, we introduce the basics of logic programming necessary to understand the rest of this paper. We cover the syntax and semantics and briefly introduce different logic programming languages. We focus on concepts necessary for understanding ILP and refer the reader to more detailed expositions of logic programming (Nienhuys-Cheng & Wolf, 1997; De Raedt, 2008; Lloyd, 2012), Prolog (Sterling & Shapiro, 1994; Bratko, 2012), and ASP (Gebser et al., 2012) for more information. We, therefore, omit descriptions of many important concepts in logic programming, such as stratified negation. Readers comfortable with logic can skip this section.

2.1 Syntax

We first define the syntax of a logic program:

- A *variable* is a string of characters starting with an uppercase letter, e.g. A, B, and C.
- A *function* symbol is a string of characters starting with a lowercase letter.
- A *predicate* symbol is a string of characters starting with a lowercase letter, e.g. `job` or happy. The *arity* n of a function or predicate symbol p is the number of arguments it takes and is denoted as p/n, e.g. $\mathrm{happy}/1$, $\mathrm{head}/2$, and $\mathrm{append}/3$.
- A *constant* symbol is a function symbol with zero arity, e.g. `alice` or bob.
- A *term* is a variable, or a constant/function symbol of arity n immediately followed by a tuple of n terms.

- A term is *ground* if it contains no variables.
- An *atom* is a formula $p(t_1, \ldots, t_n)$, where p is a predicate symbol of arity n and each t_i is a term, e.g. `lego_builder(alice)`, where `lego_builder` is a predicate symbol of arity 1 and `alice` is a constant symbol.
- An atom is *ground* if all of its terms are ground, e.g. `lego_builder(alice)` is ground but `lego_builder(A)`, where `A` is a variable, is not ground.
- The symbol `not` denotes *negation as failure*, where an atom is false if it cannot be proven true.
- A *literal* is an atom `A` (a *positive literal*) or its negation `not A` (a *negative literal*). For instance, `lego_builder(alice)` is both an atom and a literal but `not lego_builder(alice)` is only a literal because it includes the negation symbol `not`.
- A *clause* is of the form h_1, ..., h_n :- b_1, ..., b_m where each h_i and b_j is a literal and the symbol , denotes conjunction. The symbols h_i are called the *head* of the clause. The symbols b_i are called the *body* of the clause. We sometimes use the name *rule* instead of *clause*.
- A *Horn* clause is a clause with at most one positive literal.
- A *definite* clause is clause of the form h :- b_1, b_2, ..., b_n, i.e. a clause with only one head literal, e.g. `qsort(A,B):- empty(A),empty(B).` Informally, a definite clause states that the head is true if the body is true, i.e. all of the body literals are proven true. For instance, the rule `happy(A):- lego_builder(A),enjoys_lego(A)` says that `happy(A)` is true when both `lego_builder(A)` and `enjoys_lego(A)` are true.
- A clause is *ground* if it contains no variables.
- A clausal *theory* is a set of clauses.
- A *goal* (also called a *constraint*) is a clause of the form :- b_1, b_2, ..., b_n, i.e. a clause without a head, e.g. `:- head(A,B),head(B,A).`
- A *unit* clause is a clause with no body. For unit clauses, we usually omit the `:-` symbol, e.g. `loves(alice,X).`
- A *fact* is a ground unit clause `loves(andrew,laura).`
- Simultaneously replacing variables $v_1, \ldots, v_n$ in a clause with terms $t_1, \ldots, t_n$ is called a *substitution* and is denoted as $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$. For instance, applying the substitution $\theta = \{A/bob\}$ to `loves(alice,A)` results in `loves(alice,bob)`.
- A substitution θ *unifies* atoms A and B in the case $A\theta = B\theta$. Note that atoms A and B need to have a distinct set of variables, i.e., they should not have a variable with the same name, for unification to work properly.

2.2 Semantics

The semantics of logic programs is based on the concepts of a Herbrand *universe*, *base*, and *interpretation*. All three concepts build upon a given *vocabulary* $\mathcal{V}$ containing all constants, functions, and predicate symbols of a program. The Herbrand universe is the set of all ground terms that can be formed from the constants and functions symbols in $\mathcal{V}$. For instance, the Herbrand universe of the lego builder example (Section 1.1) is

```
{alice, bob, claire, dave}
```

If the example also contained the function symbol `age/1`, then the Herbrand universe would be the infinite set:

Law, M., Russo, A., & Broda, K. (2016). Iterative learning of answer set programs from context dependent examples. *Theory Pract. Log. Program.*, *16*(5-6), 834–848.

Law, M., Russo, A., & Broda, K. (2018). The complexity and generality of learning answer set programs. *Artif. Intell.*, *259*, 110–146.

Law, M., Russo, A., & Broda, K. (2020). The ilasp system for inductive learning of answer set programs. *The Association for Logic Programming Newsletter*.

Leban, G., Zabkar, J., & Bratko, I. (2008). An experiment in robot discovery with ILP. In Zelezný, F., & Lavrac, N. (Eds.), *Inductive Logic Programming, 18th International Conference, ILP 2008, Prague, Czech Republic, September 10-12, 2008, Proceedings*, Vol. 5194 of *Lecture Notes in Computer Science*, pp. 77–90. Springer.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.

Legras, S., Rouveirol, C., & Ventos, V. (2018). The game of bridge: A challenge for ILP. In Riguzzi, F., Bellodi, E., & Zese, R. (Eds.), *Inductive Logic Programming - 28th International Conference, ILP 2018, Ferrara, Italy, September 2-4, 2018, Proceedings*, Vol. 11105 of *Lecture Notes in Computer Science*, pp. 72–87. Springer.

Levin, L. A. (1973). Universal sequential search problems. *Problemy Peredachi Informatsii*, *9*(3), 115–116.

Lieberman, H. (2001). Your wish is my command: Programming by example..

Lin, D., Dechter, E., Ellis, K., Tenenbaum, J. B., & Muggleton, S. (2014). Bias reformulation for one-shot function induction. In Schaub, T., Friedrich, G., & O'Sullivan, B. (Eds.), *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, Vol. 263 of *Frontiers in Artificial Intelligence and Applications*, pp. 525–530. IOS Press.

Lloyd, J. W. (1994). Practical advtanages of declarative programming. In *1994 Joint Conference on Declarative Programming, GULP-PRODE'94 Peñiscola, Spain, September 19-22, 1994, Volume 1*, pp. 18–30.

Lloyd, J. W. (2012). *Foundations of logic programming*. Springer Science & Business Media.

Maher, M. J. (1988). Equivalences of logic programs. In Minker, J. (Ed.), *Foundations of Deductive Databases and Logic Programming*, pp. 627–658. Morgan Kaufmann.

Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T., & De Raedt, L. (2018). Deepproblog: Neural probabilistic logic programming. In Bengio, S., Wallach, H. M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., & Garnett, R. (Eds.), *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pp. 3753–3763.

Manna, Z., & Waldinger, R. J. (1980). A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, *2*(1), 90–121.

Marcus, G. (2018). Deep learning: A critical appraisal. *CoRR*, *abs/1801.00631*.

Martínez, D., Alenyà, G., Torras, C., Ribeiro, T., & Inoue, K. (2016). Learning relational dynamics of stochastic domains for planning. In Coles, A. J., Coles, A., Edelkamp, S., Magazzeni, D., & Sanner, S. (Eds.), *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016, London, UK, June 12-17, 2016*, pp. 235–243. AAAI Press.

Kaur, N., Kunapuli, G., Joshi, S., Kersting, K., & Natarajan, S. (2019). Neural networks for relational data. In Kazakov, D., & Erten, C. (Eds.), *Inductive Logic Programming - 29th International Conference, ILP 2019, Plovdiv, Bulgaria, September 3-5, 2019, Proceedings*, Vol. 11770 of *Lecture Notes in Computer Science*, pp. 62–71. Springer.

Kaur, N., Kunapuli, G., & Natarajan, S. (2020). Non-parametric learning of lifted restricted boltzmann machines. *Int. J. Approx. Reason.*, *120*, 33–47.

Kietz, J.-U., & Wrobel, S. (1992). Controlling the complexity of learning in logic through syntactic and task-oriented models. In *Inductive logic programming*. Citeseer.

King, R. D., Muggleton, S., Lewis, R. A., & Sternberg, M. J. (1992). Drug design by machine learning: the use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. *Proceedings of the National Academy of Sciences*, *89*(23), 11322–11326.

King, R. D., Rowland, J., Oliver, S. G., Young, M., Aubrey, W., Byrne, E., Liakata, M., Markham, M., Pir, P., Soldatova, L. N., et al. (2009). The automation of science. *Science*, *324*(5923), 85–89.

King, R. D., Whelan, K. E., Jones, F. M., Reiser, P. G., Bryant, C. H., Muggleton, S. H., Kell, D. B., & Oliver, S. G. (2004). Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature*, *427*(6971), 247–252.

Kowalski, R. A. (1974). Predicate logic as programming language. In Rosenfeld, J. L. (Ed.), *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, pp. 569–574. North-Holland.

Kowalski, R. A. (1988). The early years of logic programming. *Commun. ACM*, *31*(1), 38–43.

Kowalski, R. A., & Kuehner, D. (1971). Linear resolution with selection function. *Artif. Intell.*, *2*(3/4), 227–260.

Kramer, S. (1995). Predicate invention: A comprehensive view. *Rapport technique OFAI-TR-95-32, Austrian Research Institute for Artificial Intelligence, Vienna*.

Lake, B. M., Salakhutdinov, R., & Tenenbaum, J. B. (2015). Human-level concept learning through probabilistic program induction. *Science*, *350*(6266), 1332–1338.

Lake, B. M., Ullman, T. D., Tenenbaum, J. B., & Gershman, S. J. (2016). Building machines that learn and think like people. *CoRR*, *abs/1604.00289*.

Law, M. (2018). *Inductive learning of answer set programs*. Ph.D. thesis, Imperial College London, UK.

Law, M., Russo, A., Bertino, E., Broda, K., & Lobo, J. (2019). Representing and learning grammars in answer set programming. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019*, pp. 2919–2928. AAAI Press.

Law, M., Russo, A., Bertino, E., Broda, K., & Lobo, J. (2020). Fastlas: Scalable inductive logic programming incorporating domain-specific optimisation criteria. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pp. 2877–2885. AAAI Press.

Law, M., Russo, A., & Broda, K. (2014). Inductive learning of answer set programs. In Fermé, E., & Leite, J. (Eds.), *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, Vol. 8761 of *Lecture Notes in Computer Science*, pp. 311–325. Springer.

```
{alice, bob, claire, dave, age(alice), age(bob), age(age(alice)), . . . }
```

The Herbrand base is the set of all ground atoms that can be formed from the predicate symbols in $\mathcal{V}$ and the terms in the corresponding Herbrand universe. For instance, the Herbrand base of the lego builder example is:

$$\left\{\begin{array}{l}\texttt{happy(alice), happy(bob), happy(claire), happy(dave),}\\ \texttt{lego\_builder(alice), lego\_builder(bob), lego\_builder(claire), lego\_builder(dave),}\\ \texttt{estate\_agent(alice), estate\_agent(bob), estate\_agent(claire), estate\_agent(dave),}\\ \texttt{enjoys\_lego(alice), enjoys\_lego(bob), enjoys\_lego(claire), enjoys\_lego(dave)}\end{array}\right\}$$

A Herbrand interpretation assigns truth values to the elements of a Herbrand base. By convention, a Herbrand interpretation includes true ground atoms, assuming that every atom not included is false. For instance, the Herbrand interpretation corresponding to the example in Section 1.1 is:

$$\left\{\begin{array}{l}\texttt{happy(alice), lego\_builder(alice), lego\_builder(bob), estate\_agent(claire),}\\ \texttt{estate\_agent(dave), enjoys\_lego(alice), enjoys\_lego(claire)}\end{array}\right\}$$

A Herbrand interpretation I is a Herbrand model for a set of clauses C if for all clauses $\mathrm{h}_1 ; \ldots ; \mathrm{h}_n$:- $\mathrm{b}_1 , . . , \mathrm{b}_m \in C$ and for all ground substitutions θ: $\{\mathrm{b}_1\theta , \ldots , \mathrm{b}_m\theta\} \subset I \rightarrow \{\mathrm{h}_1\theta , \ldots , \mathrm{h}_n\theta\} \cap I \neq \emptyset$. That is, a Herbrand interpretation I is a Herbrand model of a clause if for all substitutions θ for which the body literals, after applying the substitution θ, are true in I, at least one of the head literals is also true in I. For instance, the Herbrand interpretation from the previous paragraph is a model for the clause:

```
happy(A):- lego_builder(A), enjoys_lego(A).
```

because every substitution that makes the body (θ={A/alice}) true also makes the head true. By contrast, the following interpretation is not a model of the clause because the substitution θ={A/dave} makes the body true but not the head:

```
enjoys_lego(A):- estate_agent(A).
```

A definite clause c is a logical consequence of a theory T if every Herbrand model of T is also a model of c.

This brings us to one of the core concepts in ILP *entailment*. When a clause c is a logical consequence of a theory T, we say that c is *entailed* by T, written T $\models$ c. The concept of entailment will reappear throughout this text, most prominently when we discuss various learning settings in ILP (Section 3). It is therefore important to have a firm grasp of its meaning.

2.3 Logic Programming Languages

There are various logic programming languages. We now cover some of the most important ones for ILP. Logic programming is based on clausal logic. Clausal programs are sets of clauses. Robinson (1965) showed that a single rule of inference (the resolution principle) is both sound and refutation complete for clausal logic. However, reasoning about full clausal logic is computationally expensive (Nienhuys-Cheng & Wolf, 1997). Therefore, most work in ILP focuses on fragments of clausal logic, such as Horn programs: clauses with at most one positive literal. All programs mentioned in the introduction are Horn programs, such as the program for extracting the last element of the list:

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

One reason for focusing on Horn theories, rather than full clausal theories, is SLD-resolution (Kowalski & Kuehner, 1971), an inference rule that sacrifices expressibility for efficiency. For instance, the clause $p(a) \vee p(b)$ cannot be expressed in Horn logic because it has two positive literals. Horn logic is, however, still Turing complete (Tärnlund, 1977).

Prolog (Kowalski, 1988; Colmerauer & Roussel, 1993) is a logic programming language based on SLD-resolution and is therefore restricted to Horn clauses. Most Prolog implementations (Wielemaker et al., 2012; Costa et al., 2012) allow extra-logical features, such as cuts. Prolog is not purely declarative because of constructs like cut, which means that a procedural reading of a Prolog program is needed to understand it. In other words, the order of clauses in a Prolog program has a major influence on its execution and results.

Datalog is a fragment of definite clausal theories (clausal theories that contain only definite clauses). The main two restrictions are (i) every variable in the head literal must also appear in a body literal, and (ii) complex terms as arguments of predicates are disallowed, e.g. `p(f(1),2)` or lists. Therefore, the list manipulation programs from previous sections cannot (easily) be expressed in Datalog[9]. Datalog is, however, sufficient for the *happy* Scenario because structured terms are unnecessary. Compared to definite programs, the main advantage of Datalog is decidability (Dantsin et al., 2001). However, this decidability comes at the cost of expressivity as Datalog is not Turing complete. By contrast, definite programs with function symbols have the expressive power of Turing machines and are consequently undecidable (Tärnlund, 1977). Unlike Prolog, Datalog is purely declarative.

2.3.1 Non-monotonic Logic

A logic is monotonic when adding knowledge to it does not reduce the logical consequences of that theory. A logic is non-monotonic if some conclusions can be removed/invalidated by adding more knowledge. Definite programs are monotonic because anything that could be deduced before a (definite) clause is added to it can still be deduced after it is added. In other words, adding a (definite) clause to a definite program cannot remove the logical consequences of the program. For instance, consider the following propositional program:

```
sunny.
happy:- sunny.
```

This program states it is sunny and that I am happy if it is sunny. We can therefore deduce that I am happy because it is sunny. Now suppose that we added another rule:

```
sunny.
happy:- sunny.
happy:- rich.
```

This new rule states that I am also happy if I am rich. Note that by the closed world assumption, we know I am not rich. After adding this rule, we can still deduce that I am happy from the first rule.

The logic of definite clauses with negation as failure (NAF) (Clark, 1977) is non-monotonic, which brings us to the special class of *normal logic programs*, which take the following form:

$$\texttt{h :- } \texttt{b}_1\texttt{,...,}\texttt{b}_n\texttt{, not } \texttt{b}_{n+1}\texttt{,..., not } \texttt{b}_m\texttt{.}$$

[9] It is possible to represent lists as a set of facts.

Goodacre, J. (1996). *Inductive learning of chess rules using Progol*. Ph.D. thesis, University of Oxford.

Gottlob, G., Leone, N., & Scarcello, F. (1997). On the complexity of some inductive logic programming problems. In Lavrac, N., & Dzeroski, S. (Eds.), *Inductive Logic Programming, 7th International Workshop, ILP-97, Prague, Czech Republic, September 17-20, 1997, Proceedings*, Vol. 1297 of *Lecture Notes in Computer Science*, pp. 17–32. Springer.

Grobelnik, M. (1992). Markus: an optimized model inference system. In *Proceedings of the Logic Approaches to Machine Learning Workshop*.

Gulwani, S. (2011). Automating string processing in spreadsheets using input-output examples. In Ball, T., & Sagiv, M. (Eds.), *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pp. 317–330. ACM.

Gulwani, S., Hernández-Orallo, J., Kitzelmann, E., Muggleton, S. H., Schmid, U., & Zorn, B. G. (2015). Inductive programming meets the real world. *Commun. ACM*, *58*(11), 90–99.

Gulwani, S., Polozov, O., Singh, R., et al. (2017). Program synthesis. *Foundations and Trends® in Programming Languages*, *4*(1-2), 1–119.

Harrison, J. (2009). *Handbook of Practical Logic and Automated Reasoning* (1st edition). Cambridge University Press, USA.

Hoare, C. A. R. (1961). Algorithm 64: Quicksort. *Commun. ACM*, *4*(7), 321.

Hocquette, C., & Muggleton, S. H. (2020). Complete bottom-up predicate invention in meta-interpretive learning. In Bessiere, C. (Ed.), *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pp. 2312–2318. ijcai.org.

Inoue, K. (2004). Induction as consequence finding. *Machine Learning*, *55*(2), 109–135.

Inoue, K. (2016). Meta-level abduction. *FLAP*, *3*(1), 7–36.

Inoue, K., Doncescu, A., & Nabeshima, H. (2013). Completing causal networks by meta-level abduction. *Machine Learning*, *91*(2), 239–277.

Inoue, K., & Kudoh, Y. (1997). Learning extended logic programs. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pp. 176–181. Morgan Kaufmann.

Inoue, K., Ribeiro, T., & Sakama, C. (2014). Learning from interpretation transition. *Machine Learning*, *94*(1), 51–79.

Kaalia, R., Srinivasan, A., Kumar, A., & Ghosh, I. (2016). Ilp-assisted de novo drug design. *Machine Learning*, *103*(3), 309–341.

Kaiser, L., & Sutskever, I. (2016). Neural gpus learn algorithms. In Bengio, Y., & LeCun, Y. (Eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

Kaminski, T., Eiter, T., & Inoue, K. (2018). Exploiting answer set programming with external sources for meta-interpretive learning. *Theory Pract. Log. Program.*, *18*(3-4), 571–588.

Katzouris, N., Artikis, A., & Paliouras, G. (2015). Incremental learning of event definitions with inductive logic programming. *Machine Learning*, *100*(2-3), 555–585.

Katzouris, N., Artikis, A., & Paliouras, G. (2016). Online learning of event definitions. *Theory Pract. Log. Program.*, *16*(5-6), 817–833.

on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada, pp. 9165–9174.

Emde, W., Habel, C., & Rollinger, C. (1983). The discovery of the equator or concept driven learning. In Bundy, A. (Ed.), *Proceedings of the 8th International Joint Conference on Artificial Intelligence. Karlsruhe, FRG, August 1983*, pp. 455–458. William Kaufmann.

Evans, R. (2020). *Kant's cognitive architecture*. Ph.D. thesis, Imperial College London, UK.

Evans, R., & Grefenstette, E. (2018). Learning explanatory rules from noisy data. *J. Artif. Intell. Res.*, *61*, 1–64.

Evans, R., Hernández-Orallo, J., Welbl, J., Kohli, P., & Sergot, M. J. (2021). Making sense of sensory input. *Artif. Intell.*, *293*, 103438.

Ferilli, S., Esposito, F., Basile, T. M. A., & Mauro, N. D. (2004). Automatic induction of first-order logic descriptors type domains from observations. In Camacho, R., King, R. D., & Srinivasan, A. (Eds.), *Inductive Logic Programming, 14th International Conference, ILP 2004, Porto, Portugal, September 6-8, 2004, Proceedings*, Vol. 3194 of *Lecture Notes in Computer Science*, pp. 116–131. Springer.

Feser, J. K., Chaudhuri, S., & Dillig, I. (2015). Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pp. 229–239.

Finn, P. W., Muggleton, S., Page, D., & Srinivasan, A. (1998). Pharmacophore discovery using the inductive logic programming system PROGOL. *Machine Learning*, *30*(2-3), 241–270.

Flach, P. A. (1993). Predicate invention in inductive data engineering. In Brazdil, P. (Ed.), *Machine Learning: ECML-93, European Conference on Machine Learning, Vienna, Austria, April 5-7, 1993, Proceedings*, Vol. 667 of *Lecture Notes in Computer Science*, pp. 83–94. Springer.

Flach, P. A., & Hadjiantonis, A. (2013). *Abduction and Induction: Essays on their relation and integration*, Vol. 18. Springer Science & Business Media.

Flener, P. (1996). Inductive logic program synthesis with DIALOGS. In Muggleton, S. (Ed.), *Inductive Logic Programming, 6th International Workshop, ILP-96, Stockholm, Sweden, August 26-28, 1996, Selected Papers*, Vol. 1314 of *Lecture Notes in Computer Science*, pp. 175–198. Springer.

Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2012). *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2014). Clingo = ASP + control: Preliminary report. *CoRR*, *abs/1405.3694*.

Gebser, M., Kaufmann, B., & Schaub, T. (2012). Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, *187*, 52–89.

Gelder, A. V., Ross, K. A., & Schlipf, J. S. (1991). The well-founded semantics for general logic programs. *J. ACM*, *38*(3), 620–650.

Gelfond, M., & Lifschitz, V. (1988). The stable model semantics for logic programming. In Kowalski, R. A., & Bowen, K. A. (Eds.), *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*, pp. 1070–1080. MIT Press.

Genesereth, M. R., & Björnsson, Y. (2013). The international general game playing competition. *AI Magazine*, *34*(2), 107–111.

Informally, a normal rule states that the head is true if all $b_1, \ldots, b_n$ are true and all $b_{n+1}, \ldots, b_m$ are false. That is, an atom is false if it cannot be proven true. That an atom cannot be proven true does not mean that it is missing from a knowledge base. Instead, it additionally means that there no rule can prove it true. Assuming that a logical statement is false if it cannot be proven true is known as following the *closed world assumption*.

Now consider the following non-monotonic program:

```
sunny.
happy:- sunny, not weekday.
```

This program states it is sunny and I am happy if it is sunny and it is not a weekday. By the closed world assumption, we can deduce that it is not a weekday, so we can deduce that I am happy because it is sunny and it is not a weekday. Now suppose we added knowledge that it is a weekday.

```
sunny.
happy:- sunny, not weekday.
weekday.
```

Then we can no longer deduce that I am happy. In other words, by adding knowledge that it is a weekday, the conclusion that I am happy no longer holds.

There are many different semantics ascribed to non-monotonic programs, including completion (Clark, 1977), well-founded (Gelder et al., 1991), and stable model (answer set) (Gelfond & Lifschitz, 1988) semantics. Discussing the differences of these semantics is beyond the scope of this paper.

Answer set programming. Answer set programming is a form of logic programming based on stable model (answer set) semantics (Gelfond & Lifschitz, 1988). Whereas a definite logic program has only one model (the least Herbrand model), an ASP program can have one, many, or even no models (answer sets). This makes ASP particularly attractive for expressing common-sense reasoning (Law et al., 2018). Similar to Datalog, an answer set program is purely declarative. ASP also supports additional language features, such as aggregates and weak and hard constraints. Computation in ASP is the process of finding models. Answer set solvers perform the search and thus generate models. Most ASP solvers (Gebser et al., 2012), in principle, always terminate (unlike Prolog query evaluation, which may lead to an infinite loop). We refer the reader to the excellent book by Gebser et al. (2012) for more information.

2.4 Generality

A key concept in ILP is the generality order over hypotheses (logic programs). A generality order helps structure the search over the hypothesis space by reasoning about the relative properties of two programs. A generality ordering tells us whether a program p_1 is more general than a program p_2, i.e., whether all logical consequence of p_2 are also logical consequences of p_1. Equivalently, if p_1 is *more general* than p_2, then p_2 is *more specific* than p_1.

Most approaches reason about the generality of programs syntactically through θ-subsumption (or *subsumption* for short) (Plotkin, 1971). To understand subsumption, we need to understand that a clause can be seen as a finite (possibly empty) set of literals, implicitly representing their disjunction. For instance, the clause:

```
happy(A) :- lego_builder(A), enjoys_lego(A).
```

is equivalent to (where ¬ denotes classical negation):

$$\{\texttt{happy(A)}, \neg\texttt{lego\_builder(A)}, \neg\texttt{enjoys\_lego(A)}\}.$$

We representing clauses as sets, we can define subsumption:

Definition 1 (Clausal subsumption). A clause C_1 *subsumes* a clause C_2 if and only if there exists a substitution θ such that $C_1\theta \subseteq C_2$.

Example 1 (Clausal subsumption). Let $\texttt{C}_1$ and $\texttt{C}_2$ be the clauses:

$$\begin{aligned} \texttt{C}_1 &= \texttt{f(A,B):- head(A,B)} \\ \texttt{C}_2 &= \texttt{f(X,Y):- head(X,Y),odd(Y).} \end{aligned}$$

Then $\texttt{C}_1$ subsumes $\texttt{C}_2$ because

$$\big\{\texttt{f(A,B)},\ \neg\texttt{head(A,B)}\big\}\theta \subseteq \big\{\texttt{f(X,Y)},\ \neg\texttt{head(X,Y)}, \neg\texttt{odd(Y)}\big\}$$

with $\theta = \{A/X, Y/B\}$.

Conversely, a clause C_2 is *more specific* than a clause C_1 if C_1 subsumes C_2.[10]

In principle, we could check the generality of two programs by comparing the consequences they entail. However, a program might entail an infinite set of consequences (e.g. when structured terms, such as lists, are involved) which would prevent us from establishing the generality relation between two programs. In other words, checking entailment between clauses is undecidable (Church, 1936). By contrast, checking subsumption between clauses is decidable (Plotkin, 1971), although, in general, deciding subsumption is a NP-complete problem (Nienhuys-Cheng & Wolf, 1997).

3. Inductive Logic Programming

In the introduction, we described four ILP scenarios. In each case, the problem was formed of three sets B (background knowledge), E^+ (positive examples), and E^- (negative examples). We informally stated the ILP problem is to induce a hypothesis H that with B *generalises* E^+ and E^-. We now formalise this problem.

According to De Raedt (1997), there are three main ILP learning settings: learning from *entailment* (LFE), *interpretations* (LFI), and *satisfiability* (LFS). LFE and LFI are by far the most popular learning settings, so we only cover these two. Other recent work focuses on *learning from transitions* (Inoue et al., 2014; Evans et al., 2021; Ribeiro et al., 2020) and *learning from answer sets* (Law et al., 2014). We refer the reader to these other works for an overview of those learning settings.

In each setting, the symbol $\mathcal{X}$ denotes the *example/instance* space, the set of examples for which a concept is defined; $\mathcal{B}$ denotes the language of *background knowledge*, the set of all clauses that could be provided as background knowledge; and $\mathcal{H}$ denotes the *hypothesis space*, the set of all possible hypotheses.

[10] This notion of subsumption is known as *weak subsumption*. An alternative notion is *strong subsumptions* which additionally performs factoring, i.e., it remove redundant literals. As an example, p(X,Y) :- q(X,Y), q(Y,X). strongly subsumes p(Z,Z) :- q(Z,Z). with $\theta = \{X/Z, Y/Z\}$ because p(Z,Z) :- q(Z,Z), q(Z,Z). is equivalent to p(Z,Z) :- q(Z,Z).. It does not, however, weakly subsume it as the number of literals is different.

De Raedt, L., & Kersting, K. (2008a). Probabilistic inductive logic programming. In De Raedt, L., Frasconi, P., Kersting, K., & Muggleton, S. (Eds.), *Probabilistic Inductive Logic Programming - Theory and Applications*, Vol. 4911 of *Lecture Notes in Computer Science*, pp. 1–27. Springer.

De Raedt, L., & Kersting, K. (2008b). *Probabilistic Inductive Logic Programming*, p. 1–27. Springer-Verlag, Berlin, Heidelberg.

De Raedt, L., Kersting, K., Kimmig, A., Revoredo, K., & Toivonen, H. (2008). Compressing probabilistic prolog programs. *Machine Learning*, *70*(2-3), 151–168.

De Raedt, L., Kersting, K., Natarajan, S., & Poole, D. (2016). *Statistical Relational Artificial Intelligence: Logic, Probability, and Computation*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

De Raedt, L., Kimmig, A., & Toivonen, H. (2007). Problog: A probabilistic prolog and its application in link discovery. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pp. 2462–2467.

Domingos, P. (2015). *The master algorithm: How the quest for the ultimate learning machine will remake our world*. Basic Books.

Domingos, P. M. (1999). The role of occam's razor in knowledge discovery. *Data Min. Knowl. Discov.*, *3*(4), 409–425.

Dong, H., Mao, J., Lin, T., Wang, C., Li, L., & Zhou, D. (2019). Neural logic machines. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Dumancic, S., Guns, T., & Cropper, A. (2021). Knowledge refactoring for inductive program synthesis. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pp. 7271–7278. AAAI Press.

Dumančić, S., & Blockeel, H. (2017). Clustering-based relational unsupervised representation learning with an explicit distributed representation. In Sierra, C. (Ed.), *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pp. 1631–1637. ijcai.org.

Dumančić, S., Guns, T., Meert, W., & Blockeel, H. (2019). Learning relational representations with auto-encoding logic programs. In Kraus, S. (Ed.), *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pp. 6081–6087. ijcai.org.

Dzeroski, S., Cussens, J., & Manandhar, S. (1999). An introduction to inductive logic programming and learning language in logic. In Cussens, J., & Dzeroski, S. (Eds.), *Learning Language in Logic*, Vol. 1925 of *Lecture Notes in Computer Science*, pp. 3–35. Springer.

Ellis, K., Morales, L., Sablé-Meyer, M., Solar-Lezama, A., & Tenenbaum, J. (2018). Learning libraries of subroutines for neurally-guided bayesian program induction. In *NeurIPS 2018*, pp. 7816–7826.

Ellis, K., Nye, M. I., Pu, Y., Sosa, F., Tenenbaum, J., & Solar-Lezama, A. (2019). Write, execute, assess: Program synthesis with a REPL. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., & Garnett, R. (Eds.), *Advances in Neural Information Processing Systems 32: Annual Conference*

Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015, pp. 3423–3429. AAAI Press.

Cropper, A., & Muggleton, S. H. (2016). Metagol system. https://github.com/metagol/metagol.

Cropper, A., & Muggleton, S. H. (2019). Learning efficient logic programs. *Machine Learning*, *108*(7), 1063–1083.

Cropper, A., Tamaddoni-Nezhad, A., & Muggleton, S. H. (2015). Meta-interpretive learning of data transformation programs. In Inoue, K., Ohwada, H., & Yamamoto, A. (Eds.), *Inductive Logic Programming - 25th International Conference, ILP 2015, Kyoto, Japan, August 20-22, 2015, Revised Selected Papers*, Vol. 9575 of *Lecture Notes in Computer Science*, pp. 46–59. Springer.

Cropper, A., & Tourret, S. (2020). Logical reduction of metarules. *Machine Learning*, *109*(7), 1323–1369.

Dai, W., & Muggleton, S. (2021). Abductive knowledge induction from raw data. In Zhou, Z. (Ed.), *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pp. 1845–1851. ijcai.org.

Dai, W., Muggleton, S., Wen, J., Tamaddoni-Nezhad, A., & Zhou, Z. (2017). Logical vision: One-shot meta-interpretive learning from real images. In Lachiche, N., & Vrain, C. (Eds.), *Inductive Logic Programming - 27th International Conference, ILP 2017, Orléans, France, September 4-6, 2017, Revised Selected Papers*, Vol. 10759 of *Lecture Notes in Computer Science*, pp. 46–62. Springer.

Dai, W., Xu, Q., Yu, Y., & Zhou, Z. (2019). Bridging machine learning and logical reasoning by abductive learning. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., & Garnett, R. (Eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 2811–2822.

Dantsin, E., Eiter, T., Gottlob, G., & Voronkov, A. (2001). Complexity and expressive power of logic programming. *ACM Comput. Surv.*, *33*(3), 374–425.

Dash, T., Srinivasan, A., Vig, L., Orhobor, O. I., & King, R. D. (2018). Large-scale assessment of deep relational machines. In Riguzzi, F., Bellodi, E., & Zese, R. (Eds.), *Inductive Logic Programming - 28th International Conference, ILP 2018, Ferrara, Italy, September 2-4, 2018, Proceedings*, Vol. 11105 of *Lecture Notes in Computer Science*, pp. 22–37. Springer.

d'Avila Garcez, A. S., Gori, M., Lamb, L. C., Serafini, L., Spranger, M., & Tran, S. N. (2019). Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning. *FLAP*, *6*(4), 611–632.

De Raedt, L. (1997). Logical settings for concept-learning. *Artif. Intell.*, *95*(1), 187–201.

De Raedt, L. (2008). *Logical and relational learning*. Cognitive Technologies. Springer.

De Raedt, L., & Bruynooghe, M. (1992). Interactive concept-learning and constructive induction by analogy. *Machine Learing*, *8*, 107–150.

De Raedt, L., & Dehaspe, L. (1997). Clausal discovery. *Machine Learning*, *26*(2-3), 99–146.

De Raedt, L., Dries, A., Thon, I., den Broeck, G. V., & Verbeke, M. (2015). Inducing probabilistic relational rules from probabilistic examples. In Yang, Q., & Wooldridge, M. J. (Eds.), *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pp. 1835–1843. AAAI Press.

3.1 Learning From Entailment

LFE is by far the most popular ILP setting (Shapiro, 1983; Muggleton, 1987; Muggleton & Buntine, 1988; Muggleton & Feng, 1990; Quinlan, 1990; Muggleton, 1995; Bratko, 1999; Srinivasan, 2001; Ray, 2009; Ahlgren & Yuen, 2013; Muggleton et al., 2015; Cropper & Muggleton, 2016; Kaminski et al., 2018; Cropper & Morel, 2021). The LFE problem is based on the notion of *entailment*, which we discussed in Section 2.2, and two properties of the hypotheses: completeness and consistency. A hypothesis is *complete* if it entails all positive examples. A hypothesis is *consistent* if it does not entail any negative example.

The LFE problem is:

Definition 2 (Learning from entailment). Given a tuple (B, E^{+}, E^{-}) where:

- $B \subseteq \mathcal{B}$ denotes background knowledge
- $E^{+} \subseteq \mathcal{X}$ denotes positive examples of the concept
- $E^{-} \subseteq \mathcal{X}$ denotes negative examples of the concept

The goal LFE is to return a hypothesis $H \in \mathcal{H}$ such that:

- $\forall e \in E^{+},\ H \cup B \models e$ (i.e. H is *complete*)
- $\forall e \in E^{-},\ H \cup B \not\models e$ (i.e. H is *consistent*)

A hypothesis can be a single clause or multiple clauses. Often, a single clause is insufficient to describe a target concept. For instance, to learn a definition of a recursive concept, the hypothesis needs to capture at least the base and recursive case. The setup in which the hypothesis needs to capture at least two dependent clauses is known as multi-clause learning (Muggleton et al., 2011).

Example 2. Consider the LFE tuple:

$$\mathrm{B} = \left\{ \begin{array}{l} \texttt{lego\_builder(alice).} \\ \texttt{lego\_builder(bob).} \\ \texttt{estate\_agent(claire).} \\ \texttt{estate\_agent(dave).} \\ \texttt{enjoys\_lego(alice).} \\ \texttt{enjoys\_lego(claire).} \end{array} \right\} \mathrm{E}^{+} = \left\{ \begin{array}{l} \texttt{happy(alice).} \end{array} \right\} \mathrm{E}^{-} = \left\{ \begin{array}{l} \texttt{happy(bob).} \\ \texttt{happy(claire).} \\ \texttt{happy(dave).} \end{array} \right\}$$

Also assume we have the hypothesis space:

$$\mathcal{H} = \left\{ \begin{array}{l} \mathrm{h}_1\texttt{: happy(A):- lego\_builder(A).} \\ \mathrm{h}_2\texttt{: happy(A):- estate\_agent(A).} \\ \mathrm{h}_3\texttt{: happy(A):- likes\_lego(A).} \\ \mathrm{h}_4\texttt{: happy(A):- lego\_builder(A),estate\_agent(A).} \\ \mathrm{h}_5\texttt{: happy(A):- lego\_builder(A),enjoys\_lego(A).} \\ \mathrm{h}_6\texttt{: happy(A):- estate\_agent(A),enjoys\_lego(A).} \end{array} \right\}$$

Then we can consider which hypotheses an ILP system should return:

- $B \cup \mathrm{h}_1 \models happy(bob)$ so is inconsistent
- $B \cup \mathrm{h}_2 \not\models happy(alice)$ so is incomplete

- $B \cup h_3 \models happy(claire)$ so is inconsistent
- $B \cup h_4 \not\models happy(alice)$ so is incomplete
- $B \cup h_5$ is both complete and consistent
- $B \cup h_6 \not\models happy(alice)$ so is incomplete

The LFE problem in Definition 2 is general. ILP systems impose strong restrictions on $\mathcal{X}$, $\mathcal{B}$, and $\mathcal{H}$. For instance, some restrict $\mathcal{X}$ to only contain atoms whereas others allow clauses. Some restrict $\mathcal{H}$ to contain only Datalog clauses. We discuss these biases in Section 4.

According to Definition 2, a hypothesis must entail every positive example (be *complete*) and no negative examples (be *consistent*). However, training examples are often noisy, so it is difficult to find a hypothesis that is both complete and consistent. Therefore, most approaches relax this definition and try to find a hypothesis that entails as many positive and as few negative examples as possible. Precisely what this means depends on the system. For instance, the default cost function in Aleph (Srinivasan, 2001) is *coverage*, defined as the number of positive examples entailed subtracted by the number of negative examples entailed by the hypothesis. Other systems also consider the size of a hypothesis, typically the number of clauses or literals in it. We discuss noise handling in Section 5.1.

3.2 Learning From Interpretations

The second most popular (De Raedt & Dehaspe, 1997; Blockeel & De Raedt, 1998; Law et al., 2014) learning setting is LFI where an example is an interpretation, i.e. a set of facts. The LFI problem is:

Definition 3 (Learning from interpretations). Given a tuple (B, E^+, E^-) where:

- $B \subseteq \mathcal{B}$ denotes background knowledge
- $E^+ \subseteq \mathcal{X}$ denotes positive examples of the concept, each example being a set of facts
- $E^- \subseteq \mathcal{X}$ denotes negative examples of the concept, each example being a set of facts

The goal of LFI is to return a hypothesis $H \in \mathcal{H}$ such that:

- $\forall e \in E^+,$ *e is a model of* $H \cup B$
- $\forall e \in E^-,$ *e is not a model of* $H \cup B$

When learning from interpretations, it is implicitly assumed that every example is completely specified. That is, every atom in the interpretation has to be true or false, and there is no room for missing values. As providing a complete interpretation might be unfeasible in many cases, many ILP systems focus on partial interpretations (De Raedt, 1997).

Example 3. To illustrate LFI, we use the example from (De Raedt & Kersting, 2008a). Consider the BK:

$$\mathrm{B} = \left\{ \begin{array}{lll} \texttt{father(henry,bill).} & \texttt{father(alan,betsy).} & \texttt{father(alan,benny).} \\ \texttt{mother(beth,bill).} & \texttt{mother(ann,betsy).} & \texttt{mother(alice,benny).} \end{array} \right\}$$

and the examples:

$$\mathrm{E}^+ = \left\{ \begin{array}{l} \mathrm{e}_1 = \left\{ \begin{array}{l} \texttt{carrier(alan).} \\ \texttt{carrier(ann).} \\ \texttt{carrier(betsy).} \end{array} \right\} \\ \mathrm{e}_2 = \left\{ \begin{array}{l} \texttt{carrier(benny).} \\ \texttt{carrier(alan).} \\ \texttt{carrier(alice).} \end{array} \right\} \end{array} \right\}$$

Corapi, D., Russo, A., & Lupu, E. (2010). Inductive logic programming as abductive search. In Hermenegildo, M. V., & Schaub, T. (Eds.), *Technical Communications of the 26th International Conference on Logic Programming, ICLP 2010, July 16-19, 2010, Edinburgh, Scotland, UK*, Vol. 7 of *LIPIcs*, pp. 54–63. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Corapi, D., Russo, A., & Lupu, E. (2011). Inductive logic programming in answer set programming. In Muggleton, S., Tamaddoni-Nezhad, A., & Lisi, F. A. (Eds.), *Inductive Logic Programming - 21st International Conference, ILP 2011, Windsor Great Park, UK, July 31 - August 3, 2011, Revised Selected Papers*, Vol. 7207 of *Lecture Notes in Computer Science*, pp. 91–97. Springer.

Costa, V. S., Rocha, R., & Damas, L. (2012). The YAP prolog system. *Theory Pract. Log. Program.*, *12*(1-2), 5–34.

Cropper, A. (2017). *Efficiently learning efficient programs*. Ph.D. thesis, Imperial College London, UK.

Cropper, A. (2019). Playgol: Learning programs through play. In Kraus, S. (Ed.), *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pp. 6074–6080. ijcai.org.

Cropper, A. (2020). Forgetting to learn logic programs. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, New York, NY, USA, February 7-12, 2020*, pp. 3676–3683. AAAI Press.

Cropper, A. (2022). Learning logic programs through divide, constrain, and conquer. In *Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI-22)*.

Cropper, A., & Dumančić, S. (2020). Learning large logic programs by going beyond entailment. In Bessiere, C. (Ed.), *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pp. 2073–2079. ijcai.org.

Cropper, A., Dumančić, S., & Muggleton, S. H. (2020a). Turning 30: New ideas in inductive logic programming. In Bessiere, C. (Ed.), *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pp. 4833–4839. ijcai.org.

Cropper, A., Evans, R., & Law, M. (2020b). Inductive general game playing. *Machine Learning*, *109*(7), 1393–1434.

Cropper, A., & Morel, R. (2021). Learning programs by learning from failures. *Machine Learning.*, *110*(4), 801–856.

Cropper, A., Morel, R., & Muggleton, S. (2020). Learning higher-order logic programs. *Machine Learning*, *109*(7), 1289–1322.

Cropper, A., & Muggleton, S. (2015). Can predicate invention compensate for incomplete background knowledge?. In Nowaczyk, S. (Ed.), *Thirteenth Scandinavian Conference on Artificial Intelligence - SCAI 2015, Halmstad, Sweden, November 5-6, 2015*, Vol. 278 of *Frontiers in Artificial Intelligence and Applications*, pp. 27–36. IOS Press.

Cropper, A., & Muggleton, S. H. (2014). Logical minimisation of meta-rules within meta-interpretive learning. In Davis, J., & Ramon, J. (Eds.), *Inductive Logic Programming - 24th International Conference, ILP 2014, Nancy, France, September 14-16, 2014, Revised Selected Papers*, Vol. 9046 of *Lecture Notes in Computer Science*, pp. 62–75. Springer.

Cropper, A., & Muggleton, S. H. (2015). Learning efficient logical robot strategies involving composable objects. In Yang, Q., & Wooldridge, M. J. (Eds.), *Proceedings of the Twenty-Fourth International*

Bohan, D. A., Vacher, C., Tamaddoni-Nezhad, A., Raybould, A., Dumbrell, A. J., & Woodward, G. (2017). Next-generation global biomonitoring: large-scale, automated reconstruction of ecological networks. *Trends in Ecology & Evolution*, *32*(7), 477–487.

Bratko, I. (1999). Refining complete hypotheses in ILP. In Dzeroski, S., & Flach, P. A. (Eds.), *Inductive Logic Programming, 9th International Workshop, ILP-99, Bled, Slovenia, June 24-27, 1999, Proceedings*, Vol. 1634 of *Lecture Notes in Computer Science*, pp. 44–55. Springer.

Bratko, I. (2012). *Prolog Programming for Artificial Intelligence, 4th Edition*. Addison-Wesley.

Bratko, I., Muggleton, S., & Varsek, A. (1991). Learning qualitative models of dynamic systems. In Birnbaum, L., & Collins, G. (Eds.), *Proceedings of the Eighth International Workshop (ML91), Northwestern University, Evanston, Illinois, USA*, pp. 385–388. Morgan Kaufmann.

Buntine, W. L. (1988). Generalized subsumption and its applications to induction and redundancy. *Artif. Intell.*, *36*(2), 149–176.

Calimeri, F., Cozza, S., Ianni, G., & Leone, N. (2008). Computable functions in ASP: theory and implementation. In de la Banda, M. G., & Pontelli, E. (Eds.), *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, Vol. 5366 of *Lecture Notes in Computer Science*, pp. 407–424. Springer.

Caruana, R. (1997). Multitask learning. *Machine Learning*, *28*(1), 41–75.

Castillo, L. P., & Wrobel, S. (2003). Learning minesweeper with multirelational learning. In Gottlob, G., & Walsh, T. (Eds.), *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pp. 533–540. Morgan Kaufmann.

Chollet, F. (2019). On the measure of intelligence. *CoRR*, *abs/1911.01547*.

Church, A. (1936). A note on the entscheidungsproblem. *J. Symb. Log.*, *1*(1), 40–41.

Clark, K. L. (1977). Negation as failure. In Gallaire, H., & Minker, J. (Eds.), *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977*, Advances in Data Base Theory, pp. 293–322, New York. Plemum Press.

Cohen, W. W. (1994a). Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artif. Intell.*, *68*(2), 303–366.

Cohen, W. W. (1994b). Recovering software specifications with inductive logic programming. In Hayes-Roth, B., & Korf, R. E. (Eds.), *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1*, pp. 142–148. AAAI Press / The MIT Press.

Cohen, W. W. (1995a). Inductive specification recovery: Understanding software by learning from example behaviors. *Autom. Softw. Eng.*, *2*(2), 107–129.

Cohen, W. W. (1995b). Pac-learning non-recursive prolog clauses. *Artif. Intell.*, *79*(1), 1–38.

Cohen, W. W. (1995c). Pac-learning recursive logic programs: Efficient algorithms. *J. Artif. Intell. Res.*, *2*, 501–539.

Cohen, W. W. (1995d). Pac-learning recursive logic programs: Negative results. *J. Artif. Intell. Res.*, *2*, 541–573.

Colmerauer, A., & Roussel, P. (1993). The birth of prolog. In Lee, J. A. N., & Sammet, J. E. (Eds.), *History of Programming Languages Conference (HOPL-II), Preprints, Cambridge, Massachusetts, USA, April 20-23, 1993*, pp. 37–52. ACM.

$$E^- = \left\{ e_3 = \left\{ \begin{array}{l} \texttt{carrier(henry).} \\ \texttt{carrier(beth).} \end{array} \right\} \right\}$$

Also assume the hypothesis space:

$$\mathcal{H} = \left\{ \begin{array}{l} h_1 = \texttt{carrier(X):- mother(Y,X),carrier(Y),father(Z,X),carrier(Z).} \\ h_2 = \texttt{carrier(X):- mother(Y,X),father(Z,X).} \end{array} \right\}$$

To solve the LFI problem (Definition 3), we need to find a hypothesis H such that e_1 and e_2 are models of $H \cup B$ and e_3 is not. That is, we need to find a hypothesis H that satisfies the following property for every example $e_i \in E^+$: for every substitution θ such that $body(h_1)\theta \subseteq B \cup e_i$ holds, it also holds that $head(h_1)\theta \subseteq B \cup e_i$. e_3 is not the model of h_1 as there exists a substitution $\theta = \{\texttt{X/bill}, \texttt{Y/beth}, \texttt{Z/henry}\}$ such that body holds but the head does not. For the same reason, none of the examples is a model of h_2.

We often say that a hypothesis *covers* an example. The meaning of a hypothesis covering an example changes depending on the learning setup. In LFE, the hypothesis H covers an example if the example is entailed by $H \cup B$. In LFI, the hypothesis H covers an example if the example is a model of $H \cup B$.

4. Building An ILP System

Building an ILP system requires making several choices or assumptions, which are part of the *inductive bias* of a learner. An inductive bias is essential and all ML approaches impose an inductive bias (Mitchell, 1997). Understanding these assumptions is key to understanding ILP. The choices can be categorised as:

- **Learning setting**: how to represent examples
- **Representation language**: how to represent BK and hypotheses
- **Language bias**: how to define the hypothesis space
- **Search method**: how to search the hypothesis space

Table 3 shows the assumptions of some systems. This table excludes many important systems, including interactive systems, such as Marvin (Sammut, 1981), MIS (Shapiro, 1983), DUCE (Muggleton, 1987), Cigol (Muggleton & Buntine, 1988), and Clint (De Raedt & Bruynooghe, 1992), theory revision systems, such as FORTE (Richards & Mooney, 1995), and probabilistic systems, such as SLIPCOVER (Bellodi & Riguzzi, 2015) and ProbFOIL (De Raedt et al., 2015). Covering all systems is beyond the scope of this paper. We discuss these differences/assumptions.

4.1 Learning Setting

The two main learning settings are LFE and LFI (Section 3). Within the LFE setting, there are further distinctions. Some systems, such as Progol (Muggleton, 1995), allow for clauses as examples. Most systems, however, learn from sets of facts, so this dimension of comparison is not useful.

[11] The original FOIL setting is more restricted than the table shows and can only have BK in the form of facts and does not allow functions (De Raedt, 2008).

[12] The FOIL paper does not discuss its language bias.

[13] LFIT employs many implicit language biases.

[14] The LFIT approach of Inoue et al. (2014) is bottom-up and the approach of Ribeiro and Inoue (2014) it top-down.

[15] ∂ILP uses rule templates which can be seen as a generalisation of metarules.

System	Setting	Hypotheses	BK	Language Bias	Search method
FOIL (Quinlan, 1990)	LFE	Definite	Definite[11]	n/a[12]	TD
Progol (Muggleton, 1995)	LFE	Normal	Normal	Modes	BU+TD
Claudien (De Raedt & Dehaspe, 1997)	LFI	Clausal	Definite	DLab	TD
TILDE (Blockeel & De Raedt, 1998)	LFI	Logical trees	Normal	Modes	TD
Aleph (Srinivasan, 2001)	LFE	Normal	Normal	Modes	BU+TD
XHAIL (Ray, 2009)	LFE	Normal	Normal	Modes	BU
ASPAL (Corapi et al., 2011)	LFE	Normal	Normal	Modes	ML
Atom (Ahlgren & Yuen, 2013)	LFE	Normal	Normal	Modes	BU+TD
QuickFOIL (Zeng et al., 2014)	LFE	Definite	Facts	Schema	TD
LFIT (Inoue et al., 2014)	LFT	Normal	None	n/a[13]	BU+TD[14]
ILASP[a] (Law et al., 2014)	LFI[b]	ASP	ASP	Modes	ML
Metagol (Muggleton et al., 2015)	LFE	Definite	Normal	Metarules	ML
∂ILP (Evans & Grefenstette, 2018)	LFE	Datalog	Facts	Metarules[15]	ML
HEXMIL (Kaminski et al., 2018)	LFE	Datalog	Datalog	Metarules	ML
FastLAS (Law et al., 2020)	LFI	ASP	ASP	Modes	ML
Apperception (Evans et al., 2021)	LFT	Datalog$^{\ni}$	None	Types	ML
Popper (Cropper & Morel, 2021)	LFE	Definite	Normal	Declarations	ML

Table 3: Assumptions of popular ILP systems. LFE stands for *learn from entailment*, LFI stands for *learning from interpretations*, LFT stands for *learning from transitions*. TD stands for *top-down*, BU stands for *bottom-up*, and ML stands for *meta-level*. This table is meant to provide a very high-level overview of the systems. Therefore, the table entries are coarse and should not be taken absolutely literally. For instance, Progol, Aleph, and ILASP support other types of language biases, such as constraints on clauses. Popper also, for instance, supports ASP programs as BK, but *usually* takes normal programs.

[a] ILASP is a suite of ILP systems. For simplicity, we refer to all the systems as ILASP. However, some features, such as noise handling, are not in the original ILASP paper.

[b] ILASP learns from answer sets. However, to avoid having to introduce a problem setting for a single system, we have classified it as learning from interpretations, which is the most similar setting. See the original ILASP paper for more information (Law et al., 2014).

4.2 Hypotheses

Although some systems learn propositional programs, such as Duce (Muggleton, 1987), most learn first-order (or higher-order) programs. For systems that learn first-order programs, there are classes of programs that they learn. Some systems induce full (unrestricted) clausal theories, such as Claudien (De Raedt & Dehaspe, 1997) and CF-induction (Inoue, 2004). However, reasoning about full clausal theories is computationally expensive, so most systems learn fragments of clausal logic, usually definite programs. Systems that focus on program synthesis (Shapiro, 1983; Bratko, 1999; Ahlgren & Yuen, 2013;

Conference, ILP 2015, Kyoto, Japan, August 20-22, 2015, Revised Selected Papers, Vol. 9575 of *Lecture Notes in Computer Science*, pp. 1–14. Springer.

Athakravi, D., Corapi, D., Broda, K., & Russo, A. (2013). Learning through hypothesis refinement using answer set programming. In Zaverucha, G., Costa, V. S., & Paes, A. (Eds.), *Inductive Logic Programming - 23rd International Conference, ILP 2013, Rio de Janeiro, Brazil, August 28-30, 2013, Revised Selected Papers*, Vol. 8812 of *Lecture Notes in Computer Science*, pp. 31–46. Springer.

Athanasopoulos, G., Paliouras, G., Vogiatzis, D., Tzortzis, G., & Katzouris, N. (2018). Predicting the evolution of communities with online inductive logic programming. In Alechina, N., Nørvåg, K., & Penczek, W. (Eds.), *25th International Symposium on Temporal Representation and Reasoning, TIME 2018, Warsaw, Poland, October 15-17, 2018*, Vol. 120 of *LIPIcs*, pp. 4:1–4:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Bain, M., & Muggleton, S. (1991). Non-monotonic learning. In Michie, D. (Ed.), *Machine Intelligence 12*, pp. 105–120. Oxford University Press.

Bain, M. (1994). *Learning logical exceptions in chess*. Ph.D. thesis, University of Strathclyde.

Bain, M., & Srinivasan, A. (2018). Identification of biological transition systems using meta-interpreted logic programs. *Machine Learning*, *107*(7), 1171–1206.

Balduccini, M., Lierler, Y., & Schüller, P. (2013). Prolog and ASP inference under one roof. In Cabalar, P., & Son, T. C. (Eds.), *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings*, Vol. 8148 of *Lecture Notes in Computer Science*, pp. 148–160. Springer.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., & Tarlow, D. (2017). Deepcoder: Learning to write programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Banerji, R. B. (1964). A language for the description of concepts. *General Systems*, *9*(1), 135–141.

Bartha, S., & Cheney, J. (2019). Towards meta-interpretive learning of programming language semantics. In Kazakov, D., & Erten, C. (Eds.), *Inductive Logic Programming - 29th International Conference, ILP 2019, Plovdiv, Bulgaria, September 3-5, 2019, Proceedings*, Vol. 11770 of *Lecture Notes in Computer Science*, pp. 16–25. Springer.

Bellodi, E., & Riguzzi, F. (2015). Structure learning of probabilistic logic programs by searching the clause space. *Theory Pract. Log. Program.*, *15*(2), 169–212.

Bengio, Y., Courville, A. C., & Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.*, *35*(8), 1798–1828.

Bengio, Y., Deleu, T., Rahaman, N., Ke, N. R., Lachapelle, S., Bilaniuk, O., Goyal, A., & Pal, C. J. (2019). A meta-transfer objective for learning to disentangle causal mechanisms. *CoRR*, *abs/1901.10912*.

Blockeel, H., & De Raedt, L. (1998). Top-down induction of first-order logical decision trees. *Artif. Intell.*, *101*(1-2), 285–297.

Blumer, A., Ehrenfeucht, A., Haussler, D., & Warmuth, M. K. (1987). Occam's razor. *Inf. Process. Lett.*, *24*(6), 377–380.

Bohan, D. A., Caron-Lormier, G., Muggleton, S., Raybould, A., & Tamaddoni-Nezhad, A. (2011). Automated discovery of food webs from ecological data using logic-based machine learning. *PLoS One*, *6*(12), e29028.

by first training a neural network to recognise the digits. Ideally, we would not want to treat the two problems separately, but rather simultaneously learn how to recognise the digits and learn a program to perform the addition. A handful of approaches have started to tackle this problem (Manhaeve et al., 2018; Dai et al., 2019; Evans et al., 2021; Dai & Muggleton, 2021), but developing better ILP techniques that can both perceive sensory input and learn complex relational programs would be a major breakthrough not only for ILP, but the whole of AI.

Further Reading

For an introduction to the fundamentals of logic and automated reasoning, we recommend the book of Harrison (2009). To read more about ILP, then we suggest starting with the founding paper by Muggleton (1991) and a survey paper that soon followed (Muggleton & De Raedt, 1994). For a detailed exposition of the theory of ILP, we thoroughly recommend the books of Nienhuys-Cheng and Wolf (1997) and De Raedt (2008).

Acknowledgements

We thank Céline Hocquette, Jonas Schouterden, Jonas Soenen, Tom Silver, Tony Ribeiro, and Oghenejokpeme Orhobor for helpful comments and suggestions. We also sincerely thank the anonymous reviewers, especially Reviewer 2, whose feedback and suggestions greatly improved the paper. Andrew Cropper was supported by the EPSRC fellowship *The Automatic Computer Scientist* (EP/V040340/1). Sebastijan Dumančić was partially supported by Research Foundation - Flanders (FWO; 12ZE520N).

References

Adé, H., De Raedt, L., & Bruynooghe, M. (1995). Declarative bias for specific-to-general ILP systems. *Machine Learning*, *20*(1-2), 119–154.

Adé, H., Malfait, B., & De Raedt, L. (1994). RUTH: an ILP theory revision system. In Ras, Z. W., & Zemankova, M. (Eds.), *Methodologies for Intelligent Systems, 8th International Symposium, ISMIS '94, Charlotte, North Carolina, USA, October 16-19, 1994, Proceedings*, Vol. 869 of *Lecture Notes in Computer Science*, pp. 336–345. Springer.

Ahlgren, J., & Yuen, S. Y. (2013). Efficient program synthesis using constraint satisfaction in inductive logic programming. *J. Machine Learning Res.*, *14*(1), 3649–3682.

Ai, L., Muggleton, S. H., Hocquette, C., Gromowski, M., & Schmid, U. (2021). Beneficial and harmful explanatory machine learning. *Machine Learning.*, *110*(4), 695–721.

Albarghouthi, A., Koutris, P., Naik, M., & Smith, C. (2017). Constraint-based synthesis of datalog programs. In Beck, J. C. (Ed.), *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, Vol. 10416 of *Lecture Notes in Computer Science*, pp. 689–706. Springer.

Alrajeh, D., Kramer, J., Russo, A., & Uchitel, S. (2013). Elaborating requirements using model checking and inductive learning. *IEEE Trans. Software Eng.*, *39*(3), 361–383.

Antanas, L., Moreno, P., & De Raedt, L. (2015). Relational kernel-based grasping with numerical features. In Inoue, K., Ohwada, H., & Yamamoto, A. (Eds.), *Inductive Logic Programming - 25th International*

Cropper & Muggleton, 2016, 2019; Cropper & Morel, 2021) tend to induce definite programs, typically as Prolog programs.

4.2.1 Normal Programs

One motivation for learning normal programs (Section 2.3.1) is that many practical applications require non-monotonic reasoning. Moreover, it is often simpler to express a concept with negation as failure (NAF). For instance, consider the following problem by Ray (2009):

$$B = \left\{ \begin{array}{l} \texttt{bird(A):- penguin(A)} \\ \texttt{bird(alvin)} \\ \texttt{bird(betty)} \\ \texttt{bird(charlie)} \\ \texttt{penguin(doris)} \end{array} \right\} E^{+} = \left\{ \begin{array}{l} \texttt{flies(alvin)} \\ \texttt{flies(betty)} \\ \texttt{flies(charlie)} \end{array} \right\} E^{-} = \left\{ \begin{array}{l} \texttt{flies(doris)} \end{array} \right\}$$

Without NAF it is difficult to induce a general hypothesis for this problem. By contrast, with NAF a system could learn the hypothesis:

$$H = \left\{ \texttt{flies(A):- bird(A), not penguin(A)} \right\}$$

ILP approaches that learn normal logic programs can further be characterised by their semantics, such as whether they are based on completion (Clark, 1977), well-founded (Gelder et al., 1991), or stable model (answer set) (Gelfond & Lifschitz, 1988) semantics. Discussing the differences between these semantics is beyond the scope of this paper.

4.2.2 Answer Set Programs

There are many benefits to learning ASP programs (Otero, 2001; Law et al., 2014). When learning Prolog programs with NAF, the programs must be stratified; otherwise, the learned program may loop under certain queries (Law et al., 2018). By contrast, some systems can learn unstratified ASP programs (Law et al., 2014). In addition, ASP programs support rules that are not available in Prolog, such as choice rules and weak and hard constraints. For instance, ILASP (Law et al., 2014), can learn the following definition of a Hamiltonian graph (taken from Law et al. (2020)) as an ASP program:

```
0{in(V0, V1)}1 :- edge(V0, V1).
reach(V0):- in(1, V0).
reach(V1):- reach(V0), in(V0, V1).
:- not reach(V0), node(V0).
:- V1 != V2, in(V0, V2), in(V0, V1).
```

This program illustrates useful language features of ASP. The first rule is a *choice* rule, which means that an atom *can* be true. In this example, the rule indicates that there can be an in edge from the vertex *V1* to *V0*. The last two rules are *hard constraints*, which essentially enforce integrity constraints. The first hard constraint states that it is impossible to have a node that is not reachable. The second hard constraint states that it is impossible to have a vertex with two in edges from distinct nodes. For more information about ASP we recommend the book by Gebser et al. (2012).

Approaches to learning ASP programs can be divided into two categories: *brave learners*, which aim to learn a program such that at least one answer set covers the examples, and *cautious learners*, which aim to find a program which covers the examples in all answer sets. We refer to existing work of Otero (2001), Sakama and Inoue (2009, 2009), Law et al. (2018) for more information about these different approaches.

4.2.3 Higher-order Programs

As many programmers know, there are benefits to using higher-order representations. For instance, suppose you have some encrypted/decrypted strings represented as Prolog facts:

$$E^+ = \left\{ \begin{array}{l} \texttt{decrypt([d,b,u],[c,a,t])} \\ \texttt{decrypt([e,p,h],[d,o,g])} \\ \texttt{decrypt([h,p,p,t,f],[g,o,o,s,e])} \end{array} \right\}$$

Given these examples and suitable BK, a system could learn the first-order program:

$$H = \left\{ \begin{array}{ll} \texttt{decrypt(A,B):-} & \texttt{empty(A),empty(B).} \\ \texttt{decrypt(A,B):-} & \texttt{head(A,C),chartoint(C,D),prec(D,E),inttochar(E,F),} \\ & \texttt{head(B,F),tail(A,G),tail(B,H),decrypt(G,H).} \end{array} \right\}$$

This program defines a Caesar cypher which shifts each character back once (e.g. $z \mapsto y$, $y \mapsto x$, etc). Although correct (ignoring the modulo operation for simplicity), this program is long and difficult to read. To overcome this limitation, some systems (Cropper et al., 2020) learn higher-order programs, such as:

$$H = \left\{ \begin{array}{l} \texttt{decrypt(A,B):- map(A,B,inv)} \\ \texttt{inv(A,B):- char\_to\_int(A,C),prec(C,D),int\_to\_char(D,B)} \end{array} \right\}$$

This program is higher-order because it allows literals to take predicate symbols as arguments. The symbol `inv` is *invented* (we discuss *predicate invention* in Section 5.5) and is used as an argument for `map` in the first rule and as a predicate symbol in the second rule. The higher-order program is smaller than the first-order program because the higher-order background relation `map` abstracts away the need to learn a recursive program. Cropper et al. (2020) show that inducing higher-order programs can drastically improve learning performance in terms of predictive accuracy, sample complexity, and learning times.

4.3 Background Knowledge

BK is similar to features used in other forms of ML. However, whereas features are finite tables, BK is a logic program. Using logic programs to represent data allows ILP to learn with complex relational information. For instance, suppose we want to learn list or string transformation programs, we might want to supply helper relations, such as head, `tail`, and `last` as BK:

$$B = \left\{ \begin{array}{l} \texttt{head([H|\_],H).} \\ \texttt{tail([\_|T],T).} \\ \texttt{last([H],H).} \\ \texttt{last([\_|T1],A):- tail(T1,T2),last(T2,A).} \end{array} \right\}$$

learned programs are frequently reused. However, these approaches have only been demonstrated on a small number of tasks. To reach human-level AI, we would expect a learner to learn thousands or even millions of concepts. But handling the complexity of thousands of tasks is challenging because, as we explained in Section 4.3, ILP systems struggle to handle large amounts of BK. This situation leads to the problem of *catastrophic remembering* (Cropper, 2020): the inability for a learner to forget knowledge. Although there is initial work on this topic (Cropper, 2020), we think that a key area for future work is handling the complexity of lifelong learning.

Relevance. The *catastrophic remembering* problem is essentially the problem of *relevance*: given a new ILP problem with lots of BK, how does an ILP system decide which BK is relevant? Although too much irrelevant BK is detrimental to learning performance (Srinivasan et al., 1995, 2003), there is almost no work in ILP on trying to identify relevant BK. One emerging technique is to train a neural network to score how relevant programs are in the BK and to then only use BK with the highest score to learn programs (Balog et al., 2017; Ellis et al., 2018). However, the empirical efficacy of this approach has yet to be clearly demonstrated. Moreover, these approaches have only been demonstrated on small amounts of BK and it is unclear how they scale to BK with thousands of relations. Without efficient relevancy methods, it is unclear how lifelong learning can be achieved.

Noisy BK. Another issue related to lifelong learning is the underlying uncertainty associated with adding learned programs to the BK. By the inherent nature of induction, induced programs are not guaranteed to be correct (i.e. are expected to be noisy), yet they are the building blocks for subsequent induction. Building noisy programs on top of other noisy programs could lead to eventual incoherence of the learned program. This issue is especially problematic because, as mentioned in Section 5.1, most ILP approaches assume noiseless BK, i.e. a relation is true or false without any room for uncertainty. One of the appealing features of ∂ILP is that it takes a differentiable approach to ILP, where it can be provided with fuzzy or ambiguous data. Developing similar techniques to handle noisy BK is an under-explored topic in ILP.

Probabilistic ILP. A principled way to handle noise is to unify logical and probabilistic reasoning, which is the focus of *statistical relational artificial intelligence* (StarAI) (De Raedt et al., 2016). While StarAI is a growing field, inducing probabilistic logic programs has received little attention, with few notable exceptions (Bellodi & Riguzzi, 2015; De Raedt et al., 2015), as inference remains the main challenge. Addressing this issue, i.e. unifying probability and logic in an inductive setting, would be a major achievement (Marcus, 2018).

Explainability. Explainability is one of the claimed advantages of a symbolic representation. Recent work (Muggleton et al., 2018; Ai et al., 2021) evaluates the comprehensibility of ILP hypotheses using Michie's (1988) framework of *ultra-strong ML*, where a learned hypothesis is expected to not only be accurate but to also demonstrably improve the performance of a human being provided with the learned hypothesis. Muggleton et al. (2018) empirically demonstrate improved human understanding directly through learned hypotheses. However, more work is required to better understand the conditions under which this can be achieved, especially given the rise of PI.

Learning from raw data. Most ILP systems require data in perfect symbolic form. However, much real-world data, such as images and speech, cannot easily be translated into a symbolic form. Perhaps the biggest challenge in ILP is to learn how to both perceive sensory input and learn a symbolic logic program to explain the input. For instance, consider a task of learning to perform addition from MNIST digits. Current ILP systems need to be given as BK symbolic representations of the digits, which could be achieved a learning task in Progol, Aleph, TILDE, and ILASP varies considerably. If it is difficult for ILP researchers to use ILP tools, then what hope do non-ILP researchers have? For ILP to be more widely adopted both inside and outside of academia, we must develop more standardised, user-friendly, and better-engineered tools.

Language biases. ILP allows a user to provide BK and a language bias. Both are important and powerful features, but only when used correctly. For instance, Metagol employs metarules (Section 4.4.2) to restrict the syntax of hypotheses and thus the hypothesis space. If a user can provide suitable metarules, then Metagol is extremely efficient. However, if a user cannot provide suitable metarules (which is often the case), then Metagol is almost useless. This same brittleness applies to ILP systems that employ mode declarations (Section 4.4.1). In theory, a user can provide very general mode declarations, such as only using a single type and allowing unlimited recall. In practice, however, weak mode declarations often lead to very poor performance. For good performance, users of mode-based systems often need to manually analyse a given learning task to tweak the mode declarations, often through a process of trial and error. Moreover, if a user makes a small mistake with a mode declaration, such as giving the wrong argument type, then the ILP system is unlikely to find a good solution. This need for an almost perfect language bias is severely holding back ILP from being widely adopted. To address this limitation, we think that an important direction for future work is to develop techniques for automatically identifying suitable language biases. Although there is some work on mode learning (McCreath & Sharma, 1995; Ferilli et al., 2004; Picado et al., 2017) and work on identifying suitable metarules (Cropper & Tourret, 2020), this area of research is largely under-researched.

PI and abstraction. Russell (2019) argues that the automatic invention of new high-level concepts is the most important step needed to reach human-level AI. New methods for PI (Section 5.5) have improved the ability of ILP to invent such high-level concepts. However, PI is still difficult and there are many challenges to overcome. For instance, in *inductive general game playing* (Cropper et al., 2020b), the task is to learn the symbolic rules of games from observations of gameplay, such as learning the rules of *connect four*. The reference solutions for the games come from the general game playing competition (Genesereth & Björnsson, 2013) and often contain auxiliary predicates to make them simpler. For instance, the rules for *connect four* are defined in terms of definitions for lines which are themselves defined in terms of columns, rows, and diagonals. Although these auxiliary predicates are not strictly necessary to learn the reference solution, inventing such predicates significantly reduces the size of the solution (sometimes by multiple orders of magnitude), which in turn makes them much easier to learn. Although new methods for PI (Section 5.5) can invent high-level concepts, they are not yet sufficiently powerful enough to perform well on the IGGP dataset. Making progress in this area would constitute a major advancement in ILP and a major step towards human-level AI.

Lifelong learning. Because of its symbolic representation, a key advantage of ILP is that learned knowledge can be remembered and explicitly stored in the BK. For this reason, ILP naturally supports *lifelong* (Silver et al., 2013), *multi-task* (Caruana, 1997), and *transfer learning* (Torrey & Shavlik, 2009), which are considered essential for human-like AI (Lake et al., 2016). The general idea behind all of these approaches is to reuse knowledge gained from solving one problem to help solve a different problem. Although early work in ILP explored this form of learning (Sammut, 1981; Quinlan, 1990), it has been under-explored until recently (Lin et al., 2014; Cropper, 2019, 2020; Hocquette & Muggleton, 2020; Dumancic et al., 2021), mostly because of new techniques for PI. For instance, Lin et al. (2014) learn 17 string transformations programs over time and show that their multi-task approach performs better than a single-task approach because

These relations hold for lists of any length and any type.

As a second example, suppose you want to learn the definition of a prime number. Then you might want to give a system the ability to perform arithmetic reasoning, such as using the Prolog relations:

$$B = \left\{ \begin{array}{l} \texttt{even(A):- 0 is mod(A,2).} \\ \texttt{odd(A):- 1 is mod(A,2).} \\ \texttt{sum(A,B,C):- C is A+B.} \\ \texttt{gt(A,B):- A>B.} \\ \texttt{lt(A,B):- A<B.} \end{array} \right\}$$

These relations are general and hold for arbitrary numbers and we do not need to pre-compute all the logical consequences of the definitions, which is impossible because there are infinitely many. By contrast, table-based ML approaches are restricted to finite propositional data. For instance, it is impossible to use the greater than relation over the set of natural numbers in a decision tree learner because it would require an infinite feature table.

4.3.1 Constraints

BK allows a human to encode prior knowledge of a problem. As a trivial example, if learning banking rules to determine whether two companies can lend to each other, you may encode a prior constraint to prevent two companies from lending to each other if they are owned by the same parent company:

```
:- lend(A,B), parent_company(A,C), parent_company(B,C).
```

Constraints are widely used in ILP (Zeng et al., 2014; Evans, 2020; Cropper & Morel, 2021). For instance, Inoue et al. (2013) represent knowledge as a causal graph and use constraints to denote impossible connections between nodes. Evans et al. (2021) use constraints to induce theories to explain sensory sequences. For instance, one requirement of their *unity condition* is that objects (constants) are connected via chains of binary relations. The authors argue that such constraints are necessary for the induced solutions to achieve good predictive accuracy.

4.3.2 Discussion

As with choosing appropriate features, choosing appropriate BK in ILP is crucial for good learning performance. ILP has traditionally relied on predefined and hand-crafted BK, often designed by domain experts. However, it is often difficult and expensive to obtain such BK. Indeed, the over-reliance on hand-crafted BK is a common criticism of ILP (Evans & Grefenstette, 2018). The difficulty is finding the balance of having enough BK to solve a problem, but not too much that a system becomes overwhelmed. We discuss these two issues.

Too little BK. If we use too little or insufficient BK then we may exclude a good hypothesis from the hypothesis space. For instance, reconsider the string transformation problem from the introduction, where we want to learn a program that returns the last character of a string from examples.

$$E^+ = \left\{ \begin{array}{l} \texttt{last([m,a,c,h,i,n,e], e)} \\ \texttt{last([l,e,a,r,n,i,n,g],g)} \\ \texttt{last([a,l,g,o,r,i,t,m], m)} \end{array} \right\}$$

To induce a hypothesis from these examples, we need to provide an ILP system with suitable BK. For instance, we might provide BK that contains relations for common list/string operations, such as `empty`, head, and `tail`. Given these three relations, an ILP system could learn the program:

$$\texttt{H} = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),empty(C),head(A,B).} \\ \texttt{last(A,B):- tail(A,C),last(C,B).} \end{array} \right\}$$

However, suppose that the user had not provided `tail` as BK. Then how could a system learn the above hypothesis? This situation is a major problem, as most systems can only use BK provided by a user. To mitigate this issue, there is research on enabling a system to automatically *invent* new predicate symbols, known as *predicate invention*, which we discuss in Section 5.5, which has been shown to mitigate missing BK (Cropper & Muggleton, 2015). However, ILP still heavily relies on much human input to solve a problem. Addressing this limitation is a major challenge.

Too much BK. As with too little BK, a major challenge is too much irrelevant BK. Too many relations (assuming that they can appear in a hypothesis) is often a problem because the size of the hypothesis space is a function of the size of the BK. Empirically, too much irrelevant BK is detrimental to learning performance (Srinivasan et al., 1995, 2003; Cropper, 2020), this also includes irrelevant language biases (Cropper & Tourret, 2020). Addressing the problem of too much BK has been under-researched. In Section 9, we suggest that this topic is a promising direction for future work, especially when considering the potential for ILP to be used for lifelong learning (Section 5.5.4).

4.4 Language Bias

The fundamental ILP problem is to search the hypothesis space for a suitable hypothesis. The hypothesis space contains all possible programs that can be built in the chosen representation language. Unrestricted, the hypothesis space is infinite, so it is important to restrict it to make the search feasible. To restrict the hypothesis space, systems enforce an *inductive bias* (Mitchell, 1997). A *language bias* enforces restrictions on hypotheses, such as restricting the number of variables, literals, and rules in a hypothesis. These restrictions can be categorised as either *syntactic* bias, restrictions on the form of a rule in a hypothesis, and *semantic* bias, restrictions on the behaviour of induced hypotheses (Adé et al., 1995). For instance, in the happy example (Example 1.1), we assumed that a hypothesis only contains predicate symbols that appear in the BK or examples. However, we need to encode this bias to give an ILP system. There are several ways of encoding a language bias, such as *grammars* (Cohen, 1994a), Dlabs (De Raedt & Dehaspe, 1997), production fields (Inoue, 2004), and predicate declarations (Cropper & Morel, 2021). We focus on *mode declarations* (Muggleton, 1995) and *metarules* (Cropper & Tourret, 2020), two popular language biases.

4.4.1 Mode Declarations

Mode declarations are the most popular form of language bias (Muggleton, 1995; Blockeel & De Raedt, 1998; Srinivasan, 2001; Ray, 2009; Corapi et al., 2010, 2011; Athakravi et al., 2013; Ahlgren & Yuen, 2013; Law et al., 2014; Katzouris et al., 2015). Mode declarations state which predicate symbols may appear in a rule, how often, and also their argument types. In the mode language, *modeh* declarations denote which literals may appear in the head of a rule and *modeb* declarations denote which literals may appear in the body of a rule. A mode declaration is of the form:

$$mode(recall, pred(m_1, m_2, \ldots, m_a))$$

8.4 Representation Learning

As introduced in Section 5.5, predicate invention (PI) – changing the representation of a problem by introducing new predicates symbols defined in terms of provided predicates – is one of the major open challenges in ILP. PI shares the goal with *feature* or *representation learning* (Bengio et al., 2013) stream of research originating in the deep learning community (LeCun et al., 2015). Representation learning aims to re-represent given data, be it an image or a relational knowledge base, in a *vector space* such that certain semantic properties are preserved. For instance, to represent a knowledge base (specified as a logic program) in a vector space, representation learning methods replace every constant and predicate with a vector such that the vectors of constants often appears in facts together are close in vector space. The benefit of such representation change is that now any tabular machine learning approach can operate on complex relational structures. The central goal of representation learning coincides with the one behind PI: improving learning performance by changing the representation of a problem.

Despite strong connections, there is little interaction between PI and representation learning. The main challenges in transferring the ideas from representation learning to PI are their different operating principles. It is not clear how symbolic concepts can be invented through table-based learning principles that current representation learning approaches use. Only a few approaches (Dumančić & Blockeel, 2017; Dumančić et al., 2019; Sourek et al., 2018) start from the core ideas in representation learning, strip them of numerical principles and re-invent them from symbolic principles. A more common approach is to transform relational data into a propositional tabular form that can be used as an input to a neural network (Dash et al., 2018; Kaur et al., 2019, 2020). A disadvantage of the latter approaches is that they only apply to propositional learning tasks, not to first-order program induction tasks where infinite domains are impossible to propositionalise. Approaches that force neural networks to invent symbolic constructs, such as ∂ILP and *neural theorem provers* (Rocktäschel & Riedel, 2017), do so by sacrificing the expressivity of logic (they can only learn short Datalog programs).

9. Summary And Limitations

In a survey paper from a decade ago, Muggleton et al. (2012) proposed directions for future research. There have since been major advances in many of these directions, including in PI (Section 5.5), using higher-order logic as a representation language (Section 4.4.2) and for hypotheses (Section 4.2.3), and applications in learning actions and strategies (Section 7). We think that these and other recent advances put ILP in a prime position to have a significant impact on AI over the next decade, especially to address the key limitations of standard forms of ML. There are, however, still many limitations that future work should address.

9.1 Limitations

User-friendly systems. Muggleton et al. (2012) argue that a problem with ILP is the lack of well-engineered tools. They state that whilst over 100 ILP systems have been built since the founding of ILP in 1991, less than a handful of systems can be meaningfully used by ILP researchers. One reason is that ILP systems are often only designed as prototypes and are often not well-engineered or maintained. Another major problem is that ILP systems are notoriously difficult to use: you often need a PhD in ILP to use any of the tools. Even then, it is still often only the developers of a system that know how to properly use it. This difficulty of use is compounded by ILP systems often using many different biases or even different syntax for the same biases. For instance, although they all use mode declarations, the way of specifying literals can be `true` at a time. For instance, the following annotated disjunction states that a ball can be either green, red, or blue, but not a combination of colours:

$\frac{1}{3}$`::colour(B,green);` $\frac{1}{3}$`::colour(B,red);` $\frac{1}{3}$`::colour(B,blue) :- ball(B).`

Though StarAI frameworks allow for incorrect BK, they add another level of complexity to learning: besides identifying the right program (also called structure in StarAI), the learning task also consists of learning the corresponding probabilities of probabilistic choices (also called parameters). Learning probabilistic logic programs is largely unexplored, with only a few existing approaches (De Raedt et al., 2015; Bellodi & Riguzzi, 2015).

8.3 Neural ILP

The successes of deep learning on various tasks have prompted researchers to investigate whether these techniques can be used to solve the ILP problem. What makes this research direction interesting is that the techniques are based on numerical optimisation and, if successful, could learn programs without combinatorial search, which is the core reason why ILP is a difficult problem. However, to leverage these techniques, ILP needs to be framed as a problem over a fixed set of variables, which is perhaps unnatural given that hypothesis spaces in ILP are combinatorial and essentially infinite. Additionally, integrating neural networks into ILP would make it possible for ILP to deal with unstructured data, such as images and sound, and noise associated with unstructured data. While integrating ILP and program synthesis with neural networks is a proliferating area, relatively few approaches tackle the ILP problem.

The majority of existing neural-ILP systems reframe the ILP problem as *structure learning by parameter learning*. Neural-ILP techniques make the ILP problem amenable to numerical optimisation by assuming that the entire hypothesis space is the solution program, instead of one member of it. As such a solution program would be able to entail almost anything, neural-ILP techniques relax the notion of entailment. That is, every member c of the hypothesis space is associated with valuation w_c which indicate the confidence that the entailments of c are correct. A fact entailed by c is said to be entailed with valuation w_c; a valuation of a fact is then the sum of valuations of all c that entail it. Consequently, the notion of entailment loses its crispness and needs to be interpreted in the continuous spectrum. The learning task is then to fit the valuations such that positive examples have high valuations and the valuations of negative examples are low, for which any numerical optimisation techniques can be used. These techniques make the hypothesis space finite by limiting its complexity and consequently only learn syntactically simple programs (e.g., the ones up to two literals and two clauses). The prominent examples of this paradigm are ∂ILP, neural theorem provers (Rocktäschel & Riedel, 2017), DiffLog (Si et al., 2019) and LRNN (Sourek et al., 2018).

At the time of publishing, only one approach can simultaneously learn a logic program that trains a neural network to solve a sensory part of the task (Dai & Muggleton, 2021). Dai and Muggleton (Dai & Muggleton, 2021) extends ILP with an abductive step (Flach & Hadjiantonis, 2013) that deduces labels for training neural networks from background knowledge and currently explored program. The authors demonstrate that their framework can learn arithmetic and sorting operations from digits represented as images. This is an underexplored research direction that holds a lot of promise in making ILP more capable of handling noise and unstructured data.

The following are all valid mode declarations:

```
modeh(1,happy(+person)).
modeb(*,member(+list,-element)).
modeb(1,head(+list,-element)).
modeb(2,parent(+person,-person)).
```

The first argument of a mode declaration is an integer denoting the *recall*. Recall is the maximum number of times that a mode declaration can be used in a rule[16]. Another way of understanding recall is that it bounds the number of alternative solutions for a literal. Providing a recall is a hint to a system to ignore certain hypotheses. For instance, if using the *parent* kinship relation, then we can set the recall to two, as a person has at most two parents. If using the *grandparent* relation, then we can set the recall to four, as a person has at most four grandparents. If we know that a relation is functional, such as head, then we can bound the recall to one. The symbol * denotes no bound.

The second argument denotes that the predicate symbol that may appear in the head (*modeh*) or body (*modeb*) of a rule and the type of arguments it takes. The symbols +, −, and # denote whether the arguments are *input*, *output*, or *ground* arguments respectively. An *input* argument specifies that, at the time of calling the literal, the corresponding argument must be instantiated. In other words, the argument needs to be bound to a variable that already appears in the rule. An *output* argument specifies that the argument should be bound after calling the corresponding literal. A *ground* argument specifies that the argument should be ground and is often used to learn rules with constant symbols in them.

To illustrate mode declarations, consider the modes:

```
modeh(1,target(+list,-char)).
modeb(*,head(+list,-char)).
modeb(*,tail(+list,-list)).
modeb(1,member(+list,-list)).
modeb(1,equal(+char,-char)).
modeb(*,empty(+list)).
```

Given these modes, the rule `target(A,B):- head(A,C),tail(C,B)` is mode inconsistent because `modeh(1,target(+list,-char))` requires that the second argument of target (B) is char and the mode `modeb(*,tail(+list,-list))` requires that the second argument of tail (B) is a list, so this rule is mode inconsistent. The rule `target(A,B):- empty(A),head(C,B)` is also mode inconsistent because `modeb(*,head(+list,-char))` requires that the first argument of head (C) is instantiated but the variable C is never instantiated in the rule.

By contrast, the following rules are all mode consistent:

```
target(A,B):- tail(A,C),head(C,B).
target(A,B):- tail(A,C),tail(C,D),equal(C,D),head(A,B).
target(A,B):- tail(A,C),member(C,B).
```

Depending on the specific system, modes can also support the introduction of constant symbols. In Aleph, an example of such a declaration is `modeb(*,length(+list,#int))`, which would allow integer values to be included in rules.

[16] The recall of a *modeh* declaration is almost always useless and is often set to 1.

Different systems use mode declarations in slightly different ways. Progol and Aleph use mode declarations with input/output argument types because they induce Prolog programs, where the order of literals in a rule matters. By contrast, ILASP induces ASP programs, where the order of literals in a rule does not matter, so ILASP does not use input/output arguments.

4.4.2 Metarules

Metarules[17] are a popular form of syntactic bias and are used by many systems (De Raedt & Bruynooghe, 1992; Flener, 1996; Kietz & Wrobel, 1992; Wang et al., 2014; Muggleton et al., 2015; Cropper & Muggleton, 2016; Kaminski et al., 2018; Evans & Grefenstette, 2018; Bain & Srinivasan, 2018). Metarules are second-order rules which define the structure of learnable programs which in turn defines the hypothesis space. For instance, to learn the `grandparent` relation given the `parent` relation, the *chain* metarule would be suitable:

```
P(A,B):- Q(A,C), R(C,B).
```

The letters P, Q, and R denote second-order variables (variables that can be bound to predicate symbols) and the letters A, B and C denote first-order variables (variables that can be bound to constant symbols). Given the *chain* metarule, the background `parent` relation, and examples of the `grandparent` relation, ILP approaches will try to find suitable substitutions for the second-order variables, such as the substitutions {P/grandparent, Q/parent, R/parent} to induce the theory:

```
grandparent(A,B):- parent(A,C),parent(C,B).
```

Despite their widespread use, there is little work determining which metarules to use for a given learning task. Instead, these approaches assume suitable metarules as input or use metarules without any theoretical guarantees. In contrast to other forms of bias in ILP, such as modes or grammars, metarules are themselves logical statements, which allows us to reason about them. For this reason, there is preliminary work in reasoning about metarules to identify universal sets suitable to learn certain fragments of logic programs (Cropper & Muggleton, 2014; Tourret & Cropper, 2019; Cropper & Tourret, 2020). Despite this preliminary work, deciding which metarules to use for a given problem is still a major challenge, which future work must address.

4.4.3 Discussion

Choosing an appropriate language bias is essential to make an ILP problem tractable because it defines the hypothesis space. If the bias is too *weak*, then the search can become intractable. If the bias is too *strong* then we risk excluding a good solution from the hypothesis space. This trade-off is one of the major problems holding ILP back from being widely used[18]. To understand the impact of an inappropriate

[17] Metarules were introduced as *clause schemata* by Emde et al. (1983) and were notably used in Mobal system (Kietz & Wrobel, 1992). Metarules are also called *second-order schemata* (De Raedt & Bruynooghe, 1992) and *program schemata* (Flener, 1996), amongst many other names.

[18] The Blumer bound (Blumer et al., 1987) (the bound is a reformulation of Lemma 2.1) helps explain this trade-off. This bound states that given two hypothesis spaces, searching the smaller space will result in fewer errors compared to the larger space, assuming that the target hypothesis is in both spaces. Here lies the problem: how to choose a learner's hypothesis space so that it is large enough to contain the target hypothesis yet small enough to be efficiently searched. To know more about this aspect of ILP, we first recommend Chapter 7 of Mitchell's (1997) Machine Learning book, which is, in our view, still the best introductory exposition of computational learning theory, and then work specific to ILP (Cohen, 1995b, 1995d, 1995c; Gottlob et al., 1997).

ming languages (PL). The two major[46] differences between ML and PL approaches are (i) the generality of solutions (synthesised programs) and (ii) noise handling. PL approaches often aim to find *any* program that fits the specification, regardless of whether it generalises. Indeed, PL approaches rarely evaluate the ability of their systems to synthesise solutions that generalise, i.e. they do not measure predictive accuracy (Feser et al., 2015; Osera & Zdancewic, 2015; Albarghouthi et al., 2017; Si et al., 2018; Raghothaman et al., 2020). By contrast, the major challenge in ML (and thus ILP) is learning hypotheses that *generalise* to unseen examples. Indeed, it is often trivial to learn an overly specific solution for a given problem. For instance, an ILP system can trivially construct the bottom clause (Muggleton, 1995) for each example. Similarly, noise handling is a major problem in ML, yet is rarely considered in the PL literature.

Besides ILP, inductive program synthesis has been studied in many areas of ML, including deep learning (Balog et al., 2017; Ellis et al., 2018, 2019). The main advantages of neural approaches are that they can handle noisy BK, as illustrated by ∂ILP, and can harness tremendous computational power (Ellis et al., 2019). However, neural methods often require many more examples (Reed & de Freitas, 2016; Dong et al., 2019) to learn concepts that symbolic ILP can learn from just a few. Another disadvantage of neural approaches is that they often require hand-crafted neural architectures for each domain. For instance, the REPL approach (Ellis et al., 2019) needs a hand-crafted grammar, interpreter, and neural architecture for each domain. By contrast, because ILP uses logic programming as a uniform representation for examples, BK, and hypotheses, it can easily be applied to arbitrary domains.

8.2 StarAI

As ILP builds upon logic programs and logical foundations of knowledge representation, ILP also inherits one of their major limitations: the inability to handle uncertain or incorrect BK. To overcome this limitation, the field of statistical relational artificial intelligence (StarAI) (De Raedt & Kersting, 2008b; De Raedt et al., 2016) unites logic programming with probabilistic reasoning.

StarAI formalisms allow a user to explicitly quantify the confidence in the correctness of the BK by annotating parts of BK with probabilities. Perhaps the simplest flavour of StarAI languages, and the one that directly builds upon logic programming and Prolog, is a family of languages based on distribution semantics (Sato, 1995; Sato & Kameya, 2001; De Raedt et al., 2007). Problog (De Raedt et al., 2007), a prominent member of this family, represents a minimal extension of Prolog that supports such stochastic execution. Problog introduces two types of probabilistic choices: probabilistic facts and annotated disjunctions. Probabilistic facts are the most basic stochastic unit in Problog. They take the form of logical facts labeled with a probability p and represent a Boolean random variable that is `true` with probability p and `false` with probability $1 - p$. For instance, the following probabilistic fact states that there is 1% chance of an earthquake in Naples.

```
0.01::earthquake(naples).
```

In contrast to deterministic logic programs in which a fact is always true if so stated, the Problog engine determines the truth assignment of a probabilistic fact when it encounters it during SLD resolution, the main execution principle of logic programs. How often a fact is deemed true or false is guided by the stated probability. An alternative interpretation of this statement is that 1% of executions of the probabilistic program would observe an earthquake. Whereas probabilistic facts introduce non-deterministic behaviour on the level of facts, annotated disjunctions introduce non-determinism on the level of clauses. Annotated disjunctions allow for multiple literals in the head, but only one of the head

[46] Minor differences include the form of specification and theoretical results.

in simulations and real-world environments. Metagol$_O$ (Cropper & Muggleton, 2015) learns robot strategies considering their resource efficiency and Antanas et al. (2015) recognise graspable points on objects through relational representations of objects.

Games. Inducing game rules has a long history in ILP, where chess has often been the focus (Goodacre, 1996; Morales, 1996; Muggleton et al., 2009). For instance, Bain (1994) studies inducing rules to determine the legality of moves in the chess KRK (king-rook-king) endgame. Castillo and Wrobel (2003) uses a top-down ILP system and active learning to induce a rule for when a square is safe in the game minesweeper. Legras et al. (2018) show that Aleph and TILDE can outperform an SVM learner in the game of Bridge. Law et al. (2014) uses ILASP to induce the rules for Sudoku and show that this more expressive formalism allows for game rules to be expressed more compactly. Cropper et al. (2020b) introduce the ILP problem of *inductive general game playing*: the problem of inducing game rules from observations, such as *Checkers*, *Sokoban*, and *Connect Four*.

Other. Other notable applications include learning event recognition systems (Katzouris et al., 2015, 2016), tracking the evolution of online communities (Athanasopoulos et al., 2018), the MNIST dataset (Evans & Grefenstette, 2018), and requirements engineering (Alrajeh et al., 2013).

8. Related Work

8.1 Program Synthesis

Because ILP induces programs, it is also a form of *program synthesis* (Shapiro, 1983), where the goal is to build a program from a specification. Universal induction methods, such as Solomonoff induction (Solomonoff, 1964a, 1964b) and Levin search (Levin, 1973) are forms of program synthesis. However, universal methods are impractical because they learn only from examples and, as Mitchell (1997) points out, bias-free learning is futile.

Deductive program synthesis approaches (Manna & Waldinger, 1980) take full specifications as input and are efficient at building programs. Universal induction methods take only examples as input and are inefficient at building programs. There is an area in between called *inductive program synthesis*[45]. Similar to universal induction methods, inductive program synthesis systems learn programs from incomplete specifications, typically input/output examples. In contrast to universal induction methods, inductive program synthesis systems use BK, and are thus less general than universal methods, but are more practical because the BK is a form of inductive bias (Mitchell, 1997) which restricts the hypothesis space. When given no BK, and thus no inductive bias, inductive program synthesis methods are equivalent to universal induction methods.

Early work on inductive program synthesis includes Plotkin (1971) on least generalisation, Vera (1975) on induction algorithms for predicate calculus, Summers (1977) on inducing Lisp programs, and Shapiro (1983) on inducing Prolog programs. Interest in inductive program synthesis has grown recently, partly due to applications in real-world problems, such as end-user programming (Gulwani, 2011). Inductive program synthesis interests researchers from many areas of computer science, notably ML and program-

[45] Inductive program synthesis is often called *program induction* (Lin et al., 2014; Lake et al., 2015; Cropper, 2017; Ellis et al., 2018), *programming by example* (Lieberman, 2001), and *inductive programming* (Gulwani et al., 2015), amongst many other names. Gulwani et al. (2017) divide inductive program synthesis into two categories: (i) program induction, and (ii) program synthesis. They say that program induction approaches are neural architectures that learn a network that is capable of replicating the behaviour of a program. By contrast, they say that program synthesis approaches output or return an interpretable program.

language bias, consider the string transformation example in Section 1.2. Even if all necessary background relations are provided, not providing a recursive metarule (e.g. `R(A,B):- P(A,C), R(C,B)`) would prevent a metarule-based system from inducing a program that generalises to lists of any length. Similarly, not providing a recursive mode declaration for the target relation would prevent a mode-based system from finding a good hypothesis.

Different language biases offer different benefits. Mode declarations are expressive enough to enforce a strong bias to significantly prune the hypothesis space. They are especially appropriate when a user has much knowledge about their data and can, for instance, determine suitable recall values. If a user does not have such knowledge, then it can be very difficult to determine suitable mode declarations. Moreover, if a user provides weak mode declarations (for instance with infinite recall, a single type, and no input/output arguments), then the search quickly becomes intractable. Although there is some work on learning mode declarations (McCreath & Sharma, 1995; Ferilli et al., 2004; Picado et al., 2017), it is still a major challenge to choose appropriate ones.

A benefit of metarules is that they require little knowledge of the BK and a user does not need to provide recall values, types, or specify input/output arguments. Because they precisely define the form of hypotheses, they can greatly reduce the hypothesis space, especially if the user knows about the class of programs to be learned. However, as previously mentioned, the major downside with metarules is determining which metarules to use for an arbitrary learning task. Although there is some preliminary work in identifying universal sets of metarules (Cropper & Muggleton, 2014; Tourret & Cropper, 2019; Cropper & Tourret, 2020), deciding which metarules to use for a given problem is a major challenge, which future work must address.

4.5 Search Method

Having defined the hypothesis space, the next problem is to efficiently search it. There are two traditional search methods: *bottom-up* and *top-down*. These methods rely on notions of generality, where one program is more *general* or more *specific* than another (Section 2.4). A generality relation imposes an order over the hypothesis space. Figure 2 shows this order using theta-subsumption, the most popular ordering relation. A system can exploit this ordering during the search for a hypothesis. For instance, if a clause does not entail a positive example, then there is no need to explore any of its specialisations because it is logically impossible for them to entail the example. Likewise, if a clause entails a negative example, then there is no need to explore any of its generalisations because they will also entail the example.

The above paragraph refers only to generality orders on single clauses, as many systems employ the covering algorithm whereby hypotheses are constructed iteratively clause by clause (Quinlan, 1990; De Raedt & Dehaspe, 1997; Muggleton, 1995; Blockeel & De Raedt, 1998; Srinivasan, 2001). However, some systems (Shapiro, 1983; Bratko, 1999; Cropper & Morel, 2021) induce theories formed of multiple clauses and thus require generality orders over clausal *theories*. We refer an interested reader to Chapter 7 in the work of De Raedt (2008) for more information about inducing theories.

4.5.1 Top-down

Top-down algorithms, (Quinlan, 1990; Blockeel & De Raedt, 1998; Bratko, 1999; Muggleton et al., 2008) start with a general hypothesis and then specialise it. For instance, HYPER (Bratko, 1999) searches a tree in which the nodes correspond to hypotheses. Each child of a hypothesis in the tree is more specific than or equal to its predecessor in terms of theta-subsumption, i.e. a hypothesis can only entail a subset

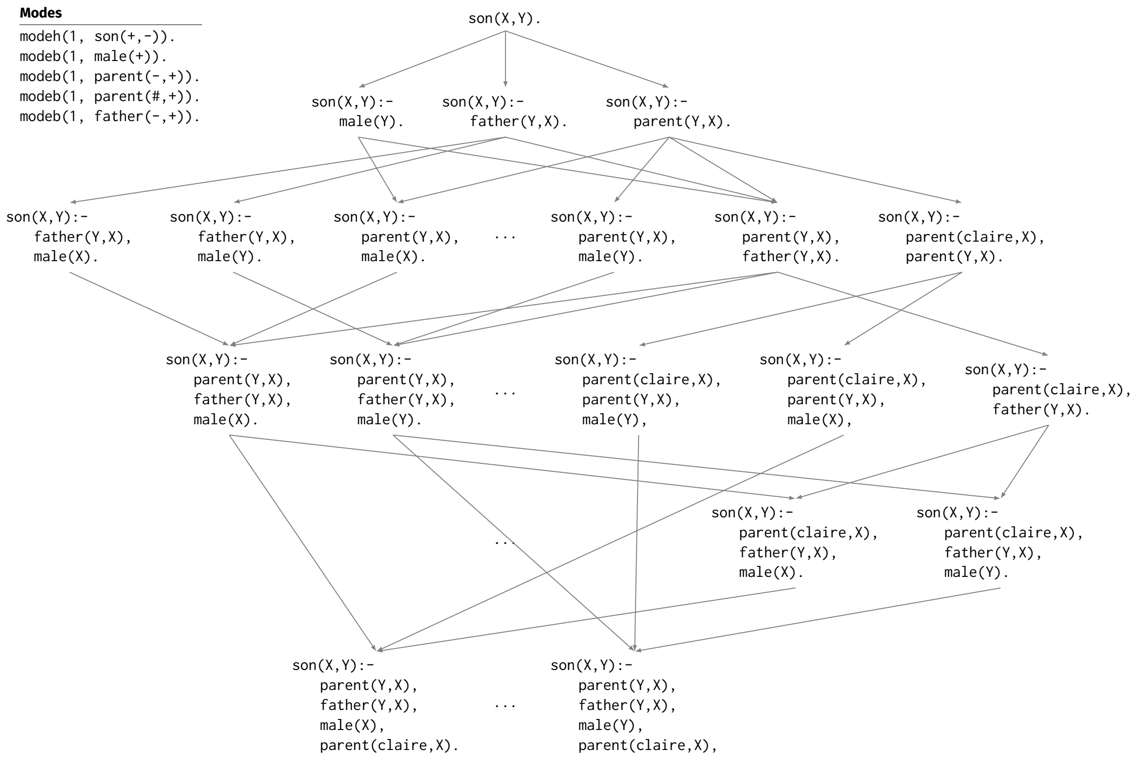

Modes
modeh(1, son(+,-)).
modeb(1, male(+)).
modeb(1, parent(-,+)).
modeb(1, parent(#,+)).
modeb(1, father(-,+)).
son(X,Y).
son(X,Y):- male(Y).
son(X,Y):- father(Y,X).
son(X,Y):- parent(Y,X).
son(X,Y):- father(Y,X), male(X).
son(X,Y):- father(Y,X), male(Y).
son(X,Y):- parent(Y,X), male(X).
...
son(X,Y):- parent(Y,X), male(Y).
son(X,Y):- parent(Y,X), father(Y,X).
son(X,Y):- parent(claire,X), parent(Y,X).
son(X,Y):- parent(Y,X), father(Y,X), male(X).
son(X,Y):- parent(Y,X), father(Y,X), male(Y).
...
son(X,Y):- parent(claire,X), parent(Y,X), male(Y),
son(X,Y):- parent(claire,X), parent(Y,X), male(X),
son(X,Y):- parent(claire,X), father(Y,X).
...
son(X,Y):- parent(claire,X), father(Y,X), male(X).
son(X,Y):- parent(claire,X), father(Y,X), male(Y).
son(X,Y):- parent(Y,X), father(Y,X), male(X), parent(claire,X).
...
son(X,Y):- parent(Y,X), father(Y,X), male(Y), parent(claire,X),

Figure 2: The generality relation orders the hypothesis space into a lattice (an arrow connects a hypothesis with its specialisation). The hypothesis space is built from the modes and only shown partially (# indicates that a constant needs to be used as an argument; only claire is used as a constant here). The most general hypothesis sits on the top of the lattice, while the most specific hypotheses are at the bottom. The *top-down* lattice traversal starts at the top, with the most general hypothesis, and specialises it moving downwards through the lattice. The *bottom-up* traversal starts at the bottom, with the most specific hypothesis, and generalises it moving upwards through the lattice.

of the examples entailed by its parent. The construction of hypotheses is based on *hypothesis refinement* (Shapiro, 1983; Nienhuys-Cheng & Wolf, 1997). If a hypothesis is considered that does not entail all the positive examples, it is immediately discarded because it can never be refined into a complete hypothesis.

4.5.2 Bottom-up

Bottom-up algorithms start with the examples and generalise them (Muggleton, 1987; Muggleton & Buntine, 1988; Muggleton & Feng, 1990; Muggleton et al., 2009; Inoue et al., 2014). For instance, Golem (Muggleton & Feng, 1990) generalises pairs of examples based on relative least-general generalisation (RLGG) (Buntine, 1988). To introduce RLGG, we start by introducing Plotkin's (1971) notion of least-general generalisation (LGG), which tells us how to generalise two clauses. Given two clauses, the LGG operator returns the most specific single clause that is more general than both of them.

Data curation and transformation. Another successful application of ILP is in data curation and transformation, which is again largely because ILP can learn executable programs. The most prominent example of such tasks is string transformations, such as the example given in the introduction. There is much interest in this topic, largely due to success in synthesising programs for end-user problems, such as string transformations in Microsoft Excel (Gulwani, 2011). String transformations have become a standard benchmark for some recent ILP papers (Lin et al., 2014; Cropper et al., 2020; Cropper & Dumančić, 2020; Cropper & Morel, 2021; Cropper, 2019). Other transformation tasks include extracting values from semi-structured data (e.g. XML files or medical records), extracting relations from ecological papers, and spreadsheet manipulation (Cropper et al., 2015).

Learning from trajectories. Learning from interpretation transitions (LFIT) (Inoue et al., 2014) automatically constructs a model of the dynamics of a system from the observation of its state transitions[44]. Given time-series data of discrete gene expression, it can learn gene interactions, thus allowing to explain and predict states changes over time (Ribeiro et al., 2020). LFIT has been applied to learn biological models, like Boolean Networks, under several semantics: memory-less deterministic systems (Inoue et al., 2014; Ribeiro & Inoue, 2014), probabilistic systems (Martínez et al., 2015) and their multi-valued extensions (Ribeiro et al., 2015; Martínez et al., 2016). Martínez et al. (2015, 2016) combine LFIT with a reinforcement learning algorithm to learn probabilistic models with exogenous effects (effects not related to any action) from scratch. The learner was notably integrated into a robot to perform the task of clearing the tableware on a table. In this task external agents interacted, people brought new tableware continuously and the manipulator robot had to cooperate with mobile robots to take the tableware to the kitchen. The learner was able to learn a usable model in just five episodes of 30 action executions. Evans et al. (2021) apply the *Apperception Engine* to explain sequential data, such as rhythms and simple nursery tunes, image occlusion tasks, and sequence induction intelligence tests. They show that their system can perform human-level performance.

Natural language processing. Many natural language processing tasks require an understanding of the syntax and semantics of the language. ILP is well-suited for addressing such tasks for three reasons (i) it is based on an expressive formal language that can capture/respect the syntax and semantics of the natural language, (ii) linguistics knowledge and principles can be integrated into ILP systems, and (iii) the learnt clauses are understandable to a linguist. ILP has been applied to learn grammars (Mooney & Califf, 1995; Muggleton et al., 2014; Law et al., 2019) and parsers (Zelle & Mooney, 1996, 1995; Mooney, 1999) from examples. For an extensive overview of language tasks that can benefit from ILP see the paper by Dzeroski et al. (1999).

Physics-informed learning. A major strength of ILP is its ability to incorporate and exploit background knowledge. Several ILP applications solve problems from *first principles*: provided physical models of the basic primitives, ILP systems can induce the target hypothesis whose behaviour is derived from the basic primitives. For instance, ILP systems can use a theory of light to understand images (Dai et al., 2017; Muggleton et al., 2018). Similarly, simple electronic circuits can be constructed from the examples of the target behaviour and the physics of basic electrical components (Grobelnik, 1992) and models of simple dynamical systems can be learned given the knowledge about differential equations (Bratko et al., 1991).

Robotics. Similarly to the previous category, robotics applications often require incorporating domain knowledge or imposing certain requirements on the learnt programs. For instance, The Robot Engineer (Sammut et al., 2015) uses ILP to design tools for robots and even complete robots, which are tests

[44] The LFIT implementations are available at `https://github.com/Tony-sama/pylfit`

Disadvantages. Deciding which metarules to use for a given task is a major problem. Given too many metarules, the hypothesis space might be so large that the search is intractable. Given insufficient metarules, the hypothesis space might be too small as to exclude a good hypothesis. For some tasks, such as string transformations, it is straightforward to choose a suitable set of metarules because one already knows the general form of hypotheses. However, when one has little knowledge of the solutions, then Metagol is unsuitable. Although there is preliminary work in identifying universal sets of metarules (Cropper & Muggleton, 2014; Tourret & Cropper, 2019; Cropper & Tourret, 2020), this work mostly focuses on dyadic logic. If a problem contains predicates of arities greater than two, then Metagol is unsuitable. Finally, Metagol cannot handle noisy examples and struggles to learn large programs (Cropper, 2017; Cropper & Dumančić, 2020; Cropper & Morel, 2021).

7. Applications

We now briefly discuss some application areas of ILP.

Bioinformatics and drug design. Perhaps the most prominent application of ILP is in bioinformatics and drug design. ILP is especially suitable for such problems because biological structures, including molecules and protein interaction networks, can easily be expressed as relations: molecular bonds define relations between atoms and interactions define relations between proteins. Moreover, as mentioned in the introduction, ILP induces human-readable models. ILP can, therefore, make predictions based on the (sub)structured present in biological structures which domain experts can interpret. The types of task ILP has been applied to include identifying and predicting ligands (substructures responsible for medical activity) (Finn et al., 1998; Srinivasan et al., 2006; Kaalia et al., 2016), predicting mutagenic activity of molecules and identifying structural alerts for the causes of chemical cancers (Srinivasan et al., 1997, 1996), learning protein folding signatures (Turcotte et al., 2001), inferring missing pathways in protein signalling networks (Inoue et al., 2013), and modelling inhibition in metabolic networks (Tamaddoni-Nezhad et al., 2006).

Robot scientist. One of the most notable applications of ILP was in the *Robot Scientist* project (King et al., 2009). The Robot Scientist uses logical BK to represent the relationships between protein-coding sequences, enzymes, and metabolites in a pathway. The Robot Scientist uses ILP to automatically generate hypotheses to explain data and then devises experiments to test hypotheses, run the experiments, interpret the results, and then repeat the cycle (King et al., 2004). Whilst researching yeast-based functional genomics, the Robot Scientist became the first machine to independently discover new scientific knowledge (King et al., 2009).

Ecology. There has been much recent work on applying ILP in ecology (Bohan et al., 2011; Tamaddoni-Nezhad et al., 2014; Bohan et al., 2017). For instance, Bohan et al. (2011) use ILP to generate plausible and testable hypotheses for trophic relations ('who eats whom') from ecological data.

Program analysis. Due to the expressivity of logic programs as a representation language, ILP systems have found successful applications in software design. ILP systems have proven effective in learning SQL queries (Albarghouthi et al., 2017; Sivaraman et al., 2019) and programming language semantics (Bartha & Cheney, 2019). Other applications include code search (Sivaraman et al., 2019), in which an ILP system interactively learns a search query from examples, and software specification recovery from execution behaviour (Cohen, 1994b, 1995a).

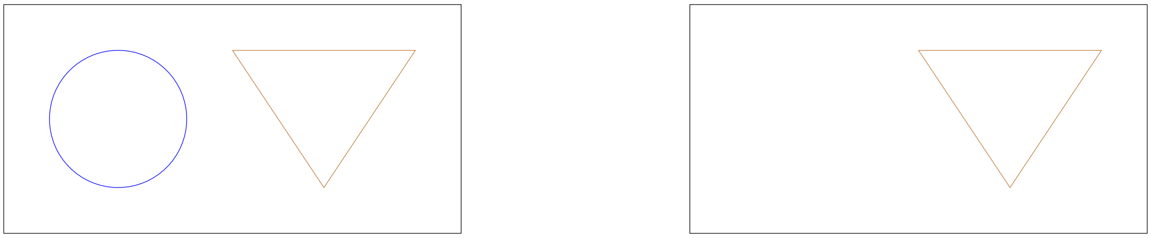


Figure 3: Bongard problems

To define the LGG of two clauses, we start with the LGG of terms:

- $\mathtt{lgg(f(s_1,\ldots,s_n),\ f(t_1,\ldots,t_m))} = \mathtt{f(lgg(s_1,t_1),\ldots,lgg(s_n,t_n))}$.
- $\mathtt{lgg(f(s_1,\ldots,s_n),\ g(t_1,\ldots,t_m))} = \mathrm{V}$ (a variable),
- $\mathtt{lgg(f(s_1,\ldots,s_n),\ V)} = \mathrm{V}'$ (a new variable).

Note that a constant is a functor with zero arguments, and thus the above rules apply. We define the LGG of literals:

- $\mathtt{lgg(p(s_1,\ldots,s_n),\ p(t_1,\ldots,t_n))} = \mathtt{p(lgg(s_1,t_1),\ldots,lgg(s_n,t_n))}$.
- $\mathtt{lgg(\neg p(s_1,\ldots,s_n),\ \neg p(t_1,\ldots,t_n))} = \mathtt{\neg p(lgg(s_1,t_1),\ldots,lgg(s_n,t_n))}$
- $\mathtt{lgg(p(s_1,\ldots,s_n),\ q(t_1,\ldots,t_n))}$ is undefined
- $\mathtt{lgg(p(s_1,\ldots,s_n),\ \neg p(t_1,\ldots,t_n))}$ is undefined
- $\mathtt{lgg(\neg p(s_1,\ldots,s_n),\ p(t_1,\ldots,t_n))}$ is undefined.

Using the set representation of a clause, we define the LGG of two clauses:

$$\mathtt{lgg(cl_1,cl_2)} = \{\, \mathtt{lgg(l_1,l_2)} \text{ for } \mathtt{l_1} \in \mathtt{cl_1} \text{ and } \mathtt{l_2} \in \mathtt{cl_2}\text{, such that } \mathtt{lgg(l_1,l_2)\ is\ defined} \,\}$$

In other words, the LGG of two clauses is a LGG of all *pairs* of literals of the two clauses.

Having defined the notion of LGG, we move to defining RLGG. Buntine's (1988) notion of *relative least-general generalisation* computes a LGG of two examples *relative* to the BK (assumed to be a set of facts):

$$\mathtt{rlgg(e_1,e_2)} = \mathtt{lgg(e_1\ \text{:-}\ BK,\ e_2\ \text{:-}\ BK)}.$$

Example 4. To illustrate RLGG, consider the Bongard problems in Figure 3 where the goal is to spot the common factor in both images. Assume that the images are described with the BK:

$$B = \left\{ \begin{array}{l} \mathtt{triangle(o1).} \\ \mathtt{triangle(o3).} \\ \mathtt{circle(o2).} \\ \mathtt{points(o1,down).} \\ \mathtt{points(o3,down).} \\ \mathtt{contains(1,o1).} \\ \mathtt{contains(1,o2).} \\ \mathtt{contains(2,o3).} \end{array} \right\}$$

That is, the BK states that objects o1 and o3 are triangles, object o2 is a circle, objects o1 and o2 point down, image 1 contains objects o1 and o2, while image 2 contains object 3. We can use RLGG to identify the common factor, i.e., to find a program representing the common factor. We will denote the example images as bon(1) and bon(2). We start by formulating the clauses describing examples relative to BK and removing irrelevant parts of BK:

```
lgg(
      (bon(1) :-  contains(1,o1), contains(1,o2), triangle(o1), points(o1,down),
                  circle(o2) , ~~contains(2,o3), triangle(o3), points(o3,down)~~.),
      (bon(2) :-  ~~contains(1,o1), contains(1,o2), triangle(o1), points(o1,down),
                  circle(o2)~~, contains(2,o3), triangle(o3), points(o3,down).)
)
```

We proceed by computing LGG for the heads and the bodies of the two clauses separately[19]. The LGG of the head literals is lgg(bon(1),bon(2)) = bon(lgg(1,2)) = bon(X). An important thing to note here is that we have to use the *same variable* for the *same ordered pair of terms* everywhere. For instance, we have used the variable X for lgg(1,2) and we have to use the same variable every time we encounter the same pair of terms. To compute the LGG of the body literals, we compute the LGG for all pairs of body literals:

```
{ lgg(contains(1,o1),contains(2,o3)), ~~lgg(contains(1,o1),triangle(o3))~~, ~~lgg(contains(1,o1),points(o3,down))~~
lgg(contains(1,o2),contains(2,o3)), ~~lgg(contains(1,o2),triangle(o3))~~, ~~lgg(contains(1,o1),points(o3,down))~~,
~~lgg(triangle(o1),contains(2,o3))~~, lgg(triangle(o1),triangle(o3)), ~~lgg(triangle(o1),points(o3,down))~~,
~~lgg(points(o1,down),contains(2,03))~~, ~~lgg(points(o1,down),triangle(o3))~~, lgg(points(o1,down),points(o3,down)),
~~lgg(circle(o2),contains(2,o3))~~, ~~lgg(circle(o2),triangle(o3))~~, ~~lgg(circle(o2),points(o3,down))~~}.
```

Eliminating undefined LGGs leaves us with:

```
{ lgg(contains(1,o1),contains(2,o3)),  lgg(contains(1,o2),contains(2,o3)),
  lgg(triangle(o1),triangle(o3)), lgg(points(o1,down),points(o3,down))  }.
```

Finally, computing the individual LGGs[20]

```
{contains(X,Y),  contains(X,Z),  triangle(Y), points(Y,down)  }.
```

and eliminating the redundant literal contains(X,Z) (as it is subsumed by contains(X,Y)) gives us the clause:

```
bon(X):- contains(X,Y),triangle(Y),points(Y,down).
```

We suggest the book by De Raedt (2008) for more information about generality orders.

4.5.3 Top-down And Bottom-up

Progol is one of the most important systems and has inspired many other approaches (Srinivasan, 2001; Ray, 2009; Ahlgren & Yuen, 2013), including Aleph, which we cover in detail in Section 6.1. Progol is, however, slightly confusing because it is a top-down system but it first uses a bottom-up approach to bound the search space. Indeed, many authors only consider it a top-down approach. Progol uses a set covering algorithm. Starting with an empty program, Progol picks an uncovered positive example to generalise. To generalise an example, Progol uses mode declarations (Section 4.4.1) to build the *bottom clause* (Muggleton, 1995), the logically most-specific clause that explains the example. The use of a bottom clause bounds the search from above (the empty set) and below (the bottom clause). In this way, Progol is a bottom-up approach because it starts with a bottom clause and tries to generalise it. However,

[19] We can do this because when a clause is converted to the set representation, the literals in the body and head have different signs (body literals are negative, while the head literals are positive) which results in an undefined LGG.

[20] with the following LGGs on terms: lgg(1,2) = X, lgg(o1,o3) = Y, lgg(o2,o3) = Z

No combination of these clauses can prove the examples, so Metagol must use PI to learn a solution. To use PI, Metagol will try to prove the example using the *chain* metarule, which will lead to the construction of the program:

```
f([i,l,p],p):- Q([i,l,p],C),R(C,p).
```

Metagol would save a metasub for this clause:

```
sub(chain,[f,Q,R]).
```

Metagol will then try to recursively prove both `Q([i,l,p],C)` and `R(C,p)`. To prove `Q([i,l,p],C)`, Metagol will say that it cannot prove it using a relation in the BK, so it will try to invent a new predicate symbol, which leads to the new atom `f_1([i,l,p],C)` and the program:

```
f([i,l,p],p):- f_1([i,l,p],C),R(C,p).
```

Note that this binds `Q` in the metasub to `f_1`. Metagol then tries to prove the `f_1([i,l,p],C)` and `R(C,p)` atoms. To prove `f_1([i,l,p],C)`, Metagol could use the chain metarule to form the clause:

```
f_1([i,l,p],C):- Q2([i,l,p],D),R2(D,C).
```

Metagol would save another metasub for this clause:

```
sub(chain,[f_1,Q2,R2]).
```

Metagol then tries to prove the `Q2([i,l,p],D)` and `R2(D,C)` atoms. Metagol can prove `Q2([i,l,p],D)` by binding `Q2` to `tail` so that `D` is bound to `[l,p]`. Metagol can then prove `R2([l,p],C)` by binding `R2` to `tail` so that `C` is bound `[p]`. Remember that the binding of variables is propagated through the program, so `C` in `R(C,p)` is now bound to `R([p],p)`. Metagol then tries to prove the remaining atom `R([p],p)`, which it can by binding `R` to head. The proof of all the atoms is now complete and the final metasubs are:

```
sub(chain,[f,f_1,head]).
sub(chain,[f_1,tail,tail]).
```

These metasubs correspond to the program:

```
f(A,B):- f_1(A,C),head(C,B).
f_1(A,B):- tail(A,C),tail(C,B).
```

6.4.4 Discussion

Advantages. Metagol supports PI, learning recursive programs, and is guaranteed to learn the smallest program. Because it uses metarules, Metagol can tightly restrict the hypothesis space, which means that it is extremely efficient at finding solutions. The basic Metagol implementation is less than 100 lines of Prolog code, which makes Metagol easy to adapt, such as to support NAF (Siebers & Schmid, 2018), types (Morel et al., 2019), learning higher-order programs (Cropper et al., 2020), learning efficient programs (Cropper & Muggleton, 2015, 2019), and Bayesian inference (Muggleton et al., 2013).

This metasub means that Metagol has induced the clause:

```
grandparent(A,B):- mother(A,C),mother(C,B).
```

After proving the example, Metagol moves to Step 1 and picks another example to generalise. Suppose it picks the example grandparent(steve,amelia). In Step 2a, Metagol tries to prove this atom using the BK, which again fails, so tries to prove this atom using an already induced clause, which also fails. Therefore, Metagol tries to prove this atom using a metarule. Metagol can again use the *chain* metarule but with different substitutions to form the metasub:

```
sub(chain,[grandparent,father,mother])
```

This metasub corresponds to the clause:

```
grandparent(A,B):- father(A,C),mother(C,B).
```

Metagol has now proven the first two examples by inducing the clauses:

```
grandparent(A,B):- mother(A,C),mother(C,B).
grandparent(A,B):- father(A,C),mother(C,B).
```

If given no bound on the program size, then Metagol would prove the other two examples the same way by inducing two more clauses to finally form the program:

```
grandparent(A,B):- mother(A,C),mother(C,B).
grandparent(A,B):- father(A,C),mother(C,B).
grandparent(A,B):- father(A,C),father(C,B).
grandparent(A,B):- mother(A,C),father(C,B).
```

In practice, however, Metagol would not learn this program. It would induce the following program:

```
grandparent(A,B):- grandparent_1(A,C),grandparent_1(C,B).
grandparent_1(A,B):- father(A,B).
grandparent_1(A,B):- mother(A,B).
```

In this program, the symbol grandparent_1 is *invented* and corresponds to the parent relation. However, it is difficult to concisely illustrate PI in this example. We, therefore, illustrate PI in Metagol with a simpler example.

Example 11 (Predicate invention). Suppose we have the single positive example:

$$E^+ = \Big\{ \texttt{f([i,l,p],p).} \Big\}$$

Also suppose that we only have the *chain* metarule and the background relations head and tail. Given this input, in Step 2b, Metagol will try to use the *chain* metarule to prove the example. However, using only the given the BK and metarules, the only programs that Metagol can construct are combinations of the four clauses:

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

to find a generalisation of the bottom clause, Progol uses an A* algorithm to search for a generalisation in a top-down (general-to-specific) manner and uses the other examples to guide the search[21]. In this way, Progol is a top-down approach. When the search for a generalisation of the bottom clause has finished, Progol adds the clause to its hypothesis (and thus makes it more general) and removes any positive examples entailed by the new hypothesis. It repeats this process until there are no more positive examples uncovered. In Section 6.1, we discuss this approach in more detail when we describe Aleph (Srinivasan, 2001), a system similar to Progol.

4.5.4 Meta-level

A third new approach has recently emerged called *meta-level* ILP (Inoue et al., 2013; Muggleton et al., 2015; Inoue, 2016; Law et al., 2020; Cropper & Morel, 2021). There is no agreed-upon definition for what meta-level ILP means, but most approaches encode the ILP problem as a meta-level logic program, i.e. a program that reasons about programs. Such meta-level approaches often delegate the search for a hypothesis to an off-the-shelf solver (Corapi et al., 2011; Athakravi et al., 2013; Muggleton et al., 2014; Law et al., 2014; Kaminski et al., 2018; Evans et al., 2021; Cropper & Dumančić, 2020; Cropper & Morel, 2021) after which the meta-level solution is translated back to a standard solution for the ILP problem. In other words, instead of writing a procedure to search in a top-down or bottom-up manner, meta-level approaches formulate the learning problem as a declarative problem, often as an ASP problem (Corapi et al., 2011; Athakravi et al., 2013; Muggleton et al., 2014; Law et al., 2014; Kaminski et al., 2018; Evans et al., 2021; Cropper & Dumančić, 2020; Cropper & Morel, 2021). For instance, ASPAL (Corapi et al., 2011) translates an ILP problem into a meta-level ASP program which describes every example and every possible rule in the hypothesis space (defined by mode declarations). ASPAL then uses an ASP system to find a subset of the rules that cover all the positive but none of the negative examples. In other words, ASPAL delegates the search to an ASP solver. ASPAL uses an ASP optimisation statement to find the hypothesis with the fewest literals.

Meta-level approaches can often learn optimal and recursive programs. Moreover, meta-level approaches use diverse techniques and technologies. For instance, Metagol (Muggleton et al., 2015; Cropper & Muggleton, 2016) uses a Prolog meta-interpreter to search for a proof of a meta-level Prolog program. ASPAL (Corapi et al., 2011), ILASP (Law et al., 2014), HEXMIL (Kaminski et al., 2018), and the Apperception Engine (Evans et al., 2021) translate an ILP problem into an ASP problem and use powerful ASP solvers to find a model of the problem – note that these systems all employ very different algorithms. ∂ILP (Evans & Grefenstette, 2018) uses neural networks to solve the problem. Overall, the development of meta-level ILP approaches is exciting because it has diversified ILP from the standard clause refinement approach of earlier systems.

For more information about meta-level reasoning, we suggest the work of Inoue (2016), who provides an introduction to meta-level reasoning and learning. Law et al. (2020) also provide an overview of *conflict-driven* ILP, which the systems ILASP3 (Law, 2018) and Popper (Cropper & Morel, 2021) adopt.

4.5.5 Discussion

The different search methods discussed above have different advantages and disadvantages, and there is no ‘best’ approach. Moreover, as Progol illustrates, there is not necessarily clear distinctions between

[21] The A* search strategy employed by Progol can easily be replaced by alternative search algorithms, such as stochastic search (Muggleton & Tamaddoni-Nezhad, 2008).

top-down, *bottom-up*, and *meta-level* approaches. We can, however, make some general observations about the different approaches.

Bottom-up approaches can be seen as being *data-* or *example-driven*. The major advantage of these approaches is that they are typically very fast. However, as Bratko (1999) points out, there are several disadvantages of bottom-up approaches, such as (i) they typically use unnecessarily long hypotheses with many clauses, (ii) it is difficult for them to learn recursive hypotheses and multiple predicates simultaneously, and (iii) they do not easily support predicate invention.

The main advantages of top-down approaches are that they can more easily learn recursive programs and textually minimal programs. The major disadvantage is that they can be prohibitively inefficient because they can generate many hypotheses that do not cover even a single positive example. Another disadvantage of top-down approaches is their reliance on iterative improvements. For instance, TILDE keeps specialising every clause which leads to improvement (i.e., a clause covers fewer negative examples). As such, TILDE can get stuck with suboptimal solutions if the necessary clauses are very long and intermediate specialisations do not improve the score (coverage) of the clause. To avoid this issue, these systems rely on lookahead (Struyf et al., 2006) which increases the complexity of learning.

The main advantage of meta-level approaches is that they can learn recursive programs and optimal programs (Corapi et al., 2011; Law et al., 2014; Kaminski et al., 2018; Evans & Grefenstette, 2018; Evans et al., 2021; Cropper & Morel, 2021). They can also harness the state-of-the-art techniques in constraint solving, notably in ASP. However, some unresolved issues remain. A key issue is that many approaches encode an ILP problem as a single (often very large) ASP problem (Corapi et al., 2011; Law et al., 2014; Kaminski et al., 2018; Evans et al., 2021), so struggle to scale to problems with very large domains. Moreover, since most ASP solvers only work on ground programs (Gebser et al., 2014), pure ASP-based approaches are inherently restricted to tasks that have a small and finite grounding. Although preliminary work attempts to tackle this issue (Cropper & Morel, 2021; Cropper, 2022), work is still needed for these approaches to scale to very large problems. Many approaches also precompute every possible rule in a hypothesis (Corapi et al., 2011; Law et al., 2014), so struggle to learn programs with large rules, although preliminary work tries to address this issue (Cropper & Dumančić, 2020).

5. ILP Features

Table 4 compares the same systems from Table 3 on a small number of dimensions. This table excludes many other important dimensions of comparison, such as whether a system supports non-observational predicate learning, where examples of the target relations are not directly given (Muggleton, 1995). We discuss these features in turn.

5.1 Noise

Noise handling is important in ML. In ILP, we can distinguish between three types of noise:

- **Noisy examples**: where an example is misclassified

[23] A logical decision tree learned by TILDE can be translated into a logic program that contains invented predicate symbols. However, TILDE is unable to reuse any invented symbols whilst learning.

[24] LFIT does not support recursion in the rules but allows recursion in their usage. The input is a set of pairs of interpretations and the output is a logic program that can be recursively applied on its output to produce sequences of interpretations.

[25] ILASP precomputes every rule defined by a given mode declaration M to form a rule space S_M. Given background knowledge B and an example E, ILASP requires that the grounding of $B \cup S_M \cup E$ must be finite.

[26] Given background knowledge B and an example E, FastLAS requires that the grounding of $B \cup S_M \cup E$ must be finite.

$$B = \left\{ \begin{array}{l} \texttt{mother(ann,amy).~~mother(ann,andy).} \\ \texttt{mother(amy,amelia).~~mother(amy,bob).} \\ \texttt{mother(linda,gavin).} \\ \texttt{father(steve,amy).~~father(steve,andy).} \\ \texttt{father(andy,spongebob).~~father(gavin,amelia).} \end{array} \right\}$$

And the following metarules represented in Prolog:

```
metarule(ident,[P,Q], [P,A,B], [[Q,A,B]]).
metarule(chain,[P,Q,R], [P,A,B], [[Q,A,C],[R,C,B]]).
```

We can call Metagol with a lists of positive (E^+) and negative (E^-) examples:

$$E^+ = \left\{ \begin{array}{l} \texttt{grandparent(ann,amelia).} \\ \texttt{grandparent(steve,amelia).} \\ \texttt{grandparent(steve,spongebob).} \\ \texttt{grandparent(linda,amelia).} \end{array} \right\} E^- = \left\{ \texttt{grandparent(amy,amelia).} \right\}$$

In Step 1, Metagol selects an example to generalise. Suppose Metagol selects `grandparent(ann,amelia)`. In Step 2a, Metagol tries to prove this atom using the BK or an already induced clause. Since `grandparent` is not part of the BK and Metagol has not yet induced any clauses, this step fails. In Step 2b, Metagol tries to prove this atom using a metarule. Metagol can, for instance, unify the atom with the head of the *ident* metarule to form the clause:

```
grandparent(ann,amelia):- Q(ann,amelia).
```

Metagol saves a metasub for this clause:

```
sub(indent,[grandparent,Q])
```

The symbol `Q` in this metasub is still a variable. Metagol recursively tries to prove the atom `Q(ann,amelia)`. Since there is no `Q` such that `Q(ann,amelia)` is true, this step fails. Because the *ident* metarule failed, Metagol removes the metasub and backtracks to try a different metarule. Metagol unifies the atom with the *chain* metarule to form the clause:

```
grandparent(ann,amelia):- Q(ann,C),R(C,amelia).
```

Metagol saves a metasub for this clause:

```
sub(chain,[grandparent,Q,R])
```

Metagol recursively tries to prove the atoms `Q(ann,C)` and `R(C,amelia)`. Suppose the recursive call to prove `Q(ann,C)` succeeds by substituting `Q` with `mother` to form the atom `mother(ann,amy)`. This substitution binds `Q` to `mother` and `C` to `amy` which is propagated to the other atom which now becomes `R(amy,amelia)`. Metagol also proves this second atom by substituting `R` with `mother` to form the atom `mother(amy,amelia)`. The proof is now complete and the metasub is:

```
sub(chain,[grandparent,mother,mother])
```

6.4.2 Metagol Algorithm

Metagol uses the following procedure to find a hypothesis:

1. Select a positive example (an atom) to generalise. If one exists, proceed to step 2. If none exists, test the hypothesis on the negative examples. If the hypothesis does not entail any negative example stop and return the hypothesis; otherwise backtrack to a choice point at step 2 and continue.
2. Try to prove the atom by:
   (a) using given BK or an already induced clause
   (b) unifying the atom with the head of a metarule (Section 4.4.2), binding the variables in a metarule to symbols in the predicate and constant signatures, saving the substitutions, and then proving the body of the metarule through meta-interpretation (by treating the body atoms as examples and applying step 2 to them)

In other words, Metagol induces a logic program by constructing a proof of the positive examples. It uses metarules to guide the proof search. After proving all the examples, Metagol checks the consistency of the hypothesis against the negative examples. If the program is inconsistent, Metagol backtracks to explore different proofs (hypotheses).

Metarules. Metarules are fundamental to Metagol. For instance, the *chain* metarule is `P(A,B):- Q(A,C), R(C,B)`. The letters P, Q, and R denote second-order variables. Metagol internally represents metarules as Prolog atoms of the form `metarule(Name,Subs,Head,Body)`. Here Name denotes the metarule name, Subs is a list of variables that Metagol should find substitutions for, and Head and Body are list representations of a clause. For example, the internal representation of the *chain* metarule is `metarule(chain,[P,Q,R], [P,A,B], [[Q,A,C],[R,C,B]])`. Metagol represents substitutions, which we will call *metasubs*, as atoms of the form `sub(Name,Subs)`, where Name is the name of the metarule and Subs is a list of substitutions. For instance, binding the variables P, Q, and R with `second`, `tail`, and `head` respectively in the chain metarule leads to the metasub `sub(chain,[second,tail,head])` and the clause `second(A,B):- tail(A,C),head(C,B)`.

Optimality. To learn optimal programs, Metagol enforces a bound on the program size (the number of metasubs). Metagol uses iterative deepening to search for hypotheses. At depth $d = 1$ Metagol is allowed to induce at most one clause, i.e. use at most one metasub. If such a hypothesis exists, Metagol returns it. Otherwise, Metagol continues to the next iteration $d + 1$. At each iteration d, Metagol introduces $d - 1$ new predicate symbols and is allowed to use d clauses. New predicates symbols are formed by taking the name of the task and adding underscores and numbers. For example, if the task is `f` and the depth is 4 then Metagol will add the predicate symbols `f_1`, `f_2`, and `f_3` to the predicate signature.

6.4.3 Metagol Example

Example 10 (Kinship example)**.** To illustrate Metagol, suppose you have the following BK:

System	Noise	Optimality	Infinite domains	Recursion	Predicate invention
FOIL (Quinlan, 1990)	Yes	No	Yes	Partly	No
Progol (Muggleton, 1995)	Yes	No	Yes	Partly	No
Claudien (De Raedt & Dehaspe, 1997)	Yes	No	Yes	Party	No
TILDE (Blockeel & De Raedt, 1998)	Yes	No	Yes	No	No[22]
Aleph (Srinivasan, 2001)	Yes	No	Yes	Partly	No
XHAIL (Ray, 2009)	Yes	No	Yes	Partly	No
ASPAL (Corapi et al., 2011)	No	Yes	No	Yes	No
Atom (Ahlgren & Yuen, 2013)	Yes	No	Yes	Partly	No
QuickFOIL (Zeng et al., 2014)	Yes	No	No	Partly	No
LFIT (Inoue et al., 2014)	No	Yes	No	No[23]	No
ILASP (Law et al., 2014)	Yes[24]	Yes	Partly[25]	Yes	Partly
Metagol (Muggleton et al., 2015)	No	Yes	Yes	Yes	Yes
∂ILP (Evans & Grefenstette, 2018)	Yes	Yes	No	Yes	Partly
HEXMIL (Kaminski et al., 2018)	No	Yes	No	Yes	Yes
FastLAS (Law et al., 2020)	Yes	Yes	Partly[26]	No	No
Apperception (Evans et al., 2021)	Yes	Yes	No	Yes	Partly
Popper (Cropper & Morel, 2021)	Yes	Yes	Yes	Yes	Yes

Table 4: A vastly simplified comparison of ILP systems. As with Table 3, this table is meant to provide a very high-level overview of some systems. Therefore, the table entries are coarse and should not be taken absolutely literally. For instance, Metagol does not support noise and thus has the value *no* in the noise column, but there is an extension (Muggleton et al., 2018) that samples examples to mitigate the issue of misclassified examples. ILASP and ∂ILP support predicate invention, but only a restricted form. See Section 5.5 for an explanation. FOIL, Progol, and XHAIL can learn recursive programs when given sufficient examples. See Section 5.4 for an explanation.

- **Incorrect BK**: where a relation holds when it should not (or does not hold when it should)
- **Imperfect BK**: where relations are missing or there are too many irrelevant relations

We discuss these three types of noise.

Noisy examples. The problem definitions from Section 3 are too strong to account for noisy (incorrectly labelled) examples because they expect a hypothesis that entails all of the positive and none of the negative examples. Therefore, most systems relax this constraint and accept a hypothesis that does not necessarily cover all positive examples or that covers some negative examples[27]. Most systems that use a set covering loop naturally support noise handling. For instance, TILDE extends a decision tree learner (Quinlan, 1986, 1993) to the first-order setting and uses the same information gain methods to induce hypotheses. The noise-tolerant version of ILASP (Law, 2018) uses ASP's optimisation abilities to provably

[27] It is, unfortunately, a common misconception that ILP cannot handle mislabelled examples (Evans & Grefenstette, 2018).

learn the program with the best coverage. In general, handling noisy examples is a well-studied topic in ILP.

Noisy BK. Just as training examples can potentially be noisy/misclassified, the facts/rules in the BK can be noisy/misclassified. For instance, if learning rules to forecast the weather, the BK might include facts about historical weather, which might not be 100% correct. However, most systems assume that the BK is perfect, i.e. that atoms are true or false, and there is no room for uncertainty. This assumption is a major limitation because real-world data, such as images or speech, cannot always be easily translated into a purely noise-free symbolic representation. We discuss this limitation in more detail in Section 9.1. One of the key appealing features of ∂ILP is that it takes a differentiable approach to ILP and can be given fuzzy or ambiguous data. Rather than an atom being true or false, ∂ILP gives atoms continuous semantics, which maps atoms to the real unit interval [0, 1]. The authors successfully demonstrate the approach on the MNIST dataset.

Imperfect BK. Handling imperfect BK is an under-explored topic in ILP. We can distinguish between two types of imperfect BK: missing BK and too much BK, which we discussed in Section 4.3.2.

5.2 Optimality

There are often multiple (sometimes infinite) hypotheses that solve the ILP problem (or have the same training error). In such cases, which hypothesis should we choose?

5.2.1 Occamist Bias

Many systems try to learn a textually minimal hypothesis. This approach is justified as following an *Occamist bias* (Schaffer, 1993). The most common interpretation of an Occamist bias is that amongst all hypotheses consistent with the data, the simplest is the most likely[28]. Most approaches use an Occamist bias to find the smallest hypothesis, measured in terms of the number of clauses (Muggleton et al., 2015), literals (Law et al., 2014), or description length (Muggleton, 1995). Most systems are not, however, guaranteed to induce the smallest programs. A key reason for this limitation is that many approaches learn a single clause at a time leading to the construction of sub-programs that are sub-optimal in terms of program size and coverage. For instance, Aleph, described in detail in the next section, offers no guarantees about the program size and coverage. Newer systems address this limitation (Corapi et al., 2011; Law et al., 2014; Cropper & Muggleton, 2016; Kaminski et al., 2018; Cropper & Morel, 2021) through meta-level reasoning (Section 4.5). For instance, ASPAL (Corapi et al., 2011) is given as input a hypothesis space with a set of candidate clauses. The ASPAL task is to find a minimal subset of clauses that entails all the positive and none of the negative examples. ASPAL uses ASP's optimisation abilities to provably learn the program with the fewest literals.

[28] Domingos (1999) points out that this interpretation is controversial, partly because Occam's razor is interpreted in two different ways. Following Domingos (1999), let the *generalisation error* of a hypothesis be its error on unseen examples and the *training error* be its error on the examples it was learned from. The formulation of the razor that is perhaps closest to Occam's original intent is *given two hypotheses with the same generalisation error, the simpler one should be preferred because simplicity is desirable in itself*. The second formulation, which most ILP systems follow, is different and can be stated as *given two hypotheses with the same training error, the simpler one should be preferred because it is likely to have lower generalisation error*. Domingos (1999) points out that the first razor is largely uncontroversial, but the second one, taken literally, is provably and empirically false (Zahálka & Zelezný, 2011). Many systems do not distinguish between the two cases. We therefore also do not make any distinction.

```
0 {rule(r1),rule(r2,fly),rule(r2,swim),rule(r3,fly,swim)}4.
```

This statement is a choice rule, which states none or at most four of the literals {rule(r1), rule(r2,fly), rule(r2,swim), rule(r3,fly,swim)} could be true. The job of the ASP solver is to determine which of those literals should be true (formulated as an ASP optimization problem), which corresponds to an answer set for this program:

```
rule(r2,c(fly)).
```

Which is translated to a program:

```
penguin(A):- not can(A,fly).
```

6.3.4 Discussion

Advantages. A major advantage of ASPAL is its sheer simplicity, which has inspired other approaches, notably ILASP. It also learns optimal programs by employing ASP optimisation constraints.

Disadvantages. The main limitation of ASPAL is scalability. It precomputes every possible rule in a hypothesis, which is infeasible on all but trivial problems. For instance, when learning game rules from observations (Cropper et al., 2020b), ASPAL performs poorly for this reason.

6.4 Metagol

An *interpreter* is a program that evaluates (interprets) programs. A *meta-interpreter* is an interpreter written in the same language that it evaluates. Metagol (Muggleton et al., 2015; Cropper & Muggleton, 2016; Cropper et al., 2020) is a form of ILP based on a Prolog meta-interpreter.

6.4.1 Metagol Setting

The Metagol problem setting is:

Given:
- A set of metarules M
- B in the form of a normal program
- E^+ positive examples represented as a set of facts
- E^- negative examples represented as a set of facts

Return: A definite program hypothesis H such that:
- $\forall e \in E^+,\ H \cup B \models e$ (i.e. is complete)
- $\forall e \in E^-,\ H \cup B \not\models e$ (i.e. is consistent)
- $\forall h \in H,\ \exists m \in M$ such that $h = m\theta$, where θ is a substitution that grounds all the existentially quantified variables in m

The last condition ensures that a hypothesis is an instance of the given metarules. It is this condition that enforces the strong inductive bias in Metagol.

6.3.3 ASPAL Example

Example 9 (ASPAL)**.** To illustrate ASPAL, we slightly modify the example from Corapi et al. (2011). We also ignore the penalty statement. ASPAL is given as input B, E^{+}, E^{-}, and M:

$$B = \left\{ \begin{array}{l} \texttt{bird(alice).} \\ \texttt{bird(betty).} \\ \texttt{can(alice,fly).} \\ \texttt{can(betty,swim).} \\ \texttt{ability(fly).} \\ \texttt{ability(swim).} \end{array} \right\}$$

$$E^{+} = \left\{ \texttt{ penguin(betty). } \right\} E^{-} = \left\{ \texttt{ penguin(alice). } \right\}$$

$$M = \left\{ \begin{array}{l} \texttt{modeh(1, penguin(+bird)).} \\ \texttt{modeb(1, bird(+bird)).} \\ \texttt{modeb(*,not can(+bird,\#ability))} \end{array} \right\}$$

Given these modes[43] , the possible rules are:

```
penguin(X):- bird(X).
penguin(X):- bird(X), not can(X,fly).
penguin(X):- bird(X), not can(X,swim).
penguin(X):- bird(X), not can(X,swim), not can(X,fly).
```

ASPAL generates skeleton rules which replace constants with variables and adds an extra literal to each rule as an abducible literal:

```
penguin(X):- bird(X), rule(r1).
penguin(X):- bird(X), not can(X,C1), rule(r2,C1).
penguin(X):- bird(X), not can(X,C1), not can(X,C2), rule(r3,C1,C2).
```

ASPAL forms a meta-level ASP program from these rules that is passed to an ASP solver:

```
bird(alice).
bird(betty).
can(alice,fly).
can(betty,swim).
ability(fly).
ability(swim).
penguin(X):- bird(X), rule(r1).
penguin(X):- bird(X), not can(X,C1), rule(r2,C1).
penguin(X):- bird(X), not can(X,C1), not can(X,C2), rule(r3,C1,C2).
0 {rule(r1),rule(r2,fly),rule(r2,swim),rule(r3,fly,swim)}4.
goal : - penguin(betty), not penguin(alice).
: - not goal.
```

The key statement in this meta-level program is:

[43] Note that 'notcan' is used in the original ASPAL paper but we think this usage is a typo.

5.2.2 Cost-minimal Programs

Learning efficient logic programs has long been considered a difficult problem (Muggleton & De Raedt, 1994; Muggleton et al., 2012), mainly because there is no declarative difference between an efficient program, such as mergesort, and an inefficient program, such as bubble sort. To address this issue, Metaopt (Cropper & Muggleton, 2019) learns efficient programs. Metaopt maintains a cost during the hypothesis search and uses this cost to prune the hypothesis space. To learn minimal time complexity logic programs, Metaopt minimises the number of resolution steps. For instance, imagine learning a *find duplicate* program, which finds a duplicate element in a list e.g. `[p,r,o,g,r,a,m]` ↦ `r`, and `[i,n,d,u,c,t,i,o,n]` ↦ `i`. Given suitable input data, Metagol induces the program:

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

This program goes through the elements of the list checking whether the same element exists in the rest of the list. Given the same input, Metaopt induces the program:

```
f(A,B):- mergesort(A,C),f1(C,B).
f1(A,B):- head(A,B),tail(A,C),head(C,B).
f1(A,B):- tail(A,C),f1(C,B).
```

This program first sorts the input list and then goes through the list to check for duplicate adjacent elements. Although larger, both in terms of clauses and literals, the program learned by Metaopt is more efficient ($O(n \log n)$) than the program learned by Metagol ($O(n^2)$).

Other systems can also learn optimal programs (Schüller & Benz, 2018). For instance, FastLAS (Law et al., 2020) follows this idea and takes as input a custom scoring function and computes an optimal solution for the given scoring function. The authors show that this approach allows a user to optimise domain-specific performance metrics on real-world datasets, such as access control policies.

5.3 Infinite Domains

Some systems, mostly meta-level approaches, cannot handle infinite domains (Corapi et al., 2011; Athakravi et al., 2013; Evans & Grefenstette, 2018; Kaminski et al., 2018; Evans et al., 2021). Pure ASP-based systems (Corapi et al., 2011; Kaminski et al., 2018; Evans et al., 2021) struggle to handle infinite domains because (most) current ASP solvers only work on ground programs, i.e. they need a finite grounding. ASP can work with infinite domains as long as the grounding is finite. This finite grounding restriction is often achieved by enforcing syntactic restrictions on the programs, such as finitely-ground programs Calimeri et al. (2008). ASP systems (which combine a grounder and a solver), such as Clingo (Gebser et al., 2014), first take a first-order program as input, ground it using an ASP grounder, and then use an ASP solver to determine whether the ground problem is satisfiable. This approach leads to the *grounding bottleneck* problem (Balduccini et al., 2013), where grounding can be so large that it is simply intractable. This grounding issue is especially problematic when reasoning about complex data structures, such as lists. For instance, grounding the permutation relation over the ASCII characters would require 128! facts. The grounding bottleneck is especially a problem when reasoning about real numbers[29]. For instance, ILASP (Law et al., 2014) can represent real numbers as strings and can delegate reasoning to Python (via Clingo's

[29] Most ASP implementations do not natively support lists nor real numbers, although both can be represented using other means.

scripting feature). However, in this approach, the numeric computation is performed when grounding the inputs, so the grounding must be finite, which makes it impractical. This grounding problem is not specific to ASP-based systems. For instance, ∂ILP is an ILP system based on a neural network, but it only works on BK in the form of a finite set of ground atoms. This grounding problem is essentially the fundamental problem faced by table-based ML approaches that we discussed in Section 4.3.

One approach to mitigate this problem is to use *context-dependent examples* (Law et al., 2016), where BK can be associated with specific examples so that an ILP systems need only ground part of the BK. Although this approach is shown to improve the grounding problem compared to not using context-dependent examples, the approach still needs a finite grounding for each example and still struggles as the domain size increases (Cropper & Morel, 2021).

5.4 Recursion

The power of recursion is that an infinite number of computations can be described by a finite recursive program (Wirth, 1985). In ILP, recursion is often crucial for generalisation. We illustrate this importance with two examples.

Example 5 (Reachability). Consider learning the concept of *reachability* in a graph. Without recursion, an ILP system would need to learn a separate clause to define reachability of different lengths. For instance, to define reachability depths for 1-4 would require the program:

```
reachable(A,B):- edge(A,B).
reachable(A,B):- edge(A,C),edge(C,B).
reachable(A,B):- edge(A,C),edge(C,D),edge(D,B).
reachable(A,B):- edge(A,C),edge(C,D),edge(D,E),edge(E,B).
```

This program does not generalise because it does not define reachability for arbitrary depths. Moreover, most systems would need examples of each depth to learn such a program. By contrast, a system that supports recursion can learn the program:

```
reachable(A,B):- edge(A,B).
reachable(A,B):- edge(A,C),reachable(C,B).
```

Although smaller, this program generalises reachability to any depth. Moreover, systems can learn this definition from a small number of examples of arbitrary reachability depth.

Example 6 (String transformations). Reconsider the string transformation problem from the introduction (Section 1.2). As with the reachability example, without recursion, a system would need to learn a separate clause to find the last element for each list of length *n*, such as:

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

By contrast, a system that supports recursion can learn the compact program:

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

However, this candidate clause would be rejected as it yields no information gain (every example covered by the first clause is also covered by the second clause). The introduction of a literal with the `number_of_components` predicate is only helpful if it is introduced together with the inequality related to the second argument of the literal. Informing TILDE about this dependency is known as lookahead.

6.3 ASPAL

ASPAL (Corapi et al., 2011) was one of the first meta-level ILP systems, which directly influenced other ILP systems, notably ILASP. ASPAL builds on TAL (Corapi et al., 2010), but is simpler to explain[42]. Indeed, ASPAL is one of the simplest ILP systems to explain. It uses the mode declarations to build every possible clause that could be in a hypothesis. It adds a flag to each clause indicating whether the clause should be in a hypothesis. It then formulates the problem of deciding which flags to turn on as an ASP problem.

6.3.1 ASPAL Setting

The ASPAL problem setting is:

Given:
- A set of mode declarations M
- B in the form of a normal program
- E^+ positive examples represented as a set of facts
- E^- negative examples represented as a set of facts
- A penalty function γ

Return: A normal program hypothesis H such that:
- H is consistent with M
- $\forall e \in E^+,\ H \cup B \models e$ (i.e. is complete)
- $\forall e \in E^-,\ H \cup B \not\models e$ (i.e. is consistent)
- The penalty function γ is minimal

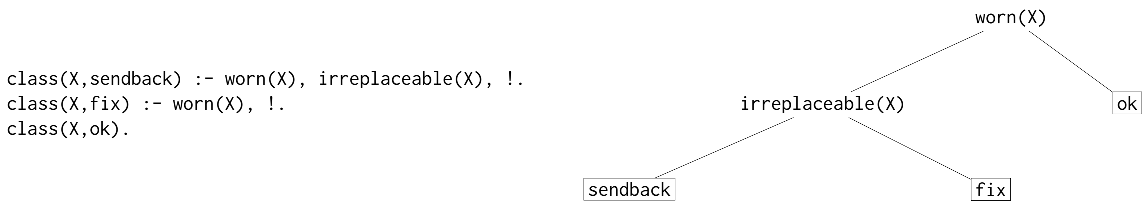
6.3.2 ASPAL Algorithm

ASPAL encodes an ILP problem as a meta-level ASP program. The answer sets of this meta-level program are solutions to the ILP problem. The ASPAL algorithm is one of the simplest in ILP:

1. Generate all possible rules consistent with the given mode declarations. Assign each rule a unique identifier and add that as an abducible (guessable) atom in each rule.
2. Use an ASP solver to find a minimal subset of the rules (by formulating the problem as an ASP optimization problem).

Step 1 is a little more involved, and we explain why below. Also, similar to Aleph, ASPAL has several input parameters that constrain the size of the hypothesis space, such as the maximum number of body literals and the maximum number of clauses. Step 2 uses an ASP optimisation statement to learn a program with a minimal penalty.

[42] A key difference is that TAL is implemented in Prolog and uses list structure to iteratively build rules. By contrast, using list-like structure in ASP is inefficient and often impossible as the solver completely grounds the program before solving it.



```
class(X,sendback) :- worn(X), irreplaceable(X), !.
class(X,fix) :- worn(X), !.
class(X,ok).
```

Figure 6: TILDE learns tree-shaped (normal) programs. Clauses in the program correspond to paths along the tree.

The candidate refinement `worn(X), irreplaceable(X)` perfectly divides the remaining examples and thus `irreplaceable(X)` is added as the subsequent test. All examples are classified correctly, and thus the learning stops.

The final TILDE tree is (illustrated in Figure 6):

```
class(sendback):- worn(X),irreplaceable(X),!.
class(fix):- worn(X),!.
class(ok).
```

Note the usage of the cut (`!`) operator, which is essential to ensure that only one branch of the decision tree holds for each example.

6.2.3 Discussion

Advantages. An interesting aspect of TILDE is that it learns normal logic programs (which include negation) instead of definite logic programs. Another advantage of TILDE is that, compared to other ILP systems, it supports both categorical and numerical data. Indeed, TILDE is an exception among ILP systems, which usually struggle to handle numerical data. At any refinement step, TILDE can add a literal of the form `<(X,V)`, or equivalently `X < V` with `V` being a value. TILDE's stepwise refinement keeps the number of inequality tests tractable.

Disadvantages. Although TILDE learns normal programs, it requires them to be in the shape of a tree and does not support recursion. Furthermore, TILDE inherits the limitations of top-down systems, such as generating many needless candidates. Another weakness of TILDE is the need for lookahead. Lookahead is needed when a single literal is useful only in a conjunction with another literal. Consider, for instance, that the machine repair scenario has a relation `number_of_components` and the target rule that a machine needs to be fixed when a part consisting of more than three parts is worn out:

```
class(fix):- worn(X),number_of_components(X,Y),Y > 3.
```

To find this clause, TILDE would first refine the clause:

```
class(fix):- worn(X).
```

into:

```
class(fix):- worn(X),number_of_components(X,Y).
```

Because of the symbolic representation and the recursive nature, this program generalises to lists of arbitrary length and which contain arbitrary elements (e.g. integers and characters).

Without recursion it is often difficult for a system to generalise from small numbers of examples (Cropper et al., 2015). Moreover, recursion is vital for many program synthesis tasks, such as the quicksort scenario from the introduction. Despite its importance, learning recursive programs has long been a difficult problem (Muggleton et al., 2012). Moreover, there are many negative theoretical results on the learnability of recursive programs (Cohen, 1995d). As Table 4 shows, many systems cannot learn recursive programs, or can only learn it in a limited form.

A common limitation is that many systems rely on *bottom clause construction* (Muggleton, 1995), which we discuss in detail in Section 6.1. In this approach, for each positive example, a system creates the most specific clause that entails the example and then tries to generalise the clause to entail other examples. However, because a system learns only a single clause per example[30], this covering approach requires examples of both the base and inductive cases, which means that such systems struggle to learn recursive programs, especially from small numbers of examples.

Interest in recursion has resurged recently with the introduction of meta-interpretive learning (MIL) (Muggleton et al., 2014, 2015; Cropper et al., 2020) and the MIL system Metagol (Cropper & Muggleton, 2016). The key idea of MIL is to use metarules (Section 4.4.2) to restrict the form of inducible programs and thus the hypothesis space. For instance, the *chain* metarule ($P(A, B) \leftarrow Q(A, C), R(C, B)$) allows Metagol to induce programs[31] such as:

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

Metagol induces recursive programs using recursive metarules, such as the *tail recursive* metarule *P(A,B) ← Q(A,C), P(C,B)*. Metagol can also learn mutually recursive programs, such as learning the definition of an even number by also inventing and learning the definition of an odd number (even_1):

```
even(0).
even(A):- successor(A,B),even_1(B).
even_1(A):- successor(A,B),even(B).
```

Many systems can now learn recursive programs (Law et al., 2014; Evans & Grefenstette, 2018; Kaminski et al., 2018; Evans et al., 2021; Cropper & Morel, 2021). With recursion, systems can generalise from small numbers of examples, often a single example (Lin et al., 2014; Cropper, 2019). For instance, Popper (Cropper & Morel, 2021) can learn list transformation programs from only a handful of examples, such as a program to drop the last element of a list:

```
droplast(A,B):- tail(A,B),empty(B).
droplast(A,B):- tail(A,C),droplast(C,D),head(A,E),cons(E,D,B).
```

The ability to learn recursive programs has opened ILP to new application areas, including learning string transformations programs (Lin et al., 2014), robot strategies (Cropper & Muggleton, 2015), context-free grammars (Muggleton et al., 2014), and answer set grammars (Law et al., 2019).

[30] This statement is not true for all systems that employ bottom clause construction. XHAIL (Ray, 2009), for instance, can induce multiple clauses per example.

[31] Metagol can induce longer clauses though predicate invention, which we discuss in Section 5.5.

5.5 Predicate Invention

Most systems assume that the given BK is suitable to induce a solution. This assumption may not always hold. Rather than expecting a user to provide all the necessary BK, the goal of *predicate invention* (PI) is for a system to automatically invent new auxiliary predicate symbols, i.e. to introduce new predicate symbols in a hypothesis that are not given the examples nor the BK[32] This idea is similar to when humans create new functions when manually writing programs, such as to reduce code duplication or to improve readability. For instance, to learn the quicksort algorithm, a learner needs to be able to partition the list given a pivot element and append two lists. If `partition` and append are not provided in BK, the learner would need to invent them.

PI has repeatedly been stated as an important challenge (Muggleton & Buntine, 1988; Stahl, 1995; Muggleton, 1994b; Muggleton et al., 2012). Russell (2019) even argues that the automatic invention of new high-level concepts is the most important step needed to reach human-level AI. A classical example of PI is learning the definition of `grandparent` from only the background relations `mother` and `father`. Given suitable examples and no other background relations, a system can learn the program:

```
grandparent(A,B):- mother(A,C),mother(C,B).
grandparent(A,B):- mother(A,C),father(C,B).
grandparent(A,B):- father(A,C),mother(C,B).
grandparent(A,B):- father(A,C),father(C,B).
```

Although correct, this program is large and has 4 clauses and 12 literals. By contrast, consider the program learned by a system which supports PI:

```
grandparent(A,B):- inv(A,C),inv(C,B).
inv(A,B):- mother(A,B).
inv(A,B):- father(A,B).
```

To learn this program, a system has *invented* a new predicate symbol `inv`. This program is semantically equivalent[33] to the previous one, but is shorter both in terms of the number of literals and clauses. The invented symbol `inv` can be interpreted as *parent*. In other words, if we rename `inv` to `parent` we have the program:

```
grandparent(A,B):- parent(A,C),parent(C,B).
parent(A,B):- mother(A,B).
parent(A,B):- father(A,B).
```

As this example shows, PI can help learn smaller programs, which, in general, is preferable because most systems struggle to learn large programs (Cropper et al., 2020b; Cropper & Dumančić, 2020).
PI has been shown to help reduce the size of programs, which in turn reduces sample complexity and improves predictive accuracy (Dumančić & Blockeel, 2017; Cropper, 2019; Cropper et al., 2020; Dumančić et al., 2019; Dumancic et al., 2021).

[32] PI is a form of *non-observational predicate learning*, where examples of the target relations are not directly given (Muggleton, 1995).

[33] This use of the term *semantically equivalent* is imprecise. Whether these two programs are strictly equivalent depends on the definition of logical equivalence, for which there are many (Maher, 1988). Moreover, equivalence between the two programs is further complicated because they have different vocabularies (because of the invented predicate symbol). Our use of equivalence is based on the two programs having the same logical consequences for the target predicate symbol `grandparent`.

machine with parts that are worn out, and (ii) an action an engineer should perform: `fix` the machine, `send it` back to the manufacturer, or nothing if the machine is ok. These actions are the classes to predict.

$$E = \left\{ \begin{array}{l} \texttt{E1: \{worn(gear). worn(chain). class(fix).\}} \\ \texttt{E2: \{worn(engine). worn(chain). class(sendback).\}} \\ \texttt{E3: \{worn(wheel). class(sendback).\}} \\ \texttt{E4: \{class(ok).\}} \end{array} \right\}$$

Background knowledge contains information which parts are replaceable and which are not:

$$B = \left\{ \begin{array}{l} \texttt{replaceable(gear).} \\ \texttt{replaceable(chain).} \\ \texttt{irreplaceable(engine).} \\ \texttt{irreplaceable(wheel).} \end{array} \right\}$$

Like any top-down approach, TILDE starts with the most general program; in this case, the initial program assigns the majority class to all examples. TILDE then gradually refines the program (specialises it) until satisfactory performance is reached. To refine the program, TILDE uses mode declarations. Due to the top-down nature of TILDE, it is more natural to understand modes as conjunctions that can be added to a current clause. This interpretation does not conflict the explanation given in Section 4.4. TILDE interprets an *input* argument as bounding to a variable that already exists in the current clause; this is identical to stating that the argument needs to be instantiated when literal is called as it will be instantiated by the existing literal that introduces that variable. Similarly, in TILDE, an *output* argument introduces a new variable; this variable will be instantiated after the literal is called.

Assume the mode declarations:

```
modeb(*,replaceable(+X)).
modeb(*,irreplaceable(+X)).
modeb(*,worn(+X)).
```

Each mode declaration forms a candidate split:

```
worn(X).
replaceable(X).
irreplaceable(X).
```

Computing the information gain for all of the candidate splits (as with propositional C4.5), the conjunction `worn(X)` results in the highest gain and is set as the root of the tree. For details about the C4.5 and information gain, we refer the reader to the excellent machine learning book by Mitchell (1997).

TILDE proceeds by recursively repeating the same procedure over both outcomes of the test: when `worn(X)` is `true` and `false`. When the root test fails, the dataset contains a single example (E4); TILDE forms a branch by creating the leaf predicting the class ok. When the root test succeeds, not all examples (`E1, E2, E3`) belong to the same class. TILDE thus refines the root node further:

```
worn(X), worn(X).                  worn(X), worn(Y).
worn(X), replaceable(X).           worn(X), replaceable(Y).
worn(X), irreplaceable(X)          worn(X), irreplaceable(Y)
```

6.2 TILDE

TILDE (Blockeel & De Raedt, 1998) is a first-order generalisation of decision trees, and specifically the C4.5 (Quinlan, 1993) learning algorithm. TILDE learns from interpretations, instead of entailment as Aleph, and is an instance of top-down methodology.

The learning setup of TILDE is different from other ILP systems in the sense that it closely mimics the standard classification setup from machine learning. That is, TILDE learns a program that assigns a class to an example (an interpretation). Assigning a class is done by checking which class an example entails, given the program and background knowledge. Class assignment is indicated as a special literal of the form `class(c)`, where c comes from a set of classes C. For instance, if a classifier differentiates between cats and dog then $C = \{cat, dog\}$.

6.2.1 TILDE Setting

The TILDE problem setting is:

Given:

- A set of classes C
- A set of mode declarations
- A set of examples E (a set of interpretations)
- BK in the form of a definite program

Return: A (normal) logic program hypothesis H such that:

- $\forall e \in E,\ H \wedge BK \wedge e \models class(c), c \in C$, where c is the class of the example e
- $\forall e \in E,\ H \wedge BK \wedge e \not\models class(c'), c' \in C - \{c\}$

6.2.2 TILDE Algorithm

TILDE behaves almost the same as C4.5 limited to binary attributes, meaning that it uses the same heuristics and pruning techniques. What TILDE does differently is the generation of candidates splits. Whereas C4.5 generates candidates as attribute-value pairs (or value inequalities in case of continuous attributes), TILDE uses conjunctions of literals. The conjunctions are explored gradually from the most general to the most specific ones, where θ-subsumption (Section 2) is used as an ordering.

To find a hypothesis, TILDE employs a divide-and-conquer strategy recursively repeating the following steps:

- if all examples belong to the same class, create a leaf predicting that class
- for each candidate conjunction $conj$, find the normalised information gain when splitting on $conj$
  - if no candidate provides information gain, turn the previous node into a leaf predicting the majority class
- create a decision node n that splits on the candidate conjunction with the highest information gain
- Recursively split on the subsets of data obtained by the splits and add those nodes as children of n

Example 8 (Machine repair example (Blockeel & De Raedt, 1998)). To illustrate TILDE's learning procedure, consider the following example. Each example is an interpretation (a set of facts) and it describes (i) a

To further illustrate the power of PI, imagine learning a droplasts program, which removes the last element of each sublist in a list, e.g. [alice,bob,carol] ↦ [alic,bo,caro]. Given suitable examples and BK, Metagol$_{ho}$ (Cropper et al., 2020) learns the higher-order program:

```
droplasts(A,B):- map(A,B,droplasts1).
droplasts1(A,B):- reverse(A,C),tail(C,D),reverse(D,B).
```

To learn this program, Metagol$_{ho}$ invents the predicate symbol droplasts1, which is used twice in the program: once as term in the literal map(A,B,droplasts1) and once as a predicate symbol in the literal droplasts1(A,B). This higher-order program uses map to abstract away the manipulation of the list to avoid the need to learn an explicitly recursive program (recursion is implicit in map).

Now consider learning a *double droplasts* program (ddroplasts), which extends the droplast problem so that, in addition to dropping the last element from each sublist, it also drops the last sublist, e.g. [alice,bob,carol] ↦ [alic,bo]. Given suitable examples, metarules, and BK, Metagol$_{ho}$ learns the program:

```
ddroplasts(A,B):- map(A,C,ddroplasts1),ddroplasts1(C,B).
ddroplasts1(A,B):- reverse(A,C),tail(C,D),reverse(D,B).
```

This program is similar to the aforementioned droplasts program, but additionally reuses the invented predicate symbol ddroplasts1 in the literal ddroplasts1(C,B). This program illustrates the power of PI to help learn substantially more complex programs.

Most early attempts at PI were unsuccessful, and, as Table 4 shows, most systems do not support it. As Kramer (1995) points out, PI is difficult for at least three reasons:

- When should we invent a new symbol? There must be a reason to invent a new symbol; otherwise, we would never invent one.
- How should you invent a new symbol? How many arguments should it have?
- How do we judge the quality of a new symbol? When should we keep an invented symbol?

There are many PI techniques. We briefly discuss some approaches now.

5.5.1 Inverse Resolution

Early work on PI was based on the idea of inverse resolution (Muggleton & Buntine, 1988) and specifically *W operators*. Discussing inverse resolution in depth is beyond the scope of this paper. We refer the reader to the original work of Muggleton and Buntine (1988) or the overview books by Nienhuys-Cheng and Wolf (1997) and De Raedt (2008) for more information. Although inverse resolution approaches could support PI, they never demonstrated completeness, partly because of the lack of a declarative bias to delimit the hypothesis space (Muggleton et al., 2015).

5.5.2 Placeholders

One approach to PI is to predefine invented symbols through mode declarations, which Leban et al. (2008) call *placeholders* and which Law (2018) calls *prescriptive PI*. For instance, to invent the parent relation, a suitable modeh declaration would be required, such as:

```
modeh(1,inv(person,person)).
```

However, this placeholder approach is limited because it requires that a user manually specify the arity and argument types of a symbol (Law et al., 2014), which rather defeats the point, or requires generating all possible invented predicates (Evans & Grefenstette, 2018; Evans et al., 2021), which is computationally expensive.

5.5.3 Metarules

To reduce the complexity of PI, Metagol uses metarules (Section 4.4.2) to define the hypothesis space. For instance, the *chain* metarule ($P(A,B) \leftarrow Q(A,C), R(C,B)$) allows Metagol to induce programs such as:

```
f(A,B):- tail(A,C),tail(C,B).
```

This program drops the first two elements from a list. To induce longer clauses, such as to drop first three elements from a list, Metagol can use the same metarule but can invent a new predicate symbol and then chain their application, such as to induce the program[34]:

```
f(A,B):- tail(A,C),inv(C,B).
inv(A,B):- tail(A,C),tail(C,B).
```

A side-effect of this metarule-driven approach to PI is that problems are forced to be decomposed into smaller problems. For instance, suppose you want to learn a program that drops the first four elements of a list, then Metagol could learn the following program, where the invented predicate symbol inv is used twice:

```
f(A,B):- inv(A,C),inv(C,B).
inv(A,B):- tail(A,C),tail(C,B).
```

To learn this program, Metagol invents the predicate symbol inv and induces a definition for it using the *chain* metarule. Metagol uses this new predicate symbol in the definition for the target predicate f.

5.5.4 Lifelong Learning

The aforementioned techniques for PI are aimed at single-task problems. PI can be performed by continually learning programs (meta-learning). For instance Lin et al. (2014) use a technique called *dependent learning* to enable Metagol to learn string transformations programs over time. Given a set of 17 string transformation tasks, their learner automatically identifies easier problems, learn programs for them, and then reuses the learned programs to help learn programs for more difficult problems. To determine which problems are easier to solve, the authors initially start with a very strong bias in the form of allowing a learner to only use one rule to find a solution. They then progressively relax this restriction, each time allowing more rules in a solution. The authors use PI to reform the bias of the learner where after a solution is learned not only is the target predicate added to the BK but also its constituent invented predicates. The authors experimentally show that their multi-task approach performs substantially better than a single-task approach because learned programs are frequently reused. Moreover, they show that this approach leads to a hierarchy of BK composed of reusable programs, where each builds on simpler programs. Figure 4 shows this approach. Note that this lifelong setting raises challenges, which we discuss in Section 9.1.

[34] We could unfold (Tamaki & Sato, 1984) this program to remove the invented symbol to derive the program f(A,B):- tail(A,C),tail(C,D),tail(D,B).

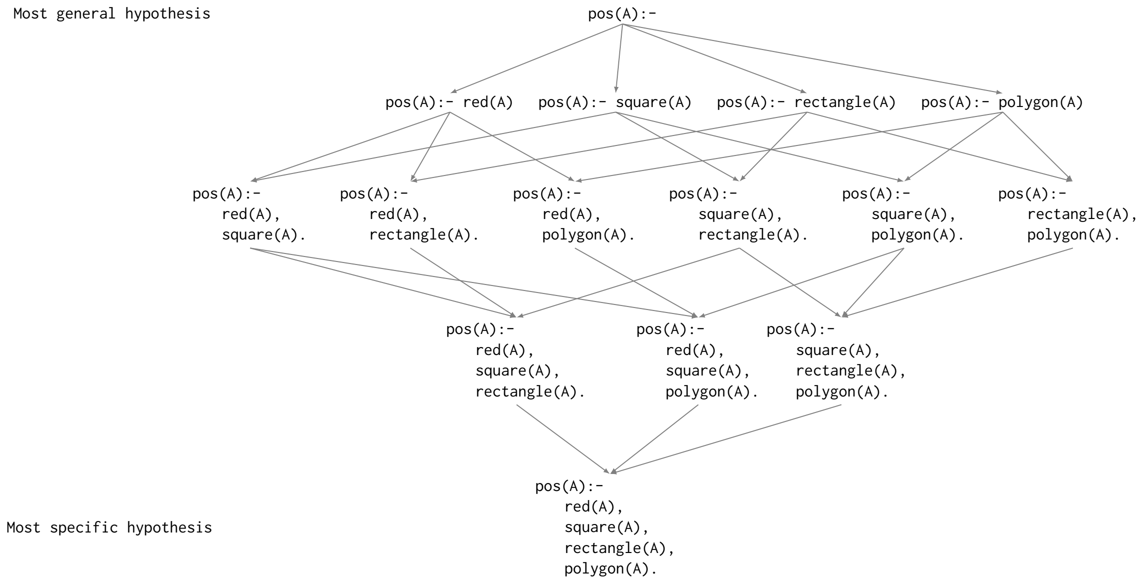

Most general hypothesis
pos(A):-
pos(A):- red(A)
pos(A):- square(A)
pos(A):- rectangle(A)
pos(A):- polygon(A)
pos(A):- red(A), square(A).
pos(A):- red(A), rectangle(A).
pos(A):- red(A), polygon(A).
pos(A):- square(A), rectangle(A).
pos(A):- square(A), polygon(A).
pos(A):- rectangle(A), polygon(A).
pos(A):- red(A), square(A), rectangle(A).
pos(A):- red(A), square(A), polygon(A).
pos(A):- square(A), rectangle(A), polygon(A).
Most specific hypothesis
pos(A):- red(A), square(A), rectangle(A), polygon(A).

Figure 5: Aleph bounds the hypothesis space from above (the most general hypothesis) and below (the most specific hypothesis). Aleph starts the search from the most general hypothesis and specialises it (by adding literals from the bottom clause) until it finds the best hypothesis.

6.1.3 Discussion

Advantages. Aleph is one of the most popular ILP systems because (i) it has a stable and easily available implementation with many options, and (ii) it has good empirical performance. Moreover, it is a single Prolog file, which makes it easy to download and use[40]. Because it uses a bottom clause to bound the search, Aleph is also efficient at identifying relevant constant symbols that may appear in a hypothesis, which is not the case for pure top-down approaches[41]. Aleph also supports many other features, such as numerical reasoning, inducing constraints, and allowing user-supplied cost functions.

Disadvantages. Because it is based on inverse entailment, and only learns a single clause at a time, Aleph struggles to learn recursive programs and optimal programs and does not support PI. Aleph also uses many parameters, such as parameters that change the search strategy when generalising a bottom clause (step 3) and parameters that change the structure of learnable programs (such as limiting the number of literals in the bottom clause). These parameters can greatly influence learning performance. Even for experts, it is non-trivial to find a suitable set of parameters for a problem.

[40] Courtesy of Fabrizio Riguzzi and Paolo Niccolò Giubelli, Aleph is now available as a SWIPL package at https://www.swi-prolog.org/pack/list?p=aleph

[41] As the Aleph manual states, "the bottom clause is really useful to introduce constants (these are obtained from the seed example".

Let B be the BK:

$$B = \left\{ \begin{array}{l} \texttt{red(s1).} \\ \texttt{blue(s2).} \\ \texttt{square(s1).} \\ \texttt{triangle(s2).} \\ \texttt{polygon(A):- rectangle(A).} \\ \texttt{rectangle(A):- square(A).} \end{array} \right\}$$

Let e be the positive example pos(s1). Then:

$$\bot\texttt{(e) = pos(A):- red(A),square(A),rectangle(A),polygon(A).}$$

This bottom clause contains the literal rectangle(A) because it is implied by square(A). The inclusion of rectangle(A) in turn implies the inclusion of polygon(A). Although blue and triangle appear in B, they are irrelevant to e, so do not appear in the bottom clause.

Any clause that is not more general than the bottom clause cannot entail e so can be ignored. For instance, we can ignore the clause pos(A):- blue(A) because it is not more general than $\bot(e)$.

Constant symbols. Note that $\bot(e)$ contains variables, rather than constant symbols, which would make the bottom clause even more specific. The reason is that the given mode declarations forbid constant symbols. Had modeb(*,polygon(#shape)) been given in M, then $\bot(e)$ would also contain polygon(s1).

Step 3: Clause Search

Having constructed a bottom clause, Aleph searches for generalisations of it. The importance of constructing the bottom clause is that it bounds the search space from below (the bottom clause). Figure 5 illustrates the search space of Aleph when given the bottom clause ⊥(e) from our previous shape example. Aleph performs a bounded breadth-first search to enumerate shorter clauses before longer ones, although a user can easily change the search strategy[38]. The search is bounded by several parameters, such as a maximum clause size and a maximum proof depth. In this scenario, Aleph starts with the most general generalisation of ⊥(e), which is *pos(A)*, which simply says that everything is true. Aleph evaluates (assigns a score) to each clause in the search, i.e. each clause in the lattice. Aleph's default evaluation function is *coverage* defined as $P - N$, where P and N are the numbers of positive and negative examples respectively entailed by the clause[39]. Aleph then tries to specialise a clause by adding literals to the body of it, which it selects from the bottom clause or by instantiating variables. Each specialisation of a clause is called a *refinement*. Properties of refinement operators (Shapiro, 1983) are well-studied in ILP (Nienhuys-Cheng & Wolf, 1997; De Raedt, 2008), but are beyond the scope of this paper. The key thing to understand is that Aleph's search is bounded from above (the most general clause) and below (the most specific clause). Having found the best clause, Aleph adds it to the hypothesis, removes all the positive examples covered by the new hypothesis, and returns to Step 1. Describing the full clause search mechanism and how the score is computed is beyond the scope of this work, so we refer to the reader to the Progol tutorial by Muggleton and Firth (2001) for a more detailed introduction.

[38] Progol, by contrast, uses an A* search (Muggleton, 1995).

[39] Aleph comes with 13 evaluation functions, such as *entropy* and *compression*.

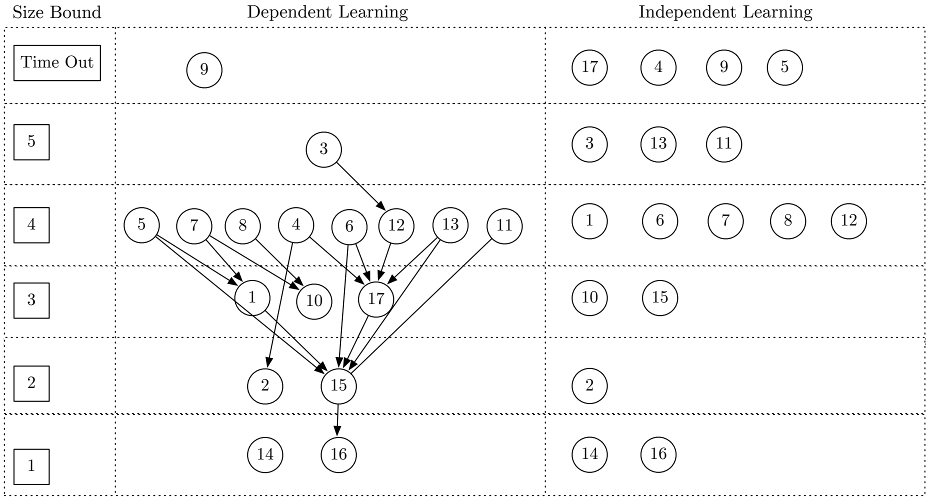


Figure 4: This figure is taken from the work of Lin et al. (2014). It shows the programs learned by dependent (left) and independent (right) learning approaches. The size bound column denotes the number of clauses in the induced program. The nodes correspond to programs and the numbers denote the task that the program solves. For the dependent learning approach, the arrows correspond to the calling relationships of the induced programs. For instance, the program to solve task 3 reuses the solution to solve task 12, which in turn reuses the solution to task 17, which in turn reuses the solution to task 15. Tasks 4, 5, and 16 cannot be solved using an independent learning approach, but can when using a dependent learning approach.

5.5.5 Theory Refinement

The aim of theory *refinement* (Wrobel, 1996) is to improve the *quality* of a theory. Theory *revision* approaches (Adé et al., 1994; Richards & Mooney, 1995) revise a program so that it entails missing answers or does not entail incorrect answers. Theory *compression* (De Raedt et al., 2008) approaches select a subset of clauses such that the performance is minimally affected with respect to certain examples. Theory *restructuring* changes the structure of a logic program to optimise its execution or its readability (Flach, 1993; Wrobel, 1996). We discuss two recent refinement approaches based on PI.

Auto-encoding logic programs. *Auto-encoding logic programs* (ALPs) (Dumančić et al., 2019) invent predicates by simultaneously learning a pair of logic programs: (i) an *encoder* that maps the examples given as interpretations to new interpretations defined entirely in terms of invented predicates[35], and (ii) a *decoder* that reconstructs the original interpretations from the invented ones. The invented interpretations compress the given examples and invent useful predicates by capturing regularities in the data[36]. ALPs, therefore, change the representation of the problem. The most important implication of the approach is that the target programs are easier to express via the invented predicates. The authors experimentally show that learning from the representation invented by ALPs improves the learning per-

[35] The head of every clause in the encoder invents a predicate.

[36] Evaluating the usefulness of invented predicates via their ability to compress a theory goes back to some of the earliest work in ILP by the Duce system (Muggleton, 1987).

formance of generative Markov logic networks (MLN) (Richardson & Domingos, 2006). Generative MLNs learn a (probabilistic) logic program that explains all predicates in an interpretation, not a single target predicate. The predicates invented by ALPs, therefore, aid the learning of all predicates in the BK.

Program refactoring. Knorf (Dumancic et al., 2021) pushes the idea of ALPs even further. After learning to solve user-supplied tasks in the lifelong learning setting, Knorf compresses the learnt program by removing redundancies in it. If the learnt program contains invented predicates, Knorf revises them and introduces new ones that would lead to a smaller program[37]. By doing so, Knorf optimises the representation of obtained knowledge. The refactored program is smaller in size and contains fewer redundant clauses. The authors experimentally demonstrate that refactoring improves learning performance in lifelong learning. More precisely, Metagol learns to solve more tasks when using the refactored BK, especially when BK is large. Moreover, the authors also demonstrate that Knorf substantially reduces the size of the BK program, reducing the number of literals in a program by 50% or more.

6. ILP Case Studies

We now describe in detail four ILP systems: Aleph (Srinivasan, 2001), TILDE (Blockeel & De Raedt, 1998), ASPAL (Corapi et al., 2011), and Metagol (Cropper & Muggleton, 2016). These systems are not necessarily the best, nor the most popular, but use considerably different techniques and are relatively simple to explain. Aleph is based on *inverse entailment* (Muggleton, 1995) and uses *bottom clause construction* to restrict the hypothesis space. Despite its age, Aleph is still one of the most popular systems. TILDE is a first-order generalisation of decision trees and uses information gain to divide and conquer the training examples. ASPAL is a meta-level system that uses an ASP solver to solve the ILP problem, which has influenced much subsequent work, notably ILASP. Finally, Metagol uses a Prolog meta-interpreter to construct a proof of a set of examples and extracts a program from the proof. We discuss these systems in turn.

6.1 Aleph

Progol (Muggleton, 1995) is arguably the most influential ILP system, having influenced many systems (Inoue, 2004; Srinivasan, 2001; Ray, 2009; Ahlgren & Yuen, 2013), which in turn have inspired many other systems (Katzouris et al., 2015, 2016; Schüller & Benz, 2018). Aleph is based on Progol. We discuss Aleph, rather than Progol, because the implementation, written in Prolog, is easier to use and the manual is more detailed.

6.1.1 Aleph Setting

The Aleph problem setting is:

Given:

- A set of mode declarations M
- Background knowledge B in the form of a normal program
- Positive (E^+) and negative (E^-) examples represented as sets of ground facts

Return: A normal program hypothesis H such that:

[37] The idea of learning new predicates to restructure knowledge bases goes back at least to Flach (1993).

- H is consistent with M
- $\forall e \in E^+,\ H \cup B \models e$ (i.e. is complete)
- $\forall e \in E^-,\ H \cup B \not\models e$ (i.e. is consistent)

Note that the examples can be different relations to generalise, i.e. they can have different predicate symbols.

6.1.2 Aleph Algorithm

Aleph starts with an empty hypothesis and uses the following set covering approach:

1. Select a positive example to generalise. If none exists, stop and return the current hypothesis; otherwise proceed to the next step.
2. Construct the most specific clause (the bottom clause) (Muggleton, 1995) that is consistent with the mode declarations (Section 4.4.1) and entails the example.
3. Search for a clause more general than the bottom clause and has the best score.
4. Add the clause to the hypothesis and remove all the positive examples covered by it. Return to step 1.

We discuss the basic approaches to steps 2 and 3.

Step 2: Bottom Clause Construction

The purpose of constructing a bottom clause is to bound the search in step 3. The bottom clause is the most specific clause that entails the example to be generalised. In general, a bottom clause can have infinite cardinality. Therefore, Aleph uses *mode declarations* (Section 4.4.1) to restrict them. Describing how to construct bottom clauses is beyond the scope of this paper. See the paper by Muggleton (1995) or the book of De Raedt (2008) for contrasting methods. Having constructed a bottom clause, Aleph can ignore any clauses that are not more general than it. In other words, Aleph only considers clauses that are generalisations of the bottom clause, which must all entail the example. We use the bottom clause definition provided by De Raedt (2008):

Definition 4 (Bottom clause). Let H be a clausal hypothesis and C be a clause. Then the bottom clause $\bot(C)$ is the most specific clause such that:

$$H \cup \bot(C) \models C$$

Example 7 (Bottom clause). To illustrate bottom clauses in Aleph, we use a modified example from De Raedt (2008). Let M be the mode declarations:

$$\mathtt{M} = \left\{ \begin{array}{l} \texttt{:- modeh(*,pos(+shape)).} \\ \texttt{:- modeb(*,red(+shape)).} \\ \texttt{:- modeb(*,blue(+shape)).} \\ \texttt{:- modeb(*,square(+shape)).} \\ \texttt{:- modeb(*,triangle(+shape)).} \\ \texttt{:- modeb(*,polygon(+shape)).} \end{array} \right\}$$