Typed Meta-interpretive Learning of Logic Programs

Rolf Morel(✉), Andrew Cropper, and C.-H. Luke Ong

University of Oxford, Oxford, UK
{rolf.morel,andrew.cropper,luke.ong}@cs.ox.ac.uk

Abstract. Meta-interpretive learning (MIL) is a form of inductive logic programming that learns logic programs from background knowledge and examples. We claim that adding types to MIL can improve learning performance. We show that type checking can reduce the MIL hypothesis space by a cubic factor. We introduce two typed MIL systems: Metagol$_T$ and HEXMIL$_T$, implemented in Prolog and Answer Set Programming (ASP), respectively. Both systems support polymorphic types and can infer the types of invented predicates. Our experimental results show that types can substantially reduce learning times.

1 Introduction

Meta-interpretive learning (MIL) [8,22,23] is a form of inductive logic programming (ILP) [20]. MIL learns logic programs from examples and background knowledge (BK) by instantiating *metarules*, second-order Horn clauses with existentially quantified predicate variables. Metarules are a form of declarative bias [28] that define the structure of learnable programs. For instance, to learn the *grandparent/2* relation given the *parent/2* relation, the *chain* metarule would be suitable:

$$P(A, B) \leftarrow Q(A, C), R(C, B)$$

In this metarule[1] the letters P, Q, and R denote existentially quantified second-order variables (variables that can be bound to predicate symbols) and the letters A, B and C denote universally quantified first-order variables (variables that can be bound to constant symbols). Given the *chain* metarule, the background *parent/2* relation, and examples of the *grandparent/2* relation, a MIL learner will try to find the correct predicate substitutions, such as:

$$\{P/grandparent, Q/parent, R/parent\}$$

When applied to the *chain* metarule, these substitutions result in the theory:

$$grandparent(A, B) \leftarrow parent(A, C), parent(C, B)$$

[1] The fully quantified rule is $\exists P \exists Q \exists R \forall A \forall B \forall C\ P(A, B) \leftarrow Q(A, C), R(C, B)$.

R. Morel—Supported by Engineering and Physical Sciences Research Council [grant number EP/N509711/1].

F. Calimeri et al. (Eds.): JELIA 2019, LNAI 11468, pp. 198–213, 2019.
https://doi.org/10.1007/978-3-030-19570-0_13

The MIL hypothesis space grows quickly given more background relations [6,11]. For instance, suppose that when learning the *grandparent/2* relation we have an additional k background relations, such as *head/2*, *tail/2*, *length/2*, etc. Then for the *chain* metarule, there are $k+2$ substitutions for each predicate variable and thus $(k+2)^3$ total substitutions. Existing MIL systems, such as Metagol [9] and HEXMIL [16], would potentially consider all these possible substitutions.

We claim that considering the types of predicates can significantly improve learning performance by reducing the number of predicate substitutions. For instance, suppose that when learning the *grandparent/2* relation we add types to the relations, such as $(person, person)$ to *parent/2*, $(list(T), int)$ to *length/2*, etc. Then given an example of the *grandparent/2* relation with the type $(person, person)$, only the *parent/2* relation (and *grandparent/2* itself) matches the example's type, and so the number of substitutions is reduced from $(k+2)^3$ to 2^3.

Our main contributions are:

- We extend the MIL framework to support polymorphic types (Sect. 3.3).
- We show that type checking can reduce the MIL hypothesis space by a cubic factor (Sect. 3.4).
- We introduce Metagol$_T$ and HEXMIL$_T$ which extend Metagol and HEXMIL respectively. Both support polymorphic types and both can infer types for invented predicates (Sect. 4).
- We conduct experiments which show that types can substantially reduce learning times when there are irrelevant background relations (Sect. 5).

2 Related Work

Program Induction. Program synthesis is the automatic generation of a computer program from a specification. Deductive approaches [19] *deduce* a program from a full specification that precisely states the requirements and behaviour of the desired program. By contrast, program induction approaches *induce* (learn) a program from an incomplete specification, typically input/output examples. Many program induction approaches learn specific classes of programs, such as string transformations [33]. By contrast, MIL is general-purpose, and is, for instance, capable of grammar induction [22], learning robot strategies [7], and learning efficient algorithms [10].

Types in Program Induction. Functional program induction approaches often use types. For instance, bidirectional type checking is the foundation of the MYTH systems [26], where MYTH2 [14] supports polymorphic types. SYNQUID [27] forgoes input/output examples and only uses refinement types as its specification. The authors argue that refinement specifications are terser than examples. However, because of the need to supply correct and informative refinement types, SYNQUID is more similar to deductive synthesis approaches. In contrast to these inductive approaches, we focus on learning logic programs, including support for predicate invention, i.e. the introduction of new predicate symbols [36].

Inductive Logic Programming. ILP is a form of program induction which learns logic programs. ILP systems are typically untyped. The use of types in ILP is mostly restricted to *mode declarations* [21], which are used by many systems [17,21,29,31,35]. Mode declarations define what literals can appear in a program. In the mode language, *modeh* are declarations for head literals and *modeb* are declarations for body literals, where + and − are followed by the type of each argument and represent input and output arguments respectively, e.g. `:-modeh(1,mult(+int,+int,-int))`. Mode declarations are metalogical statements. By contrast, we introduce typed atoms (Definition 4) which are logical statements. As far as we are aware, our work is the first to declaratively represent types. In addition, in contrast to the existing approaches in ILP, our approach supports polymorphic types and we can also infer the types of invented predicates. Finally, to our best knowledge, we are the first to provide theoretical results that show that types can improve learning performance (Theorem 1).

MIL is a form of ILP that supports predicate invention and learning recursive programs. MIL is typically based on a Prolog meta-interpreter [9] but has also been encoded as SMT [1] and ASP problems [16]. We extend MIL to support learning with types. We demonstrate the approach in both Prolog and ASP settings. Farquhar et al. [13] considered adding types to MIL. However, their work is mainly concerned with applying MIL to learn strategies for interactive theorem proving and their work on types is minimal with only two simple types considered.

Types in Logic Programming. The main Prolog [5,38] and ASP [15] implementations do not explicitly support types. There are, however, typed Prolog-like languages, such as the functional-logic language Mercury [34] and the higher-order logic language λProlog [25]. Most work on adding types to logic programming [24,30] is motivated by reducing runtime errors by restricting the range of variables. By contrast, we are motivated by reducing learning times by restricting the range of variables.

3 Framework

3.1 Preliminaries

We assume familiarity with logic programming. We do, however, restate key terminology. We denote the predicate, constant, and function signatures as $\mathcal{P}$, $\mathcal{C}$, and $\mathcal{F}$ respectively. A variable is first-order if it can be bound to a constant symbol, a function symbol, or another first-order variable. A variable is second-order if it can be bound to a predicate symbol or another second-order variable. We denote the sets of first-order and second-order variables as $\mathcal{V}_1$ and $\mathcal{V}_2$ respectively. A term is a variable, a constant symbol, or a function symbol of arity n immediately followed by a bracketed n-tuple of 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. An atom is ground if all of its terms

are ground. We denote as p/n a predicate or function symbol p with arity n. A second-order term is a second-order variable or a predicate symbol. An atom is second-order if it has at least one second-order term. A literal is an atom A (a positive literal) or its negation $\neg A$ (a negative literal). A clause is a disjunction of literals. The variables in a clause are implicitly universally quantified. A Horn clause is a clause with at most one positive literal. A definite clause is a Horn clause with exactly one positive literal. A clause is second-order if it contains a second-order atom. A logic program is a set of Horn clauses. The constant symbols are distinct from the functional symbols, as the latter all have non-zero arity. We call a logic program without proper functional symbols a *datalog* program.

3.2 Meta-interpretive Learning

MIL was originally based on a Prolog meta-interpreter. The key difference between a MIL learner and a standard Prolog meta-interpreter is that whereas a standard Prolog meta-interpreter attempts to prove a goal by repeatedly fetching first-order clauses whose heads unify with a given goal, a MIL learner additionally attempts to prove a goal by fetching second-order metarules, supplied as BK, whose heads unify with the goal. The resulting predicate substitutions are saved and can be reused later in the proof. Following the proof of a set of goals, a logic program is induced by projecting the predicate substitutions onto their corresponding metarules.

We formally define the MIL setting, which we then extend with types. We first define metarules [6]:

Definition 1 ***(Metarule).*** *A metarule is a second-order formula of the form:*

$$\exists \pi \forall \mu \;\; A_0 \leftarrow A_1, \ldots, A_m$$

where $\pi \subseteq \mathcal{V}_1 \cup \mathcal{V}_2$, $\mu \subseteq \mathcal{V}_1 \cup \mathcal{V}_2$, π *and* μ *are disjoint, and each* A_i *is an atom of the form* $p(t_1, \ldots, t_n)$ *such that* $p/n \in \mathcal{P} \cup \pi \cup \mu$ *and each* $t_i \in \mathcal{C} \cup \mathcal{P} \cup \pi \cup \mu$.

When describing metarules, we typically omit the quantifiers and use the more terse notation shown in Fig. 1.

Name	Metarule
indent	$P(A,B) \leftarrow Q(A,B)$
dident	$P(A,B) \leftarrow Q(A,B), R(A,B)$
precon	$P(A,B) \leftarrow Q(A), R(A,B)$
curry	$P(A,B) \leftarrow Q(A,B,R)$
chain	$P(A,B) \leftarrow Q(A,C), R(C,B)$
tailrec	$P(A,B) \leftarrow Q(A,C), P(C,B)$

Fig. 1. Example metarules. The letters P, Q, and R denote existentially quantified second-order variables. The letters A, B, and C denote universally quantified first-order variables.

We define the standard MIL input:

Definition 2 (MIL input). *The MIL input is a triple* (B, E^+, E^-) *where:*

- $B = B_C \cup M$ *where* B_C *is a logic program representing BK and* M *is a set of metarules*
- E^+ *and* E^- *are disjoint sets of ground atoms representing positive and negative examples respectively*

We now define the hypotheses that MIL will consider. Given a set of metarules M, a logic program H is a hypothesis if each clause of H can be obtained by grounding the existentially quantified variables of a metarule in M. This hypothesis space definition enforces a strong inductive bias in MIL.

We define the standard MIL problem:

Definition 3 (MIL problem). *Given a MIL input* $(B_C \cup M, E^+, E^-)$*, the MIL problem is to find a logic program hypothesis* H *such that* $H \cup B_C \models E^+$ *and* $H \cup B_C \not\models E^-$*. We call* H *a solution to the MIL problem.*

3.3 Typed Meta-interpretive Learning

We extend MIL to support types. We assume a finite set $T_b \subseteq \mathcal{C}$ of base types (e.g. *int*, *bool*), a finite set $T_c \subseteq \mathcal{F}$ of polymorphic type constructors (e.g. *list*/1, *array*/1), and a set of type variables V_t. We inductively define a set $\mathcal{T}$ of types:

- if $\tau \in T_b \cup V_t$ then $\tau \in \mathcal{T}$
- if $c/n \in T_c$ and $\tau_1, \ldots, \tau_n \in \mathcal{T}$ then $c(\tau_1, \ldots, \tau_n) \in \mathcal{T}$
- if $\tau_1, \ldots, \tau_n \in \mathcal{T}$ then $(\tau_1, \ldots, \tau_n) \in \mathcal{T}$

The last case concerns types for predicates. For instance $(list(S), list(T), (S, T))$ is the type for the *map/3* predicate. We introduce *typed atoms*:

Definition 4 (Typed atom). *A typed atom is a formula* $p(\tau_1, \ldots, \tau_m, t_1, \ldots, t_m)$*, where* p *is a predicate symbol of arity* n*,* $m + m = n$*,* $\tau_1, \ldots, \tau_m \in \mathcal{T}$*, and each* t_i *is a first-order or second-order term.*

We can extend this notion to logic programs:

Definition 5 (Typed logic program). *A typed logic program is a logic program with typed atoms in place of atoms.*

To aid readability, in the rest of this paper we label each atom with its type. For instance we denote $succ(int, int, A, B)$ as $succ(A, B){:}(int, int)$, and $head(list(T), T, [H|\_], H)$ as $head([H|\_], H){:}(list(T), T)$. Note that the definition of typed logic programs also applies to metarules. For instance, the typed *chain* metarule is:

$$P(A, B){:}(Ta, Tb) \leftarrow Q(A, C){:}(Ta, Tc), R(C, B){:}(Tc, Tb)$$

We define the *typed MIL input*:

Definition 6 (Typed MIL input). *A typed MIL input is a triple* (B, E^+, E^-) *where:*

- $B = B_C \cup M$ *where* B_C *is a typed logic program and* M *is a set of typed metarules*
- E^+ *and* E^- *are disjoint sets of typed ground atoms representing positive and negative examples respectively*

The typed MIL problem easily follows:

Definition 7 (Typed MIL problem). *Given a typed MIL input* $(B_C \cup M, E^+, E^-)$*, the typed MIL problem is to find a typed logic program hypothesis* H *such that* $H \cup B_C \models E^+$ *and* $H \cup B_C \not\models E^-$.

3.4 Hypothesis Space Reduction

We now show that types can improve learning performance by reducing the size of the MIL hypothesis space which in turn reduces sample complexity and expected error. Note that in this section any reference to MIL typically also refers to typed MIL. In MIL, the size of the hypothesis space is a function of the number of metarules m, the number of predicate symbols p, and the maximum program size n. We typically restrict metarules by their body size and literal arity. For instance, the *chain* metarule is restricted to two body literals of arity two. We say that a metarule is in the fragment $\mathcal{M}^i_j$ if it has at most j literals in the body and each literal has arity at most i. By restricting the form of metarules, we can calculate the size of a MIL hypothesis space:

Proposition 1 (MIL hypothesis space [11]). *Given a MIL input with* p *predicate symbols and* m *metarules in* $\mathcal{M}^i_j$*, the number of programs expressible with at most* n *clauses is* $O((mp^{j+1})^n)$.

Proposition 1 shows the MIL hypothesis space grows exponentially both in the size of the target program and the number of body literals in a clause. For simplicity, let us only consider metarules in $\mathcal{M}^2_2$, such as the *chain* metarule. Then the corresponding MIL hypothesis space's size is $O((mp^3)^n)$.

We now consider the advantages of adding types, which we show can improve learning performance when they allow us to ignore irrelevant BK predicates. Informally, given a typed MIL input, a predicate symbol is *type relevant* when it can be used in a hypothesis that is type consistent with the BK and the examples. We define the *relevant ratio* to characterise the reduction of the hypothesis space:

Definition 8 (Relevant ratio). *Given a typed MIL input with* p *predicate symbols where only* p' *are type relevant, the relevant ratio is* $r = p'/p$.

The relevant ratio will always be between 0 and 1 with lower values indicating a greater reduction in the hypothesis space. We characterise this reduction:

Theorem 1 (Hypothesis space reduction). *Given a typed MIL input with p predicate symbols, m metarules in $\mathcal{M}_2^2$, a maximum program size n, and a relevant ratio r, typing reduces the size of the MIL hypothesis space by a factor of r^{3n}.*

Proof. Replacing p with rp in Proposition 1 and rearranging terms leads to $O(r^{3n}(mp^3)^n)$.

Theorem 1 shows that types can considerably reduce the size of hypothesis spaces[2]. The Blumer bound [2] says that given two hypothesis spaces of different sizes, searching the smaller space will result in less error and lower learning times compared to the larger space, assuming the target hypothesis is in both spaces. This result implies that types should improve learning performance, so long as they do not exclude the target hypothesis from the hypothesis space. In this next section we introduce Metagol$_T$ and HEXMIL$_T$ which implement this idea.

4 Metagol$_T$ and HEXMIL$_T$

We present two typed MIL systems: Metagol$_T$ and HEXMIL$_T$, which extend Metagol and HEXMIL respectively.

4.1 Metagol$_T$

Metagol$_T$ is based on an adapted Prolog meta-interpreter. Figure 2 shows the Metagol$_T$ algorithm described as Prolog code. Given a set of atoms representing positive examples, Metagol$_T$ tries to prove each atom in turn. Metagol$_T$ first tries to prove an atom using BK by delegating the proof to Prolog (line 9). Failing this, Metagol$_T$ tries to unify the atom with the head of a metarule (line 16) and to bind the existentially quantified variables in a metarule to symbols in the signature. Metagol$_T$ saves the resulting predicate substitution and tries to prove the body of the metarule. The predicate substitutions can be reused to prove atoms later on (line 11). After proving all atoms, Metagol$_T$ induces a logic program by projecting the predicate substitutions onto their corresponding metarules. Metagol$_T$ checks the consistency of the induced program with the negative examples. If the program is inconsistent, then Metagol$_T$ backtracks to explore different branches of the SLD-tree. Metagol uses iterative deepening to ensure that the first consistent hypothesis returned has the minimal number of clauses. At each depth d, Metagol$_T$ searches for a consistent hypothesis with at most d clauses. At each depth d, Metagol$_T$ introduces d-1 new predicate symbols, formed by taking the name of the task and adding underscores and numbers.

Metagol$_T$ extends Metagol to support types. We annotate each atom with its type using the syntax described in Sect. 3.3. For instance, the following Prolog code denotes an atom with $(list(char), int)$ as its type:

[2] It is not hard too see that Theorem 1 generalizes to a reduction factor of $r^{(j+1)n}$ for any hypothesis space $\mathcal{M}_j^i$.

```
1  learn(Pos,Neg,Prog):-
2      prove(Pos,[],Prog),
3      not(prove(Neg,Prog,Prog)).
4  prove([],Prog,Prog).
5  prove([Atom|Atoms],Prog1,Prog2):-
6      prove_aux(Atom,Prog1,Prog3),
7      prove(Atoms,Prog3,Prog2).
8  prove_aux(Atom:DT:GT,Prog,Prog):-
9      call(Atom:DT:GT).
10 prove_aux(Atom:DT:GT,Prog1,Prog2):-
11     member(sub(Name,GT,Subs),Prog1),
12     unifiable(DT,GT),
13     metarule(Name,Subs,(Atom:DT:GT:-Body)),
14     prove(Body,Prog1,Prog2).
15 prove_aux(Atom:DT:GT,Prog1,Prog2):-
16     metarule(Name,Subs,(Atom:DT:GT:-Body)),
17     prove(Body,[sub(Name,GT,Subs)|Prog1],Prog2).
```

Fig. 2. The Metagol$_T$ algorithm.

```
f([a,b,c],5):(list(char),int)
```

In Fig. 2, each atom and its type is denoted by the variables $Atom{:}DT$. The variable DT represents the *derivation type* of an atom. The derivation type is the type of the values of that atom. When trying to prove an atom, Metagol$_T$ ignores predicates whose derivation types do not match, which allows it to prune the hypothesis space (relative to untyped Metagol). This type check is done through unification. For instance, when trying to prove an atom using BK (line 9), unification ensures that Metagol$_T$ will only call a predicate in the BK if its derivation type matches the derivation type of the atom it is trying to prove. For invented predicate symbols, the derivation type is inferred from the type of the values used to induce that symbol. For instance, suppose we have induced the following theory to explain the above *f/2* atom:

$$f(A,B){:}(list(char),int) \leftarrow f_1(A,C){:}(list(char),int), succ(C,B){:}(int,int)$$
$$f_1(A,B){:}(list(char),int) \leftarrow length(A,C){:}(list(char),int), succ(C,B){:}(int,int)$$

In this theory the derivation type of the invented predicate symbol $f_1/2$ is $(list(char),int)$. Because $f_1/2$ is sufficiently general to be applied to lists of any type, we want to assign it a *general type* that will allow it to be polymorphically reused. For instance, we want the theory to also entail the atom $f([1,2,3,4],6){:}(list(int),int)$. To support polymorphic reuse, we annotate each atom with a second type that denotes the general type of its predicate symbol. The general type is the least general generalisation of the derivation types for an atom. For instance, given the atoms:

$$f([a,b,c],5) : (list(char), int)$$
$$f([1,2,3,4],6) : (list(int), int)$$

We say that $(list(T), int)$ is the general type of $f/2$. When trying to prove an atom using an already invented predicate, line 12 in Fig. 2 checks that the derivation type of atom is an instance of the general type of the invented predicate.

4.2 HEXMIL$_T$

HEXMIL$_T$ extends the forward-chained state-based encoding of HEXMIL [16]. Forward-chained refers to a restricted class of metarules. For brevity we refer the reader to [16] for a full description of HEXMIL. Our main contribution is to extend HEXMIL with types. We do so by augmenting every atom in the ASP encoding with an additional argument that represents the type of that atom. For instance, the untyped successor relation

```
binary_bg(succ,A,B):-B=A+1,state(A).
```

becomes:

```
binary_bg(succ,(int,int),A,B):-B=A+1,state(A,int).
```

We likewise augment all the deduction rules with types.

Our second contribution is to extend the HEXMIL encoding to support learning second-order programs. However, as this extension is not crucial to the claims of this paper we leave a description to future work.

The full typed encoding is available as an online appendix[3].

5 Experiments

We now experimentally[4] examine the effect of adding types to MIL. We test the null hypothesis:

Null Hypothesis 1. *Adding types to MIL cannot reduce learning times.*

To test this null hypothesis we compare the learning times of the typed versus the untyped systems, i.e. Metagol$_T$ versus Metagol, and HEXMIL$_T$ versus HEXMIL.

5.1 Experiment 1: Ratio Influence

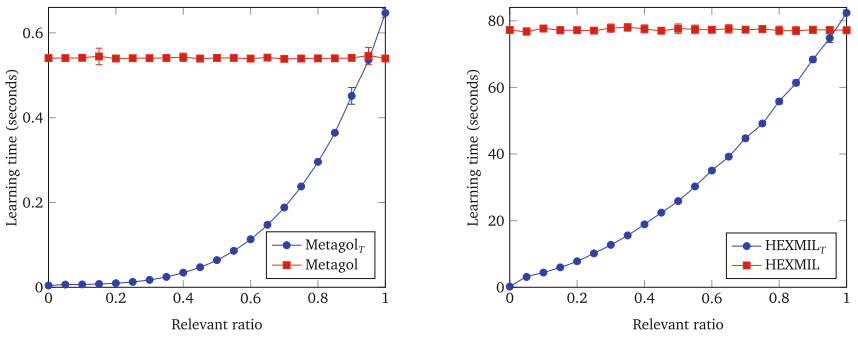
Theorem 1 shows that types can reduce the MIL hypothesis by a cubic factor depending on the relevant ratio (Definition 8), where a lower ratio indicates a greater reduction in the hypothesis space. In this experiment we vary the relevant ratio and measure the effect on learning times. In this experiment there is no solution to the MIL problem. The purpose of the experiment is to measure the time it takes to search the entire hypothesis space.

[3] HEXMIL$_T$ encoding file on https://github.com/rolfmorel/jelia19-typedmil.
[4] Experimental data available at https://github.com/rolfmorel/jelia19-typedmil.

Materials. We use a single positive example $p(1,0) : (int, int)$. We use 20 BK predicates, each a uniquely named copy of the *succ/2* relation, e.g. $succ_1/2$, $succ_2/2, \ldots, succ_{20}/2$. The type of each predicate is either (int, int) or $(\bot, \bot)$, where $\bot$ is a dummy type. We use the *chain* metarule.

Methods. For each relevant ratio rp in $\{0, 0.05, 0.1, \ldots, 1.0\}$ we set the proportion of types (int, int) versus $(\bot, \bot)$ to rp. We consider program hypotheses with at most 3 clauses. We measure mean learning times and standard errors over 10 repetitions. For the HEXMIL experiments, we bound integers to the range 0 to 5000 to ensure the grounding is finite and tractable.

Results. Figure 3 shows that varying the relevant ratio (rp) does not affect the learning times of the untyped systems. By contrast, varying rp affects the learning times of the typed systems. Specifically, types reduce learning times for both typed systems when $rp \leq 0.95$. When rp is 0 the typed systems almost instantly determine that there is no solution. When rp is 0.5, types reduce learning time by approximately 500% with Metagol$_T$ and 300% with HEXMIL$_T$. When rp is 1 the typed systems take slightly longer than their untyped versions because of the small overhead in handling types. The flatter curve of HEXMIL$_T$ compared to Metagol$_T$ is because of implementation differences. The main cost of Metagol$_T$ is trying different predicate substitutions. By contrast, the main cost of HEXMIL$_T$ is grounding the $succ_i/2$ predicates. Overall these results suggest that we can reject the null hypothesis.



(a) Prolog systems (b) ASP systems

Fig. 3. Relevant ratio experiment results.

5.2 Experiment 2: Droplasts

In this experiment we learn a *droplasts* program that takes lists of lists and drops the last element of each sublist. Figure 4 shows examples of this problem. We investigate how varying the amount of BK affects learning time.

```
droplasts([jelia,2019],[jeli,201]).
droplasts([artificial,intelligence],[artificia,intelligenc]).
droplasts([rende,cyprus,madeira,toulouse],[rend,cypru,madeir,toulous]).
```

Fig. 4. Example *droplasts/2* atoms.

Materials. We provide each system with two positive $droplasts(x, y)$ examples where x is the input list and y is the output list. To generate an example, for the input list we select a random integer k between 2 and 5 that represents the number of sublists. We then randomly generate k sublists, where each sublist contains between three and five lowercase characters. The output list is the input list excluding the last element of each sublist. We use small list lengths because of grounding issues with the ASP systems. The Prolog systems can handle much larger values, as previously demonstrated [8]. Figure 5 shows the BK available in the experiments. We always use the *map/3*, *tail/2*, and *reverse/2* predicates, and sample others to include. We use the *chain* and *curry* metarules.

```
tail(A,B):(list(T),list(T)).
map(A,B,F):(list(T),list(S),(S,T)).
reverse(A,B):(list(T),list(T)).
sumlist(A,B):(list(int),int).
head(A,B):(list(T),T).
succ(A,B):(int,int).
last(A,B):(list(T),T).
min_list(A,B):(list(int),int).
pred(A,B):(int,int).
max_list(A,B):(list(int),int).
```

Fig. 5. BK predicates used in the *droplasts* experiment. We omit definitions for brevity.

Methods. For each k in $\{0, 1, \ldots, 25\}$, we uniformly sample with replacement k predicates from those shown in Fig. 5 and generate 2 positive training examples. For each learning system, we learn a *droplasts/2* program using the training examples and BK augmented with the k sampled predicates. We measure mean learning times and standard errors over 10 repetitions.

Results. Figure 6 shows that types reduce learning times in almost all cases. The high variance in the ASP results is mainly because of predicates that operate over integers (e.g. *length/2*), which greatly increase grounding complexity. In all cases both the typed and untyped approaches learn programs with 100% accuracy (plot omitted for brevity). Figure 7 shows an example program learned by Metagol$_T$. The Metagol systems show a clear distinction in the learning times that they require. For the HEXMIL systems intractability prohibits us from running the experiment with the full 25 predicates, though the greater variance and higher mean learning times for the untyped system are already apparent in Fig. 6. These results suggest that we can reject the null hypothesis.

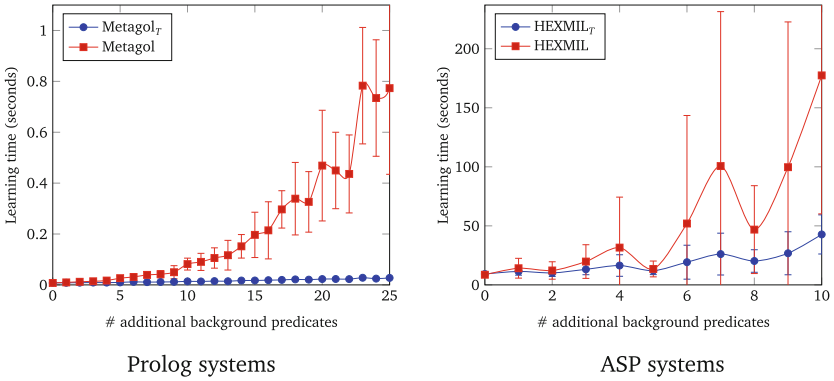


Fig. 6. Droplasts experiment results.

```
droplasts(A,B):(list(list(T)),list(list(T))):-
    map(A,C,droplasts_1):(list(list(T)),list(list(T)),(list(T),list(T))).
droplasts_1(A,B):(list(T),list(T)):-
    reverse(A,C):(list(T),list(T)),
    droplasts_2(C,B):(list(T),list(T)).
droplasts_2(A,B):(list(T),list(T)):-
    tail(A,C):(list(T),list(T)),
    reverse(C,B):(list(T),list(T)).
```

Fig. 7. An example *droplasts/2* program learned by Metagol$_T$. The predicate symbols *droplasts_1/2* and *droplasts_2/2* are invented by Metagol$_T$.

5.3 Experiment 3: More Problems

To further demonstrate that types can improve learning performance, we evaluate the untyped and typed systems on four additional problems:

- *filtercapslower/2* takes a list of characters, discards the lowercase characters, and makes the remaining letters lowercase
- *filterevendbl/2* takes a list of integers, discards the odd numbers, and doubles the even numbers
- *nestedincr/2* takes lists of lists of integers and increments each integer by two
- *finddups/2* takes a list of characters and returns the duplicate character

Materials. As with the previous experiment, we randomly generate examples of varying lengths. We omit full details for brevity. We use the BK from Experiment 2 (Fig. 5) augmented with 14 predicates (Fig. 8), i.e. a total of 24 background predicates. We use the *chain*, *curry*, *dident*, and *tailrec* metarules.

```
filter(A,B,F):(list(T),list(T),(T)).    element(A,B):(list(T),T).
flatten(A,B):(list(list(T)),list(T)).   double(A,B):(int,int).
list_to_set(A,B):(list(T),list(T)).     toupper(A,B):(char,char).
msort(A,B):(list(T),list(T)).           length(A,B):(list(T),int).
tolower(A,B):(char,char).               even(A):(int).
odd(A):(int).                           set(A):(list(T)).
uppercase(A):(char).                    lowercase(A):(char).
```

Fig. 8. Additional BK predicates used in Experiment 3. We omit definitions for brevity.

Methods. For each problem, we supply each system with all 24 BK predicates and 5 positive and 5 negative examples of each problem. We measure mean learning times and standard errors over 10 repetitions. We set a maximum learning time of 10 min.

Results. Figure 9 shows that types can significantly reduce learning times. The accuracy of the Prolog systems is identical in all cases, and is only less than 100% for the *finddups/2* program (4 out of 10 trials learned an erroneous hypothesis). The ASP timeouts are because the grounding is too large when using nested lists, integers, or recursive metarules. Again, the clear distinction in performance of the typed and untyped systems is evidence for rejecting the null hypothesis.

Problem	Metagol	Metagol$_T$	HEXMIL	HEXMIL$_T$
droplasts/2	0.93 ± 0.40	$\mathbf{0.07 \pm 0.03}$	timeout	timeout
filtercapslower/2	0.41 ± 0.21	$\mathbf{0.09 \pm 0.11}$	54 ± 27	10 ± 6
filterevendbl/2	0.34 ± 0.11	$\mathbf{0.06 \pm 0.04}$	timeout	timeout
nestedincr/2	0.67 ± 0.31	$\mathbf{0.05 \pm 0.02}$	timeout	timeout
finddups/2	4.50 ± 5.92	$\mathbf{2.09 \pm 3.13}$	timeout	timeout

Fig. 9. Experiment 3 results that show mean learning times and standard error.

6 Conclusions

We have extended MIL to support types. We have shown that types can reduce the MIL hypothesis space by a cubic factor (Theorem 1). We have introduced two typed MIL systems: Metagol$_T$, which extends Metagol, and HEXMIL$_T$ which extends HEXMIL. Both systems support polymorphic types and the inference of types for invented predicates. We have experimentally demonstrated that types can significantly reduce learning times for both systems.

Limitations and Future Work. Although we have focused on extending MIL with types, our results and techniques should be applicable to other areas of ILP and program induction. Because we declaratively represent types, our techniques should be directly transferable to other forms of ILP that use metarules [1,4,12,32,37]. Future work should study the advantages of using types in these other approaches.

The MIL problem is decidable in the datalog setting [23]. However, because typed MIL supports polymorphic types, which are represented as function symbols, the decidability of the typed MIL problem is unclear. Future work should address this issue. A possible solution involves bounding the function application depth in the type terms while all relevant types for a hypothesis space remain expressible.

We have focused on polymorphic types. A natural extension, which has not been explored in ILP, is to support more complex types, such as refinement types [18].

MIL supports predicate invention so it is sensible to ask whether it can also support *type invention*. For instance, rather than treating strings as list of characters, it would be advantageous to ascribe more precise types, such as *postcode* or *email*. This idea is closely related to the idea of *learning declarative bias* [3].

References

1. Albarghouthi, A., Koutris, P., Naik, M., Smith, C.: Constraint-based synthesis of datalog programs. In: Beck, J.C. (ed.) CP 2017. LNCS, vol. 10416, pp. 689–706. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66158-2_44
2. Blumer, A., Ehrenfeucht, A., Haussler, D., Warmuth, M.: Learnability and the Vapnik-Chervonenkis dimension. J. ACM **36**(4), 929–965 (1989)
3. Bridewell, W., Todorovski, L.: Learning declarative bias. In: Blockeel, H., Ramon, J., Shavlik, J., Tadepalli, P. (eds.) ILP 2007. LNCS, vol. 4894, pp. 63–77. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78469-2_10
4. Campero, A., Pareja, A., Klinger, T., Tenenbaum, J., Riedel, S.: Logical rule induction and theory learning using neural theorem proving. ArXiv e-prints, September 2018
5. Costa, V.S., Rocha, R., Damas, L.: The YAP Prolog system. TPLP **12**(1–2), 5–34 (2012)
6. Cropper, A.: Efficiently learning efficient programs. Ph.D. thesis. Imperial College London, UK (2017)
7. Cropper, A., Muggleton, S.H.: Learning efficient logical robot strategies involving composable objects. In: Yang, Q., Wooldridge, M. (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, 25–31 July 2015, pp. 3423–3429. AAAI Press (2015)
8. Cropper, A., Muggleton, S.H.: Learning higher-order logic programs through abstraction and invention. In: Kambhampati, S. (ed.) Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9–15 July 2016, pp. 1418–1424. IJCAI/AAAI Press (2016)
9. Cropper, A., Muggleton, S.H.: Metagol system (2016). https://github.com/metagol/metagol

10. Cropper, A., Muggleton, S.H.: Learning efficient logic programs. Mach. Learn. 1–21 (2018)
11. Cropper, A., Tourret, S.: Derivation reduction of metarules in meta-interpretive learning. In: Riguzzi, F., Bellodi, E., Zese, R. (eds.) ILP 2018. LNCS, vol. 11105, pp. 1–21. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99960-9_1
12. Evans, R., Grefenstette, E.: Learning explanatory rules from noisy data. J. Artif. Intell. Res. **61**, 1–64 (2018)
13. Farquhar, C., Grov, G., Cropper, A., Muggleton, S., Bundy, A.: Typed meta-interpretive learning for proof strategies. In: CEUR Workshop Proceedings, vol. 1636, pp. 17–32 (2015)
14. Frankle, J., Osera, P., Walker, D., Zdancewic, S.: Example-directed synthesis: a type-theoretic interpretation. In: Bodík, R., Majumdar, R. (ed.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 802–815. ACM (2016)
15. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.T.: Potassco: the Potsdam answer set solving collection. AI Commun. **24**(2), 107–124 (2011)
16. Kaminski, T., Eiter, T., Inoue, K.: Exploiting answer set programming with external sources for meta-interpretive learning. TPLP **18**(3–4), 571–588 (2018)
17. Law, M., Russo, A., Broda, K.: Inductive learning of answer set programs. In: Fermé, E., Leite, J. (eds.) JELIA 2014. LNCS, vol. 8761, pp. 311–325. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11558-0_22
18. Lovas, W., Pfenning, F.: Refinement types for logical frameworks and their interpretation as proof irrelevance. Log. Methods Comput. Sci. **6**(4) (2010)
19. Manna, Z., Waldinger, R.J.: A deductive approach to program synthesis. ACM Trans. Program. Lang. Syst. **2**(1), 90–121 (1980)
20. Muggleton, S.: Inductive logic programming. New Gener. Comput. **8**(4), 295–318 (1991)
21. Muggleton, S.: Inverse entailment and Progol. New Gener. Comput. **13**(3&4), 245–286 (1995)
22. Muggleton, S.H., Lin, D., Pahlavi, N., Tamaddoni-Nezhad, A.: Meta-interpretive learning: application to grammatical inference. Mach. Learn. **94**(1), 25–49 (2014)
23. Muggleton, S.H., Lin, D., Tamaddoni-Nezhad, A.: Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. Mach. Learn. **100**(1), 49–73 (2015)
24. Mycroft, A., O'Keefe, R.A.: A polymorphic type system for Prolog. Artif. Intell. **23**(3), 295–307 (1984)
25. Nadathur, G., Miller, D.: An overview of lambda-PROLOG. In: Kowalski, R.A., Bowen, K.A. (eds.) Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, 15–19 August 1988, vol. 2, pp. 810–827. MIT Press (1988)
26. Osera, P., Zdancewic, S.: 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, 15–17 June 2015, pp. 619–630. ACM (2015)
27. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, New York, NY, USA, pp. 522–538. ACM (2016)

28. Raedt, L.: Declarative modeling for machine learning and data mining. In: Bshouty, N.H., Stoltz, G., Vayatis, N., Zeugmann, T. (eds.) ALT 2012. LNCS, vol. 7568, pp. 12–12. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34106-9_2
29. Ray, O.: Nonmonotonic abductive inductive learning. J. Appl. Log. **7**(3), 329–340 (2009)
30. Schrijvers, T., Costa, V.S., Wielemaker, J., Demoen, B.: Towards typed Prolog. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 693–697. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89982-2_59
31. Schüller, P., Benz, M.: Best-effort inductive logic programming via fine-grained cost-based hypothesis generation - the inspire system at the inductive logic programming competition. Mach. Learn. **107**(7), 1141–1169 (2018)
32. Si, X., Lee, W., Zhang, R., Albarghouthi, A., Koutris, P., Naik, M.: 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, 04–09 November 2018, pp. 515–527. ACM (2018)
33. Singh, R., Gulwani, S.: Synthesizing number transformations from input-output examples. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 634–651. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_44
34. Somogyi, Z., Henderson, F.J., Conway, T.C.: Mercury, an efficient purely declarative logic programming language. Aust. Comput. Sci. Commun. **17**, 499–512 (1995)
35. Srinivasan, A.: The ALEPH manual. Machine Learning at the Computing Laboratory, Oxford University (2001)
36. Stahl, I.: The appropriateness of predicate invention as bias shift operation in ILP. Mach. Learn. **20**(1–2), 95–117 (1995)
37. Wang, W.Y., Mazaitis, K., Cohen, W.W.: 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, 3–7 November 2014, pp. 1199–1208. ACM (2014)
38. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. TPLP **12**(1–2), 67–96 (2012)