Best-Effort Inductive Logic Programming via Fine-grained Cost-based Hypothesis Generation

The Inspire System at the Inductive Logic Programming Competition

Peter Schüller[1] and Mishal Kazmi[2]

[1] Faculty of Engineering, Marmara University, Istanbul, Turkey
`peter.schuller@marmara.edu.tr` / `schueller.p@gmail.com`

[2] Faculty of Engineering and Natural Science, Sabanci University, Istanbul, Turkey
`mishalkazmi@sabanciuniv.edu`

Technical Report: manuscript submitted to *Machine Learning* (Springer).

Abstract

We describe the Inspire system which participated in the first competition on Inductive Logic Programming (ILP). Inspire is based on Answer Set Programming (ASP), its most important feature is an ASP encoding for hypothesis space generation: given a set of facts representing the mode bias, and a set of cost configuration parameters, each answer set of this encoding represents a single rule that is considered for finding a hypothesis that entails the given examples. Compared with state-of-the-art methods that use the length of the rule body as a metric for rule complexity, our approach permits a much more fine-grained specification of the shape of hypothesis candidate rules. Similar to the ILASP system, our system iteratively increases the rule cost limit until it finds a suitable hypothesis. Different from ILASP, our approach generates a new search space for each cost limit. The Inspire system searches for a hypothesis that entails a single example at a time, utilizing a simplification of the ASP encoding used in the XHAIL system. To evaluate ASP we use Clingo. We perform experiments with the development and test set of the ILP competition. For comparison we also adapted the ILASP system to process competition instances. Experimental results show, that Inspire performs better than ILASP, and that cost parameters for the hypothesis search space are an important factor for finding suitable hypotheses efficiently.

1 Introduction

Inductive Logic Programming (ILP) (Muggleton et al., 2015) combines several desirable properties of Machine Learning and Logic Programming: logical rules are used to formulate *background knowledge*, and *examples*, which are feature values plus predicted labels, these are then used to learn a *hypothesis.* A hypothesis is an interpretable set of logical rules which entails the examples with respect to the background knowledge,

Examples can be noisy, sometimes not all examples can be satisfied, and usually there are several possible hypotheses. By defining an appropriate *preference over hypotheses* we can make a trade-off between covering more examples and a lower hypothesis complexity, in other words we can adjust between generalization and overfitting.

The inaugural competition on Inductive Logic Programming (Law et al., 2016) featured a family of ILP tasks about agents that are moving in a grid world. Each instance required to find a hypothesis that represents the rules for valid moves of the agent. Some instances required predicate invention, i.e., finding auxiliary predicates that represent intermediate concepts. For example the ‘Unlocked’ instance required the ILP system to find rules for representing ‘the agent may move to an adjacent cell so long as it is unlocked at that time. A cell is unlocked if it was not locked at the start, or if the agent has already visited the key for that cell.’ (Law et al., 2016). The competition was open to entries for ILP systems based on Prolog and based on Answer Set Programming (ASP) (Brewka et al., 2011; Gebser et al., 2012a; Lifschitz, 2008), and featured a non-probabilistic and a probabilistic track.

In this paper we describe the Inspire system which is based on ASP and predicted 46% of test cases correctly according to official competition results (Law et al., 2016). Inspire was the winner of the non-probabilistic track of the competition, but it was was the only entry to that track. The competition was challenging, because(i) examples using mutually inconsistent possible worlds, which means that the wide-spread approach of representing all examples in one answer set is impossible, (ii) computational resources were limited to 30 sec and 2 GB, which is not much for the intractable ILP task, (iii) negative example information was given implicitly, i.e., agent movements that were not explicitly given as valid had to be considered invalid for learning, and (iv) at the time of the competition, there were no published systems that supported the competition format without the need for significant adaptations.

The Inspire system aims to provide a best-effort solution under these conditions. A central novel aspect of our approach is, that we generate the hypothesis search space using an ASP encoding that permits a fine-grained cost configuration.

In this manuscript we make the following contributions.

- We describe an ASP encoding for generating the hypothesis search space. The encoding permits to attach costs to various aspects of rule candidates. This way the search space exploration can be controlled in a fine-grained way by incrementing a cost limit, and preferences for the shape of rule candidates can be configured easily.
- Learning within that search space is performed on single examples, using a simplification of the Xhail (Ray, 2009) ASP encoding. This is feasible because examples in the competition were noiseless, and a single example was often big enough to learn the full hypothesis.
- Every learned hypothesis is validated on all examples, and if the validation score increased since the last validation, a new prediction attempt is made. This way, even with a timeout and partially correct hypothesis, the system can obtain a partial score.
- For comparison with the state-of-the-art, we created a wrapper around the Ilasp system (Law et al., 2014) for handling competition instances.
- We experimentally compare different cost configurations of the Inspire system with the Ilasp-based system. Our evaluations show that Inspire consistently outperforms Ilasp, and that there are significant score differences among cost configurations.

Fig. 1 depicts the structure of our system and its usage of ASP for all nontrivial computational tasks.

In the following, we will denote by *hypothesis space* the set of rules in the hypothesis search space. The idea to iteratively extend the hypothesis space is present in several existing systems, e.g., in Ilasp. Our novelty is that we do not simply count body literals in rule candidates to measure rule complexity; our approach enables a fine-grained configuration of rule cost based on cost parameters, for example cost for the number of negative body atoms, cost for variables that are bound only once in the rule body, cost for invented predicates used in the rule body, cost for the variables bound only in the rule head, and several further parameters.

We next provide preliminaries of ASP and ILP, and describe the ILP competition format in Section 2, we introduce our hypothesis space generation approach in Section 3, describe the Inspire system's algorithm in Section 4, report on the empirical evaluation in Section 5, give related work in Section 7 and conclude in Section 8 with a link to the open source Inspire system. Appendix A describes secondary modules of the hypothesis search space generation encoding.

2 Preliminaries

2.1 Answer Set Programming

ASP is a logic programming paradigm which is suitable for knowledge representation and finding solutions for computationally (NP-)hard problems (Brewka et al., 2011; Gelfond and Lifschitz, 1988; Lifschitz, 2008). We next give preliminaries of ASP programs with uninterpreted function symbols, aggregates

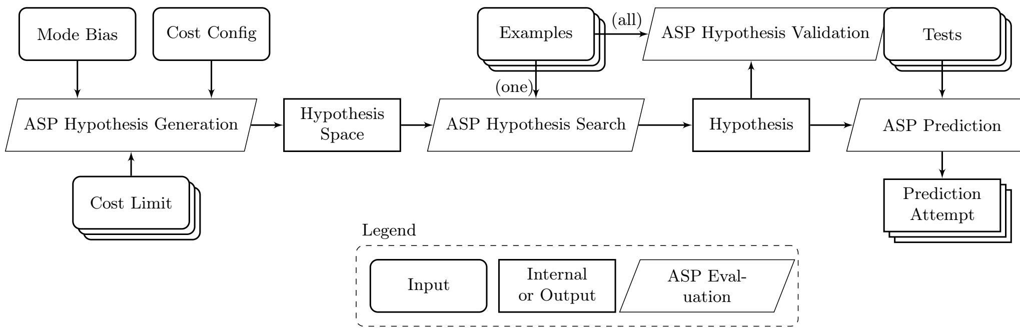

Figure 1: Data flow of the INSPIRE system, showing inputs, outputs, and ASP evaluations. Background Knowledge is implicitly used in all ASP evaluations except in Hypothesis Generation.

and choices. For a more elaborate description we refer to the ASP-Core-2 standard (Calimeri et al., 2012) and to books about ASP (Baral, 2004; Gebser et al., 2012a; Gelfond and Kahl, 2014).

Syntax. Let $\mathcal{C}$ and $\mathcal{V}$ be mutually disjoint sets of *constants* and *variables*, which we denote with first letter in lower case and upper case, respectively. Constants are used for constant terms, predicate names, and names for uninterpreted functions. The set of *terms* $\mathcal{T}$ is recursively defined, it is the smallest set containing $\mathbb{N} \cup \mathcal{C} \cup \mathcal{V}$ as well as tuples of form $(t_1, \dots, t_n)$ and uninterpreted function terms of form $f(t_1, \dots, t_n)$ where $f \in \mathcal{C}$ and $t_1, \dots, t_n \in \mathcal{T}$. An *ordinary atom* is of the form $p(t_1, \dots, t_n)$, where $p \in \mathcal{C}$, $t_1, \dots, t_n \in \mathcal{T}$, and $n \geq 0$ is the *arity* of the atom. An *aggregate atom* is of the form $X = \#agg\{\, t : b_1, \dots, b_k \,\}$ with variable $X \in \mathcal{V}$, aggregation function $\#agg \in \{\#sum, \#count\}$, with $1 < k$, $t \in \mathcal{T}$ and $b_1, \dots, b_k$ a sequence of atoms. A term or atom is *ground* if it contains no sub-terms that are variables.

A *rule* r is of the form $\alpha \leftarrow \beta_1, \dots, \beta_n, \mathbf{not}\,\beta_{n+1}, \dots, \mathbf{not}\,\beta_m$ where $m \geq 0$, α is an ordinary atom, β_j, $0 \leq j \leq m$ is an atom, and we let $B(r) = \{\beta_1, \dots, \beta_n, \mathbf{not}\,\beta_{n+1}, \dots, \mathbf{not}\,\beta_m\}$ and $H(r) = \{\alpha\}$. A *program* is a finite set P of rules. A rule r is a *fact* if $m = 0$.

Semantics. Semantics of an ASP program P are defined using its Herbrand Base HB_P and its ground instantiation $grnd(P)$. An aggregate literal in the body of a rule accumulates truth values from a set of atoms, e.g., $N = \#count\{A : p(A)\}$ is true wrt. an interpretation $I \subseteq HB_P$ iff the extension of $p/1$ in I has size N. Using the usual notion of satisfying a rule with respect to a given interpretation, the FLP-reduct (Faber et al., 2011) fP^I reduces a program P using an answer set candidate I: $fP^I = \{r \in grnd(P) \mid I \models B(r)\}$. Finally I is an answer set of P, denoted $I \in \mathbf{AS}(P)$, iff I is a minimal model of fP^I.

Syntactic Sugar. Anonymous variables of form '_' are replaced by new variable symbols. Choice constructions can occur instead of rule heads, they generate a set of candidate solutions if the rule body is satisfied; e.g., $1\{p(a); p(b)\} \leq 2$ in the rule head generates all solution candidates where at least 1 and at most 2 atoms of the set $\{p(a), p(b)\}$ are true (bounds can be omitted). If a term is given as $X..Y$, where $X, Y \in \mathbb{N}$, then the rule containing the term is instantiated with all values from $\{v \in \mathbb{N} \mid X \leq v \leq Y\}$. A rule with $\alpha = \beta_{n+1}$, and α not occurring elsewhere in the program, is a *constraint*. Constraints eliminate answer sets I where $I \models B(r) \setminus \{\mathbf{not}\,\beta_{n+1}\}$, and we omit α and $\mathbf{not}\,\beta_{n+1}$ for constraints.

ASP supports optimization, which we use to evaluate encodings found in prior work. We denote by $\mathbf{AS}^{opt,1}(P)$ the first optimal answer set of program P. Note that for the purpose of hypothesis space generation, we explicitly represent cost and hard cost limits in answer sets, and we do not use ASP optimization features.

2.2 Inductive Logic Programming

ILP (Muggleton and De Raedt, 1994; Muggleton et al., 2012) is a combination of Machine Learning with logical knowledge representation. A key advantage of ILP is the generation of human-interpretable models. A classical ILP system takes as input a set of examples E, a set B of background knowledge rules, and a set of mode declarations M, also called the mode bias. As output an ILP system produces a set of rules H called the hypothesis which entails E with respect to B in the underlying logic programming formalism. The search for H with respect to E and B is restricted by M, which defines a language that limits the shape of rules that can occur in the hypothesis.

Traditional ILP (Muggleton et al., 2012) searches for sets of Prolog rules which solve a reasoning problem using SLD resolution. ILP for ASP was introduced by Otero (2001) and searches for a set of ASP rules that solves the reasoning problem with respect to ASP semantics in (potentially) multiple answer sets. ASP hypotheses represent knowledge in a more declarative way than Prolog, i.e., without relying on the SLD(NF) algorithm.

Example 1. *Consider the following example ILP instance* (M, E, B) *(Ray, 2009).*

$$M = \left\{ \begin{array}{l} \#\textit{modeh flies(+bird).} \\ \#\textit{modeb penguin(+bird).} \\ \#\textit{modeb } \textbf{not} \textit{ penguin(+bird).} \end{array} \right\}$$

$$E = \left\{ \begin{array}{l} \#\textit{example flies(a).} \\ \#\textit{example flies(b).} \\ \#\textit{example flies(c).} \\ \#\textit{example } \textbf{not} \textit{ flies(d).} \end{array} \right\}$$

$$B = \left\{ \begin{array}{l} \textit{bird(X) :- penguin(X).} \\ \textit{bird(a).} \\ \textit{bird(b).} \\ \textit{bird(c).} \\ \textit{penguin(d).} \end{array} \right\}$$

Based on the above, an ILP system would ideally find the following hypothesis.

$$H = \left\{ \textit{flies(X) :- bird(X), } \textbf{not} \textit{ penguin(X).} \right\}$$

Note that this example does not use the full power of ILP for ASP, i.e., the program $B \cup H$ has only one answer set, and all examples are entailed by this single answer set.

Full-fledged ASP-based ILP is, e.g., used in the Sudoku domain (Law et al., 2014): given(i) a background theory that generates all answer sets of 9-by-9 grids containing digits 1 through 9, (ii) positive examples of partial Sudoku solutions, and (iii) negative examples of partial invalid Sudoku solutions, ILP is used to learn which rules define valid Sudoku solutions. Importantly, the hypothesis can satisfy each positive example in a different answer set, and negative examples must not be satisfied in any answer set.

2.3 First Inductive Logic Programming Competition

The first international ILP competition, held together with the 26th International Conference on Inductive Logic Programming, aimed to test efficiency, scalability, and adaptability, of participating ILP systems (Law et al., 2016). The competition comprised of a probabilistic and a non-probabilistic track. We consider only the non-probabilistic track here.

The initial datasets comprised of 8 problems in each track for entrants to build their system from. The datasets used for scoring systems in the competition were completely new and unseen. All runs were made on an Ubuntu 16.04 virtual machine with a 2 GHz dual core processor and resource limits of 2 GB RAM and 30 sec time.

Instances were in the domain of agents moving in a grid world where only some movements were possible. Each instance consists of a background knowledge, a language bias, a set of examples, and a set of test traces. A trace is a set of agent positions at certain time positions. An example contains a trace and a set of valid moves. The ILP system had to learn the rules for possible moves from examples, and then predict for each test trace whether the agent made only valid moves or not. These predictions were used to produce the final score.

Example 2. *In the* Gaps in the floor *instance, the agent can always move sideways, but can only move up or down in special 'gap' cells which have no floor and no ceiling.*

In the non-OPL transitive links *instance, the agent may go to any adjacent cell or use given links between cells to teleport. It can also use a chain of links in one go. In this problem, the agent has to learn the (transitive) concept 'linked'.*

```
#background
cell((0,0)). cell((0,1)). cell((1,0)). cell((1,1)).
time(0..100). gap((0,0)). link((0,0),(0,1)).
h_adjacent((X1,Y),(X2,Y)) :- cell((X1,Y)), cell((X2,Y)), X2 = X1 + 1.
v_adjacent((X,Y1),(X,Y2)) :- cell((X,Y1)), cell((X,Y2)), Y2 = Y1 + 1.
v_adjacent(F,G) :- v_adjacent(G,F).  h_adjacent(F,G) :- h_adjacent(G,F).

#target_predicate
valid_move(cell, time)
#relevant_predicates
gap(cell) agent_at(cell,time) h_adjacent(cell,cell) v_adjacent(cell,cell)

#Example(0)
#trace
agent_at((0,0),0).  agent_at((0,1),1).
#valid_moves
valid_move((0,1),0) valid_move((1,0),0) valid_move((1,1),1) ...

#Test(0)
#trace
agent_at((0,0),0).  agent_at((0,1),1).  ...
```

Figure 2: Part of instance 6 from the development set of the competition (Law et al., 2016).

Each input instance is structured using the following meta statements:

- **#background** marks the beginning of the background knowledge.
- **#target_predicate** indicates the predicate which should be defined by the hypothesis and is similar to the *modeh* mode declaration in standard language biases.
- **#relevant_predicates** indicates the predicates from the background knowledge which may be used to define the hypothesis and are similar to *modeb* mode declarations in standard language biases.
- **#Example**(**X**) shows the start of an example with identifier **X**. Subsection **#trace** contains the path taken by the agent, and subsection **#valid_moves** gives the complete set of valid moves the agent could take for each time step.
- **#Test**(**X**) contains a test trace with identifier **X**. Subsection **#trace** contains the path taken by the agent.

Predicates in the language bias contained only variable types as arguments, never constants.

System output consists of answer attempts, which start with **#attempt**, followed by a sequence of lines **VALID**(**X**) or **INVALID**(**X**), predicting validity of agent movements in each test traces **X**. Multiple answer attempts were accepted, but only the last one was scored.

Example 3. *A simplification of instance 6 of the development set is given in Figure 2. To conserve space we do not use a separate line for each atom in modes, examples, and tests.*

3 Declarative Hypothesis Generation

A central part of our ILP approach is, that we use an ASP encoding to generate the hypothesis space, which is the set of rules that is considered to be part of a hypothesis. Concretely, we represent the mode bias of the instance as facts, add them to our answer set program, and we add a fact that configures the cost limit for rules in the hypothesis space. Each answer set of this program represents a single rule, the union of these rules is the hypothesis space.

Our ASP encoding uses the following schema for representing a predicate $P(t_1, \dots, t_N)$ with predicate name P, arity N, and argument types $t_1, \dots, t_N$.

pred(I, P, N) represents predicate and arity, where I is a unique ID for predicate and arity,

arg(I, j, t_j) represents the type t_j of argument position j of predicate I.

Example 4. *The predicate* **valid_move(cell, time)***, which was the target predicate in the competition, is represented by the following atoms, where* `p1` *is the unique predicate ID.*

```
pred(p1,valid_move,2).
arg(p1,1,cell).
arg(p1,2,time).
```

3.1 Input Representation

Hypothesis space generation is based on a target predicate and relevant predicates of the instance at hand. We represent this input in atoms of the following form, using above schema:

tpred(I, P, N) for the target predicate,

targ(I, J, T) for arguments of the target predicate,

rpred(I, P, N) for relevant predicates,

rarg(I, J, T) for arguments of relevant predicates, and

type_id(T, ID) for all types T used in the target predicate and in relevant predicates, where ID is the *type ID*, a unique integer associated with T. The set of all type IDs must form a zero-based continuous sequence.

Example 5. *The mode bias* `#target_predicate` *and* `#relevant_predicates` *in Figure 2 is represented by the following facts, where* `t1`*,...,*`r4` *are predicate IDs and* $0, 1$ *are type IDs.*

```
tpred(t1,valid_move,2). targ(t1,1,cell). targ(t1,2,time).
rpred(r1,gap,1).        rarg(r1,1,cell).
rpred(r2,agent_at,2).   rarg(r2,1,cell). rarg(r2,2,time).
rpred(r3,h_adjacent,2). rarg(r3,1,cell). rarg(r3,2,cell).
rpred(r4,v_adjacent,2). rarg(r4,1,cell). rarg(r4,2,cell).
type_id(cell,0).
type_id(time,1).
```

3.2 Output Representation

We represent a single rule per each answer set during hypothesis space generation.

A rule is represented in atoms of the following form:

use_var_type(V, T) represents that the rule uses variable V with type T.

V is a term of form $v(\mathit{Idx})$ denoting variable with index Idx and T is a type as provided in input atoms as first argument of predicate **type_id**. for relevant predicates,

use_head_pred$(\mathit{Id}, \mathit{Pred}, A)$ represents that the rule head is an atom with predicate Pred, arity A, and predicate ID Id.

use_body_pred$(\mathit{Id}, \mathit{Pred}, \mathit{Pol}, A)$ represents that the rule has a body literal of polarity Pol which contains an atom with predicate Pred, arity A, and literal identifier Id. Importantly, Id uniquely refers to multiple body literals with the same Pol, Pred, and A.

bind_hvar(J, V) represents that the argument position J in the rule head contains variable V, where V is a term of form $v(\mathit{Idx})$.

bind_bvar$(\mathit{Id}, \mathit{Pol}, J, V)$ represents that the argument position J of the rule body literal with ID Id and polarity Pol contains variable V, where Id refers to a unique body literal as represented in the first argument Id of **use_body_pred**.

Example 6. *The hypothesis candidate*

```
valid_move(V5,V10) :- cell(V5), time(V10), not agent_at(V5,V10).
```

is represented in an answer set by the following atoms.

```
use_var_type(v(5),cell)
use_var_type(v(10),time)
use_head_pred(t1,valid_move,2)
bind_hvar(1,v(5))
bind_hvar(2,v(10))
use_body_pred(id_idx(r2,1),agent_at,neg,2)
bind_bvar(id_idx(r2,1),neg,1,v(5))
bind_bvar(id_idx(r2,1),neg,2,v(10))
```

Lines 1–2 represent variables and their types, lines 3–5 represent the rule head, and lines 6–8 represent the body literal. Predicates of variable types are implicitly given by **use_var_type***.*

3.3 Cost Configuration

For fine-grained control over the shape of rules in the hypothesis space, we define several cost components on rules generated by the above ASP encoding. In addition to costs, it is also necessary to impose hard limits on the hypothesis space.

We use the following hard limits that restrict the shape of rules in the hypothesis search space, where default values are given in brackets:

maxvars (4) Maximum number of variables per type. This limits how many variables of a single type can occur simultaneously in one rule.

maxuseppred (2) Maximum occurrence of a single predicate as a positive body literal. This limits how often we can use the same predicate in the positive rule body. For example for obtaining a transitive closure in the hypothesis space, this value needs to be at least 2, and **maxvars** needs to be at least 3.

maxusenpred (2) Maximum occurrence of a single predicate as a negative body literal.

maxliterals (4) Maximum number of overall literals in a rule. This is a hard limit on the size hypothesis rule bodies.

maxinventpred (1) Maximum number of predicates to be invented.

inv_minarity (2) Minimum arity of invented predicates.

inv_maxarity (2) Maximum arity of invented predicates.

Hard limits determine whether a suitable hypothesis can be found at all, moreover they determine the size of the instantiation of the ASP encoding, which influences the efficiency of enumerating answer sets.

For configuring fine-grained costs on various aspects of rules, we use the following cost parameters (defaults are again in brackets):

free_vars (2) Number of variables that do not incur cost.

cost_vars (1) Cost for each variable beyond **free_vars**.

cost_type_usedmorethantwice (2) Cost for each usage of a type beyond the second usage. For example, this cost is incurred if we use three variables of type **time** in one rule.

cost_posbodyliteral (1) Cost for each positive body literal.

cost_negbodyliteral (2) Cost for each negative body literal. The default value is higher than the one for **cost_posbodyliteral**, because usually programs have more positive body literals than negative body literals.

cost_pred_multi (2) Cost for each repeated usage of a predicate in the rule body beyond the first usage.

cost_varonlyhead (5) Cost for each variable that is used only in the head. Rules with such variables are still safe because each variable has a type. It is possible, but rare, that such rules are useful, so they obtain high cost.

cost_varonlyoncebody (5) Cost for each variable that is not used in the head and used only once in the body of the rule. Such variables are like anonymous variables and project away the argument where they are bound. We expected such cases to be rare so we incur a high default cost.

cost_var_boundmorethantwice (2) Cost for each variable that is bound in more than two literals.

cost_reflexive (5) Cost for each binary atom in the rule body with the same variable in both arguments.

cost_inv (2) Cost for inventing any type of predicate. This cost is used to adjust, above which cost limit predicate invention is performed, independent from the cost of inventing each predicate.

cost_inv_pred (2) Cost for each invented predicate.

cost_inv_headbody (3) Cost for using the same invented predicate both in head and body.

cost_inv_bodymulti (5) Cost for using multiple invented predicates in the rule body.

cost_inv_headbodyorder (5) Cost for using an invented predicates in the head and a different invented predicate in the body of a rule, in a way that the head predicate is lexically greater than the body predicate. This incurs higher cost to programs with cycles over invented predicates, and less cost to those that have no such cycles.

The above costs can independently be adjusted and influence performance of our approach. Costs determine the stage of the ILP search at which a certain rule will be used as a candidate for the hypothesis. Adjusting these costs will not change correctness, i.e., whether a hypothesis can be found at all.

Example 7. *The hypothesis candidate shown in Example 6 has a cost of 2 using the default cost settings, because of one cost component from* **cost_negbodyliteral**.

3.4 Main Encoding

The main encoding for generating the hypothesis space is given in Fig. 3. Hard limits and cost parameters are added to this encoding as constant definitions.

We define distinct typed variables in atoms of form $\mathbf{var\_type}(v(\mathit{Index}), T)$ in line 1. Such an atom represents, that the variable of form $v(\mathit{Index})$ has type T, where Index is a running index over all variables. This defines **maxvars** variables of each type. Note that we use $v(\mathit{Idx})$ to enable a later extension of our encodings with constant strings as arguments in the mode bias. Constant strings were not required for the ILP competition.

Head predicates are represented as **hpred** and **harg**, and body predicates are represented as **bpred** and **barg** in lines 2–5. These are defined from target predicate and relevant predicate, respectively. We define further head and body predicates in the encoding for invented predicates (see Section A.3).

We guess how many variables (up to **maxvars**) are in the rule in the current answer set candidate (line 6). We guess which concrete variables (including their type) are in the rule (line 7). Each variable V with type T is represented as $\mathbf{use\_var\_type}(V, T)$.

We represent unique placeholders for all potentially existing literals in the rule body in lines 8–9 according to given hard limits. Such placeholders are represented in atoms of the form $\mathbf{body\_pred}(\mathit{ID}, \mathit{Pred}, \mathit{Pol}, A)$ where ID is a unique term built for that predicate from its predicate ID Id and a running index Idx, Pred is the predicate name, Pol the polarity, and A the arity of the predicate.

A subset of these placeholders is guessed as a body literal in lines 10–11, up to a maximum of **maxliterals** literals. A guess in line 12 determines the head predicate. So far the program represents

```
 1  var_type(v(ID*maxvars+Idx),T) :- type_id(T,ID), Idx=0..(maxvars-1).

 2  hpred(I,P,N) :- tpred(I,P,N).
 3  harg(I,J,T) :- targ(I,J,T).
 4  bpred(I,P,N) :- rpred(I,P,N).
 5  barg(I,J,T) :- rarg(I,J,T).

 6  1 { varcount(1..maxvars) } 1.
 7  VC { use_var_type(V,T): var_type(V,T) } VC :- varcount(VC).

 8  body_pred(id_idx(Id,Idx),Pred,pos,Arity) :- bpred(Id,Pred,Arity), Idx = 1..maxuseppred.
 9  body_pred(id_idx(Id,Idx),Pred,neg,Arity) :- bpred(Id,Pred,Arity), Idx = 1..maxusenpred.

10  1 { use_body_pred(id_idx(Id,Idx),Pred,Polarity,Arity)
11        : body_pred(id_idx(Id,Idx),Pred,Polarity,Arity) } maxliterals.
12  1 { use_head_pred(Id,Pred,NArgs) : hpred(Id,Pred,NArgs) } 1.

13  1 { bind_hvar(Pos, VId) : use_var_type(VId,Type) } 1 :-
14        use_head_pred(PredId,_,_), harg(PredId,Pos,Type).
15  1 { bind_bvar(id_idx(PredId,Idx),Polarity,Position,VId) : use_var_type(VId,Type) } 1 :-
16        use_body_pred(id_idx(PredId,Idx),_,Polarity,_), barg(PredId,Position,Type).

17  hbound_var(VId) :- bind_hvar(_,VId).
18  bbound_var(VId) :- bind_bvar(_,_,_,VId).

19  :- use_var_type(VId,_), not hbound_var(VId), not bbound_var(VId).
```

Figure 3: Main module for ASP Hypothesis Generation.

variables including their type, which variables are going to be used, and which head and body predicates to use as literals.

A guess which of these variables is bound to which argument position of which head or body predicate in lines 13-16. The limits of these choice rules require, that each position is bound exactly once. Moreover the conditions within the choice ensures, that variables are bound to argument positions of the correct type. We represent which variables are bound in the head and in the body of the rule in lines 17-18 (this separation is used for cost representations). Finally, an answer set where a variable is used but neither bound to the head nor to the body of the rule, is eliminated by constraint in line 19. We do not forbid 'unsafe rules' (where a variable exists only in the head of a rule) because all variables are typed and therefore have an implicit domain predicate in the body.

If we evaluate this program module together with a mode bias given as facts according to Section 3.1 and together with constant definitions of hard limits according to Section 3.3, we obtain answer sets that represent single rules according to Section 3.2, and according to the given mode bias and hard limits.

To give an easier overview of our approach, we only give the main encoding for hypothesis generation here. Further program modules for eliminating redundant answer sets (symmetry breaking), for representing and restricting the cost of rule candidates, and for predicate invention, are described in Appendix A.

4 Best-Effort Learning and Prediction

Algorithm 1 shows the main algorithm of our entry to the ILP competition which has been visualized from a conceptual point of view in Figure 1. We next describe the algorithm. Major design decisions are discussed in Section 6.

The algorithm obtains one competition instance (see Section 2.3) as input: background knowledge *bk* is a set of ASP rules, the set of examples e is of form $\langle \mathit{trace}, \mathit{label} \rangle$ where *trace* is a set of atoms for predicate **agent_at** and *label* is a set of atoms for predicate **valid_move**, the mode bias m is given in the form of target predicate and relevant predicates, and finally the set of test traces *tests* is of form $\langle \mathit{trace}, \mathit{id} \rangle$ where *trace* is a set of atoms for predicate **agent_at** and *id* is required for labeling prediction outputs with the correct trace.

Initially, Algorithm 1 sorts examples by length of their trace. The variable *bestquality*, initially zero,

Algorithm 1: Inspire-ILP(KB bk, Examples e, Mode-bias m, Test-traces $tests$)

```
1  Sort examples $e$ by length of trace                       // process smaller examples first
2  $bestquality := 0$
3  $besthypothesis := null$
4  for $\langle trace, label\rangle \in e$ do                 // process sorted examples one by one
5      for $climit = 4, \dots, 15$ do
6          $ha := \mathbf{AS}(P_{hypgen}(m, climit))$
7          $hspace :=$ rules extracted from answer sets $ha$ as described in Section 3.2
8          $P_{hs} := bk \cup trace \cup \{P_{rule}(r) \mid r \in hspace\} \cup P_{verify}(label)$   // build search program
9          $h := \mathbf{AS}^{opt,1}(P_{hs})$                // Hypothesis search and optimization in ASP
10         if $h$ exists then                                 // found a hypothesis for this example
11             $quality := \big|\{\langle etrace, elabel\rangle \in e \mid elabel$ is exactly reproduced in $\mathbf{AS}(bk \cup h \cup etrace)\}\big|$
12             if $quality > bestquality$ then
13                 $bestquality := quality$
14                 $besthypothesis := h$
15                 Print "#attempt"                           // best-effort output
16                 for $\langle testtrace, testid\rangle \in tests$ do
17                     if all agent movements in $\mathbf{AS}(bk \cup h \cup testtrace)$ are predicted as valid then
18                         Print "VALID($testid$)"
19                     else
20                         Print "INVALID($testid$)"
21                 if $quality = |e|$ then                    // $h$ predicts all examples in $e$ correctly
22                     return $h$
23 return $besthypothesis$
```

stores the number of examples that we can predict correctly with the best hypothesis found so far. The loop in line 4 iterates over the sorted examples, starting with the smallest. For each example, the loop in line 5 iterates over cost limit values starting at 4 up to 15. (In a non-competition setting, this would be $1, \dots, \infty$.) For each cost limit *climit*, in line 6 we enumerate all answer sets of $P_{hypgen}(m, climit)$ which denotes the ASP encodings for hypothesis generation as described in Fig. 3, 4, 5, and 6 (see Appendix A), with additional input facts derived from mode bias m as described in Section 3.1 and constant **maxcost** set to value *climit*. In line 7 each answer set is transformed into a (nonground) rule which yields the hypothesis search space *hspace*.

Given *hspace*, we search for an optimal hypothesis using the current example's *trace* and *label*. A hypothesis $h \subseteq hspace$ must predict the extension of **valid_move** in the label correctly with respect to background knowledge *bk* and *trace*. Note that predicate **valid_move** is specific to the competition, however our encodings are flexible with respect to using different or even multiple predicates. A correct hypothesis is optimal, if it has lower or equal cost compared with all other correct hypotheses, where cost is the sum of costs of rules used in the hypothesis, and cost of a single rule is computed according to Section 3.3.

Hypothesis search is done using ASP optimization on a program P_{hs} whose encoding is a simplification of the Inductive Phase encoding used in Xhail (Ray, 2009, 3.3). Briefly, P_{hs} contains background knowledge, the current example's trace, and all rules r in the hypothesis space, transformed via $P_{rule}(r)$, and a module $P_{verify}(label)$ which eliminates solutions that do not satisfy the example. $P_{rule}(r)$ contains(i) the original rule r with an additional body condition $use(r)$, (ii) a guess whether $use(r)$ is true, and (iii) a weak constraint that incurs the cost of r if $use(r)$ is true. The optimal answer sets of P_{hs} represent the subset of rules $h \subseteq hspace$ that entails the given example $\langle trace, label\rangle$ such that there is no $h' \subseteq hspace$ that also entails the given example with smaller cost.

If such a hypothesis h exists, in line 11 we measure the quality of h by testing how many of the given examples e are correctly predicted by h. This test is performed by repeatedly evaluating an ASP program on *bk*, h, and the *trace* of the respective example. If the obtained quality is higher than the best previously obtained quality, in lines 15–20 we store quality and hypothesis and produce a prediction attempt for all test traces. Prediction is performed by evaluating an ASP program on *bk*, h, and each trace. If we have correctly predicted the labels of all training examples e, we immediately return the hypothesis after the prediction attempt (because we have no possibility to measure hypothesis improvements beyond this point). In case we do not find a hypothesis that predicted all examples correctly we return the currently known best hypothesis. If we do not find any hypothesis, we return *null*.

For the competition, it was only required that the system makes prediction attempts. To provide a more general approach, our algorithm also returns a hypothesis.

5 Evaluation

The INSPIRE system is theoretically based on Sections 3 and 4. The implementation is based on Python and the ASP system CLINGO 5 (which consists of the instantiator GRINGO (Gebser et al., 2011) and the solver CLASP (Gebser et al., 2012b)). CLINGO is used for all ASP evaluations shown in Fig. 1, for optimal performance of ASP Hypothesis Search, i.e., evaluating $\mathbf{AS}^{opt,1}(P_{hs})$ in Algorithm 1, it is important to activate unsat-core optimization (Andres et al., 2012) and stratification (Alviano et al., 2015; Ansótegui et al., 2013) in CLINGO.

5.1 Comparison System based on ILASP

As there was no other participant in the competition, and as the competition used a novel input format, there was no state-of-the-art system we could use for comparison. Therefore we adapted the state-of-the-art ILASP system (Law et al., 2014) version 2.6.2, using a wrapper script. Our wrapper converts target and relevant predicates into mode bias commands, adds a rule that guesses zero or one position of the agent for each time point (this guess is required due to the way ILASP represents examples, see below), and converts each example into a positive ILASP example. A positive ILASP example is a partial interpretation, which is a pair $\langle E^{inc}, E^{exc}\rangle$ of sets of atoms. Such an example is counted as entailed by a hypothesis H with respect to a background knowledge B if the set of atoms E^{inc} is *included* in a witnessing answer set $I \in \mathbf{AS}(B \cup H)$, and if that answer set I does not contain any atom from the set E^{exc} of *excluded* atoms (Otero, 2001). For each example in the competition input, we assemble E^{inc} from trace and valid moves. The exclusion list was specified implicitly in the competition: if a move was not given as valid, it was invalid. However ILASP requires an explicit input E^{inc}, therefore we compute (using CLINGO and *bk*) the set of invalid moves and use these nonvalid moves for E^{exc}.

Example 8. *Trace and valid moves from* **#Example(0)** *in Fig. 2 are converted into the following* ILASP *example.*

```
#pos(ex0, {agent_at((0,0),0), agent_at((0,1),1),
           valid_move((0,1),0), valid_move((1,0),0), valid_move((1,1),1)},
          {valid_move((1,1),0), valid_move((0,0),0),
           valid_move((0,0),1), valid_move((0,1),1), valid_move((1,0),1)}).
```

To increase ILASP performance, we generate invalid atoms only for those time points where an agent position exists in the trace (most competition examples define 100 time points but use less than 20). We configure ILASP with `#maxv(3)` and `#max_penalty(100)` because lower values did not yield any hypothesis and higher values yielded worse performance.

In summary, we did our best to make ILASP perform well.

5.2 Results

We performed experiments on the development (D, 33 instances) and test (T, 45 instances) sets which were provided on the ILP competition homepage (Law et al., 2016). Runs were performed for the resource limits of the competition (30 sec, 2 GB RAM) and for higher resource limits (600 sec, 5 GB).

Table 1 shows experimental results for the ILASP system, for the INSPIRE system using default configuration values as given in Section 3.3, and for three variations of the INSPIRE system: INSPIRE$^{N\text{-}}$ where we consider fewer negative body literals by setting **cost_negbodyliteral** = 3 (instead of 2), INSPIRE$^{I\text{-}}$ where we consider fewer invented predicates by setting **cost_inv** = **cost_inv_pred** = 3 (instead of 2), and INSPIRE^{I+} where we consider more invented predicates by setting **cost_inv** = **cost_inv_pred** = 1. (Here, fewer and more compares the parameters with respect to a certain cost limit during hypothesis

Table 1: Experimental Results for Development Set (D, 33 instances) and Test Set (T, 45 instances).

			Timeout		Accuracy		Attempts	T (sec)	M (MB)
DS	Resources	System	#	%	#	%	#	avg	avg
D	30 sec 2 GB	INSPIRE	18	54.5	15.0	45.5	54	17.5	127.2
		INSPIRE$^{N\text{-}}$	**16**	**48.5**	**17.0**	**51.5**	**58**	**17.3**	122.2
		INSPIRE$^{I\text{-}}$	18	54.5	15.0	45.5	52	17.4	**105.7**
		INSPIRE^{I+}	21	63.6	12.0	36.4	49	20.3	232.1
		ILASP	30	90.9	3.0	9.1	36	28.9	205.3
	600 sec 5 GB	INSPIRE	1	3.0	22.6	68.5	65	117.5	527.8
		INSPIRE$^{N\text{-}}$	**0**	**0.0**	**23.4**	**70.9**	**67**	**68.7**	**371.0**
		INSPIRE$^{I\text{-}}$	1	3.0	22.6	68.5	63	92.2	439.7
		INSPIRE^{I+}	8	24.2	18.0	54.5	58	232.1	674.5
		ILASP	4	12.1	14.8	44.8	55	243.3	2543.7
T	30 sec 2 GB	INSPIRE	27	60.0	15.0	33.3	63	18.6	117.4
		INSPIRE$^{N\text{-}}$	27	60.0	15.0	33.3	63	18.6	102.2
		INSPIRE$^{I\text{-}}$	**24**	**53.3**	**18.0**	**40.0**	**66**	**18.1**	114.2
		INSPIRE^{I+}	29	64.4	13.0	28.9	61	20.0	136.9
		ILASP	42	93.3	0.0	0.0	45	28.1	**61.7**
	600 sec 5 GB	INSPIRE	4	8.9	19.3	42.8	72	172.6	797.6
		INSPIRE$^{N\text{-}}$	**0**	**0.0**	22.8	50.6	**79**	**105.3**	**526.8**
		INSPIRE$^{I\text{-}}$	2	4.4	**22.9**	**50.8**	74	136.1	600.8
		INSPIRE^{I+}	8	17.8	19.1	42.4	74	244.4	1067.6
		ILASP	13	28.9	15.2	33.8	73	360.7	2550.2

space generation.) The table shows the number of timeouts (i.e., the system did not terminate within the time limit) both in absolute and in percent (lower is better); the accuracy of the last attempt for predicting test traces (this value contains fractions if a part of the traces of some instance was predicted correctly); the number of attempts performed, and the average time T and memory usage M over all runs.

Results show that INSPIRE$^{N\text{-}}$ has best performance on the development set with 51 % and 70.9 % prediction accuracy for low and high resource limits, respectively. INSPIRE$^{I\text{-}}$ has best performance on the test set with 40.0 % and 50.8 % accuracy, respectively, although it has more timeouts and fewer attempts than INSPIRE$^{N\text{-}}$ for high resource limits. INSPIRE^{I+} generally performs worse than the other tested INSPIRE-based systems.

ILASP a high number of timeouts for low resource limits, and with high resource limits ILASP produces an accuracy comparable with INSPIRE-based systems using low resource limits. While ILASP can learn from multiple examples at once, it does not support automatic predicate invention.[1] This might explain the lower accuracy and the low number of prediction attempts (predictions are performed only if ILASP returns some hypothesis). If we compare on an instance-by-instance basis (not shown in the table), we find that four instances obtain a partially correct hypothesis only from ILASP and no hypothesis from INSPIRE. This shows the benefit of learning complex structural properties that are distributed across multiple examples.

Overall the INSPIRE system clearly outperforms ILASP for both resource limits. The performance difference is clearer for low resource settings. For longer timeouts, ILASP uses significantly more memory which is not surprising given that it processes all examples at once. Note that the memory limit was never exceeded in our experiments.

[1]Invented predicates can manually be added in a manual specification of the search space, however we provided ILASP only with a mode bias.

6 Discussion

Our approach uses a hypothesis search space of stepwisely increasing complexity. This is a commonly used approach in ILP for investigating more likely hypotheses first. However, usually a very coarse-grained measure of rule complexity is used, such as the number of body atoms in a rule. We extend this idea by using an ASP encoding that provides a fine-grained, flexible, easily adaptable, and highly configurable way of describing hypothesis cost and for controlling the search space. Experiments show the importance of choosing the right cost parameters for the application at hand, in order to explore more promising areas of the search space first.

Enumerating answer sets of our hypothesis space generation encodings in Algorithm 1 line 6 is computationally cheap compared with hypothesis search (line 9). This method for hypothesis space generation can be taken independent from other parts of our approach and reused in another ILP system for configuring the hypothesis search space.

We support predicate invention. Invented predicates with arity one are implicitly created by hypotheses containing only reflexive usage of an invented predicate **inv**, i.e., hypotheses containing only $\mathbf{inv}(V, V)$ for some variable V. We think, that a better solution is possible.

The INSPIRE system learns from single examples and sorts them by trace length, which reduces resource consumption in the hypothesis search step. This is a strategy we chose specific to the competition: we observed, that even short examples often provide sufficient structure for learning the final hypothesis, moreover competition examples were noiseless. We think that future ILP competitions should prevent success of such a strategy by using instances that require learning from all examples at once, e.g., by providing smaller, partial, or noisy examples (even in the non-probabilistic track).

Testing whether a hypothesis correctly predicts an input example is computationally cheap. Therefore, for each newly found hypothesis we perform this check on all examples. If this increases the amount of correctly predicted examples, we make a prediction attempt on the test cases. If all examples were correctly predicted, we terminate the search, because we have no metric for improving the hypothesis after predicting all examples correctly (competition examples are noiseless and our search finds hypotheses with lower cost first).

A major trade-off in our approach is the blind search: we avoid to extract hypotheses from examples as done in the XHAIL (Ray, 2009) and ILED (Katzouris et al., 2015) systems. This means we rely on the mode bias and on our incrementally increasing cost limit to obtain a reasonably sized search space for hypothesis search. The ILASP (Law et al., 2014) system also uses blind search and a search space of increasing complexity, however ILASP uses the number of body literals in rules as a measure of hypothesis complexity, hence the search space of ILASP grows significantly faster than in our appraoch.

Note that we did not compare INSPIRE with XHAIL (Ray, 2009), MIL (Muggleton et al., 2014), or ILED (Katzouris et al., 2015), because these approaches do not support examples that are based on multiple possible worlds (Otero, 2001) which is required to solve competition instances.

7 Related Work

Inductive Logic Programming (ILP) is a multidisciplinary field and has been greatly impacted by Machine Learning (ML), Artificial Intelligence (AI) and relational databases. Quite a few surveys (Gulwani et al., 2015; Muggleton et al., 2012) mention about the systems and applications of ILP in interdisciplinary areas. Important theoretical foundations of ILP comprise Inverse Resolution and Predicate Invention (Muggleton, 1995; Muggleton and Buntine, 1992).

Most ILP research has been based on Prolog and aimed at Horn programs that exclude Negation as Failure which provides monotonic commonsense reasoning under incomplete information. Recently, research on ASP-based ILP (Law et al., 2014; Otero, 2001; Ray, 2009) has made ILP more declarative (no necessity for cuts, unrestricted negation) but also introduced new limitations (scalability, predicate invention). Predicate invention is indeed a distinguishing feature of ILP: Dietterich et al. (2008) writes that 'without predicate invention, learning always will be shallow'. Predicate invention enables learning an explicit representation of a 'latent' logical structure that is neither present in background knowledge nor in the input, which is related to successful machine learning methods such as Latent Dirichlet Allocation (Blei et al., 2003) and hidden layers in neural networks (LeCun et al., 2015). Muggleton et al. (2015) recently introduced a novel predicate invention method and, to the best of our knowledge for the first time, compared ASP and Prolog-based ILP Other ASP-based ILP solvers do not support predicate invention, or they support it only with an explicit specification of rules involving invented predicates (Law et al., 2014). In purely Prolog-based ILP, several systems with predicate invention have been built (Craven, 2001; Flach, 1993; Muggleton, 1987), however these systems also do not support the full power of ASP-based ILP with examples in multiple answer sets (Otero, 2001). Note that predicate invention in general is still considered an unsolved and very hard problem (Muggleton et al., 2012).

A recent application of ASP-based ILP was done by Mitra and Baral (2016), who perform Question Answering on natural language texts, based on statistical NLP methods to gather knowledge required for learning with ILP how to answer questions similar to a given training set. They used Xhail (Ray, 2009) to learn non-monotonic hypotheses in a formalization of an agent theory with events.

8 Conclusion

We created the Inspire Inductive Logic Programing system which supports predicate invention and generates the hypothesis search space from the given mode bias using an ASP encoding. This encoding provides many parameters for a fine-grained control over the cost of rules in the search space. The system generates search spaces with incrementally increasing cost, until a hypothesis can be found with respect to the given examples.

Performance of our system is provided by(i) appropriately chosen cost parameters for the shape of rules in the hypothesis search space, (ii) incremental exploration of the search space, (iii) an algorithm that learns from a single example at a time, and (iv) the usage of modern ASP optimization techniques for hypothesis search.

On competition instances, Inspire clearly outperforms Ilasp, which we adapted to the competition instance format.

While Inspire was created specific for the first ILP competition, it can be generalized to become a generic ILP system. Moreover, the fine-grained hypothesis search space generation is independent from our learning algorithm, and could be integrated into other systems, for example into Ilasp.

Hypothesis candidates are generated in a blind search, i.e., independent from examples. This might seem like a bad choice, however it is a viable option in ASP-based ILP because we found in recent research (under review) that existing methods for non-blind search have major issues wich make their usage problematic: the Xhail algorithm (Ray, 2009) produces many redundancies in hypothesis generation, leading to a very expensive search (Induction), while the Iled algorithm (Katzouris et al., 2015) is unable to handle even small inconsistencies in input data, leading to mostly empty hypotheses or program aborts.

We conclude that the good performance of the Inspire system is based partially on our novel fine-grained hypothesis search space generation, and partially on the usage of pecularities of competition instances. To advance the field of Inductive Logic Programming for Answer Set Programming, future competitions should use more diverse instances that disallow finding solutions from single examples, while at the same time requiring the hypothesis for separate examples to be entailed in separate answer sets. To avoid blind search, it will be necessary to improve algorithms and systems to make them resistant to noise and scalable in the presence of large amounts of training examples. To that end, we think that theoretical methods developed in Prolog-based ILP, for example (Muggleton et al., 2015), will be important and useful for advancing ASP-based ILP.

The Inspire system and the Ilasp wrapper are open source software and publicly available at `https://bitbucket.org/knowlp/inspire-ilp-comp` .

Acknowledgements

This work has been supported by The Scientific and Technological Research Council of Turkey (TUBITAK) under grant agreement 114E777.

References

Alviano, M., Dodaro, C., Marques-Silva, J., and Ricca, F. (2015). Optimum stable model search: algorithms and implementation. *Journal of Logic and Computation*, exv061.

Andres, B., Kaufmann, B., Matheis, O., and Schaub, T. (2012). Unsatisfiability-based optimization in clasp. In *International Conference on Logic Programming (ICLP), Technical Communications*, pages 212–221.

Ansótegui, C., Bonet, M. L., and Levy, J. (2013). SAT-based MaxSAT algorithms. *Artificial Intelligence*, 196:77–105.

Baral, C. (2004). *Knowledge Representation, Reasoning, and Declarative Problem Solving.* Cambridge University Press.

Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent Dirichlet Allocation. *Journal of Machine Learning*, 3:993–1022.

Brewka, G., Eiter, T., and Truszczynski, M. (2011). Answer set programming at a glance. *Communications of the ACM*, 54(12).

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F., and Schaub, T. (2012). ASP-Core-2 Input language format. Technical report, ASP Standardization Working Group.

Craven, M. (2001). Relational learning with statistical predicate invention. *Machine Learning*, pages 97–119.

Dietterich, T. G., Domingos, P., Getoor, L., Muggleton, S., and Tadepalli, P. (2008). Structured machine learning: The next ten years. *Machine Learning*, 73(1):3–23.

Faber, W., Pfeifer, G., and Leone, N. (2011). Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278–298.

Flach, P. A. (1993). Predicate Invention in Inductive Data Engineering. *European Conference on Machine Learning (EMCL)*, pages 83–94.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2012a). *Answer Set Solving in Practice.* Morgan Claypool.

Gebser, M., Kaminski, R., König, A., and Schaub, T. (2011). Advances in gringo series 3. In *International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR)*, pages 345–351.

Gebser, M., Kaufmann, B., and Schaub, T. (2012b). Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89.

Gelfond, M. and Kahl, Y. (2014). *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach.* Cambridge University Press.

Gelfond, M. and Lifschitz, V. (1988). The Stable Model Semantics for Logic Programming. In *International Conference and Symposium on Logic Programming (ICLP/SLP)*, pages 1070–1080.

Gulwani, S., Hernandez-Orallo, J., Kitzelmann, E., Muggleton, S. H., Schmid, U., and Zorn, B. (2015). Inductive programming meets the real world. *Communications of the ACM*, 58(11):90–99.

Katzouris, N., Artikis, A., and Paliouras, G. (2015). Incremental learning of event definitions with Inductive Logic Programming. *Machine Learning*, 100(2-3):555–585.

Law, M., Russo, A., and Broda, K. (2014). Inductive Learning of Answer Set Programs. In *European Conference on Logics in Artificial Intelligence (JELIA)*, pages 311–325.

Law, M., Russo, A., Cussens, J., and Broda, K. (2016). The 2016 competition on Inductive Logic Programming. retrieved 29 March 2017.

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

Lifschitz, V. (2008). What Is Answer Set Programming? In *AAAI Conference on Artificial Intelligence*, pages 1594–1597.

Mitra, A. and Baral, C. (2016). Addressing a question answering challenge by combining statistical methods with inductive rule learning and reasoning. In *Association for the Advancement of Artificial Intelligence*, pages 2779–2785.

Muggleton, S. (1987). Duce, an Oracle-based Approach to Constructive Induction. In *IJCAI*, pages 287—-292.

Muggleton, S. (1995). Inverse entailment and Progol. *New generation computing*, 13(3-4):245–286.

Muggleton, S. and Buntine, W. (1992). Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–352.

Muggleton, S. and De Raedt, L. (1994). Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19:629–679.

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

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

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

Otero, R. P. (2001). Induction of Stable Models. In *Conference on Inductive Logic Programming*, pages 193–205.

Ray, O. (2009). Nonmonotonic abductive inductive learning. *Journal of Applied Logic*, 7:329–340.

A Appendix: Further Program Modules for ASP Hypothesis Generation

The main module for hypothesis generation (Fig. 3) yields as answer sets the representations of all rules that can be created from a mode bias given as facts, and that obey hard limits as described in Section 3.3.

In the following we give additional program modules that the Inspire system uses for generating the hypothesis space. Section A.1 explains how we eliminate redundancies in the hypothesis space. Section A.2 provides the encoding for representing the cost of a solution, according to cost configuration parameters described in Section 3.3. Finally, Section A.3 gives the encoding module that enables predicate invention.

A.1 Redundancy Elimination Module

The main encoding in Fig. 3 creates many solutions that produce the same or a logically equivalent rule. As an example, line 1 defines **maxvars** variables of the same type, and line 7 guesses which of these variables to use. If variables with index 1 and 2 have the same type, there can be two answer sets which represent two rules that are different modulo variable renaming. For example the rules '`foo(V1) :- bar(V1).`' and '`foo(V2) :- bar(V2).`' are redundant. Redundant rules make hypothesis search slower and should be avoided.

Fig. 4 gives an encoding that eliminates many redundancies. Line 1 ensures that if we use variables of a certain type, we use only variables with the lowest index of that type. This is realized by using variable $v(\mathit{Id}\text{-}1)$ if we use $v(\mathit{Id})$ given that *Id* and *Id-1* have the same type. Similarly, lines 2–3 require that those body literals with the lowest indexes are used. Lines 4–5 canonicalize which variables are used in rule heads: line 4 requires that the variables with lowest index are bound in the head, and line 5 rules out solutions where two variables of same type are used in the head where the variable with the lower index is used in the second argument of the predicate. This rules out, e.g., a rule with head `foo(X2,X1)` if `X1` and `X2` have same type. For further redundancy elimination we rely on the lexicographic order of terms in ASP. In lines 6–10 we represent a 'variable signature' in atoms of form $\textbf{lit\_vsig}(I, \mathit{Idx}, \mathit{Pol}, A, \mathit{Sig})$

```
 1 use_var_type(v(Id-1),Type) :- use_var_type(v(Id),Type), var_type(v(Id-1),Type).
 2 use_body_pred(id_idx(PredId,Idx-1),Pred,Polarity,Arity) :-
 3     use_body_pred(id_idx(PredId,Idx),Pred,Polarity,Arity), Idx > 1.

 4 :- bind_hvar(Pos,v(Id)), var_type(v(Id),T), var_type(v(Id-1),T), not hbound_var(v(Id-1)).
 5 :- bind_hvar(1,Id1), bind_hvar(2,Id2), var_type(Id1,T), var_type(Id2,T), Id1 > Id2.

 6 lit_vsig(Id,Idx,Polarity,1,V) :- use_body_pred(id_idx(Id,Idx),_,Polarity,1),
 7                                  bind_bvar(id_idx(Id,Idx),Polarity,1,V).
 8 lit_vsig(Id,Idx,Polarity,2,vv(V1,V2)) :- use_body_pred(id_idx(Id,Idx),_,Polarity,2),
 9                                          bind_bvar(id_idx(Id,Idx),Polarity,1,V1),
10                                          bind_bvar(id_idx(Id,Idx),Polarity,2,V2).

11 :- lit_vsig(Id,Idx1,Pol,A,Sig1), lit_vsig(Id,Idx2,Pol,A,Sig2), Idx1 < Idx2, Sig1 > Sig2.

12 litequal_upto(Id1,Id2,Polarity1,Polarity2,0,Arity) :-
13     use_body_pred(Id1,Pred,Polarity1,Arity),
14     use_body_pred(Id2,Pred,Polarity2,Arity), (Id1, Polarity1) < (Id2, Polarity2).
15 litequal_upto(Id1,Id2,Polarity1,Polarity2,Upto+1,Arity) :-
16     litequal_upto(Id1,Id2,Polarity1,Polarity2,Upto,Arity), Upto < Arity,
17     bind_bvar(Id1,Polarity1,Upto+1,VId), bind_bvar(Id2,Polarity2,Upto+1,VId).
18 litequal(Id1,Id2) :- litequal_upto(Id1,Id2,_,_,Arity,Arity).
19 :- litequal(Id1,Id2).
```

Figure 4: Redundancy elimination module for ASP Hypothesis Generation.

where I is the predicate ID, Idx the index, Pol the polarity, A the arity, and Sig is a composite term containing all variables in the literal for which the signature is defined. Using signatures, the constraint in line 11 requires that literals with equal predicates and polarities are sorted in the same way as their variable signatures. This would, for example, eliminate a rule with body '`foo(X2,X3), foo(X1,X2)`' while it would allow to use the (logically equivalent) body '`foo(X1,X2), foo(X2,X3)`'. Finally we represent which literals have equal predicate and arguments in lines 12–16, and exclude solutions with equal literals in line 19.

Lines 5–10 of this encoding are suitable only for predicates with arity one or two. This was sufficient for the ILP competition and can be generalized to higher arities easily.

A.2 Fine-Grained Cost Module

Fig. 5 shows our encoding module for representing the cost of a rule based on Section 3.3. Cost atoms of form **cost**(*Name*, *Data*, *Cost*) represent various costs that add up to the total rule cost: each cost atom bears a name *Name* used to distinguish different aspects of cost, moreover it contains *Data* to incur cost for different parts of the rule under the same aspect. Finally *Cost* is the actual cost incurred for *Name* and *Data*.

Example 9. *If the cost aspect is Name =* **vartype_morethantwice** *then the data contains the variable type for which this cost is incurred. This aspect might incur cost for multiple variable types, leading to multiple atoms with different Data values and as a result higher total cost.*

Lines 1–2 define cost for the number of distinct variables that are used in the rule, where the first **free_distinct_variables** variables incur no cost. Lines 3–4 define cost for variable types that are used more than twice (using a type once or twice is free). Lines 5–6 define costs for positive and negative body literals, and line 7 defines cost for using a predicate multiple times in the body. Note, that line 7 relies on the property that the body literals of lowest index are used, which is ensured by the redundancy elimination module (Fig. 4 lines 2–3). Line 8 defines cost for each variable that is bound only in the head. Lines 9–10 define cost for variables that occur only once in the body of the rule and not in head. Lines 11–13 define cost for variables that occur more than twice in the rule. Lines 14–17 define cost for binary predicates that contain the same variable in both arguments. Lines 18–19 sum up the total cost if that total is below **maxcost**, otherwise the total cost is fixed to **maxcost**.[2] Finally solutions that reach or exceed a total cost of **maxcost** are excluded in line 20.

[2]The rule `totalcost(C) :- C = #sum { Cost,U,V : cost(U,V,Cost) }.` would sum up total cost in a single rule, with-

```
 1 cost(varcount,dummy,(Count-free_vars)*cost_vars) :-
 2     varcount(Count), Count > free_vars.

 3 cost(vartype_morethantwice,Type,cost_type_usedmorethantwice*(NUse-2)) :-
 4   NUse > 2, NUse = #count { Id : use_var_type(v(Id),Type) }, type_id(Type,_).

 5 cost(pbodylit,IdIdx,cost_posbodyliteral) :- use_body_pred(IdIdx,_,pos,_).
 6 cost(nbodylit,IdIdx,cost_negbodyliteral) :- use_body_pred(IdIdx,_,neg,_).
 7 cost(pmulti,id_idx(P,Idx),cost_pred_multi) :- use_body_pred(id_idx(P,Idx),_,_,_),Idx > 1.

 8 cost(varonlyh,V,cost_varonlyhead) :- use_var_type(V,_), hbound_var(V), not bbound_var(V).

 9 cost(varonlyonceb,V,cost_varonlyoncebody) :- use_var_type(V,_), not hbound_var(V),
10     1 = #count { Pred,Pol : bind_bvar(Pred,Pol,_,V) }.

11 cost(varmorethantwice,V,cost_varboundmorethantwice*(N-2)) :-
12     use_var_type(V,_), N > 2,
13     N = #count { h,Pos : bind_hvar(Pos,V) ; b,Pred,Pol,Pos : bind_bvar(Pred,Pol,Pos,V) }.

14 reflexive(id_idx(Pr,Idx),Pol) :-
15   bind_bvar(id_idx(Pr,Idx),Pol,1,V), use_body_pred(id_idx(Pr,Idx),Pred,Pol,2),
16   bind_bvar(id_idx(Pr,Idx),Pol,2,V).
17 cost(reflexive,id_idx_pol(Pr,Idx,Pol),cost_reflexive) :- reflexive(id_idx(Pr,Idx),Pol).

18 totalcost(C) :- C = #sum { Cost,U,V : cost(U,V,Cost) }, C < maxcost.
19 totalcost(maxcost) :- maxcost <= #sum { Cost,U,V : cost(U,V,Cost) }.
20 :- totalcost(C), C >= maxcost.
```

Figure 5: Fine-grained cost module for ASP Hypothesis Generation.

A.3 Predicate Invention Module

This ASP module extends the main encoding by adding the possibility to invent predicates in the hypothesis search space.

Line 1 guesses at most one arity for up to **maxinventpred** invented predicates. Guessing no arity means that the invented predicate is not used, and line 2 guesses for each invented predicate one type for each argument position. Lines 3–6 connect the invented predicate encoding with the main encoding by defining that invented predicates can be used both as head as well as body predicates in rules in the hypothesis search space. Line 7 requires that arguments of invented predicates are sorted lexically in the same way as the IDs of their types are. This reduces redundancy, because it does not matter in practice whether we use an invented predicate as `inv(Type1,Type2)` or as `inv(Type2,Type1)`, as long as it is used in the same way in all rules. Lines 8–10 perform further redundancy elimination by defining which invented predicates are used, and eliminating solutions where we guess the existence of an invented predicate but do not use it. Line 11 incurs a cost for predicate invention in general, line 12 incurs a cost for each invented predicate, lines 13–14 incur extra cost if the invented predicate is used both in the head and in the body of the rule in the answer set, lines 15–17 incur extra cost for multiple usages of the same predicate in the rule body, and lines 18–20 define a cost for pairs of different invented predicates where one is in the body and the other one in the head of a hypothesis rule: cost is incurred if the predicate in the head is lexicographically greater or equal to the predicate in the body. This prevents rules that can make cycles over invented predicates early in the search process, and intuitively prefers rules that are 'stratified' according to their usage of invented predicates.

out clamping the value to the maximum interesting value **maxcost**. However, this approach has a significantly larger instantiation than the rules in lines 18–19.

```
0 { ipred(ip(Id,A),A) : A = inv_minarity..inv_maxarity } 1 :- Id = 1..maxinventpred.
1 { iarg(Id,Pos,T) : type(T) } 1 :- ipred(Id,A), Pos = 1..A.

hpred(Id,Id,A) :- ipred(Id,A).
harg(I,J,T) :- iarg(I,J,T).
bpred(Id,Id,A) :- ipred(Id,A).
barg(I,J,T) :- iarg(I,J,T).

:- iarg(Id,Pos1,T1), iarg(Id,Pos2,T2), Pos1 < Pos2, T2 < T1.

use_ipred(Id) :- ipred(Id,_), use_head_pred(Id,_,_).
use_ipred(Id) :- ipred(Id,_), use_body_pred(id_idx(Id,_),_,_,_).
:- ipred(Id,_), not use_ipred(Id).

cost(inv,dummy,cost_inv) :- ipred(_,_).
cost(inv_pred,Id,cost_inv_pred) :- ipred(Id,Arity).
cost(inv_headbody,ip(Id,A),cost_inv_headbody) :-
    use_head_pred(ip(Id,A),_,_), use_body_pred(id_idx(ip(Id,A),Idx),_,_,A).
cost(inv_bodymulti,id_idx_idx(ip(Id,A),Idx1,Idx2),cost_inv_bodymulti) :-
    use_body_pred(id_idx(ip(Id,A),Idx1),_,_,A), Idx1 < Idx2,
    use_body_pred(id_idx(ip(Id,A),Idx2),_,_,A).
cost(inv_headbodyorder,h_b(ip(Id1,A1),id_idx(ip(Id2,Idx2))),cost_inv_headbodyorder) :-
    use_head_pred(ip(Id1,A1),_,A1), Id1 >= Id2,
    use_body_pred(id_idx(ip(Id2,Idx2),_),_,_,_).
```

Figure 6: Predicate invention module for ASP Hypothesis Generation.