**ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA**
**CAMPUS DI CESENA**

Scuola di Ingegneria e Architettura
Corso di Laurea (Magistrale) in Ingegneria e Scienze Informatiche

# Logic ecosystems meet meta-interpretivative learning: design and experiments on 2P-Kt

Tesi di laurea in
Autonomous Systems

*Relatore*
**Prof. Andrea Omicini**

*Correlatore*
**Dott. Giovanni Ciatto**

*Candidato*
**Luca Nannini**

Quarta Sessione di Laurea
Anno Accademico 2020-2021

# Abstract

This thesis is rooted in the field of *Inductive Logic Programming* (ILP), and, in particular, *Meta-Interpretative Learning* (MIL). ILP is a branch of *Machine Learning* where the *Artificial Intelligence* tries to induce Horn clauses from a given background knowledge and some positive/negative examples.

The goal of this thesis is the development of a system for assisting interpretative learning algorithms.

In order to achieve that, we extend the 2P-Kt logic ecosystem for *symbolic artificial intelligence*, with *meta-rule*s support. We then design and implement a system of pluggable components aiming to assist the various steps of ILP algorithms, such as generalization of induced rules and refinement of theories.

The results are: a 2P-Kt based library of various generalization, validation and refinement strategies, a brand new algorithm inspired by Metagol (named MetaPatrol) and a test suite. The system consists of a 2P-Kt module supporting the definition of *meta-rule*s; as well as the generalization, validation, and refinement of induced theories as first class mechanisms. Overall, the system enables the engineering of ILP solutions by combining multiple strategies for the many aspects of MIL.

*Le sfortune capitano, ma io me le procuro attivamente.*

# Acknowledgements

In this section I wish to spend a few words for expressing my gratitude to the people who supported me during my academic journey, and my special thanks to who drove me through the development of this thesis.

Firstly, I'd wish to thank Prof. Andrea Omicini for letting me develop such a complex thesis in such a short time window: a difficult challenge that few would have accepted to undertake. Therefore, I wish to thank him for his confidence in me.

A special thanks goes to Dr. Giovanni Ciatto who drove me through the design, implementation and the writing of this thesis. I'd wish to thank him not only for helping me rearranging my ideas, but also for his valuable advice and the time he spent (wasted) for (because of) me.

*metapatrol_would_not_exist_without(gciatto).*

and that's a fact!

I'd wish to thank all of my friends, those who accompanied me during my studies, and also those who didn't. Thanks for sharing and/or alleviating this infernal time of my life. I could write down a list, but I'd never forgive myself if I made an incomplete one.

Last but not least, I'd wish to thank my family, although most of them don't speak english, but I was told not to write the current section in italian for consistency with the rest of the thesis.

A special thanks goes to my mother: if I cut down all the trees in the world I wouldn't have enough paper to thank her. That's because trees are made of wood and not paper. Seriously: a mother is a mother, and my mother is my mother. This is emotional but luckily she will never read.

I'd wish to thank my father, especially for the little things like taking care of the car and for opening the garage for me every morning, helping me not to miss the train. Thanks for fixing my piano (thanks to my mother too) and thanks for fixing the internet antenna. My father is a professional repairman!

Finally, I'd wish to thank my brother. Thanks for friday evenings, one of the few certainties of life. Thanks for the complicity and the jokes that only brothers can understand.

# Contents

# List of Figures

# Listings

# Chapter 1

# Introduction

The most significant trait of human intelligence is probably its ability to learn, in particular its ability to acquire new knowledge on the basis of experiences and through the application of study and in-depth analysis of the phenomena that surround us. One of the most common learning mechanisms is induction: by observing examples we define general rules which abstract from the examples.

This historical period is characterized by a strong hype for the research related to *Artificial Intelligence*: the developed systems try to reproduce mechanisms of automatic induction and the produced results are disrupting.

> *"The pace of progress in artificial intelligence (I'm not referring to narrow AI) is incredibly fast. Unless you have direct exposure to groups like Deepmind, you have no idea how fast — it is growing at a pace close to exponential."*
>
> Elon Musk (Entrepreneur, investor and business magnate.)

Nowadays we can identify two main approaches in the development of intelligent systems: *Machine Learning* (ML) and *Inductive Logic Programming* (ILP). Both approaches exploit sets of examples and a *bias*: an assumption for reducing the hypothesis space.

ML deals with the problem of induction trying to generate rules and predictions based on computational statistics and mathematical optimization. Currently it is the most popular approach and consequently it is achieving quite satisfactory results nowadays.

> *"We are entering a new world. The technologies of machine learning, speech recognition, and natural language understanding are reaching a nexus of capability. The end result is that we'll soon have artificially intelligent assistants to help us in every aspect of our lives."*

> Amy Stapleton (Chatables Cofounder. Voice tech, Conversational AI, virtual beings.)

ILP, unlike traditional ML, is a knowledge-based approach, in particular it exploits logic programming. Being based on knowledge, ILP allows to develop systems that generate rules more in line with the human logical rationale, with a consequent easier verification of the produced results by users. A further advantage of ILP is its ability to generalize starting from a small set of examples, where ML requires a large amount of data.

This thesis is rooted in the field of *Inductive Logic Programming* (ILP)[17], and, in particular, *Meta-Interpretative Learning* (MIL)[18]. MIL is an ILP approach based on meta logic programming. The state of the art of ILP is full of solutions leveraging on as many induction strategies. For instance Golem[16] is based on the concept of *least general generalization* which is the construction of a unique clause which covers a given set of examples. Metagol [8] is a MIL algorithm based on meta-programming in Prolog, exploiting particular clauses called *meta-rule*s. Another example of ILP algorithm is ILASP [14], which is based on a conflict-driven approach and exploits the definition of constraints. Unfortunately, the implementation of such algorithms are unavailable, unmaintained or tailored on Prolog. That is not in itself a problem but it forces users to be able to use MIL exclusively via Prolog.

Even if these algorithms have different induction engines, they share common characteristics regarding key aspects of the induction process such as: *(i)* the exploitation of background knowledge (e.g. clauses made available to the induction engine), *(ii)* the use of positive and negative examples upon which induction should be performed, *(iii)* the *generalization* of non-ground clauses from ground ones, *(iv)* the *validation* of the induced theories on the basis of the aforementioned examples, and *(v)* the consequent *refinement* of such induced rules.

Notably, existing algorithms from the literature come with hard-coded choices concerning the generalization, validation, and refinement strategies they follow while pursuing induction. Hence, to the best of our knowledge, an ILP solution enabling the combination of different generalization/validation/refinement strategies (or the implementation of new strategies on the fly) is currently missing.

**Thesis Purpose.** The goal of this thesis is the development of a library for supporting ILP (and in particular MIL) algorithms without strictly relying upon Prolog. This, would enable the induction of knowledge bases by exploiting other resolution strategies than Prolog's depth-first, backtracking-based one. Furthermore, and more importantly, this would enable the exploitation of MIL outside the realm of LP.

In order to achieve that, we exploit 2P-Kt[2], an ecosystem for *symbolic artificial intelligence*. In particular, the contributions of this thesis are:

- the design of general, strategy-agnostic API for ILP, supporting the specification of positive and negative examples, background knowledge, and linguistic bias;
- the design of MIL-specific API, supporting the specification of *meta-rule*s as linguistic bias, and admitting the customisability of key phases, such as
  - generalization of ground clauses into non-ground ones,
  - validation of the induced theories against the provided examples, and
  - refinement of the induced theories (e.g., via predicate invention[21]);
- the design and implementation of MetaPatrol: a brand new ILP algorithm based on MIL and inspired by Metagol, adhering to the aforementioned API;
- a test suite demonstrating the effectiveness of MetaPatrol.

**Thesis Structure.** This thesis is structured as follows. Chapter 2 briefly summarises the *Logic Programming Paradigm* and stands as a contextualization of the current ILP state of art; finally it introduces the 2P-Kt framework. Chapter 3 discusses the desiderata, in particular the necessary abstractions focusing on an in-depth analysis of the identified phases of a ILP algorithm; finally it discusses the framework structure and the MetaPatrol algorithm logic. Chapter 4 discusses the major implementative details and the technological choices we adopt such as using 2P-Kt. Chapter 5 discusses the results and the tests performed on the deliverable, provided as validation of the latter. Finally, Chapter 6 poses as a conclusion of this thesis and suggests some interesting ideas for future works.

# Chapter 2

# Background

## 2.1 Logic Programming

In order to understand what is ILP about, we need to understand what is *Logic Programming* (LP) first.

LP is a *programming paradigm* based on computational logic: a logic program is a set of logical relations (facts or rules) [2]. Computation, in a logic program, is a form of deduction such as searching for a proof or a refutation.

The main difference among LP and the other programming paradigms (e.g., imperative, functional, etc.) is that LP is *declarative*:

- that means it allows to state *what* a program should do, rather than how it should work;
- usually making assumptions on the order of the rules in a logic program does not matter [5].

### Syntax

As briefly mentioned above a logic program is composed by facts and rules, all of them collected within a *theory*.

Here a more detailed syntax definition:

- *terms*: symbols representing entities in the domain
  - *constants*: denoting simple entities, a string starting with a lowercase letter (e.g., *jack, tree*);
  - *variables*: denoting placeholders for entities, a string starting with an uppercase letter (e.g, *A, Variable*);

  - *structures (functions)*: denoting composed entities, in the form *functor(arguments)* (e.g., *coordinate(1, 2)*);

  a term is *ground* if it contains no variables.

- *lists*: representing linked lists, they are written between square brackets:
  - *ground list*: e.g., [1, 2, 3] where all elements are separated by a comma;
  - *empty list*: [] ;
  - *head and tail list*: in the form $[H \mid T]$ which denotes a list whose head is $H$ and whose tail is $T$.

- *predicates*: *syntactically* equal to *structures* but *semantically* different:
  - since *terms* represent entities of the domain they just exist;
  - *predicates* represent *statements* about those entities (which can be either true or false).

  the arity $n$ of a predicate $p$ is the number of arguments it takes and is denoted as $p/n$ (e.g., *raining/0, even/1, son/2, append/3*).

- *clauses*: facts and rules are expressed as *Horn Form* clauses such as:

$$H \text{ :- } B_1, ..., B_n.$$

  where:
  - $H$ denotes the *head* of the rule;
  - $(B_1, ..., B_n)$ denotes the *body* of the rule, each element of the body is called *literal*;
  - *:-* is the implication symbol;
  - , is the logic operator *AND* (while ; is the logic operator *OR*);
  - . denotes the end of the clause.

  More specifically:
  - *rule*: a clause with head and body (e.g., *son(A, B) :- male(A), parent(B, A).*);
  - *fact*: a clause with no body (denoting a tautology) (e.g., *son(jack, albert).*);
  - *goal*: a clause with no head (denoting a contradiction) (e.g., *:- son(jack, mary).*

## 2.2 Inductive Logic Programming

ILP is a discipline within the realm of Symbolic AI, in particular in the field of Supervised ML [1].

Symbolic AI is a branch of AI based on high-level symbolic representations of problems, in contrast with Subsymbolic AI which is based on computational statistics and mathematical optimization. Exploiting high-level symbolic representations, a Symbolic AI provides more human-readable solutions and allows greater understanding of the induction processes and induced hypotheses to human users.

Supervised ML is a branch of Symbolic AI inducing hypotheses $H^*$ from a set of given evidences:

- some preliminary knowledge $B$ about the domain of interest called *background knowledge*;
- some *positive examples* $E^+$ which must be satisfied by the representation to be learnt;
- some *negative examples* $E^-$ which must not be satisfied by the representation to be learnt;
- a *language bias* $C$ constraining the hypotheses space for more efficient learning.

In particular, ILP exploits LP as a paradigm for representing knowledge in a declarative and expressive fashion. The hypotheses $H^*$, the background knowledge $B$, the examples $E^+, E^-$, and eventually the bias $C$ (or a part of it) are espressed in Prolog or another LP language. Furthermore, some ILP algorithms are coded in such paradigm themselves i. e. Aleph [20], Metagol [8].

Positive and negative examples, in LP, are expressed as facts, while the background knolwedge is expressed as a set of clauses (a theory).

ILP extends the traditional computational logic using *induction* rather than *deduction* for inference, providing a powerful formalism for expressing knowledge. Such formalism ensures a better comprehension of the induction processes and induced hypotheses, which might be more difficult to achieve in ML approaches. Furthermore, ILP can induce hypotheses from small numbers of examples, often from a single example, making it efficient for real-world applications where large numbers of examples are not always easy to obtain [5].

**Bias.** The general purpose of the linguistic bias is to constrain the hypothesis space. In the particular case of ILP, bias aims to constrain the combinations of the relations from the background knowledge, from which the hypotheses derive.

This can come in different shapes and flavours, depending on the particular ILP method in place [1].

Bias is an important feature in the field of symbolic supervised learning, and its role is similar to the one played by hyper-parameters in sub-symbolic supervised learning.

Some examples of common biases are *mode declarations* [11] and *meta-rule*s. For the purpose of this thesis we only delve into the *meta-rule* world, discussing them in Section 2.2.2.

### 2.2.1 Strategies (Search Method)

As previously mentioned, ILP algorithms are based on disparate induction engines, that consequently means that they rely on various induction strategies. Each strategy aims to explore the hypotheses space in an efficient way, featuring different levels of generalization or specialization. A more generalized hypothesis covers a plethora of cases while a more specialized one covers a most particular case.

In this section we will briefly categorize and summarize the most popular ILP search method approaches. At the current state of the art there is no evidence of a "best" search method approach, to the best of our knowledge, and there are no limitations regarding the use of hybrid approaches. However, the most common approaches for ILP are:

**Top-down** approaches, starting from a more generalized hypothesis and proceed trying to specialize it. An example of a top-down ILP algorithm is Foil [19].

**Bottom-up** approaches, starting from a more specialized hypothesis and proceed trying to generalize it. An example of a bottom-up ILP algorithm is Golem [16].

**Meta-level** approaches, representing an ILP problem as a meta-level logic program, i.e., a program that reasons about programs. Meta-level approaches formulate the learning problems as declarative problems, and the meta-level solutions are translated back to a standard solution for the ILP problem [5]. Some examples of meta-level ILP algorithms are ASPAL [4] and Metagol [8].

**Conflict-driven** approaches, exploiting the definition of constraints for building the solution incrementally. Some examples of conflict-driven ILP algorithms are ILASP [13] and Popper [7].

### 2.2.2 Techniques

The state of the art of ILP features numerous algorithms exploiting a disparate number of techniques: *Structure Learning*, *Relative least-general generalization*, *Inverse Entailment*, *Bottom-clause propositionalisation*, *Meta-interpretative learning* (MIL), *Parameter Learning* to cite some. Each technique is characterized by different advantages and disadvantages, but for the purpose of this thesis we merely focalise on MIL, one of the most modern approaches.

#### Meta-interpretative learning

MIL algorithms are *Meta-level* algorithms leveraging on the meta-level programming capabilities of logic solvers, and in particular Prolog ones [1].

For achieving meta-programming in Prolog, some ILP algorithms exploit the concept of *meta-rule* (e.g., ILASP and METAGOL).

***Meta-rule*s.** *Meta-rule*s are a form of syntactic bias introduced by Emde in 1983 [12]. The purpose of *meta-rule*s in ILP is to define the admissible structure of the induced predicates, aiming to reduce the hypothesis space.

*Meta-rule*s are higher-order clauses and, in addition to the ordinary Prolog variables, they introduce the concept of *second-order variable* (a.k.a. higher-order variable), such as:

$$\mathbf{X}(A, C) \text{ :- } \mathbf{Y}(A, B), \mathbf{Z}(B, C).$$

where the uppercase and bold letters $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$ denote second-order variables, while the letters $A$, $B$, $C$ denote first-order (ordinary) variables.

A second-order variable can be bound to a functor (predicate symbol). Based on *background knowledge* and/or *positive examples*, MIL approaches try to find a suitable substitution for the second-order variables, in order to induce a solution to the ILP problem.

The major difference between *meta-rule*s and other forms of bias in ILP is that *meta-rule*s themselves are logical statements, allowsing us to reason about them. Anyway, despite their widespread use, there is little work determining the best *meta-rule*s to apply to a learning task, making the choice of *meta-rule*s an open and promising research field [5].

*Meta-rule*s play a pivotal role in MIL, in fact, as stated in [6]:

> *"following MIL, many ILP systems can learn recursive programs. With recursion, ILP systems can now generalise from small numbers of examples, often a single example. The ability to learn recursive programs has opened up ILP to new application areas, including learning string transformations programs, robot strategies, and answer set grammars."*

### 2.2.3 State of the Art

As previously mentioned, the state of the art of ILP features numerous valuable algorithms. In this section we only focalize on a few of them, in particular we provide an overview of the most famous MIL algorithms.

#### ILASP

Although ILASP, *Inductive Learning of Answer Set Programs*, was not born as a pure MIL algorithm, its most recent versions support the concept of *meta-rule*, if provided within the *background knowledge* [22].

ILASP has been released in multiple versions during its lifetime and its team is still working on it, constantly improving its capabilities and optimization.

ILASP is based on the *Answer Set Programming* (ASP) paradigm, which allows to describe problems in terms of its specification, rather than requiring the user to define an algorithm to solve the problem [13]. The learning activity is delegated to an ASP solver, called incrementally on a program which "grows" during the learning phase.

While the versions ILASP1 and ILASP2 adopt a meta-level search mode approach, the most recent versions of ILASP are moving towards a conflict-driven search mode approach, defining constraints at each iteration of the algorithm, in order to find a solution to the learning problem. By adopting a conflict-driven approach, ILASP implements a sort of "learning by failure" technique.

Here a brief explanation of the implementation of ILASP.

For each iteration:

1. the solver generates a hypothesis which satisfies che current defined constraints;

2. the solver looks for a conflict, trying to refute the current hypothesis;

    - if there is no conflict, the solver returns the hypothesis which results as a valid inductive solution;
    - if there is a conflict, the solver computes a new constraint, based on the occured conflict, supporting the next iterations.

Referring to the relevant paper of ILASP [13], constraints can be extremely large and expensive to compute, especially for larger learning problems. The main issue is that the constraints are both sufficient and necessary for the example to be covered, and an example is considered to be covered if and only if the constraints are satisfied. In order to relax this issue, the ILASP team is currently developing ILASP4 for computing constraints which are only guaranteed to be necessary (but may not be sufficient) for the example to be covered, for overall better performance.

### Metagol

METAGOL[8] is a MIL algorithm based on a meta-level search method approach. Although its developers decided to interrupt its development for moving to a new conflict-driven algorithm (POPPER), METAGOL still stands as one of the most interesting and promising examples of ILP, in particular MIL, in our opinion.

METAGOL incarnates the true essence of *"meta-programming"* (from which takes half of its name) and is entirely coded in Prolog, specifically in *SWI-Prolog*.

METAGOL exploits the declaration of *meta-rule*s as a linguistic bias for optimally exploring the hypothesis space, applying the so-called *meta-substitutions* (a.k.a. second-order/higher-order substitutions) from positive examples and background knowledge, in order to induce a solution to the learning problem. Since Prolog does not provide a native way to express a *meta-rule*, the METAGOL Team have come to the following formalism:

$$metarule(name, [X, Y, Z], [X, A, C], [[Y, A, B], [Z, B, C]]). \tag{2.1}$$

where:

- the first argument, *name*, denotes the name of the *meta-rule*, in particular the *meta-rule* in the example is commonly referred as *chain*;
- the second argument, $[X, Y, Z]$, denotes the list of second-order variables of the *meta-rule*;
- the third argument, $[X, A, C]$, denotes the head of the *meta-rule* where $X$ denotes the second-order variable (acting as functor) while $A, C$ denote first-order (ordinary) variables;
- the fourth argument, $[[Y, A, B], [Z, B, C]]$, denotes the body of the *meta-rule*; each list denotes a literal (i. e. $[Y, A, B]$ and $[Z, B, C]$); literals are expressed following the same method adopted for the head, where the first variable denotes the second-order variable (acting as functor) while the other variables denote first-order (ordinary) variables.

The formalism from eq. (2.1) aims to represent the following *meta-rule*:

$$\mathbf{X}(A, C) \text{ :- } \mathbf{Y}(A, B), \mathbf{Z}(B, C).$$

Listing 2.1 provides a simplified implementation of METAGOL [9], here a more detailed explanation of how it works:

1. METAGOL verifies if a goal (a positive example head) is already proved, deductively delegating the proof to Prolog;

Listing 2.1: Simplified Metagol Implementation [9]

```
prove([], H, H).
prove([Atom | Atoms], H1, H2):-
    prove_aux(Atom, H1, H3),
    prove(Atoms, H3, H2).
prove_aux(Atom, H, H):-
    call(Atom).
prove_aux(Atom, H1, H2):-
    member(sub(Name, Subs), H1),
    metarule(Name, Subs,(Atom :- Body)),
    prove(Body, H1, H2),
prove_aux(Atom, H1, H2):-
    metarule(Name, Subs, (Atom :- Body)),
    new_metasub(H1, sub(Name, Subs)),
    abduce(H1, H3, sub(Name, Subs)),
    prove(Body, H3, H2).
```

(a) if the prove fails, METAGOL tries to unify the positive example head with the head of a *meta-rule*. METAGOL proceeds trying to substitute all the literals of the partially substituted *meta-rule*, basing on the background knowledge;

- if it finds a complete ground substitution, METAGOL tries to prove the body goals of such substituted *meta-rule*. If the body goals are proved, METAGOL applies a *meta-substitution* to the initial *meta-rule*, substituting the ground rule functors to the initial *meta-rule* second-order variables;
- if it does not prove the example, METAGOL tries to prove the example following the previous points with another *meta-rule*, if there is one;

(b) if the prove succeeds, METAGOL proceeds to the next example (returning at point 1).

2. when all examples are proved, METAGOL tests the hypothesis on the negative examples. If the hypothesis does not entail any negative example, it stops and returns the hypothesis; otherwise it backtracks to a choice point at (a) and continues.

**Recursion.** A recursive logic program is one where the same predicate in the head of a rule appears also in its body. Usually recursive rules feature a base case and one or more recursive cases, a recursive rule without base case would generate an infinite loop.

Learning recursive programs has long been considered a difficult problem for ILP and, without recursion, it is often difficult for an ILP system to generalize

from small numbers of examples [6].

METAGOL, exploiting *meta-rule*s, is capable of inducing recursive programs if the provided *meta-rule*s are recursive themselves (or one of the base/inductive cases are provided within background knowledge).

**Higher-Order.** METAGOL supports the induction of higher-order programs.

Suppose to provide some pre-made utility rules to METAGOL within the background knowledge. Since the second argument of a *meta-rule*, expressed in the previously specified formalism, denotes the list of second-order variables of the *meta-rule*, it is possible to specify a higher-order variable to use within a literal (instead of as a functor), such as:

$$metarule(name, [X, Y, C], [X, A, B], [[Y, A, B, C]]).$$

where $C$ denotes a second-order variable present in the body of the *meta-rule*, specifically:

$$\mathbf{X}(A, B) \text{ :- } \mathbf{Y}(A, B, \mathbf{C}).$$

METAGOL will then look for a substitution for $C$ from the background knowledge in the same way it does for functors. For a clearer explanation: suppose we provide a higher-order predicate of arity 3 within the background knowledge, METAGOL would substitute $Y$ with it, then METAGOL would search for a higher-order substitution for $C$.

**Predicate Invention.** The goal of predicate invention is for an ILP system to automatically invent new auxiliary predicate symbols, reducing the required background knowledge size. Usually, on the other hand, for predicate invention to work properly, the smaller size of the required background knowledge comes at the cost of larger *meta-rule*s.

The idea behind predicate invention is to reduce code duplication or to improve readability, similar to when humans create new functions when manually writing programs [6].

METAGOL features an example of predicate invention, here we provide a pragmatic case [5].

Supposing to provide METAGOL the background knowledge at Listing 2.2 and the *chain* metarule

$$\mathbf{X}(A, C) \text{ :- } \mathbf{Y}(A, B), \mathbf{Z}(B, C).$$

(appropriately expressed in the formalism supported by METAGOL).

Supposing to provide METAGOL some positive and negative examples for learning the rule *grandparent*/2.

Listing 2.2: Metagol Predicate Invention Example – Background Knowledge

```
mother(ann, amy).
mother(ann, andy).
mother(amy, amelia).
mother(amy, bob).
mother(linda, gavin).
father(steve, amy).
father(steve, andy).
father(andy, spongebob).
father(gavin, amelia).
```

Listing 2.3: Metagol Predicate Invention Example – Solution without Predicate Invention

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

A possible solution may be the one at Listing 2.3, in which the induced theory is composed by four similar clauses, whose body is composed by a permutation of the predicates *mother/2* and *father/2*. A Prolog programmer would find the aforementioned solution too verbose. METAGOL, for dealing with that issue, would instead apply predicate invention and produce the solution at Listing 2.4, where *grandparent_1/2* is invented and corresponds to the *parent/2* relation. Overall, this means that, with the predicate invention feature, METAGOL is capable of inducing more compact solutions.

## 2.3 2P-Kt

**tuProlog (2p).** *tuProlog* [10] (2P) is an open-source logic programming framework supporting multi-paradigm programming. 2P features a bidirectional integration between the logic and object-oriented paradigms.

Listing 2.4: Metagol Predicate Invention Example – Solution with Predicate Invention

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

**tuProlog-Kotlin (2P-Kt).** 2P-Kt is a reboot of 2p, re-engineered as a Kotlin multi-platform library. The main purpose of 2P-Kt is to provide a general-purpose, extensible, interoperable, and open LP ecosystem supporting multiple platforms and programming paradigms [2].

2P-Kt features a *Kotlin DSL* for *Prolog*, enriching the Kotlin multi-paradigm (OOP+FP) language with LP in a straightforward and effective way.

2P-Kt aims to become an open ecosystem supporting symbolic AI. For this reason it is composed by a set of interdependent modules with the purpose of supporting manipulation and symbolic reasoning in an extensible and flexible way [3]. Figure 2.1 is an overview of the 2P-Kt API and shows how 2P-Kt exposes a Kotlin type for each main LP concept:

- logic Terms (including each particular sort of term, i.e. Variables, or Structures, etc.);
- logic Substitutions and Unification;
- Horn Clauses (including Rules, Facts, and Directives)
- knowledge bases and logic theories (Theory);
- automatic reasoning via Prolog's resolution strategy (Solver).

### Core module

Within the `:core` module, 2P-Kt features a hierarchy representing Prolog terms, supporting the instantiation of Prolog terms and some useful operations over them.

**Term.** In 2P-Kt logic terms are represented by the `Term` base type. As illustrated in Figure 2.2 `Term` poses as the root of the term hierarchy.

`Term`s are implemented as immutable data structures, therefore each `Term` subtype has public methods and fields, favoring their implementation and an overall finest control over them.

**Substitution.** Within the `:core` module, 2P-Kt features a representation of *substitutions*, which we described in Chapter 3. The `Substitution` type is a base type for all substitutions. There are two types of `Substitution`:

- `Unifier`s actually mapping a (possibly empty) set of `Var`s to their corresponding `Term`
- `Fail` representing the failed substitution.

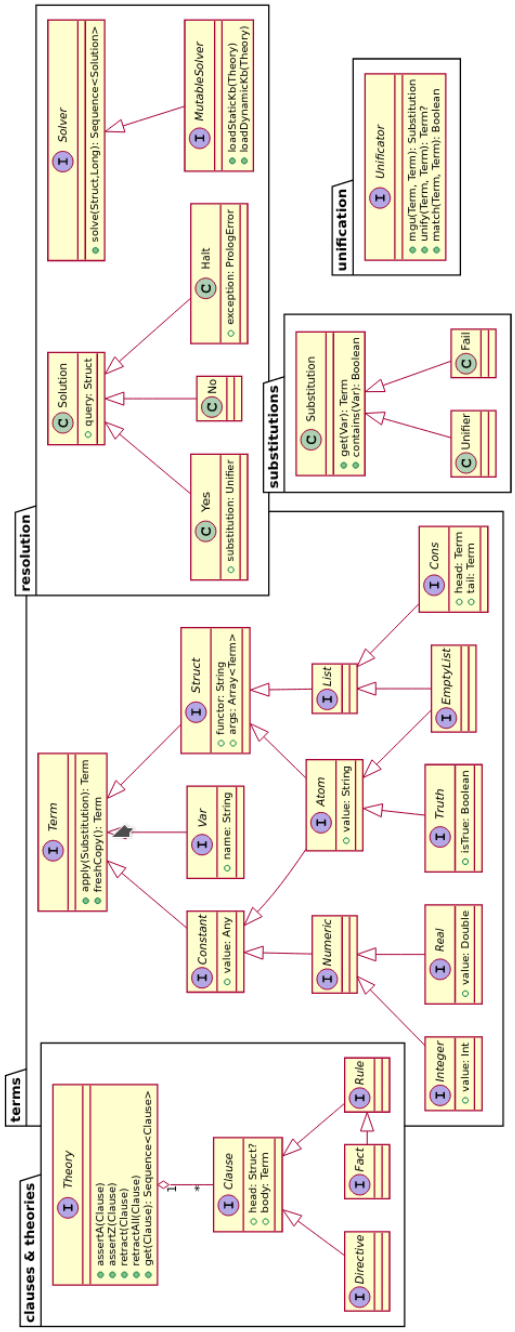


Figure 2.1: Overview on the Public API of 2P-Kt

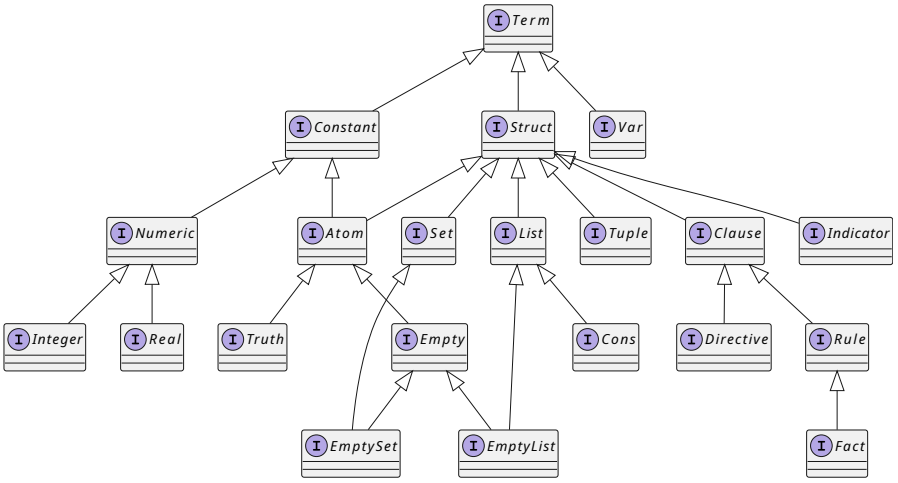


Figure 2.2: 2P-Kt – Simplified Term Hierarchy

**TermVisitor.** 2P-Kt provides the entity `TermVisitor<T>` with the following features:

- *Visitor* pattern applied to `Term`;
- supports use cases where a particular operation must be performed depending on the actual type of a `Term`.

An instance of `TermVisitor<T>` $v$ can be applied to a `Term` via the method `accept(v)`, this returns an object of type `T`.

`TermVisitor<T>` can be implemented by:

- overriding the method `defaultValue` to handle the general case;
- overriding some `visit<Type>` method to handle the particular case of `<Type>`.

**Term equality.** 2P-Kt provides `Term`s with two equality methods:

- `structurallyEquals`: for structural equality, which we describe in Chapter 3;
- `equals`: a strong equality check over the effective instances of two `Term`s within the same scope (`Scope`).

### Unify module

Within the `:unify` module, 2P-Kt features a representation of *unifications*, which we describe in Chapter 3.

**Unificator.** The `Unificator` type matches two `Term`s by finding a suitable substitution to their variables, i.e., the most general unifier (MGU).

- `mgu(Term, Term): Substitution`: returns the MGU between two `Term`s if a substitution is found, or a `Fail` object otherwise
- `match(Term, Term): Boolean`: checks if two `Term`s can be unified
- `unify(Term, Term): Term?`: tries to unify two `Term`s, possibly returning the unified `Term` as a result
- `merge(Substitution, Substitution): Substitution`: merges two `Substitution`s with occurs check enabled

Note: *occurs check* is a feature of some implementations of unification which causes unification of a logic variable $V$ and a structure $S$ to fail if $S$ contains $V$. Binding a variable to a structure containing that variable results in a cyclic structure which may subsequently cause unification to loop forever.

### Theory module

Within the `:theory` module, 2P-Kt features a representation of *theories*, which we described in Chapter 3.

**Theory.** The `Theory` type:

- provides high-level management of `Clause`s;
- is meant as a façade through which client code can access a knowledge base.

2P-Kt `Theory`(s) can be either *mutable* or *immutable*.

### Solve modules

Within the `:solve*` modules, 2P-Kt provides generic support to logic resolution. These modules provide the abstractions `Solver` and `Solution`. `Solver`s are reactive entities capable of answering to users' queries by producing one or more solutions, exploiting logic resolution.

In these modules 2P-Kt provides:

- libraries: containers of built-in functionalities exploitable by resolution;
- flags: configurable aspects of a solver;
- knowledge bases: containers of the logic knowledge used by resolution;

- operators: set of operators used to parse queries and to present solutions;

In particular, knowledge bases are of two sorts:

- static KB: which cannot be altered during resolution
- dynamic KB: which can be altered during resultion

### Parser/DSL modules

**Parser.** Within the `parser-*` modules 2P-Kt features the `TermParser` interface, which provides methods to parse `Term`s from strings (throwing a `ParseException` upon failure) and allows to select the operators used for parsing.

**DSL.** The `dsl-*` modules let Kotlin expoit LP with Prolog-like syntax. The 2P-Kt `DSL` does not rely a parser and has the followin purpose:

- integrating OOP, FP and LP;
- bringing OOP software engineering techniques to LP and FP;
- bringing declarativity from FP and LP to OOP;
- enriching the Kotlin language in a fruitful and natural way.

# Chapter 3

# Design

## 3.1 Desiderata

In this section we provide a categorization of the most common phases of an ILP algorithm. Every ILP algorithm, in addition to the actual inductive phase, may be composed by a set of earlier or later stages, which can be extrapolated as they may be valuable in a larger pool of ILP algorithms. In the particular case of MIL algorithms we identify the following phases:

- an *induction* phase in which the induction engine generates basic hypotheses; this can come in different shapes and flavours and, consequently, it represents the most distinctive feature of an ILP algorithm;
- a *generalization* phase in which the ground rules produced after substitutions from positive examples and/or background knowledge on a *meta-rule* are transformed in more general rules, for making them cover a larger set of examples;
- a *validation* phase in which the hypotheses are validated on the basis of the provided background knowledge, positive and negative examples. Usually a hypothesis covering negative examples is discarded while a hypothesis covering only a subset of the positive examples may be accepted;
- a *refinement* phase in which the validated hypotheses are subjected to a perfecting process which makes them more similar to what a skilled human programmer would produce while coding.

After the provided categorization we can then imagine to design and implement a library for supporting each individual MIL common phase, specifically: generalization, validation, and refinement. Furthermore, each of these phases can be

performed via various techniques and various purposes, in order for a client user to choose and plug them only if needed or, even more, to allow a client to implement new ones on the fly.

## 3.2 Necessary Abstractions

In order to implement a library for supporting MIL we need to identify all the necessary abstractions first.

Since MIL is a case of ILP and, since ILP is based on LP, we need to provide an abstraction for each entity which can be expressed with LP's syntax, in particular we choose Prolog's one. Therefore, we need to provide an abstraction for *meta-rule*s since there is no native way to express them in Prolog syntax. Finally we need to provide an abstraction for each operation of interest over the previously identified entities.

### 3.2.1 Prolog entities

Referring to the entities we reported at section 2.1 we can identify some higher level abstractions for representing knowledge:

- *terms*: for representing entities in the domain;
- *predicates*: for representing statements about entities;
- *clauses*: for representing properties of entities or relations among them.

More in detail, *terms* may be ordinary variables, constants, or structures composed by a string functor and a combination of other terms. Variables may be reused within the same scope or in different scopes, referring to the same entity. Constants may be either string values or numbers. A structure may be nested, i.e., composed by inner structures. Since lists follow the same behavior of structures we can include them within the same concept.

As far as Prolog syntax is concerned, *predicates* are syntactically equal to structures but semantically different. *Predicates* represent statements about entities of the domain (i.e., *terms*) and are characterized by a given arity. Suppose the arity of a predicate $p$ is an integer value $n$, we shall denote $p$'s arity as '$p/n$'. Arity may be then a *predicate*'s attribute and we can identify a *predicate* by some sort of identifier in the form $p/n$ as above.

We use *clauses* to define propositions, sets, or relations concerning the entities of the domain of the discourse. *Horn* clauses, in particular, are logic formulæ of the following sorts:

- *facts*: a clause with no body, denoting predicates which are known to hold true;

- *rules*: a clause with both head and body, denoting that a predicate holds true if a number of other predicates hold true;

- *goals*: a clause with no head, denoting a number of predicates to be proven (either true or false).

We need then an entity containing a set of *terms*, *predicates*, and *clauses* (except goals), which is the entity *theory*.

### 3.2.2 Meta-clauses

In the previous chapter we introduced the definition of *meta-rule*, which is a linguistic bias which nowadays is starting to catch on in the world of ILP.

As far as we know there is no logic engine natively supporting the representation of a *meta-rule* as first class abstractions—i.e., similarly to how one can express ordinary clauses. This is not in itself a problem because we aim to develop a library detached from Prolog, providing a higher-level view on MIL.

We must then provide an abstraction for representing *meta-rule*s. A *meta-rule* is very similar to an ordinary clause, with the difference that a *meta-rule* presents some second-order (higher-order) variables in its head and/or in its body. At the current state of the art we did not find evidence of the valuability of a *meta-goal* (i.e., a goal containing second-order variables), nor we can identify it by ourselves yet. Furthermore, since the most common *meta-rule*s are not *meta-facts* (i.e., facts containing second-order variables), we can focalize on the *meta-rule* entity itself, for simplicity sake. Since a *meta-fact* is basically a *meta-rule* with `true` body, we can easily express a *meta-fact* as an ordinary *meta-rule*. Anyway, for taking under consideration the extreme cases of *meta-goal* and *meta-fact*, it would be a good practice to design the entity representing *meta-rule*s in such a way it could be easily extensible for eventual future works, where the distinction of different kinds of *meta-rule*s might be crucial.

In light of the previous considerations, we provide some *meta-rule* examples at Listing 3.1.

**Second-order variables.** Second-order (higher-order) variables are what distinguishes a *meta-rule* from an ordinary clause. That consideration means that, in order to define an abstraction representing *meta-rule*s, we must introduce an abstraction representing second-order variables first.

Listing 3.1: Some Meta-rule Examples

```
1 A(X, Y) :- B(X, Z), C(Z, Y).
2 f(X, Y) :- A(Y, X).
3 A(X, Y) :- f(Y, X).
4 A(X) :- true.
5 A(X, constant) :- B(10, X, constant).
6 A(struct(X, Y)) :- B(X), C(Y).
```

First-order (ordinary) and second-order variables share the same feature: they are both variables! Considering the examples reported at Listing 3.1, every uppercase letter denotes a variable. We can easily identify as second-order variables each variable acting as functor. But how can we distinguish the kind of variables which are not functor placeholders? We cannot. Suppose ones provides a *meta-rule* to a Prolog parser: if such parser does not support a specific syntax for expressing second-order variables, the parser would not be able to parse the *meta-rule* with a unique solution. That would occur because of many possible combinations of ordinary/second-order variables.

We then have to make a choice: since first-order and second-order variables are very similar concepts, we may decide to either define a syntax for distinguishing second-order variables from ordinary variables, or to express both of them through the same abstraction. Let us consider a common case of higher-order in Prolog. Suppose we define a syntax for second-order variables such as each second-order variable is a string starting with the symbol *$* immediately followed by an uppercase letter i.e. *$Variable*. Suppose we want a MIL algorithm to learn a rule which maps every element $e$ of a list of integers as $e + 2$. We could provide the higher-order *maplist/3* predicate within background knolwedge and a special *meta-rule* with a second-order variable nested in a predicate within bias, i.e listing 3.2 where $\$C$ is the second-order variable nested in a predicate (note that the background knolwedge rules are expressed with arity for simplicity, in a real case an implementation of such rules should be needed). Note that since in listing 3.2 we want the inducer to learn two rules, we have to provide examples for both.

We now identify an alternative. Instead of providing, within the background knolwedge, an higher-order *maplist/3* implementation, we can provide a set of *meta-rule*s for binary operations on lists. Listing 3.3 depicts how we can provide the *meta-rule*s at lines 7 and 8, compensating the cost by reducing the background knowledge size, avoiding to provide *maplist/3*.

***Meta-rule*s with lists.** Expressing *meta-rule*s in the same way we can commonly express clauses in Prolog, we need to be able to support *meta-rule*s containing lists. That means that our entity representing *meta-rule*s could be effectively

Listing 3.2: Map Plus 2 Induction Case – Hypothetical Meta-rule Syntax

```
% background knowledge

maplist/3
succ/2

% meta-rules

$A(X, Y) :- $B(X, Y, $C).
$A(X, Y) :- $B(X, Z), $C(Z, Y).

% expected inductions

f(A, B) :- maplist(A, B, f_1).
f_1(A, B) :- succ(A, C), succ(C, B).
```

Listing 3.3: Map Plus 2 Induction Case – No Hypothetical Meta-rule Syntax

```
% background knowledge

succ/2

% meta-rules

A([], []).
A([X | Xs], [Y | Ys]) :- B(X, Y), C(Xs, Ys).
A(X, Y) :- B(X, Z), C(Z, Y).

% expected inductions
f([], []).
f([X | Xs], [Y | Ys]) :- f_1(X, Y), f(Xs, Ys).
f_1(X, Y) :- succ(X, Z), succ(Z, Y).
```

Listing 3.4: Background Knowledge, Meta-rules and Expected Inductions for append/3 – Meta-rules not Supporting Lists

```
% background knowledge

head([H|_],H).
tail([_|T],T).
empty([]).

% metarules

A(X, Y, Y) :- B(X).
A(Xs, Ys, Zs) :- B(Xs, X), C(Zs, X), D(Xs, Xs1), E(Zs, Zs1), A(Xs1, Ys, Zs1).

% expected induction

myappend(E, L, L) :- empty(E).
myappend(Xs, Ys, Zs) :-
    head(Xs, X), head(Zs, X), tail(Xs, Xs1), tail(Zs, Zs1), myappend(Xs1, Ys, Zs1)
        .
```

exploited providing to ILP algorithms more efficient and compact *meta-rule*s for learning tasks which deal with lists. Being able to provide a compact *meta-rule* containing lists might significantly reduce the required background knowledge and bias size, with an overall positive impact on the performances of an ILP algorithm. Suppose a common case of induction with lists, if we cannot provide *meta-rule*s with lists we are forced to provide some basic list operations within the background knowledge (e.g., head/2, tail/2, etc.) and larger *meta-rule*s. We provide an example for a better explanation. Supposing to verify if an inducer is capable of inducing *append/3* and supposing to provide some positive and negative examples—if the *meta-rule* entity cannot support lists, we need to provide some common operations with lists and larger *meta-rule*s as depicted in Listing 3.4. Instead, if the *meta-rule* entity supports lists, we can provide a less verbose set of background knowledge and bias such as illustrated in Listing 3.5. This more compact solution would easily outperform the one at Listing 3.4 because of less necessary substitutions (due to smaller *meta-rule*s) and no necessary refinements for expressing the induced theory in a more readable way.

### 3.2.3 Substitution

*Substitution*, together with *unification*, is one of the two fundamental mechanisms for manipulating knowledge.

The purpose of a substitution is to denote variables assignments. Applying a substitution to a logic formula (i.e. a term, a predicate, or a Horn cluase) means assigning some variables of the formula with some values, as denoted by some

Listing 3.5: Background knowledge, Meta-rules and Expected Inductions for append/3 – Meta-rules Supporting Lists

```
% empty bk!

% metarules

X([], B, C).
X([A | B], C, [D | E]) :- X(B, C, E).

% expected induction

append([], A, A).
append([A | As], B, [A | Cs]) :- append(As, B, Cs).
```

given substitution: a formula is then rewritten be replacing all its variables with some other terms, as prescribed by a given substitution.

We now provide a formal definition for substitution. Let $\Phi, \Psi, \Psi', \Psi_1, \ldots, \Psi_n$ be logic formulæ (i.e. terms, predicates, or Horn cluases) of any sort, let $p$ be a $n$-ary predication, let $\mathtt{f}$ be a $m$-ary functor, let $\mathtt{k}$ be a constant, let $t$ be a term of any sort, and let $\sigma$ be a (possibly empty) substitution; then

$$\Phi/\sigma = \begin{cases} \Psi/\sigma \Leftarrow \Psi'/\sigma & \text{if } \Phi \equiv \Psi \Leftarrow \Psi' \\ \Psi/\sigma \wedge \Psi'/\sigma & \text{if } \Phi \equiv \Psi \wedge \Psi' \\ p(\Psi_1/\sigma, \ldots, \Psi_n/\sigma) & \text{if } \Phi \equiv p(\psi_1, \ldots, \psi_n) \\ \mathtt{f}(\Psi_1/\sigma, \ldots, \Psi_m/\sigma) & \text{if } \Phi \equiv \mathtt{f}(\psi_1, \ldots, \psi_m) \\ \mathtt{k} & \text{if } \Phi \equiv \mathtt{k} \\ t & \text{if } \Phi \equiv X \text{ and } (X \mapsto t) \in \sigma \\ X & \text{if } \Phi \equiv X \text{ and } (X \mapsto t) \notin \sigma \end{cases}$$

### 3.2.4 Unification

After discussing about *substitution*, we need to analyze the other fundamental mechanism for manipuplatin knowledge: *unification*.

The purpose of unification is to compute which variables assignment may let a given formula be equal to another one, hence fitting a particular context. The operation may compute:

- an unifier among any two formulæ (i.e. a substitution making the two formulæ syntactically equal);
- or figure out that it is impossible to do so.

Here a formal definition for unification.

$$\text{unify}(\Phi, \Psi) = \begin{cases} \sigma & \text{if } \exists\sigma : \Phi = \Psi/\sigma \\ \square & \text{otherwise} \end{cases}$$

where:

- $\sigma$ is called **unifier** (i.e., the unifying substitution);
- $\square$ denotes the **failed** substitution (i.e., the impossibility to unify).

It is important to remark that in the general case, several unifieriers may exist for any 2 formulæ. We are commonly interested in the *most general unifier* (MGU), usually computed via the algorithm proposed by Martelli and Montanari in 1982 [15].

Unfortunately the basic MGU is not enough for introducing *meta-rule*s or *meta-clauses* in general. For this purpose, the table at Table 3.1 illustrates the MGU formal definition, that we extended with *meta-clauses*, where:

- $\Phi, \Psi$ are arbitrary formulæ;
- $f, g$ are either functors or predications (or logic connectives of any sort);
- $t_1, \ldots, t_n$ and $x_1, \ldots, x_m$ are terms.

| $\mathrm{mgu}(\Phi, \Psi)$ | $\Phi \equiv \mathtt{c}$ | $\Phi \equiv X$ | $\Phi \equiv f(t_1, \ldots, t_n)$ | $\Phi \equiv \mathbf{X}(t_1, \ldots, t_n)$ |
|---|---|---|---|---|
| $\Psi \equiv \mathtt{k}$ | $\begin{cases} \varnothing & \text{if } \mathtt{c} = \mathtt{k} \\ \square & \text{if } \mathtt{c} \neq \mathtt{k} \end{cases}$ | $\{X \mapsto \mathtt{k}\}$ | $\square$ | $\square$ |
| $\Psi \equiv Y$ | $\{Y \mapsto \mathtt{c}\}$ | $\{X \mapsto Y\}$ | $\{Y \mapsto f(t_1, \ldots, t_n)\}$ | $\begin{cases} \square & \text{if } \mathbf{X} \equiv Y \\ \{\mathbf{X} \mapsto Y\} & \text{otherwise} \end{cases}$ |
| $\Psi \equiv g(x_1, \ldots, x_m)$ | $\square$ | $\{X \mapsto g(x_1, \ldots, x_m)\}$ | $\begin{cases} \bigcup_i \mathrm{mgu}(t_i, x_i) & \text{if } f = g \wedge n = m \\ & \quad \wedge \nexists j : \mathrm{mgu}(t_j, x_j) = \square \\ \square & \text{otherwise} \end{cases}$ | $\begin{cases} \{\mathbf{X} \mapsto \mathtt{g}\} \wedge \bigcup_i \mathrm{mgu}(t_i, x_i) & \text{if } n = m \\ & \quad \wedge \nexists j : \mathrm{mgu}(t_j, x_j) = \square \\ \square & \text{otherwise} \end{cases}$ |
| $\Psi \equiv \mathbf{Y}(x_1, \ldots, x_m)$ | $\square$ | $\begin{cases} \square & \text{if } X \equiv \mathbf{Y} \\ \{X \mapsto \mathbf{Y}\} & \text{otherwise} \end{cases}$ | $\begin{cases} \{\mathbf{Y} \mapsto \mathtt{f}\} \wedge \bigcup_i \mathrm{mgu}(t_i, x_i) & \text{if } n = m \\ & \quad \wedge \nexists j : \mathrm{mgu}(t_j, x_j) = \square \\ \square & \text{otherwise} \end{cases}$ | $\begin{cases} \{\mathbf{X} \mapsto \mathbf{Y}\} \wedge \bigcup_i \mathrm{mgu}(t_i, x_i) & \text{if } n = m \\ & \quad \wedge \nexists j : \mathrm{mgu}(t_j, x_j) = \square \\ \square & \text{otherwise} \end{cases}$ |

Table 3.1: MGU Formal Definition - Extended with Meta-clauses

Listing 3.6: Two Examples of Structural Equality

```
% the following two rules are structurally equal
% note the different variable distribution!

f(A, B) :- g(A), h(B).
f(A, B) :- g(B), h(A).


% the following two rules are structurally equal
% note the different variable distribution!

f(A, constant, [B]) :- g(A), h(B, 1).
f(A, constant, [B]) :- g(B), h(A, 1).
```

### 3.2.5 Operations on terms

We know describe some useful utility operations we need for operating with the entities described in the previous sections of this chapter.

**Visiting terms.** Since the recursive nature of Prolog terms (they are usually composed in nested fashion) we need a way to classify, navigate and perform different operations on terms at runtime. We could then apply to our Term abstraction the *Visitor* pattern.

That would be useful in a plethora of cases, expecially when generalizing or refining theories. Suppose to have to generalize a variable which is located inside a literal of a rule, within a list of lists. In such case, being able to deeply explore the structure of a Term would definitely come in handy.

**Structural, strict structural and semantical equality.** In order to be able to induce distinct theories and distinct clauses within the same theory, we have to design some methods to prove if two clauses are equal.

We define *structural equality* the relation of equality between two terms which have the same structure, i.e., the same functors in the literals with the same arity, with the same constants – occuring in the same order between the two terms. Listing 3.6 depicts some examples of structural equality.

Structural equality is a weak equality relation since it is:

- not sensitive to variable distribution;
- sensitive to literals order.

We define *strict structural equality* the relation of equality between two structurally equals terms whose variables occur following the same distribution, without assumptions on the variable naming between the two terms (i.e., variables with

Listing 3.7: Two Examples of Strict Structural Equality

```
% the following two rules are strict structurally equal
% note the same variable distribution!

f(A, B) :- g(A), h(B).
f(C, D) :- g(C), h(D).


% the following two rules are strict structurally equal
% note the same variable distribution!

f(A, constant, [B]) :- g(A), h(B, 1).
f(C, constant, [D]) :- g(C), h(D, 1).


% the following two rules are not strict structurally equal

f(A, B, B).
f(C, C, D).
```

Listing 3.8: Two Semantically Equal Rules, the Order of the Literals in the Body is Irrelevant

```
% the following two rules are structurally equal
% note the same variable distribution but different literal order

f(A, B) :- g(A, B), h(B, A).
f(C, D) :- h(D, C), g(C, D).
```

the same name in the two terms). We denote strict structural equality among any two formulæ as $\Phi \overset{strict}{\equiv} \Psi$. Listing 3.7 depicts some examples of strict structural equality.

Structural equality is a non-complete equality relation since it is sensitive to literals order.

We define *semantical equality* the relation of equality between two strict structurally equals terms with the same body literals without assumptions on their order. We denote semantical equality among any two formulæ as $\Phi \overset{sem}{\equiv} \Psi$. Listing 3.8 depicts some examples of semantical equality.

We can now extend the meaning of *semantical equality* over theories. Let two theories $\mathcal{T}_1$ and $\mathcal{T}_2$, let $\Phi_1, ..., \Phi_n$ the clauses within $\mathcal{T}_1$, let $\Psi, ..., \Psi_m$ the clauses

Listing 3.9: Some Weight Examples

```
f([1, 2, 3], a, A) :- g(a, A). % 7
f([A|As], a, A) :- g(a, A). % 6
f([A, 2 | As]) :- g(A, []). % 5
f([[1, 2, F], [a, 4, b], g(4)]) :- g(a, A, []). % 10
```

within $\mathcal{T}_2$; then $\mathcal{T}_1$ and $\mathcal{T}_2$ are semantically equals if:

$$\mathcal{T}_1 \overset{sem}{\equiv} \mathcal{T}_2 = \begin{cases} \Phi_1 \overset{sem}{\equiv} \Psi_1 & \text{if } n = m = 1 \\ \forall \Phi_i \; \exists \Psi_j \text{ s.t. } \Phi_i \overset{sem}{\equiv} \Psi_j \wedge & \\ \qquad (\mathcal{T}_1 - \{\Phi_i\}) \overset{sem}{\equiv} (\mathcal{T}_2 - \{\Psi_j\}) & \text{if } n = m > 1 \\ \bot & \text{otherwise} \end{cases}$$

**Clause weight.** The validation phase is the phase of an ILP algorithm in which the hypotheses are validated on the basis of the provided background knowledge, positive, and negative examples. Therefore, when validating a theory, the validator has the task to prune the wrong and redundant induced rules.

Supposing we decide to design some sort of validator, we need to make a choice on which rules are redundant and, which rules among them, actually we should prune. Applying the law of parsimony we may decide to keep, from a pool of redundant non-semantically equal rules, the simplest one, pruning the others, in an occamistic fashion.

For that purpose we define *weight* the sum of all occurrences of the constants and variables within a term, including single empty lists (note: we do not identify functors as constants). We provide some examples of weight in listing 3.9.

Let $t_1$ a term with $w_1$ weight, let $t_2$ a term with $w_2$ weight: if $w_1 > w_2$ then $t_1$ is simpler than $t_2$.

**Literal combinations.** The refinement phase is the phase of an ILP algorithm in which the validated hypotheses are subjected to a perfecting process which makes them more similar to what a skilled human programmer would produce while coding. One of the most common best practices in writing code is code reuse. Code reuse aims to reduce redundancy by taking advantage of assets that have already been created in some form within the software development process.

For identifying code which can be reused within an induced theory, we may design an algorithm capable of finding and combining all the literals of a term. With such algorithm it would be possible to identify the most occurring literals and sets of literals, for applying some sort of refinement in an efficient way.

Since we already defined that two rules with the same body literals arranged in different orders are semantically equals (Listing 3.8), we can establish that the

Listing 3.10: Permutations of 3 Literals – the Order is Relevant

```
f(a, b) :- g(a), h(a, b), i(b).

% permutations of the body literals of f/2:

% size 1
g(a)
g(a, b)
i(b)

% size 2
g(a), h(a, b)
h(a, b), g(a)
g(a), i(b)
i(b), g(a)
h(a, b), i(b)
i(b), h(a, b)

% size 3
g(a), h(a, b), i(b)
g(a), i(b), h(a, b)
h(a, b), g(a), i(b)
h(a, b), i(b), g(a)
i(b), g(a), h(a, b)
i(b), h(a, b), g(a)
```

literal combinations we need are a subset of literal permutations. Furthermore, let $n$ the number of literals in the body of a clause, we need to compute only the combinations with size $s$ such as $s \leq n$ since adding literals to the body of a rule would change the rule itself.

After these considerations, we can deduce that a good algorithm for verifying *semantical equality* of two clauses (i.e. based upon some sort of sorting order) would let us significantly reduce the computational cost for computing all literals combinations. Listing 3.10 depicts how many possible permutations of size $s <= 3$ can be defined based on 3 literals. Limiting to computing combinations instead of permutations would allow us to define a smaller solution, as illustrated in Listing 3.11.

## 3.3 Inducer

After identifying the necessary abstractions, it is now possible to engineer a design for our ILP library. Figure 3.1 illustrates the designed hierarchy, containing all the induction phases we previously identified.

In the following sections we provide a deeper explanation for the more relevant entities composing the reported hierarchy. For now we skip over Prolog entities, which we discuss more in detail later on.

Listing 3.11: Combinations of 3 Literals – the Order is not Relevant

```
f(a, b) :- g(a), h(a, b), i(b).

% combinations of the body of f/2:

% size 1
g(a)
g(a, b)
i(b)

% size 2
g(a), h(a, b)
g(a), i(b)
h(a, b), i(b)

% size 3
g(a), h(a, b), i(b)
```

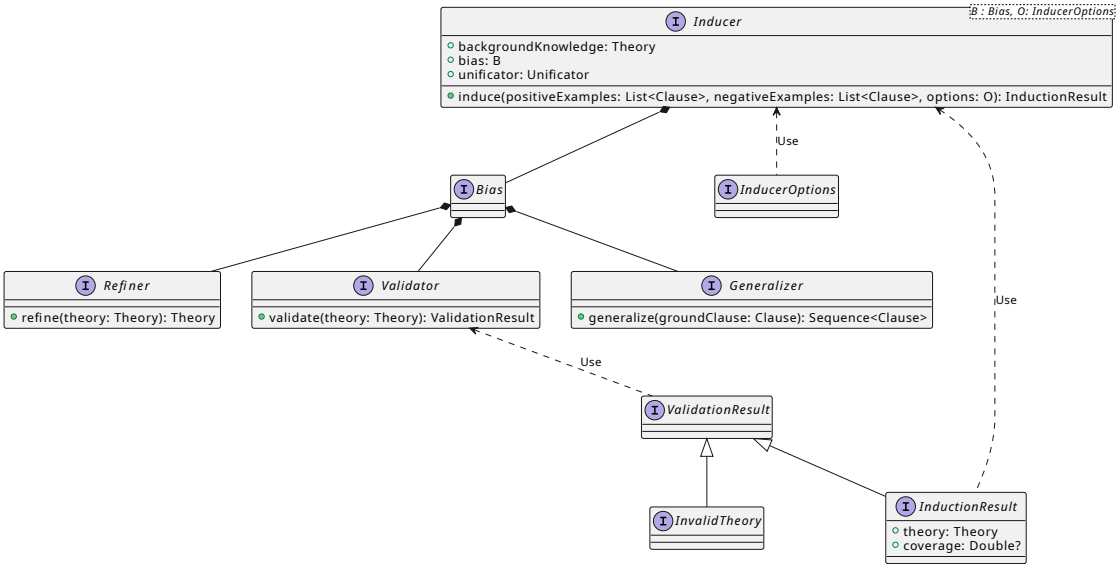


Figure 3.1: Inducer – Hierarchy

### 3.3.1 Inducer

The purpose of an `Inducer` is to solve learning problems, in particular ILP learning problems. The most important feature of an `Inducer` is its induction engine, which can be called by the method `induce(...)`.

**Background Knowledge and Examples.** In order to be able to successfully solve learning tasks, an `Inducer` needs some prior knowledge about the problem domain. We provide such knowledge in the form of `backgroundKnowledge` and examples, in particular divided into `positiveExamples` and `negativeExamples`. As deductible from Figure 3.1, our interpretation of `backgroundKnowledge` is a some prior general-purpose knowledge, which can be useful in different learning tasks. For that reason we decide to represent `backgroundKnowledge` as a field of `Inducer` instead of as an argument for the `induce(...)` method (unlike positive and negative examples).

**Bias.** As stated in the previous chapters, an `Inducer` may need some mechanisms for reducing the hypothesis space, in order to explore only a subset of all the possible solutions for efficiency sake i.e. *bias*. For that purpose we designed the entity `Bias`. Since the `Bias` composition strictly depends on the theoric induction mechanisms of any particular ILP algorithm, we do not provide a strict implementation of the `Bias` interface, instead, we allow great freedom of customization via the use of generics. Anyway, since the main purpose of this thesis is to design and implement a library supporting ILP, we still provide some entities which can be plugged within a `Bias`, such as `Generalizer`, `Validator` and `Refiner`, which we discuss more in detail in the next sections of this chapter.

**Options.** Similarly to what concerns `Bias`, an induction engine may need to have pluggable values/components strictly related to the induction technique it adopts, which are more "practical" than what we usually provide within `Bias`. A simple example could be a maximum depth value in the case of an inducton algorithm exploring a tree of substitutions. The depth value would avoid the case in which some branch of the tree has an infinite path. We may want those values/components to be pluggable. For that purpose we designed the hierarchy so we can provide those values/components `Options`. As for the `Bias`, we do not provide a strict implementation of the `Options` interface, instead, we allow great freedom of customization via the use of generics.

**Result.** As depicted in Figure 3.1 the `induce(...)` method returns an instance of `InductionResult`. We now focalize on `InductionResult` itself. We discuss the interface `ValidationResult` it implements later on.

An induction engine returns:

- one or more `Theory`(/ies), if it successfully found one or more solutions to the learning task;
- no `Theory`, if it failed in trying to find a solution to the learning task.

In the first case, the designer of a particular `Inducer` implementation may provide an algorithm which accepts partial solutions, as well as complete solutions. A partial solution is a solution which covers only a sub-set of the povided `positiveExamples`. A complete solution, on the other hand, is a solution which covers all the povided `positiveExamples`. For that purpose we provide an optional `InductionResult` field representing the percentage of coverage of the proposed `Theory` over the `positiveExamples`: we call it `coverage`. Each `InductionResult` features a single `Theory` with its `coverage`. If the `Inducer` is designed for returning multiple results it will return a `Sequence<InductionResult>` (differently from what reported in Figure 3.1 which is a simplified design for more overall clarity).

### 3.3.2 Generalizer

The *generalization* phase is the phase of an MIL algorithm in which the ground (or "partially ground") clauses, produced after substitutions from positive examples and/or background knowledge on a *meta-rule*, are transformed in more general rules, for making them cover a larger set of examples.

Generalizing a ground clause means substituting all the occurrences of a set of its ground subterms (most of the time all of them) with variables.

Generally, MIL algorithms tend to provide a greedy and simple generalization: after finding a ground rule substituting all the terms of a *meta-rule* with matching terms from background knowledge and positive examples, the generalization step is a simple mapping of the starting *meta-rule* with the ground rule functors. We provide an example of this kind of generalization in listing 3.12.

Unfortunately, this generalization mechanism is too *meta-rule* dependent. Suppose that the *meta-rule* in Listing 3.12 was a less constraining one. As reported in Listing 3.13 the generalized rule would be wrong!

Therefore, we need to provide some generalization mechanisms which are not *meta-rule* dependent. A further quality of a non *meta-rule* dependent generalization is the possibility of its application over a wider set of ILP algorithms.

As previously stated, generalization means substituting a set of ground subterms (of a clause) with variables. Therefore, a clause can be generalized in multiple different ways, with as many generalized versions of the initial clause. Consider the ground clause in Listing 3.14. The rule at line 8 is a simple generalization

Listing 3.12: Simple Generalization by Mapping the Functors of a Meta-rule

```
% starting meta-rule

X(A, B) :- Y(A, C), Z(C, B).

% ground substitution

grandparent(jack, albert) :- parent(jack, ann), parent(ann, albert).

% second-order variables (functors) substitutions:
% X = grandparent
% Y = parent
% Z = parent

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

Listing 3.13: Wrong Generalization by Mapping the Functors of a too Permissive Meta-rule

```
% starting meta-rule

X(A, B) :- Y(C, D), Z(D, E).

% ground substitution

grandparent(jack, albert) :- parent(jack, ann), parent(ann, albert).

% second-order variables (functors) substitutions:
% X = grandparent
% Y = parent
% Z = parent

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

Listing 3.14: A Clause can be Generalized in Multiple Ways

```
% ground rule to generalize:

f([1, 2, 3], [1, 2]) :- g([2, 3], [1]).


% some possible generalizations:

f([A, B, C], [A, B]) :- g([B, C], [A]).
f([A, B | C], [A | B]) :- g([B | C], [A]).
f([A | B], [A | C]) :- g(B, [A]).
```

Listing 3.15: A Clause which can be either a Fact or a Rule

```
% ground rule to generalize:

f(1, g(1, true)) :- true.

% possible generalizations:

f(A, g(A, B)) :- B.
f(A, g(A, true)).
```

mapping each constant number to a variable. The rule at line 9 provides a generalization with the last element of each list as tail. The rule at 10 is a generalization of each list in the common form $[head \mid tail]$.

Since starting from the same ground rule various possible generalizations exist, knowing which generalization to prefer in which context is an extremely difficult issue. Imagine a grund rule containing `true` both in its head and as unique element of its body. As illustrated in Listing 3.15 we could interpret the `true` within the body of the clause as a variable occurring 2 times, or we could interpret the whole clause as a fact keeping the head `true` as a constant!

Therefore we identify and design a set of different `Generalizer`s, each with a specific generalization method. All the designed `Generalizer`s can be plugged *"ad libitum"* within `Bias` to an `Inducer`, depending on which ones the user considers to be more suitable to the learning task. Figure 3.2 illustrates the whole `Generalizer` hierarchy.

Since the whole `Generalizer` hierarchy looks very intricate, we discuss about smallest portions of it. We identify 3 main portions:

- utility generalizers: abstract or instantiable generalizers exploited by other generalizers for maximizing code reuse;
- basic generalizers: operating directly on constants such as strings and numbers;

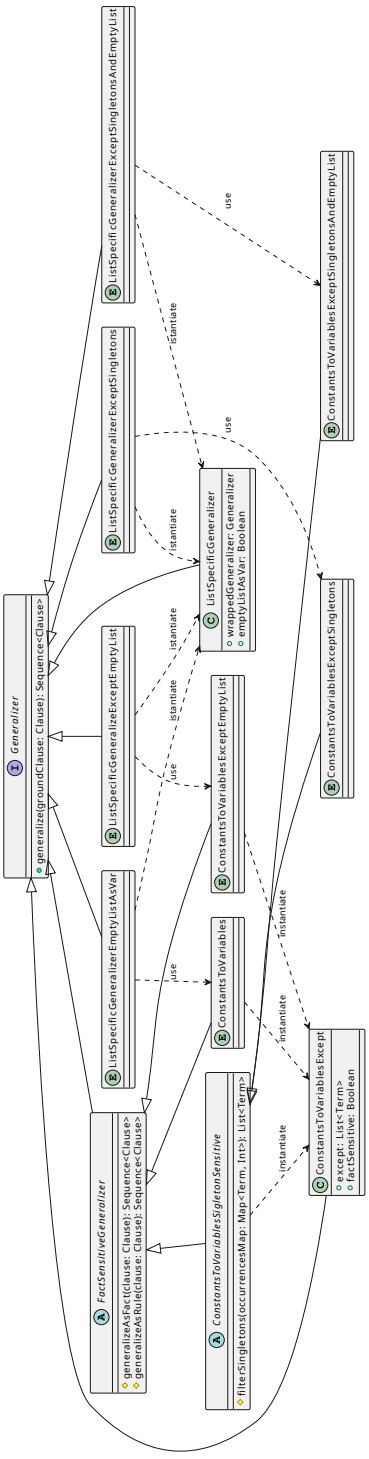


Figure 3.2: Generalizer – Hierarchy

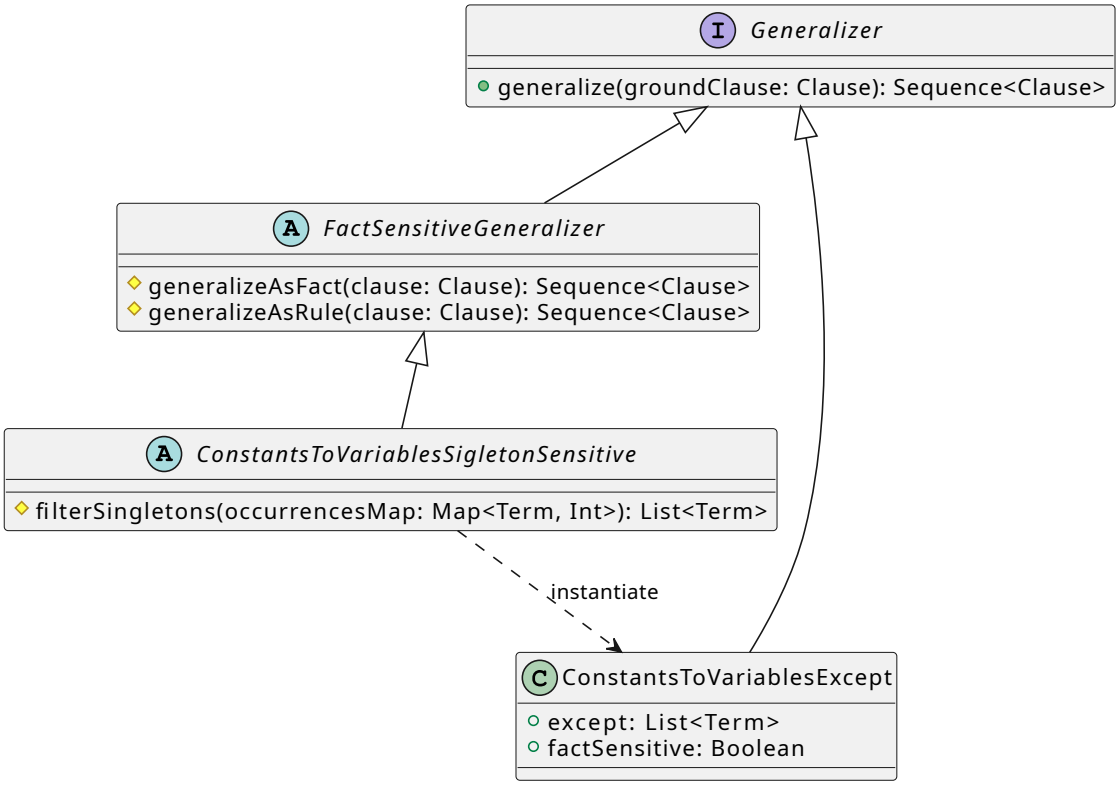


Figure 3.3: Generalizer – Utility Generalizers Hierarchy

- list specific generalizers: exploiting the previous generalizers and generating more readable list expressions whereas possible.

### Utility Generalizers

What we call *utility generalizers* are basic abstract or instantiable `Generalizers` which can be implemented or instantiated by/within other `Generalizers`. These utility generalizers collect the most common generalization methods maximing code reuse. Figure 3.3 illustrates the utility generalizers hierarchy.

**FactSensitiveGeneralizer.** As previously discussed, providing an example in Listing 3.15, certain clauses may be generalized either as rules or as facts. Since the most appropriate generalization may vary depending on the circumstances (i.e. different bias, different positive examples etc.) we design a `FactSensitiveGeneralizer` which generalize such ground clauses in both ways.

`FactSensitiveGeneralizer` is abstract, that means that it evaluates if a clause may be generalized both as a fact or as a rule, and it forks the two possible

solutions by the two abstract methods:

- `generalizeAsFact(...)`
- `generalizeAsRule(...)`

provided by its implementation.

**ConstantsToVariablesExcept.** Sometimes we may want to partially generalize a ground rule, leaving some constants alone. That may occur for various reasons, for example we may want to avoid ti generalize constants which occur only once within a clause.

Therefore we provide the class `ConstantsToVariablesExcept` which is an utility class whose method `generalize(...)` generalizes a ground clause, substituting each occurrence of each constant with an appropriate variable, except for a list of `Term`s provided within the field `except`.

Due to the widespread use of this class among the other Generalizers via delegation, and due to the abstract class `FactSensitiveGeneralizer` only exposing abstract methods, we design `ConstantsToVariablesExcept` for being able to operate either as a fact sensitive generalizer or not. In order to achieve that we provide the class with the `factSensitive` boolean field.

Listing 3.16 depicts some example of generalizations, reported as Prolog code, applied by `ConstantsToVariablesExcept`.

**ConstantsToVariablesSingletonsSensitive.** One of the most common cases in which we may want to avoid generalizing all the constants of a ground clause, is when one or more constants within such clause occur only one time. We call those constants *singletons*.

Therefore we design a `Generalizer` for dealing with such cases, we call it `ConstantsToVariablesSingletonsSensitive`. The purpose of the abstract class `ConstantsToVariablesSingletonsSensitive` is to count all the occurrences of constants such as strings, numbers, booleans and empty lists. We then define an abstract method `filterSingletons` which should be overridden for filtering the desired singleton to avoid to generalize.

Counting the occurrences of all strings, numbers and empty lists is simple and produces a unique result. Counting booleans may produce different results instead, based on the position of `true`(s) within the clause and based on if it is chosen to count it or not. As previously stated, if a `true` is the single element of the body of a clause, we can generalize that clause either as a fact or as a rule. For that reason `ConstantsToVariablesSingletonsSensitive` extends the abstract class `FactSensitiveGeneralizer`, counting the occurrences in different ways in the overridden methods `generalizeAsFact(...)` and `generalizeAsRule(...)`.

Listing 3.16: ConstantsToVariableExcept – Examples

```
% ground clause

f(1, s) :- g(1).

% constants to variables except "s" generalization

f(A, s) :- g(A).

% -----------------------------------------------------
% ground clause

f([1, 3], s) :- g(1, t), f([3]).

% constants to variables except "s, t" generalization

f([A, B | E], s) :- g(A, t), f([B | E]).


% -----------------------------------------------------
% ground clause

f([1, 2], true) :- true.

% constants to variables except nothing, fact sensitive
f([A, B | E], C).

% constants to variables except nothing, not fact sensitive
f([A, B | E], C) :- C.
```

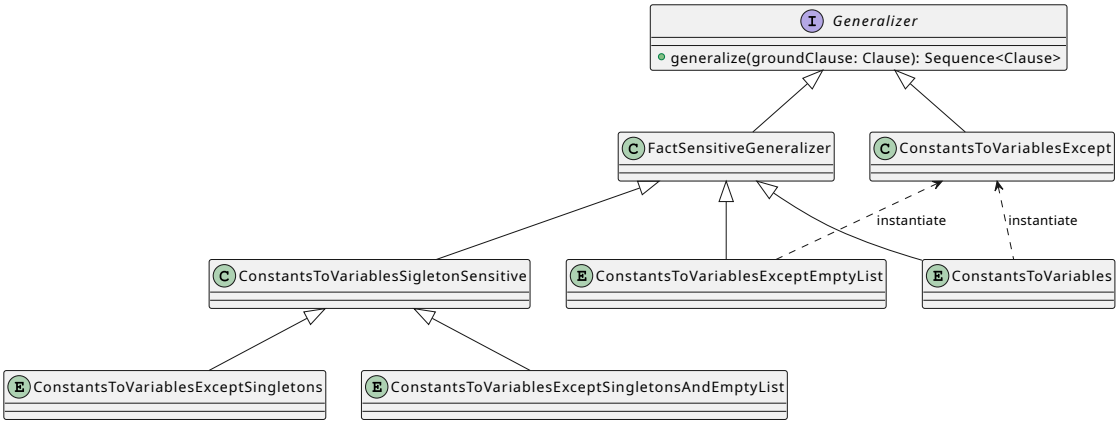


Figure 3.4: Generalizer – Basic Generalizers Hierarchy

Furthermore, since the purpose of the class is to generalize all constants to variables except singletons, after filtering the singletons we do not want to generalize via the abstract method `filterSingletons`, we should delegate the actual generalization to a `ConstantsToVariablesExcept` instance.

### Basic Generalizers

In this section we discuss about some basic `Generalizer`s which can be plugged by `Bias` to any `Inducer`. In particular, referring to what reported in Figure 3.4:

- `ConstantsToVariables`
- `ConstantsToVariablesExceptEmptyList`
- `ConstantsToVariablesExceptSingletons`
- `ConstantsToVariablesExceptSingletonsAndEmptyList`

**ConstantsToVariables.** `ConstantsToVariables` is a `Generalizer` which generalizes every constant as a variable, including empty lists. Each occurrence of the same constant will be generalized with the same variable.

`ConstantsToVariables` is fact sensitive and should delegate its generalization to an instance of the class `ConstantsToVariablesExcept`.

Listing 3.17 depicts some examples of generalizations, reported as Prolog code, applied by `ConstantsToVariables`.

Listing 3.17: ConstantsToVariables – Examples

```
% ground clause

f(1, s) :- g(1).

% constants to variables generalization

f(A, B) :- g(A).

% ------------------------------------------------
% ground clause

f([1, 3], s) :- g(1, t), f([3]).

% constants to variables generalization

f([A, B | E], C) :- g(A, D), f([B | E]).


% ------------------------------------------------
% ground clause

f([1, 2, 5], true) :- true.

% constants to variables, as fact generalization
f([A, B, C | E], D).

% constants to variables, as rule generalization
f([A, B, C | E], D) :- D.
```

Listing 3.18: ConstantsToVariablesExceptEmptyListExamples – Examples

```
% ground clause

f(1, s) :- g(1).

% constants to variables except [] generalization

f(A, B) :- g(A).

% -------------------------------------------------
% ground clause

f([1, 3], s) :- g(1, []), f([3]).

% constants to variables except [] generalization

f([A, B], C) :- g(A, []), f([B]).


% -------------------------------------------------
% ground clause

f([1, 2, 5], [], true) :- true.

% constants to variables except [], as fact
f([A, B, C], [], D).

% constants to variableexcept [], as rule
f([A, B, C], [], D) :- D.
```

**ConstantsToVariablesExceptEmptyList.** `ConstantsToVariablesExceptEmptyList` is a `Generalizer` which generalizes every constant as a variable, excluding empty lists. Each occurrence of the same constant will be generalized with the same variable.

`ConstantsToVariablesExceptEmptyList` is fact sensitive and should delegate its generalization to an instance of the class `ConstantsToVariablesExcept`.

Listing 3.18 depicts some examples of generalizations, reported as Prolog code, applied by `ConstantsToVariablesExceptEmptyList`.

**ConstantsToVariablesExceptSingletons.** `ConstantsToVariablesExceptSingletons` is a `Generalizer` which generalizes every constant as a variable, including empty lists. Each occurrence of the same constant will be generalized with the same variable. Constants and empty lists will not be generalized if they occur only once within the clause.

`ConstantsToVariablesExceptSingletons` extends `ConstantsToVariablesSingletonSensitive` such that the method `filterSingletons(...)` filters the map of occurrences (`Map<Term, Int>`) retrieving only the `Terms` occuring one time.

Listing 3.19 depicts some examples of generalizations, reported as Prolog code,

Listing 3.19: ConstantsToVariablesExceptSingletons – Examples

```
% ground clause

f(1, 2) :- g(1).

% constants to variables except singletons generalization

f(A, 2) :- g(A).

% -------------------------------------------------
% ground clause

f([1, 3], s) :- g(1, []), f([3]).

% constants to variables except singletons generalization

f([A, B | E], s) :- g(A, E), f([B | E]).

% -------------------------------------------------
% ground clause

f([], s) :- g(s).

% constants to variables except singletons generalization

f([], A) :- g(A).


% -------------------------------------------------
% ground clause

f([1, 1], true) :- true.

% constants to variables except singletons, as fact
f([A, A | []], true).

% constants to variables except singletons, as rule
f([A, A | []], B) :- B.
```

applied by `ConstantsToVariablesExceptSingletons`.

**ConstantsToVariablesExceptSingletonsAndEmptyList.** `ConstantsToVariablesExceptSingletonsAndEmptyList` is a `Generalizer` which generalizes every constant as a variable, excluding empty lists. Each occurrence of the same constant will be generalized with the same variable. Constants will not be generalized if they occur only once within the clause.

`ConstantsToVariablesExceptSingletonsAndEmptyList` extends `ConstantsToVariablesSingletonSensitive` such that the method `filterSingletons(...)` filters the map of occurrences (`Map<Term, Int>`) retrieving only the `Terms` (different from empty list) occuring one time.

Listing 3.20 depicts some examples of generalizations, reported as Prolog code,

Listing 3.20: ConstantsToVariablesExceptSingletonsAndEmptyList – Examples

```
% ground clause

f(1, 2) :- g(1).

% constants to variables except singletons and [] generalization

f(A, 2) :- g(A).

% ------------------------------------------------
% ground clause

f([1, 3], s) :- g(1, []), f([3]).

% constants to variables except singletons and [] generalization

f([A, B], s) :- g(A, []), f([B]).

% ------------------------------------------------
% ground clause

f([], s) :- g(s).

% constants to variables except singletons and []generalization

f([], A) :- g(A).


% ------------------------------------------------
% ground clause

f([1, 1], true) :- true.

% constants to variables except singletons and [], as fact
f([A, A], true).

% constants to variables except singletons and [], as rule
f([A, A], B) :- B.
```

applied by `ConstantsToVariablesExceptSingletonsAndEmptyList`.

### List Specific Generalizers

In this section we discuss about some `Generalizers` which can be plugged by `Bias` to any `Inducer`. Each of these `Generalizers` is designed for execute some specifical generalization on lists. In particular, referring to what reported in Figure 3.5:

- `ListSpecificGeneralizerEmptyListAsVar`
- `ListSpecificGeneralizeExceptEmptyList`
- `ListSpecificGeneralizerExceptSingletons`

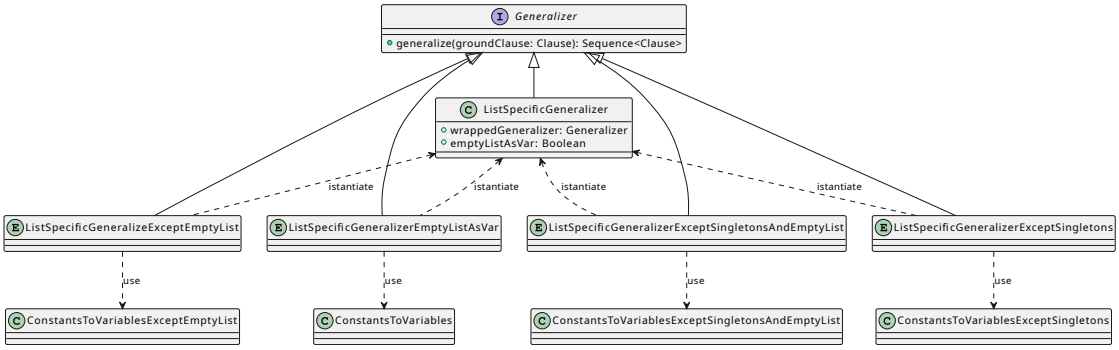


Figure 3.5: Generalizer – List Specific Generalizers Hierarchy

- `ListSpecificGeneralizerExceptSingletonsAndEmptyList`

As easily inferred from Figure 3.5, each of the previously listed `Generalizers` exploits an instance of the class `ListSpecificGeneralizer`. A `ListSpecificGeneralizer` applies a further generalization over the generalization of a wrapped `Generalizer` (as the field `wrappedGeneralizer`). Similarly to the previously mentioned `ConstantsToVariablesExcept`, a `ListSpecificGeneralizer` could be instantiated in different list generalizers, we then design it to detect (via the boolean field `emptyListAsvar`) how it should deal with the empty lists occurrences.

The further generalization applied by the `ListSpecificGeneralizer` is the following:

- for each list $L$ in the form $[A, B, C, ... \mid D]$,
  - if there is already a variable $X$ representing the whole list $L$, generalize $L$ as $X$ (e.g., the list is the tail of another list – in order to detect those cases we should generalize lists sorting them by descending size)
  - otherwise
    - if there is already a variable $Xs$ representing the tail of the list $L$, generalize $L$ as $[A \mid Xs]$
    - otherwise generalize $L$ it in the form $[A \mid As]$
- after generalizing each list as the previous point, apply a further generalization on the lists $[A \mid As]$ whose:
  - head $A$ does not occur in any form different from $[A \mid As]$
  - tail $As$ does not occur in any form different from $[A \mid As]$

  generalizing $[A \mid As]$ as a single variable $A$.

Listing 3.21: ListSpecificGeneralizerEmptyListAsVar – Examples

```
% ground clause

f([1, 3], s) :- g(1, t), h([3]).

% list specific generalizer
% [] as variable generalization

f([A | As], C) :- g(A, B), h(As).


% ---------------------------------------------------
% ground clause

f([1, 2, 3], []) :- g([2, 3], [1]).

% list specific generalizer
% [] as variable generalization

f([A | As], B) :- g(As, [C | B]).


% ---------------------------------------------------
% ground clause

f([1, 2, 5], [], true) :- true.

% list specific generalizer
% [] as variable, as fact generalization
f(A, B, C).

% list specific generalizer
% [] as variable, as rule generalization
f(A, B, C) :- C.
```

Note: the second point of the previous *generalization* technique may be useful in some cases also as a *refinement* technique. Therefore we provide a specific design of it in the next sections of the current chapter.

**ListSpecificGeneralizerEmptyListAsVar.** `ListSpecificGeneralizerEmptyListAsVar` is a `Generalizer` which generalizes every constant as a variable, including empty lists. Each occurrence of the same constant will be generalized with the same variable. Each list will then generalized as illustrated for `ListSpecificGeneralizer` exploiting the latter.

`ListSpecificGeneralizerEmptyListAsVar` exploits the previously discussed `ConstantsToVariables`, so it is fact sensitive.

Listing 3.21 depicts some examples of generalizations, reported as Prolog code, applied by `ListSpecificGeneralizerEmptyListAsVar`.

Listing 3.22: ListSpecificGeneralizerExceptEmptyList – Examples

```
% ground clause

f([1, 3], s) :- g(1, t), h([3]).

% list specific generalizer
% except [] generalization

f([A | As], C) :- g(A, B), h(As).


% ----------------------------------------------------
% ground clause

f([1, 2, 3], []) :- g([2, 3], [1]).

% list specific generalizer
% except [] generalization

f([A | As], []) :- g(As, [A]).


% ----------------------------------------------------
% ground clause

f([1, 2, 5], [], true) :- true.

% list specific generalizer
% except [], as fact generalization
f(A, [], C).

% list specific generalizer
% except [], as rule generalization
f(A, [], C) :- C.
```

**ListSpecificGeneralizerExceptEmptyList.** `ListSpecificGeneralizerExceptEmptyList` is a `Generalizer` which generalizes every constant as a variable, excluding empty lists. Each occurrence of the same constant will be generalized with the same variable. Each list will then generalized as illustrated for `ListSpecificGeneralizer` exploiting the latter.

`ListSpecificGeneralizerEmptyListAsVar` exploits the previously discussed `ConstantsToVariablesExceptEmptyList`, so it is fact sensitive.

Listing 3.22 depicts some examples of generalizations, reported as Prolog code, applied by `ListSpecificGeneralizeExceptEmptyList`.

**ListSpecificGeneralizerExceptSingletons.** `ListSpecificGeneralizerExceptSingletons` is a `Generalizer` which generalizes every constant as a variable, including empty lists. Each occurrence of the same constant will be generalized with the same variable. Each list will then generalized as illustrated for `ListSpecificGeneralizer` exploiting the latter.

Listing 3.23: ListSpecificGeneralizerExceptSingletons – Examples

```
% ground clause

f([1, 3], s) :- g(1, t), h([3]).

% list specific generalizer
% except singletons generalization

f([A | As], s) :- g(A, t), h(As).


% ---------------------------------------------------
% ground clause

f([1, 2, 3], []) :- g([2, 3], [1]).

% list specific generalizer
% except singletons generalization

f([A | As], B) :- g(As, [A | B]).


% ---------------------------------------------------
% ground clause

f([1, 2, 5], [], true) :- true.

% list specific generalizer
% except singletons, as fact generalization
f([1 | As], As, C).

% list specific generalizer
% except singletons, as rule generalization
f([1 | As], As, C) :- C.
```

`ListSpecificGeneralizerExceptSingletons` exploits the previously discussed `ConstantsToVariablesExceptSingletons`, so it is fact sensitive.

Listing 3.23 depicts some examples of generalizations, reported as Prolog code, applied by `ListSpecificGeneralizerExceptSingletons`.

**ListSpecificGeneralizerExceptSingletonsAndEmptyList.** `ListSpecificGeneralizerExceptSingletonsAndEmptyList` is a `Generalizer` which generalizes every constant as a variable, excluding empty lists. Each occurrence of the same constant will be generalized with the same variable. Each list will then generalized as illustrated for `ListSpecificGeneralizer` exploiting the latter.

`ListSpecificGeneralizerExceptSingletonsAndEmptyList` exploits the previously discussed `ConstantsToVariablesExceptSingletonsAndEmptyList`, so it is fact sensitive.

Listing 3.24 depicts some examples of generalizations, reported as Prolog code, applied by `ListSpecificGeneralizerExceptSingletonsAndEmptyList`.

Listing 3.24: ListSpecificGeneralizerExceptSingletonsAndEmptyList – Examples

```
% ground clause

f([1, 3], s) :- g(1, t), h([3]).

% list specific generalizer
% except singletons and [] generalization

f([A | As], s) :- g(A, t), h(As).


% ---------------------------------------------------
% ground clause

f([1, 2, 3], []) :- g([2, 3], [1]).

% list specific generalizer
% except singletons and [] generalization

f([A | As], []) :- g(As, [A]).


% ---------------------------------------------------
% ground clause

f([1, 2, 5], [], true) :- true.

% list specific generalizer
% except singletons and [], as fact generalization
f([1 | As], [], C).

% list specific generalizer
% except singletons and [], as rule generalization
f([1 | As], [], C) :- C.
```

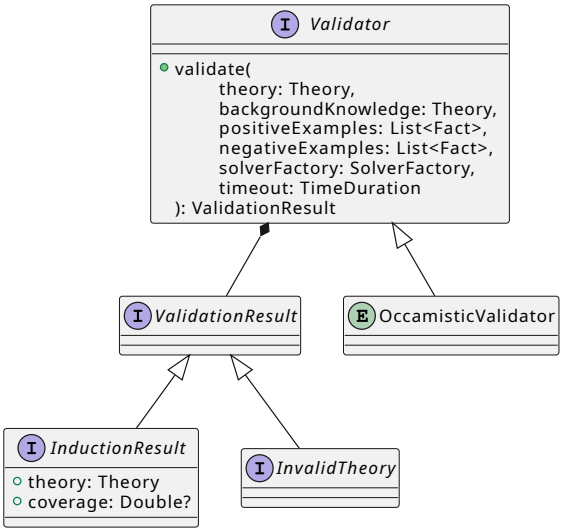


Figure 3.6: Validator – Hierarchy

### 3.3.3 Validator

The *validation* phase is the phase of an ILP algorithm in which the hypotheses are validated on the basis of the provided background knowledge, positive and negative examples. Usually a hypothesis covering any negative example is discarded while a hypothesis covering only a subset of the positive examples may be accepted.

Figure 3.6 illustrates the hierarchy of our `Validator` design.

**Validator.** The `Validator` interface defines the method `validate(...)`. Such method validates the provided `theory`, proving it on `backgroundKnowledge`, `positiveExamples` and `negativeExamples`. Since the purpose of a `Validator` is trying to prove positive examples as goals and trying to fail negative examples (always as goals), both with the provided theory, we design the method `validate(...)` to accept a `solverFactory`. Being able to plug a custom `solverFactory` to a `Validator` makes the latter more versatile.

Furthermore, since we aim to detach MIL from Prolog for having a greater level of abstraction over LP, our `Validator` should be able to detect looping rules within the induced theory. Therefore we design the method `validate(...)` to accept a `timeout`, for recognizing and pruning looping rules on the fly.

**ValidationResult.** Since a theory may be proven valid or invalid, we need an entity for representing both results. The main reason behind not designing the method `validate(...)` with a boolean return type is that during the *validation* phase we do not just prove if a theory is acceptable: we also prune the inadequate

rules, as deductible from the previous discussion about looping rules. Therefore, a `Validator` may edit the induced theory, making it more adequate and only composed by correct rules.

We design `ValidationResult` as an interface implemented by:

- `InductionResult` representing a valid theory;
- `InvalidTheory` representing an invalid theory.

We already discussed about `InductionResult` in Section 3.3.1.

**OccamisticValidator.** `OccamisticValidator` is a `Validator` which validates a theory and prunes all the redundant clauses trying to produce the most simple solution, in an occamistic fashion. `OccamisticValidator` does not require a theory to prove all the positive examples, it is based on the percentage of covered positive examples instead (i.e., coverage). It works as follows:

1. it first checks if the theory is covering negative examples: if the theory covers negative examples the theory is invalid;
2. it proceeds pruning all the clauses semantically equal to any previous clause, within the theory;
3. it proceeds pruning all the clauses which cause infinite loops;
4. it filters the redundant clauses.

We provide a *redundant* definition for clauses within a theory. Let a theory $\mathcal{T}_1$ composed by $n$ clauses $C$ such as $c_1, ...c_n$, let $\mathcal{S}$ the space of possible solutions of $\mathcal{T}_1$. We define *redundant* a clause $c_i$ such as $c_i \in C$, if a theory $\mathcal{T}'$ composed by the $n-1$ clauses $C'$ such as $C' = C - c_i$ has a space of solutions $\mathcal{S}'$ such as $\mathcal{S} \equiv \mathcal{S}'$. Listing 3.25 depicts an example of theory with some redundant clauses. In particular it is *append/3*. The redundant clauses are the two at lines 4 and 5 since they work only on specific cases with empty lists, while the one at line 6 is more general.

When filtering the redundant clauses we prioritize the smaller one, pruning the larger ones. For pruning the redundant clauses we:

1. first compute the initial positive coverage of the whole theory;
2. proceed sorting the clauses by weight, in a descending way;
3. compute the coverage of the theory *without* one clause at a time, basing on the order from point 2: if the coverage of the reduced theory is equal to the initial coverage from point 1 the removed clause can be pruned.

Listing 3.25: *append/3* with Redundant Clauses

```
% append/3 with redundant clauses

append([], A, A).
append([A | As], [], [A | As]) :- append(As, [], As).
append([A | As], B, [A | B]) :- append([], B, B).
append([A | As], B, [A | Cs]) :- append(As, B, Cs).


% optimal append/3

append([], A, A).
append([A | As], B, [A | Cs]) :- append(As, B, Cs).
```

The descending sorting from point 2 allows us to prune the larger redundant clauses first, letting the `OccamisticValidator` producing the smallest valid theory.

Note: the redundant clauses set is larger than the semantically equal clauses one. Therefore, pruning the semantically equals clauses before pruning the redundant clauses may seem inefficient. We decide to prune the semantically equal clauses first because pruning the clauses which cause infinite loops is extremely costly (i.e. given $n$ clauses it may take $n * \texttt{validationTimeout}$ in the worst scenario).

### 3.3.4 Refiner

The *refinement* phase is the phase of an ILP algorithm in which the validated hypotheses are subjected to a perfecting process which makes them more similar to what a skilled human programmer would produce while coding. As for *generalization*, given a theory multiple strategies of refinement exist, and each strategy may be useful in different cases. For example Listing 3.26 illustrates two different predicate invention strategies over the same initial theory, where two clauses have the common literal $h(A, B)$:

- the first method groups the common literal $h(A, B)$ within a new predicate;
- the second method groups the non common literals $g(A)$ and $i(A)$ within a new predicate.

We identify three examples of predicate invention (i.e., the extrapolation of common code among multiple rules in a new specific rule) and a list refiner, although other refinement strategies may, and certainly do exist:

- a predicate inventor grouping the literals in common among multiple clauses within a new predicate;

Listing 3.26: Multiple Predicate Invention Strategies Exist

```
% to refine

f(A, B) :- g(A), h(A, B).
f(A, B) :- i(A), h(A, B).

% possible refinement, grouping common literals

f(A, B) :- g(A), invented(A, B).
f(A, B) :- i(A), invented(A, B).
invented(A, B) :- h(A, B).

% possible refinement, grouping non common literals

f(A, B) :- invented(A), h(A, B).
invented(A) :- g(A).
invented(A) :- i(A).
```

- a predicate inventor grouping the literals not in common among multiple clauses within a new predicate;
- a predicate inventor working as the one proposed by METAGOL, discussed in Section 2.2.3;
- a list refiner working as the second step of the list generalization we already discussed in Section 3.3.2.

Figure 3.7 illustrates the hierarchy of our `Refiner` design.

**ListSpecificRefiner.** The `ListSpecificRefiner` is a `Refiner` with the purpose of simplifying clauses containing lists. In order to achieve that, the `ListSpecificRefiner` expresses lists as single variables, if the variables and the tail of such lists do not occur outside lists equal to the list in question. Therefore, the `ListSpecificRefiner` operates on every single clause of the provided `theory` instead of operating on the theory as a whole. For example, given a list in the form $[A \mid As]$ whose:

- head $A$ does not occur in any form different from $[A \mid As]$
- tail $As$ does not occur in any form different from $[A \mid As]$

the `ListSpecificRefiner` refines $[A \mid As]$ as a single variable $A$.

We provide some examples of the just described refinement method in Listing 3.27.

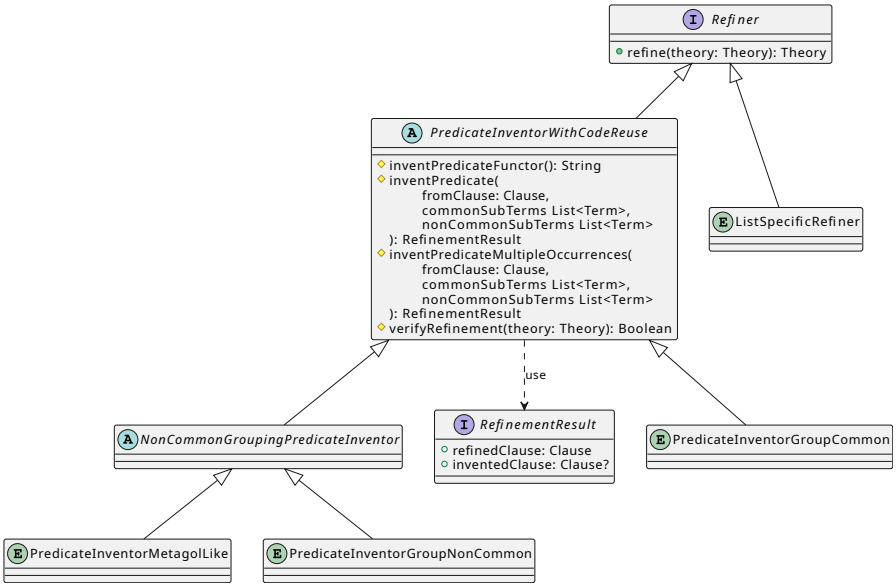


Figure 3.7: Refiner – Hierarchy

Listing 3.27: ListSpecificRefiner – Examples

```
% to refine

append([A | As], [B | Bs], [A | Cs]) :- append(As, [B | Bs], Cs).

% after list specific refiner

append([A | As], B, [A | Cs]) :- append(As, B, Cs).


% ------------------------------------------------
% to refine

f([A, B, C]) :- g(C).

% after list specific refiner
% (C occurs outside of [A, B, C])

f([A, B, C]) :- g(C).


% ------------------------------------------------
% to refine

f([A, B, C], [B, C]) :- g(D).

% after list specific refiner

f([A | As], As) :- g(D).
```

**PredicateInventorWithCodeReuse.** The `PredicateInventorWithCodeReuse` is an abstract `Refiner` grouping all the common mechanisms from the different predicate inventors we identified. It works as follows:

1. for each clause, counts the occurrences of all the possible combinations of its body literals;
2. merges all the occurrences of literals of the theory from point 1;
3. filters the combinations keeping only the combinations occurring more than once;
4. on each clause containing the literals from point 3, applies some sort of predicate invention, via the abstract methods `inventPredicate(...)` etc.

More in detail, the predicate invention phase at point 4 works as follows:

1. if all the literals within the body of the current clause are to refine do nothing, otherwise proceed at point 2;
2. looks for an already existing predicate with the literal combination as body (except for the current clause to refine):
   (a) if any exists, edit the current clause for calling such predicate instead of the literal combination. This is the case of code reuse without predicate invention.
   (b) otherwise, proceed at point 3
3. if a sub-set of the current clause body literals are to refine, apply predicate invention via the abstract methods.

**PredicateInventorGroupCommon.** The `PredicateInventorGroupCommon` is a `PredicateInventorWithCodeReuse` which groups all the common literals among different clauses, within an invented predicate. Listing 3.28 depicts some examples of refinement applied by `PredicateInventorGroupCommon`.

This kind of predicate invention is useful when the common literals are at least 2.

**PredicateInventorGroupNonCommon.** The `PredicateInventorGroupNonCommon` is a `PredicateInventorWithCodeReuse` which groups all the non common literals among different clauses, within an invented predicate. In particular it creates a new predicate composed by multiple clauses with the same head (i.e.

Listing 3.28: PredicateInventorGroupCommon – Examples

```
% to refine

f(A, B) :- g(A), h(A, B), l(B).
f(A, B) :- i(A), h(A, B), l(B).

% after predicate inventor group common

f(A, B) :- g(A), invented(A, B).
f(A, B) :- i(A), invented(A, B).
invented(A, B) :- h(A, B), l(B).


% ------------------------------------------------
% to refine

f(A, B) :- g(A), h(A, B), l(B).
f(A, B) :- i(A), h(A, B), l(B).
m(A) :- l(A).

% after predicate inventor group common
% (note the reuse of m/1)

f(A, B) :- g(A), invented(A, B).
f(A, B) :- i(A), invented(A, B).
invented(A, B) :- h(A, B), m(B).
m(A) :- l(A).


% ------------------------------------------------
% to refine

f(A, B) :- g(A, B), h(A).
i(A, B) :- g(A, B), h(A).

% after predicate inventor group common

f(A, B) :- i(A, B).
i(A, B) :- g(A, B), h(A).
```

Listing 3.29: PredicateInventorGroupNonCommon – Examples

```
% -----------------------------------------------
% to refine

f(A, B) :- g(A), h(A, B), l(B).
f(A, B) :- i(A), h(A, B), l(B).

% after predicate inventor group non common

f(A, B) :- h(A, B), l(B), invented(A).
invented(A) :- g(A).
invented(A) :- i(A).


% -----------------------------------------------
% to refine

f(A, B) :- g(A), h(A, B), l(B).
f(A, B) :- i(A), h(A, B), l(B).
m(A) :- l(A).

% after predicate inventor group non common
% (note the reuse of m/1)

f(A, B) :- h(A, B), m(B), invented(A).
invented(A) :- g(A).
invented(A) :- i(A).
m(A) :- l(A).


% -----------------------------------------------
% to refine

f(A, B) :- g(A, B), h(A).
i(A, B) :- g(A, B), h(A).

% after predicate inventor group non common

f(A, B) :- i(A, B).
i(A, B) :- g(A, B), h(A).
```

practically a "or" among multiple clauses). Listing 3.29 depicts some examples of refinement applied by `PredicateInventorGroupNonCommon`.

This kind of predicate invention is useful when the non common literals have the same number of distinct variables.

**PredicateInventorMetagolLike.** The `PredicateInventorGroupNonCommon` is a `PredicateInventor` which groups all the common literals among different clauses, within an invented predicate. Specifically, it covers a particular case where the clauses literals are all in common, but with different distribution such as the previously discussed example in Listing 2.3, where both $father/2$ and $mother/2$ occur in all clauses, in different combinations. We already provided a predicate invention example for that case, in Listing 2.4.

### 3.3.5 MetaPatrol

MetaPatrol is a brand new ILP algorithm, based on MIL and inspired by Metagol, completely designed on the framework discussed in the previous paragraphs of this chapter.

The induction engine of MetaPatrol is similar to the one proposed by Metagol, but we extended it for exploiting many different generalization, validation e refinement strategies. Furthermore, MetaPatrol is able to induce multiple clauses from different positive examples (i.e., from facts with different head functors) within the same learning task, eventually reusing the already induced clauses for inducing new ones.

The main strength of MetaPatrol is its ability to generate multiple induced theories, if possible, for the same learning task, based on different generalizations and refinement methods. For achieving that, MetaPatrol lazily explores the tree of the possible solutions it is able to induce for the current learning task. Furthermore, different `Generalizator`s, `Validator`s and/or `Refiner`s are pluggable to MetaPatrol in a custom way.

Since our new algorithm is inspired by Metagol, and since the ability of the new algorithm to lazily explore many different solutions to the same learning task, we decided to name it in a similar way to Metagol, as a form of respect, changing only a small part of the name of the latter, to make it a *Patrol* scouting the space of solutions.

For being able to lazily explore a tree of solutions, the MetaPatrol algorithm exploits a context representing the state of the algorithm in each node of the tree. In particular we decide to explore the tree horizontally, exploring the branches at the same depth level, before proceeding deeper in the tree.

The MetaPatrol algorithm works as follows:

1. for each positive example, verify if its already proved
   - if it is proved, proceed with the next example, point 1
   - if it is not proved, proceed at point 2
   - if there are no more positive examples proceed at point 5
2. for each *meta-rule* within bias: substitute its head with the positive example, then try to substitute all the body literals, basing on *background knowledge* and on *already induced clauses*. This generates multiple branches in the solution tree.
3. for each `Generalizer` within bias, generalize each rule with all second-order variables substituted, generated at point 2. This generates multiple branches in the solution tree.
4. add the generalized rule, if it does not prove negatives and if it is new, to the current induced theory. Proceed with the next example, point 1
5. if the theory is new, validate the induced theory with each `Validator` within bias
6. if the theory is valid, refine the validated theory with each `Refiner` within bias, then return the refined theory. This generates multiple branches in the solution tree.

Since the same clauses and theories may be induced following different paths, we provide the points 3 and 5 with some mechanism for recognizing it, in order to avoid duplicates.

Since the points 3 and 5 generate multiple branches, the resulting theories will be composed by different combinations of induced and generalized clauses, in order to try to induce the most adequate theory for solving the current problem, which the user may choose. In fact, for MetaPatrol being lazy, the user might then decide to continue or stop the algorithm at will, if the algorithm already induced (or did not yet) an interesting solution.

Since the `Refiner`s may be inadequate for some theories, MetaPatrol at point 6 (of the previously discussed algorithm) returns the non refined theories too, together with their refined versions.

Figure 3.8 illustrates the `MetaPatrol` hierarchy. As reported in the figure, `MetaPatrol` implements the previously described `Inducer` interface. For achieving that we need to design `MetaPatrol` for exploiting some specific `Bias` and `InducerOptions`. In particular we design a `Bias` implementation: `MetaPatrolBias` composed by the bias strategies we described above: `MetaRule`s, `Generalizer`s,

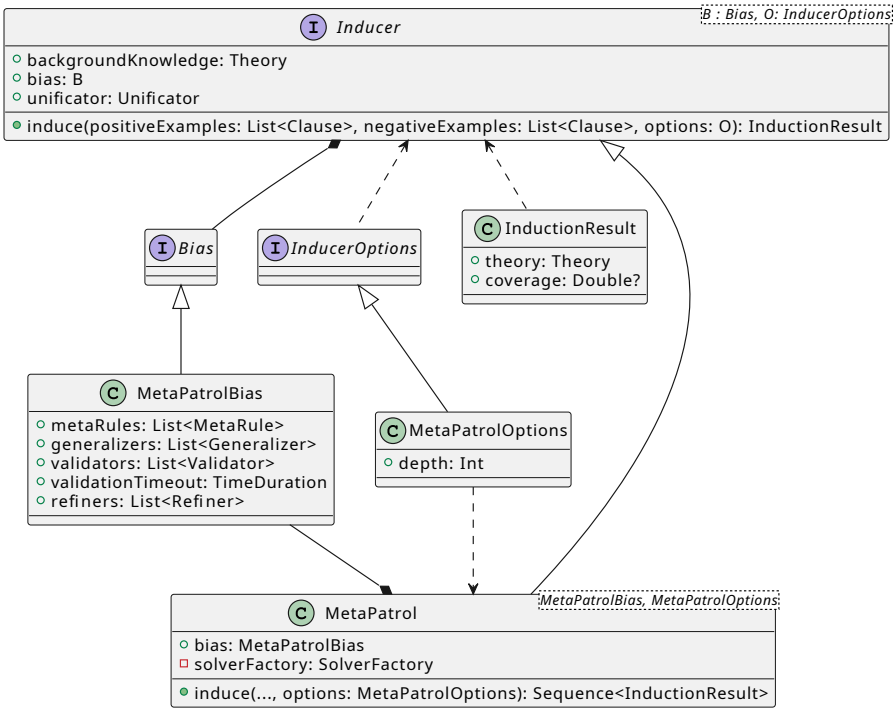


Figure 3.8: MetaPatrol – Hierarchy

`Validator`s and `Refiner`s, as well as the allowed `validatorTimeout`. Then we design an `InducerOptions` implementation: `MetaPatrolOptions`, composed by the maximum `depth` we allow to explore in the solution tree.

Since, in point 4 of the MetaPatrol algorithm described above, the hypoteses are proved on negative examples, we provide `MetaPatrol` a pluggable `solverFactory`. Similary to `Validator`, a pluggable `solverFactory` allows more versatility to MetaPatrol itself.

# Chapter 4

# Implementation

In the current chapter we describe the main implementative choices we adopt for developing the library designed in the previous chapters.

## 4.1 Reducing the Abstraction Gap: 2P-Kt

Since ILP is based on LP, and since we aim to develop a library supporting MIL from a higher perspective, we need various abstractions representing LP entities. 2P-Kt is a framework for LP, aiming to become an ecosystem for symbolic AI. Being 2P-Kt a framework for LP it already features most of the entities we need for our purpose. All we need is to identify the abstraction gap starting from what 2P-Kt already provides, then we can extend the latter for supporting the entities and the operations we need but which 2P-Kt does not feature yet.

## 4.2 Identifying the Abstraction Gap

2P-Kt is composed by several modules, each providing some specific functionalities. We now identify, for each 2P-Kt module we need for developing our framework, the functionalities not provided by 2P-Kt which we need to implement, extending the latter.

**Core module abstraction gap.** Unfortunately, as discussed in the previous chapters of this thesis, LP does not come with a native support for expressing *meta-rule*s in a similar way they allow to express clauses. Neither does 2P-Kt.

Furthermore, 2P-Kt does not come with the following utility operations we defined in Section 3.2.5:

- strict structural equality (should not depend on scope);

- semantical equality (should not depend on scope);
- clause weight;
- literal combinations.

**Unify module abstraction gap.** Since we have to extend the `Term` hierarchy for supporting *meta-rule*s and since we need to apply *substitutions* in MetaPatrol, we have to extend `Unificator` too. Furthermore, since `Unificator` features the flag `occurCheckEnabled` we need to extend our design of `Inducer` for being able to plug that flag.

**Parser/DSL modules abstraction gap.** Since we have to extend 2P-Kt for supporting *meta-rule*s, we may need to extend either parser or DSL (or both) for being able to express *meta-rule*s in a declarative fashion.

## 4.3 Filling the Abstraction Gap

After our inspection for identifying the abstraction gap from 2P-Kt, the tasks we need to accomplish are:

- being able to plug a flag `occurCheckEnabled` within our `Inducer` hierarchy;
- extending 2P-Kt for supporting the concept of *meta-rule*, in particular:
  - extending the `:core` module;
  - extending the `:unify` module;
  - extending the `:dls*` and/or the `:parser*` modules;
- implementing the utility operations not provided by 2P-Kt.

### 4.3.1 Occurs check

Since `occurCheckEnabled` can be easily identified as an option, we extend `InducerOptions` with a boolean field `occurCheckEnabled`, thus making it configurable.

### 4.3.2 MetaRules

#### Core module extension

Figure 4.1 illustrates the full initial 2P-Kt `Term` hierarchy.

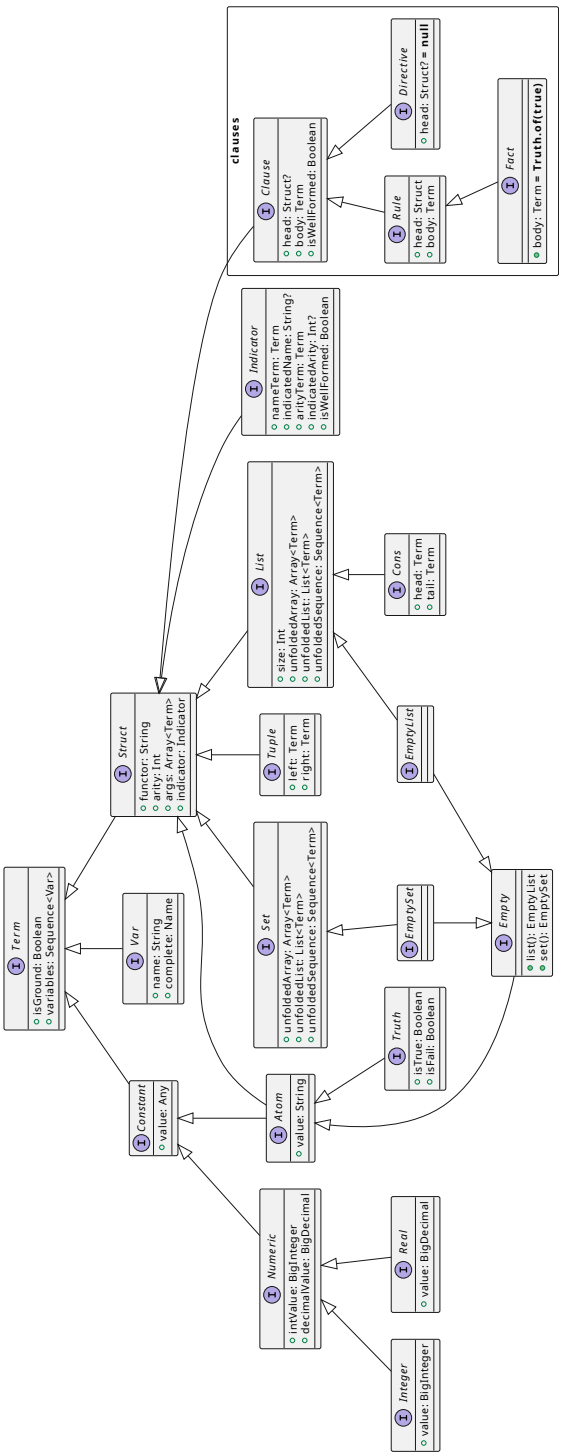


Figure 4.1: 2P-Kt: Initial Term Hierarchy

**Dealing with second-order variables.** The main difference between a *meta-rule* and an ordinary clause are second-order variables. For simplicity, since second-order variables act in a very similar way to ordinary variables, we decide to exploit the already existing `Var` implementation. That choice allows us to express second-order variables in the same way as we express first-order variables.

Anyway, since second-order variables occur as functors of structures, we need to extend the already existing `Struct` for supporting functors of type `Var`. We could either edit the already existing `Struct` functor type, or provide a higher abstraction between `Term` and `Struct` itself, grouping the common features between a `Struct` with a string functor and a `Struct` with a `Var` type functor. Since editing the already existing `Struct` would eventually generate cascading complications over all its implementations, we choose to provide a middle abstraction between `Term` and `Struct`, as well as a new entity representing `Struct` with a `Var` type functor, we call it `Pattern`. Therefore we introduce the following entities:

- `Composed`: a `Struct`-like entity, where we group all the common features between a `Struct` and a `Pattern`. The `Composed` type of the field `functor` is `Any`, for supporting either strings or `Var`, depending on its implementation. We transfer each shared `Struct` field and method within this new entity, properly changing some of their types from `Struct` to `Composed`.
- `Pattern`: a `Composed` entity featuring a `Var` type functor.

The previously existing `Struct` entity is now a `Composed` specialization, overriding its `functor` type as a `String`.

Introducing the new entity `Composed` maximizes *code reuse*, allowing us to exploit the most of the already existing methods.

**Meta-clauses.** 2P-Kt `Clause`s are special `Struct`s with functor *":-"*. A `Clause` has a `head` of type `Struct?` (optional in case of `Directive`s a.k.a. goals) and multiple `Term`s composing its `body`.

Since *meta-rule*s are very similar to clauses and share most of their features, we need to extend `Clause` for supporting second-order variables.

We already decided to represent second-order variables as `Var`s and we added the `Pattern` entity in case of second-order variables acting as functors, adding the `Composed` entity implemented by both `Pattern` and `Struct`. Since the `Clause` field `body` is composed by `Term`s and, since `Pattern`s are `Term`s, we just have to focalize on the field `head` of `Clause`. Therefore, for our extension purpose, we could either edit the already existing `Clause`'s field `head` type for supporting `Composed?` heads, or provide a higher abstraction between `Struct` and `Clause` itself, grouping the common features between an ordinary clause and a meta-clause. Since editing the already existing `Clause` would eventually generate cascading complications

over all its implementations, we choose to provide a middle abstraction, as well as a new entity representing *meta-rule*s. That choice is similar to the one we adopted for `Composed` and `Pattern`, therefore it is a good uniformity practice. We call the new middle entity `Clausal`. Furthermore, since a *meta-rule* has common features with the already existing `Rule`, we make a similar choice adding a further entity between `Clausal` and `Rule`: we call such entity `RuleLike`. We now add the `MetaRule` entity. We provide a description of the introduced entities:

- `Clausal`: a `Clause`-like entity, where we group all of the common features among `Clause`, `Rule`, `Fact`, `Directive` and `MetaRule`. The `Clausal` type of the field `head` is `Composed?`, for supporting either `Struct?` or `Pattern`, depending on its implementation. We transfer each shared `Clause` field and method within this new entity, properly changing some of their types from `Clause` to `Clausal`. Note: we also transfer all methods which can edit a `Clausal` implementation, changing its type to another `Clausal` implementation.

- `RuleLike`: a `Clausal` entity with `head`, where we group all of the common features between `Rule` and `MetaRule`. The `head` field type is `Composed` (not optional).

- `MetaRule`: a `Clausal` entity which `head` and/or any `body` term are `Composed` (a partially substituted *meta-rule* may have a `Struct` type `head` and only a `Pattern` term within its `body`!).

Figure 4.2 illustrates the full extended 2P-Kt `Term` hierarchy.

Note: because of our extension we have to provide some new cast methods to `Term`, and some new methods to `TermVisitor`.

Note: since the legacy `Clause` constructor handled all `Clause` implementations, we need to implement the `Clausal` constructor in a similar way, providing the correct instance of its specializations upon object instantiation.

### Unify module extension

After introducing the `MetaRule` entity, we have to extend the 2P-Kt's `:unify` module following the already discussed table reported at Table 3.1. In particular, 2P-Kt already supports all unifications within such table, except for the last row and column.

`Equation`, within the `:unify` module, is a class representing an equation of logic terms, to be unified. In order to extend the 2P-Kt unification for supporting *meta-rule*s we need to extend `Equation` adding equations for `Pattern`s.

The `Equation` constructor `Equation.of(...)` compares two logic terms and returns one of the the following `Equation`s:

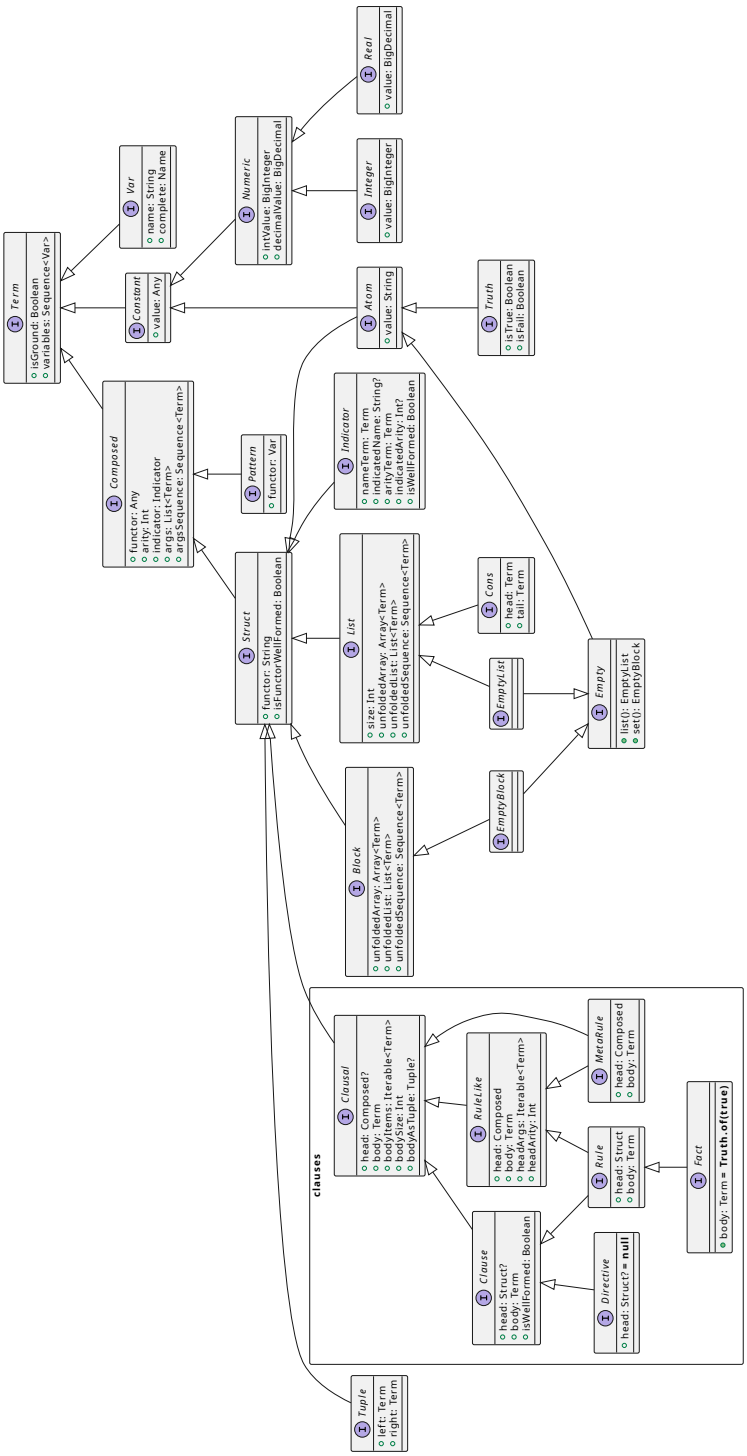


Figure 4.2: 2P-Kt: Extended Term Hierarchy

- `Identity`: an equation of identical `Term`s;
- `Contradiction`: a contradicting equation, trying to equate non equal `Term`s;
- `Assignment`: an equation stating `Var` = `Term`;
- `Comparison`: an equation comparing `Term`s, possibly different.

We extend such constructor adding the following cases:

- if the two terms are a `Var` and a `Pattern`
  - if the `Var` and the functor of the `Pattern` are incompatible returns a `Contradiction` of the two terms
  - returns an `Assignment` of the `Pattern` to the `Var` otherwise
- if one of the two terms is a `Pattern` and the other is either a `Struct` or a `Pattern`
  - if the two terms have different arity returns a `Contradiction`
  - returns a `Comparison` of the two terms otherwise.

The other cases are already covered by the initial implementation of `Equation`.

#### DSL module extension

Since we are working with *meta-rule*s in *Kotlin*, a way for expressing *meta-rule*s without directly using verbose constructors could come in handy, letting us produce more readable code.

We decide to extend the 2P-Kt's `dls:core` module for being able to express *meta-rule*s, within *Kotlin*, via the already existing DSL. For that purpose we just need to edit the already existing methods returning a `Clause` for making them return a `Clausal`, since the latter constructor already manages all `Clausal` implementations.

Listing 4.1 depicts the extended DSL features.

### 4.3.3 Utility operations on terms

Most of the following operations on terms are implemented exploiting the *Kotlin extension functions*. *Kotlin extension functions* allow to extend an already existing type, without editing it, and have a declaration such as the following: `fun Type.extensionMethodName(...):...` where `Type` is the type to be extended. An extension function can then be called like any ordinary method of `Type`.

Listing 4.1: Extended DSL Examples – `MetaRules`

```
import it.unibo.tuprolog.dsl.theory.prolog

...

prolog {
    // impliedBy means :-
    val metaRule = (A(X, Y)).impliedBy(B(X, Z), C(Z, Y))
    val anotherMetaRule = "f"(X, Y) impliedBy A(Y, X)
}

...
```

**Strict structural equality.** We implement the concept of strict structural equality via the method `Struct.strictStructurallyEquals(compared: Struct): Boolean`. The two `Structs` must be `structurallyEquals`.

It works mapping all the variables of the first `Struct` with the same of the `compared` one occurring in the same position. In order for the `Structs` to be strict structurally equals, the number of distinct mapped values must be 1.

**Semantical equality.** We implement the concept of semantical equality via the method `Struct.semanticallyEquals(compared: Struct): Boolean`.

Since two semantically equals terms have the same literals and variable distribution, but the literals have different order, we exploit the already defined `strictStructurallyEquals` method, but we first *sort* all the literals of both `Structs` by `weight` and *alphabetical order*, as follows:

- `Vars`
- `Atoms`
- `Structs`

In that way we convert two potential semantically equals `Structs` as two potential strict structurally equals ones.

**Clause weight.** We extend the concept of weight to the `Term` entity via the method `Term.weight(): Int`.

Such method implementation exploits a `TermVisitor` which explores all the `Terms` composing the `Term` in question, and counts the total occurrences of `Constants`, `Vars` and `EmptyLists`. It does not count the `Truth` in `Fact` bodies.

**Literal combinations.** We implement the concept of combinations via a generic function `<T> combinations(list: List<T): Sequence<List<T>>` which computes all the possible combinations of the element of the provided `list` of `T` type elements.

Such method is recursive and at each iteration yields:

- the one sized combination with the `list` head;
- if the list has more than one element:
  - the combinations of the `list` tail, recursively;
  - the combinations of `list` head with the all the tail combinations from the previous point.

The `combinations` function may then be called passing the list of literals of a clause as parameter.

# Chapter 5

# Validation

For validating the designed and implemented framework we provide a test suite.

Although we tested all the new entities and methods of our 2P-Kt extension, in this chapter, for the purpose of this thesis, we just delve into tests over the library components (i.e. `Generalizer`s, `Validator`s, `Refiner`s). We then provide a set of MIL learning tasks with the relating MetaPatrol inductions. Such inductions serve both as further validation of the library components and as validation of the performances of MetaPatrol.

For more readable tests we exploit the 2P-Kt parser for expressing clauses and theories, while we exploit the extended DSL for expressing *meta-rule*s. Nonetheless, for a clearer demonstration, in this thesis we express the starting values and expected results of each test as Prolog clauses.

## 5.1 Generalizer tests

For better showcasing the differences among the developed `Generalizer`s, we provide a test for each one, generalizing the same clause.

Furthermore, since `Generalizer`s generate variable names like `X_n` where `n` is a natural number (or `Xs_n` for list tails), it would be difficult to reproduce them while testing: using the method `semanticallyEquals` for checking equality allows us to choose any variable name.

**ConstantsToVariables.** The test in Listing 5.1 shows how each constant is substituted by a variable, in particular each occurrence of the same constant is substituted by the same variable.

This `Generalizer` also substitutes empty lists, the $C$ variable in the example.

Listing 5.1: `ConstantsToVariables` Test

```
% to generalize

f([1, 2], [s]) :- g([]), h([1, 4, 5], [1, 4, 5]).

% generalized

f([A, B | C], [D | C]) :- g(C), h([A, E, F | C], [A, E, F | C]).
```

Listing 5.2: `ConstantsToVariablesExceptSingletons` Test

```
% to generalize

f([1, 2], [s]) :- g([]), h([1, 4, 5], [1, 4, 5]).

% generalized

f([A, 2 | B], [s | B]) :- g(B), h([A, C, D | B], [A, C, D | B]).
```

**ConstantsToVariablesExceptSingletons.** The test in Listing 5.2 shows how each constant occurring more than once is substituted by a variable, in particular each occurrence of the same constant is substituted by the same variable.

Note how the constants occurring only once are 2 and $s$, and they are not generalized.

This `Generalizer` also substitutes empty lists (if occurring more than once), the $B$ variable in the example.

**ConstantsToVariablesExceptEmptyList.** The test in Listing 5.3 shows how each constant is substituted by a variable, in particular each occurrence of the same constant is substituted by the same variable.

This `Generalizer` does not substitute empty lists: lists do not have the final pipe as the two `Generalizers` at the previous tests, the empty list within the first body literal is not generalized.

Listing 5.3: `ConstantsToVariablesExceptEmptyList` Test

```
% to generalize

f([1, 2], [s]) :- g([]), h([1, 4, 5], [1, 4, 5]).

% generalized

f([A, B], [C]) :- g([]), h([A, D, E], [A, D, E]).
```

Listing 5.4: `ConstantsToVariablesExceptSingletonsAndEmptyList` Test

```
% to generalize

f([1, 2], [s]) :- g([]), h([1, 4, 5], [1, 4, 5]).

% generalized

f([A, 2], [s]) :- g([]), h([A, B, C], [A, B, C]).
```

Listing 5.5: `ListSpecificGeneralizerEmptyListAsVar` Test

```
% to generalize

f([1, 2], [s]) :- g([]), h([1, 4, 5], [1, 4, 5]).

% generalized

f([A | As], [B | Bs]) :- g(Bs), h([A | Cs], [A | Cs]).
```

**ConstantsToVariablesExceptSingletonsAndEmptyList.** The test in Listing 5.4 shows how each constant occurring more than once is substituted by a variable, in particular each occurrence of the same constant is substituted by the same variable.

Note how the constants occurring only once are 2 and $s$, and they are not generalized.

This `Generalizer` does not substitute empty lists: lists do not have the final pipe as some of the `Generalizers` at the previous tests, the empty list within the first body literal is not generalized.

**ListSpecificGeneralizerEmptyListAsVar.** The test in Listing 5.5 shows how each constant is substituted by a variable, in particular each occurrence of the same constant is substituted by the same variable.

Specifically, this `Generalizer` applies a generalization on lists expressing them in the form $[A \mid As]$, carefully reusing the head and tail variables where necessary. It furtherly generalizes as single variables the lists of the previous form, whose variables of head and tail do not occur in any other disposition.

This `Generalizer` also substitutes empty lists, the $Bs$ variable in the example.

In this particular example the constant 1 occurs as head of multiple lists, and is substited by the variable $A$. The empty list alone and the list $[4, 5]$ occurs as tail of multiple lists, and they are substituted respectively by the variables $[Bs]$ and $[Cs]$.

Listing 5.6: `ListSpecificGeneralizerExceptSingletons` Test

```
% to generalize

f([1, 2], [s]) :- g([]), h([1, 4, 5], [1, 4, 5]).

% generalized

f([A | As], [s | Bs]) :- g(Bs), h([A | Cs], [A | Cs]).
```

Listing 5.7: `ListSpecificGeneralizerExceptEmptyList` Test

```
% to generalize

f([1, 2], [s]) :- g([]), h([1, 4, 5], [1, 4, 5]).

% generalized

f([A | As], B) :- g([]), h([A | Cs], [A | Cs]).
```

**ListSpecificGeneralizerExceptSingletons.** The test in Listing 5.6 shows how each constant occurring more than once is substituted by a variable, in particular each occurrence of the same constant is substituted by the same variable.

Specifically, this `Generalizer` applies the same further generalizations on lists as the one in the previous paragraph.

This `Generalizer` also substitutes empty lists (if occurring more than once), the $Bs$ variable in the example.

In this particular example the constant 1 occurs as head of multiple lists, and is substited by the variable $A$. The empty list alone and the list $[4, 5]$ occur as tails of multiple lists, and they are substituted respectively by the variables $[Bs]$ and $[Cs]$.

Note how the constants occurring only once are 2 and $s$. Since 2 occurs in a tail and we apply the further generalizations on lists as the `Generalizer` in the previous paragraph, the tail $[2|[]]$ is generalized as $As$. The constant $s$, being a head of a list (and for the head of a list not being a list itself), is not generalized.

**ListSpecificGeneralizerExceptEmptyList.** The test in Listing 5.7 shows how each constant is substituted by a variable, in particular each occurrence of the same constant is substituted by the same variable.

Specifically, this `Generalizer` applies the same further generalizations on lists as the one in the previous paragraph.

This `Generalizer` does not substitute empty lists: the empty list within the first body literal is not generalized.

In this particular example the constant 1 occurs as head of multiple lists, and

Listing 5.8: `ListSpecificGeneralizerExceptSingletonsAndEmptyList` Test

```
% to generalize

f([1, 2], [s]) :- g([]), h([1, 4, 5], [1, 4, 5]).

% generalized

f([A | As], [s]) :- g([]), h([A | Cs], [A | Cs]).
```

is substited by the variable $A$. The list $[4, 5]$ occurs as tails of multiple lists, and it is substituted by the variable $[Cs]$.

Note how the list $[s]$ in the head of clause for its head and tail variables do not occur in different forms from $[B \mid []]$, therefore it is generalized as a single variable $B$.

**ListSpecificGeneralizerExceptSingletonsAndEmptyList.** The test in Listing 5.8 shows how each constant occurring more than once is substituted by a variable, in particular each occurrence of the same constant is substituted by the same variable.

Specifically, this `Generalizer` applies the same further generalizations on lists as the one in the previous paragraph.

This `Generalizer` does not substitute empty lists: the empty list within the first body literal is not generalized.

In this particular example the constant 1 occurs as head of multiple lists, and is substited by the variable $A$. The list $[4, 5]$ occurs as tails of multiple lists, and it is substituted by the variable $[Cs]$.

Note how the constants occurring only once are 2 and $s$. Since 2 occurs in a tail and we apply the further generalizations on lists as the `Generalizers` in the previous paragraphs, the tail $[2|[]]$ is generalized as $As$. The constant $s$, being a head of a list (and for the head of a list not being a list itself), is not generalized.

**Fact sensitive generalization example.** The test in Listing 5.9 shows the different possible generalization if the body of a clause is `true`. All the `Generalizers` we implemented are fact sensitive and, in the case of a possible fact, the return the generalization as fact first.

**More on list generalization.** Since the previous examples were designed for showcasing the differences among `Generalizers`, we provide a best list generalization case in Listing 5.10.

Note how the list $[4, 5]$, instead of being generalized as $[B \mid Bs]$, is generalized as a single variable $B$ in both its occurrences. That generalization occurs because,

Listing 5.9: A Fact Sensitive Generalization Test

```
% to generalize

f(true).

% generalized

f(A).
f(A) :- A.
```

Listing 5.10: A Strong List Generalization Test

```
% to generalize

f([1, 2, 3], [4, 5], [1, 2, 3, 4, 5]) :- f([2, 3], [4, 5], [2, 3, 4, 5]).

% generalized

f([A | As], B, [A | Cs]) :- f(As, B, Cs).
```

after the previous generalization step where $[4, 5]$ became $[B \mid Bs]$, the variables $B$ and $Bs$ do not occur outside the form $[B \mid Bs]$. Although $[4, 5]$ is also a sub-list of the third and last lists, such lists were already generalized at the previous step as $[C \mid Cs]$ and $Cs$, since $[2, 3, 4, 5]$ is the tail of $[1, 2, 3, 4, 5]$.

## 5.2 Validator tests

The following `OccamisticValidator` tests all are based on the predicate *append/3* and are executed providing a set of positive and negative examples.

**Pruning redundant clauses.** The third clause within the initial `theory` in Listing 5.11 is redundant since it covers a specific *append/3* case where the second list is empty. `OccamisticValidator` detects it and prunes it.

**Pruning infinite loop clauses.** The third clause within the initial `theory` in Listing 5.12 causes an infinite loop. `OccamisticValidator` detects it and prunes it.

**Invalid theories.** In Listing 5.13 we provide to the validator the same set of positive and negative examples for the predicate *myappend/3* as the previous tests, and an empty background knowledge. However, we call it to validate a wrong

Listing 5.11: `OccamisticValidator` Pruning a Redundant Clause

```
% to validate

myappend([], A, A).
myappend([A | As], B, [A | Cs]) :- myappend(As, B, Cs).
myappend([A | As], [], [A | As]) :- myappend(As, [], As).

% validated

myappend([], A, A).
myappend([A | As], B, [A | Cs]) :- myappend(As, B, Cs).
```

Listing 5.12: `OccamisticValidator` Pruning an Infinite Loop Clause

```
% to validate

myappend([], A, A).
myappend([A|As], B, [A|Cs]) :- myappend(As, B, Cs).
myappend(A, B, C) :- myappend(D, E, F).

% validated

myappend([], A, A).
myappend([A | As], B, [A | Cs]) :- myappend(As, B, Cs).
```

`theory`, therefore it proves such theory to be wrong and returns a result of type `InvalidTheory`.

## 5.3 Refiner tests

We already discussed about the multiple utility of `ListSpecificRefiner`: as a `Refiner`, but also in list generalization. Since in this chapter we already provided a test (in Listing 5.10) in which a generalizer exploits `ListSpecificRefiner`, in this section we only provide test examples for the other `Refiner`s, i.e., refiners for predicate invention.

Listing 5.13: `OccamisticValidator` Identifying an Invalid Theory

```
% to validate

myappend([], [], X).

% result: invalid theory
```

Listing 5.14: `PredicateInventorMetagolLike` Test

```
% to refine

f(A, B) :- g(A, B), g(A, B).
f(A, B) :- g(A, B), h(A, B).
f(A, B) :- h(A, B), h(A, B).

% refined

f(A, B) :- invented_0(A, B), invented_0(A, B).
invented_0(A, B) :- g(A, B).
invented_0(A, B) :- h(A, B).
```

Listing 5.15: `PredicateInventorGroupNonCommon` Test

```
% to refine

f(A, B) :- g(A), h(A, B), l(B).
f(A, B) :- i(A), h(A, B), l(B).

% refined

f(A, B) :- h(A, B), l(B), invented_0(A).
invented_0(A) :- g(A).
invented_0(A) :- i(A).
```

**Metagol-like predicate invention.** Listing 5.14 shows a test in which the `theory` to refine is composed by multiple predicates featuring the same head and very similar bodies. For instance, each body is composed by a combination of the same two literals. `PredicateInventorMetagolLike` detects it and invents a fresh predicate for each of the two body literals (i.e., `invented_0`), grouping the three starting predicates in a unique one.

**Grouping non common literals.** Listing 5.15 shows a test in which the `theory` to refine is composed by some clauses (two in this case) which have multiple body literals in common. For instance: $h(A, B), l(B)$.

`PredicateInventorGroupNonCommon` detects it and invents a fresh predicate in which it groups the *non* common literals of such clauses: $g(A)$ and $i(A)$. The initial two clauses are grouped in an unique one.

**Grouping common literals.** We refine the same `theory` from the previous example in Listing 5.16, but with another approach. `PredicateInventorGroupNonCommon` detects the common literals and invents a fresh predicate in which it groups the common literals of the clauses: $h(A, B), l(B)$.

Listing 5.16: `PredicateInventorGroupCommon` Test

```
% to refine

f(A, B) :- g(A), h(A, B), l(B).
f(A, B) :- i(A), h(A, B), l(B).

% refined

f(A, B) :- g(A), invented_0(A, B).
f(A, B) :- i(A), invented_0(A, B).
invented_0(A, B) :- h(A, B), l(B).
```

Listing 5.17: Reuse of Already Existing Predicates Test

```
% to refine

f(A, B) :- g(A, B), h(A).
i(A, B) :- g(A, B), h(A).

% refined

f(A, B) :- i(A, B).
i(A, B) :- g(A, B), h(A).
```

**Already existing predicates.** The clauses of the `theory` to refine in Listing 5.17 have the same body. The developed predicate inventors are able to detect it and insert a call of one of the two predicates within the other, removing the code duplication.

Note: for our predicate invention algorithm implementation, such feature also works within invented predicates.

**Multiple inventions.** `PredicateInventorGroupNonCommon` and `PredicateInventorGroupCommon` are able to invent multiple predicates, on a suitable theory. In Listing 5.18 the `theory` to refine is composed by two clauses with $h(A, B)$ in common and two clauses with $n(A, B)$ in common. Our predicate inventors detect both of them and invent two different predicates.

## 5.4 MetaPatrol

### 5.4.1 Learning task: *grandparent/2*

One of the very first rules everyone learn while studying LP is usually *grandparent/2*, where the first argument denotes the grandparent while the second denotes the grandchild. We subject MetaPatrol to the learning task of inducing such

Listing 5.18: Multiple Predicate Inventions Test

```
% to refine

f(A, B) :- g(A), h(A, B).
f(A, B) :- i(A), h(A, B).
l(A, B) :- m(A), n(A, B).
l(A, B) :- o(A), n(A, B).

% refined

f(A, B) :- h(A, B), invented_0(A).
invented_0(A) :- g(A).
invented_0(A) :- i(A).
l(A, B) :- invented_1(A), n(A, B).
invented_1(A) :- m(A).
invented_1(A) :- o(A).
```

rule.

We provide a background knowledge, as well as some positive and negative examples, based on such background knowledge (Listing 5.19).

As bias we choose the chain metarule, a simple generalizer not specialized on lists (`ConstantsToVariables`), the `OccamisticValidator`, and the refiner `PredicateInventorMetagolLike`. The chain metarule is:

$$\mathbf{X}(A, B) \text{ :- } \mathbf{Y}(A, C), \mathbf{Z}(C, B).$$

The induced results are reported in Listing 5.20. MetaPatrol induces two different correct theories:

- as first the non refined theory;
- as second the refined theory by `PredicateInventorMetagolLike`, inventing the predicate *parent/2* (with the less specific functor *invented_0*, of course).

### 5.4.2 Learning task: *tail/2*

*tail/2* is a simple rule on lists. The first argument denotes a whole list, the second argument denotes the tail of such list. We subject MetaPatrol to the learning task of inducing such rule.

We do not provide a background knowledge, we only provide some positive and negative examples (Listing 5.21).

As bias we choose the very simple metarule

$$\mathbf{X}(A, B).$$

two list specific generalizers:

Listing 5.19: Grandparent – BK, Positive/Negative Examples

```
% backgroundKnowledge

father(albert, jack).
father(jack, mary).
father(john, charlotte).
mother(charlotte, mark).
father(mark, delia).
mother(charlotte, april).
mother(april, george).

% positive examples

grandparent(albert, mary).
grandparent(john, mark).
grandparent(charlotte, delia).
grandparent(charlotte, george).

% negative examples

grandparent(albert, jack).
grandparent(albert, albert).
grandparent(mary, jack).
grandparent(mary, albert).
```

Listing 5.20: Grandparent – MetaPatrol Inductions

```
grandparent(X0, X1) :- mother(X0, X2), mother(X2, X1)
grandparent(X0, X1) :- mother(X0, X2), father(X2, X1)
grandparent(X0, X1) :- father(X0, X2), mother(X2, X1)
grandparent(X0, X1) :- father(X0, X2), father(X2, X1)
% coverage: 100%
% ---

grandparent(X0, X1) :- invented_0(X0, X2), invented_0(X2, X1)
invented_0(X0, X2) :- mother(X0, X2)
invented_0(X0, X2) :- father(X0, X2)
% coverage: 100%
% ---
```

Listing 5.21: Tail – BK, Positive/Negative Examples

```
% empty background knowledge

% positive examples

tail([1, 2, 3, 4], [2, 3, 4]).
tail([3, 5, 7], [5, 7]).

% negative examples

tail([1, 2], [1]).
tail([3], 3).
```

Listing 5.22: Tail – MetaPatrol Inductions

```
tail([X0 | Xs0], Xs0) :- true
% coverage: 100%
```

Listing 5.23: Append – BK, Positive/Negative Examples

```
% empty background knowledge

% positive examples

myappend([1, 2, 3], [], [1, 2, 3]).
myappend([], [1, 2, 3], [1, 2, 3]).
myappend([1], [2, 3], [1, 2, 3]).
myappend([1, 2], [3, 4], [1, 2, 3, 4]).

% negative examples

myappend([], [], [1]).
myappend([], [1, 2, 3], []).
myappend([1, 2], [], []).
myappend([1, 2], [3, 4], [1, 2]).
myappend([1, 2], [3, 4], [3, 4]).
```

- `ListSpecificGeneralizerEmptyListAsVar`
- `ListSpecificGeneralizerExceptEmptyList`

the `OccamisticValidator`, and no refiners.

The induced result is reported in Listing 5.22. MetaPatrol manages to induce *tail/2* without needing a background knowledge and without needing a very constraining *meta-rule*.

### 5.4.3 Learning task: *append/3*

*append/3* is a more complex rule on lists: it is recursive. The first two arguments denote two lists, the third argument denotes the second list added to the end of the first one. We subject MetaPatrol to the learning task of inducing such rule.

We do not provide a background knowledge, we only provide some positive and negative examples (Listing 5.23).

As bias we choose the two metarules:

$$\mathbf{X}([], B, C).$$

$$\mathbf{X}([A \mid B], C, [D \mid E]) \text{ :- } \mathbf{X}(B, C, E).$$

two list specific generalizers:

Listing 5.24: Append – MetaPatrol Inductions

```
myappend([], X0, X0) :- true
myappend([X0 | Xs1], X3, [X0 | Xs0]) :- myappend(Xs1, X3, Xs0)
% coverage: 100%
```

Listing 5.25: Append – MetaPatrol Optimal and Sub-Optimal Inductions

```
myappend([], X0, X0) :- true
myappend([X0 | Xs1], X2, [X0 | Xs0]) :- myappend(Xs1, X2, Xs0)
% coverage: 100%
% ---
myappend([], X0, X0) :- true
myappend([X0 | Xs0], [], [X0 | Xs0]) :- myappend(Xs0, [], Xs0)
myappend([X0], [X1, X2], [X0, X1, X2]) :- myappend([], [X1, X2], [X1, X2])
myappend([X0, X1], [X2, X3], [X0, X1, X2, X3]) :- myappend([X1], [X2, X3], [X1, X2
    , X3])
% coverage: 100%
% ---
myappend([], X0, X0) :- true
myappend([X0], Xs0, [X0 | Xs0]) :- myappend([], Xs0, Xs0)
myappend([X0 | Xs0], [], [X0 | Xs0]) :- myappend(Xs0, [], Xs0)
myappend([X0, X1], [X2, X3], [X0, X1, X2, X3]) :- myappend([X1], [X2, X3], [X1, X2
    , X3])
% coverage: 100%
% ---
```

- `ListSpecificGeneralizerEmptyListAsVar`
- `ListSpecificGeneralizerExceptEmptyList`

the `OccamisticValidator`, and no refiners.

The induced result is reported in Listing 5.24. MetaPatrol manages to induce *append/3* without needing a background knowledge with returning a very readable rule.

If adding some generalizers within the bias, the algorithm explores and finds some other sub-optimal solutions, which prove the examples. Listing 5.25 shows the induced rules if we also plug the generalizer `ConstantsToVariablesExceptEmptyList`.

### 5.4.4 Learning task: *reverse/2*

*reverse/2* is an even more complex rule on lists: it is recursive and it requires *append/3*. The first argument of *reverse/2* denotes a list, the second argument denotes the reversed list. We subject MetaPatrol to the learning task of inducing such rule.

We provide *append/3* within background knowledge, and we provide some positive and negative examples (Listing 5.26).

Listing 5.26: Reverse – BK, Positive/Negative Examples

```
% backgroundKnowledge

myappend([], F, F).
myappend([F | Fs], G, [F | Hs]) :- myappend(Fs, G, Hs).

% positive examples

myreverse([], []).
myreverse([1, 2, 3, 4, 5], [5, 4, 3, 2, 1]).

% negative examples

myreverse([], [1]).
myreverse([1, 2], []).
myreverse([1, 2], [1, 2]).
myreverse([1, 2, 3], [3, 1, 2]).
```

Listing 5.27: Reverse – MetaPatrol Inductions

```
myreverse([], []) :- true
myreverse([X0 | Xs0], X4) :- myreverse(Xs0, D), myappend(D, [X0], X4)
% coverage: 100%
```

As bias we choose the two metarules:

$$\mathbf{X}([],[]).$$

$$\mathbf{X}([A \mid B],C) \text{ :- } \mathbf{X}(B,D),\mathbf{Y}(D,[A],C).$$

the `ListSpecificGeneralizerExceptEmptyList`, the `OccamisticValidator`, and no refiners.

The induced result is reported in Listing 5.27. MetaPatrol manages to induce *reverse/* correctly exploiting *append/3* from background knowledge, and returning a very readable rule.

### 5.4.5 Learning task: *map_plus_two/2*

*map_plus_two/2* is an even more complex rule on lists: it is recursive and it requires a predicate which computes the sum of an intenger plus 2, we call such predicate *plus_two/2*. The first argument of *map_plus_two/2* denotes a list, the second argument denotes the mapped list where each element $e$ of the first list is mapped as $e + 2$. We subject MetaPatrol to the learning task of inducing such rule, without directly providing *plus_two/2*.

We provide *succ/2* within background knowledge (which computes the successor of an integer), and we provide some positive and negative examples for both *map_plus_two/2* and *plus_two/2* (Listing 5.28).

Listing 5.28: Map Plus Two – BK, Positive/Negative Examples

```
% backgroundKnowledge

mysucc(A, B) :- B is A + 1.

% positive examples

plus_two(1, 3).
plus_two(2, 4).
map_plus_two([], []).
map_plus_two([1, 2, 3, 4, 5], [3, 4, 5, 6, 7]).

% negative examples

plus_two(1, 1).
plus_two(3, 1).
map_plus_two([1, 2], [1, 2]).
map_plus_two([3, 5], [1, 3]).
```

Listing 5.29: Map Plus Two – MetaPatrol Inductions

```
map_plus_two([], []) :- true
plus_two(X0, X1) :- mysucc(X0, C), mysucc(C, X1)
map_plus_two([X0 | Xs0], [X2 | Xs1]) :- plus_two(X0, X2), map_plus_two(Xs0, Xs1)
% coverage: 100%
```

As bias we choose the three metarules:

$$\mathbf{X}([],[]).$$

$$\mathbf{X}(A,B) \text{ :- } \mathbf{Y}(A,C),\mathbf{Z}(C,B).$$

$$\mathbf{X}([A \mid B],[C \mid D]) \text{ :- } \mathbf{Y}(A,C),\mathbf{X}(B,D).$$

the `ListSpecificGeneralizerExceptEmptyList`, the `OccamisticValidator`, and no refiners.

The induced result is reported in Listing 5.29. MetaPatrol manages to induce *plus_two/2* exploiting *succ/2* from background knowledge, and it also manages to reuse such induced *plus_two/2* for correctly inducing *map_plus_two/2*, and returning a very readable rule.

### 5.4.6 MetaPatrol use case

Listing 5.30 illustrates an example of how to use MetaPatrol, in particular we consider the *grandparent/2* learning task. We provide a `backgroundKnowledge`, `positiveExamples` and `negativeExamples` exploiting the 2P-Kt parser. We create an instance of `MetaPatrolBias`, `bias` containing:

- `metaRules`: expressed with the extended DSL (since it is a `Clausal` we need to explicitly cast all *meta-rule*s);
- `generalizers`: containing the generalizers we desire to plug;
- `validators`: containing the validators we desire to plug;
- `refiners`: containing the refiners we desire to plug.

We create an instance of `MetaPatrolOptions`, `options` with a value for max depth.

Within the method `metaPatrolInduction()` we instanciate the `metaPatrolInducer`, providing the aforementioned `backgroundKnowledge`, `bias`, the default `Unificator` provided by 2P-Kt, and a `Solver.prolog` provided by 2P-Kt. We then call the method `induce` of such `metaPatrolInducer`, providing the aforementioned `positiveExamples`, `negativeExamples`, and `options`. In this particular example we print each induced theory, along with their coverage.

Listing 5.30: MetaPatrol – use case

```
class MetaPatrolUseCase {

    private val parser = ClausesParser.withDefaultOperators

    private val backgroundKnowledge = parser.parseTheory(
        """
        father(albert, jack).
        father(jack, mary).
        father(john, charlotte).
        mother(charlotte, mark).
        father(mark, delia).
        mother(charlotte, april).
        mother(april, george).
        """
    )

    private val positiveExamples = parser.parseClauses(
        """
        grandparent(albert, mary).
        grandparent(john, mark).
        grandparent(charlotte, delia).
        grandparent(charlotte, george).
        """
    )

    private val negativeExamples = parser.parseClauses(
        """
        grandparent(albert, jack).
        grandparent(albert, albert).
        grandparent(mary, jack).
        grandparent(mary, albert).
        """
    )

    private val bias: MetaPatrolBias = prolog {
        MetaPatrolBias(
            metaRules = setOf(
                X(A, B).impliedBy(Y(A, C), Z(C, B)) as MetaRule
            ),
            generalizers = ktListOf(ConstantsToVariables),
            validators = ktListOf(OccamisticValidator),
            refiners = ktListOf(PredicateInventorMetagolLike)
        )
    }

    private val maxDepth = 10

    private val options = MetaPatrolOptions(depth = maxDepth)

    fun main() {
        val metaPatrolInducer = MetaPatrol(backgroundKnowledge, bias, Unificator.default, Solver.prolog)

        metaPatrolInducer.induce(positiveExamples, negativeExamples, options).forEach {
            for (clause in it.theory) {
                println(clause.format(TermFormatter.prettyExpressions()))
            }
            println("coverage: " + (it.coverage!! * 100).toInt() + "%")
            println("---")
        }
    }
}
```

# Chapter 6

# Conclusions

In the last few years, the research on ILP is experiencing a remarkable development. From the first algorithms inducing simple rules, researchers managed to design more complex algorithms able to produce more articulated inductions. New frontiers have opened up: predicate invention, higher order, *meta-rule*s, conflict-driven approaches etc. Unfortunately, ILP has not achieved results comparable with those achieved by ML yet, nor accessible libraries for supporting the implementation of ILP algorithms exist.

This thesis, via the multi-platform framework 2P-Kt, proposes to enrich ILP (in particular MIL) providing a library of pluggable tools able to support the design and implementation of ILP algorithms. We designed and implemented multiple tools supporting each phase of a ILP algorithm: we made available numerous new generalization techniques, theory validators and different theory refiners, extending the field of predicate invention. Those tools, thanks to 2P-Kt, allow us to reason at a higher level of abstraction than basic LP, opening new possibilities. MetaPatrol itself poses as a showcase of what it is possible to achieve with the library, fully exploiting its pluggability, therefore being highly configurable.

Since the library allows a way for clearly expressing *meta-rule*s, now we can provide to MIL algorithms more expressive *meta-rule*s containing lists and, thanks to our generalization mechanisms, MIL algorithms can induce more complex rules which operate with lists. MetaPatrol again showcases these new possibilities, being able to induce rules like *tail/2*, *append/3*, *reverse/2*, and *map_plus_two/2* with just a bunch of examples, expressing the induced rules in human-like fashion, even inducing multiple different rules within the same learning task.

## 6.1 Future works

Since the library we developed is fully extendible we can imagine new generalization, validation and refinement mechanism to grow the library utility range. Furthermore, there is room for improvement. Here some interesting ideas concerning the library:

- extending the 2P-Kt parser for supporting *meta-rule*s;
- a `Generalizer` generalizing whole structures as single variables, in a similar way to what we can do with lists;
- a `Generalizer` generalizing list of lists, since by now list specific generalizers treat any possible term within a list in the same way;
- improving the detection of infinite loop rules in `OccamisticValidator`, instead of exploiting timeouts. That could be probably achieved by identifying recursive rules with head semantically equals to body and/or recursive rules not accompanied by a base case;
- a `Validator` which, after detecting the wrong rules of a theory, generates some conflict-driven constraints.

MetaPatrol can be improved too, since it is still a prototype. Here some interesting ideas for extending MetaPatrol:

- providing a user interface;
- implementing higher order mechanisms within the algorithm induction engine;
- designing a way for combining different *meta-rule*s creating larger ones, in order to induce more complex rules with smaller biases.

# Bibliography


[1] Giovanni Ciatto. On the role of computational logic in data science: representing, learning, reasoning, and explaining knowledge. `http://amsdottorato.unibo.it/id/eprint/10192`, 2022.

[2] Giovanni Ciatto, Roberta Calegari, and Andrea Omicini. 2P-Kt: A logic-based ecosystem for symbolic AI. *SoftwareX*, 16:100817:1–7, December 2021.

[3] Giovanni Ciatto, Roberta Calegari, and Andrea Omicini. Lazy stream manipulation in prolog via backtracking: The case of 2p-kt. In Wolfgang Faber, Gerhard Friedrich, Martin Gebser, and Michael Morak, editors, *Logics in Artificial Intelligence - 17th European Conference, JELIA 2021, Virtual Event, May 17-20, 2021, Proceedings*, volume 12678 of *Lecture Notes in Computer Science*, pages 407–420. Springer, 2021.

[4] Domenico Corapi, Alessandra Russo, and Emil Lupu. Inductive logic programming in answer set programming. In Stephen H. Muggleton, Alireza Tamaddoni-Nezhad, and Francesca A. Lisi, editors, *Inductive Logic Programming - 21st International Conference, ILP 2011, Windsor Great Park, UK, July 31 - August 3, 2011, Revised Selected Papers*, volume 7207 of *Lecture Notes in Computer Science*, pages 91–97. Springer, 2011.

[5] Andrew Cropper and Sebastijan Dumancic. Inductive logic programming at 30: a new introduction. *CoRR*, abs/2008.07912, 2020.

[6] Andrew Cropper, Sebastijan Dumancic, and Stephen H. Muggleton. Turning 30: New ideas in inductive logic programming. *CoRR*, abs/2002.11002, 2020.

[7] Andrew Cropper and Rolf Morel. Learning programs by learning from failures. *CoRR*, abs/2005.02259, 2020.

[8] Andrew Cropper and Stephen H. Muggleton. Metagol system. https://github.com/metagol/metagol, 2016.

[9] Andrew Cropper, Alireza Tamaddoni-Nezhad, and Stephen Muggleton. Meta-interpretive learning of data transformation programs, 02 2015.

[10] Enrico Denti, Andrea Omicini, and Alessandro Ricci. tu prolog: A light-weight prolog for internet applications and infrastructures. In I. V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages, Third International Symposium, PADL 2001, Las Vegas, Nevada, USA, March 11-12, 2001, Proceedings*, volume 1990 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2001.

[11] Ana Luísa Duboc, Aline Paes, and Gerson Zaverucha. Using the bottom clause and mode declarations on FOL theory revision from examples. In Filip Zelezný and Nada Lavrac, editors, *Inductive Logic Programming, 18th International Conference, ILP 2008, Prague, Czech Republic, September 10-12, 2008, Proceedings*, volume 5194 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2008.

[12] Werner Emde, Christopher Habel, and Claus-Rainer Rollinger. The discovery of the equator or concept driven learning. In Alan Bundy, editor, *Proceedings of the 8th International Joint Conference on Artificial Intelligence. Karlsruhe, FRG, August 1983*, pages 455–458. William Kaufmann, 1983.

[13] Mark Law. Conflict-driven inductive logic programming. *CoRR*, abs/2101.00058, 2021.

[14] Mark Law, Alessandra Russo, and Krysia Broda. The ILASP system for inductive learning of answer set programs. *CoRR*, abs/2005.00904, 2020.

[15] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, apr 1982.

[16] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *New Generation Computing*. Academic Press, 1990.

[17] Stephen H. Muggleton. Inductive logic programming. *New Gener. Comput.*, 8(4):295–318, 1991.

[18] Stephen H. Muggleton, Dianhuan Lin, Jianzhong Chen, and Alireza Tamaddoni-Nezhad. Metabayes: Bayesian meta-interpretative learning using higher-order stochastic refinement. In Gerson Zaverucha, Vítor Santos Costa, and Aline Paes, editors, *Inductive Logic Programming - 23rd International Conference, ILP 2013, Rio de Janeiro, Brazil, August 28-30, 2013, Revised Selected Papers*, volume 8812 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2013.

[19] J. Ross Quinlan. Learning logical definitions from relations. *Mach. Learn.*, 5:239–266, 1990.

[20] Ashwin Srinivasan. The aleph manual. `https://www.cs.ox.ac.uk/activities/programinduction/Aleph/aleph.html`, 2007.

[21] Irene Stahl. Predicate invention in ILP - an overview. In Pavel Brazdil, editor, *Machine Learning: ECML-93, European Conference on Machine Learning, Vienna, Austria, April 5-7, 1993, Proceedings*, volume 667 of *Lecture Notes in Computer Science*, pages 313–322. Springer, 1993.

[22] ILASP Team. Ilasp - learning logically - mtarules. `https://doc.ilasp.com/specification/metarules.html`, 2022.