

Relational Program Synthesis with Numerical Reasoning

Céline Hocquette, Andrew Cropper

University of Oxford

celine.hocquette@cs.ox.ac.uk, andrew.cropper@cs.ox.ac.uk

Abstract

Learning programs with numerical values is fundamental to many AI applications, including bio-informatics and drug design. However, current program synthesis approaches struggle to learn programs with numerical values. An especially difficult problem is learning continuous values from multiple examples, such as intervals. To overcome this limitation, we introduce an inductive logic programming approach which combines relational learning with numerical reasoning. Our approach, which we call NUMSYNTH, uses satisfiability modulo theories solvers to efficiently learn programs with numerical values. Our approach can identify numerical values in linear arithmetic fragments, such as real difference logic, and from infinite domains, such as real numbers or integers. Our experiments on four diverse domains, including game playing and program synthesis, show that our approach can (i) learn programs with numerical values from linear arithmetical reasoning, and (ii) outperform existing approaches in terms of predictive accuracies and learning times.

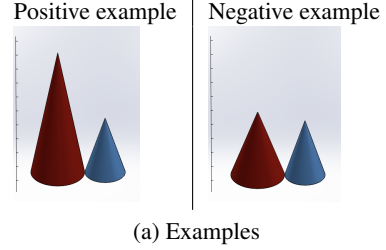
1 Introduction

Zendo is a game in which one player, the Master, creates a rule that structures made of pieces must follow. The rest of the players, as Students, try to discover this rule by building and studying structures which follow or break the rule. The first student to correctly guess the rule wins. For instance, suppose the structure on the left of Figure 1a follows the secret rule while the one on the right does not. Figure 1b shows a possible secret rule. It states that structures must have two pieces in contact, one with size at least 7. Discovering this rule involves identifying the numerical value 7.

Suppose we want to use machine learning to play Zendo, i.e. to learn secret rules from examples of structures. Then we need an approach that can (i) learn explainable rules, and (ii) generalise from small numbers of examples. However, these requirements are difficult for standard machine learning techniques, yet are crucial for many real-world problems (Cropper et al. 2022) including protein folding (Turcotte, Muggleton, and Sternberg 2001), mutagenic activity (Srinivasan et al. 1996) or drug design (Finn et al. 1998).

Inductive logic programming (ILP) (Muggleton 1991) is a form of machine learning that can learn explainable rules

Copyright © 2023, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.



$\text{zendo}(A) \leftarrow \text{piece}(A,B), \text{contact}(B,C),$
 $\text{size}(C,D), \mathbf{\text{geq}(D,7)}.$

(b) A rule which states that two pieces are in contact,
one with size *greater or equal* than 7.

$\text{zendo}(A) \leftarrow \text{piece}(A,B), \text{contact}(B,C), \text{size}(C,D),$
 $\mathbf{\text{geq}(D,N)}, @numerical(N).$
(c) Intermediate hypothesis.

Figure 1: Learning a hypothesis for Zendo. Learning this hypothesis involves reasoning with the numerical predicate *geq* represented in bold to identify the numerical value 7.

from small numbers of examples. Existing ILP techniques could, for instance, learn rules for simple Zendo problems. However, existing approaches struggle to learn rules that require identifying numerical values from infinite domains (Corapi, Russo, and Lupu 2011; Evans and Grefenstette 2018; Cropper and Morel 2021). Moreover, although some ILP approaches can learn programs with numerical values (Muggleton 1995; Hocquette and Cropper 2022), they cannot perform complex numerical reasoning, such as identifying numerical values by reasoning over multiple examples jointly. For instance, they struggle to learn that the size of one piece must be greater than some particular numerical value, or that the sum of the coordinates describing the position of a piece must be lower than some particular numerical value. These limitations are not specific to ILP and, as far as we are aware, apply to all current program synthesis approaches (Raghothaman et al. 2019; Ellis et al. 2018; Shi et al. 2022).

To overcome these limitations, we introduce an approach that can identify numerical values from infinite domains and reason from multiple examples. The key idea of our approach is to decompose the learning task into two stages (i) *program*

search, and (ii) *numerical search*.

In the *program search* stage, the learner searches for partial hypotheses (sets of rules) with variables in place of numerical symbols. This step follows ALEPH’s lazy evaluation procedure (Srinivasan and Camacho 1999). For example, to learn a rule for Zendo, the learner may generate the partial hypothesis shown in Figure 1c. In this hypothesis, the first-order variable N is marked as a numerical variable with the predicate symbol `@numerical` but is not yet bound to any particular value.

In the *numerical search* stage, the learner searches for values for the numerical variables using the training examples. We encode the search for numerical values as a satisfiability modulo theories (SMT) formula. For instance, to find values for N in the hypothesis in Figure 1c, the learner executes the partial hypothesis without its numerical literal $geq(D, N)$ against the examples to find possible substitutions for the variable D , from which it builds a system of linear inequations. These inequations constrain the search for the numerical variable N with the values obtained for D from the examples. Finally, the learner substitutes N in the partial program with any solution found for the inequations.

To implement our idea, we build on the state-of-the-art *learning from failures* (LFF) (Cropper and Morel 2021) ILP approach. LFF is a constraint-driven ILP approach where the learner accumulates constraints on the hypothesis space. A LFF learner continually generates and tests hypotheses, from which it infers constraints. We implement our numerical reasoning approach in NUMSYNTH, which, as it builds on the LFF learner POPPER, supports predicate invention and learning recursive and optimal (in textual complexity) programs. NUMSYNTH additionally uses built-in numerical literals to support linear arithmetic reasoning over integers and real numbers.

Novelty and Contributions Compared to existing approaches, the main novelty of our approach is expressivity: NUMSYNTH can learn programs with numerical values whose identification requires reasoning over multiple examples in linear arithmetic fragments. In other words, our approach can learn programs that existing approaches cannot. For instance, our experiments show that our approach can learn programs of the form shown in Figure 1b. In addition, our approach can (i) efficiently search for numerical values in infinite domains such as real numbers or integers, (ii) identify numerical values which may not appear in the background knowledge, and (iii) learn programs with several chained numerical literals. For instance, it can learn that the sum of two variables is lower than some particular numerical value. As far as we are aware, no existing approach can efficiently solve such problems. Overall, we make the following contributions:

1. We introduce an approach for numerical reasoning in infinite domains. Our approach supports numerical reasoning in linear arithmetic fragments.
2. We implement our approach in NUMSYNTH, which can learn programs with numerical values, perform predicate invention, and learn recursive and optimal programs.
3. We experimentally show on four domains (geometry, biology, game playing, and program synthesis) that our

approach can (i) learn programs requiring numerical reasoning, and (ii) outperform existing ILP systems in terms of learning time and predictive accuracy.

2 Related Work

Program Synthesis Program synthesis approaches that enumerate the search space (Raghothaman et al. 2019; Ellis et al. 2018; Evans et al. 2021) can only learn from small and finite domains and, by contrast with NUMSYNTH, cannot learn from infinite domains. Several program synthesis systems delegate the search for programs to an SMT solver (Jha et al. 2010; Gulwani et al. 2011; Reynolds et al. 2015; Albargouthi et al. 2017). By contrast, we delegate the search for numerical values to an SMT solver. Moreover, NUMSYNTH can learn programs with numerical values from infinite domains. Sketch (Solar-Lezama 2009) uses a SAT solver to search for suitable constants given a partial program, where the constants can be numerical values. This approach is similar to our numerical search stage. However, Sketch does not learn the structure of programs but expects as input a skeleton of a solution: it requires a partial program and its task is to fill in missing values with constants symbols. By contrast, NUMSYNTH learns both the program and numerical values.

ILP Many ILP approaches (Muggleton 1995; Srinivasan 2001) use bottom clause construction to search for programs. However, these approaches can only identify numerical values that appear in the bottom clause of a single example. They cannot reason about multiple examples jointly, which is, for instance, necessary to learn inequations.

Constraint inductive logic programming (Sebag and Rouveirol 1996) uses constraint logic programming to learn programs with numerical values. This approach generalises a single positive example given some negative examples and is restricted to numerical reasoning in difference logic.

Anthony and Frisch (1997) propose an algorithm to learn hypotheses with numerical literals. FORS (Karalić and Bratko 1997) fits regression lines to subsets of the positive examples in a positive example only setting. In contrast to NUMSYNTH, these two approaches allow some error in numerical values predicted by numerical literals. However, these two approaches follow top-down refinement with one specialisation step at a time, which prevents them from learning hypotheses with multiple chained numerical literals.

TILDE (Blockeel and De Raedt 1998) uses a discretization procedure to find relevant candidate numerical constants (Blockeel and De Raedt 1997). However, TILDE cannot learn recursive programs and struggles to learn from small numbers of examples.

Many recent ILP systems enumerate every possible rule in the search space (Corapi, Russo, and Lupu 2011; Kaminski, Eiter, and Inoue 2018; Evans and Grefenstette 2018; Schüller and Benz 2018) or all constant symbols as unary predicate symbols (Evans and Grefenstette 2018; Cropper and Morel 2021; Purgał, Cerna, and Kaliszyk 2022) and therefore cannot handle infinite domains.

LFF Recent LFF systems represent constant symbols with unary predicate symbols (Cropper and Morel 2021; Purgał,

Cerna, and Kaliszyk 2022), which prevents them from learning in infinite domains. MAGICPOPPER (Hocquette and Cropper 2022) can identify constant symbols from infinite domains. Similar to ALEPH’s lazy evaluation approach and our program search approach, MAGICPOPPER builds partial hypotheses with variables in place of constant symbols. It then executes the partial hypotheses independently over each example to identify particular candidate constant symbols. However, it may find an intractable number of candidate constants when testing hypotheses with non-deterministic predicates with a large or infinite number of answer substitutions, such as *greater than*. Moreover, it cannot perform numerical reasoning from multiple examples jointly. By contrast, NUMSYNTH uses all the examples simultaneously when reasoning about numerical values and can for instance learn intervals whereas MAGICPOPPER cannot.

Lazy Evaluation The most related work is an extension of ALEPH that supports *lazy evaluation* (Srinivasan and Camacho 1999). During the construction of the bottom clause, ALEPH replaces numerical values with existentially quantified variables. During the refinement search of the bottom clause, ALEPH finds substitutions for these variables by executing the partial hypothesis on the examples. This procedure can predict output numerical variables using custom loss functions measuring error (Srinivasan et al. 2006), while NUMSYNTH cannot. However, ALEPH needs the user to write background definitions to find numerical values, such as a definition for computing a threshold or linear regression coefficients from data. By contrast, NUMSYNTH has built-in numerical literals. Moreover, ALEPH executes each definition used during lazy evaluation independently which prevents it from learning hypotheses with multiple literals requiring lazy evaluation sharing variables, such as an upper and a lower bound for the same variable. By contrast, NUMSYNTH can learn hypotheses with multiple chained numerical literals. Finally, ALEPH does not support predicate invention, is not guaranteed to learn optimal (textually minimal) programs, and struggles to learn recursive programs.

3 Problem Setting

We now describe our problem setting. We assume familiarity with logic programming (Lloyd 2012). Our problem setting is the learning from failures (LFF) (Cropper and Morel 2021) setting, which is based on the learning from entailment setting (Muggleton and De Raedt 1994) of ILP. LFF assumes a meta-language \mathcal{L} , which is a language about hypotheses. LFF uses hypothesis constraints, expressed in \mathcal{L} , to restrict the hypothesis space. A LFF input is defined as:

Definition 1 A LFF input is a tuple $(E^+, E^-, B, \mathcal{H}, C)$ where E^+ and E^- are sets of ground atoms representing positive and negative examples respectively, B is a definite program representing background knowledge, \mathcal{H} is a hypothesis space i.e a set of possible hypotheses as definite programs, and C is a set of hypothesis constraints expressed in the meta-language \mathcal{L} .

Given a set of hypotheses constraints C , we say that a hypothesis H is consistent with C if, when written in \mathcal{L} , H does

not violate any constraint in C . We call \mathcal{H}_C the subset of \mathcal{H} consistent with C . We define a LFF solution:

Definition 2 Given a LFF input $(E^+, E^-, B, \mathcal{H}, C)$, a LFF solution is a hypothesis $H \in \mathcal{H}_C$ such that H is complete with respect to E^+ ($\forall e \in E^+, B \cup H \models e$) and consistent with respect to E^- ($\forall e \in E^-, B \cup H \not\models e$).

Conversely, given a LFF input, a hypothesis H is *incomplete* when $\exists e \in E^+, H \cup B \not\models e$, and is *inconsistent* when $\exists e \in E^-, H \cup B \models e$.

In general, there might be multiple solutions given a LFF input. We associate a cost to each hypothesis and prefer *optimal* solutions, which are solutions with minimal cost. In the following, we use as cost function the size of hypotheses, measured as the number of literals in it.

A hypothesis which is not a solution is called a failure. A LFF learner identifies constraints from failures to restrict the hypothesis space. For instance, if a hypothesis is inconsistent, a *generalisation* constraint prunes its generalisations, as they are provably also inconsistent.

4 Numerical Reasoning

We extend the framework presented in the previous section to allow numerical reasoning in possibly infinite domains. We assume familiarity with SMT theory (De Moura and Bjørner 2011). The idea is to separate the search into two stages (i) *program search*, and (ii) *numerical search*. First, the learner generates partial programs with first-order numerical variables in place of numerical values. Then, the learner searches for numerical values to fill in the numerical variables.

4.1 Program Search

The learner first searches for partial programs with variables, called numerical variables, in place of numerical values.

Numerical Variables. We extend the meta-language \mathcal{L} of LFF to contain numerical variables. A numerical variable is a first-order variable that can be substituted by a numerical value, i.e. a numerical variable acts as a placeholder for a numerical symbol. In the following, we represent numerical variables with the unary predicate symbol *@numerical*. For example, in the program in Figure 1c, the variable N marked with the syntax *@numerical* is a numerical variable.

Numerical Literals. A numerical literal is a literal which requires numerical reasoning and whose arguments all are numerical. A numerical literal may contain numerical variables as arguments. During the *program search* stage, the learner builds partial hypotheses with variables in place of numerical variables in numerical literals. For example, the learner may generate the following program, where the literal *leq(B, N)* is a numerical literal which contains the numerical variable N :

$$H: f(A) \leftarrow \text{length}(A, B), \text{leq}(B, N), @numerical(N)$$

Related Variables. A related variable is a variable that appears both in a numerical literal and in a regular literal. Related variables act as bridges between relational learning and numerical reasoning. For instance, the variable B is a variable related to the numerical variable N in the program H above. Possible substitutions for the related variables are identified

by executing the hypothesis without its numerical literals over the positive and negative examples. For instance, given the positive examples $\{f([a, b]), f([\])\}$ and the negative examples $\{f([b, c, a, d, e, f]), f([c, e, d, a, b])\}$, the hypothesis H above has the following positive $S_P(B)$ and negative $S_N(B)$ substitutions for the related variable B : $S_P(B) = \{2, 0\}$ and $S_N(B) = \{6, 5\}$.

4.2 Numerical Search

During the numerical search stage, the learner builds an SMT formula from the definition of the numerical literals and the possible substitutions for the related variables. It generates a constraint for each positive example to ensure the learned hypothesis covers it. It generates a constraint for each negative example to ensure the learned hypothesis does not cover it. For instance, the learner translates the numerical search in the hypothesis above as the following SMT formula:

$$2 \leq N \wedge 0 \leq N \wedge \neg(6 \leq N) \wedge \neg(5 \leq N)$$

The appendix includes more details of how NUMSYNTH builds the SMT formula. The solutions for the formula represent possible numerical values for the partial program tested. In other words, if the formula is satisfiable, any solution is a substitution for the numerical variables in the partial program such that the resulting program is a solution to the LFF input. For instance, the substitution $N = 3$ is a solution to the formula above. Applying this substitution to the program H above forms the following LFF solution:

$$H: f(A) \leftarrow \text{length}(A, B), \text{leq}(B, 3)$$

In practice, to account for non-deterministic literals, we build one expression from each of the substitutions found for the related variables. The constraints assert that at least one of these expressions is verified for each positive example and none are verified for any negative examples. In other words, the multi-instance problem (Dietterich, Lathrop, and Lozano-Pérez 1997) is delegated to the solver through a disjunction.

The number of literals in the resulting SMT formula is upper bounded by $n_e * s * n_v$, where n_e is the number of examples, s is the maximum number of substitutions per example, and n_v is the number of variables in the candidate hypothesis. A proof of this result is in the appendix.

4.3 Constraints

If a candidate program is not a solution to the LFF input, we generate constraints to prune other programs from the hypothesis space and constrain subsequent *program search* stages. Following Hocquette and Cropper (2022), we use the following constraints. Given a partial program P with numerical variables generated in the *program search* stage:

1. If there is no solution in the *numerical search* stage, then P cannot cover any of the positive examples and therefore P is too specific. We prune programs which include one of the specialisations of P as a subset.
2. If all solutions found in the *numerical search* stage result in programs which are too specific, then P is too specific. We prune specialisations of P without additional numerical literals.

Literal	Definition	Example
$\text{geq}(A, N)$	$A \geq N$	$\text{geq}(A, 3)$
$\text{leq}(A, N)$	$A \leq N$	$\text{leq}(A, 5.2)$
$\text{add}(A, B, C)$	$A + B = C$	$\text{add}(A, B, C)$
$\text{mult}(A, N, C)$	$A * N = C$	$\text{mult}(A, 2, C)$

Figure 2: Numerical literals in NUMSYNTH. N is a numerical variable which can be substituted for a numerical value. Variables A, B, C, N range over real numbers or integers.

3. If all solutions found in the *numerical search* stage result in programs which are too general, then P is too general. We prune non-recursive generalisations of P .

These constraints are optimally sound (Hocquette and Cropper 2022) as they do not prune optimal solutions from the hypothesis space. The appendix contains an example of constraints generated.

5 Implementation

We present our implementation called NUMSYNTH. We first briefly describe POPPER (Cropper and Morel 2021), on which NUMSYNTH is based.

5.1 POPPER

POPPER takes as input a LFF input, which contains a set of positive and negative examples, a background knowledge B , a bound over the size of hypotheses allowed in \mathcal{H} , and a set of hypothesis constraints C . POPPER learns hypotheses as definite programs. To generate hypotheses, POPPER uses an ASP program P whose models are hypothesis solutions represented in the meta-language \mathcal{L} . In other words, each model (answer set) of P represents a hypothesis. POPPER follows a *generate*, *test*, and *constrain* loop to find a solution. First, it generates a hypothesis as a solution to the ASP program P with the ASP system Clingo (Gebser et al. 2014). Then, POPPER tests this hypothesis given the background knowledge against the examples, typically using Prolog. If the hypothesis is a solution, POPPER returns it. Otherwise, the hypothesis is a failure: POPPER identifies the kind of failure and builds constraints accordingly. For instance, if the hypothesis is inconsistent, POPPER builds a generalisation constraint. POPPER adds these constraints to the ASP program P to constrain the subsequent *generate* steps. This loop repeats until a hypothesis solution is found or until there are no more models to the ASP program P .

5.2 NUMSYNTH

NUMSYNTH builds on POPPER. It also follows a *generate*, *test*, and *constrain* loop.

Partial programs. First, NUMSYNTH generates partial programs which may contain numerical literals. The maximum number of numerical literals in a clause is a user parameter, with a default value of 2. This setting expresses the trade-off between search complexity and expressivity.

Numerical literals. NUMSYNTH supports the built-in numerical literals shown in Figure 2. While *add* reasons from

Fragment	NUMSYNTH	Example
Linear real arithmetic	✓	$X + 6.3 * Y \leq 3$
Linear integer arithmetic	✓	$U + 6 * V \leq 3$
Mixed real / integer	✓	$X + 6.3 * V \leq 3$
Integer difference logic	✓	$U - V \leq 4$
Real difference logic	✓	$X - Y \leq 4$
Unit two-variable / inequality	✓	$X + Y \leq 4$
Polynomial real arithmetic	X	$X^2 + Y^2 = 2$
Non-linear integer arithmetic	X	$U^2 = 2$

Figure 3: Arithmetical fragments supported by NUMSYNTH. X and Y range over real numbers and U and V over integers.

regular numerical first-order variables and does not have numerical variables which are substituted for constant symbols, other literals have such numerical variables represented by N . As shown in Figure 3, the numerical literals in NUMSYNTH are sufficient to reason about standard linear arithmetic fragments. Other fragments which currently are not supported by NUMSYNTH include non-linear arithmetic for complexity reasons. More details about numerical literals are provided in the appendix. The user can specify a subset of these numerical literals to use if this bias is known. Otherwise, NUMSYNTH automatically identifies which of these literals to use, at the expense of more search. If known, the user also can optionally specify argument types (real or integer) and domains for the numerical variables to restrict the search.

Numerical Reasoning NUMSYNTH performs numerical reasoning during the *test* stage. It first identifies possible substitutions for the related variables. To do so, it adds related variables in numerical literals as new arguments to the head literal. Then, it removes numerical literals from the hypothesis. For instance, the hypothesis H below becomes H' :

$$H: f(A) \leftarrow \text{length}(A,B), \text{leq}(B,C) \\ H': f(A,B) \leftarrow \text{length}(A,B)$$

NUMSYNTH executes the resulting hypothesis over the examples with Prolog. We use Prolog because of its ability to handle lists and large, potentially infinite, domains. NUMSYNTH saves the substitutions found for the newly added head variables. It then builds an SMT formula from the definition of the numerical literals and the values found for the related variables. Finally, NUMSYNTH uses the SMT solver Z3 (Moura and Bjørner 2008) to determine the satisfiability of the resulting SMT formula. If a solution exists, it saves a possible value for each numerical variable and substitutes these values into the original program. Otherwise, it repeats the loop and generates more programs.

We set the SMT solver to return any solution to the formula. We do not optimise the choice of numerical values because it is unclear how to trade off learning textually minimal programs and learning optimal numerical values (potentially multiple ones in a program). Addressing this limitation is future work.

6 Experiments

We claim that NUMSYNTH can learn programs with numerical values from numerical reasoning. Therefore, our experiments aim to answer the following question:

$\text{halfplane}(A,B) \leftarrow \text{mult}(A,3,D), \text{add}(B,D,E), \text{leq}(E,6).$

Figure 4: Example *halfplane* hypothesis. Numerical literals and examples of numerical values are in bold.

$\text{zendo2}(A) \leftarrow \text{piece}(A,B), \text{position}(B,C,D),$
 $\text{add}(C,D,E), \text{leq}(E,6.92).$
 $\text{zendo2}(A) \leftarrow \text{piece}(A,B), \text{rotation}(B,D),$
 $\text{leq}(D,4.12), \text{geq}(D,3.23)$

Figure 5: Example *zendo2* hypothesis. Numerical literals and examples of numerical values are in bold.

Q1 Can NUMSYNTH learn programs with numerical values?

To answer **Q1**, we evaluate NUMSYNTH on a variety of tasks requiring numerical reasoning.

We also claim that our approach can reduce search complexity and thus improve learning performance. Therefore, our experiments aim to answer the following question:

Q2 How well does NUMSYNTH perform compared to other approaches?

To answer **Q2**, we compare NUMSYNTH against MAGICPOPPER and ALEPH, which are the only program synthesis systems capable of learning programs with numerical constants¹.

As described in Section 4.2, the size of the SMT formula built by NUMSYNTH is an increasing function of the number of examples. Therefore, to evaluate how well our system scales, we investigate the following question:

Q3 How well does NUMSYNTH scale with the number of examples?

To answer **Q3**, we vary the number of examples and evaluate the performance of NUMSYNTH.

Domains We consider four domains which we briefly describe. The appendix includes more details.

Geometry. These tasks involve learning that points belong to geometrical objects (interval, halfplane), which parameters are numerical values to be learned. Figure 4 shows an example hypothesis for the task *halfplane*.

Zendo. Zendo is a multiplayer game in which players aim to identify a rule which structures made from a set of pieces with varying attributes must follow. We consider four increasingly complex tasks. Figures 1b and 5 show examples of target hypotheses for tasks 1 and 2, respectively.

Pharmacophores. The goal is to identify properties of pharmacophores responsible for medicinal activity (Finn et al. 1998). This domain requires reasoning about distances between atoms with varying properties and bonds linking each other. We consider four increasingly complex tasks. Figure 6 shows an example of a target hypothesis for task 4.

¹We also considered other systems (Corapi, Russo, and Lupu 2011; Evans and Grefenstette 2018; Kaminski, Eiter, and Inoue 2018; Schüller and Benz 2018). However, these systems cannot handle infinite domains and thus cannot solve any of the tasks proposed or require user-provided metarules (Muggleton, Lin, and Tamaddoni-Nezhad 2015) making them unusable in practice (Cropper et al. 2022).

$\text{pharma4}(A) \leftarrow \text{zinc}(A,B), \text{hacc}(A,C), \text{dist}(A,B,C,D),$
 $\text{leq}(D, \mathbf{4.18}), \text{geq}(D, \mathbf{2.22}).$
 $\text{pharma4}(A) \leftarrow \text{hacc}(A,B), \text{hacc}(A,C), \text{dist}(A,B,C,D),$
 $\text{geq}(D, \mathbf{1.23}), \text{leq}(D, \mathbf{3.41}).$
 $\text{pharma4}(A) \leftarrow \text{zinc}(A,B), \text{zinc}(A,C), \text{bond}(B,C,du),$
 $\text{dist}(A,B,C,D), \text{leq}(D, \mathbf{1.23}).$

Figure 6: Example *pharma4* hypothesis. Numerical literals and examples of numerical values are in bold.

Program Synthesis. We consider three program synthesis tasks. These tasks are list transformation tasks which involve learning recursive programs and numerical reasoning.

Systems To evaluate **Q2**, we compare NUMSYNTH against MAGICPOPPER and ALEPH. We briefly describe each of these systems. The appendix contains more details.

MAGICPOPPER and NUMSYNTH. We provide NUMSYNTH and MAGICPOPPER with the same input. We allow MAGICPOPPER to learn constant symbols for variables of type real or integer in literals with a finite number of answer substitutions. The experimental difference is the ability to perform numerical reasoning for NUMSYNTH.

ALEPH. We provide ALEPH with definitions adapted from NUMSYNTH’s numerical literals to fit its lazy evaluation procedure. ALEPH uses a different bias than NUMSYNTH to bound the hypothesis space. Therefore, the comparison is less fair and should be interpreted as indicative only.

Experimental Setup We enforce a timeout of 10 minutes per task. We measure predictive accuracy and learning time. We measure the mean and standard error over 10 trials. We use an 8-Core 3.2 GHz Apple M1 and a single CPU.

6.1 Experiment 1: Comparison Against SOTA

Task	ALEPH	MAGICPOPPER	NUMSYNTH
<i>interval</i>	69 ± 1	70 ± 0	99 ± 1
<i>halfplane</i>	99 ± 0	84 ± 7	96 ± 1
<i>zendo1</i>	98 ± 0	68 ± 3	99 ± 0
<i>zendo2</i>	51 ± 1	56 ± 1	96 ± 1
<i>zendo3</i>	71 ± 1	51 ± 1	96 ± 1
<i>zendo4</i>	63 ± 1	52 ± 1	94 ± 1
<i>pharma1</i>	82 ± 1	64 ± 3	99 ± 0
<i>pharma2</i>	83 ± 1	77 ± 2	95 ± 1
<i>pharma3</i>	81 ± 1	82 ± 1	98 ± 1
<i>pharma4</i>	76 ± 1	62 ± 2	92 ± 1
<i>member_between</i>	49 ± 0	75 ± 4	97 ± 1
<i>last_leq</i>	50 ± 0	51 ± 1	98 ± 1
<i>next_geq</i>	50 ± 0	50 ± 0	92 ± 5

Table 1: Predictive accuracies. We round accuracies to integer values. The error is standard error.

Tables 1 and 2 show the results. They show that NUMSYNTH consistently achieves high accuracy on all tasks. The accuracy is not maximal because, given a training set, several numerical values may result in a complete and consistent hypothesis, and NUMSYNTH does not optimise the choice

Task	ALEPH	MAGICPOPPER	NUMSYNTH
<i>interval</i>	1 ± 0	0 ± 0	0 ± 0
<i>halfplane</i>	1 ± 0	60 ± 26	2 ± 1
<i>zendo1</i>	25 ± 8	timeout	10 ± 1
<i>zendo2</i>	68 ± 18	97 ± 11	17 ± 1
<i>zendo3</i>	106 ± 26	112 ± 9	69 ± 2
<i>zendo4</i>	147 ± 30	timeout	76 ± 2
<i>pharma1</i>	2 ± 0	3 ± 0	1 ± 0
<i>pharma2</i>	10 ± 2	7 ± 0	2 ± 0
<i>pharma3</i>	24 ± 3	66 ± 5	20 ± 1
<i>pharma4</i>	3 ± 0	62 ± 2	20 ± 0
<i>member_between</i>	1 ± 0	161 ± 38	2 ± 0
<i>last_leq</i>	0 ± 0	589 ± 10	13 ± 1
<i>next_geq</i>	0 ± 0	336 ± 17	39 ± 6

Table 2: Learning times. We round times over one second to the nearest second. The error is standard error.

of numerical values. For instance, given the SMT formula, $2 \leq N \wedge 0 \leq N \wedge \neg(6 \leq N) \wedge \neg(5 \leq N)$, NUMSYNTH may return any value N such that $2 \leq N < 5$.

These results demonstrate that NUMSYNTH can learn programs with numerical values in a reasonable time (less than 80s) in a variety of domains. NUMSYNTH can identify numerical values which require reasoning from multiple examples and which may not appear in the background knowledge. For instance, it can solve *pharma1* which involves learning that the distance between two atoms must be smaller than a particular value. It also can learn programs with numerical values from infinite domains, such as real numbers or integers. Finally, it can learn hypotheses with multiple chained numerical literals for instance for *halfplane* or *pharma4* (Figures 4 and 6). Given these results, we answer **Q1** positively.

We compare NUMSYNTH against ALEPH and MAGICPOPPER. Table 1 shows NUMSYNTH achieves higher or equal accuracies than both ALEPH and MAGICPOPPER. An independent t-test confirms the significance of the difference at the $p < 0.01$ level for all tasks except *halfplane* and *zendo1*. These results show that NUMSYNTH can solve tasks other systems struggle with. For instance, ALEPH struggles to learn hypotheses with multiple numerical literals sharing variables such as for *zendo2* or *zendo3*. ALEPH performs lazy evaluation over all substitutions for positive and negative examples and, therefore, struggles to learn disjunctive numerical hypotheses such as for *zendo2* or *pharma2*. ALEPH may instead learn a hypothesis as facts, which do not generalise to the test set. Finally, ALEPH struggles to learn recursive programs and performs poorly on the program synthesis tasks. ALEPH can perform well on other tasks, such as *halfplane* or *zendo1*.

MAGICPOPPER can learn programs, potentially recursive ones, with constant symbols from infinite domains. However, it cannot reason from multiple examples jointly and cannot identify constants in literals with large or infinite number of substitutions, such as *greater than*. These limitations prevent it from learning inequalities, such as in *pharma2*.

Table 2 shows the learning times. It shows ALEPH can have longer learning times than NUMSYNTH. For instance,

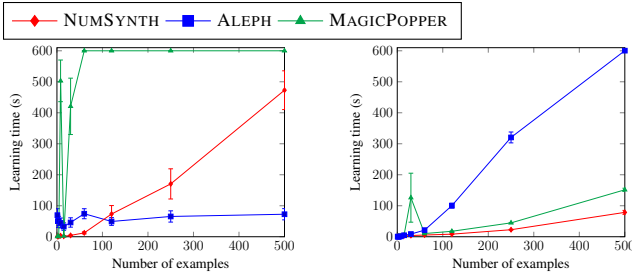


Figure 7: Learning time versus the number of examples for *zendo1*.

Figure 8: Learning time versus the number of examples for *pharma2*.

ALEPH solves *zendo1* in 25s while NUMSYNTH requires 10s. Yet, in contrast to ALEPH, NUMSYNTH searches for textually optimal solutions. NUMSYNTH also outperforms MAGICPOPPER in terms of learning times. Owing to the lack of numerical reasoning ability, MAGICPOPPER is unable to express a concise hypothesis on some tasks and therefore searches up to larger depth. Moreover, MAGICPOPPER follows a generate-and-test approach to identify numerical values: it first generates candidate numerical values from the execution of the partial programs over single examples. Then, it tests candidate values over all examples. Conversely, NUMSYNTH solves a single problem with all examples jointly. It thus avoids the need to consider possibly many candidate values and can be more efficient. Given these results, the answer to **Q2** is that NUMSYNTH can outperform existing approaches in terms of learning times and predictive accuracies when learning programs with numerical values.

6.2 Experiment 2: Scalability

We compare the performance of NUMSYNTH against ALEPH and MAGICPOPPER when varying the number of training examples. We use the task *zendo1*, which other systems can solve. However, the main advantage of our approach is that it can learn concepts existing approaches cannot. Therefore, we also evaluate scalability on the task *pharma2*, which existing systems struggle to solve. Figures 7 and 8 show the learning times versus the number of examples. The appendix shows the predictive accuracies. They are not maximal for MAGICPOPPER and ALEPH on *pharma2*.

As the number of examples grows, the complexity of the learning task, and thus the learning time, increases. Predictive accuracies degrade when timeout is reached for ALEPH and MAGICPOPPER. NUMSYNTH has shorter learning times than MAGICPOPPER on both tasks and its learning time increases slower. MAGICPOPPER generates all candidate numerical values derivable from single examples, then tests them against the remaining examples. Conversely, NUMSYNTH generates constraints from all examples, which it can propagate when solving the SMT formula. It thus achieves shorter learning times. However, owing to the complexity of the SMT problem, NUMSYNTH can struggle to scale to large numbers of examples. The SMT formula can include disjunctions in the case of non-deterministic literals which further adds complexity. The appendix includes a breakdown of the learning

time of NUMSYNTH. It shows its learning time is dominated by the construction and solving of the SMT formula. Finally, NUMSYNTH scales better than ALEPH on *pharma2* but worse on *zendo1*. Therefore, the answer to **Q3** is that NUMSYNTH scales better than MAGICPOPPER with the number of examples and can scale better than ALEPH. However, scalability is limited by the complexity of the numerical reasoning stage. This result highlights a limitation of NUMSYNTH.

7 Conclusions and Future Work

Learning programs with numerical values is essential for many AI applications. However, existing program synthesis systems struggle to identify numerical values from infinite domains and reason about multiple examples. To overcome these limitations, we have introduced NUMSYNTH, an ILP system that combines relational learning and numerical reasoning to efficiently learn programs with numerical values. The key idea of our approach is to decompose learning into two stages: (i) the search for a program, and (ii) the search for numerical values. During the search for a program, NUMSYNTH builds partial programs with variables in place of numerical values. Then, given a partial program, NUMSYNTH searches for numerical values by building an SMT formula using the training examples. NUMSYNTH uses a set of built-in numerical literals (Figure 2) to support a large class of arithmetical fragments (Figure 3), such as *linear integer arithmetic*. Our experiments on four domains (geometry, game playing, biology, and program synthesis) show that our approach can (i) learn programs with numerical values, and (ii) improve predictive accuracies and reduce learning times compared to state-of-the-art ILP systems. In particular, it can learn programs with multiple numerical values, including recursive programs. In other words, we have shown that NUMSYNTH can solve numerical tasks that existing systems cannot. At a higher level, we think that this paper helps bridge relational and numerical learning.

Limitations and Future Work

Scalability. A limitation to the scalability of our approach is the complexity of the numerical reasoning stage, which is a function of the number of (i) examples, and (ii) numerical variables. Future work will aim to identify a subset of the examples which are sufficient to identify suitable numerical values (Anthony and Frisch 1997).

Cost Function. NUMSYNTH learns optimal programs, where the cost function is the size of the hypothesis (the number of literals in it). However, it might be desirable to prefer hypotheses based on different criteria, such as maximum margin or mean square error in the case of numerical prediction (Srinivasan and Camacho 1999). Future work should explore learning with alternative cost functions.

Noise. In contrast to other ILP systems (Karalič and Bratko 1997; Blockeel and De Raedt 1998; Srinivasan 2001), NUMSYNTH cannot identify numerical values from noisy examples. Wahlig (2022) extended LFF to support learning from noisy examples. This extension should be directly applicable to NUMSYNTH.

Code, Data, and Appendices

The experimental code and data are available at <https://github.com/celinehocquette/numsynth-aaai23>.

Acknowledgements

This research was funded in whole, or in part, by the EPSRC grant *Explainable Drug Design* and the EPSRC fellowship *The Automatic Computer Scientist* (EP/V040340/1). For the purpose of Open Access, the author has applied a CC BY public copyright licence to any Author Accepted Manuscript version arising from this submission. The authors thank Sebastijan Dumančić, Håkan Kjellerstrand, Oghenejokpeme Orhobor, and Jean-Christophe Rohner for valuable feedback.

Appendices

This appendix contains supplementary material for the paper *Relational program synthesis with numerical reasoning*. Its outline is as follows.

- Appendix A provides details about the encoding of the numerical search stage, and a proposition providing a bound over the number of literals in the SMT formula generated by NUMSYNTH.
- Appendix B describes the constraints used in NUMSYNTH and an example.
- Appendix C provides details on the implementation of NUMSYNTH.
- Appendix D describes our experiments, including sample solutions. It also provides additional results, namely the accuracy versus the number of examples for *zendo1* and for *pharma2* and the average proportion of learning time spent in the different learning stages for all tasks.

A Numerical Search

We describe in this section the *numerical search* stage of NUMSYNTH. We assume a partial program H has been identified in the *program search* stage.

NUMSYNTH is based on POPPER+ (Cropper and Hocquette 2022). POPPER+ maintains a set of candidate programs which are partially complete (i.e. cover some of the positive examples) and consistent (i.e. cover no negative examples). POPPER+ then combines some of these candidate programs into a program which is complete (i.e. covers all of the positive examples) and consistent. Therefore, we look for candidate programs which are (i) partially complete, and (ii) consistent. To ensure programs are optimal (textually minimal), we additionally look for programs which (iii) cover the maximum number of positive examples.

During the *numerical search* stage, NUMSYNTH builds an SMT formula, such that any solution, once substituted in the partial program, results in a program which verifies the conditions (i), (ii), and (iii) above. We now describe the SMT formula construction.

Each clause in the partial program H is a conjunction of numerical literals and regular literals. NUMSYNTH builds the

SMT formula from the numerical literals. Each numerical literal represents a relation R . Built-in relations R are part of $\{=, \leq, \geq, +, *\}$. We call xRy the expression built from the variables or constants x and y and the relation R . For instance, if R is \leq , x is the variable v and $y = 3$, then xRy means $v \leq 3$.

Each numerical literal may contain a numerical variable. For each numerical variable, NUMSYNTH generates a variable v . NUMSYNTH assigns this variable v the type and domain provided by the user, with type *real* by default.

For each example e_i , we call $s_{i,m,k}$ the k^{th} possible substitution for the related variables in the m^{th} numerical literal. Each positive example must have at least one substitution that fulfills the hypothesis H . Therefore, each positive example must have at least one substitution that fulfills the conjunction of numerical literals in H . Then, for each positive example, NUMSYNTH generates a Boolean variable b_i and an expression of the form:

$$b_i == \bigvee_k \bigwedge_m s_{i,m,k} R_m v_m$$

To ensure the maximum number of positive examples are covered, NUMSYNTH maximises the objective:

$$\max \sum_i b_i$$

NUMSYNTH generates a constraint to cover at least one positive example:

$$\sum_i b_i \geq 1$$

Each negative example must have no substitution which fulfills the hypothesis. Therefore, each negative example must have no substitution which fulfills the conjunction of numerical literals in H . For each negative example e_i , NUMSYNTH generates a constraint of the form below to ensure it is not covered:

$$\text{not}(\bigvee_k \bigwedge_m s_{i,m,k} R_m v_m)$$

If a solution for this formula exists, NUMSYNTH saves an optimal solution. Then, NUMSYNTH adds a constraint specifying that at least one positive example uncovered by the solution found must now be covered. It searches for another solution covering the maximum number of positive examples that are not already covered. It repeats this process to obtain a set of candidate solutions, each covering at least one uncovered example.

To illustrate the SMT formula construction, we provide the following example:

Example 1 Assume NUMSYNTH generates the following hypothesis:

`zendo(A):-piece(A,B), contact(B,C), size(C,D),
geq(D,E), @numerical(E).`

First, NUMSYNTH executes the hypothesis over the examples. Assume it finds the following substitutions for the related variable D from the positive and negative examples:

$$S_P(D) = \{[8.2, 9.4], [2.3, 10.3]\}$$

$$S_N(D) = \{[2.4, 4.6], [5.3, 1.2]\}$$

In both $S_P(D)$ and $S_N(D)$, the first list corresponds to the first example and the second to the second example. Each example has two substitutions for the variable D . NUMSYNTH generates a variable v_E for the numerical variable E . NUMSYNTH generates a Boolean variable for each positive example:

$$\begin{aligned} b_1 &== (8.2 \geq v_E) \vee (9.4 \geq v_E) \\ b_2 &== (2.3 \geq v_E) \vee (10.3 \geq v_E) \end{aligned}$$

NUMSYNTH maximises the number of positive examples covered:

$$\max \sum_i b_i$$

NUMSYNTH generates a constraint to cover at least one positive example:

$$\sum_i b_i \geq 1$$

For each negative example, NUMSYNTH generates a constraint to ensure it is not covered:

$$\begin{aligned} &\text{not}((2.4 \geq v_E) \vee (4.6 \geq v_E)) \\ &\text{not}((5.3 \geq v_E) \vee (1.2 \geq v_E)) \end{aligned}$$

Finally, NUMSYNTH solves the resulting SMT formula. For this example, the two positive examples can be covered and any solution such that $5.3 < v_B \leq 8.2$ may be returned.

Proposition 1 Assume n_e is the number of training examples, s is the maximum number of substitutions per example, and n_v is the number of variables in the candidate hypothesis, the number of literals in the SMT formula generated NUMSYNTH is at most $n_e * s * n_v$.

Proof 1 For each example e , NUMSYNTH generates one formula. This formula is a disjunction of $s_e \leq s$ terms, one per possible substitution for the example e . Each of these terms is a conjunction of n_v literals, one per variable in the candidate hypothesis. Therefore, the resulting formula has at most $n_e * s * n_v$ literals.

B Constraints

NUMSYNTH is based on the LFF (Cropper and Morel 2021) setting of ILP. A LFF learner accumulates constraints to restrict the hypothesis space. It repeatedly generates hypotheses. If a hypothesis is not a solution, it generates constraints to guide future program generation. Following (Hocquette and Cropper 2022), we add constraints which apply to partial programs with variables in place of constant symbols. For instance, if a partial program H has no substitutions for its numerical variables such that the resulting hypothesis (i) covers no negative examples (ii) covers at least one positive example, we add a specialisation constraint which prunes specialisation of H without additional numerical literals.

Example 2 Consider the examples:

$$\begin{aligned} \text{Pos} &= \{f(4), f(6), f(8)\} \\ \text{Neg} &= \{f(2), f(5), f(11)\} \end{aligned}$$

and the partial program H :

$$H: f(A) \leftarrow \text{geq}(A, B), @numerical(B).$$

There is no value for B such that, once substituted in H , the resulting hypothesis is consistent (i.e. covers no negative examples) and covers at least one positive example. Therefore, we can prune its specialisations which do not have additional numerical literals. In particular, we prune the following partial programs H_1 and H_2 :

$$\begin{aligned} H_1: f(A) &\leftarrow \text{geq}(A, B), @numerical(B), \text{odd}(A). \\ H_2: f(A) &\leftarrow \text{geq}(A, B), @numerical(B), \text{div3}(A). \end{aligned}$$

We provide a counter-example to explain why we cannot prune specialisation with additional numerical literals. Consider the following partial program H_3 :

$$H_3: f(A) \leftarrow \text{geq}(A, B), @numerical(B), \text{leq}(A, C), @numerical(C).$$

If applying the substitution $B = 6$ and $C = 10$, the resulting hypothesis covers the last positive example and no negative example.

C Implementation

C.1 Numerical literals

We provide NUMSYNTH with four built-in numerical literals shown in Figure 2 (*geq*, *leq*, *add* and *mult*). The user currently cannot define new numerical literals, since it would be cumbersome to specify their usage and definitions in SMT format. Moreover, as shown in Figure 3, the four built-in literals provided to NUMSYNTH already cover usual linear arithmetic fragments. We could consider adding more built-in literals or allowing the user to add custom ones if there is a need for some applications.

We set a bound over the maximum number of numerical literal per clause. This bound is a user parameter with a default value of 2. This value has been chosen as it supports most problems encountered.

The user can specify domains for the variables of numerical literals as part of the bias file, if these domains are known. This information is optional but can improve learning performance as it guides the search. The user also can specify the type of variables (currently supported types are *real* or *int*). There is a default value of *real*.

D Experiments

D.1 Domain and Tasks.

We describe the characteristics of the domains and tasks used in our experiments in Tables 3 and 4. Numerical values in target hypotheses are randomised for each run. Figure 9 shows example solutions for each of the tasks. Some of these tasks require identifying non-numerical constant symbols. All systems tested support learning programs with constant symbols. For instance, solving *zendo3* involves identifying

the particular color *blue*. Other tasks require learning recursive programs, which NUMSYNTH and MAGICPOPPER fully support but ALEPH only partially supports.

D.2 Systems

NUMSYNTH and POPPER. We use for both systems the version POPPER+ of POPPER which learns non-separable programs and combines them (Cropper and Hocquette 2022). We allow NUMSYNTH and MAGICPOPPER to use numerical predicates shown in Figure 2 (*geq*, *leq*, *mult*, *add*). We set for both systems the maximum number of numerical predicates in a clause to 2, apart from *halfplane* for which we set it to 3. MAGICPOPPER cannot learn constant symbols in non-deterministic predicates with a large or infinite number of answer substitutions, such as *greater than*. Therefore, we disallow output constant symbols in numerical predicates for MAGICPOPPER. However, we allow MAGICPOPPER to learn constant symbols for other variables of type real or integer. For fairness in the experimental comparison, we disallow parallelisation in the solver during the *numerical search* stage for NUMSYNTH. However, this setting can easily be changed to improve performance on real-world applications.

ALEPH. We allow lazy evaluation for any of the numerical predicates. Since ALEPH does not automatically account for non-determinate predicates, we tailor the background definitions accordingly.

D.3 Experimental Set-up

Figures 10, 11 and 12 show the experimental files for the task *zendo1* for the systems evaluated. MAGICPOPPER can learn constant symbols. Variables which are allowed to be constant symbols are flagged with the predicate *magic_value_type*, which means that any variable of the type specified can be a constant symbol. NUMSYNTH additionally uses *numerical_predicates*. This predicate indicates which built-in numerical predicates are allowed in the learning task. These user can define bounds for the numerical variables in numerical predicates through the predicate *bounds*, although this information is optional.

The experimental files also include magic values declarations. Magic variables can be variables of any types and can appear in any background predicate. For instance, magic variables can be used to find constant symbols which are not numeric, such as string or lists, from user defined background relations. By contrast, numerical predicates are restricted to built-in numerical predicates, and they must have type *real* or *int*. However, the use of numerical predicates allows more complex and elaborated numerical reasoning from multiple examples, but this numerical reasoning comes at the expense of more complexity.

D.4 Results: Scalability

For successive values of n , we sample n positive and n negative examples. Figures 13 and 14 represent the predictive accuracies versus the number of examples for *pharma2*.

D.5 Results: Learning Time

Figure 15 represents the proportion of the learning time spent in the different stages. It shows the learning time is dominated by the numerical stage: about 40% on average over tasks of the learning time is used to build and solve SMT formula, and about 6.5% is used to find substitutions for the related variables with Prolog.

References

- Albarghouthi, A.; Koutris, P.; Naik, M.; and Smith, C. 2017. Constraint-based synthesis of Datalog programs. In *International Conference on Principles and Practice of Constraint Programming*, 689–706. Springer.
- Anthony, S.; and Frisch, A. M. 1997. Generating numerical literals during refinement. In *International Conference on Inductive Logic Programming*, 61–76.
- Blockeel, H.; and De Raedt, L. 1997. Lookahead and discretization in ILP. In *Inductive Logic Programming*, 77–84.
- Blockeel, H.; and De Raedt, L. 1998. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101(1-2): 285–297.
- Corapi, D.; Russo, A.; and Lupu, E. 2011. Inductive Logic Programming in Answer Set Programming. In *Inductive Logic Programming*, 91–97.
- Cropper, A.; Dumancic, S.; Evans, R.; and Muggleton, S. H. 2022. Inductive logic programming at 30. *Mach. Learn.*, 111(1): 147–172.
- Cropper, A.; and Hocquette, C. 2022. Learning programs by combining programs.
- Cropper, A.; and Morel, R. 2021. Learning programs by learning from failures. *Mach. Learn.*, 1–56.
- De Moura, L.; and Bjørner, N. 2011. Satisfiability modulo theories: introduction and applications. *CACM*, 54(9): 69–77.
- Dietterich, T. G.; Lathrop, R. H.; and Lozano-Pérez, T. 1997. Solving the multiple instance problem with axis-parallel rectangles. *Artificial intelligence*, 89(1-2): 31–71.
- Ellis, K.; Morales, L.; Sablé-Meyer, M.; Solar-Lezama, A.; and Tenenbaum, J. 2018. Learning Libraries of Subroutines for Neurally-Guided Bayesian Program Induction. In *Advances in Neural Information Processing Systems*, volume 31.
- Evans, R.; and Grefenstette, E. 2018. Learning explanatory rules from noisy data. *JAIR*, 61: 1–64.
- Evans, R.; Hernández-Orallo, J.; Welbl, J.; Kohli, P.; and Sergot, M. 2021. Making sense of sensory input. *Artificial Intelligence*, 293: 103438.
- Finn, P.; Muggleton, S.; Page, D.; and Srinivasan, A. 1998. Pharmacophore Discovery Using the Inductive Logic Programming System PROGOL. *Mach. Learn.*, 30(2): 241–270.
- Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2014. Clingo= ASP+ control: Preliminary report. *arXiv preprint arXiv:1405.3694*.
- Gulwani, S.; Jha, S.; Tiwari, A.; and Venkatesan, R. 2011. Synthesis of loop-free programs. *ACM SIGPLAN Notices*, 46(6): 62–73.

Listing 1: interval

```
1 interval(A):- leq(A,9816.37), geq(A,4827.12).
```

Listing 2: halfplane

```
1 halfplane(A,B) :- mult(A,3,D), add(B,D,E), leq(E,6).
```

Listing 3: zendo1

```
1 zendo1(A) :- piece(A,B), contact(B,C), size(C,D), geq(E,7.32).
```

Listing 4: zendo2

```
1 zendo2(A) :- piece(A,B), position(B,C,D), add(C,D,E), leq(E,6.92).  
2 zendo2(A) :- piece(A,B), rotation(B,D), leq(D,4.12), geq(D, 3.23).
```

Listing 5: zendo3

```
1 zendo3(A):- piece(A,B), position(B,C,D), leq(C,4.82), leq(D,2.61).  
2 zendo3(A):- piece(A,B), color(B,blue), size(B,C), leq(C,6.39), geq(C,2.96).
```

Listing 6: zendo4

```
1 zendo4(A):- piece(A,B), size(B,C), geq(C,1.23), leq(C,4.66).  
2 zendo4(A):- piece(A,B), position(B,X,Y), leq(X,3.45), leq(Y,6.87).  
3 zendo4(A):- piece(A,B), contact(B,C), rotation(C,D), leq(D,1.18).
```

Listing 7: pharma1

```
1 pharma1(A):- zincsite(A,E),hacc(A,D),dist(A,E,D,B),leq(B,4.98).
```

Listing 8: pharma2

```
1 pharma2(A):- zincsite(A,B),hacc(A,C),dist(A,B,C,D),leq(B,2.94).  
2 pharma2(A):- hacc(A,B),hacc(A,C),dist(A,B,C,D),geq(B,1.92).
```

Listing 9: pharma3

```
1 pharma3(A):- zincsite(A,B), hacc(A,C), dist(A,B,C,D), leq(D,3.58), geq(D,1.78).  
2 pharma3(A):- hacc(A,B), hacc(A,C), bond(A,B,C,du), dist(A,B,C,D), leq(D,2.78).
```

Listing 10: pharma4

```
1 pharma4(A):- zincsite(A, B), hacc(A, C), dist(A, B, C, D), leq(D,4.18),geq(D,2.22).  
2 pharma4(A):- hacc(A, C), hacc(A, E), dist(A, B, C, D), geq(D,1.23), leq(D,3.41).  
3 pharma4(A):- zincsite(A, C), zincsite(A, B), bond(B,C,du), dist(A, B, C, D), leq(D,1.23).
```

Listing 11: member_between

```
1 f(A):- head(A,D),leq(D,53),geq(D,45).  
2 f(A):- tail(A,B),f(B).
```

Listing 12: last_leq

```
1 f(A):- tail(A,C),empty(C),head(A,B),leq(B,14).  
2 f(A):- tail(A,B),f(B).
```

Listing 13: next_geq

```
1 f(A):- head(A,19),tail(A,E),head(E,C),geq(C,27).  
2 f(A):- tail(A,B),f(B).
```

Figure 9: Example solutions.

Domain	# pos examples	# neg examples	# relations in bk
<i>geometry</i>	30	30	4
<i>zendo</i>	30	30	11
<i>pharma</i>	30	30	9
<i>program synthesis</i>	10	10	14

Table 3: Domains description.

Task	# clauses	# literals	# numerical predicates
<i>interval</i>	1	3	2
<i>half plane</i>	1	4	3
<i>zendo1</i>	1	5	1
<i>zendo2</i>	2	10	4
<i>zendo3</i>	2	11	4
<i>zendo4</i>	3	15	5
<i>pharma1</i>	1	5	1
<i>pharma2</i>	2	10	2
<i>pharma3</i>	2	12	3
<i>pharma4</i>	3	18	6
<i>member_between</i>	2	7	2
<i>last_leq</i>	2	8	1
<i>next_geq</i>	2	8	1

Table 4: Learning tasks description.

Hocquette, C.; and Cropper, A. 2022. Learning programs with magic values. *Machine Learning*.

Jha, S.; Gulwani, S.; Seshia, S. A.; and Tiwari, A. 2010. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, 215–224. IEEE.

Kaminski, T.; Eiter, T.; and Inoue, K. 2018. Exploiting Answer Set Programming with External Sources for Meta-Interpretive Learning. *Theory and Practice of Logic Programming*, 18(3-4): 571–588.

Karalič, A.; and Bratko, I. 1997. First order regression. *Mach. Learn.*, 26(2): 147–176.

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

Moura, L. d.; and Bjørner, N. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 337–340. Springer.

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

Muggleton, S. H. 1995. Inverse Entailment and Progol. *New Generation Comput.*, 13(3&4): 245–286.

Muggleton, S. H.; and De Raedt, L. 1994. Inductive Logic Programming: Theory and Methods. *The Journal of Logic Programming*, 19-20: 629–679.

Muggleton, S. H.; Lin, D.; and Tamaddoni-Nezhad, A. 2015. Meta-Interpretive Learning of Higher-Order Dyadic Datalog: Predicate Invention revisited. *Mach. Learn.*, 100(1): 49–73.

Purgał, S. J.; Cerna, D. M.; and Kaliszyk, C. 2022. Learning Higher-Order Logic Programs From Failures. In *IJCAI 2022*, 2726–2733.

Raghothaman, M.; Mendelson, J.; Zhao, D.; Naik, M.; and Scholz, B. 2019. Provenance-guided synthesis of Datalog programs. *Proceedings of the ACM on Programming Languages*, 4(POPL): 1–27.

Reynolds, A.; Deters, M.; Kuncak, V.; Tinelli, C.; and Barrett, C. 2015. Counterexample-guided quantifier instantiation for synthesis in SMT. In *International Conference on Computer Aided Verification*, 198–216. Springer.

Schüller, P.; and Benz, M. 2018. Best-effort inductive logic programming via fine-grained cost-based hypothesis generation. *Mach. Learn.*, 107(7): 1141–1169.

Sebag, M.; and Rouveirol, C. 1996. Constraint inductive logic programming.

Shi, K.; Dai, H.; Ellis, K.; and Sutton, C. 2022. CrossBeam: Learning to search in bottom-up program synthesis. *arXiv preprint arXiv:2203.10452*.

Solar-Lezama, A. 2009. The sketching approach to program synthesis. In *Asian Symposium on Programming Languages and Systems*, 4–13. Springer.

Srinivasan, A. 2001. The ALEPH manual. *Machine Learning at the Computing Laboratory*.

Srinivasan, A.; and Camacho, R. 1999. Numerical reasoning with an ILP system capable of lazy evaluation and customised search. *The Journal of Logic Programming*, 40(2): 185–213.

Srinivasan, A.; Muggleton, S. H.; Sternberg, M. J.; and King, R. D. 1996. Theories for mutagenicity: A study in first-order

```

1  max_vars(6).
2  max_body(5).
3
4  head_pred(zendo,1).
5  body_pred(piece,2).
6  body_pred(color,2).
7  body_pred(size,2).
8  body_pred(position,3).
9  body_pred(rotation,2).
10 body_pred(orientation,2).
11 body_pred(contact,2).
12
13 type(zendo,(state,)).
14 type(piece,(state,piece)).
15 type(color,(piece,color)).
16 type(size,(piece,real)).
17 type(position,(piece,real,real)).
18 type(rotation,(piece,real)).
19 type(orientation,(piece,orientation)).
20 type(contact,(piece,piece)).
21
22 direction(zendo,(in,)).
23 direction(piece,(in,out)).
24 direction(color,(in,out)).
25 direction(size,(in,out)).
26 direction(position,(in,out,out)).
27 direction(rotation,(in,out)).
28 direction(orientation,(in,out)).
29 direction(contact,(in,out)).
30
31 magic_value_type(color).
32 magic_value_type(orientation).
33
34 numerical_pred(geq,2).
35 numerical_pred(leq,2).
36
37 type(geq,(real,real)).
38 type(leq,(real,real)).
39
40 direction(geq,(in, out)).
41 direction(leq,(in, out)).
42
43 numerical_pred(add,3).
44 type(add,(real, real, real)).
45 direction(add,(in,in,out)).
46
47 numerical_pred(mult,3).
48 type(mult,(real, real, real)).
49 direction(mult,(in,out,in)).
50
51 bounds(geq,1,(-10,10)).
52 bounds(leq,1,(-10,10)).
53 bounds(mult,1,(-10,10)).
54 bounds(add,1,(-10,10)).

```

Figure 10: Example of experimental set-up for NUMSYNTH for the learning task *zendo1*.

```

1  max_vars(6).
2  max_body(5).
3
4  head_pred(zendo,1).
5  body_pred(piece,2).
6  body_pred(color,2).
7  body_pred(size,2).
8  body_pred(position,3).
9  body_pred(rotation,2).
10 body_pred(orientation,2).
11 body_pred(contact,2).
12
13 type(zendo,(state,)).
14 type(piece,(state,piece)).
15 type(color,(piece,color)).
16 type(size,(piece,real)).
17 type(position,(piece,real,real)).
18 type(rotation,(piece,real)).
19 type(orientation,(piece,orientation)).
20 type(contact,(piece,piece)).
21
22 direction(zendo,(in,)).
23 direction(piece,(in,out)).
24 direction(color,(in,out)).
25 direction(size,(in,out)).
26 direction(position,(in,out,out)).
27 direction(rotation,(in,out)).
28 direction(orientation,(in,out)).
29 direction(contact,(in,out)).
30
31 magic_value_type(color).
32 magic_value_type(orientation).
33 magic_value_type(real).
34
35 body_pred(my_geq,2).
36 body_pred(my_leq,2).
37 body_pred(my_add,3).
38 body_pred(my_mult,3).
39
40 type(my_geq,(real,real)).
41 type(my_leq,(real,real)).
42 type(my_add,(real,real,real)).
43 type(my_mult,(real,real,real)).
44
45 direction(my_geq,(in, in)).
46 direction(my_leq,(in, in)).
47 direction(my_add,(in, in, out)).
48 direction(my_mult,(in, in, out)).
49
50 num_p(my_geq,2).
51 num_p(my_leq,2).
52 num_p(my_add,3).
53 num_p(my_mult,3).
54
55 incompatible(my_geq,my_geq).
56 incompatible(my_leq,my_leq).
57 incompatible(eq,my_geq).
58 incompatible(eq,leq).
59
60 :- clause(C), #count{P,Vars :
    body_literal(C,P,A,Vars), num_p(P,A)}
    > 2.

```

Figure 11: Example of experimental set-up for MAGICPOP-PER for the learning task *zendo1*.

and feature-based induction. *Artificial Intelligence*, 85(1-2): 277–299.

Srinivasan, A.; Page, D.; Camacho, R.; and King, R. 2006. Quantitative pharmacophore models with inductive logic programming. *Mach. Learn.*, 64(1): 65–90.

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

Wahlig, J. 2022. Learning Logic Programs From Noisy Failures. *CoRR*, abs/2201.03702.

```
1 :- aleph_set(verbosity, 1).
2 :- aleph_set(interactive, false).
3 :- aleph_set(i,4).
4 :- aleph_set(clauselength,6).
5 :- aleph_set(nodes,10000).
6
7 :- modeb(*,zendo(+state)).
8 :- modeb(*,piece(+state,-piece)).
9 :- modeb(*,color(+piece,#color)).
10 :- modeb(*,size(+piece,-real)).
11 :- modeb(*,position(+piece,-real,-real))
12
12 :- modeb(*,rotation(+piece,-real)).
13 :- modeb(*,orientation(+piece,#
    orientation)).
14 :- modeb(*,contact(+piece,-piece)).
15 :- modeb(*,my_geq(+real,#real)).
16 :- modeb(*,my_leq(+real,#real)).
17 :- modeb(*,my_add(+real,+real,-real)).
18 :- modeb(*,my_mult(+real,#real,-real)).
19
20 :- determination(zendo/1,piece/2).
21 :- determination(zendo/1,color/2).
22 :- determination(zendo/1,size/2).
23 :- determination(zendo/1,position/3).
24 :- determination(zendo/1,rotation/2).
25 :- determination(zendo/1,orientation/2).
26 :- determination(zendo/1,contact/2).
27 :- determination(zendo/1,my_geq/2).
28 :- determination(zendo/1,my_leq/2).
29 :- determination(zendo/1,my_add/3).
30 :- determination(zendo/1,my_mult/3).
31
32 :- lazy_evaluate(my_geq/2).
33 :- lazy_evaluate(my_leq/2).
```

Figure 12: Example of experimental set-up for ALEPH for the learning task *zendo1*.

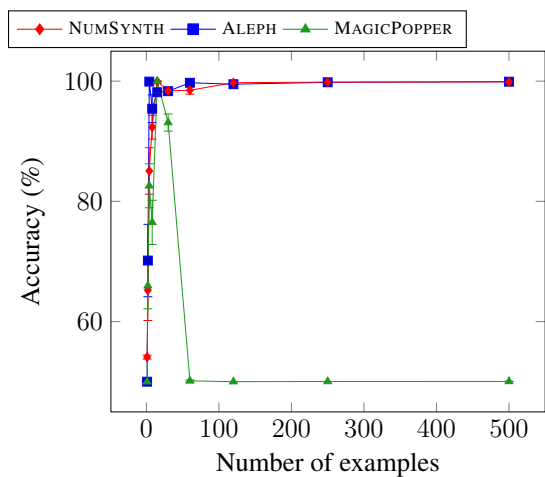


Figure 13: Accuracy versus the number of examples for *zendo1*.

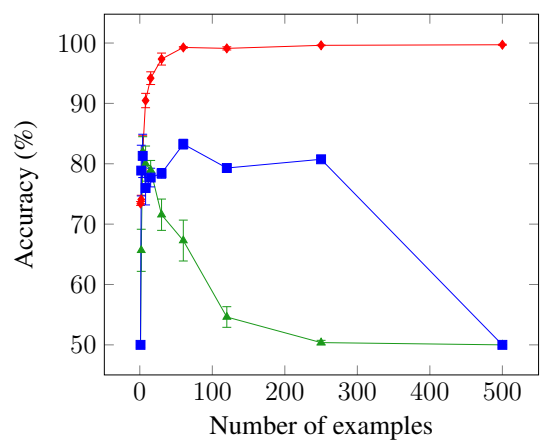


Figure 14: Accuracy versus the number of examples for *pharma2*.

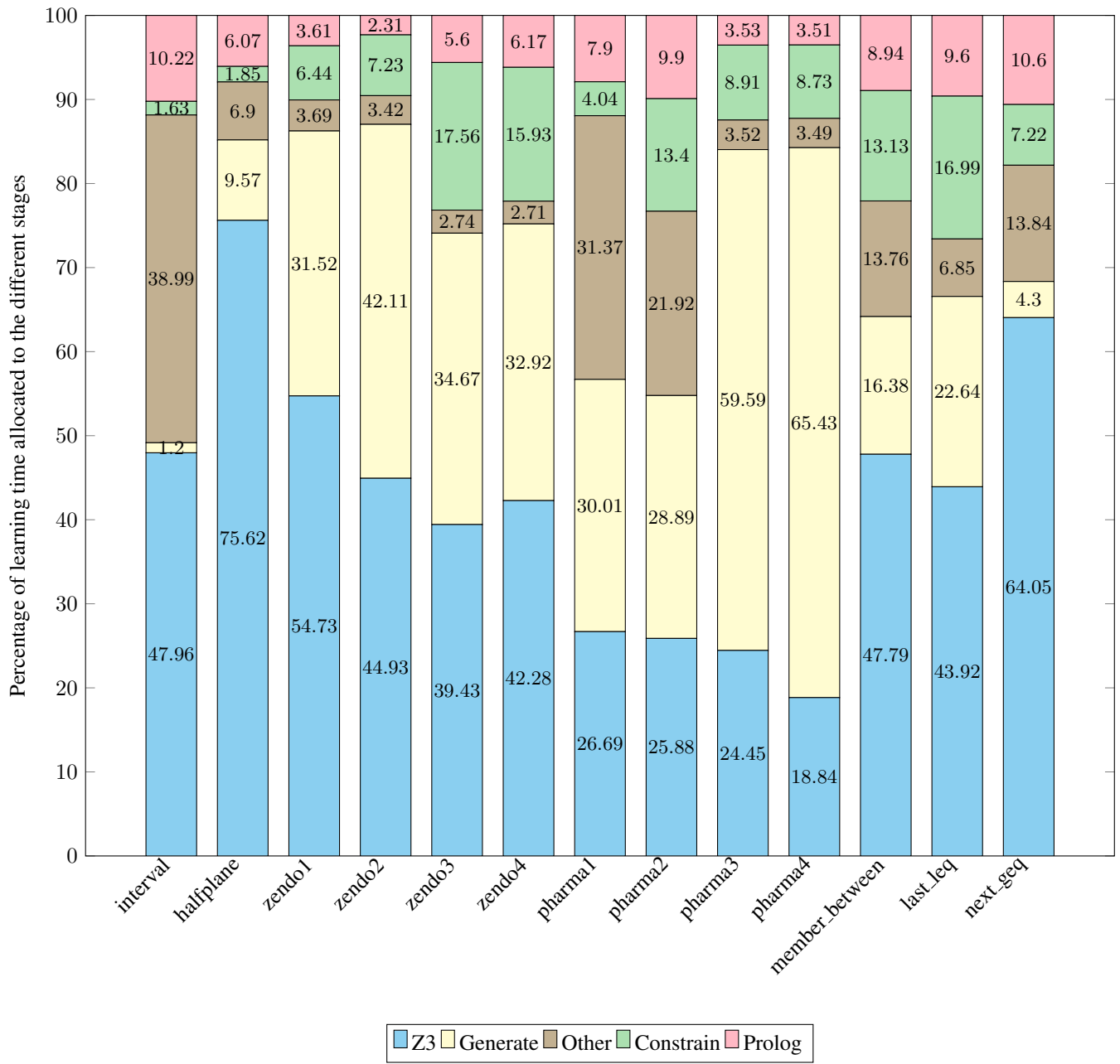


Figure 15: Proportion of learning time spent in each of the different stages. *Z3* refers to building and solving the SMT formula. *Generate* refers to the program generation stage. *Constrain* refers to the generation of constraints to prune the hypothesis space. *Prolog* refers to finding substitutions for the related variables. *Other* refers to any other operation.