Inductive Logic Programming as Satisfiability Modulo Theories

Rolf Morel

St. John's College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2023

Abstract

At the intersection of machine learning, program synthesis and automated reasoning lies the field of Inductive Logic Programming (ILP). The aim of ILP is to automatically learn relational programs from input/output examples, typically through logic-based techniques. Inspired by Karl Popper's falsification perspective on science, this dissertation sets out a new approach to ILP: Learning From Failures (LFF).

In science, starting from a huge set of *a priori* viable hypotheses, we select a hypothesis to test. This hypothesis typically gets falsified due to failing in some specific way. By examining the failure we learn that an entire space of related hypotheses is now ruled out. Having thus reduced our set of viable hypotheses, we subsequently select from just those that remain.

LFF applies this methodology to program induction, codifying it as a three-stage loop. The *generate* stage maintains a formula whose satisfying assignments correspond to the set of viable hypotheses. The *test* stage takes a satisfying assignment, interprets it as a logic program and tests it against training examples – imperfect fit is considered a failure. The *constrain* stage turns a failure into constraints to add to the generate stage's formula, thereby eliminating a class of hypotheses which will fail for the same reason.

The thesis of this dissertation is three-fold. The **first claim** is that *LFF frames the ILP problem as one of Satisfiability Modulo Theories (SMT)*. With the search for viable hypotheses handed-off to a SAT-solver, the test stage can be regarded as a theory solver collaborating with the SAT-solver, effectively making ILP's notion of background *knowledge* into a (Horn) background *theory*.

The **second claim** is that *LFF's three-stage loop is an effective basis for falsification-based program induction*. Chapter 4 develops the above correspondence into a feature-rich and flexible three-stage ILP system. Experimental evidence is provided for this system going beyond the state-of-the-art in ILP, e.g., by supporting large hypothesis spaces and large domains.

The **third claim** is that *the LFF-as-SMT-perspective helps apply satisfiability solving techniques to ILP*, in particular to reduce hypothesis space exploration. In Chapter 5, we show that SMT-based techniques for explaining conflicts have a natural analog in terms of explaining which parts of a hypothesis are responsible for its failure. In Chapter 6, we incorporate other theory solvers alongside the test stage to reason about the (satisfiability of) *over-approximating* properties of hypotheses. We show that both of these techniques can significantly reduce the number of iterations of the three-stage loop.

Acknowledgements

First, I would like to thank Jeremy Gibbons and Luke Ong for their support and encouragement and for allowing me to pursue topics I would like to work on. Sincere thanks go to Andrew Cropper for introducing me to Inductive Logic Programming and suggesting that we work on a new approach. Our collaboration on the foundational framework of Learning From Failures was especially fruitful. Without his overly enthusiastic supervision of extensions of this approach, this particular dissertation would not have been possible.

I would like to thank Katsumi Inoue for hosting me at his lab in Tokyo – this three-month period turned out to be crucial for formulating the thesis of this dissertation. Hereby also my thanks to Sebastijan Dumančić and Georg Gottlob for serving as examiners and for their helpful comments.

Hereby too, I would like to gratefully acknowledge the financial support from scholarships by the Engineering and Physical Sciences Research Council, by a collaboration of Oxford and DeepMind, and by Google. My thanks also go to Google's Albert Cohen and Denys Shabalin for giving me the opportunity to work on exciting and practical applications of program synthesis.

Finally I would like give my sincere thanks for the support that I have received from St John's College, Oxford – by its staff, students, and fellows – throughout the DPhil. Particular thanks go to Stefan Kiefer, especially during the period that I was teaching the St John's undergraduates.

For Lei

Contents

Chapter 1

Introduction

This dissertation presents the Learning From Failures approach to Inductive Logic Programming. By way of introduction, we motivate our approach by arguing for solver-oriented ILP and highlighting current shortcomings. Next we view program induction from three different logical perspectives, which then come together to introduce the Learning From Failures framework. Following on, we see our three-stage loop approach as the archetype of a recently emerging class of *conflict-driven* ILP, whereupon we note a correspondence with Satisfiability Modulo Theories solving. This is sufficient context to then state our three-fold thesis. Finally, we declare where some content appeared previously, list our contributions, and provide an overview of the remainder of this document.

1.1 Motivation

In *Inductive Logic Programming*, the aim is to machine learn relational programs from data [117, 136]. The central task is to automatically find a logic program – which can make use of a library of existing predicate definitions – which fits our positive and negative examples. Generally, this task can be phrased as a search problem [31]: given a huge (though typically finite) set of *hypotheses*, i.e. candidate programs, efficiently search for a program that best generalises the examples, i.e. has the best fit. Compared to other more prominent methods for machine learning, major advantages of ILP are that few examples are required and that learned relations are expressed as logic programs. Hence the learned representations are human-readable and interpretable [116] and also allow for such techniques as transfer learning [28]. For this reason, ILP has been used to address tasks such as learning (explainable) classifiers, robot strategies, game rules, and preference learning, among others [31].

Program induction, i.e. learning programs from examples, fits within the wider framework of *program synthesis* [148], where, given a specification, the goal is to generate a program that *provably* possesses the required property. Various specifications are used [67]: encodings of pre- and post conditions, different kinds of types, traces as well as test cases, i.e. examples. Often functional programs are targeted, in particular those that perform transformations on structured data, such as lists and trees [51, 130]. For example, the following serves as the *refinement type*-encoded property [71] on list lengths that must

hold for *all* output lists given every possible input list[1]:

$$\text{droplast} : \{\, xs : [\alpha] \mid \text{len}(xs) > 0 \,\} \rightarrow \{\, ys : [\alpha] \mid \text{len}(xs) = \text{len}(ys) + 1 \,\}$$

A popular avenue to finding solutions is by a deductive divide-and-conquer approach, obtaining specifications for sub-problems that can be recursively solved. Typically an off-the-shelf solver for standard logic fragments [11] determines if a given decomposition gives strong enough sub-problems to guarantee the validity of the current property.

Recently, within both functional program synthesis [3, 51] as well as in ILP [95, 139, 13], there is a trend to delegate the search for hypotheses to off-the-shelf solvers. The idea is to represent the space of candidate programs by a logical formula and have state-of-the-art SAT/SMT-solvers [44, 10, 58] search for satisfying assignments. Each satisfying assignment can then be checked for whether its corresponding candidate program is actually a solution, with the solver gaining information about how to prune the search space in case the candidate fails.

Modern solver-oriented ILP systems either reduce the whole problem to a single (propositional) solver [76, 26, 13] or separate solvers for search and evaluation [95, 139, 2, 13]. Even when evaluation is handed off to a separate solver, evaluation is often still essentially propositional (by way of (a form of) grounding), leading to scalability issues with large (or infinite) domains [114]. Generally, these systems use propositional-variable-per-clause encodings to represent the hypothesis space, and hence limit themselves to small sets of candidate rules. In all, these modern ILP systems are unsuited for program synthesis tasks which often involve infinite domains, like learning transformations on lists, and searching through large hypothesis spaces.

In this dissertation, we develop a general approach to ILP that leverages state-of-the-art solver technology to efficiently search the hypothesis space for us while addressing the above issues. By allowing non-propositional solvers – such as a Prolog interpreter – to check whether hypotheses are a solution, we can address tasks that involve infinite domains, like list transformations. By addressing the hypothesis space encoding issue, we can search through large hypothesis spaces. This encoding change also enables us to learn better constraints from failed programs and to apply techniques from program synthesis (such as reasoning about refinement type-like properties) to ILP.

1.2 A fully logical perspective on inducing programs

This document takes a fully logical stance on how to automate programming. Our point of departure is a logical perspective on the philosophy of science. Following on, we represent all aspects of program induction in formal logic – including the programs that we are learning. Wanting to benefit from the advances made in solvers for logic problems, we subsequently cast our approach in their framework.

[1] The syntax is that of a *typing judgement*. This "judgment" asserts that the function associated with the symbol droplast satisfies a refinement type, i.e. the formula to the right of the colon. This refinement type denotes the class of functions which for every non-empty list (over an arbitrary element type α) as an input, here named xs, yield an output, here named ys, of the same type such that the output list is exactly one element shorter than the input list.

1.2.1 Falsification-based methodology

We take Karl Popper's falsification principle [133] – that we can never fully accept theories, but rather only get to reject them as they get falsified – as the basis of a very simple account of the practice of science.

Starting from a huge set of *a priori* viable hypotheses, a scientist selects a hypothesis to test. This hypothesis typically gets falsified as it fails to fit the data in some specific way. By examining the failure, the scientist learns that an entire space of related hypotheses fails due to the same reason and hence is ruled out as well. Having thus reduced their set of viable hypotheses, the scientist now selects from those that remain.

We apply this same methodology to program synthesis. In this setting, hypotheses are programs. The tests these hypotheses are submitted to is to run them on training data. A failure occurs when a program incorrectly relates an input to an output.

1.2.2 Inductive Logic Programming

In addition to harnassing the logic of science – i.e. the above empirically-based methodology – we also embrace *formal* logic for its expressiveness and well-developed theory. In particular, we choose to represent our specification, i.e. data (and later on properties), as well as the programs we set out to learn in formal logic.

This setting – logic-based induction of logic programs – corresponds to the field of Inductive Logic Programming (ILP) [117, 122, 136]. In ILP, the data we are trying to learn from – positive and negative examples – are represented as sets of as logical atoms, e.g. $\mathcal{E}^+ = \{\text{last}([a,b,c],c), \text{last}([1,1,2,3,5],5)\}$ and $\mathcal{E}^- = \{\text{last}([a,l,i,c,e],c)\}$, respectively. These examples introduce a new symbol – in this case the predicate last/2 – for which we would like to find a definition. As we are interested in terse and expressive definitions, we target relational programs, i.e. programs that do not assume a functional relationship between arguments. We use first-order logic programs to represent the learned relations, i.e. sets of first-order logic rules defining predicates in terms of other predicates, e.g. $\text{last}(A,B) \leftarrow \text{reverse}(A,C), \text{head}(C,B)$.

This brings us to the question of which hypotheses to consider. First, we need to provide a library of definitions for existing predicates, such as reverse/2 and head/2. This library is known as *Background Knowledge* (BK) in the ILP literature. With respect to our choice of BK, we can say which hypotheses are well-defined. This sets us up for fixing our hypothesis space: any set of logic programs well-defined with respect to our BK.

1.2.3 Satisfiability Modulo Theories

As we aim to produce systems capable of logic-based program induction, we turn to the state-of-the-art in solvers for logic problems. Despite dealing with theoretically hard problems, solvers for satisfiability (SAT) problems keep on making huge strides and are able to deal with larger and more complex problems year-on-year [54].

A key insight we will rely on is that keeping track of the viable hypotheses can be framed as a (series of) SAT-problem(s) [3]. First, consider encoding a hypothesis space of choice as a propositional formula such that its satisfying assignments are in one-to-one correspondence with the hypotheses in this space (i.e. there is an isomorphic mapping from the models of this formula to the hypotheses). Then, whenever we need to rid

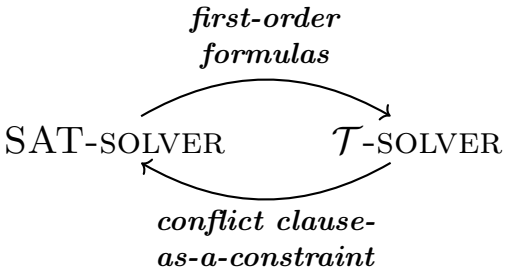


Figure 1.1: In the SMT-paradigm, a SAT-solver, reasoning about propositional formulas, collaborates with a solver for determining satisfiability of first-order (sub-)formulas with respect to a theory $\mathcal{T}$.

the hypothesis space of no-longer-viable programs, we can alter the formula so that the assignments for these programs no longer satisfy the formula.

Many modern satisfiability solvers tackle problems going beyond propositional logic: Satisfiability Modulo Theories (SMT) solvers [44, 10, 70] additionally allow first-order atoms to occur in our formulas, making use of predicate symbols whose interpretation come from *background theories* [12]. In this paradigm, the SAT-solver co-operates with a theory solver – see Figure 1.1 – where the former makes guesses for propositional assignments and derives which first-order formulas must hold while the latter checks that the conjunction of these first-order formulas is satisfiable with respect to the background theory. When this is not the case the SAT-solver is informed by the theory solver (by means of injected conflict clauses) that related guesses will always lead to unsatisfiability.

1.3 Learning From Failures

By applying falsification-based methodology to Inductive Logic Programming we derive our *Learning From Failures* framework. Our approach fits within the recently formulated class of *Conflict-Driven* ILP [92] and can furthermore be interpreted as the Satisfiability Modulo Theories approach to ILP. After contrasting our LFF implementation with existing conflict-driven systems, we have enough context to state the thesis of this dissertation.

1.3.1 Three-stage approach

Like many others working on ILP before us [115, 81, 80, 151], we are inspired by the process of scientific discovery. Like ILP's originator, Shapiro [145], we are particularly guided by Popper's perspective on science [133]. We hence apply our preceding description of the methodology of the empirical sciences to the setting of program induction.

We recognize three distinct tasks in our falsification-based account of induction:

- Select a program from the set of remaining viable hypotheses.
- Execute the selected program on the training data and observe how it fails.
- Eliminate hypotheses which must fail, as learned from analysing the observed failure.

According to our account, these tasks occur one after another in a loop. As we seek to automate program induction as a whole, we assign a component to each task: a *stage*. We are now ready to introduce the core of the Learning From Failures approach to solving Inductive Logic Programming problems: three stages in a loop – see Figure 1.2.

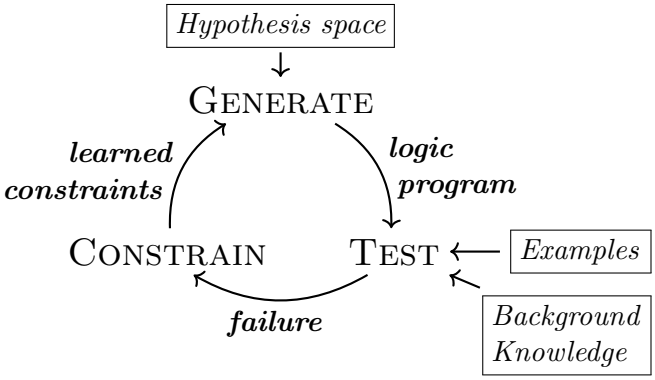


Figure 1.2: Learning From Failures' generate, test, and constrain loop.

Generate The *generate* stage is responsible for selecting a new hypothesis from among the viable hypotheses that are left. We choose to represent the hypothesis space by a logical formula such that each viable hypothesis, i.e. a logic program, is represented by a unique satisfying assignment of this formula. This stage is also tasked with ensuring this logical representation correctly tracks the set of all still viable hypotheses.

Test The *test* stage takes a satisfying assignment from the generate stage and interprets it as a logic program. The test stage evaluates this program on the positive and negative training examples, with the BK providing definitions for predicates that the program uses. When a positive example is not *entailed*, the test stage observed the hypothesis failing as it is too specific. When a negative example is entailed, the failure observed is that the hypothesis is too general.

Constrain The *constrain* stage is responsible for refining the hypothesis space based on the just observed failure. When the guessed hypothesis was too specific, i.e. failed on a positive example, all programs which are more specific can be pruned. Similarly, when the guessed program was too general, i.e. failed on a negative example, all hypotheses more general than it can be discarded. Based on this analysis, the constrain stage injects constraints into the generate stage's formula, eliminating a class of hypotheses which must necessarily fail.

By running these stages in a loop, each iteration reduces (the size of) the set of viable hypotheses. Hence, given a suitable search strategy, if there are viable hypotheses that do fit the training data, one will eventually be found.

1.3.2 Conflict-Driven Inductive Logic Programming

In the last decade or so, *meta-level* ILP systems [69, 31] – which encode the search for hypotheses as a logic program – have gained popularity. Many of these systems use an (effectively) propositional encoding [26, 76, 95, 139, 13] that can be handed off to an off-the-shelf SAT-solver [44, 58, 47]. These systems almost exclusively[2] keep to *synthesis-as-rule-selection*: they enumerate the set of possible rules, i.e. clauses, associate

[2]INSPIRE [144] and Zaatar [2] are exceptions – see Section 2.1.2.

a propositional variable to each rule, place some constraints on rule combinations and then hand off the problem to a SAT-solver.

Some of these systems [13, 26, 76] additionally encode the checking of whether a hypothesis is a solution in the same meta-level program. While conceptually simple, these systems scale poorly both in the size of the hypothesis space – due to needing to enumerate all rules to associate a decision variable to each one – and the size of the domain – where up-front grounding[3] proliferates terms from the domain in all possible places [114]. Both these factors cause a blow-up in the size of the SAT-encoding, making tackling larger problems infeasible.

We avoid the latter problem by separating hypothesis selection from hypothesis evaluation, by having separate generate and test stages. The idea is to have a meta-level encoding just for hypothesis selection, to iteratively find a satisfying assignment of the encoding, interpret the assignment as a hypothesis that can then be checked separately and from whose failure we can infer how to adjust the encoding, i.e. learn a constraint from the encountered *conflict*.

For ILP systems that delegate search for viable hypotheses to a SAT-solver, have a separate solver check these guessed hypotheses and then explicitly learn constraints from the failed hypotheses, we adopt the name *Conflict-Driven Inductive Logic Programming* (CDILP), after Law [92], who recently coined the term for his version of this approach for learning answer set programs. With our characterization of Conflict-Driven ILP in hand, we recognize that the ProSynth [139] and (a version of) ASPSynth [13] (and, to a degree, Zaatar [2]) systems for synthesizing Datalog programs from examples, from the Programming Languages community, also fit the paradigm.

1.3.3 The Satisfiability Modulo Theories-perspective

One of our central observations is that the Conflict-Driven ILP approach – of having a SAT-solver make guesses whereupon (first-order) formulas derived from these guesses are checked by another solver – is exactly the set-up of Satisfiability Modulo Theories [12] (something Bambenek et al. (2023) [13] also hint at). With this in mind, our Learning From Failures framework serves to make this SMT-perspective on ILP formal.

While SMT-based solving of program synthesis problems has spawned its own subfield – Syntax-Guided Synthesis (SyGuS) [3], see Section 2.3.3 – our approach differs in a couple of important respects from both conventional SyGuS and SMT:

1. Most SyGuS work is oriented towards *deductive* program synthesis. Here the goal is to prove that a formula – stating a desired property of a to-be-generated program – holds given the (user-provided) formulas which express properties of the functions that make up the hypotheses. Hence, in SyGuS the focus is generally on how (the semantics of) the properties relate to one another rather than how the program behaves on test cases (i.e. examples) w.r.t. the actual concrete semantics of the program (i.e. the actual mapping/relation it implements).
2. As such, the usual approach is to use the two-stage set-up of Figure 3.1, but to have the theory solvers perform validity checks, i.e. show that the desired property provably follows from the properties of the functions that occur in a selected hypothesis. Our formalization has all theory solvers, i.e. the *Horn*-solver and the solvers for the

[3]The process of turning a first-order program into a propositional one.

property theories, perform satisfiability checks, as befits their conventional usage in SMT.

3. While the conventional SMT-paradigm fixes the theory by which symbols from first-order formulas are interpreted upfront, this is not a good fit for the logic programming paradigm. The intended interpretation of the predicate symbols defined by a logic program hypothesis depends on (the interpretation of) the predicates occurring in the definitions that make up the hypothesis. We hence make the interpretation of the selected first-order formulas depend on the formulas, in particular by their least fixed-point semantics so as to support recursive definitions (as this is not possible with conventional SMT-solvers, many SyGuS approaches eschew synthesising recursive programs).
4. Relatedly, in conventional SMT there are only a limited number of background theories [11, 14], each specifying a fixed (set of) interpretation(s) for predicate (and function) symbols. However, by our perspective, each background knowledge library serves to fix the interpretation for a set of predicates – by its least model/least fixpoint semantics – thereby making each choice of $\mathcal{BK}$ into a separate background theory.
5. Finally, SMT (and by extension, SyGuS) is conventionally two-staged and has the SAT-solver learn straightforward conflict clauses [105, 17] from the theory solvers, i.e. formulas which say that (a subset of) the currently selected formulas cannot be selected together again. By contrast, we go further by learning constraints from conflicts detected by the theory solvers, i.e. test stage, which not only prune future guesses involving the tested formulas but also related programs, which, for instance, might have some variables renamed.

Additionally, our framework serves to demonstrate how techniques found in automated reasoning and SMT-based program synthesis can be applied to ILP, e.g. dynamic symmetry pruning [63] and over-approximating property checking [51]. We also investigate the connection between *failure explanation* of logic programs, i.e. identifying which parts of programs are the cause of a failure, and *conflict explanation*, i.e. identifying which sub-formulas are the cause of a conflict, as used by satisfiability solvers [12, 17].

1.3.4 Popper

This dissertation also presents our implementation of Learning From Failures: `Popper`. `Popper` directly implements the three-stage loop of Figure 1.2. The generate stage is implemented by a (propositional) ASP-solver [58], primarily so we can leverage Answer Set Programming – widely adopted by the ILP community due to its effectiveness at modeling meta-level programs and reducing these encodings to propositional problems – for the encoding of the hypothesis space. For the test stage, `Popper` uses a Prolog interpreter[4] to check whether a guessed hypothesis covers the examples. Upon a hypothesis failing, `Popper`'s constrain stage turns the hypothesis into ASP-constraints based directly on the syntax of the hypothesis.

In Table 1.1 we compare `Popper` against the other Conflict-Driven Inductive Logic Programming systems, as well as against Neo [51], a conflict-driven program synthesis

[4]The solver used for the test stage is effectively a parameter. While not described in this document, we have versions of `Popper` which use ASP-& Datalog-solvers for the test stage.

	ILASP3	ProSynth	ASPSynth	Neo	`Popper`
hypotheses	ASP	Datalog	Datalog	DSL	Definite
encoding – *variable per*	rule	rule	rule	AST-node	literal
recursion	yes	yes	yes	no	yes
optimality	yes	no	no	no	yes
large & infinite domains	no	no	no	yes	yes
over-approximating properties	no	no	no	yes	yes
hypothesis constraints	no	no	no	no	yes
reasoning about redundancy	no	no	no	no	yes

Table 1.1: A comparison of `Popper` against the other Conflict-Driven ILP systems as well as Neo, a conflict-driven synthesis system for functional programs.

system exploiting SMT-solvers which served as inspiration for bringing reasoning about over-approximating properties to ILP. We now highlight how `Popper` distinguishes itself from these other conflict-driven systems.

Synthesis-as-literal-selection Building on prior *meta-level* encodings [146, 49, 26, 76], the existing CDILP systems use the *synthesis-as-rule-selection* paradigm: their hypothesis selection encoding is *a-variable-per-rule*. As demonstrated by the INSPIRE system [144], modelling with ASP allows for more granularity. Our encoding is more fine-grained by using *a-variable-per-literal*, meaning whether or not a particular atom, e.g. last(A, B), occurs in the head or body of a clause has its own decision variable[5]. As Section 4.6 shows, `Popper` can find solutions in hypothesis spaces with up to 10^{16} distinct rules, while ILASP requires (and fails) to enumerate all these clauses. Similarly, the experiments used to evaluate ProSynth [139] (and reused by ASPSynth's authors [13]) only consider up to 1000 candidate rules.

As our encoding reflects the structure of the hypothesis space, `Popper`'s constrain stage directly transforms programs into constraints that prune non-solutions based on the (subsumption-)lattice [129, 109] structure of the hypothesis space. This encoding also makes it trivial for users to carefully control the inductive bias, altering the hypothesis space by imposing their own *hypothesis constraints*.

Infinite domains Whereas all Conflict-Driven ILP systems, including `Popper`, turn the search for hypotheses into a propositional problem and avoid the combinatorial blow-up of having evaluation of hypotheses rolled into the same propositional problem, the "test stage" used by ILASP, ProSynth, ASPSynth and Zaatar is still effectively propositional, as the former requires finite groundings of ASP programs and the latter three focus only on the Datalog fragment. Our fragment of definite programs[6] properly contains the Datalog fragment by adding support for arbitrary (nesting of) function symbols. We use these to deal with unbounded arithmetic and to express structured data, like lists, both of which cannot be handled by the other Conflict-Driven ILP systems.

[5]Versus the synthesis-as-rule-selection paradigm where each variable decides all literals that occur in a clause all at once.

[6]Except for the domain issue, ASP programs are more expressive than our definite program. It is ongoing work to expand Learning From Failures techniques – such as pruning based on subsumption – to more expressive fragments.

Recursion & optimality Neo's lack of support for learning *recursive* functions – a mainstay of functional programming – demonstrates the need for careful design for this feature. One of the main advantages of the introduction of meta-level ILP systems is their support for recursion [31]. The existing CDILP systems maintain this capability. Another advantage of the meta-level approach as developed by the ILP community is the ability to optimize for textually minimal solutions. As is common for inductive program synthesis not from the ILP community, the authors of the Zaatar, ProSynth and ASPSynth systems take little stock in the need for found solutions to generalize to non-training data, and hence, like Neo, these systems do not guarantee optimality. Both ILASP and `Popper` support finding solutions with the minimal number of literals.

Redundancy reasoning The other Conflict-Driven ILP systems all explicitly adopt the Counter-Example Guided Synthesis paradigm [149]: they treat the examples as an oracle to consult and try to minimize how much evaluation on examples occurs. Each iteration, ILASP learns a *coverage constraint* which it learned with respect to just one example[7]. In contrast, we note that a hypothesis failing on not just one (positive) example but on *all* of them gives additional information. Namely, it tells us that this program fragment cannot help entail positive examples at all, making it effectively redundant. We derive a separate class of constraints for this case – see Section 4.3.3.

Failure explanation While all systems listed in Table 1.1 already exploit the notion of *conflict explanation* – identifying which sub-formulas from the checked formulas are responsible for a conflict and adding constraints based on the sub-formulas rather than the whole set of failed formulas – they do not evaluate the effectiveness of introducing this feature to their system. Our system `Hempel` is `Popper` extended with support for failure explanation. Whereas the failure explanation of the existing CDILP systems is limited to identifying subsets of clauses as being responsible, our variable-per-literal encoding makes it possible for `Hempel` to make use of finer-grained per-literal failure explanation.

Over-approximating properties We included Neo [51] in Table 1.1 as it is a prime example of a system fully adopting the SMT-perspective on program synthesis [3]. It uses a SAT-solver to track partially built-up programs and evaluates these candidates using theory solvers. Crucially, rather than evaluating them on the actual semantics, these programs get checked with respect to their *properties*. In case an example is already incompatible with the properties of a partial candidate program, all completions of that program get pruned. We apply this idea to ILP and extend the three-stage loop with further loops that are responsible for reasoning about the satisfiability of combinations of properties. In particular, our `Popper`$[\mathcal{DL}]$ system is capable of reasoning about Difference Logic properties of hypotheses.

1.4 Thesis statement: three claims

Having provided the necessary context, we now state the three-fold thesis of this dissertation:

[7]Note that ILASP's example are partial interpretations and are meant to exercise control over the *answer sets* [60] of solutions.

Claim C1. *Learning From Failures frames the Inductive Logic Programming problem as one of Satisfiability Modulo Theories.*

Claim C2. *Learning From Failures' three-stage loop serves as a flexible framework for Conflict-Driven Inductive Logic Programming systems with state-of-the-art performance.*

Claim C3. *Automated reasoning & program synthesis techniques can be applied to ILP – by the LFF-as-SMT-perspective – to significantly reduce hypothesis space exploration.*

The remainder of this chapter explains how this document defends these claims.

1.5 Structure of this document

This last section sets out the contributions of this document, highlights which material has already occurred elsewhere, and provides an outline for the remainder of the document[8].

1.5.1 Contributions

The contributions of this dissertation are as follows. The markers **C1**, **C2**, and **C3** indicate which claims a contribution primarily lends support to.

C1&C2 By applying falsification-based methodology to Inductive Logic Programming, we derive the Learning From Failures framework and formally show the correctness of a constraint-accumulation approach to refining the hypothesis space through constraints derived from failed hypotheses.

• Section 4.1 • Section 4.2.3

C1 We define θ-subsumption-based constraints that, based on a hypothesis that failed on positive and/or negative examples, soundly prune generalisations and/or specialisations of this program from the hypothesis space. We also show how to soundly prune hypotheses that cannot be optimal solutions by reasoning about redundancy.

• Section 4.3

C1 We show how Inductive Logic Programming problems can be framed as Satisfiability Modulo Theories problems, in particular by viewing background knowledge in ILP as a background theory in SMT and LFF's test stage as a *Horn*-theory solver.

• Section 4.2.4 • Section 4.4

C2 We present the generate-test-and-constrain loop approach to solving LFF problems. We introduce synthesis-as-literal-selection as an alternative to synthesis-as-rule-selection, as exemplified by our implementation of it: `Popper`. We argue that `Popper` is a featureful and flexible ILP system with state-of-the-art performance.

[8]We note here that when dealing with formulas which combine symbols from different logic fragments, we will often highlight that the symbols come from different fragments by colouring them according to their fragment.

• Section 4.4 • Section 4.5 • Section 4.6

C3 We provide a framework for reasoning about which sub-programs, i.e. parts of a program, are responsible for a hypothesis' failure on examples. We introduce a meta-interpreter-based approach to identifying sub-programs – at the granularity of literals within clauses – that cause failure.

• Section 5.2 • Section 5.3 • Section 5.5.1

C2&C3 We incorporate failure explanation into LFF's test stage. We show that failing sub-programs lead to more effective constraints, which leads to better refinement of the hypothesis space, which experimentally can reduce learning times. We argue that failure explanation corresponds to conflict explanation as done by theory solvers in SMT-solving.

• Section 5.4 • Section 5.5.2 • Section 5.6

C3 We introduce the framework of over-approximating properties, enabling reasoning about the (in-)consistency of combinations of literals within clauses. This framework makes it possible to identify classes of hypotheses that cannot be (optimal) solutions, without executing any program on examples.

• Section 6.2 • Section 6.3

C2&C3 We extend Learning From Failures, and our implementation `Popper`, to support reasoning with over-approximating properties by incorporating other theory solvers alongside the test stage. We show experimentally that this can reduce the number of iterations of the three-stage loop which leads to lower learning times.

• Section 6.4 • Section 6.5

C3 We argue that automated reasoning techniques – such as dynamic symmetry breaking, conflict explanation, property-based reasoning and parallel solving – become more readily applicable through the SMT-perspective on ILP that LFF provides.

• Section 4.4 • Section 5.4 • Section 7.2

1.5.2 Manuscripts

This thesis contains material from previous publications[9]. The following is a summary of which material appeared where previously[10]. I hereby declare that this dissertation and the work presented in it is entirely my own, except for as indicated per the following listing[11].

[9]The following work on ILP was also published during the DPhil and influenced an independently developed extension [134] of Learning From Failures for learning higher-order programs:
Andrew Cropper, Rolf Morel, and Stephen Muggleton. "Learning higher-order logic programs". In: *Machine Learning* 109.7 (2020), pp. 1289–1322

[10]The material on ILP-as-SMT and over-approximating properties will be converted into articles after the submission of this dissertation.

[11]Andrew Cropper and Jeremy Gibbons have also provided helpful comments on drafts of this document. Sebastijan Dumančić and Georg Gottlob provided helpful comments for a final revision of this document.

1. An extended abstract advocating for most of the techniques set out in this dissertation was published as:
   - Rolf Morel. "Constraint-Driven Learning of Logic Programs". In: *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI*. 2021
2. For a large part, the work in Chapter 4 was done in collaboration with Andrew Cropper. While much material already appeared in the following paper – where each of us contributed about half of the work – I have extensively rewritten it, to increase formalization and to account for my SMT-perspective on LFF:
   - Andrew Cropper and Rolf Morel. "Learning programs by learning from failures". In: *Machine Learning* 110.4 (2021), pp. 801–856
3. Chapter 4 additionally contains my work on redundancy within hypotheses and redundancy constraints, an older version of which I first contributed to:
   - Andrew Cropper and Rolf Morel. "Predicate Invention by Learning From Failures". In: *CoRR* abs/2104.14426 (2021). arXiv: 2104.14426
4. Chapter 5 is based on the following paper by me, with Andrew Cropper supervising the work, with the relation between conflict explanation and failure explanation appearing in this thesis for the first time:
   - Rolf Morel and Andrew Cropper. "Learning logic programs by explaining their failures". In: *Machine Learning* 112.10 (2023), pp. 3917–3943
5. The ideas around how to speed-up `Popper` using parallelization techniques from the satisfiability community spring from work supervised by myself and Andrew Cropper. In particular, the figures in Chapter 7 I contributed to the following paper:
   - Andrew Cropper, Oghenejokpeme I. Orhobor, Cristian Dinu, and Rolf Morel. "Parallel Constraint-Driven Inductive Logic Programming". In: *CoRR* abs/2109.07132 (2021). arXiv: 2109.07132
6. Finally, the following paper – focussed on extending prior systems with polymorphic type checking – was published early on during the DPhil. It provides an argument for that reducing the whole ILP problem to a single (propositional) solver problem leads to combinatorial blow-up:
   - Rolf Morel, Andrew Cropper, and C.-H. Luke Ong. "Typed Meta-interpretive Learning of Logic Programs". In: *Logics in Artificial Intelligence - 16th European Conference, JELIA*. vol. 11468. 2019, pp. 198–213

1.5.3 Outline

We provide the following outline for the remainder of this dissertation:

- Chapter 2 provides a survey of related work, setting out lessons learned from previous systems, the relation to other Conflict-Driven ILP, the connection to SMT-based synthesis, and existing techniques for failure explanation and reasoning with over-approximating properties.
- Chapter 3 includes logical preliminaries – fragments of first-order logic and how they get combined within the Satisfiability Modulo Theories paradigm as well as how first-order logic underpins logic programming – and can be skipped on a first pass through the document.

- Chapter 4 lays out the Learning From Failures framework and its correspondence to SMT-solving. This chapter shows the soundness of θ-subsumption-based (generalization, specialisation and redundancy) constraints and introduces the three-stage loop as well as our synthesis-by-literal-selection implementation `Popper`. We experimentally evaluate `Popper` against other systems.
- Chapter 5 adds support for identifying which sub-programs of a hypothesis caused a failure – at the granularity of literals – to the test stage. We show that within the SMT-paradigm the analog of such *failure explanation* is the conflict explanation capability of theory solvers. We implement our failure explanation with a meta-interpreter and show that identifying failing sub-programs leads to more effective constraints which can reduce hypothesis space exploration as well as learning times.
- Chapter 6 introduces *over-approximating properties*, which enable reasoning about the (in-)consistency of properties of predicates that occur together in programs. We show how satisfiability checking of the combination of these properties, with respect to first-order theories, allows us to soundly prune the hypothesis space. Making full use of the SMT-perspective, we achieve the checking of the properties through theory solvers that sit alongside the test stage, rejecting guesses for (sub-)programs before they can reach the test stage.
- Finally, Chapter 7 serves as the Conclusion. After a brief summary, we consider limitations of the presented work – along with references to already developed extensions tackling some of these shortcomings – as well as directions for future work.

Chapter 2

Related Work

In this chapter we survey the most important related approaches to Learning From Failures. We first point a reader unfamiliar with Inductive Logic Programming to resources of interest. Next, we consider how the design of Learning From Failures and `Popper` draws on five major lessons learned from existing approaches and systems. Following on, we zoom in on the systems most akin to `Popper`, the category of Conflict-Driven ILP. This leads into how Learning From Failures can be seen as Satisfiability Modulo Theories, and how SMT has been used for synthesis in other fields. Finally, we have a section on work related to failure explanation, introduced in Chapter 5, and another section on over-approximating properties, which we introduce in Chapter 6.

Inductive Logic Programming

We assume the reader is familiar with the field of Inductive Logic Programming [117]. For those who are not, we suggest the following: the textbook by Nienhuys-Cheng and De Wolf [122] and the textbook by De Raedt [136], as well as a recent survey paper by Cropper and Dumancic [31] that serves as an introduction to modern ILP.

We restrict the following discussion to modern noiseless logic program induction. Unlike the above resources, we do not stick to systems that self-identify as ILP, but also consider logic program induction systems coming from the program synthesis community.

2.1 Lessons learned

This section provides a cross-section of the literature based on the lessons that we draw from previous systems. We highlight the aspects we identify as important in earlier approaches, how the Learning From Failures approach incorporates them, and in what regard we differ from previous work.

2.1.1 Subsumption lattice structure of the hypothesis space

Classical approaches to ILP work by top-down and/or bottom-up refinement of clauses along a generality lattice [31]. As working with the entailment lattice directly is infeasible – see Section 4.3 – often the preferred relation to work with is θ-subsumption on clauses [129], making for many early systems constructing hypotheses clause-by-clause [135, 118,

16, 150, 1], with each new clause covering new positive examples. Like the rare top-down *theory* refinement [156] systems [145, 18], we use θ-subsumption on entire programs [109], though instead of θ-subsumption informing our search strategies, we encode the consequences for hypothesis selection as constraints. In contrast to these older systems – which refine a single clause or hypothesis at a time – Learning From Failures uses constraints to refine the entire hypothesis space at once.

Like the refinement approaches that came before, even a modern system like Metagol [39] has issues with search redundancy – reconsidering program fragments already rejected previously. Learning From Failures' constraint learning addresses the search redundancy issue by precisely tracking the parts of the hypothesis space that are already ruled out, and ensuring these are never entered again. The idea is similar to that of version space algebra [110, 91], where a lattice structure on the hypothesis space is used to track a minimal representation of all programs (still) consistent with the (seen) examples. The learning of constraints can be seen as the *dual* of version space algebra: the learned constraints are a representation of the hypotheses which *cannot* be solutions.

2.1.2 Meta-level search

A major concern for many of the older ILP systems is how to implement their own search, e.g. by coming up with refinement operators [122] to guide the search along a lattice. A relatively recent shift in ILP is the move towards *meta-level* systems [69, 120]: systems which treat clauses and/or hypotheses as logical objects, such that we have logic programs that reason about logic programs [31]. The appeal is at least three-fold:

Off-the-shelf solver With a *meta-program* encoding the hypotheses, search for a solution of the meta-program can be delegated to existing solvers and interpreters, after which we can translate these solutions to solutions to the ILP problem. Many systems delegate this search to (propositional[1]) SAT-solvers [2, 146, 139], with Answer Set Programming encodings being especially popular [18, 26, 95, 13, 76]. The Meta-Interpretive Learning approach [121], exemplified by Metagol [39], stands apart as it uses a Prolog *meta-interpreter*, i.e. a program that extends a normal interpreter with additional rules – see Section 5.5.1 – effectively interleaving the search for (clauses of) hypotheses with evaluation of partial programs, relying on first-order SLD-resolution for both. Learning From Failures adopts the idea of using off-the-shelf solvers, though insists on two separate solvers. The generate stage is envisioned as any standard (propositional) SAT-solver searching for hypotheses. The test stage is a separate solver capable of evaluating first-order logic programs against examples.

Recursion ILP systems find it hard to generalise without recursion [41]. Learning recursive programs has long been considered a difficult problem in ILP [119]. Systems that use (bottom-)clause refinement [135, 118, 16, 150, 1, 144] learn programs clause-by-clause which often leads to learning very specific programs. In particular, support for learning recursive programs is very limited, e.g. only under condition that the examples explicitly include the base case and all relevant step cases. Like other meta-level systems

[1]Either the meta-level logic programs are written in propositional logic directly, or they are written in a variant of first-order logic, such as Answer Set Programming, which, in all systems surveyed, gets *grounded* to a propositional logic program.

[18, 26, 95, 39, 50, 76], we overcome this limitation by searching over hypotheses instead of clauses.

Optimality In ILP, we are interested in programs that best generalise from the training data/examples. A central observation is that smaller programs generalise better, with the (implicit) assumption being that the smallest program(s) generalise best. Many ILP systems cannot guarantee optimal solutions – like the above clause refinement systems and even some modern conflict-driven systems [139, 13] – where optimality is defined with regard to program size. By searching over entire hypotheses, meta-level systems can learn optimal solutions, either w.r.t. the number of clauses [39, 76], or, like `Popper`, w.r.t. the number of literals [26, 95].

2.1.3 Granularity of encoding: rule selection vs. literal selection

In the last decade, *synthesis-as-rule-selection*, pioneered by Corapi [25, 26], has emerged as the dominant paradigm for meta-level systems. The idea is to have a representation of the set of candidate rules and a mechanism for selecting viable combinations of these rules, e.g. [146, 49, 147]. The most popular approach [26, 139, 95, 76, 13] is to maintain the viable rule combinations by a propositional/constraint formula over rule identifiers. By asking a solver to look for an (assignment identifying a) *minimal* (i.e., smallest size) subset of the candidate rules that is a solution, optimality can be guaranteed. While conceptually simple, all these systems share issues around needing to pre-compute the set of candidate rules – often infeasible for large hypothesis spaces – and that constraints that aim to prune based on symmetries/structure of the hypothesis space scale with the size of the rule space, i.e. may need to refer to many rule ids as a variable-per-rule encoding exposes very little structure.

We adopt the idea of tracking feasible hypotheses by a formula and address the above issues by moving to a *synthesis-as-literal-selection* approach, where we encode which hypothesis gets selected at the granularity of a variable per literal. Our encoding makes it possible to talk about the structure that clauses and hypotheses share, thereby making constraints that prune based on subsumption feasible – see Section 4.3. Unlike Atom [1] and INSPIRE [144] – which use a variable-per-literal and variable-per-predicate/argument encodings, respectively, and essentially implement a clause refinement approach [118] with a SAT-solver – our encoding reasons about entire hypotheses and not just clauses.

2.1.4 Declarative, constraint-based encoding from a relaxed bias

ILP systems use a language bias [122] to specify the hypothesis space. The most popular one [16, 150, 26, 140, 25, 26, 7, 1, 95] is *mode declarations*. Mode declarations [118] state which predicate symbols may appear in a clause, how often they may appear, the types of their arguments, and whether an argument must be assigned a constant symbol or a variable. We simplify this form of bias (similar to Aleph's determinations [150]): our *declaration bias* is in terms of predicate declarations (Section 4.5.1), which only state which predicates can appear in the heads and bodies of clauses. This relaxed bias gives a very general hypothesis space which we can optionally refine through imposing constraints.

Many modern ILP systems impose a strong syntactic bias: *metarules* – i.e. higher-order clauses which act as templates that clauses of hypotheses must keep to – are particularly

popular [138, 155, 39, 49, 76], also to select the set of candidate rules [146, 139, 13, 2, 146]. Compared with predicate declarations, metarules are a much stronger inductive bias and are hence effective at restricting the search space, though justifying which combination of metarules to provide is often hard [42]. With `Popper`, if a user is able to decide on suitable metarules, we can optionally impose them using *hypothesis constraints*, see Section 4.5.3.

A number of ILP systems allow a user to constrain the hypothesis space through clause constraints [118, 150, 16, 1, 95]. Unlike these systems, we specify constraints on *entire* programs, i.e. sets of clauses. Following a recent trend [95, 76, 144, 13], we model our hypothesis spaces as a first-order constraint problem written in Answer Set Programming [57] (ASP). The combination of ASP and synthesis-as-literal-selection allows for a natural, declarative way of encoding hypothesis selection. That is, we show that many types of constraints on the hypothesis space [69, 144] can naturally be represented as first-order program fragments, e.g. type checking, imposing metarules, and, most importantly, learned constraints. For example,

```
:- body_literal(_,p,2,_),body_literal(_,q,2,_).
```

says that predicates `p/2` and `q/2` cannot occur in (the bodies of) a hypothesis together, even if they are in different clauses. Athakravi et al. [6] introduce *domain-dependent constraints*, which are constraints on the hypothesis space provided as input by a user. INSPIRE [144] also uses pre-defined constraints to remove redundancy from the hypothesis space (in INSPIRE's case, each hypothesis is a single clause). `Popper` also supports such constraints but goes further by learning constraints from failed hypotheses.

2.1.5 Decompose generate & test

Much recent work in ILP uses Answer Set Programming to learn Datalog [49, 139, 107, 13], definite [120, 76, 32], normal [140, 26, 7], and answer set programs [95]. All of the synthesis-as-rule-selection systems [146, 49, 147, 26, 139, 95, 76, 13] as well as many other modern systems [146, 49] in practice only support hypotheses from logic fragments which are essentially propositional, as the solving techniques these systems employ require finite groundings (of the (program representing the) hypotheses along with the Background Knowledge and the examples) in order to evaluate them. Some of these systems [26, 76, 13] even attempt to reduce the entire ILP problem down to single program solvable by a single (propositional) solver. This approach is particularly prone to combinatorial blow-up [114]. Systems that do separate their testing from their hypothesis selection, like ILASP [95], rely on grounding all rules with all potentially relevant constant symbols, and hence still easily experience combinatorial blow-up – see Section 2.2.1 and Section 4.6.2.

Like most ILP systems developed to address program synthesis tasks [145, 18, 1, 39], such as list transformations, we target definite programs. In order to overcome the grounding limitation – which occurs for each of the above systems on the (infinite) lists domain – `Popper` combines ASP and Prolog. `Popper` uses a Prolog interpreter to evaluate definite programs[2], which allows it to reason about large and infinite problem domains, such as lists and numbers, as SLD-resolution-based proof search only involves the terms necessary for proving entailment. Hence, like many modern systems, we utilize

[2]Furthermore, all that `Popper` actually require of the $\mathcal{BK}$ program is that it behaves like a query-able infinite set of facts, and hence could be specified as a normal-program, ASP-program, or something else.

state-of-the-art solvers, but see hypothesis selection and hypothesis evaluation as distinct (combinatorial) problems.

2.2 Conflict-Driven Inductive Logic Programming

A number of logic program induction systems effectively do reduce the entire ILP problem to a single SAT-solver invocation [76, 26, 7, 13]. When the resulting propositional problem is not too large, good performance can be achieved with modern SAT-solvers. A key technique employed by these solvers is Conflict-Driven Clause Learning [105], which allows a SAT-solver to infer which decisions leading up to a conflict are actually responsible and should be saved as a constraint on subsequent search in the form of a *conflict clause* (also called a *nogood* in the ASP community [60]). As discussed, these reductions lead to intractability in case of large hypothesis spaces and/or large domains, which makes addressing tasks such as learning list transformations infeasible.

We address this limitation by separating hypothesis selection from hypothesis evaluation. We assign a solver for each task, thereby keeping the two combinatorial problems separate. The search for hypotheses we still delegate to a SAT-solver, as these solvers are very effective at keeping track – by way of learned constraints – which parts of the search space are already ruled out. The evaluation of hypotheses we delegate to a separate solver, which in effect adds an additional check for whether an assignment guessed by the SAT-solver is conflicting or not. If the evaluation solver detects a conflict, i.e. that a guessed hypothesis is not a solution, we are responsible for learning an appropriate constraint from the detected conflict and injecting it into the SAT-solver.

After the publication of Learning From Failures [35], Law [92] – the author of the ILASP systems [93] – coined the term *Conflict-Driven Inductive Logic Programming* for essentially this approach to ILP. The three key ingredients are (1) using a SAT-solver to search through a propositional encoding of the hypothesis space[3], (2) using a separate solver to test/check/verify guesses for hypotheses made by the SAT-solver, (3) explicitly learn a constraint if a guessed hypotheses fails to be a solution. Unlike Law, we identify multiple systems that fit this general template.

Whereas in Section 1.3.4 we already gave an overview of the main points where our approach differs from/improves upon the existing CDILP systems, we now zoom in on how these systems work individually and contrast them directly, one-to-one, with `Popper`.

2.2.1 Inductive Learning of Answer Set Programs

Popper does not rely on an "external propositional solver in the way that ILASP does: ILASP uses an ASP-solver both for hypothesis selection, like we do, and a separate instance of an ASP-solver as its "external" solver, i.e. theory solver. more instancen an attempt to convey it more clearly, I have restated the general point (of littering symbols necessary for evaluation through the encoding of the hypotheses, even when they are entirely extraneous leading to a straightforward Cartesian product type blow-up) in Section 2.1.5. I have improved the discussion on ILASP by making it clearer that ILASP still has

[3] In this section we restrict ourselves to systems that use meta-level encodings representing hypothesis spaces of full programs and not just clauses, due to the issues with clause-by-clause learning outlined in the previous section.

the full Cartesian blow-up due to it involving all rules from the rule space in its evaluation stage.

First we consider Law's Inductive Learning of Answer Set Programs approach to ILP, which covers an array of systems all focussed on learning answer set programs. Whereas ILASP1 [95] and ILASP2 [97] are meta-level systems that combine hypothesis search and hypothesis evaluation, the ILASP3 and ILASP4 systems [92] – which Law identifies as *the* Conflict-Driven ILP systems – consist of four distinct stages: (1) hypothesis search, (2) counter-example search, (3) conflict analysis, and (4) constraint propagation. As the latter stage is optional and only used in the setting were examples may be misclassified, which the techniques in this dissertation do not address, we will disregard it here.

The hypothesis selection phase uses a synthesis-as-rule-selection encoding, maintaining a propositional constraint problem over rule ids, with an ASP-solver [60] looking for new satisfying assignments. Upon a new satisfying assignment being found, the selected rule ids are mapped to actual ASP-rules and handed off to a separate solver.

This ASP-solver does counter-example search, i.e. it looks for a positive or negative example – which are partial interpretations [95] which either should or should not be included in the answer sets of solutions – that shows that the guessed hypothesis is not a solution.

The next stage is conflict analysis, which takes the counter-example and the full space of candidate rules and uses them to determine which rule combinations also fail to satisfy this counter-example. In ILASP3, this leads to a *coverage constraint* that only holds for rule combinations which necessarily cover the current counter-example. As Law himself points out [92], this step is enormously expensive and leads to large constraints involving many rule ids. ILASP4 addresses this issue by looking for weaker constraints, ones that prune non-solutions but might not prune all of them (while still needing to ground all of the candidate rules with the terms relevant to the counter-example to determine such a constraint). In either case, the coverage constraint is added to the constraints of the hypothesis selecting solver and the next iteration of the loop starts. In ILASP3, the number of iterations is bounded by the number of examples, as each counter-example is *translated* into a constraint that only holds of solutions.

Versus `Popper`

Each learning task in this dissertation can be represented as a single ILASP positive partial interpretation example, which in case of ILASP3 would mean only a single iteration of the loop would occur[4]. In `Popper`, each iteration evaluates just the currently guessed hypothesis on the examples – a relatively cheap operation versus the evaluation that happens in ILASP's analysis phase. Additionally, as we use a Prolog interpreter for evaluation, we do not require that our program (along with the BK) can be grounded by all possible relevant terms, which makes it possible for us to address list transformation tasks while ILASP cannot – see Section 4.6.3.

The variable-per-rule encoding employed by all ILASP systems requires enumerating the set of candidate rules and makes for many propositional variables. This causes search

[4]For this reason, ILASP3 performs much better if the examples are split into one partial interpretation example per atomic example. Hence in our experiments we provide each `Popper` example as a separate ILASP partial interpretation example.

through large hypothesis spaces to be infeasible[5]. By contrast, `Popper`'s synthesis-as-literal-selection encoding does not fully construct the set of all rules in the hypothesis space. Our encoding makes it possible to find solutions in hypothesis spaces that contain approximately 10^6 and 10^{16} rules, as in the experiments of Section 4.6.1 and Section 4.6.2. On these same experiments, ILASP3 fails to learn hypotheses with clauses containing more than a couple body literals. Furthermore, as our encoding reflects the structure of the hypothesis space, we are able to derive constraints that prune based on θ-subsumption.

Ignoring the domain size issue, the ASP-programs learned by ILASP are more expressive than `Popper`'s definite programs, supporting, e.g., default negation and disjunction in the heads of rules. Another advantage of the ILASP systems is that they support noisy examples. In Section 7.2, we consider ways `Popper` might be extended to more expressive program fragments and how noise might be handled.

2.2.2 Inductive synthesis of Datalog programs

We now describe the three systems from the programming languages/synthesis community that qualify as Conflict-Driven ILP. We highlight how our Learning From Failures framework and implementation `Popper` differ from these approaches.

Zaatar To our knowledge, Zaatar [2] is the oldest instance of our template for Conflict-Driven ILP. Zaatar is a meta-level system that tries to reduce the learning of Datalog programs from examples to a single SMT-solver problem (using the Z3-solver [44]), though requires a verification stage as this reduction is incomplete. Zaatar employs a fine-grained hypothesis selection encoding, similar to that of INSPIRE, separately selecting predicate symbols and the variables that occur as arguments. Hypothesis evaluation is encoded alongside hypothesis selection: a theory of arrays [106] encoding emulates how *derivation trees* for the examples get build up to a bounded depth in accord with the current guess for a hypothesis. Due to the depth bound, failing to prove entailment does not mean an example is not entailed. The guessed hypotheses for which coverage of examples cannot be fully determined by the SMT-solver itself get handed off to be verified by Z3's *Horn* solver, which evaluates these hypotheses on the examples as normal. In case a thus tested hypotheses fails, a *blocking constraint* is derived and injected into the SMT-solver. This constraint prunes the failed hypothesis and variants of it, i.e. this particular program as well versions of it where the variables have been renamed.

`Popper` differs in that we do not attempt to (under-)approximate the semantics of guesses for hypotheses. Zaatar's under-approximating scheme requires a small Herbrand base, meaning it not suitable for learning (definite) programs addressing domains like lists. Zaatar's learned blocking constraints only account for the symmetry around variable renaming, while `Popper` manages to prune entire sub-spaces of related programs with its subsumption-based constraints. Zaatar does not guarantee learning textually minimal programs.

ProSynth The ProSynth system [139] is a straightforward meta-level system using a synthesis-as-rule-selection encoding, with Z3's SAT-solver making guesses for rule combi-

[5]The FastLAS systems [94, 98] are variants of ILASP meant to address this limitation. In doing so they give up a number of features, including the main benefit of meta-level systems: learning recursive programs.

nations. Each thus selected viable hypothesis is run on examples using the Souffle Datalog solver [72]. ProSynth uses Souffle's *query provenance* [23] to identify which rules of the selected hypothesis caused a negative example to be entailed or a positive example to not be entailed. These rule sets are straightforwardly translated into constraints: in case a set of rules failed to entail a positive example, no subset of these rule ids cannot be selected as a hypothesis, and in case a negative example is entailed by a rule set, no superset of the corresponding rule ids can be selected as a hypothesis.

ProSynth is similar to `Popper` in that failed hypotheses get directly translated into constraints and the learned constraints prune along the subsumption lattice. However, due to ProSynth variable-per-rule encoding (versus `Popper`'s variable-per-literal), its constraints are very coarse and do not account for the variable renaming aspect of θ-subsumption. Due to this change in encoding, `Popper` can also handle much bigger hypothesis spaces. Again, in the experiments of Section 4.6.1 and Section 4.6.2, we consider hypothesis spaces that contain approximately 10^6 and 10^{16} rules, whereas the experiments used to evaluate ProSynth only use up to a 1000 candidate rules. As opposed to ProSynth's learning of Datalog program, we learn definite programs allowing us to address tasks with arbitrarily large domains, like lists.

ASPSynth In their 2023 paper [13], Bambenek et al. present a gradation of three Datalog induction systems. *MonoSynth* is ProSynth but adds incremental evaluation of guesses for Datalog clauses, making the Datalog solver used for evaluation essentially into a proper SMT-theory solver. *LoopSynth* is like Zaatar in that the semantics of the logic program hypotheses get approximated in an incomplete manner, this time by a reduction to classical propositional logic. As Z3's SAT-solver does not capture the least model semantics of Datalog, *LoopSynth* needs to verify guessed hypotheses by checking them on the examples with an actual Datalog solver. If this verification fails, a *loop constraint* is derived and injected into the SAT-solver. Finally, *ASPsynth* is just ASPAL [26]: it implements rule-selection-based synthesis by adding enable-/disable-flags to all possible Datalog rules, asks the Clingo solver [58] to ground this first-order program whereupon it has the Clasp SAT-solver [60] find a setting of the flags that leads to correct entailment of examples (now using Clasp's stable model semantics to guarantee examples are evaluated with respect to the least model of each guessed hypothesis).

ASPSynth itself is not a CDILP system: it reduces the entire ILP problem to a single propositional problem. All three systems inherit the limitation of rule-selection systems, namely intractability when it comes to large hypothesis spaces. Both LoopSynth and ASPSynth compound this issue by effectively grounding all these rules with all constant symbols, yielding very large propositional problems on non-trivially sized domains. MonoSynth is most similar to `Popper`, but by relying on a Datalog solver still does not address arbitrarily large domains. In their 2023 paper [13], it is claimed that `Popper` was not competitive on their experiments. Most likely they had trouble specifying a language bias that captured the rather restricted hypothesis spaces they consider (as they reuse the ProSynth benchmarks which only consider hypothesis spaces that contain up to a 1000 candidate rules).

2.3 The Satisfiability Modulo Theories-perspective

We now give an overview of the relation of Learning From Failures to Satisfiability Modulo Theories, including how prior systems – especially the other Conflict-Driven ILP systems – can be said to possess a similar make-up.

As covered in more detail in Section 3.3, SMT-solvers [44, 10, 58] make it possible to reason about complex problems by a single formula, interpreting part of the formula as propositional, such that it can be reasoned about by a SAT-solver, with guesses for propositional assignments leading to first-order sub-formulas becoming enabled and disabled. These first-order formulas are then reasoned about by theory solvers, i.e. solvers which use a theory to give interpretations to predicate and function symbols.

We use this setup for our SMT-perspective on Learning From Failures. Typically SMT-solvers support a number of different first-order logic fragments [11]. We differ in that we allow *Horn*-formulas as the first-order formulas (similar to the formalization of MonoSynth [13]), a fragment not supported as a background theory by any SMT-solver.

2.3.1 Generate stage as SAT-solver & test stage as theory solver

In the preceding sections, we argued for having a component for selecting hypotheses and a separate component for evaluating selected hypotheses. We also chose to use a constraint formula for maintaining the set of viable hypotheses, putting each satisfying assignment, a *model*, in correspondence with a program, i.e. a set of first-order definite *Horn*-formulas. Whenever we have a model, we check if the program it represents is correct by testing it on the examples. When this is not the case, we *refine* our hypothesis space-representing formula by adding constraints to it. Generalizing a similar observation made by Bembenek et al. (2023), we argue that the template of Conflict-Driven ILP is essentially the same setup as that of Satisfiability Modulo Theories (SMT) [12].

To make the ILP problem into an instance of Satisfiability Modulo Theories, we just need to connect propositional decision making about which hypotheses to select with appropriately selecting the corresponding set of (first-order) *Horn*-formulas, i.e. the actually selected hypothesis. To actually find solutions – now models in terms of the overall formula – we need a SAT-solver and a *Horn-solver* to collaborate. From this perspective, the Background Knowledge serves as a background theory for the *Horn*-solver, as it fixes the interpretation of the predicates that can occur in the hypotheses. Our implementation, `Popper`, is in effect a domain-specific SMT-solver, making use of multi-shot solving [59] to allow incorporating the constrain stage in between the SAT-solver and the *Horn*-theory solver.

2.3.2 Constrain stage as symmetry breaking

In conventional SMT-solving, when a theory solver detects that the current set of first-order formulas is inconsistent it generates a conflict clause to inject into the SAT-solver. This conflict clauses essentially enforces that this set of conflicting formulas cannot be reconsidered, i.e. cannot reoccur as a subset. That is, the learned constraints are there to make sure we do not revisit already rejected parts of the search space.

We take this one step further: when a hypothesis fails we can infer by θ-subsumption [129, 109] many related programs, i.e. many sets of *Horn*-formulas, must fail as well. θ-

subsumption relates programs on the basis (1) of (effectively) being subsets or supersets of each other and/or (2) that their variables have been renamed. This is hence what the constrain stage accomplishes: instead of just pruning the sets of *Horn*-formulas that are a subset/superset of a conflicting set of *Horn*-formulas, we prune more sets of *Horn*-formulas, namely (up to) all those that are more general or more specific according to θ-subsumption.

The constraint programming community has long been aware of the importance of *symmetry breaking* [63]: employing constraints to rule out assignments that are considered equivalent, e.g. because of renaming. As our approach is to derive our θ-subsumption-based constraints during the search, the constrain stage can be understood as applying *dynamic* symmetry breaking [128, 45, 108] to SMT-solving. That is, normally in SMT, conflict clauses learned by theory solvers only prune subsequent guesses involving (a subset of) the formulas involved in an encountered conflict. By contrast, the constraints we learn from these same conflicts prune based on (essentially) the same subset-lattice but additionally prune assignments whose formulas correspond to renamed versions of formulas that the normal conflict clauses are capable of pruning. Hence the symmetry our constraints account for is that of programs being *variants* of one another, i.e. variable-renamed versions of each other.

2.3.3 Syntax-Guided Synthesis: SMT-based program synthesis

The SMT-community has developed its own *Syntax-Guided Synthesis* (SyGuS) paradigm [3]. In SyGuS, there is a first-order formula ϕ, using symbols which are interpreted by a background theory of choice, expressing the correctness of an unknown function f. Given a grammar to derive terms, the goal is to find a term t such that $\phi[f \mapsto t]$ can be proven valid with respect to the background theory. The most common setup [3] is as in Figure 1.1: there is a SAT-solver reasoning about the space of terms and each guess for a term triggers not a *satisfiability* check but a *validity* check by the theory solver[6]. By contrast, our formalization in Section 4.2.4 frames the ILP problem as one of pure Satisfiability Modulo Theories. That is, we are interested in satisfiability for both the hypothesis selection component and for the checking of hypotheses component[7]. Furthermore, our specification formula is simply a conjunction of ground literals, far simpler than most first-order formulas used by SyGuS systems.

CEGIS SyGuS approaches to inductive synthesis [146, 139, 13] often follow the Counter-Example Guided Inductive Synthesis framework [149]: the theory solver acts as an oracle coming up with a reason for why the current guess is not a solution, i.e. a counter-example, that can then be transformed into a conflict for the SAT-solver. Hence, CEGIS systems – including CDILP systems ILASP [92], ProSynth [139] and MonoSynth and LoopSynth [13] – attempt to minimize the number of (counter-)examples that are necessary to reject guesses. We however make use of the observation that a hypothesis failing on *all* (positive) examples gives rise to reasoning about redundancy, yielding another type of constraint

[6]The difference between satisfiability and validity is that of proving an existential versus proving a universal.

[7]To be fair, as we needed to frame our intended theory for the checking component, i.e. the theory solver, such that there is only a single interpretation to consider, our approach could be equally well-viewed as SyGuS.

that prunes hypotheses which cannot be *optimal* solutions – see Section 4.3.

The programming language community rarely concerns itself with how well solutions generalize, i.e. they rarely measure predictive accuracies [53, 126, 55, 2, 139]. In ILP we do want solutions that generalise to unseen data, and as such we report test accuracy in our experiments. In fact, many of the symbolic ML approaches can often trivially construct an overly specific solution. For instance, for each example, ILP systems can construct the bottom clause [118].

When a theory solver detects inconsistency in the formulas it has been given, it can identify a subset of the formulas as being at fault. Such *conflict explanation* leads to learning smaller, more effective conflict clauses [12, 17].

2.4 Failure explanation

We compare our introduction of failure explanation to Learning From Failures with other fields seeking to identify which parts of logic programs cause failure. We also contrast our failure explaining version of `Popper`, named `Hempel`, with other program induction systems which can be said to have similar capabilities.

2.4.1 Algorithmic debugging

Algorithmic debugging [20] explains failures in terms of sub-programs. Alongside his seminal work on logic program synthesis, Shapiro [145] introduced the notion of *debugging trees* for semi-automated identification of failing clauses. Only being able to return clauses responsible for entailing an atom is still the standard for logic programming debugging [84, 154]. Unlike these systems, `Hempel` automatically identifies literals within clauses which cause an example, i.e. atom, to not be entailed, and integrates the failure explanation process in a program synthesis system.

2.4.2 Theory revision and repair

Shapiro's Model Inference System (MIS) [145] is a theory revision system which, through interaction with a user, is capable of synthesising programs. MIS uses SLD-trees to determine which clauses of a program are responsible for entailing a negative example, at which point the user needs to say which of these clauses is wrong. To cover a non-covered positive example, additional clauses get added, possibly involving user interaction, without regard for why the current clauses do not entail this example. By contrast, `Hempel` does not require the user to act as an oracle and can automatically identify clauses *and* literals within clauses as being responsible for not entailing a positive example.

There are theory revision systems [156] able to identify literals as *revision points* within theories, i.e. programs, though often with limitations. Some require user interaction [138, 127]. FORTE [141] uses hill climbing to gradually revise a theory, heuristically following revisions that improve training accuracy. Unlike FORTE, `Hempel` is guaranteed to find an optimal solution if one exists. FORTE can automatically identify responsible literals of a sub-program, given that the examples are trace-complete, i.e. all necessary recursive calls of the target predicate are included as positive examples. Our failure explanation

algorithm automatically identifies responsible clauses and literals which cause a program to not entail an atom, without any condition on the examples.

In general, theory revision and theory repair [19] are concerned with updating a current hypothesis by applying generalisation and specialisation operators to the identified revision points. Whereas these systems refine a single program at a time, `Hempel` uses the failure of a (sub-)program to refine the hypothesis space, each time pruning away a large class of programs.

2.4.3 Conflict-Driven ILP

As the constraints that `Popper` learns are always based on entire hypotheses, `Hempel`'s failure explanation can hence be viewed as allowing `Popper` to detect smaller, finer-grained conflicts, yielding smaller and more general constraints which prune more effectively.

Some modern ILP systems can be said to have a degree of failure explanation. Metagol [39] is a meta-interpreter which uses examples to drive the search, gradually building up a program whilst partially evaluating it on an example. When a failure occurs, Metagol infers it is due to the last literal that was added, which causes it to backtrack. However, due to its iterative deepening strategy, Metagol will reconsider these program fragments many times, and has no way to learn from failures. By contrast, `Popper` and `Hempel` learn constraints which ensure that failing program fragments are never reconsidered.

The Conflict-Driven ILP systems ILASP [95, 92], ProSynth [139], and (versions of) ASPsynth [13] also all incorporate failure explanation. ILASP3 learns recursive ASP programs, with *partial interpretations* serving as examples. When a model of a selected hypothesis does not correctly extend the given partial interpretations, the hypothesis fails with the model being its *violating reason*. Constraints can be derived from a violating reason by checking which combinations of candidate rules also have it as a model, which is an expensive operation. `Hempel`'s learning of constraints by identifying sub-programs is more efficient and, by defining its hypothesis selection problem over literals, it is not restricted to identifying just clauses as causing a failure.

Like ILASP, ProSynth precomputes every possible clause and employs a select-test-constrain loop over clause ids. ProSynth uses the notion of *query provenance* [23] for identifying which clauses of a hypothesis are responsible for (not) entailing an example, encoding identified subsets as constraints. ProSynth's failure explanation is limited to Datalog programs, which is just a fragment of the definite programs which can be learned by `Hempel`. Additionally, `Hempel`'s failure explanation is finer grained as it also identifies which literals cause failure, which is enabled by our use of a synthesis-as-literal-selection encoding. The only true CDILP-version of ASPSynth, MonoSynth, explicitly builds on the select-test-and-constrain loop of ProSynth to learn Datalog program and derives its subset of clauses-based failure explanation from ProSynth. Unlike the existing CDILP systems, `Hempel` prunes programs related by θ-subsumption to sub-programs that cause failure.

None of the above systems evaluate the effectiveness of introducing failure explanation to their systems.

2.4.4 Conflict explanation in SMT

Identifying the responsible part of a conflicting assignment has been a crucial technique in making SAT-solvers more performant [105].

Conflict explanation in propositional solving

In Conflict-Driven Clause Learning [157], SAT-solvers gradually build up partial assignments of a propositional formula expressed as a set of clauses. Upon a new decision a partial assignment can become *conflicting* with the clauses, i.e. at least one clause is violated (and will be violated for any possible extension of the assignment). At this point an *implication graph*-based algorithm determines a subset of the variables assigned thus far that caused the conflict to occur. This assignment is converted into a conflict clause that gets added to the existing clauses. The solver will use this clause to detect (and avoid) a re-occurrence of the (partial) assignment that actually caused the conflict.

Given the above description of CDCL, we can infer that reducing the entire ILP problem to a single propositional problem might not be a good idea. The systems that do attempt this use grounding to turn first-order logic programs into propositional problems by proliferating ground terms in all places where variables occur [114]. In particular each first-order rule of a logic program will lead to many ground propositional versions of it. One problem of this is that many of these clauses are extraneous – will have body literals which cannot be satisfied at the same time – and will hence lead to useless unit-propagation.

A problem related to the topic at hand is that even when CDCL identifies that only certain ground literals are necessary for a conflict, it has only learned this for a particular grounding. For all other groundings of the same rules that are involved, CDCL needs to expend the same effort to determine that these same literals, now grounded to distinct propositional variables, are or are not sufficient for a conflict. Hence, CDCL naively applied to propositional problems that result from grounding first-order logic programs is doomed to re-learn essentially the same conflict many times over.

The SMT-perspective

Satisfiability Modulo Theories solvers also learn conflict clauses but expand on the CDCL-mechanism by additionally having theory solvers contribute conflict clauses [17]. A SMT-solver is a SAT-solver that additionally informs a theory solver of the first-order formulas that must hold based on the current propositional assignment. These theory solvers determine whether their set of formulas is still satisfiable in an incremental fashion. For example, suppose a Difference Logic solver collaborates with a SAT-solver and gets communicated the formula $A < B$ first, then $B < C$ and in a yet later step $B < A$. This solver should now return a conflict clause. Either it can do as `Popper` does and say the set of three literals together are responsible or it can expend additional effort to determine that only the first and last atoms were the cause and use just these two literals to return a conflict clause.

With the SMT-perspective on Learning From Failures, `Hempel` is effectively applying the notion of *conflict explanation* done by theory solvers to `Popper`. Note that while SMT-solvers [44, 10, 58] typically only support quantifier-free logic fragments our theory solver deals with quantified *Horn*-formulas. Whereas theory solvers use unsat-core techniques

[17] to identify conflict clauses, they effectively just identify a subset of asserted clauses as being the cause of a conflict, we additionally identify which parts of these clauses are responsible. Finally, versus SMT-solvers, we generalize the learned conflict clauses by reasoning about θ-subsumption to derive more effective constraints, which we express as first-order constraints that are then grounded so that they can be communicated to the underlying SAT-solver.

2.5 Over-approximating properties

We now consider how our introduction of reasoning with over-approximating properties to Learning From Failures relates to existing work. Our approach is primarily inspired by the use of SMT-encoded, property-based reasoning in functional program synthesis. We compare our `Popper`[$\mathcal{DL}$] extension of `Popper`, which adds the capacity to reason about the satisfiability of Difference Logic-encoded properties, to the most comparable conflict-driven synthesis system, Neo. Finally, we consider the select few notions and systems in ILP related to this line of work.

2.5.1 Functional program synthesis using properties

The functional programming (FP) community has long been interested in synthesis [152, 77, 82, 89]. Many FP synthesis systems [130, 52, 53, 126, 55, 68] ask for a function's properties, in terms of pre- and post-conditions, and come up with a function definition – by composing provided functions that come with similarly annotated specifications – such that the pre-condition implies the post-condition. One recent popular formalism for expressing correctness is *refinement types* [71], i.e. pre- and post-conditions encoded into function types. For example, an important property regarding the lengths of the input and output of the droplast function can be represented in a type as

$$\text{droplast} : \{\, xs : [\alpha] \mid \text{len}(xs) > 0 \,\} \rightarrow \{\, ys : [\alpha] \mid \text{len}(xs) = \text{len}(ys) + 1 \,\}$$

Largely sticking to the Syntax-Guided Synthesis paradigm [3], these approaches are interested in establishing the validity of these properties[8], e.g. that the above property holds for *all* possible input and output lists. To do so they assume that the annotated properties of the provided functions are sufficiently strong to *prove* that the composed program has its required property [130, 83, 52, 68].

Coming at it from an inductive perspective, where information is often incomplete, we judge this assumption as too strong. We use *over-approximating* properties, not to prove that our programs are guaranteed to have certain properties but to prune (sub-)programs that provably cannot satisfy our examples. The Neo system [51] is one of the few FP synthesis systems that takes a similar view.

Program Synthesis using Conflict-Driven Learning

Neo [51] synthesizes non-recursive functional programs (keeping to a user-provided DSL) assuming that the examples are encoded as a single SMT-formula and that each provided

[8] A couple of these refinement type approaches do extend their formalism to reason about examples as well [126, 55].

background knowledge function comes with a property encoded as a SMT-formula as well. Like the other conflict-driven systems we have already considered, Neo consists of three stages. Neo's first stage has a SAT-solver gradually build up programs. Its second stage asks an SMT-solver if the conjunction of the properties of a partially constructed program are consistent with the SMT-encoded examples. If not, this means executing the program on an example cannot yield the desired result. This triggers the conflict analysis stage, where an SMT-solver is asked to check if other combinations of properties of the provided functions are implied by the combined property of the just considered partial program. Each program fragment which possesses a combined property thus identified must necessarily be inconsistent with the SMT-encoded examples as well. Hence Neo's conflict analysis stage learns *lemmas*, i.e. conflict clauses, that prune all these programs. Thereupon Neo continues by searching for the next partially constructed program, until a full program that passes the property check has been found.

Neo, like close to all other SyGuS systems, only reasons with SMT-encoded properties and only guarantees that a returned program is a solution with respect to these properties, not with respect to the actual concrete semantics of the program. In contrast, `Popper`$[\mathcal{DL}]$ makes use of properties of background predicates if they are provided but works fine if they are elided as well. Additionally `Popper`$[\mathcal{DL}]$ guarantees correctness with respect to the actual semantics by maintaining the presence of the *Horn*-solver test stage. Neo's conflict analysis stage is similar to `Popper`'s constrain stage, though the constrain stage does not need to reason with an SMT-solver to determine the class of programs that can be pruned. In terms of our `Popper`$[\mathcal{DL}]$ implementation, where the generate stage's SAT-solver is directly connected to a theory solver that checks whether Difference Logic-encoded properties are still satisfiable, we lack this generalization step when a property-based conflict is identified. Neo cannot synthesise recursive programs, nor is it guaranteed to synthesise optimal (textually minimal) programs.

2.5.2 Related systems and notions within ILP

To our knowledge, besides the master's dissertation [112] of the author of this DPhil dissertation, there is no work in ILP that dynamically reasons about properties of partially constructed programs to soundly determine that they cannot be solutions. In addition to this work, we survey how the notion of user-provided constraints relates to our user-provided over-approximating properties, and another recent extension of `Popper` that automatically infers sound hypothesis constraints from the BK.

Refinement Type Directed Search for Meta-Interpretive Learning of Higher-Order Logic Programs

The most related work in ILP is the master's dissertation of the current author [112], which introduced polymorphic and refinement type checking to the Meta-Interpretive Learning framework [120]. The idea is to take a version of Metagol [39] (that learns higher-order logic programs [37]) and have it perform type checking along the way as it is building up programs. Like our over-approximating properties, the refinement types considered are directly inspired by their usage in functional program synthesis. BK predicates are annotated with *two* formulas, one expressing a polymorphic type and the other expressing a property. By extending Metagol's meta-interpreter, when a decision for a new predicate in the body of a clause is made, this forces a unification check for the polymorphic type

associated to this predicate with the types already associated with the variables that now occur as arguments to this predicate. Upon a type check failing, the search backtracks thereby eliding unnecessary evaluation. For the refinement types, the idea is to accumulate all the property formulas for the predicates that occur in a partially constructed hypothesis and each time the current partially constructed program is expanded, all the collected property formulas as a whole are handed to the Z3 solver to determine satisfiability. In case this check fails, the decision for the last selected predicate is backtracked. The master's thesis shows that there are scenarios where polymorphic type checking can have significant impact. For the refinement types, the main conclusion was that they can potentially be beneficial, but the implementation considered was very slow.

Clearly, the above work directly inspired the development of the work in Chapter 6. Contrary to this earlier work, we provide an entire framework to reason about the soundness of introducing reasoning with properties, including those of the target predicate. A major improvement is our adoption of the SMT-framework, where the theory solver responsible for checking the property formulas does so in an incremental fashion, i.e. only needing to do work on incremental guesses and not needing to re-build state for earlier property formulas that remain unchanged from the last check. Finally, this earlier approach inherits the issues of the Meta-Interpretative Learning approach, namely needing the strong inductive bias of metarules and that Metagol has search redundancy problems, which are especially problematic upon the introduction of expensive property checking.

Learning Logic Programs by Learning Where Not To Search

The other relevant ILP system is another recent extension (2023) of `Popper`, DiscoPopper [34], which infers hypothesis constraints by analysing the user-provided Background Knowledge. DiscoPopper requires that the BK is essentially a finite set of facts. Before learning starts, a set of seven templates is tested against the BK, for all possible substitutions of predicates in these templates. Each template corresponds to a semantic property that can be expressed syntactically. For example, given the "asymmetry" template $p(A, B) \rightarrow \neg p(B, A)$, for each predicate p defined by the BK, a check is performed if this implication holds for all possible substitutions of A and B. If this is the case for a predicate q, the constraint $\leftarrow q(A, B), q(B, A)$ is derived, signifying that no (optimal) solution can have both of these atoms occur in a clause.

Hence DiscoPopper analyses the BK before learning starts, yielding constraints that encode the exact opposite of metarules: a clause should be pruned when it *does* conform to the template. DiscoPopper's approach does not extend to properties that do not correspond to a simple syntactic template, whereas we provide a generic framework for reasoning about properties of predicates, including properties of the target predicate. Where DiscoPopper needs the BK to be available for inspection, to `Popper`$[\mathcal{DL}]$ the BK itself can be entirely opaque. While DiscoPopper only needs templates of possible properties, and then infers the properties themselves automatically, our approach assumes properties are provided by the user. DiscoPopper's approach does not extend to infinite domains, such as list transformations which are our primary motivation.

User-provided constraints

Finally, the over-approximating properties that our users are asked to provide can be seen as a form of *user-specified* constraints. A number of ILP systems allow a user to constrain

the hypothesis space through clause constraints [118, 150, 16, 1, 95]. All these systems include some mechanism to scrutinise candidate clauses and reject them on arbitrary grounds. In contrast, when a user provides an over-approximating property, they are essentially associating an alternative semantics to the predicates which have this property, in the vein of abstract interpretation [27]. Whereas user-provided constraints adjust the hypothesis space upfront, the reasoning with over-approximating properties prunes away those programs in the hypothesis space that could not be solutions (assuming the provided properties are actually over-approximating) during the search.

Chapter 3

Preliminaries

This chapter sets out the existing logical concepts and definitions which form the formal basis for work presented in the subsequent chapters. This chapter is meant as a reference – on a first pass through this document it can be skipped[1].

We define the syntax and semantics of the logic fragments that we will repeatedly refer to in this dissertation. Next we provide an introduction to how the Satisfiability Modulo Theories paradigm addresses finding models of formulas which use symbols from more than one fragment. Finally, we provide a brief overview of the relevant logic programming languages.

3.1 Syntax & semantics

The following lays out a fairly standard formalisation of many-sorted first-order logic, based on [12]. We use many-sorted logic as we will be combining formulas from different logic fragments. We use the sorts to track which symbols belong to which logic fragment. In the context of Satisfiability Modulo Theories this helps determine which solver is responsible for reasoning about the satisfiability of which sub-formulas.

3.1.1 Vocabulary & formulas

In specifying our logic fragments, we first need to establish which symbols can occur within each fragment. To do this, we need to provide a *vocabulary* for each fragment.

Definition 3.1.1. A tuple $\Sigma = (\Sigma_\sigma, \Sigma_V, \Sigma_f, \Sigma_p, \Sigma_L)$ is a *vocabulary* when the following are sets of symbols for which we have a function σ assigning sorts

1. Σ_σ are *sort symbols* – without an associated arity – such that $\mathcal{B} \in \Sigma_\sigma$.
2. Σ_V are *variable symbols* s.t. for each $V \in \Sigma_V$ we have $\sigma(V) \in \Sigma_\sigma$.
3. Σ_f are *function symbols* s.t. for each $f \in \Sigma_f$ we have $\sigma(f) = \sigma_1 \times \ldots \times \sigma_n \to \sigma_0$.
4. Σ_p are *predicate symbols* s.t. for each $p \in \Sigma_p$ we have $\sigma(p) = \sigma_1 \times \ldots \times \sigma_n \to \mathcal{B}$.
5. Σ_L are *logical symbols*, typically a subset of $\{ \top, \bot, \neg, \wedge, \vee, \to, \forall, \exists \}$.

where $\sigma_0, \sigma_1, \ldots, \sigma_n \in \Sigma_\sigma$ in the foregoing.

[1] We note here that when dealing with formulas which combine symbols from different logic fragments, we will often highlight that the symbols come from different fragments by colouring them according to their fragment.

When we refer to *constant*s or constant symbols, we mean arity-zero function symbols. Instead of $f \in \Sigma_f$ and $\sigma(f) = \sigma_1 \times \ldots \times \sigma_n \to \sigma_o$ or $p \in \Sigma_p$ and $\sigma(p) = \sigma_1 \times \ldots \times \sigma_n \to \mathcal{B}$, we will write $f : \sigma_1 \times \ldots \times \sigma_n \to \sigma_o$ and $p : \sigma_1 \times \ldots \times \sigma_n \to \mathcal{B}$.

Having defined the symbols that are part of our logic fragment, we can start building *terms*. In constructing terms, we make use of the sorts associated to symbols to only derive intended terms (e.g. if $1 : \text{int}$, $\top : \mathcal{B}$ and $\geq : \text{int} \times \text{int} \to \mathcal{B}$, then $1 \geq 1$ is an intended term but $\top \geq 1$ is not).

Definition 3.1.2. Given a vocabulary Σ, the Σ-terms T_Σ are inductively defined by:

1. if $V \in \Sigma_V$ then $V : \sigma(V) \in T_\Sigma$.
2. if $f : \sigma_1 \times \ldots \times \sigma_n \to \sigma_0 \in \Sigma_f$ and $t_1 : \sigma_1 \ldots, t_n : \sigma_n \in T_\Sigma$ then $f(t_1, \ldots, t_n) : \sigma_0 \in T_\Sigma$.

Almost always we will refer to terms *without* also referring to their associated sorts.

Having defined both our symbols and our terms, we can combine them to derive all possible formulas.

Definition 3.1.3. Given a vocabulary Σ (and optionally the set $\mathcal{F}^*_\Sigma$), the Σ-formulas $\mathcal{F}_\Sigma$ are inductively defined by:

1. if $\top \in \Sigma_L$ then $\top \in \mathcal{F}_\Sigma$.
2. if $\bot \in \Sigma_L$ then $\bot \in \mathcal{F}_\Sigma$.
3. if $V \in \Sigma_V$ and $\sigma(V) = \mathcal{B}$ then $V \in \mathcal{F}_\Sigma$.
4. if $p : \sigma_1 \times \ldots \times \sigma_n \to \mathcal{B} \in \Sigma_p$ and $t_1 : \sigma_1, \ldots, t_n : \sigma_n \in T_\Sigma$ then $p(t_1, \ldots, t_n) \in \mathcal{F}_\Sigma$.
5. if $\neg \in \Sigma_L$ and $F_1 \in \mathcal{F}_\Sigma$ then $\neg F_1 \in \mathcal{F}_\Sigma$.
6. if $\wedge \in \Sigma_L$ and $F_1, F_2 \in \mathcal{F}_\Sigma$, then $F_1 \wedge F_2 \in \mathcal{F}_\Sigma$.
7. if $\vee \in \Sigma_L$ and $F_1, F_2 \in \mathcal{F}_\Sigma$, then $F_1 \vee F_2 \in \mathcal{F}_\Sigma$.
8. if $\to \in \Sigma_L$ and $F_1, F_2 \in \mathcal{F}_\Sigma$, then $F_1 \to F_2 \in \mathcal{F}_\Sigma$.
9. if $\forall \in \Sigma_L$, $V \in \Sigma_V$ and $F \in \mathcal{F}_\Sigma$, then $\forall V.F \in \mathcal{F}_\Sigma$.
10. if $\exists \in \Sigma_L$, $V \in \Sigma_V$ and $F \in \mathcal{F}_\Sigma$, then $\exists V.F \in \mathcal{F}_\Sigma$.
11. if $F \in \mathcal{F}^*_\Sigma$ then $F \in \mathcal{F}_\Sigma$, given that a set $\mathcal{F}^*_\Sigma$ is provided.

The formulas constructed by rule 4 we refer to as the (Σ-)*atoms*. We refer to atoms and their negations (rule 5 after rule 4) as *literals*. Formulas which are composed of just disjunctions of literals we will often refer to as *clauses*.

A formula $F \in \mathcal{F}_\Sigma$ has a *free variable* V when there is no sub-formula $\forall V.F_1$ or $\exists V.F_1$ of F such that F_1 contains the occurrance of V. A formula $F \in \mathcal{F}_\Sigma$ is *closed* if it contains no free variables. In the context of first-order clausal logic, such as Horn logic, we will often implicitly assume these clauses to be closed off by quantifiers and hence omit them when we write out the clauses.

The option to provide the set $\mathcal{F}^*_\Sigma$ will later be used as a mechanism for combining logic fragments.

3.1.2 Interpretations & valuation

Having defined how we define the syntax of our logic fragments, we now prescribe the way we associate interpretations to this syntax.

Definition 3.1.4. Given a vocabulary Σ, $I = (\mathcal{D}_I = \{\, \mathcal{D}_\sigma \mid \sigma \in \Sigma_\sigma \,\}, -^I)$ defines a Σ-*interpretation* when $-^I$ is a (partial) mapping such that

1. for each $\sigma \in \Sigma_\sigma$, with each $\mathcal{D}_\sigma$ a set, it holds $\sigma^I \in \mathcal{D}_I$.
   with that for $\mathcal{B} \in \Sigma_\sigma$ we have $\mathcal{B}^I = \{ \top, \bot \}$.
2. for each $V \in \Sigma_V$, either $V^I \in \sigma(V)^I$
   or $-^I$ is undefined on V, which we denote as $V^I = ??$.
3. for each $f : \sigma_1 \times \ldots \times \sigma_n \to \sigma_0 \in \Sigma_f$ we have $f^I : \sigma_1^I \times \ldots \times \sigma_n^I \to \sigma_0^I$.
4. for each $p : \sigma_1 \times \ldots \times \sigma_n \to \mathcal{B} \in \Sigma_p$ we have $p^I : \sigma_1^I \times \ldots \times \sigma_n^I \to \{ \top, \bot \}$.

$\mathcal{D}_I$ is a collection of *domains* where $\mathcal{D}_\sigma = \sigma^I$ is the sort σ's *domain*, according to I. For our purposes, variables are the only inputs on which an interpretation is allowed to be partial. With respect to a formula $F \in \mathcal{F}_\Sigma$, a Σ-interpretation I is a *partial assignment* when F contains a free variable V such that $V^I = ??$, i.e. is not defined on all free variables of F. An interpretation I can be *extended* on variable V by the notation $I\{ V \mapsto v \}$, such that $V^{I\{ V \mapsto v \}} = v$. When I is defined on all free variables of F, I is a *full assignment* for F. At times we use the abbreviations *full-* and *partial interpretations*.

We now evaluate formulas at interpretations to determine valuations, i.e. truth values, of said formulas.

Definition 3.1.5. Given a vocabulary Σ, a formula $F \in \mathcal{F}_\Sigma$ and a Σ-interpretation I, the *valuation of F at I* is inductively defined by:

1. $[\![\top]\!]_I = \top$
2. $[\![\bot]\!]_I = \bot$
3. $[\![V]\!]_I = V^I$ if $\sigma(V) = \mathcal{B}$ and $-^I$ is defined on V
   $??$ if $\sigma(V) = \mathcal{B}$ and $-^I$ is undefined on V
4. $[\![\neg F_1]\!]_I = \top$ if $[\![F_1]\!]_I = \bot$
   $\bot$ if $[\![F_1]\!]_I = \top$
5. $[\![p(t_1, \ldots, t_n)]\!]_I = p^I(t_1^I, \ldots, t_n^I)$ if $-^I$ defined for all (variable-)terms in $t_1, \ldots, t_n$
   $??$ if there is a variable v in any t_i such that $v^I = ??$
6. $[\![F_1 \wedge F_2]\!]_I = [\![F_1]\!]_I \wedge [\![F_2]\!]_I$
7. $[\![F_1 \vee F_2]\!]_I = [\![F_1]\!]_I \vee [\![F_2]\!]_I$
8. $[\![F_1 \to F_2]\!]_I = [\![(\neg F_1) \vee F_2]\!]_I$
9. $[\![\forall V.F_1]\!]_I = \top$ if $[\![F_1]\!]_{I\{V \mapsto v\}} = \top$ for every $v \in \sigma(V)^I$
   $\bot$ if $[\![F_1]\!]_{I\{V \mapsto v\}} = \bot$ for any $v \in \sigma(V)^I$
10. $[\![\exists V.F_1]\!]_I = \top$ if $[\![F_1]\!]_{I\{V \mapsto v\}} = \top$ for any $v \in \sigma(V)^I$
    $\bot$ if $[\![F_1]\!]_{I\{V \mapsto v\}} = \bot$ for every $v \in \sigma(V)^I$

Note that only on full assignments/interpretations, we are guaranteed to be able to derive a valuation (assuming we can deal with infinite domains for variables). An assignment/interpretation being partial, i.e. mapping a variable to $??$, means that we are not guaranteed a $\top/\bot$ valuation of the formula: in case of disjunction only one disjunct needs to hold to derive $\top$ and for a conjunction only one conjunct needs to be $\bot$. However, when a variable is uninterpreted and necessary to derive a valuation, we say the partial interpretation did not produce a valuation, in which case we derive $??$.

We extend the notion of full and partial assignments as well as valuations to sets of formulas by treating the set as a conjunction of the formulas.

3.1.3 Important relations on formulas and interpretations

We define a number of relations on formulas and interpretations that we refer to often in this document.

Definition 3.1.6. Given a vocabulary Σ and a formula $F \in \mathcal{F}_\Sigma$, a Σ-interpretation I is a *model of* F if and only if $[\![F]\!]_I = \top$.

Definition 3.1.7. Given a vocabulary Σ and a formula $F \in \mathcal{F}_\Sigma$, we denote the set of *intended* Σ*-interpretations* by $\mathcal{I}(F)$.

We denote the set of intended Σ-interpretations *which are full assignments* by $\mathcal{A}(F)$.

Definition 3.1.8. Given a vocabulary Σ, a formula $F \in \mathcal{F}_\Sigma$ is *satisfiable* if there exists an (intended) Σ-interpretation $I \in \mathcal{A}(F)$ which is a *model* of F.

For any Σ and formula $F \in \mathcal{F}_\Sigma$, the set of all models we denote by $\mathcal{M}(F)$. Clearly, $\mathcal{M}(F) \subseteq \mathcal{A}(F) \subseteq \mathcal{I}(F)$. We say a Σ-formula F is *unsatisfiable* – *unsat* for short — if it is not satisfiable, i.e. no (intended) interpretation is a model of F: $\mathcal{M}(F) = \emptyset$.

Definition 3.1.9. Given a vocabulary Σ, a formula $F \in \mathcal{F}_\Sigma$ is *valid* if each (intended) Σ-interpretation $I \in \mathcal{A}(F)$ is a *model* of F – in which case $\mathcal{A}(F) = \mathcal{M}(F)$.

Definition 3.1.10. Given a vocabulary Σ and formulas $F, G \in \mathcal{F}_\Sigma$, G is a *consequence* of F if every Σ-interpretation I that is a model of F is also a model of G, which we denote as $F \models G$.

When G is a consequence of F, we also say that F *entails* G.

Definition 3.1.11. Given a vocabulary Σ, a Σ-*theory* is a set of Σ-interpretations.

A (Σ-)theory can either be explicitly fixed, as we will do for Difference Logic and Linear Integer Arithmetic, or can be specified as the models of a (set of) formula(s), as is common for Horn clauses. Given a Σ-theory $\mathcal{T}$, we can talk about satisfiability and validity of formulas with regard to the interpretations that make up $\mathcal{T}$.

Definition 3.1.12. Given a set of Σ-formulas Fs that is unsatisfiable, an *unsat-core* is a set of formulas $Fs' \subseteq Fs$ which is also unsatisfiable.

Given Fs, an unsat-core Fs' is minimal when removing any formula of Fs' yields a satisfiable set of formulas.

3.2 Fragments & intended interpretations

We now use the above definitions to define fragments of first-order logic relevant to this thesis.

Definition 3.2.1. A *logic fragment* is a triple $(\Sigma, \mathcal{F}, \mathcal{T})$, consisting of a vocabulary Σ, a set of formulas, $\mathcal{F}$, constructed from the symbols of Σ, and a theory $\mathcal{T}$, which is a set of interpretations for the symbols in Σ.

In this dissertation, we will strictly use the term *theory* in this model-theoretic sense: as a set of interpretations. Hence a set of formulas can be thought of as a theory only in terms of the actual interpretations of the set of formulas itself.

3.2.1 Propositional logic

We introduce Propositional logic as (a degenerate) fragment of first-order logic. This is especially useful in the context of Satisfiability Modulo Theories, where Propositional logic is usually extended with formulas (now treated as atoms) from first-order logic fragments.

Definition 3.2.2. *Propositional logic*'s vocabulary is

$$\Sigma^{\mathcal{B}} = (\{\,\mathcal{B}\,\}, \{\,V_i \mid \sigma(V_i) = \mathcal{B} \text{ for } i = 1, 2, \dots\,\}, \emptyset, \emptyset, \{\neg, \wedge, \vee, \rightarrow\})$$

That is, there is the sole sort of Booleans for which we have variables and the typical connectives, but no function symbols and no predicate symbols. The formulas of propositional logic, $\mathcal{F}_{\mathcal{B}}$, are the formulas $\mathcal{F}_{\Sigma^{\mathcal{B}}}$, i.e. all formulas constructable from this vocabulary. The interpretations $\mathcal{I}_{\mathcal{B}}$ of $\Sigma^{\mathcal{B}}$ are the different assignments for the Boolean variables.

3.2.2 Difference logic

We will use Difference logic as our main example of a decidable fragment of first-order logic [123]. We will extend other logic fragments, such as Propositional logic, with formulas from Difference logic, in line with the Satisfiability Modulo Theories paradigm.

Definition 3.2.3. *Difference logic*'s vocabulary is

$$\Sigma^{DL} = (\{\,\mathcal{B}, \mathit{Int}\,\}, \{\,V_i \mid \sigma(V_i) = \mathit{Int} \text{ for } i = 1, 2, \dots\,\}, \{\,-, 0, 1, -1, \dots\,\}, \{<, \leq\}, \{\wedge, \vee, \neg\})$$

That is, difference logic has variables and constants of sort Int, as well as the predicate symbols $<$ and $\leq$, which, when applied to two terms, yield a Boolean-valuation.

The symbol for taking the difference is the only (non-zero arity) function.

Definition 3.2.4. The *difference logic-formulas* $\mathcal{F}_{DL} \subset \mathcal{F}_{\Sigma^{DL}}$ are the Σ^{DL}-formulas which consist of conjunctions, disjunctions and negations of atoms of the form $t_1 - t_2 < c$ and $t_1 - t_2 \leq c$, where t_1 and t_2 are either variables or integer constants and c is an integer constant.

That is, we limit the formulas of Difference logic to the Boolean combinations of constant-bounded differences of at most two variables.

Formally, we interpret each constant symbol representing an integer as an integer. Likewise, we interpret $<$, $\leq$ and $-$ according to their standard arithmetic interpretation. Variables need to be interpreted as integers as well:

Definition 3.2.5. A Σ_{DL}-interpretation $I = (\{\,\{\,\top, \bot\,\}, \{\,0, 1, -1, \dots\,\}\,\}, -^I)$ interprets $\mathcal{B}^I = \{\top, \bot\}$ and $\mathit{Int}^I = \{\,0, 1, -1, \dots\,\}$, has $-^I$ assign the integer constants to themselves, $(-)^I = \lambda x, y.x - y$, $<^I = \lambda x, y.x < y$, $\leq^I = \lambda x, y.x \leq y$, and for each variable V_i, V_i^I is either undefined or an integer in Int^I.

We hence define Difference logic's theory, i.e. set of (intended) interpretations, as follows:

Definition 3.2.6. The Σ^{DL}-theory $\mathcal{T}_{DL}$ consists of all Σ^{DL}-interpretations, i.e. all possible assignments of variables to integers and $-$, $<$ and $\leq$ with the standard arithmetic interpretation.

When we explicitly write down a $\Sigma^{\mathcal{DL}}$-interpretation, we will just write how the variables get interpreted (as the function and predicates must always be as specified above).

3.2.3 Horn logic

As we will soon see, Horn logic forms the basis of logic programming. The techniques in this dissertation seek to address learning sets of definite Horn formulas, also know of as definite clauses. We will refer to such sets of formulas as *programs*. We will give Horn logic a formal treatment and discuss logic programming itself later.

Syntax First we fix the vocabulary of Horn logic:

Definition 3.2.7. *Horn logic*'s vocabulary is

$$\Sigma^{Horn} = \begin{array}{l}(\{\mathcal{B}, *\}, \{V_i \mid \sigma(V_i) = * \text{ for } i = 1, 2, \dots\}, \\ \{f_i \text{ for } i = 1, 2, \dots\}, \{p_i \text{ for } i = 1, 2, \dots\}, \{\bot, \neg, \wedge, \vee, \rightarrow, \forall\})\end{array}$$

Due to the central importance of Herbrand interpretations [136], which essentially interpret syntax as itself, we introduce a sort $*$ that stands in for "terms". Each argument of a predicate p_i is of sort $*$, as is each argument of a function f_i, as well as f_i's return sort. Variables are also taken to range over terms, i.e. their sort is $*$ as well. Horn logic is the only logic we will treat which supports quantifiers, and only universals at that.

Definition 3.2.8. The *Horn formulas* $\mathcal{F}_{Horn} \subseteq \mathcal{F}_{\Sigma^{Horn}}$ are the closed formulas of $\mathcal{F}_{\Sigma^{Horn}}$ such that either

1. they consist of a disjunction of literals with at most one non-negated atom, or
2. they have the form $a_1 \wedge \dots \wedge a_n \rightarrow l$ where $a_1, \dots, a_n$ are atoms and l is either an atom or $\bot$,

with these formulas nested inside universal quantifiers binding all contained variables.

The above two characterizations of Horn formulas are equivalent, per the equivalence $p_1 \wedge \dots \wedge p_n \rightarrow q \equiv \neg p_1 \vee \dots \vee \neg p_n \vee q$.

A set of formulas $Hs \subseteq \mathcal{F}_{Horn}$ is *function-free* if no function symbol occurs in Hs. A set of Horn-formulas Hs is *Datalog* if there occurs no non-zero arity function symbol in Hs. A set of Horn-formulas Hs is *definite* if there is exactly one non-negative atom in each formula, or, equivalently, no $\bot$ occurs as a consequent of an implication.

At times we will interchange – even implicitly – between the perspective of a Horn formula being an disjunction or an implication. We will often refer to the individual Horn formulas as *clauses* – per the disjunctive-perspective – and *rules* – per the implication-perspective. In case of Datalog Horn- and definite Horn formulas, we will often drop the Horn qualifier.

Semantics We now turn to how to interpret Horn formulas. Horn formulas enjoy the property that if there is any interpretation that is a model, then there is a Herbrand interpretation that is a model [136]. Hence, as is usual, we restrict ourselves to Herbrand interpretations which we now define.

Definition 3.2.9. Given a set of (non-function free) formulas $Hs \subseteq \mathcal{F}_{Horn}$, the *Herbrand universe* $HU(Hs)$ is all terms inductively constructable from solely the function symbols occurring in Hs.

The Herbrand universe is said to consist of all *ground* terms, i.e. all the terms which do not contain variables. We use the set of all such terms to construct all possible ground atoms that are relevant to syntactic interpretations of Horn clauses.

Definition 3.2.10. Given a set of (non-function free) formulas $Hs \subseteq \mathcal{F}_{Horn}$, the *Herbrand base* $HB(Hs)$ is all atoms constructable from the predicate symbols in Hs applied to all (combinations of) Herbrand universe terms $HU(Hs)$.

Every subset of the Herbrand base corresponds to an interpretation:

Definition 3.2.11. Given a set of formulas $Hs \subseteq \mathcal{F}_{Horn}$ and a subset $HB' \subseteq HB(Hs)$, the corresponding *Herbrand interpretation* $I_{HB'} = (\{\{\top, \bot\}, HU(Hs)\}, -^{I_{HB'}})$ such that $\mathcal{B}^{I_{HB'}} = \{\top, \bot\}$, $*^{I_{HB'}} = HU(Hs)$ and $-^{I_{HB'}}$ maps ground terms to themselves and maps each predicate p occurring in Hs such that $[\![p(t_1, \ldots, t_n)]\!]_{I_{HB'}} = \top$ if and only if $p(t_1, \ldots, t_n) \in HB'$.

If a Herbrand interpretation I is a model of a set of formulas $Hs \subseteq \mathcal{F}_{Horn}$, then we call I a *Herbrand model*. The process of generating the Herbrand base – turning every formula in a set of quantified formulas into non-quantified formulas where the variables have been substituted with terms from the Herbrand universe – is often called *grounding*. The resulting set of *ground formulas* is essentially propositional (though, in general, is an infinite set).

Given a set of clauses Hs, we can hence see the intended theory as all those Herbrand interpretations that model Hs:

Definition 3.2.12. Given a set of Horn formulas Hs, the corresponding *Horn theory* is

$$\mathcal{T}_{Hs} = \{ I_{HB'} \mid HB' \subseteq HB(Hs) \wedge [\![HS]\!]_{I_{HB'}} = \top \}$$

We hence have that $\mathcal{T}_{Hs} = \mathcal{M}(Hs)$.

Besides the sufficiency of Herbrand interpretations, logic programming makes use of the following property to give meaning to Horn formulas when seen as programs:

Proposition 3.2.13. For any set of formulas $Hs \subseteq \mathcal{F}_{Horn}$, $I_{HB^\cap} \in \mathcal{M}(Hs)$ where

$$HB^\cap = \cap \{ HB' \mid I_{HB'} \in \mathcal{M}(Hs) \}.$$

That is, the intersection of all Herbrand base subsets that correspond to models (when interpreted as an interpretation) corresponds to a model as well. We denote this *least model* by $LM(Hs) = I_{HB^\cap}$. This model is equivalently the least fixed point model of Hs when Hs is seen as set of recursive definitions [15]. In logic programming, this model is seen as the *canonical* way of interpreting the clauses Hs. The following result characterises why:

Proposition 3.2.14. For any set of formulas $Hs \cup \{ F \} \subseteq \mathcal{F}_{Horn}$, F is a consequence of Hs if and only if $[\![F]\!]_{LM(Hs)} = \top$.

That is, all consequences of Hs can be determined just from its least model $LM(Hs)$. Hence we have that the only interpretation of real interest is the least model of Hs:

Definition 3.2.15. Given a set of Horn formulas Hs, $\mathcal{T}_{LM(Hs)} = \{ LM(Hs) \}$ is the corresponding *least Horn theory*, i.e. (the set just containing) Hs's least model.

We will use this set of interpretations in our Satisfiability Modulo Theories characterization of Learning From Failures.

3.2.4 Other decidable first-order fragments

Besides Difference Logic and Horn formulas with finite terms, there are many other decidable fragments of first-order logic. As we do not make use of them in this document, we elide their introduction. Instead we direct the interested reader to the tutorial by Bjørner and Nachmanson on the many theories (and their solvers) supported by the Z3 SMT-solver [14].

3.3 Satisfiability Modulo Theories

We now turn our focus to combining logic fragments. The idea is that we would like to use simple logics such as Propositional logic as much as possible, however certain constraints must be (or are more easily) expressed in (fragments of) first-order logic.

3.3.1 Syntax

For the purposes of this thesis, we assume that there is no overlap in the non-logical symbols of one logic fragment and another[2]. This makes it is easy to assign a single solver as being responsible for determining satisfiability of the relevant sub-formulas. With our non-overlapping assumption, combining vocabularies is simply taking the union:

Definition 3.3.1. Given vocabularies Σ^1 and Σ^2, vocabulary $\Sigma^{1,2}$ is given by

$$(\Sigma^{1,2}_\sigma = \Sigma^1_\sigma \cup \Sigma^2_\sigma, \Sigma^{1,2}_V = \Sigma^1_V \cup \Sigma^2_V, \Sigma^{1,2}_f = \Sigma^1_f \cup \Sigma^2_f, \Sigma^{1,2}_p = \Sigma^1_p \cup \Sigma^2_p, \Sigma^{1,2}_L = \Sigma^1_L \cup \Sigma^2_L)$$

Often we want to say that formulas from certain fragments occur as if they were atoms within a simpler fragment, such as Propositional logic. Given a set $\mathcal{F}_2$ of Σ_2-formulas, we can supply $\mathcal{F}_2$ as the optional formulas of Definition 3.1.3, i.e. set $\mathcal{F}^*_{\Sigma_1} = \mathcal{F}_2$, when building the formulas derivable from a vocabulary Σ_1.

Example 3.3.2. If we supply the set of Difference logic formulas $\mathcal{F}_{\mathcal{DL}}$ while building the set of formulas from Propositional logic's symbols, $\Sigma^{\mathcal{B}}$, we obtain a set of formulas over vocabulary $\Sigma^{\mathcal{B},\mathcal{DL}}$, where the Difference logic formulas occur exactly in place of propositional variables. An example of such a formula from this set is

$$F = (p \vee q) \wedge (\neg p \rightarrow B > A) \wedge (p \rightarrow B > C) \wedge (q \rightarrow (A > B \wedge C > A))$$

In this formula, p and q are propositional variables while the atoms marked red come from the Difference logic fragment.

3.3.2 Semantics

We now turn to how to interpret formulas which contain sub-formulas that we identify as coming from distinct logic fragments. Given that we assume each non-logical symbol in our formula to come from just one fragment, we can easily compose the interpretations for the separate fragments into a single combined interpretation for the whole formula:

[2] In general, this does not have to be the case. See Nelson-Oppen theory combination [87] for how this more general setting can be handled.

Definition 3.3.3. Given Σ^1-interpretation I and Σ^2-interpretation J, $\Sigma^{1,2}$-interpretation $I,J = (\mathcal{D}_{I,J}, -^{I,J})$ is given by $\mathcal{D}_{I,J} = \mathcal{D}_I \cup \mathcal{D}_J$ and $x^{I,J} = x^I$ if x is a symbol from Σ_1 else $x^{I,J} = x^J$.

As we assumed symbols from distinct fragments are distinct, we assume that the domains for the sorts in I and J are distinct as well (except for that $\mathcal{B}^I = \mathcal{B}^J = \{\top, \bot\}$, which is forced by condition 1 of Definition 3.1.4). Clearly, if each interpretation I and J is a full interpretation for the variables from Σ^1 and Σ^2 of a $\Sigma^{1,2}$-formula F, respectively, then I,J is a full interpretation of F.

Proposition 3.3.4. When all symbols of Σ^1 and Σ^2 are distinct, any $\Sigma^{1,2}$-interpretation K can always be split into a Σ^1-interpretation I and Σ^2-interpretation J such that $K = I,J$.

In our simple SMT-setting, we therefore have that if there is a satisfying interpretation of a $\Sigma^{1,2}$-formula then this interpretation is always characterized by two completely separate interpretations and hence the task of finding these separate interpretations can be handed-off to separate solvers.

Valuation of a formula $F \in \mathcal{F}_{\Sigma^{1,2}}$ at I,J happens in accord with Definition 3.1.5.

Example 3.3.5. Suppose we have $\Sigma^{\mathcal{B},\mathcal{DL}}$-formula F from Example 3.3.2. This formula has both propositional variables p and q, i.e. $\Sigma^{\mathcal{B}}$-symbols, and first-order variables A, B, and C, i.e. $\Sigma^{\mathcal{DL}}$-symbols, that need to be interpreted. If we specify $\Sigma^{\mathcal{B}}$-interpretation $I = \{p \mapsto \top, q \mapsto \bot\}$ and $\Sigma^{\mathcal{DL}}$-interpretation $J = \{A \mapsto 3, B \mapsto 2, C \mapsto 1\}$, then I,J is a full interpretation of F and $[\![F]\!]_{I,J} = \top$, i.e. I,J is a model of F.

If $I = \{p \mapsto \bot, q \mapsto \bot\}$ then there is no J such that $[\![F]\!]_{I,J} = \top$ as the first clause, $(p \vee q)$, is not satisfied irrespective of the choice of interpretation J.

3.3.3 Sub-formulas at interpretations

We now look at what we can determine about a $\Sigma^{1,2}$-formula F given that we only have decided on (part of) a Σ^1-interpretation. Intuitively, when certain variables of F already have their valuation fixed then some sub-formulas of F, e.g. clauses, have already been determined as well, either in the sense of already having a valuation or that a sub-formula from the other fragment is now forced to hold. The idea is that based on the Σ^1-interpretation we can figure out which Σ^2-sub-formulas of F must hold together, at which point we can reason about the satisfiability of just these specific sub-formulas and just with respect to Σ^2-interpretations.

The following definition helps us to reduce a $\Sigma^{1,2,3}$-formula to a Σ^2-formula based on a Σ^1-interpretation.

Definition 3.3.6. Given a formula $F \in \mathcal{F}_{\Sigma^{1,2,3}}$ such that Σ^1 does not have quantifiers and a Σ^1-interpretation I, the Σ^2-*interpretation of* F *at* I is inductively defined by:

1. $\langle\!\langle F \rangle\!\rangle_I^{\Sigma^2} = F$ if F is a Σ^2-formula
2. $\langle\!\langle F \rangle\!\rangle_I^{\Sigma^2} = ??$ if F is a Σ^3-formula
3. $\langle\!\langle \top \rangle\!\rangle_I^{\Sigma^2} = \top$
4. $\langle\!\langle \bot \rangle\!\rangle_I^{\Sigma^2} = \bot$
5. $\langle\!\langle V \rangle\!\rangle_I^{\Sigma^2} = V^I$ – i.e. $\sigma(V) = \mathcal{B}$ – can be ??

6. $\langle\!\langle \neg F_1 \rangle\!\rangle_I^{\Sigma^2} = \neg \langle\!\langle F_1 \rangle\!\rangle_I^{\Sigma^2}$.
7. $\langle\!\langle p(t_1, \ldots, t_n) \rangle\!\rangle_I^{\Sigma^2} = p^I(t_1^I, \ldots, t_n^I)$ if $-^I$ defined for all (variable-)terms in $t_1, \ldots, t_n$
   $??$ if there is a variable in any t_i such that $t_i^I = ??$
8. $\langle\!\langle F_1 \wedge F_2 \rangle\!\rangle_I^{\Sigma^2} = \langle\!\langle F_1 \rangle\!\rangle_I^{\Sigma^2} \wedge \langle\!\langle F_2 \rangle\!\rangle_I^{\Sigma^2}$
9. $\langle\!\langle F_1 \vee F_2 \rangle\!\rangle_I^{\Sigma^2} = \langle\!\langle F_1 \rangle\!\rangle_I^{\Sigma^2} \vee \langle\!\langle F_2 \rangle\!\rangle_I^{\Sigma^2}$
10. $\langle\!\langle F_1 \rightarrow F_2 \rangle\!\rangle_I^{\Sigma^2} = \neg \langle\!\langle F_1 \rangle\!\rangle_I^{\Sigma^2} \vee \langle\!\langle F_2 \rangle\!\rangle_I^{\Sigma^2}$

Hence $\langle\!\langle F \rangle\!\rangle_I^{\Sigma^2}$ gives us back a formula consisting of Σ^2-symbols and ??s. The ??-symbols represent the under-determined portions of the formula. Nonetheless, even with these under-determined parts of the formula, we can know which Σ^2-sub-formulas must necessarily hold.

We rid ourselves of the ??-symbols by using the rules $\neg ?? \implies ??$, $?? \wedge F_1 \implies F_1$ and $?? \vee F_1 \implies ??$ (and $F_1 \wedge ?? \implies F_1$ and $F_1 \vee ?? \implies ??$). Clearly, if we rewrite $\langle\!\langle F \rangle\!\rangle_I^{\Sigma^2}$ to conjunctive normal form and apply these simplification rules, we obtain a conjunctive-normal-form Σ^2-formula (that is possibly just $\top$). In effect, our simplification rules turn a ternary logic formula involving ?? back into a formula which can be given a (Boolean-)valuation and which must necessarily hold for there to be any satisfying interpretation I',J of F where I' extends I.

Example 3.3.7. Let us again consider formula F from Example 3.3.2. If we have $\Sigma^{\mathcal{B}}$-interpretation $I = \{ p \mapsto \bot, q \mapsto \top \}$ then $\langle\!\langle F \rangle\!\rangle_I^{\mathcal{DL}} = B > A \wedge A > B \wedge C > A$. Now, as there is no assignment of integers for A, B, and C such that $B > A \wedge A > B$ holds (as there are no integers n and m to substitute for A and B such that both $n > m$ and $m > n$), there is no $\Sigma^{\mathcal{DL}}$-interpretation J such that $[\![F]\!]_{I,J} = \top$.

Now, if I is a partial interpretation, we need to use the simplification rules to obtain a Σ^2-formula implied by the Σ^1-interpretation:

Example 3.3.8. As before, we use Example 3.3.2's formula F. If we take a partial interpretation $I = \{ q \mapsto \top \}$ (so that $p^I = ??$), then $\langle\!\langle F \rangle\!\rangle_I = \top \wedge ?? \wedge ?? \wedge (A > B \wedge C > A)$. We have that the first clause can (lazily) be evaluated to $\top$ while the fourth clauses reduces to $(A > B \wedge C > A)$ and hence must hold. As for the second clause of F: $\langle\!\langle \neg p \rightarrow B > A \rangle\!\rangle_I^{\mathcal{DL}} = \langle\!\langle p \vee B > A \rangle\!\rangle_I^{\mathcal{DL}} = ?? \vee B > A$. We simplify this to ?? as $p \vee B > A$ could be satisfied in two distinct ways: either I gets extended so that $p^I = \top$ or J gets chosen such that $B > A$ is satisfied. The third clause simplifies to ?? for the same reason.

In conclusion, for this I, we simplify $\langle\!\langle F \rangle\!\rangle_I$ to $A > B \wedge C > A$. This is justified as for there to exist any interpretation I',J' that models F – with I' extending I – there must exist an interpretation J (which can then be extended to become J') such that both $A > B$ and $C > A$ hold. If this is not the case, there is now way for interpretation I to be extended to a model I',J'.

In short, $\langle\!\langle F \rangle\!\rangle_I^{\Sigma^2}$ is the combination of Σ^2-sub-formulas of F that must necessarily hold given I.

3.3.4 Solving SMT-problems

With the necessary definitions in place, we can now discuss *solving* problems posed as formulas with symbols from multiple fragments. We are in the setting of determining

satisfiability. Hence, given a formula F with symbols from multiple fragments, the task of an SMT-solver is to find a combined interpretation that is a model of F.

Typically, one of the fragments is (essentially) propositional, like Propositional logic (or propositional[3] Answer Set Programs, as we will use in our implementation in Section 4.5). For solving this propositional fragment, an SMT-solver can rely on the many advances in SAT-solvers over the years. The other symbols are from first-order fragments. While there are techniques for reducing formulas from some first-order fragments to (finite) propositional formulas [86], we will focus on the more general approach[4] by which each such fragment has its own specialised solver [12].

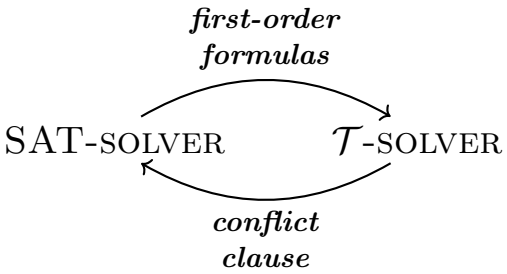


Figure 3.1: The generate, test, and constrain loop.

An SMT-solver is in essence some scaffolding on top of a SAT-solver for propositional logic so that, based on the propositional assignments, the SAT-solver appropriately communicates with the specialised solvers for first-order fragments. In a sense, the propositional fragment is primary, in that an SMT-solver primarily builds up (and backtracks) assignments for propositional variables, and based on these assignments derives which first-order formulas must hold. That is, while the SAT-solver modifies its propositional assignment for F, i.e. a $\Sigma^{\mathcal{B}}$-interpretation I, a solver for Σ^2-symbols is kept up-to-date with the relevant formulas $\langle\!\langle F \rangle\!\rangle_I^{\Sigma^2}$ as I changes. In particular, when I is a full assignment, satisfying all purely propositional constraints of F, it is the solver for Σ^2-symbols that determines if there is a (full) interpretation J for these symbols. Only when there is both a full $\Sigma^{\mathcal{B}}$-interpretation I (satisfying the propositional constraints), and a full Σ^2-interpretation J satisfying $\langle\!\langle F \rangle\!\rangle_I^{\Sigma^2}$ has a model of F been found, i.e. I,J. As the interpretation J must come from a specific first-order theory, the solver for the first-order fragments are often called *theory solvers*. Figure 3.1 illustrates how the propositional SAT-solver and a solver for determining satisfiability with respect to a theory $\mathcal{T}$ communicate.

When a guess for a (partial) assignment is shown to *not* be extendable to a model, a *conflict* clause can be learned banning this assignment (and extensions of it) from being considered again. This can happen at the propositional level, in which case we speak of Conflict-Driven Clause Learning [105], where a clause is learned consisting of the literals that are responsible for the conflict. When a theory solver detects $\langle\!\langle F \rangle\!\rangle_I^{\Sigma^2}$ does not have any satisfying interpretations, the theory solver has detected a conflict. At this point

[3]We will obtain propositional answer set programs through grounding, see Section 3.4

[4]This reduction approach is known as the "eager"-approach to SMT-solving – as the SAT-solver is always reasoning about the whole (grounded) first-order formula – while having separate solvers is described as the "lazy" approach – as the separate solvers only need to deal with the sub-formulas currently enabled due to a (partial) propositional assignment.

the theory solver is responsible for determining which first-order sub-formulas caused the conflict and injects a *conflict clause* into the SAT-solver. The SAT-solver treats the first-order literals in such clauses just like any other propositional literals. These learned conflict clauses hence help to elide (the computational cost of) invoking the theory solver again.

Example 3.3.9. Let us reconsider the $\Sigma^{\mathcal{B},\mathcal{DL}}$-formula F from Example 3.3.2:

$$F = (p \vee q) \wedge (\neg p \rightarrow B > A) \wedge (p \rightarrow B > C) \wedge (q \rightarrow (A > B \wedge C > A))$$

A SMT-solver goes about determining the satisfiability of F by gradually building up a propositional assignment for variables p and q. As the SAT-solver guesses different interpretations I for p and q, the $\mathcal{DL}$-solver is kept informed of $\langle\!\langle F \rangle\!\rangle_I^{\mathcal{DL}}$, i.e. the $\mathcal{DL}$-formulas which must hold given I.

The following table shows a sequence of guesses for assignments of p and q. The SAT-solver's variable ordering is p before q, hence we see rows where p has a truth value but q is still undetermined.

	p^I	q^I	$\langle\!\langle F \rangle\!\rangle_I^{\mathcal{DL}}$	SAT-solver	$\mathcal{T}_{DL}$-solver
1.	$\bot$	??	$B > A$	-	-
2.	$\bot$	$\bot$	$B > A$	$\lightning$	-
3.	$\bot$	$\top$	$B > A \wedge A > B \wedge C > A$	✓	$\lightning$
4.	$\top$	??	$B > C$	-	-
5.	$\top$	$\top$	$B > C \wedge A > B \wedge C > A$	✓	$\lightning$
6.	$\top$	$\bot$	$B > C$	✓	✓

In the first row $p^I = \bot$ which means that there is still a way to satisfy the first (propositional) clause of F. The second clause forces $B > A$ to hold. Hence both the SAT-solver and solver for theory $\mathcal{T}_{DL}$ do not have a full assignment yet and neither can either already establish unsatisfiability.

Next, the SAT-solver decides on the truth value of q. The second row shows that if the guess for q were false, this would violate the first clause. Hence the SAT-solver determines on its own that the guess[5] where $p^I = \bot$ and $q^I = \bot$ cannot be extended to a model, without needing to consult the $\mathcal{T}_{DL}$-solver.

Having tried $\bot$ for q, the SAT-solver backtracks and tries the remaining option for q: $\top$. The purely propositional constraint of the first clause is now satisfied. We have that the second and fourth clauses now assert that $\mathcal{DL}$-formulas must hold. As explained in Example 3.3.7, these formulas are not satisfiable together and hence the $\mathcal{T}_{DL}$-solver injects a conflict clause: as only $A < B$ and $B < A$ are in conflict with each other this clause is, in effect, $(A < B \wedge B < A) \rightarrow \bot$.

Having ruled out all possible assignments I with $p^I = \bot$, the SAT-solver backtracks and tries the only other option for p, i.e. $p^I = \top$. By the second clause, $B > C$ must now hold. Neither the SAT-solver nor the theory solver finds any satisfiability issues with this.

The fifth row shows the next guess where both p and q are true. Again, F's propositional clause is satisfied, meaning the SAT-solver is not able to reject the guess. This

[5] In practice, this guess will not actually occur as SAT-solvers will use a rule called *unit-resolution* to say that as q is the last undetermined literal in a clause, its assignment is forced. That is, the first clause is read as saying "if not p, then no need to guess q, as q now must be $\top$.

full propositional assignment leads to a new three-way conjunction of $\mathcal{DL}$-formulas. The $\mathcal{T}_{DL}$-solver rejects this guess as there is no assignment to integers of A, B, and C such that all three inequalities are true.

In the last row, the SAT-solver has backtracked the guess of q being true and instead tries $q^I = \bot$. Again, as no pure propositional constraint is violated the SAT-solver itself does not reject the interpretation. For this propositional assignment only one $\mathcal{DL}$-inequality is forced, namely by the second clause. The $\mathcal{T}_{DL}$-solver determines that there are assignments for variables B and C in this formula and hence that there exists a $\Sigma^{\mathcal{DL}}$-interpretation that satisfies the forced $\mathcal{DL}$-formula(s). Hence the SMT-solver has proved that F is satisfiable, with a model I,J with $I = \{ p \mapsto \top, q \mapsto \bot \}$ and, e.g., $J = \{ A \mapsto 0, B \mapsto 1, C \mapsto 0 \}$ as a witness (typically, a $\mathcal{T}_{DL}$-solver can produce satisfying $\Sigma^{\mathcal{DL}}$-interpretations, like J, upon request).

3.4 Logic Programming

We assume the reader of this document has good familiarity with logic programming [102]. Nevertheless, we provide a short overview of the programming languages that frequently appear in this dissertation.

3.4.1 Horn fragment-based languages

The *Horn*-fragment we introduced above forms the foundation of logic programming. We consider two seminal logic programming languages that directly address this fragment. Most of the Inductive Logic Programming systems considered in this document – including our own – target hypotheses in (extensions of) these two languages.

Datalog Datalog [21] is a programming language that takes the Horn fragment and imposes the restriction that there are a finite number of terms, often by disallowing most applications of function symbols to other function symbols. Datalog programs are sets of universally quantified definite clauses, often called rules, and sets of facts, i.e. positive ground atoms. These programs are often referred to as a Datalog *database/knowledge base*. Because of the finite terms restriction, any Datalog program has a finite number of Herbrand interpretations, which shows that most problems of interest, e.g. entailment, are decidable. As the entire Horn-fragment enjoys the least model property, all consequences of a Datalog program can be characterised by a single Herbrand interpretation. Guided by this property, the typical way of evaluating queries on a Datalog program is by so-called *forward chaining*. The idea is to saturate the sets of facts – by iteratively checking which bodies of quantified rules can be instantiated such that the instantiated head atom can be derived as a new fact – until a fixpoint is reached. This fixed point *is* the least Herbrand model of the program. There are many extensions to Datalog, such as allowing negation in the bodies of rules (typically with a stratification condition) and replacing universal quantifiers by existential ones [103].

Prolog Prolog [85] – arguably the most influential logic programming language – also addresses the Horn fragment and imposes no restriction on the occurrences of (nested) function symbols. In essence, a Prolog program is a set of definite clauses. These clauses

are then read front-to-back, namely that to prove the entailment of (an instantiation of) the head literal of such a rule, it is necessary to prove that the literals in the body of the rule hold (for the instantiation of variables shared with the head literal). This operational reading – typically referred to as *backward chaining* – corresponds to using SLD-resolution to search for refutation proofs of goal literals (i.e. the negation of an atom whose entailment we want to check). Often times there are multiple choices for which rule to resolve a goal literal with, which gives rise to a backtracking depth-first search procedure. This procedure is *not* guaranteed to terminate, reflecting the undecidable nature of the logic fragment that Prolog addresses. Typical implementations – often interpreters – add many features on top of this clean and simple language, such as negation as failure and primitives that alter the proof search procedure (e.g. cuts).

3.4.2 Answer Set Programming

Another paradigm for logic programming rose from the need for *declarative* semantics of logic programs make use of negation and disjunction. For this purpose, Gelfond and Lifschitz [61] introduced the notion of stable models, also known as *answer sets*. The *rules* of Answer Set Programming programs [104] are in essence definite clauses which also allow (default) negation to occur in bodies and disjunction (or choice) to occur in the heads of rules. The rules can quantify over (first-order) variables and the head of a rule is allowed to be empty to signify a (hard) constraint. The *meaning* of these programs is typically given in terms of the propositional fragments, with first-order programs needing to be grounded. The idea is to identify (Boolean-)interpretations that are (in a sense) minimal with respect to set inclusion, as this corresponds to the closed world assumption (i.e. only those facts that we have evidence for we take to be true). The actual definition of which interpretations are answer sets is slightly technical – we refer the interested reader to this overview by Eiter, et al. [48]. The modern approach to finding these answer sets is by first grounding the program and to then hand it off to a SAT-solver which accounts for the non-standard semantics [58, 4].

Chapter 4

Learning Programs by Learning From Failures

This chapter introduces the Learning From Failures approach to Inductive Logic Programming. We use formulas to represent hypothesis spaces, such that each model of such a formula corresponds to exactly one hypothesis. We select a viable hypothesis by searching for a model, converting it to a program, and then test it against examples. From the failure of a hypothesis we learn that related programs must also fail. We identify three classes of programs related to a failed hypothesis: generalisations, specialisations and programs which must contain redundancies. θ-subsumption-based constraints can then be added to the formula, causing the overall formula to have fewer models and hence to represent fewer viable hypotheses.

We show how Learning From Failures' falsification-based methodology addresses Learning From Entailment problems – essentially by making these problems into Satisfiability Modulo Theory problems – and can capture the popular synthesis-as-rule-selection approach to ILP. We introduce a three-stage loop approach to codify our approach to solving Learning From Failures problems. We present our synthesis-as-literal-selection implementation, `Popper`, and evaluate it by running experiments on three domains (toy game problems, robot strategies, and list transformations).

4.1 Introduction

Working within the setting of Inductive Logic Programming (ILP), we frame the program induction problem fully in logic [117, 136]. Our positive and negative examples are ground literals making use of a new predicate, our *target predicate*. The goal is to learn a logic program definition for this target predicate. This definition should be in terms of existing predicates and possibly itself[1], in case of a recursive hypothesis. Definitions for existing predicates constitute a library, which is referred to as *Background Knowledge* (BK) in ILP. With respect to a choice of BK, we say a logic program is *well-defined* if all predicates it contains are defined by itself or the BK. Our set of viable hypotheses is then a subset of this space of well-defined programs. In ILP, the sets of viable hypotheses tend to be enormous, see Section 4.2.2. The fundamental problem we hence set out to tackle is to

[1] In [36], we expand upon the approach this chapter presents and additionally support *predicate invention*, i.e. learning definitions for other new predicates at the same time as our target predicate.

efficiently search through these large spaces of hypotheses.

4.1.1 Falsification-based program induction

Just as scientists try to work out which theories fit their observations, we are interested in programs fitting our data. Hence we take inspiration from how empirical science gets conducted [64]. We take a simple account of science as a guide for methodology.

Suppose a scientist is faced with trying to explain some data. First, the scientist surveys the field and determines a set of theories that seem viable. Next, the scientist selects one of the theories as a hypothesis and tests it against the data. Either the theory is a good fit, and deemed a solution, or the theory is inconsistent with the data. According to Karl Popper [133], in the latter case the scientist rejects the hypothesis as the theory has now been falsified. Given that there are more hypotheses to examine, the scientist takes note of why the just tested hypothesis failed and then discards it. While picking the next hypothesis, the scientist takes care to not select a theory that must necessarily fail due to the same reason that a previous hypothesis failed. In our account, this iterative process now continues until the scientist comes across a hypothesis that fits the data.

Just as the first ILP system, the Model Inference System by Shapiro [145], was inspired by Popper's account of science, we too seek to automate the above process. Unlike many early ILP systems, which sought to iteratively construct programs clause-by-clause [135, 118, 137, 16, 150, 1], or refine a hypothesis by modifying it [145, 18, 7, 39], we seek to reason about the entire space of (viable) hypotheses. That is, our approach to automating this account of science is to keep track of the whole set of hypotheses and, just as our scientist, *refine* this set by what we learn from testing a hypothesis on data.

Seeking to track our belief about which hypotheses are viable, similarly to other modern ILP systems [26, 95, 139, 76, 13, 2], we use a *formula* to represent the hypothesis space. Each satisfying assignment of this formula represents a hypothesis. For example, we can make propositional formula $R_1 \vee R_2$ *represent a hypothesis space* by saying that when R_1 is assigned true this *selects* a clause $\text{last}(A, B) \leftarrow \text{head}(A, B)$ as being part of our hypothesis (i.e. program) and similarly when R_2 is assigned true this designates that clause $\text{last}(A, B) \leftarrow \text{reverse}(A, C), \text{head}(C, B)$ occurs in the selected hypothesis.

We can refine such representations by adding *constraints* to the hypothesis space-representing formula, formulas which limit the models of the overall combined formula. E.g, a formula like $(R_1 \wedge R_2) \rightarrow \bot$ functions as a constraint for above propositional formula as it eliminates the possibility of selecting the hypothesis which consists of both clauses.

We present a framework, which we call Learning From Failures, to codify the constraint-accumulation approach to Inductive Logic Programming. By abstracting away from the details of the encoding of the hypothesis space, our framework is able to capture both the popular synthesis-as-rule-selection approach, see Example 4.2.21, as well as our novel synthesis-as-literal-selection approach, see Section 4.5.

4.1.2 Generate, test, and constrain

Given our choice of using formulas to reason about the set of viable hypotheses, we now seek to emulate the process of gradually narrowing down this set. Whereas other approaches [49, 50, 76, 2] try to reduce the whole ILP problem to a formula solvable by a single solver, we instead seek to decompose the ILP problem. We identify three distinct

phases that the scientist goes through iteratively: first selecting a viable hypothesis, next testing that hypothesis, and thereupon revising the set of viable hypotheses.

We associate a *stage* with each of these phases: generate, test[2], and constrain. Figure 4.1 shows that we charge the generate stage with selecting a (still) viable hypothesis, whereupon this hypothesis, in the from of a logic program, gets handed off to the test stage. The test stage evaluates the program against the examples, with the Background Knowledge program fixing the interpretation of predicates made available for use in hypotheses. If the hypothesis fits the data, we are done. Otherwise the test stage observes that the hypothesis *fails*, in which case the hypothesis and its failure get handed over to the constrain stage. The constrain stage is responsible for coming up with a formula that, based on the observed failure, appropriately constrains the set of models which represent the viable hypotheses. This formula then gets handed to the generate stage so that its subsequent hypothesis generation is appropriately constrained.

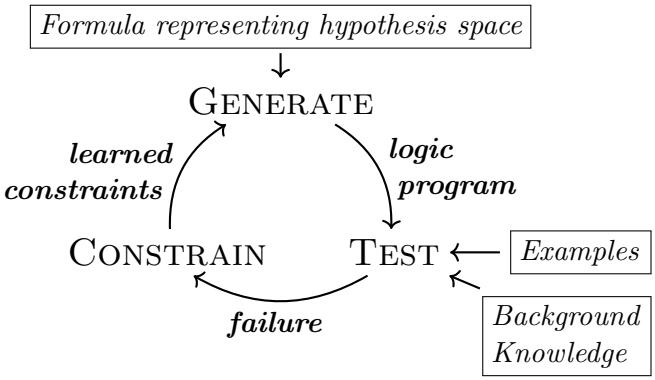


Figure 4.1: The generate, test, and constrain loop.

Example 4.1.1. To illustrate our three-stage approach, consider learning a *last/2* hypothesis to find the last element of a list. For simplicity, assume an initial hypothesis space $\mathcal{H}_1$:

$$\mathcal{H}_1 = \left\{ \begin{array}{l} H_1 = \{ \ \text{last}(A,B) \leftarrow \text{head}(A,B) \ \} \\ H_2 = \{ \ \text{last}(A,B) \leftarrow \text{head}(A,B), \text{empty}(A) \ \} \\ H_3 = \{ \ \text{last}(A,B) \leftarrow \text{tail}(A,C), \text{head}(C,B) \ \} \\ H_4 = \{ \ \text{last}(A,B) \leftarrow \text{reverse}(A,C), \text{head}(C,B) \ \} \\ H_5 = \{ \ \text{last}(A,B) \leftarrow \text{head}(A,B), \text{reverse}(A,C), \text{head}(C,B) \ \} \\ H_6 = \left\{ \begin{array}{l} \text{last}(A,B) \leftarrow \text{tail}(A,C), \text{head}(C,B) \\ \text{last}(A,B) \leftarrow \text{reverse}(A,C), \text{head}(C,B) \end{array} \right\} \\ H_7 = \left\{ \begin{array}{l} \text{last}(A,B) \leftarrow \text{tail}(A,C), \text{head}(C,B) \\ \text{last}(A,B) \leftarrow \text{tail}(A,C), \text{tail}(C,D), \text{head}(D,B) \end{array} \right\} \\ H_8 = \left\{ \begin{array}{l} \text{last}(A,B) \leftarrow \text{reverse}(A,C), \text{tail}(C,D), \text{head}(D,B) \\ \text{last}(A,B) \leftarrow \text{tail}(A,C), \text{reverse}(C,D), \text{head}(D,B) \end{array} \right\} \end{array} \right\}$$

Also assume we have the positive ($\mathcal{E}_1^+$) and negative ($\mathcal{E}_1^-$) examples:

[2]The generate and test stages carry these names in recognition of the many enumerative approaches within program synthesis and beyond.

$$\mathcal{E}_1^+ = \left\{ \begin{array}{l} \text{last}([l,a,u,r,a],a) \\ \text{last}([p,e,n,e,l,o,p,e],e) \end{array} \right\} \quad \mathcal{E}_1^- = \left\{ \begin{array}{l} \text{last}([e,m,m,a],m) \\ \text{last}([j,a,m,e,s],e) \end{array} \right\}$$

In the generate stage, the system generates a hypothesis:

$$H_1 = \left\{ \text{ last}(A,B) \leftarrow \text{head}(A,B) \right\}$$

In the test stage, the system tests H_1 – with a program $\mathcal{BK}_1$ providing definitions for head/2, empty/1, tail/2, and reverse/2 – against the examples and finds that it *fails* because it does not entail any positive example. Program H_1 is therefore said to be too *specific*. It follows that any program which is more specific cannot be a solution either. In the constrain stage, the system learns hypothesis constraints to prune specialisations of H_1 (H_2 and H_5), i.e. programs which entail at most as much H_1, from the hypothesis space. The hypothesis space is now:

$$\mathcal{H}_2 = \left\{ \begin{array}{l} H_3 = \left\{ \text{ last}(A,B) \leftarrow \text{tail}(A,C), \text{head}(C,B) \right\} \\ H_4 = \left\{ \text{ last}(A,B) \leftarrow \text{reverse}(A,C), \text{head}(C,B) \right\} \\ H_6 = \left\{ \begin{array}{l} \text{last}(A,B) \leftarrow \text{tail}(A,C), \text{head}(C,B) \\ \text{last}(A,B) \leftarrow \text{reverse}(A,C), \text{head}(C,B) \end{array} \right\} \\ H_7 = \left\{ \begin{array}{l} \text{last}(A,B) \leftarrow \text{tail}(A,C), \text{head}(C,B) \\ \text{last}(A,B) \leftarrow \text{tail}(A,C), \text{tail}(C,D), \text{head}(D,B) \end{array} \right\} \\ H_8 = \left\{ \begin{array}{l} \text{last}(A,B) \leftarrow \text{reverse}(A,C), \text{tail}(C,D), \text{head}(D,B) \\ \text{last}(A,B) \leftarrow \text{tail}(A,C), \text{reverse}(C,D), \text{head}(D,B) \end{array} \right\} \end{array} \right\}$$

Upon returning to the generate stage, the system generates the next smallest hypothesis:

$$H_3 = \left\{ \text{ last}(A,B) \leftarrow \text{tail}(A,C), \text{head}(C,B) \right\}$$

The system tests H_3 against the examples and finds that it fails because it entails the negative example $\text{last}([e,m,m,a],m)$. H_3 is therefore said to be too *general*. It follows that any program which is more general cannot be a solution either. The system now learns constraints to prune generalisations of H_3 (H_6 and H_7), i.e. programs which entail at least as much as H_3, from the hypothesis space. The hypothesis space becomes:

$$\mathcal{H}_3 = \left\{ \begin{array}{l} H_4 = \left\{ \text{ last}(A,B) \leftarrow \text{reverse}(A,C), \text{head}(C,B) \right\} \\ H_8 = \left\{ \begin{array}{l} \text{last}(A,B) \leftarrow \text{reverse}(A,C), \text{tail}(C,D), \text{head}(D,B) \\ \text{last}(A,B) \leftarrow \text{tail}(A,C), \text{reverse}(C,D), \text{head}(D,B) \end{array} \right\} \end{array} \right\}$$

The system generates another hypothesis, H_4, tests it against the examples, finds that it does not fail, and returns it.

4.1.3 Conflict-driven solving

The underlying approach of guessing assignments of a formula, checking if it is consistent and, if not, constraining the formula with other formulas is essentially that of Conflict-Driven Clause Learning [105], as used *within* SAT-solvers, e.g. [60]. Given a propositional formula, a CDCL SAT-solver builds up assignments and upon detecting that an assignment is in conflict with the formula, has a procedure to detect which part of the assignment is responsible, which then leads to learning a *conflict clause* which gets saved by conjoining

it with the original formula, thereby helping to more quickly reject other guesses which lead to the same conflict.

Our approach can be seen as delegating the detection of a conflicting assignment to an external component, the test stage. Such a two-stage set-up is essentially the same as that of an SMT-solver [12]: A SAT-solver makes guesses for propositional assignments, where, based on the guess – in addition to normal CDCL-solving – a set of (first-order) formulas gets selected and a separate solver, the *theory solver*, checks whether these formulas are in conflict with one another. Whenever this is the case, an additional conflict clause is learned that constrains subsequent guesses. See Section 3.3 for a more extensive overview.

Whereas in typical SMT-solving the theory solver is responsible for deriving the conflict clauses – i.e. constraining formulas – we add-in the constrain stage as an intermediary. We do so because when we observe a failed hypothesis we are able to reason using θ-subsumption that many related programs can be pruned as well. The conflict clauses learned by CDCL and SMT-solvers are geared solely toward pruning guesses that are subsets/supersets of a conflicting assignment. The constrain stage comes up with stronger constraints, ones that additionally prunes *symmetries* in the (encoding of the) hypothesis space, especially around variable renaming. Hence the constrain stage implements a form of dynamic symmetry breaking [128, 45, 108].

Popper

We present an implementation of the three-stage loop approach to solving Learning From Failures problems: `Popper`. `Popper` uses a declarative program, written in Answer Set Programming, to encode the hypothesis space. From a *predicate declarations* bias, this program derives a synthesis-as-literal-selection encoding – i.e. the granularity at which components of a hypothesis gets selected is that of literals rather clauses. The test stage we implement with a Prolog interpreter. The learned constraints are similarly declarative and directly encode the failed program as a constraint. We use the multi-shot solving [59] capability of an state-of-the-art ASP solver [58] to emulate the framework of SMT-solving, with Clasp [60] as the SAT-solver and a Prolog interpreter serving as a *Horn*-theory solver.

4.1.4 Contributions

The contributions of this chapter are:

- We formally define Learning From Failures as the constraint satisfaction approach to Learning From Entailment problems. We proof that when accumulating *sound* constrains, in the limit[3], we are guaranteed to find solutions (if they exists).
- We show how finding solutions to Learning From Failures problems can be rephrased as the problem of finding models of Satisfiability Modulo Theories problems, and demonstrate this approach with a synthesis-as-rule-selection example. We introduce the three-stage loop as an iterative approach to solving these problems.
- We derive three types of θ-subsumption-based constraints, generalisation, specialisation and redundancy, and proof these constraints to be sound w.r.t. hypotheses that are inconsistent, incomplete and totally incomplete, respectively.

[3]Note that formally we need decidability of our logic programs and, in case of an infinite hypothesis space, an appropriate search strategy. See Section 4.5.6.

- We present `Popper`, a synthesis-as-literal-selection ILP system which uses an ASP program to represent viable hypotheses. `Popper` learns recursive definite programs of minimal size. `Popper` supports declarative hypothesis constraints and supports infinite domains.
- We empirically evaluate `Popper` on three problem settings (toy game problems, robot strategies, and list transformations), and show that
  1. our constraints can drastically reduce hypothesis space exploration,
  2. `Popper` scales well with respect to the optimal solution size, the number of background relations, the domain size, the number of training examples, and the size of the training examples
  3. `Popper` can substantially outperform existing ILP systems, both in terms of predictive accuracies and learning times.

4.2 The Learning From Failures Framework

We start by formally defining the *Learning From Entailment* problem [136] that the techniques in this dissertation address.

4.2.1 Learning From Entailment

Of the different kinds of formulas used within logic programming – see Section 3.4 – we use the term *program* to primarily refer to:

Definition 4.2.1. A *(definite) program* is a set of first-order definite Horn clauses.

We will often refer to the single positive literal of a definite clause as the clause's *head*. Similarly, the set of negative literals of a clause will often be referred to as its *body*. Programs consist of definitions:

Definition 4.2.2. A program P is a *definition of predicate* p if the head atom of each (definite) clause of P has p as its predicate.

A program P *defines* predicate p if a subset of P is a definition of p. In order to define hypotheses, we first state when a program is well-defined with respect to the predicates that occur in the program.

Definition 4.2.3. A program H is *well-defined with regard to program* P if:
- Each predicate occurring in H has a definition in terms of a subset of $H \cup P$
- No predicate defined by H is already defined by P

Let $\mathcal{BK}$ be a first-order Horn program denoting our Background Knowledge, that is, a library of definitions for predicates. For a program H to qualify as a *hypothesis* it must be well-defined w.r.t. the $\mathcal{BK}$. Hence, a hypothesis H can refer to $\mathcal{BK}$'s predicates but the clauses defining predicates in the $\mathcal{BK}$ cannot refer to predicates defined by H. This means hypotheses must define new concepts rather than extend existing ones already defined by the $\mathcal{BK}$.

The last notion we need is that of examples to learn from: $\mathcal{E}^+$ and $\mathcal{E}^-$ are two sets of ground atoms representing positive and negative examples, respectively. We stipulate

that the predicate of each atom of $\mathcal{E}^+$ and $\mathcal{E}^-$ is the same (and does not occur in $\mathcal{BK}$), and dub this to be the *target predicate*.

As the positive examples can already be interpreted as Horn clauses ($\{ e^+ \leftarrow \mid e^+ \in \mathcal{E}^+ \}$ is logically equivalent to $\mathcal{E}^+$), we define $\neg\mathcal{E}^- = \{ \leftarrow e^- \mid e^- \in \mathcal{E}^- \}$ (which is logically equivalent to $\{ \neg e^- \mid e^- \in \mathcal{E}^- \}$), so that the examples collectively can be interpreted as a set of Horn clauses, or, alternatively, a set of literals.

Learning From Entailment definition

The central program induction problem this document tackles can now be stated as:

Definition 4.2.4. A *Learning From Entailment input* $(\mathcal{H}, \mathcal{BK}, \mathcal{E} = \mathcal{E}^+ \cup \neg\mathcal{E}^-)$ consists of:

- $\mathcal{H}$, a set of programs, all well-defined w.r.t. $\mathcal{BK}$, i.e. the set of hypotheses
- $\mathcal{BK}$, the Background Knowledge program[4], i.e. a library defining predicates
- $\mathcal{E}$, a set of literals, each corresponding to either a positive or negative example

We require $\mathcal{H}$ to be non-empty. While $\mathcal{E}$ must also be non-empty, either of $\mathcal{E}^+$ and $\mathcal{E}^-$ is allowed to be the empty set.

Example 4.2.5. We can formally make Example 4.1.1 into a Learning From Entailment problem by setting $(\mathcal{H}_1, \mathcal{BK}_1, \mathcal{E}_1^+ \cup \{ \neg e^- \mid e^- \in \mathcal{E}_1^- \})$ as the input, where $\mathcal{BK}_1$ is a set of definite clauses defining predicates head/2, empty/1, tail/2 and reverse/2 in the usual way.

In terms of these inputs, we define correctness qualifiers for hypotheses, mostly keeping to standard ILP terminology [122]:

Definition 4.2.6. Given a program $\mathcal{BK}$ and sets of atoms $\mathcal{E}^+$ and $\mathcal{E}^-$, a program H is:

- *Complete* when $\forall e^+ \in \mathcal{E}^+ \colon \mathcal{BK} \cup H \models e^+$
- *Consistent* when $\forall e^- \in \mathcal{E}^- \colon \mathcal{BK} \cup H \not\models e^-$
- *Incomplete* when $\exists e^+ \in \mathcal{E}^+ \colon \mathcal{BK} \cup H \not\models e^+$
- *Inconsistent* when $\exists e^- \in \mathcal{E}^- \colon \mathcal{BK} \cup H \models e^-$
- *Totally incomplete* when $\forall e^+ \in \mathcal{E}^+ \colon \mathcal{BK} \cup H \not\models e^+$

We now define the problem specified by an input:

Definition 4.2.7. Given a Learning From Entailment input $(\mathcal{H}, \mathcal{BK}, \mathcal{E}^+ \cup \neg\mathcal{E}^-)$, the corresponding *Learning From Entailment problem* is to find an $H \in \mathcal{H}$ that is complete and consistent with regard to $\mathcal{BK}$ and $\mathcal{E}^+$ and $\mathcal{E}^-$, in which case H is a *solution*.

A hypothesis *fails* to be a solution for multiple reasons; they are always at least incomplete or inconsistent.

Example 4.2.8. In Example 4.1.1, hypothesis H_1 failed as it was (totally) incomplete and hypothesis H_3 failed as it was inconsistent. Hypothesis H_4 is both complete and consistent and hence a solution.

[4] In what follows, including for our implementation, all we rely on is that $\mathcal{BK}$ acts as a (potentially infinite) set of facts, hence the $\mathcal{BK}$ being defined as a normal program or ASP program is fine. However, as a number of formal arguments are easier if $\mathcal{BK}$ is a Horn program, we stick with the definition that $\mathcal{BK}$ is a Horn program.

Optimality and redundancy

As there are potentially many solutions, we can state that some solutions are preferred over others. In what follows, we tend to focus on finding the smallest solutions.

Definition 4.2.9. The function $size(P)$ returns the total number of literals in the program P.

For our purposes, solutions are *optimal* with respect to their size:

Definition 4.2.10. Given Learning From Entailment problem $(\mathcal{H}, \mathcal{BK}, \mathcal{E})$, a hypothesis $H \in \mathcal{H}$ is an *optimal solution* when

- H is a solution
- For each solution $H' \in \mathcal{H}$, $size(H) \leq size(H')$

An important way for a solution to be non-optimal is because it has a clause that is not contributing to entailing the examples.

Definition 4.2.11. Given a program P and a set literals E, a clause $C \in P$ is *redundant with regard to* E if whenever $P \models E$ then $P \setminus \{ C \} \models E$.

Example 4.2.12. Suppose $\mathcal{E} = \{ \text{last}([1,2],2), \neg\text{last}([1,2,3],2) \}$ and we have a program

$$P = \left\{ \begin{array}{lcl} \text{last}(A,B) & \leftarrow & \text{reverse}(A,C), \text{head}(C,B) \\ \text{last}(A,B) & \leftarrow & \text{head}(A,B) \end{array} \right\}$$

Then the second clause of P is redundant w.r.t. $\mathcal{E}$ as by just leaving it out we still entail $\mathcal{E}$.

We can also reason that certain clauses are redundant irrespective of the examples:

Definition 4.2.13. Given a program P, a clause $C \in P$ is *redundant* when $P \setminus \{ C \} \models P$.

Example 4.2.14. Suppose $\mathcal{E} = \{ \text{last}([1,2],2), \neg\text{last}([1,2,3],2) \}$ and we have a program

$$Q = \left\{ \begin{array}{lcl} \text{last}(A,B) & \leftarrow & \text{reverse}(A,C), \text{head}(C,B) \\ \text{last}(A,B) & \leftarrow & \text{tail}(A,C), \text{tail}(C,A), \text{head}(C,B) \end{array} \right\}$$

Then the second clause of Q is redundant as it cannot contribute any consequences to Q as there is no instantiation of variables of this clause such that the body literals $\text{tail}(A,C), \text{tail}(C,A)$ hold at the same time.

As is to be expected, the following holds:

Proposition 4.2.15. If a clause C is redundant within P then C is also redundant within P with regard to E.

Given that we will use redundancy arguments to reason about the non-optimality of hypotheses, we will introduce one reasonable assumption regarding our hypothesis spaces:

Assumption 4.2.16. For each LFE-problem $(\mathcal{H}, \mathcal{BK}, \mathcal{E})$ that we work with, no optimal solution $H \in \mathcal{H}$ contains a redundant clause w.r.t. $\mathcal{E}$.

4.2.2 Hypothesis space

For the set of hypotheses, $\mathcal{H}$, we adopt the terminology *hypothesis space*. By Definition 4.2.4, the hypotheses need to be well-defined w.r.t. the $\mathcal{BK}$. Hence, in the LFE- (Learning From Entailment-) setting, when faced with a learning task, before stating our hypothesis space, we must first choose which library of predicates to provide as our Background Knowledge. Depending on this choice of $\mathcal{BK}$ there is a natural hypothesis space to consider (based on a couple of parameters):

Proposition 4.2.17. Let $\mathcal{P}_B$ be the set of predicate symbols allowed in the bodies of hypotheses, $\mathcal{P}_H$ the set of predicate symbols allowed in the head literals of hypotheses, a the maximum arity of these predicates, v the maximum number of unique variables allowed in each clause, m the maximum number of body literals allowed in a clause, and n the maximum number of clauses allowed in a hypothesis. Then the maximum number of hypotheses in a hypothesis space keeping to these parameters is:

$$\sum_{j=1}^{n} \binom{|\mathcal{P}_H|v^a \ \sum_{i=1}^{m} \binom{|\mathcal{P}_B|v^a}{i}}{j}$$

Proof. Let C be an arbitrary clause of a hypothesis. There are $|\mathcal{P}_H|v^a$ ways to define C's head literal. There are $|\mathcal{P}_B|v^a$ ways of defining a body literal of C. There are $\binom{|\mathcal{P}_B|v^a}{k}$ ways to choose k body literals. The number of body literals is bounded by m, meaning that there are $\sum_{i=1}^{m}\binom{|\mathcal{P}_B|v^a}{i}$ ways to choose at most m body literals. Hence, there are $N = |\mathcal{P}_H|v^a \ \sum_{i=1}^{m}\binom{|\mathcal{P}_B|v^a}{i}$ options for defining C. Given N options for clauses, there are $\binom{N}{k}$ ways to form a k-clause hypothesis. Therefore there are $\sum_{j=1}^{n}\binom{N}{j}$ ways to define a hypothesis with at most n clauses. □

When tackling a LFE-task, we need to decide on how many predicates our hypotheses are allowed to define, which gives us $|\mathcal{P}_H|$. Depending on our choice of $\mathcal{BK}$, we can then set $\mathcal{P}_B = \mathcal{P}(\mathcal{BK}) \cup \mathcal{P}_H$, where $\mathcal{P}(\mathcal{BK})$ are the predicates defined by $\mathcal{BK}$. If we set a to the minimal arity of any predicate symbol in $\mathcal{P}_H \cup \mathcal{P}_B$, e.g. 2 in case all are binary predicates, the above quantity appropriately under-approximates the size of the fully unconstrained hypothesis space. Note how for any non-trivial choice of v, m and n, the size of the hypothesis space is huge.

This result motivates the approach we now introduce, as it allows for pruning large chunks of the hypothesis space through constraints.

4.2.3 Learning From Failures

For the Learning From Failures approach to Inductive Logic Programming, we choose to address the Learning From Entailment problem by formalizing the search for a solution in the hypothesis space as a constraint satisfaction problem.

Representing hypothesis spaces by formulas

Our approach assumes we can *encode* a hypothesis space as a formula. As our framework is generic w.r.t. the logic fragment of these formulas, we introduce an arbitrary one to work with:

Var	$\mathcal{H}_1$-*clause*
R_1	$\text{last}(A, B) \leftarrow \text{head}(A, B)$
R_2	$\text{last}(A, B) \leftarrow \text{head}(A, B), \text{empty}(A)$
R_3	$\text{last}(A, B) \leftarrow \text{tail}(A, C), \text{head}(C, B)$
R_4	$\text{last}(A, B) \leftarrow \text{reverse}(A, C), \text{head}(C, B)$
R_5	$\text{last}(A, B) \leftarrow \text{head}(A, B), \text{reverse}(A, C), \text{head}(C, B)$
R_6	$\text{last}(A, B) \leftarrow \text{tail}(A, C), \text{tail}(C, D), \text{head}(D, B)$
R_7	$\text{last}(A, B) \leftarrow \text{reverse}(A, C), \text{tail}(C, D), \text{head}(D, B)$
R_8	$\text{last}(A, B) \leftarrow \text{tail}(A, C), \text{reverse}(C, D), \text{head}(D, B)$

Table 4.1: Mapping from propositional variables to the clauses of $\mathcal{H}_1$ from Example 4.1.1 as typically used by synthesis-as-rule-selection approaches.

Definition 4.2.18. Let $\mathcal{L}$ be an arbitrary 1st-order logic fragment with $\Sigma_\mathcal{L}$ its vocabulary, $\mathcal{F}_\mathcal{L}$ its formulas, and theory $\mathcal{T}_\mathcal{L}$ its set of intended $\Sigma_\mathcal{L}$-interpretations.

We think of $\mathcal{L}$ as a *constraint* language. That is, we take the view that an $\mathcal{L}$-formula $F \in \mathcal{F}_\mathcal{L}$ has a number of free variables and that F *constrains* the number of valid assignments for these variables. Given a formula F, we identify assignments for F's variables with interpretations in $\mathcal{T}_\mathcal{L}$.

$\mathcal{L}$ is of interest to us when its formulas can be used to represent hypothesis spaces, with assignments whose corresponding interpretation is a model being of particular importance:

Definition 4.2.19. An $\mathcal{L}$-formula F *represents a hypothesis space* $\mathcal{H}$ if for each distinct interpretation I that is a model of F, i.e. $[\![F]\!]_I = \top$, there is a unique $H \in \mathcal{H}$.

When a hypothesis space is represented by a formula, we can formalize the one-to-one correspondence between models and programs as a mapping:

Proposition 4.2.20. Whenever an $\mathcal{L}$-formula F represents a hypothesis space $\mathcal{H}$ then there exists a bijective function $\langle\!\langle F \rangle\!\rangle_\_ : \mathcal{M}(F) \to \mathcal{H}$, where $\mathcal{M}(F) = \{\, I \in \mathcal{T}_\mathcal{L} \mid [\![F]\!]_I = \top \,\}$.

Hence for each full assignment I such that $[\![F]\!]_I = \top$ we have that $\langle\!\langle F \rangle\!\rangle_I = H$ where H is a program from $\mathcal{H}$. In other words, we *interpret* a model I of F as a hypothesis $H \in \mathcal{H}$. Often we will write $F_\mathcal{H}$ for an $\mathcal{L}$-formula that represents a hypothesis space $\mathcal{H}$ and assume that the function $\langle\!\langle F_\mathcal{H} \rangle\!\rangle_\_$ is available so we can map models of $F_\mathcal{H}$ to hypotheses of $\mathcal{H}$.

Example 4.2.21. Suppose $\mathcal{L}$ is just propositional logic and we want to represent $\mathcal{H}_1$ from Example 4.1.1. By using the synthesis-as-rule-selection approach – see Section 2.1.3 – we can introduce propositional variables, each one standing for whether a distinct clause of $\mathcal{H}_1$ occurs in the hypothesis or not, such as in Table 4.1. If we take a propositional formula like

$$F_1 = (R_1) \vee (R_2) \vee (R_3) \vee (R_4) \vee (R_5) \vee (R_3 \wedge R_4) \vee (R_3 \wedge R_6) \vee (R_7 \wedge R_8)$$

we can interpret interpretations/assignments of F_1 as pinpointing the hypotheses of $\mathcal{H}_1$, like

$$I_{h_6} = \{R_1 \mapsto \bot, R_2 \mapsto \bot, R_3 \mapsto \top, R_4 \mapsto \top, R_5 \mapsto \bot, R_6 \mapsto \bot, R_7 \mapsto \bot, R_8 \mapsto \bot\}$$

corresponding to $h_6 \in \mathcal{H}_1$. As I_{h_6} is a model of F, we can set

$$\langle\!\langle F_1 \rangle\!\rangle_{I_{h_6}} = \left\{ \begin{array}{l} \text{last}(A,B) \leftarrow \text{tail}(A,C), \text{head}(C,B) \\ \text{last}(A,B) \leftarrow \text{reverse}(A,C), \text{head}(C,B) \end{array} \right\}$$

While we can think of F_1 as representing a hypothesis space, this hypothesis space cannot be $\mathcal{H}_1$ as F_1 has (far) more than 8 models. For example, we have interpretations, like $I_{h_1 \cup h_2} = \{R_1 \mapsto \top, R_2 \mapsto \top, R_3 \mapsto \bot, R_4 \mapsto \bot, R_5 \mapsto \bot, R_6 \mapsto \bot, R_7 \mapsto \bot, R_8 \mapsto \bot\}$, which models F_1 and which would most naturally correspond to

$$\langle\!\langle F_1 \rangle\!\rangle_{I_{h_1 \cup h_2}} = \left\{ \begin{array}{l} \text{last}(A,B) \leftarrow \text{head}(A,B) \\ \text{last}(A,B) \leftarrow \text{head}(A,B), \text{empty}(A) \end{array} \right\}$$

Using propositional logic, to accurately capture the highly specific hypothesis space $\mathcal{H}_1$, we need a highly specific formula, like

$$\begin{aligned} F_{\mathcal{H}_1} = \; & (R_1) \vee (R_2) \vee (R_3) \vee (R_4) \vee (R_5) \vee (R_3 \wedge R_4) \vee (R_3 \wedge R_6) \vee (R_7 \wedge R_8) \\ & \wedge (R_1 \rightarrow \neg R_2 \wedge \neg R_3 \wedge \neg R_4 \wedge \neg R_5 \wedge \neg R_6 \wedge \neg R_7 \wedge \neg R_8) \\ & \wedge (R_2 \rightarrow \neg R_1 \wedge \neg R_3 \wedge \neg R_4 \wedge \neg R_5 \wedge \neg R_6 \wedge \neg R_7 \wedge \neg R_8) \\ & \wedge (R_3 \rightarrow \neg R_1 \wedge \neg R_2 \wedge \neg R_5 \wedge \neg R_7 \wedge \neg R_8) \\ & \wedge (R_4 \rightarrow \neg R_1 \wedge \neg R_2 \wedge \neg R_5 \wedge \neg R_6 \wedge \neg R_7 \wedge \neg R_8) \\ & \wedge (R_5 \rightarrow \neg R_1 \wedge \neg R_2 \wedge \neg R_3 \wedge \neg R_4 \wedge \neg R_6 \wedge \neg R_7 \wedge \neg R_8) \\ & \wedge (R_6 \rightarrow \neg R_1 \wedge \neg R_2 \wedge \neg R_4 \wedge \neg R_5 \wedge \neg R_7 \wedge \neg R_8) \\ & \wedge (R_7 \rightarrow \neg R_1 \wedge \neg R_2 \wedge \neg R_4 \wedge \neg R_5 \wedge \neg R_6 \wedge R_8) \\ & \wedge (R_8 \rightarrow \neg R_1 \wedge \neg R_2 \wedge \neg R_4 \wedge \neg R_5 \wedge \neg R_6 \wedge R_7) \\ & \wedge (\neg R_3 \vee \neg R_4 \vee \neg R_6) \end{aligned}$$

There are indeed 8 models of $F_{\mathcal{H}_1}$, with each of these interpretations naturally corresponding to exactly one hypothesis of $\mathcal{H}_1$, in accord with Table 4.1. For example, we have that $[\![F_{\mathcal{H}_1}]\!]_{I_{h_1 \cup h_2}} = \bot$ and $[\![F_{\mathcal{H}_1}]\!]_{I_{h_6}} = \top$ and hence can set

$$\langle\!\langle F_{\mathcal{H}_1} \rangle\!\rangle_{I_{h_6}} = \left\{ \begin{array}{l} \text{last}(A,B) \leftarrow \text{tail}(A,C), \text{head}(C,B) \\ \text{last}(A,B) \leftarrow \text{reverse}(A,C), \text{head}(C,B) \end{array} \right\}$$

Pruning hypothesis spaces by constraints

As stated, our intent is to rule out large chunks of the hypothesis space. In the context of hypothesis space-representing formulas this comes down to eliminating models. Our approach is to intersect the models of a formula $F_{\mathcal{H}}$ with those of another formula.

Definition 4.2.22. An $\mathcal{L}$-formula C is a *constraint* w.r.t. an $\mathcal{L}$-formula F when the models of $F \wedge C$ are a subset of the models of F.

We call a constraint a *hypothesis constraint* if it is meant to prune models from a formula which denotes a hypothesis space. We also introduce a slight abuse of notation: given a formula F and a *set* of constraints $\mathcal{C}$, when we write $F \wedge \mathcal{C}$ we mean $F \wedge (\bigwedge_{D \in \mathcal{C}} D)$.

Example 4.2.23. Consider F_1 and $F_{\mathcal{H}_1}$ from Example 4.2.21 and let

$$\begin{aligned} C_1 = \; & (R_1 \rightarrow \neg R_2 \wedge \neg R_3 \wedge \neg R_4 \wedge \neg R_5 \wedge \neg R_6 \wedge \neg R_7 \wedge \neg R_8) \\ & \wedge (R_2 \rightarrow \neg R_1 \wedge \neg R_3 \wedge \neg R_4 \wedge \neg R_5 \wedge \neg R_6 \wedge \neg R_7 \wedge \neg R_8) \\ & \wedge (R_3 \rightarrow \neg R_1 \wedge \neg R_2 \wedge \neg R_5 \wedge \neg R_7 \wedge \neg R_8) \\ & \wedge (R_4 \rightarrow \neg R_1 \wedge \neg R_2 \wedge \neg R_5 \wedge \neg R_6 \wedge \neg R_7 \wedge \neg R_8) \\ & \wedge (R_5 \rightarrow \neg R_1 \wedge \neg R_2 \wedge \neg R_3 \wedge \neg R_4 \wedge \neg R_6 \wedge \neg R_7 \wedge \neg R_8) \\ & \wedge (R_6 \rightarrow \neg R_1 \wedge \neg R_2 \wedge \neg R_4 \wedge \neg R_5 \wedge \neg R_7 \wedge \neg R_8) \\ & \wedge (R_7 \rightarrow \neg R_1 \wedge \neg R_2 \wedge \neg R_4 \wedge \neg R_5 \wedge \neg R_6 \wedge R_8) \\ & \wedge (R_8 \rightarrow \neg R_1 \wedge \neg R_2 \wedge \neg R_4 \wedge \neg R_5 \wedge \neg R_6 \wedge R_7) \\ & \wedge (\neg R_3 \vee \neg R_4 \vee \neg R_6) \end{aligned}$$

Then $F_{\mathcal{H}_1} = F_1 \wedge C_1$. Hence, in effect, formula C_1 is a hypothesis constraint relative to F_1. Constraint C_1 trims exactly those models of F_1 which represented programs not in $\mathcal{H}_1$.

As is clear from these examples, the synthesis-as-rule-selection approach captures very little of the structure of the hypothesis space. When using propositional logic with a variable per rule/clause, this can lead to large and opaque constraints.

The following example, demonstrating constraints that are simple and effective at the same time, is our first glimpse of our synthesis-as-literal-selection approach making use of a more expressive logic.

Example 4.2.24. In Section 4.5.3 we will introduce an encoding of hypothesis spaces into Answer Set Programming formulas. In that encoding, we have that an atom

```
head_literal(Cl,Pred,Arity,Args)
```

denotes that there is a clause labelled `Cl` which has a head literal with predicate symbol `Pred`, of arity `Arity`, applied to arguments `Args`. An example of a hypothesis constraint in this encoding is:

```
:- head_literal(_,p,2,_).
```

This constraint states that there cannot be any model of our encoding which represents a hypothesis that has predicate symbol `p`, of arity 2, applied to any arguments in the head of any clause.

The same encoding uses `body_literal(Cl,Pred,Arity,Args)` to denote that the clause labelled `Cl` has a body literal with predicate symbol `Pred`, of arity `Arity`, applied to arguments `Args`. Another example of a hypothesis constraint w.r.t. to our encoding is:

```
:- head_literal(_,p,2,_), body_literal(_,p,2,_).
```

This constraint prunes any model of our encoding which represents a hypothesis which has the predicate symbol `p` appear in the body of a clause as well as in the head of a (possibly another) clause.

Learning From Failures definition

We can now state what the inputs are for a Learning From Failures problem and what it means to solve it:

Definition 4.2.25. A *Learning From Failures input* $(F_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}{=}\mathcal{E}^+{\cup}\neg\mathcal{E}^-, \mathcal{C})$ consists of

- $F_{\mathcal{H}}$, an $\mathcal{L}$-formula representing hypothesis space $\mathcal{H}$, all of which well-defined w.r.t. $\mathcal{BK}$
- $\mathcal{BK}$, a Horn program consisting of a set of definitions of predicates
- $\mathcal{E}$, a set of literals, each corresponding to either a positive or negative example
- $\mathcal{C}$, a set of $\mathcal{L}$-formulas, each of which a hypothesis constraint w.r.t. $F_{\mathcal{H}}$

Definition 4.2.26. Given a Learning From Failures input $(F_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \mathcal{C})$, the corresponding *Learning From Failures problem* is to find an interpretation I of $F_{\mathcal{H}} \wedge \mathcal{C}$ such that $\langle\!\langle F_{\mathcal{H}} \rangle\!\rangle_I$ is complete and consistent w.r.t. $\mathcal{BK}$ and $\mathcal{E}$.

Learning From Entailment by Learning From Failures

The above definitions allow for any set of hypothesis constraints. In order to use Learning From Failures to address the Learning From Entailment problem, we restrict ourselves to constraints that are *sound*:

Definition 4.2.27. Given a Learning From Failures problem $(F_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \mathcal{C})$, the hypothesis constraints $\mathcal{C}$ are *sound* when $(F_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \mathcal{C})$ has the same solutions as $(F_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \emptyset)$.

That is, hypothesis constraints are sound when they do not prune solutions from the hypothesis space. From our definition of sound hypothesis constraints, the following result is immediate:

Proposition 4.2.28. A solution to a Learning From Failures problem $(F_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \mathcal{C})$ is a solution to the Learning From Entailment problem $(\mathcal{H}, \mathcal{BK}, \mathcal{E})$, and vice versa, given that hypothesis constraints $\mathcal{C}$ are sound.

When $\mathcal{C}$ is sound w.r.t. $F_{\mathcal{H}}$, we say that $F_{\mathcal{H}} \wedge \mathcal{C}$ is representing the *viable* hypotheses. That is, a set of viable hypotheses lies somewhere between the set of solutions in $\mathcal{H}$ and the full hypothesis space $\mathcal{H}$.

Example 4.2.29. Let us again consider $F_{\mathcal{H}_1}$ representing $\mathcal{H}_1$ from Example 4.2.21. We take $\mathcal{BK}_1$ as appropriate and $\mathcal{E}_1 = \mathcal{E}_1^+ \cup \{\, \neg e^- \mid e^- \in \mathcal{E}_1^- \,\}$. If we set $\mathcal{C}_1 = \emptyset$, we get a Learning From Failures problem $(F_{\mathcal{H}_1}, \mathcal{BK}_1, E_1, \mathcal{C}_1)$ whose constraints are trivially sound. Hence the interpretation

$$I_{h_4} = \{\, R_1 \mapsto \bot, R_2 \mapsto \bot, R_3 \mapsto \bot, R_4 \mapsto \top, R_5 \mapsto \bot, R_6 \mapsto \bot, R_7 \mapsto \bot, R_8 \mapsto \bot \,\}$$

remains a model of $F_{\mathcal{H}_1} \wedge \mathcal{C}_1$ and as $\langle\!\langle F_{\mathcal{H}_1} \wedge \mathcal{C}_1 \rangle\!\rangle_{I_{h_4}}$ is complete and consistent w.r.t. $\mathcal{E}_1$, it is a solution. If we let $\mathcal{C}_2 = \mathcal{C}_1 \cup \{\, \neg R_1 \wedge \neg R_2 \wedge \neg R_5 \,\}$ we get that $F_{\mathcal{H}_1} \wedge \mathcal{C}_2$ represents $\mathcal{H}_2$ from Example 4.1.1. As explained, $\mathcal{H}_2$ is $\mathcal{H}_1$ with some non-solutions pruned, meaning the (sole) constraint of $\mathcal{C}_2$ is sound. Having thus narrowed down the viable hypotheses, we have that I_{h_4} remains a model of $F_{\mathcal{H}_1} \wedge \mathcal{C}_2$ and hence $\langle\!\langle F_{\mathcal{H}_1} \wedge \mathcal{C}_2 \rangle\!\rangle_{I_{h_4}}$ is a solution to both $(F_{\mathcal{H}_1}, \mathcal{BK}_1, \mathcal{E}_1, \mathcal{C}_2)$ and the original LFE-problem $(\mathcal{H}_1, \mathcal{BK}_1, \mathcal{E}_1)$.

Often we are interested in finding an *optimal* solution. In this case we can stretch the notion of sound constraints to allow for pruning solutions which are too big to be optimal solutions:

Definition 4.2.30. Given a Learning From Failures problem $(F_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \mathcal{C})$, the hypothesis constraints $\mathcal{C}$ are *optimally sound* when $(F_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \mathcal{C})$ has the same optimal solutions as $(F_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \emptyset)$.

4.2.4 Learning From Failures as Satisfiability Modulo Theories

We now frame the Learning From Failures problem as one overall satisfaction problem. In particular, we make the LFF-problem into a Satisfiability Modulo Theories problem.

In Satisfiability Modulo Theories, a first-order formula F contains sub-formulas from two (or more) logic fragments, say $\mathcal{X}$ and $\mathcal{Y}$. Per Section 3.3, the structure of a SMT-formula F is such that an interpretation I for the $\mathcal{X}$-fragment symbols of F fixes a formula, $\langle\!\langle F \rangle\!\rangle_I^{\mathcal{Y}}$, from the $\mathcal{Y}$ logic fragment. In the case of LFF, we see $\mathcal{L}$ as the first logic fragment and *Horn* as the second logic fragment.

We will now make use of abstract formulas – which combine the vocabularies $\Sigma^{\mathcal{L}}$ and Σ^{Horn} of the $\mathcal{L}$ and *Horn* fragments, respectively – where an interpretation of $\Sigma^{\mathcal{L}}$-symbols fixes a (set of) *Horn*-(sub-)formula(s). Unusually, as also suggested by Bembenek et al. (2023) [13], we then let the theory, i.e. set of intended interpretations, for these *Horn* formulas depend on the $\mathcal{L}$-interpretation I, namely the *Horn*-interpretation(s) of interest are the least models of the *Horn*-formula(s) that interpretation I selects.

When viewed through this SMT-perspective, we can represent a Learning From Failures problem as a single formula:

Definition 4.2.31. A $\Sigma^{\mathcal{L},Horn}$-formula $F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E}$ *represents* an LFF-problem $(F_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \emptyset)$ when, given an $\mathcal{L}$-interpretation I, $\langle\!\langle F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \rangle\!\rangle_I^{Horn} = \mathcal{BK} \cup H \cup \mathcal{E}$ is a set of *Horn*-formulas whenever $\langle\!\langle F_{\mathcal{H}} \rangle\!\rangle_I = H$.

The essence of the above definition is that the mapping of interpretations to programs that was external before, e.g. $\langle\!\langle F_1 \rangle\!\rangle_-$ being captured by Table 4.1 from Example 4.2.21, is now *reified*, that is, included in the formula itself.

We have that hypothesis constraints $\mathcal{C}$ relative to $F_{\mathcal{H}}$ are still effective in reducing the potential models of $F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E}$:

Proposition 4.2.32. When formula $F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E}$ represents LFF-problem $(F_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \emptyset)$, for any set of hypothesis constraints $\mathcal{C}$ and interpretation I interpreting all $\mathcal{L}$-symbols of $F_{\mathcal{H}}$, either $[\![F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{C}]\!]_I = \bot$ or $\langle\!\langle F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{C} \rangle\!\rangle_I^{Horn} = \mathcal{BK} \cup H \cup \mathcal{E}$ where $H \in \mathcal{H}$.

Proof. Suppose I is a full assignment of the $\mathcal{L}$-symbols of $F_{\mathcal{H}}$. As I is a full assignment, either $[\![F_{\mathcal{H}} \wedge \mathcal{C}]\!]_I = \bot$ or $[\![F_{\mathcal{H}} \wedge \mathcal{C}]\!]_I = \top$. If $[\![F_{\mathcal{H}} \wedge \mathcal{C}]\!]_I = \bot$ then clearly $[\![F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{C}]\!]_I = \bot$. If we have $[\![F_{\mathcal{H}} \wedge \mathcal{C}]\!]_I = \top$, then there exists $H \in \mathcal{H}$ s.t. $\langle\!\langle F_{\mathcal{H}} \wedge \mathcal{C} \rangle\!\rangle_I = \langle\!\langle F_{\mathcal{H}} \rangle\!\rangle_I = H$ and hence $\langle\!\langle F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{C} \rangle\!\rangle_I^{Horn} = \mathcal{BK} \cup H \cup \mathcal{E}$. □

The usual approach to SMT is to specify the syntax of your logic fragments, such as which function and predicate symbols are allowed in formulas of the fragment, and to then fix a theory consisting of the intended interpretations for these symbols.

However, we are dealing with the *Horn*-fragment where the formulas are typically used to identify the relevant theory (i.e. the set of interpretations that are models of formulas). Given a *Horn* program P, we have that $\mathcal{T}_P$ is the set of all Herbrand interpretations which are a model of P. As explained in Section 3.2, *Horn*-formulas have the property that the intersection of all Herbrand models is still a model. Furthermore, this model contains all consequences of P. We use $\mathcal{T}_{LM(P)}$ to denote the theory that consists of just the least model of P.

As previously noted [13], to ensure we are checking satisfiability w.r.t. the right interpretations, we can make the theory we use to interpret predicate symbols dependent

on (part of) the selected Horn formulas. We now fix the intended interpretation of the formula $\langle\!\langle F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \rangle\!\rangle_I^{Horn}$ to the least model of the $\mathcal{BK}$ and the selected hypothesis:

Definition 4.2.33. Given a $\Sigma^{\mathcal{L},Horn}$-formula $F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E}$ that represents LFF-input $(F_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \emptyset)$ and a $\Sigma^{\mathcal{L}}$-interpretation I from theory $\mathcal{T}_{\mathcal{L}}$, the intended Σ^{Horn}-interpretation(s) of $\langle\!\langle F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \rangle\!\rangle_I^{Horn}$ is the theory $\mathcal{T}_{LM(\mathcal{BK} \cup H)} = \{ LM(\mathcal{BK} \cup H) \}$ where $H = \langle\!\langle F_{\mathcal{H}} \rangle\!\rangle_I$.

By the above definition, as the interpretation of the predicates defined by $\mathcal{BK}$ is fixed by $\mathcal{BK}$ and independent of H (and therefore of the interpretations I), we can think of ILP's Background *Knowledge* $\mathcal{BK}$ as a background *theory* in the traditional SMT-sense. That is, the Background Knowledge ensures that all non-hypothesis defined predicates have fixed interpretations. Hence, the notion of Background Knowledge parameterizes the notion of a background theory with a program, $\mathcal{BK}$. It is this program that determines the set of predicates that is made available as well as their (fixed) interpretation in the background theory.

With the above definitions in place, it works out that models w.r.t. to the intended theories are exactly solutions. In other words, the LFF-problem has a solution if and only if the corresponding SMT-formula is satisfiable.

Proposition 4.2.34. When $\Sigma^{\mathcal{L},Horn}$-formula $F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E}$ represents LFF-input $(F_{\mathcal{H}}, \mathcal{T}_{\mathcal{BK}}, \mathcal{E}, \emptyset)$,

1. for each solution $H \in \mathcal{H}$ there is an intended $\Sigma^{\mathcal{L},Horn}$-interpretation I,J which is a model of $F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E}$
2. whenever an intended $\Sigma^{\mathcal{L},Horn}$-interpretation I,J is a model of $F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E}$, then $\langle\!\langle F_{\mathcal{H}} \rangle\!\rangle_I$ is a solution from $\mathcal{H}$.

Proof. We prove both directions separately:

1. Suppose that H is a solution. Then there must be I such that $\langle\!\langle F_{\mathcal{H}} \rangle\!\rangle_I = H$. By Definition 4.2.31, this means that $\langle\!\langle F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \rangle\!\rangle_I^{Horn} = \mathcal{BK} \cup H \cup \mathcal{E}$. As H is a solution, $\mathcal{BK} \cup H \models \mathcal{E}$ which means that $J = LM(\mathcal{BK} \cup H) \in \mathcal{T}_{LM(\mathcal{BK} \cup H)}$ is the least model of $\mathcal{BK} \cup H \cup \mathcal{E}$. Hence the combined interpretation I,J is a model $F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E}$.
2. Suppose that I,J is an (intended) model of $F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E}$. Then I must model $F_{\mathcal{H}}$ and hence there must be H s.t. $\langle\!\langle F_{\mathcal{H}} \rangle\!\rangle_I = H$. We then have that $\langle\!\langle F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \rangle\!\rangle_I^{Horn}$ maps to $\mathcal{BK} \cup H \cup \mathcal{E}$. The intended theory for that formula is $\mathcal{T}_{LM(\mathcal{BK} \cup H)} = \{ LM(\mathcal{BK} \cup H) \}$ which means that $J = LM(\mathcal{BK} \cup H)$ must be the case. As the least model of $\mathcal{BK} \cup H$ models $\mathcal{E}$ as well, we conclude that H must be a solution.

□

From these results, it is clear that Learning From Failures is essentially the Satisfiability Modulo Theories approach to Inductive Logic Programming.

Example 4.2.35. In Example 4.2.21, we followed a synthesis-as-rule-selection approach and introduced formula F_1 whose models represent a hypothesis space (which contains $\mathcal{H}_1$ from Example 4.1.1). We can use this hypothesis space in a LFF-problem $(F_1, \mathcal{BK}_1, \mathcal{E}_1, \emptyset)$ where we can, for example, take $\mathcal{E}_1 = \mathcal{E}_1^+ \cup \neg\mathcal{E}_1^-$ from Example 4.1.1. We make $\mathcal{BK}_1$ concrete by giving definitions for the relevant predicates, e.g. tail/2. See the *Horn*-formulas marked $\mathcal{BK}_1$ in Figure 4.2.

We can then write down the full formula $F_1 \wedge \mathcal{BK}_1 \wedge \mathcal{E}_1$ of the SMT-problem that represents LFF-problem $(F_1, \mathcal{BK}_1, \mathcal{E}_1, \emptyset)$. That is, $F_1 \wedge \mathcal{BK}_1 \wedge \mathcal{E}_1$ is (the conjunction of) the full set of formulas in Figure 4.2. These formulas incorporate *Horn* clauses for

$$
\overbrace{\begin{array}{ll}
\mathcal{E}_1 \left\{ \begin{array}{rcl}
\text{last}([l,a,u,r,a],a) & \leftarrow & \\
\text{last}([p,e,n,e,l,o,p,e],e) & \leftarrow & \\
 & \leftarrow & \text{last}([e,m,m,a],m) \\
 & \leftarrow & \text{last}([j,a,m,e,s],e)
\end{array}\right. \\
\mathcal{BK}_1 \left\{ \begin{array}{rcl}
\forall A,B.\text{head}(A,B) & \leftarrow & A=[B|\_] \\
\forall A,B.\text{tail}(A,B) & \leftarrow & A=[\_|B] \\
\forall A,B,C.\text{append}(A,B,C) & \leftarrow & A=[\,], B=C \\
\forall A,B,C,D.\text{append}(A,B,C) & \leftarrow & A=[H|T], \text{append}(T,B,D), C=[H|D] \\
\forall A,B.\text{reverse}(A,B) & \leftarrow & A=[\,], B=[\,] \\
\forall A,B,C,H,T.\text{reverse}(A,B) & \leftarrow & A{=}[H|T], \text{reverse}(T,C), \text{append}(C,[H],B)
\end{array}\right. \\
F_1 \left\{ (R_1)\vee(R_2)\vee(R_3)\vee(R_4)\vee(R_5)\vee(R_3\wedge R_4)\vee(R_3\wedge R_6)\vee(R_7\wedge R_8)\right. \\
\langle\!\langle F_1\rangle\!\rangle_{-} \left\{ \begin{array}{lrcl}
R_1\rightarrow & (\forall A,B.\text{last}(A,B) & \leftarrow & \text{head}(A,B)) \\
R_2\rightarrow & (\forall A,B.\text{last}(A,B) & \leftarrow & \text{head}(A,B), \text{empty}(A)) \\
R_3\rightarrow & (\forall A,B,C.\text{last}(A,B) & \leftarrow & \text{tail}(A,C), \text{head}(C,B)) \\
R_4\rightarrow & (\forall A,B,C.\text{last}(A,B) & \leftarrow & \text{reverse}(A,C), \text{head}(C,B)) \\
R_5\rightarrow & (\forall A,B,C.\text{last}(A,B) & \leftarrow & \text{head}(A,B), \text{reverse}(A,C), \text{head}(C,B)) \\
R_6\rightarrow & (\forall A,B,C,D.\text{last}(A,B) & \leftarrow & \text{tail}(A,C), \text{tail}(C,D), \text{head}(D,B)) \\
R_7\rightarrow & (\forall A,B,C,D.\text{last}(A,B) & \leftarrow & \text{reverse}(A,C), \text{tail}(C,D), \text{head}(D,B)) \\
R_8\rightarrow & (\forall A,B,C,D.\text{last}(A,B) & \leftarrow & \text{tail}(A,C), \text{reverse}(C,D), \text{head}(D,B))
\end{array}\right.
\end{array}}^{F_1 \wedge \mathcal{BK}_1 \wedge \mathcal{E}_1 \,=}
$$

Figure 4.2: The conjunction of all these formulas is $F_1 \wedge \mathcal{BK}_1 \wedge \mathcal{E}_1$. Note that this formula is more than just the conjunction of F_1 and $\mathcal{BK}_1$ and $\mathcal{E}_1$. This is because $F_1 \wedge \mathcal{BK}_1 \wedge \mathcal{E}_1$ *represents* LFF-problem $(F_1, \mathcal{BK}_1, \mathcal{E}_1, \emptyset)$, which requires that $F_1 \wedge \mathcal{BK}_1 \wedge \mathcal{E}_1$ incorporates the mapping from Propositional Logic assignments to *Horn*-formulas, here indicated with $\langle\!\langle F_1\rangle\!\rangle_{-}$.

$\mathcal{E}_1$ and $\mathcal{BK}_1$, the formulas that must hold (in each Horn model) no matter the choice of hypothesis. It also incorporates formula F_1 which represents the constraint problem for the viable hypotheses. And, finally, the thing that makes $F_1 \wedge \mathcal{BK}_1 \wedge \mathcal{E}_1$ represent $(F_1, \mathcal{BK}_1, \mathcal{E}_1, \emptyset)$ is that it *reifies* the mapping from interpretations of F_1 to Horn clauses we wrote down in Table 4.1, i.e. it has formulas which ensure the right *Horn*-formulas get enabled based on the assignment of the propositional variables. In Figure 4.2, these are the formulas marked with $\langle\!\langle F_1 \rangle\!\rangle_{\_}$.

Any model of $F_1 \wedge \mathcal{BK}_1 \wedge \mathcal{E}_1$ must consist of an interpretation I for the propositional variables (R_1, R_2, etc.) as well as an interpretation J for the predicate symbols head/2, tail/2, append/3, reverse/2 and last/2 from the *Horn* fragment. It can be checked that if

$$I_{h_4} = \{ R_1 \mapsto \bot, R_2 \mapsto \bot, R_3 \mapsto \bot, R_4 \mapsto \top, R_5 \mapsto \bot, R_6 \mapsto \bot, R_7 \mapsto \bot, R_8 \mapsto \bot \}$$

and $J_{h_4} = LM(\mathcal{BK} \cup \{ (\forall A, B, C. \text{last}(A, B) \leftarrow \text{reverse}(A, C), \text{head}(C, B)) \})$ then I_{h_4}, J_{h_4} is a model of $F_1 \wedge \mathcal{BK}_1 \wedge \mathcal{E}_1$.

This example demonstrates that the many synthesis-as-rule-selection approaches – e.g. ASPAL [26], ASPSynth [26], ProSynth [139], ILASP [95] – can all be seen as SMT-problems with a common approach to encoding how formulas represent hypothesis spaces. We will show that the Learning From Failures framework allows for other ways of representing of hypothesis spaces, in particular the synthesis-as-literal-selection encoding of Section 4.5.

Having examined how to represent the search for solutions within the whole set of viable hypotheses, we now turn to the question of how we can iteratively reduce the set of viable hypotheses through constraints.

4.3 Learning sound constraints

Given a LFF-input $(F_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \mathcal{C})$, we can obtain a currently viable hypothesis H by obtaining a model of formula $F_{\mathcal{H}} \wedge \mathcal{C}$. If we then simply test H against the examples, we can establish whether H fails (i.e. is a solution or not). Based on whether H entailed too little or entailed too much, we can reason that all hypotheses that entail less than H or more than H, respectively, must also fail. We would hence like to identify those hypotheses of $\mathcal{H}$ that entail H, i.e. entail at least as much, or are entailed by H, i.e. entail at most as much. However, as checking entailment between clauses (and hence programs) is in general undecidable, this is not tractable [24].

4.3.1 Generalisations and specialisations

Instead of trying to derive all programs that are entailed by another, we turn to a strictly weaker relation, using a syntactic condition on when a hypothesis is more specific – or, vice versa, more general – than another: θ-subsumption, or *subsumption* for short [129].

Definition 4.3.1. A clause C_1 *subsumes* a clause C_2 if and only if there exists a substitution θ such that $C_1\theta \subseteq C_2$.

Example 4.3.2. Let C_1 and C_2 be clauses:

$$C_1 = f(A, B) \leftarrow \text{head}(A, B). \qquad C_2 = f(X, Y) \leftarrow \text{head}(X, Y), \text{odd}(Y).$$

Then C_1 subsumes C_2 because $C_1\theta \subseteq C_2$ with $\theta = \{A \mapsto X, Y \mapsto B\}$.

We have that if clause C_1 subsumes clause C_2 then C_1 entails C_2 and that, in general checking entailment between two (first-order) clauses is an NP-complete decision problem, c.f. [122].

Most of the results in this section apply to sets of arbitrary first-order clauses. While the usual logic programming terminology for *sets of* first-order clauses is clausal theory, in keeping with SMT-terminology, we reserve the term *theory* for sets of interpretations. Instead we generalize our notion of program from Definition 4.2.1 to:

Definition 4.3.3. A *clausal program* is a set of first-order clauses.

As we are interested in approximating entailment with respect to (*Horn*/definite) programs, we follow Midelfart [109] in extending subsumption to clausal programs:

Definition 4.3.4. A clausal program H_1 subsumes a clausal program H_2, denoted $H_1 \preceq H_2$, if and only if $\forall C_2 \in H_2, \exists C_1 \in H_1$ such that C_1 subsumes C_2.

Example 4.3.5. Let H_1, H_2, and H_3 be programs:

$$H_1 = \left\{ \begin{array}{l} \mathrm{f}(A,B) \leftarrow \mathrm{head}(A,B) \end{array} \right\}$$
$$H_2 = \left\{ \begin{array}{l} \mathrm{f}(A,B) \leftarrow \mathrm{head}(A,B), \mathrm{odd}(B) \end{array} \right\}$$
$$H_3 = \left\{ \begin{array}{l} \mathrm{f}(A,B) \leftarrow \mathrm{head}(A,B) \\ \mathrm{f}(A,B) \leftarrow \mathrm{reverse}(A,C), \mathrm{head}(C,B) \end{array} \right\}$$

Then $H_1 \preceq H_2$, $H_3 \preceq H_1$, and $H_3 \preceq H_2$.

Like entailment, *theory* subsumption is a reflexive and transative relation. Theory subsumption is anti-symmetric in the sense that if $H_1 \preceq H_2$ and $H_2 \preceq H_1$, then H_1 and H_2 are *variants* of one another, i.e. only distinguished through variable renaming, and are semantically equivalent. Like clausal subsumption, theory subsumption implies entailment:

Proposition 4.3.6. Given clausal programs H_1 and H_2, whenever $H_1 \preceq H_2$ then $H_1 \models H_2$.

Proof. Follows from the definitions of clausal subsumption (Definition 4.3.1) and clausal program subsumption (Definition 4.3.4). □

We use subsumption to define *generalisations* and *specialisations*:

Definition 4.3.7. A clausal program T_1 is a *generalisation* of a clausal program T_2 if and only if $T_1 \preceq T_2$.

Definition 4.3.8. A clausal program T_1 is a *specialisation* of a clausal program T_2 if and only if $T_2 \preceq T_1$.

In the next section, we define sound hypothesis constraints in terms of subsumption.

4.3.2 Learning constraints from failures

Suppose $H \in \mathcal{H}$ was tested and found to be incomplete and/or inconsistent, i.e. H failed. Based on the type of failure, we can reason by subsumption about which other programs must fail as well. In case that H is incomplete, i.e. did not entail a positive example, we say H is *too specific*. Similarly, if H is inconsistent, i.e. did entail a negative example, we say H is *too general*. We now introduce *generalisation* and *specialisation* constraints – valid for clausal programs in general – for pruning programs that are *more general* and *more specific*, respectively, than a given program. Subsequently we zoom in on pruning based on the case that H is totally incomplete. For this case we introduce *redundancy constraints* capable of pruning non-optimal hypotheses not pruned by specialisation and generalisation constraints.

Pruning generalisations

Let $H \in \mathcal{H}$ be any hypothesis. Then, with respect to the negative examples, $\mathcal{E}^-$, testing H has one of the following outcomes:

Outcome	**Description**	**Formula**
$\mathcal{E}^-_{\mathbf{none}}$	H is consistent, i.e. H entails no negative example	$\forall e \in \mathcal{E}^-.\ H \cup \mathcal{BK} \not\models e$
$\mathcal{E}^-_{\mathbf{some}}$	H is inconsistent, i.e. H entails a negative example	$\exists e \in \mathcal{E}^-.\ H \cup \mathcal{BK} \models e$

If the outcome is $\mathcal{E}^-_{\mathbf{none}}$, i.e. H is consistent, we learn nothing regarding failing hypotheses in $\mathcal{H}$. In case the outcome is $\mathcal{E}^-_{\mathbf{some}}$, however, we can reason that any (clausal) program that is at least as general as H will also entail the negative examples that H entailed. Hence we would like to prune hypotheses that are more general than H:

Definition 4.3.9. A *generalisation constraint* for clausal program H only prunes generalisations of H from the hypothesis space.

Example 4.3.10. Suppose we have hypothesis H and we test it on negative examples $\mathcal{E}^-$:

$$\mathtt{h} = \{\ \text{last}(A,B) \leftarrow \text{head}(A,B)\ \} \qquad \mathcal{E}^- = \{\ \text{last}([a,l,i,c,e],a)\ \}$$

Hypothesis H is too general as it entailed a negative example. Hence we can prune generalisations of it, such as H_1 and H_2:

$$H_1 = \left\{ \begin{array}{l} \text{last}(A,B) \leftarrow \text{head}(A,B) \\ \text{last}(A,B) \leftarrow \text{tail}(A,C), \text{head}(C,B) \end{array} \right\}$$
$$H_2 = \left\{ \begin{array}{l} \text{last}(A,B) \leftarrow \text{head}(A,B) \\ \text{last}(A,B) \leftarrow \text{tail}(A,C), \text{head}(C,B), \text{head}(A,B) \end{array} \right\}$$

We show that pruning generalisations of an inconsistent hypothesis is *sound*:

Proposition 4.3.11. Given LFF input $(E^+_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \mathcal{C})$ and inconsistent hypothesis $H \in \mathcal{H}$, for any generalisation constraint C_{gen} for H, whenever $\mathcal{C}$ are sound constraints, $\mathcal{C} \cup \{\ C_{gen}\ \}$ are sound constraints.

Proof. Hypothesis constraint C_{gen} only prunes models from $E^+_{\mathcal{H}}$ which represent hypotheses that are at least as general as H. Each such hypothesis fails as it entails those negative examples that H already entailed. Hence C_{gen} only prunes non-solutions. □

Pruning specialisations

Now with respect to the positive examples $\mathcal{E}^+$, the outcome of testing hypothesis $H \in \mathcal{H}$ is one (or two, in case of total incompleteness) of the following:

Outcome	Description	Formula
$\mathcal{E}^+_{\mathbf{all}}$	H is complete, i.e. H entails all positive examples	$\forall e \in \mathcal{E}^+,\ H \cup \mathcal{BK} \models e$
$\mathcal{E}^+_{\mathbf{some}}$	H is incomplete, i.e. H does entail all positive examples	$\exists e \in \mathcal{E}^+,\ H \cup \mathcal{BK} \not\models e$
$\mathcal{E}^+_{\mathbf{none}}$	H is totally incomplete, i.e. H entails no positive examples	$\forall e \in \mathcal{E}^+,\ H \cup \mathcal{BK} \not\models e$

If the outcome is $\mathcal{E}^+_{\mathbf{all}}$, i.e. H is complete, we learn nothing regarding failing hypotheses in $\mathcal{H}$. In case the outcome is $\mathcal{E}^+_{\mathbf{some}}$, i.e. H is incomplete, however, we can reason that any program that is at least as specific as H will also not entail H's non-entailed positive examples. Hence we would like to prune hypotheses that are more specific than H:

Definition 4.3.12. A *specialisation constraint* for clausal program H only prunes specialisations of H from the hypothesis space.

Example 4.3.13. Suppose we have hypothesis H and we test it on positive examples $\mathcal{E}^+$:

$$H = \left\{ \begin{array}{l} \text{last}(A, B) \leftarrow \text{head}(A, B) \end{array} \right\} \qquad \mathcal{E}^+ = \left\{ \begin{array}{l} \text{last}([b, o, b], b) \\ \text{last}([a, l, i, c, e], e) \end{array} \right\}$$

We have that H is too specific as it does not entail the second example. We can therefore prune specialisations of H, such as H_1 and H_2:

$$H_1 = \left\{ \begin{array}{l} \text{last}(A, B) \leftarrow \text{head}(A, B), \text{empty}(A) \end{array} \right\}$$
$$H_2 = \left\{ \begin{array}{l} \text{last}(A, B) \leftarrow \text{head}(A, B), \text{tail}(A, C) \end{array} \right\}$$

We show that pruning specialisations of an incomplete hypothesis is sound:

Proposition 4.3.14. Given LFF-input $(E_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \mathcal{C})$ and incomplete hypothesis $H \in \mathcal{H}$, for any specialisation constraint C_{spec} for H, whenever $\mathcal{C}$ are sound constraints, $\mathcal{C} \cup \{ C_{spec} \}$ are sound constraints.

Proof. Hypothesis constraint C_{spec} only prunes models from $E_{\mathcal{H}}$ which represent hypotheses that are more specific than H. Each such hypothesis fails as it does not entail those positive examples that H already failed to entail. Hence C_{spec} only prunes non-solutions. □

4.3.3 Pruning redundancies

We now consider the case where testing hypothesis $H \in \mathcal{H}$ on positive examples got us outcome $\mathcal{E}^+_{\mathbf{none}}$, i.e. H is totally incomplete. As H is incomplete we are allowed to prune based on specialisation constraints. However, in addition, we can reason that extending program H leads to redundancy in many cases as H itself does not (directly) contribute to entailing positive training examples. Hence, if a specialisation of H gets included in a hypothesis H', the H-part of H' can often be removed, i.e. it is redundant. This yields a smaller program that entails as many of the positive examples as H' did. If

Assumption 4.2.16 holds, this smaller program is a hypothesis too, which then means that H' cannot be an optimal solution.

Based on the notion of redundancy we develop a kind of constraint that soundly prunes hypotheses which cannot be optimal solutions. We characterise redundancy within hypotheses by how clauses depend on one another[5]. We adapt the standard notion of a dependency graph on the predicates of a program [5] to the program's clauses:

Definition 4.3.15. Given a program P, the *dependency graph* $\mathcal{D}_P$ of P is a directed graph on the clauses of P with an edge from clause $C_1 \in P$ to $C_2 \in P$ if the predicate symbol of the head literal of C_2 occurs in the body of C_1.

We say a clause $C_1 \in P$ *depends on* $C_2 \in P$ if there is a path from C_1 to C_2 in $\mathcal{D}_P$. Based on this notion, we introduce notation to talk about all clauses reachable from and reaching a particular clause within a program's dependency graph:

Definition 4.3.16. Given a program P, its dependency graph $\mathcal{D}_P$ and a clause $C \in P$, we denote with

- $\mathcal{D}_P[C, -]$ clause C and all clauses of P that C depends on
- $\mathcal{D}_P[-, C]$ clause C and all clauses of P that depend on C

Note how each $\mathcal{D}_P[C, -]$ and $\mathcal{D}_P[-, C]$ yields back a program, namely a subset of P.

Example 4.3.17. Let H be the following program:

$$H = \left\{ \begin{array}{l} C_1 = \text{last}(A, B) \leftarrow \text{head}(A, B), \text{empty}(A) \\ C_2 = \text{last}(A, B) \leftarrow \text{tail}(A, C), \text{last}(C, B) \\ C_3 = \text{inv}(A, B) \leftarrow \text{head}(A, B), \text{even}(B) \\ C_4 = \text{last}(A, B) \leftarrow \text{reverse}(A, C), \text{inv}(C, B) \end{array} \right\}$$

The following table then denotes how H's clauses depend on one another:

$\mathcal{D}_H[C_1, -]$	$=$	$\{C_1\}$	$\mathcal{D}_H[-, C_1]$	$=$	$\{C_1, C_2\}$
$\mathcal{D}_H[C_2, -]$	$=$	$\{C_1, C_2, C_3, C_4\}$	$\mathcal{D}_H[-, C_2]$	$=$	$\{C_2\}$
$\mathcal{D}_H[C_3, -]$	$=$	$\{C_3\}$	$\mathcal{D}_H[-, C_3]$	$=$	$\{C_3, C_4\}$
$\mathcal{D}_H[C_4, -]$	$=$	$\{C_3, C_4\}$	$\mathcal{D}_H[-, C_4]$	$=$	$\{C_2, C_4\}$

We now capture what it means for a program to be partially specialised by another program:

Definition 4.3.18. Given programs P and Q and a clause $C \in Q$, clause C *P-specialises* Q if and only if $P \preceq \mathcal{D}_Q[C, -]$ and $P \preceq \mathcal{D}_Q[-, C]$.

The idea is that the clause C of Q cannot contribute to Q entailing more than P, as the clauses of Q that depend on C, as well as those that C depends on, all specialise P. If P is a totally incomplete w.r.t. $\mathcal{E}^+$, we know such a clause C is redundant in Q w.r.t. $\mathcal{E}^+$.

Proposition 4.3.19. Let P and Q be programs s.t. there is $C \in Q$ that P-specialises Q. If for each $e^+ \in \mathcal{E}^+$ we have $P \not\models e^+$ and $Q \models e^+$, then $Q \setminus \{ C \} \models \mathcal{E}^+$.

[5]The original paper on Learning From Failures [35] only considered *separable hypotheses* where clauses could not depend on each other at all. Redundancy constraints properly contain the elimination constrains that we defined for totally incomplete separable hypotheses. The definition of redundancy constraints in this chapter is an improved version of my original presentation of them [36].

Proof. Let $Q_C = \mathcal{D}_Q[C, -] \cup \mathcal{D}_Q[-, C]$. As $P \preceq \mathcal{D}_Q[C, -]$ and $P \preceq \mathcal{D}_Q[-, C]$, we have $P \preceq Q_C$ and hence $Q_C \not\models e^+$ for each $e^+ \in \mathcal{E}^+$. For each e^+, there must be $Q_{e^+} \subseteq Q$ s.t. $Q_{e^+} \models e^+$ and at least one $D \in Q_{e^+}$ not in Q_C, as otherwise Q_{e^+} would specialise Q_C and not entail e^+. But not one of the clauses necessary for e^+'s entailment, i.e. $Q_{e^+} \setminus Q_C$, depends on C. Hence, if present, C can always be removed from Q_{e^+} and $Q_{e^+} \setminus \{ C \} \models e^+$ would still hold. As this holds for each $e^+ \in \mathcal{E}^+$, conclude that C is redundant in Q w.r.t. $\mathcal{E}^+$. □

Example 4.3.20. Let H_1 be the following program:

$$\left\{ \begin{array}{l} C_1 = \text{last}(A, B) \leftarrow \text{head}(A, B), \text{empty}(A) \\ C_2 = \text{last}(A, B) \leftarrow \text{tail}(A, C), \text{last}(C, B) \end{array} \right\}$$

With respect to H from Example 4.3.17, clause C_1 H_1-specialises H. If our positive examples are $\mathcal{E}_1^+ = \{ \text{last}([1, 2], 2), \text{last}([2, 4, 8, 16], 16) \}$, then H_1 is totally incomplete. We indeed see that while H entails the positive examples, H_1's clauses are not needed for that to be the case.

We show that if a hypothesis contains a redundant clause, that hypothesis is not optimal:

Proposition 4.3.21. Let $(\mathcal{H}, \mathcal{BK}, \mathcal{E}^+ \cup \neg\mathcal{E}^-)$ be a LFE input, $H_1, H_2 \in \mathcal{H}$, and H_1 be totally incomplete. If H_2 has a clause C that H_1-specialises H_2, then H_2 is not an optimal solution.

Proof. If we instantiate Proposition 4.3.19's P and Q with H_1 and H_2, we obtain that $C \in H_2$ is redundant w.r.t $\mathcal{E}^+$. Hence whenever H_2 is complete, $H_2 \setminus \{ C \}$ is complete. As our hypotheses consist of definite clauses, removing a clause never adds consequences. Therefore if H_2 did not entail negative examples, neither does $H_2 \setminus \{ C \}$. Hence if Q is complete and consistent, then program $Q \setminus \{ C \}$ is complete and consistent as well. By Assumption 4.2.16, if H_2 is a solution, program $H_2 \setminus \{ C \}$ must be a hypothesis and hence would be a smaller solution than H_2. □

If hypothesis H_1 is totally incomplete and there is a clause that H_1-specialises hypothesis H_2, we call H_2 a *redundant hypothesis*. We now introduce a constraint that uses the notion of an H-specialising clause to prune the hypothesis space:

Definition 4.3.22. A *redundancy constraint* for program H only prunes hypotheses from the hypothesis space which contain a H-specialising clause.

We show that pruning hypotheses that are redundant with respect to a clause of a totally incomplete hypothesis is *optimally sound* as only non-optimal hypotheses get pruned.

Proposition 4.3.23. Given LFF input $(E_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \mathcal{C})$ and totally incomplete hypothesis $H \in \mathcal{H}$, for any redundancy constraint C_{red} for H, whenever $\mathcal{C}$ are optimally sound constraints, $\mathcal{C} \cup \{ C_{red} \}$ are optimally sound.

Proof. Hypothesis constraint C_{red} only prunes models from $E_{\mathcal{H}}$ which represent hypotheses that contain a H-specialising clause. Each such hypothesis is non-optimal by Proposition 4.3.21. Hence C_{red} only prunes non-optimal hypotheses. □

Summary of constraints

In summary, for each combination of different outcomes there is a different combination of constraints that apply, see Table 4.2. Note that as the redundancy constraint allows for strictly more pruning than the specialisation constraint, it is possible to elide using a specialisation constraint for outcome $\mathcal{E}^+_{\mathbf{none}}$.

Outcome	$\mathcal{E}^-_{\mathbf{none}}$	$\mathcal{E}^-_{\mathbf{some}}$
$\mathcal{E}^+_{\mathbf{all}}$	n/a	Generalisation
$\mathcal{E}^+_{\mathbf{some}}$	Specialisation	Specialisation, Generalisation
$\mathcal{E}^+_{\mathbf{none}}$	Specialisation, Redundancy	Specialisation, Redundancy, Generalisation

Table 4.2: The constraints we can learn from the outcome of testing a hypothesis. The outcome of $\mathcal{E}^+_{\mathbf{all}}$ and $\mathcal{E}^-_{\mathbf{none}}$ corresponds to the hypothesis being a solution.

4.4 Learning From Failures by a Three-stage Loop

We now turn our attention to *solving* LFF problems. We base our approach on how Satisfiability Modulo Theories problems are solved.

As discussed in Section 3.3, SMT-solvers have two major components: a SAT-solver, typically for propositional logic, and background theory solvers for fragments of first-order logic. See Figure 4.3. The process of solving involves the SAT-solver making guesses for propositional assignments and informing a solver of first-order formulas that it should check for satisfiability w.r.t. to a theory, i.e. a set of interpretations $\mathcal{T}$. When a (propositional) guess leads to that the asserted first-order formulas together are unsatisfiable, the $\mathcal{T}$-solver is responsible for injecting a *conflict*, i.e. a formula – interpreted as being propositional by the SAT-solver – which constrains the SAT-problem in such a way as to rule out the current guess and possibly related ones.

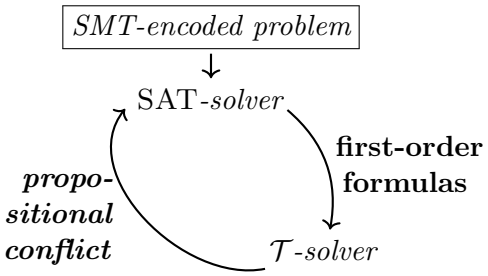


Figure 4.3: Solving SMT-encoded problems by a (propositional logic) SAT-solver communicating with a background theory solver which checks the satisfiability of first-order formulas with respect to a theory $\mathcal{T}$.

We see this process of gradually constraining the original satisfaction problem more and more – learning conflict clauses from guesses that did not work out – as a process of refinement. We can apply this refinement methodology to Learning From Failures.

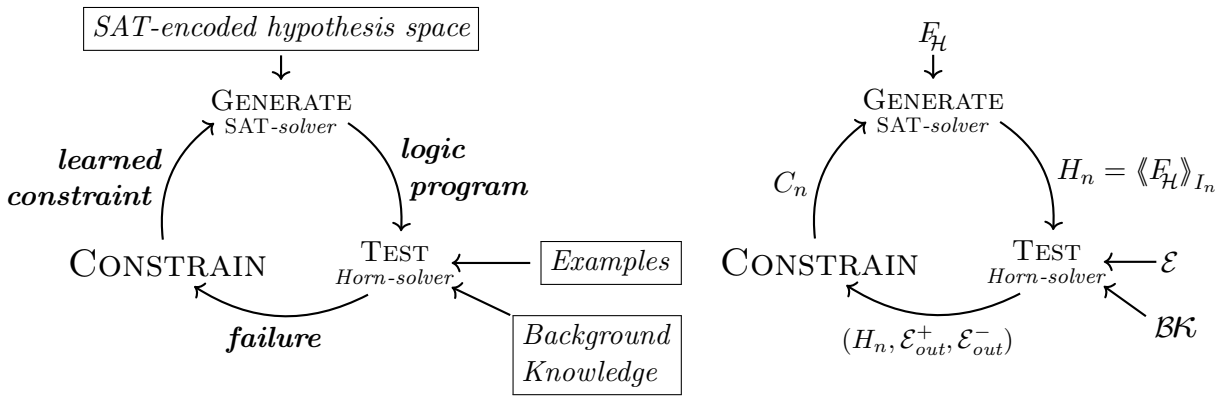


Figure 4.4: The generate, test, and constrain loop. $E_{\mathcal{H}}$ encodes the hypothesis space. Interpretation I_n is a model of $E_{\mathcal{H}} \wedge \mathcal{C}_{0..n-1}$ which represents the nth logic program to test: $\langle\!\langle E_{\mathcal{H}} \rangle\!\rangle_{I_n}$. $\mathcal{E}$ are the positive and negative examples. $\mathcal{BK}$ is the background knowledge, a Horn program fixing an interpretation for predicates that are allowed to be used in hypotheses. $\mathcal{E}^+_{out}$ and $\mathcal{E}^-_{out}$ constitute the outcome of testing $\langle\!\langle E_{\mathcal{H}} \rangle\!\rangle_{I_n}$, on the positive and negative examples, respectively. C_n is the constraint derived due to the failure of $\langle\!\langle E_{\mathcal{H}} \rangle\!\rangle_{I_n}$, which gets added on to the set $\mathcal{C}_{0..n-1}$.

4.4.1 Refining the hypothesis space

A natural way to address the Learning From Entailment problem through Learning From Failures is by gradually expanding the set of constraints. As we accumulate more and more hypothesis constraints, we prune more and more models of the formula $E_{\mathcal{H}}$. If each pruned model represented a non-solution, this process leads to a gradual *refining* of the set of viable hypotheses.

With this in mind, to solve a learning from entailment problem $(\mathcal{H}, \mathcal{BK}, \mathcal{E})$, we can set set up a Learning From Failures problem $(E_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \mathcal{C}_0)$ starting out with an empty set of constraints[6], i.e. $\mathcal{C}_0 = \emptyset$. If we manage to keep discovering new hypothesis constraints, each strictly reducing the size of the set of viable hypotheses, eventually all non-solutions will get pruned away. We can model this as a sequence of Learning From Failures problems: $(E_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \mathcal{C}_0 = \emptyset)$, $(E_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \mathcal{C}_{0..1} = \mathcal{C}_0 \cup \{ C_1 \})$, …, $(E_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \mathcal{C}_{0..n} = \mathcal{C}_{0..n-1} \cup \{ C_n \})$, … If each $E_{\mathcal{H}} \wedge \mathcal{C}_{0...n+1}$ strictly reduces the set of viable hypotheses with respect to $E_{\mathcal{H}} \wedge \mathcal{C}_{0...n}$, then, in the limit, the only viable hypotheses that remain are solutions.

We now turn to a three-stage procedure that discovers hypothesis constraints guaranteed to soundly prune currently viable hypotheses.

4.4.2 The three-stage loop

As in Section 4.1, we propose a three-stage iterative procedure for systematically searching for a solution to a Learning From Failures problem:

[6] An alternative is that $\mathcal{C}_0$, the initial set of (sound) constraints, is derived through some mechanism not described in this document.

Generate Find an assignment of the formula representing the hypothesis space that is consistent with the current set of hypothesis constraints.

Test Interpret the guessed assignment as a hypothesis and test it – with the Background Knowledge providing definitions for predicates – against our examples. The guessed hypothesis fails if we observe it to be inconsistent or incomplete.

Constrain Based on the outcome of testing, derive which hypotheses can be soundly pruned from the hypothesis space. Accomplish this pruning by adding appropriate hypothesis constraints to our current set of constraints.

Upon completing the third stage, we go back to the first, now with the set of viable hypotheses suitably restricted. This means the stages occur in a loop, as illustrated in Figure 4.4.

Note that each iteration prunes at least one hypothesis from the hypothesis space: the just tested hypothesis went from being viable to no longer being in the set of viable hypotheses. Hence, given a finite hypothesis space and sound hypothesis constraints, we necessarily narrow down the set of viable hypotheses to just the (optimal) solutions.

We now formalize the three stages of the loop and make the correspondence with SMT-solving precise. In the way we set up the stages, the nth iteration of the loop addresses LFF problem $(F_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \mathcal{C}_{0..n})$.

4.4.3 Generate stage as a satisfiability solver

The generate stage takes as input an encoding of the hypothesis space, $F_{\mathcal{H}}$, and a set of hypothesis constraints, C_n. With each iteration of the loop, it accumulates more and more hypothesis constraints. In particular, the generate stage maintains the set $\mathcal{C}_{0..n}$. We set $\mathcal{C}_{0..0} = \mathcal{C}_0$. Upon the nth iteration completing, $\mathcal{C}_{0..n}$ gets updated: $\mathcal{C}_{0..n} = \mathcal{C}_{0..n-1} \cup \{ C_n \}$.

The task of the generate stage is to find a model of $F_{\mathcal{H}} \wedge \mathcal{C}_{0..n}$. Hence we need a satisfiability solver for the constraint problem language $\mathcal{L}$ that $F_{\mathcal{H}} \wedge \mathcal{C}_{0..n}$ is written in. We require of this solver that when a (full) interpretation I that models $F_{\mathcal{H}} \wedge \mathcal{C}_{0..n}$ has been found, the program represented by the interpretation, $\langle\!\langle F_{\mathcal{H}} \wedge \mathcal{C}_{0..n} \rangle\!\rangle_I = \langle\!\langle F_{\mathcal{H}} \rangle\!\rangle_I$, gets handed over to a separate component, the test stage, for checking.

4.4.4 Test stage: Background Knowledge as Background Theory

The test stage takes three inputs: a program representing the background knowledge, $\mathcal{BK}$, a set of literals $\mathcal{E} = \mathcal{E}^+ \cup \neg\mathcal{E}^-$ representing positive and negative examples, and a hypothesis to test, H. In this document, we take $\mathcal{E}$ and $\mathcal{BK}$ to be static, that is they do not change through-out the solving of a LFF problem.

The task of the test stage is to take the program (that a model I represents) found by the generate stage and check if it is a solution or not. Per Proposition 4.2.34, this is equivalently expressed as either checking that (1) $\forall e^+ \in \mathcal{E}^+ . \mathcal{BK} \cup \langle\!\langle F_{\mathcal{H}} \rangle\!\rangle_I \models e^+$ and $\forall e^- \in \mathcal{E}^- . \mathcal{BK} \cup \langle\!\langle F_{\mathcal{H}} \rangle\!\rangle_I \not\models e^-$, or (2) that there is a *Horn*-interpretation J such that I,J is a model of $F_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E}$. From the latter perspective, we have that the Background Knowledge serves to fix a *Horn*-background theory.

In case $\langle\!\langle E_{\mathcal{H}} \rangle\!\rangle_I$ is not a solution, this means that either $\langle\!\langle E_{\mathcal{H}} \rangle\!\rangle_I$ was incomplete or inconsistent or both. Upon determining this test outcome, we require the test stage to produce a tuple $(\langle\!\langle E_{\mathcal{H}} \rangle\!\rangle_I, \mathcal{E}^+_{out}, \mathcal{E}^-_{out})$ where $\mathcal{E}^+_{out} \in \{\, \mathcal{E}^+_{\mathbf{none}}, \mathcal{E}^+_{\mathbf{some}}, \mathcal{E}^+_{\mathbf{all}} \,\}$ and $\mathcal{E}^-_{out} \in \{\, \mathcal{E}^-_{\mathbf{none}}, \mathcal{E}^-_{\mathbf{some}} \,\}$ are the outcome markers from Table 4.2 indicating how many of the positive and negative examples were entailed. This tuple then gets handed off to the constrain stage.

4.4.5 Constrain stage as conflict generalisation

In the SMT-paradigm, if a background theory solver determines that its first-order formulas are not satisfiable together, a propositional conflict gets generated to inform the SAT-solver to prune the guess it has made, as well as assignments that lead to the (exact) same set of first-order formulas being re-asserted. As discussed in Section 4.3, from a particular program failing we can infer – using θ-subsumption – that a large class of related hypotheses can be pruned. As such, the task of the constrain stage is to derive a constraint formula that soundly prunes hypotheses based on the outcome provided by the test stage.

The input of the constrain stage is hence a tuple $(H_n, \mathcal{E}^+_{out}, \mathcal{E}^-_{out})$ consisting of a program (from the hypothesis space $\mathcal{H}$) and two marker values specifying the outcome of testing.

The output of the nth iteration of the constrain stage is a formula C_n of the fragment $\mathcal{L}$, intended to be a hypothesis constraint w.r.t. $E_{\mathcal{H}}$. If C_n is a specialisation, generalisation and/or redundancy constraint derived from the failure of H_n, in conformance with Table 4.2, then adding C_n to $E_{\mathcal{H}} \wedge \mathcal{C}_{0..n-1}$ *soundly* prunes at least H_n as well as a large class of related programs. Upon deriving output C_n, it is handed to the generate stage.

Thereupon the loop starts over.

4.5 Implementation in three stages: `Popper`

We now present `Popper`, an Inductive Logic Programming system addressing Learning From Failures problems through the three-stage loop from the previous section. We instantiate abstract constraint logic fragment $\mathcal{L}$ with Answer Set Programming. Hence we use an ASP program as the formula representing our hypothesis space. This program demonstrates what we call a *synthesis-as-literal-selection* encoding. We describe the generate, test, and constrain stages in detail. We use ASP's multi-shot solving [59] to maintain state between the three stages, thereby building a kind of SMT-solver *on top of* the Clingo solver [58]. Finally, we prove the soundness and completeness of `Popper`. To facilitate learning optimal solutions, `Popper` searches for programs of increasing size.

4.5.1 Hypothesis space in terms of a declaration bias

We now describe how a *declaration bias* captures the main parameters of the hypothesis space, i.e. the parameters discussed in Section 4.2.2. In Section 4.5.3, we describe how a declaration bias is used to construct the representation of the hypothesis space.

We first decide on the predicates that can occur in the hypotheses. In order to say which predicates are allowed in the heads of clauses, we introduce head declarations.

Definition 4.5.1. A *head declaration* is a ground atom `head_pred(p,a)` where `p` is a predicate symbol of arity `a`.

Similarly, to say which predicates are allowed in the bodies of clauses, we introduce body declarations:

Definition 4.5.2. A *body declaration* is a ground atom of the form `body_pred(p,a)` where `p` is a predicate symbol of arity `a`.

Head declaration and body declarations together make up the *predicate declarations*. As in Section 4.2.2, besides specifying which predicates can occur in hypotheses, there are a couple of natural parameters to bound the hypothesis space:

- N_V, the maximum number of (distinct) variables allowed in a clause
- N_B, the maximum number of (distinct) literals in the body of a clause
- N_C, the maximum number of clauses allowed in a hypothesis

We collect all these parameters into one structure, a declaration bias:

Definition 4.5.3. A *declaration bias* D is a tuple, consisting of ground atoms,

$$(D_h, D_b, \texttt{max\_vars}(N_V), \texttt{max\_body}(N_B), \texttt{max\_clauses}(N_C))$$

where D_h is a set of head declarations, D_b a set of body declarations, N_V is the maximum number of distinct variables in a clause, N_B is the maximum number of literals in the body of a clause, N_C is the maximum number of clauses allowed in a hypothesis.

We define a *declaration consistent* clause:

Definition 4.5.4. Let $D = (D_h, D_b, \texttt{max\_vars}(N_V), \texttt{max\_body}(N_B), \texttt{max\_clauses}(N_C))$ be a declaration bias and $C = h \leftarrow b_1, b_2, \ldots, b_n$ be a definite clause. Then C is *declaration consistent* with D if and only if:

- h is an atom of the form $p(V_1, \ldots, V_m)$ and `head_pred(p,m)` is in D_h
- every b_i is a literal of the form $p(V_1, \ldots, V_m)$ and `body_pred(p,m)` is in D_b
- every V_i is a first-order variable
- $n \leq N_B$, i.e. the number of distinct body literals is at most the maximum allowed
- $\left| \{ V_i \mid p(V_1, \ldots, V_m) = h \vee p(V_1, \ldots, V_m) = b_j \text{ for } 1 \leq j \leq n \} \right| \leq N_V$, i.e. the number of distinct variables is at most the maximum allowed

Example 4.5.5. Let D be the declaration bias:

$$(\{\texttt{head\_pred(target,2)}\}, \{\texttt{body\_pred(head,2)}, \texttt{body\_pred(tail,2)}\}, \\ \texttt{max\_vars(3)}, \texttt{max\_body}(2), \texttt{max\_clauses(2)})$$

Then the following clauses are all consistent with D:

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

By contrast, the following clauses are inconsistent with D:

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

We define which programs are consistent with a declaration bias:

Definition 4.5.6. Given a declaration bias $D = (D_h, D_b, \texttt{max\_vars}(N_V), \texttt{max\_body}(N_B), \texttt{max\_clauses}(N_C))$, a program H is *declaration consistent* if each $C \in H$ is declaration consistent with D and $|H| \leq N_C$, i.e. H has at most N_C clauses.

Example 4.5.7. Let D be the declaration bias from the previous example. Then two declaration consistent hypotheses are:

$$\texttt{h}_1 : \left\{ \texttt{target(A,B):- head(A,B)} \right\}$$
$$\texttt{h}_2 : \left\{ \begin{array}{l} \texttt{target(A,B):- head(A,B).} \\ \texttt{target(A,B):- tail(A,C),head(C,B).} \end{array} \right\}$$

Example 4.5.8. Let D be the declaration bias:

$$\begin{array}{l} (\{\texttt{head\_pred(target,2)}, \texttt{head\_pred(inv,2)}\}, \\ \{\texttt{body\_pred(head,2)}, \texttt{body\_pred(tail,2)}, \texttt{body\_pred(empty,1)}, \\ \texttt{head\_pred(target,2)}, \texttt{body\_pred(inv,2)}\}, \\ \texttt{max\_vars(3)}, \texttt{max\_body}(3), \texttt{max\_clauses(3)}) \end{array}$$

Then the following programs are all consistent with D:

$$\texttt{h}_1 : \left\{ \texttt{target(A,B):- tail(A,C),target(C,B)target(A,B):- head(A,B)} \right\}$$
$$\texttt{h}_2 : \left\{ \begin{array}{l} \texttt{target(A,B) :- tail(A,C),inv(C,B).} \\ \texttt{inv(A,B) :- tail(A,C),inv(C,B).} \\ \texttt{inv(A,B) :- tail(A,B),tail(B,C),empty(C).} \end{array} \right\}$$

Note that the set of programs described by a declaration bias is finite. Hence when we use a declaration bias to describe a hypothesis space, this hypothesis space is finite.

4.5.2 Popper's loop: persisting state through multi-shot solving

Algorithm 1 contains high-level code for implementing the three-stage loop. The procedure takes arguments for the background knowledge ($\mathcal{BK}$), examples ($\mathcal{E} = \mathcal{E}^+ \cup \neg\mathcal{E}^-$), and a declaration bias (`declaration_bias`) from which the generate stage automatically derives the representation of the hypothesis space ($F_{\mathcal{H}}$). The argument `constraints` acts as a variable capturing the set of constraints $\mathcal{C}_{0..n}$. This set gets expanded with a new (set of) constraint(s) (C_n) during the nth iteration of the loop. The final argument, `max_literals`, acts as an additional (size) bound on hypotheses, now on the total number of literals in a hypothesis. This parameter can be set to an arbitrarily high number.

By iteratively increasing the total number of allowed literals (`num_literals`) in the hypotheses, we guarantee we find an optimal solution if one exists.

Emulating SMT-solving through multi-shot solving

A naive implementation of Algorithm 1, such as performing iterative deepening on the program size (`num_literals`), would duplicate grounding and solving during the generate step for each iteration of the loop. For better efficiency, we use Clingo's multi-shot solving [59] to maintain state across iterations of the three stages. In particular, we have that

Algorithm 1 Popper

```
1  def popper(ℬ𝒦, ℰ, declaration_bias, constraints, max_literals):
2    num_literals = 1
3    while num_literals ≤ max_literals:
4      program = generate(declaration_bias, constraints, num_literals)
5      if program == 'space_exhausted':
6        num_literals += 1
7        continue
8      outcome = test(ℬ𝒦, ℰ⁺, ℰ⁻, program)
9      if outcome == ('all_positive', 'none_negative')
10       return program
11     constraints += learn_constraints(program, outcome)
12   return {}
```

the solver used by the generate stage persists its state across iterations. Like with SMT-solving, state regarding the traversal of the search space is thereby maintained in between assignments being considered and models being found.

The idea behind multi-shot solving is that the state of the solving process for an ASP program can be maintained for when we would like to solve modifications of that program. The essence of the multi-shot cycle is that a (ground) ASP program P is given to an ASP-solver S, whereupon S yields (answer set) models of P, where, through some process external to solver S, these models can lead to deriving a new (first-order) ASP program fragment P' that extends program P. Only this extension P' then needs grounding and adding to the running ASP solver, rather than needing to (re-)ground the whole of $P \cup P'$. Additionally, the running solver may, for example, maintain learned conflicts.

`Popper` uses multi-shot solving as follows. The initial ASP program is the encoding described in the next section. `Popper` asks the Clingo solver to ground this initial first-order program into a propositional program and prepare for its solving. In the generate stage, the solver is asked to return an answer set, i.e. a (propositional) model, of the current program. `Popper` converts such an answer set to a definite program and tests it against the examples. If a hypothesis fails, `Popper` generates ASP constraints using the functions in Section 4.5.5 and adds them to the running Clingo instance, which grounds the constraints and adds the new propositional rules to the running solver. We employ a hard constraint on the program size that reasons about an *external atom* [59] size(N). Initially, programs need to consist of just one literal. When there are no more answer sets with the current hard constraint, we increment the program size. Every time we increment the program size, e.g. from N to N+1, we add a new atom size($N+1$) and a new constraint enforcing this program size. Only the new constraint is ground at this point. We disable the previous size constraint by setting the external atom size(N) to false. The solver knows which parts of the search space have already been considered (by the maintained constraints), and will not revisit them. This loop repeats until either (i) `Popper` finds an optimal solution, or (ii) there are no more hypotheses to test.

4.5.3 Generate stage

The generate stage of `Popper` takes as input (i) a declaration bias, (ii) hypothesis constraints, and (iii) a size for the total number of literals in a hypothesis, and returns an

answer set which represents a definite program, if one exists. The idea is to define an ASP program, i.e. a formula, where an answer set, i.e. a (stable) model of said formula, corresponds to a program, an approach also employed by other recent ILP approaches [26, 95, 76, 139, 13]. Unlike those systems, which use a synthesis-as-rule-selection encoding, we use an encoding which selects literals rather than clauses. We develop a general encoding which we combine with specific declaration bias to yield a program that actually represents a hypothesis space. Our general encoding is a first-order ASP program which gets ground in accord with the hypothesis space settings of a declaration bias to obtain a propositional ASP program that can be handed to Clingo's Clasp solver [60]. Note that as a declaration bias always describes a finite hypotheses space, we accordingly end up with a finite grounding of the program that represents the corresponding hypothesis space.

`Popper` uses ASP constraints to ensure that a definite program that gets selected is declaration consistent and obeys hypothesis constraints, such as enforcing (simple) type restrictions or disallowing mutual recursion. By subsequently adding learned hypothesis constraints, we eliminate answer sets, and thus reduce the hypothesis space.

Synthesis-as-literal-selection

Making use of that the declaration bias specifies which literals are allowed where in clauses, we now introduce our synthesis-as-literal-selection encoding. Figure 4.5 shows the base ASP program whose models correspond to programs. The idea is to find answer sets which select suitable head and body literals, which both have the arguments `(Clause,Pred,Arity,Vars)` to denote that there is a literal in the clause `Clause`, with the predicate symbol `Pred`, arity `Arity`, and variables `Vars`. For instance, `head_literal(0,p,2,(0,1))` denotes that clause `0` has a head literal with the predicate symbol `p`, arity `2`, and variables `(0,1)`, which we interpret as `(A,B)`. Likewise, `body_literal(1,q,3,(0,0,2))` denotes that clause `1` has a body literal with the predicate symbol `q`, arity `3`, and variables `(0,0,2)`, which we interpret as `(A,A,C)`. Head and body literals are restricted by `head_pred` and `body_pred` declarations, respectively. Table 4.3 shows examples of this correspondence between an answer set and a definite program.

Answer set	**Definite program**
{head_literal(0,f,2,(0,1)),body_literal(0,empty,(1,))}	`f(A,B):-empty(B).`
{head_literal(0,f,2,(0,1)),body_literal(0,head,2,(1,0))}	`f(A,B):-head(B,A).`
{head_literal(0,f,2,(0,1)),body_literal(0,tail,2,(0,1)), body_literal(0,tail,2,(0,2))}	`f(A,B):-tail(A,B),tail(A,C).`
{head_literal(0,connected,2,(0,1)),body_literal(0,edge,2,(0,1)), head_literal(1,connected,2,(0,1)),body_literal(1,edge,2,(0,2)), body_literal(1,connected,(2,1))}	`connected(A,B):-edge(A,B).` `connected(A,B):-edge(A,C),connected(C,B).`
{head_literal(0,last,2,(0,1)),body_literal(0,tail,2,(0,2)), body_literal(0,empty,1,(2,)),body_literal(0,head,2,(0,1)), head_literal(1,last,2,(0,1)),body_literal(1,tail,2,(0,2)), body_literal(1,last,2,(2,1))}	`last(A,B):-tail(A,C),empty(C),head(A,B).` `last(A,B):-tail(A,C),last(C,B).`

Table 4.3: The correspondence between an answer set and a definite program represented as a definite program.

```
% possible clauses
allowed_clause(0..N-1) :- max_clauses(N).

% variables
var(0..N-1) :- max_vars(N).

%% head literals
0 {head_literal(Clause,P,A,Vars) : head_pred(P,A), vars(A,Vars)} 1 :-
    allowed_clause(Clause).

%% body literals
1 {body_literal(Clause,P,A,Vars) : body_pred(P,A), vars(A,Vars)} N :-
    clause(Clause), max_body(N).

% clauses with a head literal
clause(Clause) :- head_literal(Clause,_,_,_).

% variable combinations
vars(1,(Var1,)) :- var(Var1).
vars(2,(Var1,Var2)) :- var(Var1),var(Var2).
vars(3,(Var1,Var2,Var3)) :- var(Var1),var(Var2),var(Var3).
```

Figure 4.5: `Popper` base ASP program. The `head_literal` literals are bounded from 0 to 1, i.e for each possible clause there can be at most 1 head literal. The `body_literal` literals are bounded from 1 to N, where N is the maximum number of literals allowed in a clause, i.e. for each clause with a head literal, there has to be at least 1 but at most N body literals.

Static hypothesis constraints

`Popper` uses hypothesis constraints – in the form of ASP constraints – to eliminate answer sets, i.e. to prune the hypothesis space. `Popper` uses constraints to prune invalid programs. For instance, Figure 4.6 shows constraints specifically for recursive programs, such as preventing recursion without a base case. `Popper` also uses static constraints to reduce redundancy. For instance, `Popper` prunes programs which contain a subsumption-redundant clause, i.e. when one clause of the hypothesis is a subset of another[7]. For example, the following program would be pruned because the first clause subsumes the second:

$$\texttt{h} = \left\{ \begin{array}{l} \texttt{p(A) :- q(A).} \\ \texttt{p(A) :- q(A),r(A).} \end{array} \right\}$$

Finally, `Popper` uses constraints to remove some symmetries. For instance, `Popper` uses constraints that enforce that variables are used in order, which prunes programs like `p(B) :- q(B)` because we could generate `p(A) :- q(A)`.

[7]Whereas the learned constraints from Section 4.5.5 also account for the variable renaming aspect of subsumption, these static constrains only check whether one clause is *syntactically* a subset of another.

```
recursive :- recursive(Clause).

recursive(Clause) :-
    head_literal(Clause,P,A,_), body_literal(Clause,P,A,_).

has_base :- clause(Clause), not recursive(Clause).

% need multiple clauses for recursion
:- recursive(_), not clause(1).

% prevent recursion without a basecase
:- recursive, not has_base.
```

Figure 4.6: Constraints used by `Popper` to prune invalid recursive programs.

Language bias constraints

`Popper` supports optional hypothesis constraints to prune the hypothesis space. Figure 4.7 shows example language bias constraints, such as to prevent singleton variables and to enforce Datalog restrictions, i.e. that head variables must appear in the body. Declarative constraints have many benefits, notably the ease to define them. For instance, to add type checking of simple types to `Popper` requires the single constraint shown in Figure 4.7. Through constraints, `Popper` also supports the standard notions of *recall* and *input/output directions*[8] of arguments á la mode declarations [118]. `Popper` also supports *functional* and *irreflexive* constraints, and constraints on recursive programs, such as disallowing left recursion or mutual recursion.

Hypothesis constraints

Like a number of other ILP systems [118, 150, 6, 95, 144], `Popper` supports *clause* constraints, which allow a user to prune specific clauses from the hypothesis space. `Popper` additionally supports the more general concept of *hypothesis constraints* (Definition 4.2.22), which are defined over a whole program (a set of clauses) rather than a single clause (also employed in previous work [6]). For instance, hypothesis constraints allow us to prune recursive programs that do not contain a base case clause (Figure 4.6), to prune left recursive or mutually recursive programs, or to prune programs which contain subsumption redundancy between clauses.

As a toy example, suppose you want two disallow two predicate symbols $p/2$ and $q/2$ from both appearing in a hypothesis. Then this hypothesis constraint is trivial to express with `Popper`:

```
:- body_literal(_,p,2,_), body_literal(_,q,2,_).
```

[8] An input argument specifies that, at the time of calling a predicate, the corresponding argument must be instantiated, which is useful when inducing Prolog programs where literal order matters.

```
head_var(Clause,Var) :- head_literal(Clause,_,_,Vars),
   var_member(Var,Vars).

body_var(Clause,Var) :- body_literal(Clause,_,_,Vars),
   var_member(Var,Vars).

% prevent singleton variables
:- clause_var(Clause,Var),
   #count{P,Vars: var_in_literal(Clause,P,Vars,Var)} == 1.

% head vars must appear in the body
:- head_var(Clause,Var), not body_var(Clause,Var).

% type checking
var_type(Clause,Var,Type):-
   var_in_literal(Clause,P,Vars,Var),
   var_pos(Var,Vars,Pos),
   type(P,Pos,Type).

:- clause_var(Clause,Var),
   #count{Type : var_type(Clause,Var,Type)} > 1.
```

Figure 4.7: Optional language bias constraints used by `Popper`.

Metarule constraints

Finally, we demonstrate the flexibility of hypothesis constrains and the ease by which they enable fine-grained control over the hypothesis space, by imposing *metarules* through hypothesis constraints.

Metarule Metarules are templates for clauses used by many ILP systems [42], which ensure that each clause in a program is an instance of a metarule. Let M be an arbitrary metarule, i.e. a second-order Horn clause which quantifies over predicate symbols. For example, `P(A,B):-Q(A,C),R(C,B)` is known as the chain metarule. All letters are quantified variables, with `P`, `Q`, and `R` being second-order, i.e. needing to be substituted for by predicate symbols.

Let $M = \texttt{head:-body}_1, \ldots, \texttt{body}_m$ be an arbitrary metarule. We use the clause encoding function *encodeSizedClause* from Section 4.5.5 to derive an encoding of a metarule.

Example 4.5.9. Consider $M = \texttt{P(A,B):-Q(A,C),R(C,B)}$. The encoded version of this clause, $\mathit{encodeSizedClause}(\texttt{Clause}, M)$, is

```
head_literal(Clause,P,2,(V0,V1)),
body_literal(Clause,Q,2,(V0,V2)),body_literal(Clause,R,2,(V2,V1)),
V0!=V1,V0!=V2,V1!=V2,clause_size(Clause,2)
```

Asserting conformance Let Ms be a set of metarules that programs are supposed to keep to. For each clause of a program, the clause must be an *instance* of one of

the metarules in Ms. A clause C is an instance of metarule $M \in Ms$ if there exists substitution θ such that $M\theta = C$.

We introduce two rules to ensure every clause of a generated program is an instance of at least one metarule. The first rule identifies when there exists some metarule for which the clause is an instance. The second rule is a constraint expressing that every clause of a program must be identified as being an instance of at least one metarule.
For each $M \in Ms$, generate the following rule of the first kind:

$$\texttt{meta\_clause(Clause):-}\textit{encodeSizedClause}(\texttt{Clause}, M)\texttt{.}$$

The second rule is:

```
:-clause(Clause),not meta_clause(Clause).
```

We are unaware of any other ILP system that supports hypothesis constraints, at least with the same ease and flexibility as `Popper`.

4.5.4 Test stage

The test stage of `Popper` takes as input (i) a Background Knowledge program, (ii) a set of ground atoms representing the positive examples and a set of ground atoms representing the negative examples and (iii) a guessed hypothesis. If the guessed hypothesis fails, the test stage identifies and returns the type of failure (inconsistent and/or (totally) incomplete).

In the test stage, Popper converts an answer set to a definite program and tests it against the training examples. As Table 4.3 shows, this conversion is straightforward, except if input/output argument directions are given, in which case Popper orders the body literals of a clause. To evaluate a hypothesis, we use a Prolog interpreter. For each example, Popper checks whether the example is entailed by the hypothesis and Background Knowledge. As in general entailment of first-order definite programs is undecidable, in practice we often enforce a timeout in case there might be non-terminating programs among our hypotheses.

4.5.5 Constrain stage

The constrain stage takes as input (i) a guessed hypothesis and (ii) an outcome indicating in which way the hypothesis failed. Based on the program and its outcome, the constrain stage derives appropriate sound hypothesis constrains.

If a hypothesis fails, then, in the constrain stage, `Popper` derives first-order ASP constraints which, when ground and added to the encoding, prune hypotheses. Once added, these constrains appropriately restrict subsequent hypothesis generation. Specifically, we describe how we transform a failed hypothesis (a definite program) to a hypothesis constraint (a (set of) first-order ASP formula(s) written asks as constraint with respect to the encoding from Section 4.5.3). We describe the generalisation, specialisation, and redundancy constraints that `Popper` uses, based on the definitions in Section 4.3.2 and Section 4.3.3. As the experiments in Section 4.6 consider a version of `Popper` without constraint pruning, we also describe the *banish* constraint, which prunes one specific hypothesis. To distinguish between Prolog and ASP code, we represent the code of definite programs in `typewriter` font and ASP code in **`bold typewriter`** font.

We note here that our first-order constraints need to be turned into propositional formulas in order to add them to the SAT-solver which reasons about the hypothesis space. The *grounding* process that does this scales with the size of the formulas, in particular with the number of variables in each rule and with how many different instantiations there are for each variable. Hence, in principle, the cost of grounding the constraints could also become significant.

Encoding atoms

In our encoding, the atom `f(A,B)` is represented as either **`head_literal(Clause,f,2,(V0,V1))`** or **`body_literal(Clause,f,2,(V0,V1))`**, depending on where in the clause it occurs. The constant **`2`** is the predicate's arity and the variable **`Clause`** indicates that the clause index is undetermined. Two functions encode atoms into ASP literals. The function *encodeHead* encodes a head atom and *encodeBody* encodes a body atom. The first argument specifies the clause the atom belongs to. The second argument is the atom. Variables of the atom are converted to variables in our ASP encoding by the *encodeVar* function.

$$encodeHead(Clause, \texttt{Pred(Var}_0, \ldots, \texttt{Var}_k\texttt{)}) :=$$
$$\quad \textbf{\texttt{head\_literal(}}Clause\textbf{\texttt{,}}\texttt{Pred}\textbf{\texttt{,}}k+1\textbf{\texttt{,(}}encodeVar(\texttt{Var}_0)\textbf{\texttt{,}}\ldots\textbf{\texttt{,}}encodeVar(\texttt{Var}_k)\textbf{\texttt{))}}$$

$$encodeBody(Clause, \texttt{Pred(Var}_0, \ldots, \texttt{Var}_k\texttt{)}) :=$$
$$\quad \textbf{\texttt{body\_literal(}}Clause\textbf{\texttt{,}}\texttt{Pred}\textbf{\texttt{,}}k+1\textbf{\texttt{,(}}encodeVar(\texttt{Var}_0)\textbf{\texttt{,}}\ldots\textbf{\texttt{,}}encodeVar(\texttt{Var}_k)\textbf{\texttt{))}}$$

For instance, using the term **`Cl`** as a clause variable, calling $encodeHead(\textbf{\texttt{Cl}}, \texttt{f(A,B)})$ returns the ASP literal **`head_literal(Cl,f,2,(V0,V1))`**. Similarly, calling $encodeBody(\textbf{\texttt{Cl}}, \texttt{f(A,B)})$ returns **`body_literal(Cl,f,2,(V0,V1))`**.

Encoding clauses

We encode clauses by building on the encoding of atoms. Let **`Cl`** be a clause index variable. Consider the clause `last(A,B):- reverse(A,C),head(C,B)`. The following ASP literals encode where these atoms occur in a single clause:

```
head_literal(Cl,last,2,(V0,V1))
body_literal(Cl,reverse,2,(V0,V2))
body_literal(Cl,head,2,(V2,V1))
```

An ASP solver will instantiate the variables **`V0`**, **`V1`**, and **`V2`** with indices representing variables of hypotheses, e.g. **`0`** for `A`, **`1`** for `B`, etc. Note that the above encoding allows for **`V0`** = **`V1`** = **`V2`** = **`0`**, where all the variables are `A`. To ensure that variables are distinct we need to impose the inequalities **`V0!=V1`** and **`V0!=V2`** and **`V1!=V2`**. The function *assertDistinct* generates such inequalities, one between each pair of variables it is given. The function *encodeClause* implements both the straightforward translation and the variable distinctness assertion:

$$encodeClause(Clause, (\texttt{head:-body}_1, \ldots, \texttt{body}_m)) :=$$
$$\quad encodeHead(Clause, \texttt{head})\textbf{\texttt{,}}encodeBody(Clause, \texttt{body}_1)\textbf{\texttt{,}}\ldots\textbf{\texttt{,}}$$
$$\quad encodeBody(Clause, \texttt{body}_m)\textbf{\texttt{,}}$$
$$\quad assertDistinct(vars(\texttt{head}) \cup vars(\texttt{body}_1) \cup \ldots \cup vars(\texttt{body}_m))$$

As clauses can occur in multiple hypotheses, it is convenient to refer to clauses by identifiers. The function *clauseIdent* maps clauses to unique ASP constants[9]. We use the ASP literal **included_clause(***cl*,*id***)** to represent that a guessed clause with index *cl* includes all literals of a clause identified by *id*. The *inclusionRule* function generates an *inclusion rule*, an ASP rule whose head is true when the literals of the provided clause occur together in a clause:

$$\begin{aligned}&\mathit{inclusionRule}(\texttt{head:-body}_1,\ldots,\texttt{body}_m) :=\\ &\quad\texttt{included\_clause(Cl,}\mathit{clauseIdent}(\texttt{head:-body}_1,\ldots,\texttt{body}_m)\texttt{):-}\\ &\qquad\mathit{encodeClause}(\texttt{Cl},(\texttt{head:-body}_1,\ldots,\texttt{body}_m))\texttt{.}\end{aligned}$$

Suppose that $\mathit{clauseIdent}(\texttt{last(A,B):- reverse(A,C),head(C,B)}) = \texttt{id}_1$. Then the rule obtained by $\mathit{inclusionRule}(\texttt{last(A,B):- reverse(A,C),head(C,B)})$ is:

```
included_clause(Cl,id1):-
  head_literal(Cl,last,2,(V0,V1)),
  body_literal(Cl,reverse,2,(V0,V2)),
  body_literal(Cl,head,2,(V2,V1)),
  V0!=V1,V0!=V2,V1!=V2.
```

Note that **included_clause(***cl*,*id***)** being true does not mean that other literals do *not* occur in the clause. For example, if a clause with index `0` encoded the clause `last(A,B):- reverse(A,C),head(C,B),tail(C,A)`, then **included_clause(0,id$_1$)** would also hold.

In our encoding, **clause_size(***cl*,*m***)** is only true when clause *cl* has exactly *m* body literals. Hence when literals **included_clause(0,id$_1$)** and **clause_size(0,2)** are both true, the clause with index `0` exactly encodes `last(A,B):- reverse(A,C),head(C,B)` (though possibly with renamed variables). The function *exactClause* derives a pair of ASP literals checking that a clause occurs exactly:

$$\begin{aligned}&\mathit{exactClause}(\mathit{Clause},(\texttt{head:-body}_1,\ldots,\texttt{body}_m)) :=\\ &\quad\texttt{included\_clause(}\mathit{Clause}\texttt{,}\mathit{clauseIdent}(\texttt{head:-body}_1,\ldots,\texttt{body}_m)\texttt{),}\\ &\quad\texttt{clause\_size(}\mathit{Clause}\texttt{,}m\texttt{)}\end{aligned}$$

Generalisation constraints

Given a hypothesis H, by Definition 4.3.7, any hypothesis that includes all of H's clauses exactly is a generalisation of H. We use this fact to define function *generalisationConstraint*, which converts a set of clauses into ASP-encoded clause inclusion checking rules as well as a generalisation constraint (Definition 4.3.9). We use *exactClause* to impose that a clause is not specialised. Each clause is given its own ASP variable, meaning that the clauses can occur in any order.

$$\begin{aligned}&\mathit{generalisationConstraint}(\{\texttt{Clause}_0,\ldots,\texttt{Clause}_{n-1}\}) :=\\ &\quad\mathit{inclusionRule}(\texttt{Clause}_0)\\ &\quad\ldots\\ &\quad\mathit{inclusionRule}(\texttt{Clause}_{n-1})\\ &\quad\texttt{:-}\ \mathit{exactClause}(\texttt{Cl}_0,\texttt{Clause}_0),\ldots,\mathit{exactClause}(\texttt{Cl}_{n-1},\texttt{Clause}_{n-1})\texttt{.}\end{aligned}$$

Figure 4.8 illustrates *generalisationConstraint* deriving both an inclusion rule and a generalisation constraint.

[9]Even though the examples use increasing numbers in the identifiers, *clauseIdent* can be any injective function, i.e. always mapping a clause to the same unique identifier.

$$\texttt{h} = \left\{ \begin{array}{l} \texttt{last(A,B) :-} \\ \quad \texttt{reverse(A,C),head(C,B).} \end{array} \right\}$$

```
included_clause(Cl,id1) :-
  head_literal(Cl,last,2,(V0,V1)),
  body_literal(Cl,reverse,2,(V0,V2)),
  body_literal(Cl,head,2,(V2,V1)),
  V0!=V1,V0!=V2,V1!=V2.
:-
  included_clause(Cl0,id1),
  clause_size(Cl0,2).
```

Figure 4.8: ASP-encoded inclusion rule and generalisation constraint for hypothesis `h`.

Specialisation constraints – Version 1

Given a hypothesis H, by Definition 4.3.8, any hypothesis which has every clause of H occur, where each of these clauses may be specialised, and includes no other clauses, is a specialisation of H. The function *specialisationConstraintV1* uses this fact to derive an ASP-encoded specialisation constraint (Definition 4.3.12) alongside inclusion rules. When **`included_clause(`**cl**`,`**id**`)`** is true, additional atoms can occur in the clause cl. The literal **`not clause(`**n**`)`** ensures that no additional clause is added to the n distinct clauses of the provided hypothesis.

$$\begin{array}{l} \mathit{specialisationConstraintV1}(\{\texttt{Clause}_0, \ldots, \texttt{Clause}_{n-1}\}) := \\ \quad \mathit{inclusionRule}(\texttt{Clause}_0) \\ \quad \ldots \\ \quad \mathit{inclusionRule}(\texttt{Clause}_{n-1}) \\ \quad \texttt{:- included\_clause(Cl}_0\texttt{,}\mathit{clauseIdent}(\texttt{Clause}_0)\texttt{),} \ldots\texttt{,} \\ \qquad \texttt{included\_clause(Cl}_{\texttt{n-1}}\texttt{,}\mathit{clauseIdent}(\texttt{Clause}_{n-1})\texttt{),} \\ \qquad \mathit{assertDistinct}(\{\texttt{Cl}_0, \ldots, \texttt{Cl}_{\texttt{n-1}}\})\texttt{,not clause(}n\texttt{).} \end{array}$$

We illustrate why asserting that specialised clauses are distinct is necessary. Consider the hypotheses $\texttt{h}_1$ and $\texttt{h}_2$:

$$\texttt{h}_1 = \left\{ \begin{array}{l} \texttt{last(A,B):- head(A,B).} \\ \texttt{last(A,B):- sumlist(A,B).} \end{array} \right\} \texttt{h}_2 = \left\{ \begin{array}{l} \texttt{last(A,B):- head(A,B),sumlist(A,B).} \\ \texttt{last(A,B):- member(A,B).} \end{array} \right\}$$

The first clause of $\texttt{h}_2$ specialises both clauses in $\texttt{h}_1$, yet $\texttt{h}_2$ is not a specialisation of $\texttt{h}_1$. According to Definition 4.3.8, *each* clause of $\texttt{h}_2$ needs to be subsumed by a clause of $\texttt{h}_1$. Note that *specialisationConstraintV1* only considers hypotheses with at most n clauses. It is not possible for one of these clauses to be non-specialising, as each of the original n clauses is required to be specialised by a distinct clause.

Figure 4.9 illustrates a specialisation constraint derived by *specialisationConstraintV1*.

Specialisation constraints – Version 2

The just described version of the specialisation constraint only prunes programs which have the exact same number of clauses as the failed hypothesis. The definition of specialisation, Definition 4.3.8, does not require this.

A more natural encoding of what it means for a program P to be a specialisation of a hypothesis H is that every clause of P specialises at least one clause of H. This is

$$\mathtt{h} = \left\{ \begin{array}{l} \mathtt{rev(A,B):-\ head(A,B).} \\ \mathtt{rev(A,B):-} \\ \mathtt{\quad tail(A,C),head(C,B).} \end{array} \right\}$$

```
included_clause(Cl,id2):-
  head_literal(Cl,rev,2,(V0,V1)),
  body_literal(Cl,head,2,(V0,V1)),
  V0!=V1.
included_clause(Cl,id3):-
  head_literal(Cl,rev,2,(V0,V1)),
  body_literal(Cl,tail,2,(V0,V2)),
  body_literal(Cl,head,2,(V2,V1)),
  V0!=V1,V0!=V2,V1!=V2.
:-
  included_clause(Cl0,id2),
  included_clause(Cl1,id3),
  Cl0!=Cl1,not clause(2).
```

Figure 4.9: ASP-encoded inclusion rules and Version 1 of the specialisation constraint for the hypothesis h.

equivalent to determining if P has any clause that does *not* specialise a clause of H. In later versions of Popper, we introduced a new version of the specialisation constraint that captures exactly this. It makes use of a function *progIdent* that assigns a unique identifier to each program.

specialisationConstraintV2($\{\mathtt{Clause}_0, \ldots, \mathtt{Clause}_{n-1}\}$) :=
  inclusionRule($\mathtt{Clause}_0$)
  ...
  inclusionRule($\mathtt{Clause}_{n-1}$)
  has_non_specialising_clause(*progIdent*($\{\mathtt{Clause}_0, \ldots, \mathtt{Clause}_{n-1}\}$)**):-**
    clause(Cl),
    not included_clause(Cl,*clauseIdent*($\mathtt{Clause}_0$)**),**
    ...
    not included_clause(Cl,*clauseIdent*($\mathtt{Clause}_{n-1}$)**).**
  :- not has_non_specialising_clause(*progIdent*($\{\mathtt{Clause}_0, \ldots, \mathtt{Clause}_{n-1}\}$)**).**

We illustrate that this version of the specialisation constraint improves upon Version 1. Consider the hypotheses $\mathtt{h}_1$, $\mathtt{h}_3$, and $\mathtt{h}_4$:

$$\mathtt{h}_1 = \left\{ \begin{array}{l} \mathtt{last(A,B):-\ head(A,B).} \\ \mathtt{last(A,B):-\ sumlist(A,B).} \end{array} \right\} \mathtt{h}_3 = \left\{ \mathtt{last(A,B):-\ head(A,B),sumlist(A,B).} \right\}$$

$$\mathtt{h}_4 = \left\{ \begin{array}{l} \mathtt{last(A,B):-\ head(A,B),length(A,B).} \\ \mathtt{last(A,B):-\ head(A,B),empty(A).} \\ \mathtt{last(A,B):-\ sumlist(A,B).} \end{array} \right\}$$

The sole clause of $\mathtt{h}_3$ specialises both clauses of $\mathtt{h}_1$, and hence $\mathtt{h}_3$ does not have a non-specialising clause. Similarly, each clause of $\mathtt{h}_4$ is subsumed by a clause of $\mathtt{h}_1$ and hence $\mathtt{h}_4$ does *not* have a non-specialising clause. This means that Version 2 of the specialisation constraint for $\mathtt{h}_1$ does prune $\mathtt{h}_3$ and $\mathtt{h}_4$, while Version 1 does not. Version 2 of the specialisation constraint for $\mathtt{h}_1$ also does not prune $\mathtt{h}_2$. This is due to its second clause, `last(A,B):-member(A,B)`, being a non-specialising clause.

Figure 4.10 illustrates a specialisation constraint derived by *specialisationConstraintV2*.

h = { rev(A,B):- head(A,B).
rev(A,B):-
  tail(A,C),head(C,B). }

```
has_non_spec_clause(prog_id1):-
  clause(Cl),
  not included_clause(Cl,id2),
  not included_clause(Cl,id3).
:- not has_non_spec_clause(prog_id1).
```

Figure 4.10: ASP-encoded Version 2 of the specialisation constraint for the hypothesis `h`. The inclusion rules it depends on are the same as in Figure 4.9.

Redundancy constraints – Version 1: separable hypotheses

We call a program without any clause depending on another *separable*. By Proposition 4.3.23, given a totally incomplete hypothesis H, any separable hypothesis which *includes* a specialisation of H, cannot be an optimal solution. We add the following code to the `Popper` encoding to detect separable hypotheses:

```
non_separable:-
    head_literal(_,P,A,_),
    body_literal(_,P,A,_).

separable:-
    not non_separable.
```

The function *redundancyConstraintV1* uses this fact to derive a weak version of the redundancy constraint (Definition 4.3.22). Just as for the specialisation constraints, **`included_clause(`***cl*`,`***id*`)`** is used to allow additional literals in clauses, ensuring that provided clauses are specialised. However, *redundancyConstraintV1* does not require that every clause is a specialisation of a provided clause. Instead, all that is required is that each clause of the totally incomplete hypothesis is specialised and that the guessed hypothesis is separable.

$$
\begin{aligned}
&\mathit{redundancyConstraintV1}(\{\texttt{Clause}_0, \ldots, \texttt{Clause}_{n-1}\}) := \\
&\quad \mathit{inclusionRule}(\texttt{Clause}_0) \\
&\quad \ldots \\
&\quad \mathit{inclusionRule}(\texttt{Clause}_{n-1}) \\
&\quad \texttt{:- included\_clause(Cl}_0\texttt{,}\mathit{clauseIdent}(\texttt{Clause}_0)\texttt{),} \ldots\texttt{,} \\
&\qquad \texttt{included\_clause(Cl}_{\texttt{n-1}}\texttt{,}\mathit{clauseIdent}(\texttt{Clause}_{n-1})\texttt{),} \\
&\qquad \texttt{separable.}
\end{aligned}
$$

Figure 4.11 illustrates a redundancy constraint derived by *redundancyConstraintV1*.

Redundancy constraint – Version 2

The condition imposed by Version 1 of the redundancy constraint, that any pruned program is separable, severely limits its applicability. The separability condition means that Version 1 only deals with the degenerate case of there not being any dependencies at all within programs. This means that all recursive programs and all programs with invented predicates cannot be pruned by Version 1 of the redundancy constraint.

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

```
included_clause(Cl,id4):-
  head_literal(Cl,last,2,(V0,V1)),
  body_literal(Cl,tail,2,(V0,V2)),
  body_literal(Cl,head,2,(V2,V1)),
  V0!=V1,V0!=V2,V1!=V2.
:-
  included_clause(Cl0,id4),
  separable.
```

Figure 4.11: ASP-encoded inclusion rule and Version 1 of the redundancy constraint for the hypothesis `h`.

We now introduce a more general redundancy constraint, Version 2, that does take the dependency of clauses on another into account.

First we encode the dependency relation on clauses, as defined by Definition 4.3.15, into ASP and add it to the `Popper` encoding:

```
depends_on(C1,C2):-
  head_literal(C2,Pred,Arity,_),
  body_literal(C1,Pred,Arity,_).
depends_on(C1,C3) :-
  depends_on(C1,C2),
  depends_on(C2,C3).
```

The following constraint works by trying to find a clause of a guessed program P that is redundant with respect to a (totally incomplete) hypothesis H. We do so by exhaustively considering all cases whereby a clause is shown to be *not* redundant. First, a clause C of P is not redundant with respect to H if C does not specialise (a clause of) H. Next, what if $C \in P$ does specialise (part of) H? Well, by Proposition 4.3.23, C would not be redundant if it depended on a clause that did not specialise H. The last case whereby C is not redundant is if a clause that did not specialise H depended on C. If none of these cases apply, then C does specialise H and only depends on or is depended upon by clauses specialising H. Hence C is actually redundant with respect to H. If there is a clause C in P for which we fail to show it is not redundant, then we know P has a redundant clause and P can be pruned.

The function *redundancyConstraintV2* encodes this principle into a strong variant of the redundancy constraint (Definition 4.3.22).

redundancyConstraintV2($P = \{\texttt{Clause}_0, \ldots, \texttt{Clause}_{n-1}\}$) :=
  inclusionRule($\texttt{Clause}_0$)
  ...
  inclusionRule($\texttt{Clause}_{n-1}$)
  specialises(Cl,*progIdent*(P)**):-**
    included_clause(Cl,*clauseIdent*($\texttt{Clause}_0$)**).**
  ...
  specialises(Cl,*progIdent*(P)**):-**
    included_clause(Cl,*clauseIdent*($\texttt{Clause}_{n-1}$)**).**
  not_redundant(Cl,*progIdent*(P)**):-**
    not specialises(Cl,*progIdent*(P)**).**
  not_redundant($\texttt{Cl}_1$,*progIdent*(P)**):-**
    specialises($\texttt{Cl}_1$,*progIdent*(P)**),**
    depends_on($\texttt{Cl}_1$,$\texttt{Cl}_2$),
    not specialises($\texttt{Cl}_2$,*progIdent*(P)**).**
  not_redundant($\texttt{Cl}_1$,*progIdent*(P)**):-**
    specialises($\texttt{Cl}_1$,*progIdent*(P)**),**
    depends_on($\texttt{Cl}_2$,$\texttt{Cl}_1$),
    not specialises($\texttt{Cl}_2$,*progIdent*(P)**).**
  :-clause(Cl),
    not not_redundant(Cl,*progIdent*(P)**).**

Figure 4.12 illustrates a redundancy constraint derived by *redundancyConstraintV2*.

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

```
specialises(Cl,prog_id2):-
  included_clause(Cl0,id4).
not_redundant(Cl,prog_id2):-
  not specialises(Cl,prog_id2).
not_redundant(Cl1,prog_id2):-
  specialises(Cl1,prog_id2),
  depends_on(Cl1,Cl2),
  not specialises(Cl2,prog_id2).
not_redundant(Cl1,prog_id2):-
  specialises(Cl1,prog_id2),
  depends_on(Cl2,Cl1),
  not specialises(Cl2,prog_id2).
:- clause(Cl),
   not not_redundant(Cl,prog_id2).
```

Figure 4.12: ASP-encoded Version 2 of the redundancy constraint for the hypothesis `h`. The inclusion rule that this constraint depends on is the same as in Figure 4.11.

Banish constraints

In the experiments in Section 4.6 we compare `Popper` against a version of itself without constraint pruning. To do so we need to remove single hypotheses from the hypothesis

space. We introduce the *banish constraint* for this purpose. To prune a specific hypothesis, hypotheses with different variables should not be pruned. We accomplish this condition by changing the behaviour of the *encodeVar* function. Normally *encodeVar* returns ASP variables which are then grounded to indices that correspond to the variables of hypotheses. Instead, by the following definition, *encodeVar* directly assigns the corresponding index for a hypothesis variable:

$$encodeVar = \{\, \texttt{A} \mapsto \texttt{0};\ \texttt{B} \mapsto \texttt{1};\ \texttt{C} \mapsto \texttt{2};\ \ldots \}$$

For a banish constraint no additional literals in clauses are allowed, nor are additional clauses. The below function *banishConstraint* ensures both conditions when converting a hypothesis to an ASP-encoded banish constraint. That provided clauses occur non-specialised is ensured by *exactClause*. The literal **not clause(*n*)** asserts that there are no more clauses than the original number.

$$\begin{aligned}
&banishConstraint(\{\texttt{Clause}_0, \ldots, \texttt{Clause}_{n-1}\}) := \\
&\quad inclusionRule(\texttt{Clause}_0) \\
&\quad \ldots \\
&\quad inclusionRule(\texttt{Clause}_{n-1}) \\
&\quad \texttt{:-}\ exactClause(\texttt{Cl}_0, \texttt{Clause}_0), \ldots, exactClause(\texttt{Cl}_{\texttt{n-1}}, \texttt{Clause}_{n-1}), \\
&\qquad \texttt{not clause(}n\texttt{).}
\end{aligned}$$

Figure 4.13 illustrates a banish constraint derived by *banishConstraint*.

$$\texttt{h} = \left\{ \begin{array}{l} \texttt{f(A):- head(A,B),one(B).} \\ \texttt{f(A):- tail(A,B),empty(B).} \end{array} \right\}$$

```
included_clause(Cl,id5):-
  head_literal(Cl,f,1,(0,)),
  body_literal(Cl,head,2,(0,1)),
  body_literal(Cl,one,1,(1,)).
included_clause(Cl,id6):-
  head_literal(Cl,f,1,(0,)),
  body_literal(Cl,tail,2,(0,1)),
  body_literal(Cl,empty,1,(1,)).
:-
  included_clause(Cl0,id5),
  clause_size(Cl0,2),
  included_clause(Cl1,id6),
  clause_size(Cl1,2),
  not clause(2).
```

Figure 4.13: ASP-encoded inclusion rules and banish constraint for the hypothesis `h`.

4.5.6 Correctness

We now show the correctness of `Popper`. However, we only show this result in case the hypothesis space[10] that is required is that hypothesis space-encoding can gradually additionally only contains decidable programs, e.g. Datalog programs. When the hypothesis

[10]By relying on declaration biases to specify the hypothesis space, we also (implicitly) make use of that the hypothesis space is finite. To support infinite hypothesis spaces, all we would require is a way to

space contains arbitrary definite programs, these results do not hold because checking for entailment of an arbitrary definite program is only semi-decidable [153]. In other words, the results in this section only hold when every hypothesis in the hypothesis space is guaranteed to terminate[11].

We first show that `Popper`'s base encoding (Figure 4.5) can generate every declaration consistent hypothesis (Definition 4.5.6):

Proposition 4.5.10. The base encoding of `Popper` has a model for every declaration consistent hypothesis.

Proof. Let $D = (D_h, D_b, \texttt{max\_var}(N_{var}), \texttt{max\_body}(N_{body}), \texttt{max\_clauses}(N_{clause}))$ be a declaration bias and H be any hypothesis that is declaration consistent with D. Let C be any clause in H. Our encoding represents the head literal $p_h(H_1, \ldots, H_n)$ of C as a choice literal $\texttt{head\_literal(i,}p_h\texttt{,}n\texttt{,(}H_1\texttt{,}\ldots\texttt{,}H_n\texttt{))}$ guarded by the condition $\texttt{head\_pred(}p_h\texttt{,}n\texttt{)} \in D_h$, which clearly holds. Our encoding represents a body literal $p_b(B_1, \ldots, B_m)$ of C as a choice literal $\texttt{body\_literal(}i\texttt{,}p_b\texttt{,}m\texttt{,(}B_1\texttt{,}\ldots\texttt{,}B_m\texttt{))}$ guarded by the condition $\texttt{body\_pred(}p_b\texttt{,}m\texttt{)} \in D_b$, which clearly holds. The base encoding only constrains the above guesses by three conditions: (i) at most N_{var} unique variables per clause, (ii) at least 1 and at most N_{body} body literals per clause, and (iii) at most N_{clause} clauses. As both the hypothesis and the guessed literals satisfy the same conditions, we conclude there exists a model representing H. □

We show the *soundness* of `Popper`, i.e. that any hypothesis returned is a solution (Definition 4.2.26):

Proposition 4.5.11. Any hypothesis returned by `Popper` is a solution.

Proof. As per Algorithm 1, any returned hypothesis has been tested against the training examples and confirmed to entail all positive examples and none of the negative examples. □

To make the next two results shorter, we introduce a lemma to show that `Popper` never prunes optimal solutions (Definition 4.2.10):

Lemma 4.5.12. `Popper` never prunes optimal solutions.

Proof. `Popper` only learns constraints from a failed hypothesis, i.e. a hypothesis that is incomplete or inconsistent. Let H be a failed hypothesis. If H is incomplete, then, as described in Section 4.3, `Popper` prunes specialisations of H. Proposition 4.3.14 shows that a specialisation constraint never prunes complete hypotheses, and thus never prunes optimal solutions. If H is inconsistent, then, as described in Section 4.3, `Popper` prunes generalisations of H. Proposition 4.3.11 shows that a generalisation constraint never prunes consistent hypotheses, and thus never prunes optimal solutions. Finally, if H is totally incomplete, then, as described in Section 4.3.3, `Popper` uses an redundancy constraint to prune hypotheses that contain (a specialisation of) a clause of H that is redundant. Proposition 4.3.23 shows that a redundancy constraint never prunes optimal solutions. Since `Popper` only uses these three constraints, it never prunes optimal solutions. □

gradually expand the encoding (with multi-shot solving being exactly such a mechanism) and a search heuristic that systematically considers all programs in the limit, e.g. by considering them by increasing size as we already do.

[11] In practice, such as in our experiments on learning list transformation programs, we enforce a timeout when testing programs, i.e. we assume that every solution terminates before the timeout.

We now show that `Popper` is *complete*, i.e. a solution will be returned if one exists:

Proposition 4.5.13. `Popper` returns a solution if one exists.

Proof. Assume, for contradiction, that `Popper` does not return a solution, which implies that (1) `Popper` returned a hypothesis that is not a solution, or (2) `Popper` did not return a hypothesis at all. Case (1) cannot hold because Proposition 4.5.11 shows that every hypothesis returned by `Popper` is a solution. For case (2), by Proposition 4.5.10, `Popper` can generate every hypothesis so it must be the case that (i) `Popper` did not terminate, (ii) a solution did not pass the test stage, or (iii) that every solution was incorrectly pruned. Case (i) cannot hold because Proposition 4.2.17 shows that the hypothesis space is finite so there are finitely many hypotheses to generate and test, and both searching for a model and testing a hypothesis is guaranteed to terminate. Case (ii) cannot hold because a solution is by definition a hypothesis that passes the test stage. Case (iii) cannot hold because Lemma 4.5.12 shows that `Popper` never prunes optimal solutions. These cases are exhaustive, so the assumption cannot hold, and thus `Popper` returns a solution if one exists. □

We show that `Popper` returns an optimal solution if one exists:

Theorem 4.5.14. `Popper` returns an optimal solution if one exists.

Proof. By Proposition 4.5.13, `Popper` returns a solution if one exists. Let H be the solution returned by `Popper`. Assume, for contradiction, that H is not an optimal solution. By Definition 4.2.10, this assumption implies that either (1) H is not a solution, or (2) H is a non-optimal solution. Case (1) cannot hold because H is a solution. Therefore, case (2) must hold, i.e. there must be at least one smaller solution than H. Let H' be such an optimal solution, meaning we know $size(H') < size(H)$. By Proposition 4.5.10, `Popper` generates every hypothesis, and `Popper` generates hypotheses of increasing size (Algorithm 1), therefore the smaller solution H' must have been considered before H, which implies that H' must have been pruned by a constraint. However, Lemma 4.5.12 shows that H' could not have been pruned and so cannot exist, which contradicts the assumption and completes the proof. □

4.5.7 Worked example

To illustrate `Popper`, reconsider the example from the introduction of learning a *last/2* hypothesis to find the last element of a list. For simplicity, assume an initial hypothesis space $\mathcal{H}_1$:

$$\mathcal{H}_1 = \left\{ \begin{array}{l} \mathtt{h}_1 = \{ \texttt{last(A,B):- head(A,B).} \} \\ \mathtt{h}_2 = \{ \texttt{last(A,B):- head(A,B),empty(A).} \} \\ \mathtt{h}_3 = \{ \texttt{last(A,B):- tail(A,C),head(C,B).} \} \\ \mathtt{h}_4 = \{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \} \\ \mathtt{h}_5 = \{ \texttt{last(A,B):- head(A,B),reverse(A,C),head(C,B).} \} \\ \mathtt{h}_6 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\} \\ \mathtt{h}_7 = \left\{ \begin{array}{l} \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \end{array} \right\} \\ \mathtt{h}_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\} \\ \mathtt{h}_9 = \left\{ \begin{array}{l} \texttt{last(A,B):- head(A,B).} \\ \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\} \end{array} \right\}$$

Also assume we have the positive ($\mathcal{E}^+$) and negative ($\mathcal{E}^-$) examples:

$$\mathcal{E}^+ = \left\{ \begin{array}{l} \texttt{last([l,a,u,r,a],a).} \\ \texttt{last([p,e,n,e,l,o,p,e],e).} \end{array} \right\} \quad \mathcal{E}^- = \left\{ \begin{array}{l} \texttt{last([e,m,m,a],m).} \\ \texttt{last([j,a,m,e,s],e).} \end{array} \right\}$$

To start, `Popper` generates the smallest hypothesis:

$$\mathtt{h}_1 = \{ \texttt{last(A,B):- head(A,B).} \}$$

`Popper` then tests $\mathtt{h}_1$ against the examples and finds that it *fails* because it does not entail any positive example and is therefore too *specific*. `Popper` then generates a specialisation constraint (Version 1) to prune specialisations of $\mathtt{h}_1$:

```
included_clause(Cl,id1):-
    head_literal(Cl,last,2,(V0,V1)),
    body_literal(Cl,head,2,(V0,V1)),
    V0!=V1.
:-
    included_clause(Cl0,id1),
    not clause(1).
```

`Popper` adds this constraint to the meta-level ASP program which prunes $\mathtt{h}_2$ and $\mathtt{h}_5$ from the hypothesis space. In addition, because $\mathtt{h}_1$ does not entail any positive example (is *totally* incomplete), `Popper` also generates a redundancy constraint (Version 1):

```
:-
    included_clause(Cl0,id1),
    separable.
```

Popper adds this constraint to the meta-level ASP program which prunes h_9 from the hypothesis space. The thus updated ASP program now represent the hypothesis space:

$$\mathcal{H}_2 = \left\{ \begin{array}{l} h_3 = \{ \texttt{last(A,B):- tail(A,C),head(C,B).} \} \\ h_4 = \{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \} \\ h_6 = \{ \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \qquad \texttt{last(A,B):- reverse(A,C),head(C,B).} \} \\ h_7 = \{ \texttt{last(A,B):- tail(A,C),head(C,B).} \\ \qquad \texttt{last(A,B):- tail(A,C),tail(C,D),head(D,B).} \} \\ h_8 = \{ \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \qquad \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \} \end{array} \right\}$$

Popper generates another hypothesis (h_3) and tests against the examples and finds that it fails because it entails the negative example `last([e,m,m,a],m)` and is therefore too *general*. Popper then generates a generalisation constraint to prune generalisations of h_3:

```
included_clause(Cl,id2):-
    head_literal(Cl,last,2,(V0,V1)),
    body_literal(Cl,tail,2,(V0,V2)),
    body_literal(Cl,head,2,(V2,V1)),
    V0!=V1,V0!=V2,V1!=V2.
:-
  included_clause(Cl0,id2),
  clause_size(Cl0,2).
```

Popper adds this constraint to the meta-level ASP program which prunes h_6 and h_7 from the hypothesis space. The updated ASP program now represents the hypothesis space:

$$\mathcal{H}_3 = \left\{ \begin{array}{l} h_4 = \{ \texttt{last(A,B):- reverse(A,C),head(C,B).} \} \\ h_8 = \{ \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \qquad \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \} \end{array} \right\}$$

Finally, Popper generates another hypothesis (h_4), tests it against the examples, finds that it does not fail, and returns it.

4.6 Experimental evaluation

We now evaluate Popper. Popper learns constraints from failed hypotheses to prune the hypothesis space to improve learning performance. We therefore claim that, compared to unconstrained learning, constraints can improve learning performance. One may think that this improvement is obvious, i.e. constraints will definitely improve performance. However, it is unclear whether in practice, and if so by how much, constraints will improve learning performance because Popper needs to (i) analyse failed hypotheses, (ii) generate constraints from them, and (iii) pass the constraints to the ASP system, which then needs to ground and solve them, which may all have non-trivial computational overheads. Our experiments therefore aim to answer the question:

Q1 Can constraints improve learning performance compared to unconstrained learning?

To answer this question, we compare `Popper` against a brute-force generate and test approach. To do so, we use a version of `Popper` with only banish constraints enabled to prevent repeated generation of a failed hypothesis. We call this system `Enumerate`.

Proposition 4.2.17 shows that the size of the Learning From Failures hypothesis space is a function of many parameters, including the number of predicate declarations, the number of unique variables in a clause, and the number of clauses in a hypothesis. To explore this result, our experiments aim to answer the question:

Q2 How well does `Popper` scale?

To answer this question, we evaluate `Popper` when varying (i) the size of the optimal solution, (ii) the number of predicate declarations, (iii) the number of constants in the problem, (iv) the number of unique variables in a clause, (v) the maximum number of literals in a clause, and (vi) the maximum number of clauses allowed in a hypothesis.

We also compare `Popper` against existing ILP systems. Our experiments therefore aim to answer the question:

Q3 How well does `Popper` perform compared to other ILP systems?

To answer this question, we compare `Popper` against Aleph [150], Metagol [39], ILASP2i [96], and ILASP3 [93]. It is important to note that a direct comparison of ILP systems is difficult because different systems excel at different problems and often employ different biases. For instance, directly comparing the Prolog-based Metagol against the ASP-based ILASP is difficult because Metagol is often used to learn recursive list manipulation programs, such as string transformations and sorting algorithms, whereas ILASP does not support explicit lists because the ASP system Clingo [58], on which ILASP is built, does not support constructing arbitrary terms from function symbols such as lists (without running into grounding issues). Likewise, Aleph and ILASP3 support noise, whereas Metagol and `Popper` do not. Moreover, because ILP systems have many learning parameters, it is often possible to show that there exist some parameter settings for which system X can perform better than system Y on a particular problem. Overall, a direct comparison between ILP systems is difficult, so a reader should not interpret the results as system X is better than system Y.

4.6.1 Buttons

The purpose of this first experiment is to evaluate how well `Popper` scales when varying the optimal solution size[12]. We therefore need a problem where we can control the optimal solution size. We consider a problem loosely based on the IGGP game *buttons and lights* [33]. The problem is purposely simple: given p buttons, learn which n buttons need to be pressed to win. For instance, for $n = 3$, a solution could be:

```
win(A):- button6(A),button4(A),button7(A)
```

The variable A denotes the player and $button_p$ denotes that player A pressed $button_p$.

In this experiment, we fix p, the number of buttons, and vary n, the number of buttons that need to be pressed, which directly corresponds to the optimal solution size.

[12]Note that, in this experiment, increasing the optimal solution size almost always also increases the size of the hypothesis space for the considered ILP systems.

Materials

We consider two variations of the problem where $p = 20$ and $p = 200$, which we name *small* and *big* respectively. We compare `Popper`, `Enumerate`, Metagol, ILASP2i, ILASP3, and Aleph. To compare the systems, we try to use settings so that each system searches approximately the same hypothesis space. However, ensuring that the systems search identical hypothesis spaces is near impossible. For instance, Metagol performs automatic predicate invention and so considers a different hypothesis space to the other systems. The exact language biases used are in Appendix A.1.

ILASP settings. We asked Mark Law, the ILASP author, for advice on how best to solve this problem with ILASP2i and ILASP3[13]. We run both ILASP2i and ILASP3 with the same settings so we simply refer to both as ILASP. We run ILASP with the 'no constraints', 'no aggregates', 'disable implication', 'disable propagation', and 'simple contexts' flags. We tell ILASP that each BK relation is positive, which prevents it from generating body literals using negation. We also make the problem propositional and use context-dependent examples [96] where the context-dependent BK for each example contains the buttons pressed in that example. We initially tried to run ILASP with at most ten body literals ('-ml=10' and '–max-rule-length=11') but when given this parameter ILASP would not terminate in the time limit because it pre-computes every rule in the hypothesis space. Therefore, for each number of buttons n, we set the maximum number of body literals to n ('-ml=n' and '–max-rule-length=n+1'), to ensure that ILASP terminates on some of the problems.

Metagol settings. Metagol needs metarules to guide the proof search. We provide Metagol with the following two metarules:

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

`Popper` and Enumerate settings. We set `Popper` and Enumerate to use at most 1 unique variable, at most 1 clause, and at most n body literals. These settings match those imposed by Metagol's metarules and somewhat ILASP's propositional representation. We restrict the clause to have at most n body literals to match ILASP's settings. When allowed up to ten body literals, Popper performs almost identically.

Aleph settings. We also set the maximum number of nodes to search to 5000. As with `Popper`, Enumerate, and ILASP, we increase the maximum clause length for Aleph for each value n.

Methods

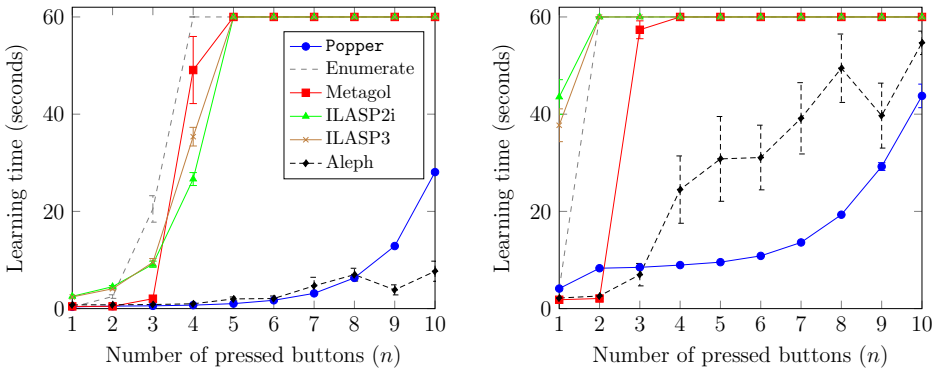
For each n in $\{1, 2, \ldots, 10\}$, we generate 200 positive and 200 negative examples. A positive example is a player that has pressed the correct n buttons. To generate a positive

[13]Law suggested an alternative representation that corresponds to learning the negation of the concept, which would have been much more suitable for ILASP. However, this alternative different representation requires NAF which not all of the other systems support.

example we sample without replacement n integers from the set $\{1, \ldots, p\}$ which correspond to the n buttons that *must* be pressed. We additionally sample extra buttons that are also pressed, but which are not necessarily pressed in all the positive examples. A negative example is a player that has not pressed the correct n buttons. To generate a negative example we sample without replacement at most $n-1$ buttons from the set that must be pressed. We then sample other buttons that should not be pressed. By including all n negative examples with $n-1$ correct buttons we guarantee that there is only one correct solution. We measure learning time as the time to learn a solution. We enforce a timeout of one minute per task. We repeat each experiment ten times and plot the standard error.

Results

Figure 4.14 shows that `Popper` clearly outperforms `Enumerate` on both datasets. On the small dataset ($p = 20$), `Enumerate` only learns a program for when three buttons must be pressed ($n = 3$). On the large dataset ($p = 200$), `Enumerate` only learns a program for when one button must be pressed ($n = 1$). By contrast, on both datasets, `Popper` learns a program for when ten buttons must be pressed ($n = 10$), i.e. a program with ten body literals. Moreover, `Popper` always learns a solution comfortably within the time limit. This result strongly suggests that the answer to **Q1** is yes, constraints can drastically improve learning performance.



Small BK (p=20) Big BK (p=200)

Figure 4.14: Buttons experiment.

`Popper` outperforms Metagol on both datasets. For the small dataset, the largest program that Metagol learns is for when $n = 4$, which takes 50 seconds to learn, compared to one second for `Popper`. For the big dataset, the largest program that Metagol learns is for when $n = 3$, which takes 57 seconds to learn, compared to eight seconds for `Popper`. Metagol struggles because of its inefficient search. Metagol performs iterative deepening over the number of clauses allowed in a solution [121]. However, if a clause or literal fails during the search, Metagol does not remember this failure, and will retry already failed clauses and literals at each depth (and even multiple times as the same depth). By contrast, if a clause fails, `Popper` learns constraints from the failure so it never tries that clause (or its specialisations) again.

`Popper` outperforms ILASP2i and ILASP3 on both datasets. ILASP2i only learns programs with four (small dataset) and one (big dataset) body literals. ILASP3 only

learns programs with four (small dataset) and one (big dataset) body literals. ILASP2i and ILASP3 both struggle on this problem because they pre-compute every clause in the hypothesis space, which means that they struggle to learn clauses with many body literals. By contrast, `Popper` can learn programs with ten body literals on both datasets.

Aleph outperforms `Popper` on the small dataset when $n > 8$. However, on the big dataset, `Popper` outperforms Aleph when $n > 3$.

Overall, the results from this experiment suggest that (i) the answer to question **Q1** is certainly yes, constraints improve learning performance, (ii) the answer to **Q2** is that `Popper` scales well in terms of the number of body literals in a solution and the number of background relations, and (iii) the answer to **Q3** is that `Popper` can outperform other ILP systems when varying the optimal solution size and the number of background relations.

4.6.2 Robots

The purpose of this second experiment is to evaluate how well `Popper` scales with respect to the domain size (i.e. the constant signature). We therefore need a problem where we can control the domain size. We consider a robot strategy learning problem [38]. There is a robot in a $n \times n$ grid world. Given an arbitrary start position, the goal is to learn a general strategy to move the robot to the topmost row in the grid. For instance, for a 10×10 world and the start position $(2, 2)$, the goal is to move to position $(2, 10)$. The domain contains all possible robot positions. We therefore vary the domain size by varying n, the size of the world. The optimal solution is a recursive strategy for *keep moving upwards until you are at the top row*. To reiterate, we purposely fix the optimal solution so that the only variable in the experiment is the domain size (i.e. the grid world size), which we progressively increase to evaluate how well the systems scale.

Materials

We consider two representations: a representation for `Popper`, `Enumerate`, Metagol, and Aleph, and then a representation designed to help ILASP solve the problem. When given the Prolog representation, neither ILASP2i nor ILASP3 could solve any of the problems because of the grounding problem. In both representations, we provide as BK four dyadic relations, *move_right*, *move_left*, *move_up*, and *move_down*, that change the state, e.g. *move_right((2,2),(3,2))*, and four monadic relations, *at_top*, *at_bottom*, *at_left*, and *at_right*, that check the state. The exact language biases used can be found in Appendix A.2.

Prolog representation. In the Prolog representation, an example is an atom of the form $f(s_1, s_2)$, where s_1 and s_2 represent start and end states. A state is a pair of discrete coordinates (x, y) denoting the column (x) and row (y) position of the robot.

ILASP representation. When given the Prolog representation, neither ILASP2i nor ILASP3 could solve any of the problems in this experiment because of the grounding problem. We therefore asked Mark Law to help us design a more suitable representation. In this representation, an example is an atom of the form $f(s_2)$ where s_2 represents the end state. Each example is a distinct ILASP example (a partial interpretation) with its own *context*, where the start state is given in the context as *start_state(s_1)*. This representation alleviates the grounding problem of the Prolog representation.

ILASP2i and ILASP3 settings. We run both ILASP2i and ILASP3 with the same settings, so we again refer to both as ILASP. We run ILASP with the 'no constraints', 'no aggregates', 'disable implication', 'disable propagation', and 'simple contexts' flags. We tell ILASP that each BK relation is *positive*, *anti_reflexive*, and *symmetric*. We also employ a set of 'bias constraints' to reduce the hypothesis space. We also restrict some of the recall values for the BK relations. We set ILASP to use at most four unique variables and at most three body literals ('-ml=3' and '–max-rule-length=4'). The full language bias restrictions can be found in Appendix A.2.

Metagol settings. We provide Metagol with the metarules in Figure 4.15. These metarules constitute an almost[14] complete set of metarules for a singleton-free fragment of monadic and dyadic Datalog [42].

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

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

Figure 4.15: The metarules used by Metagol in the robot and list transformation experiments.

Popper settings. We allow `Popper` and `Enumerate` to use at most four unique variables per clause and at most three body literals (which match the ILASP settings), and at most three clauses.

Aleph settings. We set the maximum variable depth and clause length to six and set the maximum number of search nodes to 30000.

Methods

We run the experiment with an $n \times n$ grid world for each n in $\{10, 20, \ldots, 100\}$. To generate examples, for start states, we uniformly sample positions that are not at the top of the world. For the positive examples, the end state is the topmost position, e.g. (x, n) where n is the grid size. For negative examples, we randomly sample start and end states and reject the example if it is a positive example. To ensure that there are some negative examples with the topmost position, in 25% of the examples we set the end position to be the topmost row of column y, but ensure that the start position is not y. We sample with replacement 20 positive and 20 negative training examples, and 1000 positive and 1000 negative testing examples. The default predictive accuracy is therefore 50%. We measure predictive accuracies and learning times. We enforce a timeout of one minute per task. If a system fails to learn a solution in the given time then it only achieves default predictive accuracy (50%). We repeat each experiment ten times and plot the standard error.

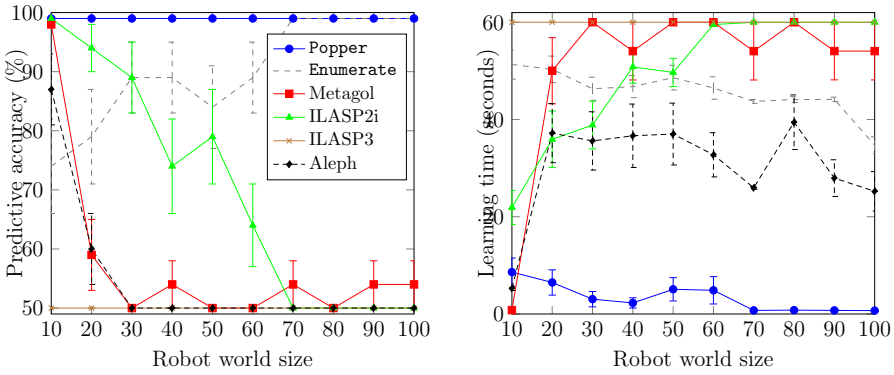
[14]It is impossible to generate a finite and complete set of metarules for a singleton-free fragment of monadic and dyadic Datalog [42].

Results

Figure 4.16 shows the results. `Popper` achieves the best predictive accuracy out of all the systems. `Enumerate` is the second best performing system, although it is does not always learn the optimal solution. `Popper` is substantially quicker than `Enumerate` (on average about 40 times quicker) and is the fastest of all the systems. The learning time of `Popper` slightly decreases as the grid size grows. The reason for this is twofold. First, when the grid world is small, there are often many small programs that cover some of the positive examples but none of the negative examples, such as:

```
f(S1,S2):- move_up(S1,S3),move_up(S3,S2).
```

Because they cover some of the examples, Popper cannot completely rule them out. However, as the grid size grows, these smaller programs are less likely to cover the examples because the examples are more spread out over the grid. Second, solutions have either five or six literals, with smaller solutions becoming more likely with increasing world size. These reasons explain why the predictive accuracy of `Enumerate` improves as the grid size grows. The reason that the learning time of `Popper` does not increase is that the domain size has no influence on the size of the Learning From Failures hypothesis space (Proposition 4.2.17). The only influence the grid size has is any overhead in executing the induced Prolog program on larger grids. This result suggests that `Popper` can scale well with respect to the domain size.



(a) Predictive accuracies (b) Learning times

Figure 4.16: Robots experimental results when varying the world size, which corresponds to the domain size.

`Popper` outperforms Metagol in all cases. For a small 10x10 grid world, Metagol learns the optimal solution and does so quicker than `Popper` (Metagol takes 1 second compared to `Popper` which takes 9 seconds). However, as the grid size grows, Metagol's performance quickly degrades. For a grid size greater than 20, Metagol almost always times out before finding a solution. The reason is that Metagol searches for a hypothesis by inducing and executing partial programs over the examples. In other words, Metagol uses the examples to guide the hypothesis search. As the grid size grows, there are more partial programs to construct, so its performance suffers. Note that when Metagol learns a solution, it is always an accurate solution.

`Popper` outperforms ILASP2i and ILASP3 both in terms of predictive accuracies and learning times. ILASP3 cannot learn any solutions in the given time, even for the 10x10

world. ILASP2i initially learns solutions in the given time limit, but struggles as the grid size grows. Note that when ILASP2i learns a solution, it is always an accurate solution.

ILASP2i outperforms ILASP3 because once ILASP2i finds a solution it terminates. By contrast, ILASP3 finds one hypothesis schema that guarantees coverage of the example (which, in this special case, also implies finding a solution), then carries on to find alternative hypothesis schemas. The extra work done by ILASP3 is needed when learning general ASP programs, but in this special case (where there no ILASP negative examples) it is unnecessary and computationally expensive. We refer the reader to Law's thesis [93] for a detailed comparison of ILASP2i and ILASP3[15].

`Popper` outperforms Aleph. For small grid worlds, Aleph sometimes learns programs that generalise to the training set (such as move up three times). But as the grid size grows, Aleph struggles because it struggles to learn recursive programs.

Overall, the results from this experiment suggest that (i) the answer to question **Q1** is certainly yes, constraints improve learning performance, (ii) the answer to **Q2** is that `Popper` scales well in terms of the domain size, and (iii) the answer to **Q3** is that `Popper` can outperform other ILP systems when varying the domain size.

4.6.3 List transformation problem

The purpose of this third experiment is to evaluate how well `Popper` performs on difficult (mostly recursive) list transformation problems. Learning recursive programs has long been considered a difficult problem in ILP [119] and most ILP and program synthesis systems cannot learn recursive programs. Because ILASP2i and ILASP3 do not support lists, we only compare `Popper`, `Enumerate`, Metagol, and Aleph.

Materials

We evaluate the systems on the ten list transformation tasks shown in Table 4.4. These tasks include a mix of monadic (e.g. `evens` and `sorted`), dyadic (e.g. `droplast` and `finddup`), and triadic (`dropk`) target predicates. The tasks also contain a mix of functional (e.g. `last` and `len`) and relational problems (e.g. `finddup` and `member`). These tasks are extremely difficult for ILP systems. To learn solutions that generalise, an ILP system needs to support recursion and large domains. As far as we are aware, no existing ILP system can learn optimal solutions for all of these tasks without being provided with a strong inductive bias[16].

We give each system the following dyadic relations *head*, *tail*, *decrement*, *geq* and the monadic relations *empty*, *zero*, *one*, *even*, and *odd*. We also include the dyadic relation *increment* in the `len` experiment. We had to remove this relation from the BK for the other experiments because when given this relation Metagol runs into infinite recursion[17] on almost every problem and could not find any solutions. We also include *member/2* in

[15] We thank Mark Law for this explanation.

[16] As discussed in Section 2.1.2, some inverse entailment methods [118] might sometimes learn solutions for them. However, these approaches need an example to learn the base case of a recursive program and then an example to learn the inductive case. Moreover, these approaches would not be guaranteed to learn the optimal solution. Metagol could possibly learn solutions for them if given the exact metarules needed, but that requires that you know the solution before you try to learn it.

[17] Because Metagol induces hypotheses by partially constructing and evaluating hypotheses, it is very difficult to impose a timeout on a particular hypothesis, which we can easily do with `Popper`.

the find duplicate problem. We also include *cons/3* in the `addhead`, `dropk`, and `droplast` experiments. We exclude this relation from the other experiments because Metagol does not easily support triadic relations. The exact language biases used can be found in Appendix A.3.

Metagol settings. For Metagol, we use almost the same metarules as in the previous robot experiment (Figure 4.15). However, when given the *inverse* metarule $P(A, B) \leftarrow Q(B, A)$, Metagol could not learn any solution, again because of infinite recursion. To aid Metagol, we therefore replace the *inverse* metarule with the *identity* metarule, i.e. $P(A, B) \leftarrow Q(A, B)$. In addition, when we first ran the experiment with randomly ordered examples, we found that Metagol struggled to find solutions for all the problems (except `member`). The reason is that Metagol is sensitive to the order of examples because it uses the examples in the order they are given to induce a hypothesis. Therefore, to aid Metagol, we provide the examples in increasing size (i.e. the length of the input lists).

`Popper` and `Enumerate` settings. We set `Popper` and `Enumerate` to use at most five unique variables, at most five body literals, and at most two clauses. In Section 4.6.5, we evaluate how sensitive `Popper` is to these parameters. For each BK relation, we also provide both systems with simple types and argument directions (whether input or output). Because `Popper` and `Enumerate` can generate non-terminating Prolog programs, we set both systems to use a testing timeout of 0.1 seconds per example. If a program times out, we view it as a failure.

Aleph settings. We give Aleph identical mode declarations and determinations to `Popper` and `Enumerate`. We set the maximum variable depth and clause length to six and set the maximum number of search nodes to 30000.

Methods

For each problem, we generate 10 positive and 10 negative training examples, and 1000 positive and 1000 negative testing examples. The default predictive accuracy is therefore 50%. Each list is randomly generated and has a maximum length of 50. We sample the list elements uniformly at random from the set $\{1, 2, \ldots, 100\}$. We measure the predictive accuracy and learning times. We enforce a timeout of five minutes per task. We repeat each experiment 10 times and plot the standard error.

Results

Table 4.5 shows that `Popper` equals or outperforms `Enumerate` on all the tasks in terms of predictive accuracies. When a system has 50% accuracy, it means that the system has failed to learn a program in the given amount of time, and so achieves the default accuracy. Table 4.6 shows that `Popper` substantially outperforms `Enumerate` in terms of learning times. For instance, whereas it takes `Enumerate` 159 seconds to find an `evens` program, it takes `Popper` only four seconds. Table 4.7 decomposes the learning times of `Popper`.

Table 4.5 shows that `Popper` equals or outperforms Metagol on all the tasks in terms of predictive accuracies, except the `finddup` problem, where Metagol has a 2% higher

Name	Description	Example solution
addhead	Prepend the head three times	addhead(A,B):−head(A,C),cons(C,A,D),cons(C,D,E),cons(C,E,B).
dropk	Drop the first k elements	dropk(A,B,C):−one(B),tail(A,C). dropk(A,B,C):−tail(A,D),decrement(B,E),dropk(D,E,C).
droplast	Drop the last element	droplast(A,B):−tail(A,B),empty(B). droplast(A,B):−tail(A,C),droplast(C,D),head(A,E),cons(E,D,B).
evens	Check all elements are even	evens(A):−empty(A). evens(A):−head(A,B),even(B),tail(A,C),evens(C).
finddup	Find duplicate elements	finddup(A,B):−head(A,B),tail(A,C),member(B,C). finddup(A,B):−tail(A,C),finddup(C,B).
last	Last element	last(A,B):−tail(A,C),empty(C),head(A,B). last(A,B):−tail(A,C),last(C,B).
len	Calculate list length	len(A,B):−empty(A),zero(B). len(A,B):−tail(A,C),len(C,D),succ(D,B).
member	Member of a list	member(A,B):−head(A,B). member(A,B):−tail(A,C),member(C,B).
sorted	Check list is sorted	sorted(A):−tail(A,B),empty(B). sorted(A):−head(A,B),tail(A,C),head(C,D),geq(D,B),sorted(C).
threesame	First three elements are identical	threesame(A):−head(A,B),tail(A,C),head(C,B),tail(C,D),head(D,B).

Table 4.4: Example solutions for the list transformation problems.

predictive accuracy. Table 4.5 also shows that Aleph struggles to learn solutions to these problems. The exceptions are `addhead` and `threesame`, which do not need recursion.

Overall, the results from this experiment suggest that (i) the answer to question **Q1** is again yes, constraints improve learning performance, and (ii) `Popper` can outperform other ILP systems when learning complex and recursive list transformation programs.

4.6.4 Scalability

Our buttons experiment (Section 4.6.1) showed that `Popper` scales well in the size of the optimal solution size and the number of background relations. Our robot experiment (Section 4.6.2) showed that `Popper` scales well in the size of the domain. The purpose of this experiment is to evaluate how well `Popper` scales in terms of the (i) number of examples and (ii) the size of examples. To do so, we repeat the `last` experiment from Section 4.6.3, where `Popper` and Metagol achieved similar performance.

Materials

We use the same materials as Section 4.6.3.

Settings

We run two experiments. In the first experiment we vary the number of examples. In the second experiment we vary the size of the examples (the size of the input list). For

Name	Popper	Enumerate	Metagol	Aleph
addhead	**100** ± 0	**100** ± 0	n/a	90 ± 10
dropk	**100** ± 0	50 ± 0	n/a	50 ± 0
droplast	**100** ± 0	50 ± 0	n/a	50 ± 0
evens	**100** ± 0	**100** ± 0	50 ± 0	50 ± 0
finddup	98 ± 0	50 ± 0	**100** ± 0	50 ± 0
last	**100** ± 0	50 ± 0	**100** ± 0	50 ± 0
len	**100** ± 0	50 ± 0	50 ± 0	50 ± 0
member	**100** ± 0	**100** ± 0	**100** ± 0	50 ± 0
sorted	**100** ± 0	50 ± 0	50 ± 0	68 ± 2
threesame	**99** ± 0	**99** ± 0	**99** ± 0	**99** ± 0

Table 4.5: List transformation predictive accuracies. We round accuracies to integer values. The error is standard error.

Name	Popper	Enumerate	Metagol	Aleph
addhead	**0.5** ± 0	2 ± 0	n/a	103 ± 49
dropk	**0.8** ± 0	300 ± 0	n/a	3 ± 0.2
droplast	**3** ± 0.1	300 ± 0	n/a	300 ± 0
evens	4 ± 0.1	159 ± 0.1	300 ± 0	**1** ± 0
finddup	36 ± 2	300 ± 0	2 ± 0.5	**1.0** ± 0.1
last	2 ± 0.1	300 ± 0	**0.7** ± 0.2	1 ± 0.1
len	12 ± 0.3	300 ± 0	300 ± 0	**1** ± 0
member	0.4 ± 0.1	7 ± 0	**0.3** ± 0	0.9 ± 0.1
sorted	23 ± 1	300 ± 0	300 ± 0	**0.8** ± 0
threesame	**0.2** ± 0.1	0.4 ± 0.2	0.9 ± 0.3	0.5 ± 0

Table 4.6: List transformation learning times. We round times over 1 second to the nearest second. The error is standard error. Note that although Aleph is sometimes faster than Popper, it only learns accurate solutions for addhead and threesame.

each experiment, we measure the predictive accuracy and learning times averaged over 10 repetitions.

Number of examples. For each n in $\{1000, 2000, \ldots, 10000\}$, we generate n positive and n negative training examples, and 1000 positive and 1000 negative testing examples and each element is a random integer from the range 1 to 1000.

Example size. For each s in $\{50, 100, 150, \ldots, 500\}$, we generate 10 positive and 10 negative training examples, and 1000 positive and 1000 negative testing examples, where each list is of length s and each element is a random integer from the range 1 to 1000.

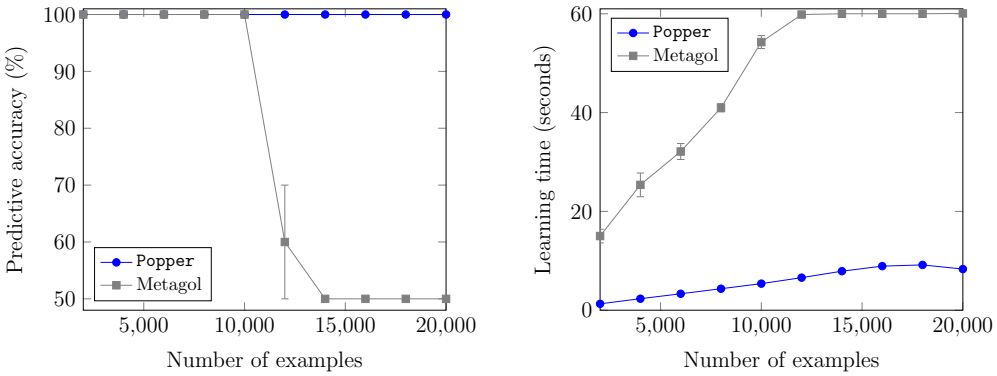
Results

Figure 4.17 shows the results when varying the number of training examples. The predictive accuracies of Popper and Metagol are almost identical until around 10,000 examples.

Name	Time	Grounding	Solving
addhead	0.5 ± 0	0.1 ± 0	0.2 ± 0
dropk	0.8 ± 0	0.3 ± 0	0.1 ± 0
droplast	3 ± 0.1	0.4 ± 0.1	1 ± 0
evens	4 ± 0.1	1 ± 0	1 ± 0.1
finddup	36 ± 2	25 ± 1	7 ± 0.5
last	2 ± 0.1	1 ± 0	0.5 ± 0
len	12 ± 0.3	7 ± 0.2	2 ± 0.1
member	0.4 ± 0.1	0.1 ± 0	0.1 ± 0
sorted	23 ± 1	12 ± 0.9	8 ± 0.6
threesame	0.2 ± 0.1	0 ± 0	0 ± 0

Table 4.7: Decomposition of Popper learning times. The unaccounted time (time not grounding or solving) is mostly the overhead of testing the induced Prolog programs.

Given this many examples, Metagol struggles to find a solution in one minute and eventually converges on the default predictive accuracy (50%). By contrast, Popper does not struggle to find a solution, even given 20,000 examples. Figure 4.17 shows the learning times of both systems. The learning time of Popper increases linearly simply because of the overhead of testing hypotheses on more examples. The results from this experiment suggest that the answer to **Q2** is that Popper scales well with respect the number of examples.

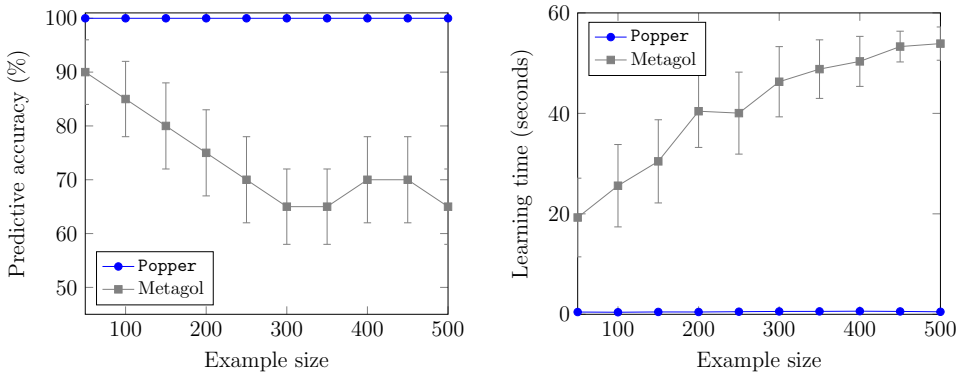


(a) Predictive accuracies (b) Learning times

Figure 4.17: The experimental results for the last task when varying the number of training examples.

Figure 4.18 shows the results when varying the size of the input (i.e. the size of the input list). Popper outperforms Metagol in all cases. The mean learning times of Popper for examples of length 50 and 500 are both less than a second. The reason is that Popper only uses the examples to test a hypothesis, so any increase in running time simply comes from executing the hypotheses using Prolog. By contrast, Metagol's performance drastically degrades as the size of the examples grow. The mean learning times for Metagol for examples of length 50 and 500 are 20 and 54 seconds respectively. The reason is that Metagol uses the examples to search for a hypothesis by inducing and executing partial programs over the examples. These results suggest that the answer to **Q3** is yes and the

answer to **Q2** is that `Popper` scales well with respect to the size of examples.



(a) Predictive accuracies (b) Learning times

Figure 4.18: The experimental results for the `last` task when varying the size (list length) of training examples.

4.6.5 Sensitivity

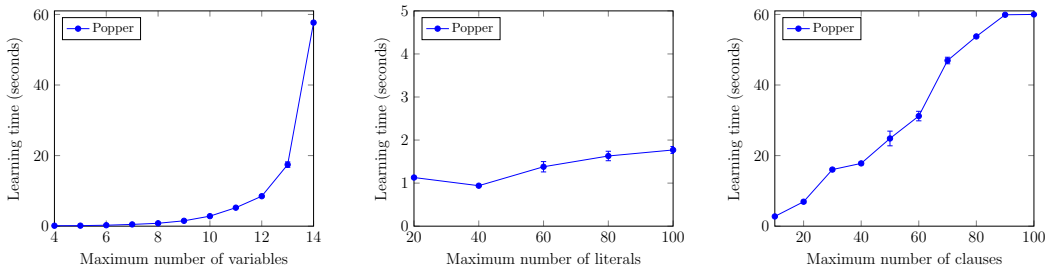
The Learning From Failures hypothesis space (Proposition 4.2.17) is a function of the number of predicate declarations and three other variables:

- the maximum number of unique variables in a clause
- the maximum number of body literals allowed in a clause
- the maximum number of clauses allowed in a hypothesis

The purpose of this experiment is to evaluate how sensitive Popper is to these parameters. To do so, we repeat the `len` experiment from Section 4.6.3 with the same BK, settings, and method, except we run three separate experiments where we vary the three aforementioned parameters.

Results

Figure 4.19 shows the experimental results. The results show that Popper is sensitive to the maximum number of unique variables, which has a strong influence on learning times. This result follows from Proposition 4.2.17 because more variables implies more ways to form literals in a clause. Somewhat surprisingly, doubling the number of variables from 4 to 8 has little difference on performance, which suggests that Popper is robust to imperfect parameters. The results show that Popper is mostly insensitive to the maximum number of body literals in a clause. The main reason is that Popper does not pre-compute every possible clause in the hypothesis space, which is, for instance, the case with ILASP2i and ILASP3. The results show that Popper scales linearly with the maximum number of clauses. Overall these results suggest that Popper scales well with the maximum number of body literals, but can struggle with very large values for the maximum number of unique variables and clauses.



(a) (b) (c)

Figure 4.19: The experimental results for the `len` task when varying the maximum number of unique variables (a), maximum body literals in a clause (b), and maximum number of clauses (c).

Chapter 5

Learning Programs by Explaining their Failures

This chapter extends the Learning From Failures approach to Inductive Logic Programming with the notion of *explaining* failures. Like in Conflict-Driven Clause Learning for SAT/SMT, see Section 3.3, the idea is to analysis a hypothesis' failure and determine which part of the hypothesis was actually responsible. We show that explaining failures leads to learning smaller, more effective constraints, which in turn can lead to reduced hypothesis space exploration and learning times.

5.1 Introduction

Explanations are ubiquitous in our cognitive lives [79]. They are crucial to the process of forming hypotheses, testing them on data, analysing the results, and forming new hypotheses, that is to say, to science [132]. For instance, imagine Alice is a chemist trying to synthesise a vial of a compound from two substances. Alice can perform actions, such as fill a vial with a substance (*fill(Vial,Sub)*) or mix two vials (*mix(V1,V2,V3)*), and sequence them to form a hypothesis, e.g.:

$$synth(A,B,C) \leftarrow fill(V1,A),\ fill(V1,B),\ mix(V1,V1,C)$$

This hypothesis says that to synthesise a vial of compound C, fill vial $V1$ with substance A, fill vial $V1$ with substance B, and mix vial $V1$ with itself to form C.

When Alice experimentally tests this hypothesis she finds that it *fails*. From this failure Alice concludes (**C1**) that hypotheses which add further actions (i.e. literals) will also fail. However, as Alice observed that the second action caused the failure, she can *explain* the failure as "vial $V1$ cannot be filled a second time". This allows her to conclude (**C2**) that any hypothesis that includes *fill(V1,A)* and *fill(V1,B)* will fail. Clearly, conclusion **C2** allows Alice to eliminate more hypotheses than **C1**. That is, by explaining failures Alice can better form new hypotheses.

We formalise this mode of reasoning for explaining failures of logical theories. We do so in the context of inductive program synthesis, where the goal is to machine learn computer programs from data [145]. Existing inductive logic programming (ILP) approaches fail to generalise from observed failures. Many ILP systems [18, 26, 2], including `Popper`, only learn from the failure of an entire hypothesis – as Alice does when she concludes **C1** –

and cannot explain why a hypothesis fails, e.g. cannot reason like Alice does to conclude **C2**. Some systems can identify parts of a program that cause a failure, but cannot learn from this information. For instance, Metagol [39] will repeatedly retry failing program fragments.

We address these limitations by automatically explaining program failures, taking inspiration from algorithmic debugging [20]. The idea is to analyse the failure of a hypothesis to identify *sub-programs* that also fail. To illustrate, consider hypothesis H_1:

$$\{\ \textit{droplast(A,B)} \leftarrow \textit{empty(A),tail(A,B)}\ \}$$

If *droplast([1,2],[1])* is a positive example, then H_1 does not cover this example. From this failure we can learn that H_1's sub-program $\{\ \textit{droplast(A,B)} \leftarrow \textit{empty(A)}\ \}$ also does not cover this example. We show that by identifying failing sub-programs and accumulating constraints generated from them, we can eliminate more hypotheses (e.g. any single clause program that expands the above sub-program). When the overhead of failure explanation is low, our approach reduces learning times.

Most logic program debugging systems [84, 154] and some synthesis systems [145, 139] can identify a subset of clauses as being the cause of a failure. We additionally identify literals *within* clauses responsible for failure (without the requirement of trace-complete examples needed by theory revision systems such as FORTE [141]). We show that this fine-grained failure analysis allows for learning finer-grained constraints on the hypothesis space.

We extend the Learning From Failures framework from the previous chapter (Section 4.2) with the notion of failure explanation. In terms of LFF's three-stage loop, this means extending the test stage with a procedure for finding sub-programs which are the cause of incompleteness/inconsistency of a guessed program with the examples. By the SMT-perspective on LFF (Section 4.2.4) – where we see the test-stage as a *Horn*-theory solver – failure explanation corresponds to the notion of *conflict explanation* [12]. That is, instead of only detecting that the formulas it has been given are in conflict, the theory solver additionally figures out *which* (parts) of the formulas are responsible for the conflict and uses that to derive a conflict to hand to the SAT-solver.

The contributions of this chapter are:

- We relate logic programs that fail on examples to their failing sub-programs. For wrong answers we identify clauses. For missing answers we additionally identify literals within clauses.
- We show that hypotheses that are specialisations and generalisations of failing sub-programs can be eliminated, and prove that hypothesis space pruning based on sub-programs is more effective than pruning without them.
- We show that in the framework of Conflict-Driven ILP, failure explanation applied to Horn hypotheses corresponds to conflict explanation as done by CDCL-based SAT-solvers and theory solvers in SMT-solvers.
- We extend the `Popper` ILP system by adding conflict explanation capabilities to the test stage, in particular by making the test stage analyse SLD-trees to identify sub-programs that cause failure.
- We experimentally show that failure explanation can drastically reduce (i) hypothesis space exploration and (ii) learning times.

5.2 Explaining failures in terms of sub-programs

We extend the Learning From Failures framework, Section 4.2, with the capacity to explain failures of a program in terms of *sub-programs*. We address exactly the same LFF problems, $(E_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \mathcal{C})$, as in the last chapter. We introduce the necessary notions to analyse which examples cause which parts of programs to exhibit which kinds of failures. We show that identifying the sub-programs responsible for a failure can be used to learn more effective constraints.

First, we introduce our running example:

Example 5.2.1. Consider an LFF-input $(E_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}^+ \cup \neg\mathcal{E}^-, \mathcal{C} = \emptyset)$ for learning a *droplast/2* program. Suppose the set of hypotheses, $\mathcal{H}$, represented by $E_{\mathcal{H}}$ are the definite programs with *droplast/2* in the head of each clause and *droplast/2*, *empty/1*, *head/2*, *tail/2* and *cons/3* occurring in bodies. Our background knowledge $\mathcal{BK}$ consists of definitions for these predicates, except for *droplast/2*. $\mathcal{E}^+ = \{droplast([1,2,3],[1,2]), droplast([1,2],[1])\}$ and $\mathcal{E}^- = \{droplast([1,2],[])\}$ are our positive and negative examples. Our set of hypothesis constraints $\mathcal{C}$ is initially empty.

5.2.1 Missing and incorrect answers

Given Background Knowledge $\mathcal{BK}$, the failure of a hypothesis H is due to at least one example. We adopt the following terminology from the algorithmic debugging community [145, 20]. A positive example e^+ is a *missing answer* when $\mathcal{BK} \cup H \not\models e^+$. Similarly, a negative example e^- is an *incorrect answer* when $\mathcal{BK} \cup H \models e^-$. We relate missing and incorrect answers to specialisations and generalisations. If H has a missing answer e^+, then, as a specialisation H' of H entails at most as much as H, e^+ is a missing answer of H' as well. Hence all specialisations of H are incomplete and can be eliminated. Similarly, as generalisations of H entail at least as much as H, if e^- is an incorrect answer of H, all generalisations of H are inconsistent and can be pruned.

Example 5.2.2 (**Missing answers and specialisations**)**.** Given the LFF input from Example 5.2.1, consider the following *droplast* hypothesis:

$$H_1 = \left\{ \begin{array}{l} droplast(A,B) \leftarrow empty(A),tail(A,B) \end{array} \right\}$$

Both $droplast([1,2,3],[1,2])$ and $droplast([1,2],[1])$ are missing answers of H_1, so H_1 is incomplete and we can prune its specialisations, e.g. programs that add literals to the clause – see Proposition 4.3.14.

Example 5.2.3 (**Incorrect answers and generalisations**)**.** Consider hypothesis H_2:

$$H_2 = \left\{ \begin{array}{l} droplast(A,B) \leftarrow tail(A,C),tail(C,B) \\ droplast(A,B) \leftarrow tail(A,B) \end{array} \right\}$$

In addition to being incomplete, H_2 is inconsistent because of the incorrect answer $droplast([1,2],[])$, so along with specialisations we can prune the generalisations of H_2, e.g. programs with additional clauses – see Proposition 4.3.11.

5.2.2 Failing sub-programs

We now consider explaining failures in terms of failing sub-programs. The idea is to identify sub-programs that cause the failure. Consider the following two examples:

Example 5.2.4 (**Explain missing answer**)**.** Consider H_1 and positive example $e^+ = \mathit{droplast}([1,2],[1])$ from Example 5.2.2. An explanation for why H_1 does not entail e^+ is that $\mathit{empty}([1,2])$ fails. It follows that e^+ is a missing answer of

$$H_1' = \{\ \mathit{droplast(A,B)} \leftarrow \mathit{empty(A)}\ \}$$

As H_1' is incomplete we can prune all of its specialisations.

Example 5.2.5 (**Explain incorrect answer**)**.** Now consider negative example $e^- = \mathit{droplast}([1,2],[])$ and H_2 from Example 5.2.3. The first clause of H_2 always entails e^- irrespective of other clauses being part of the hypothesis. It follows that e^- is an incorrect answer of

$$H_2' = \{\ \mathit{droplast(A,B)} \leftarrow \mathit{tail(A,C)}, \mathit{tail(C,B)}\ \}$$

As H_2' is inconsistent we can prune all of its generalisations.

Note that when a system like `Popper` observes that H_2 fails, it is not able to prune based on H_2'. An ILP system like ProSynth [139] can expend additional effort to learn that H_2' fails. Given H_1 and its failure, `Popper`, the ILASP systems and ProSynth are unable to determine it is possible to prune based on H_1'.

We now formalize the parts of a program we can use for pinpointing reasons for failures:

Definition 5.2.6 (**Sub-program**)**.** A Horn program P is a *sub-program* of a Horn program Q if and only if either:

- P is the empty set
- there exists clauses $C_p \in P$ and $C_q \in Q$ such that $C_p \subseteq C_q$ and $P \setminus \{C_p\}$ is a sub-program of $Q \setminus \{C_q\}$

In this definition, arguments of literals must be syntactically the same[1] for the clause subset check to succeed. In functional program synthesis, sub-programs are typically defined by leaving out nodes in the parse tree of the original program (e.g., [51]). Our definition generalises this idea by allowing for arbitrary ordering of clauses and literals.

In the above examples, H_1' is a *definite* sub-program of H_1 and so is H_2' of H_2, i.e. *Horn* sub-programs which happen to be definite programs. Note that clauses and literals can be dropped at the same time, e.g. $\{\ \mathit{droplast(A,B)} \leftarrow \mathit{tail(A,C)}\ \}$ is another (definite) sub-program of H_2.

We define the failing sub-programs problem:

Definition 5.2.7 (**Failing sub-programs**)**.** Given definite program P and sets of examples $\mathcal{E}^+$ and $\mathcal{E}^-$, the *failing sub-programs problem* is to find all sub-programs of P that do not entail an example of $\mathcal{E}^+$ or do entail an example of $\mathcal{E}^-$.

By definition, a failing sub-program has a missing answer and/or an incorrect answer. Hence we can always prune specialisations and/or generalisations of a failing sub-program. We show that sub-programs are effective at pruning:

[1] Our definition hence insists on variable names in literals of a sub-program Q being the same as variable names in the corresponding literals of program P.

Theorem 5.2.8. Let H be a definite program that fails and P ($\neq H$) be a sub-program of H that fails. Let $C(H)$ and $C(P)$ be the specialisation and/or generalisation constraints derivable for H and P, respectively. If neither of (i) *P is a specialisation of H and H is incomplete and P is not inconsistent,* or (ii) *P is a generalisation of H and H is inconsistent and P is not incomplete*, apply, then $\mathcal{H}_{C(H)\cup C(P)} \subset \mathcal{H}_{C(H)}$, i.e. there programs that are prunable programs based P prune programs not pruned by constraints derived for H.

Proof. By case distinction on how P and H are related by subsumption. Note that because $P \neq H$, either P and H are not related by subsumption, or P subsumes H, or H subsumes P.

Suppose H subsumes P, i.e. P is a specialisation of H. If H is incomplete, then all of H's specialisations can be pruned, which includes P and its specialisations. Hence if P is only incomplete then no additional pruning can be achieved, which is exception (i). If P is (additionally) inconsistent, then P's generalisations can be pruned. In addition to H being among P's generalisations, there are also programs incomparable with H among P's generalisations, so more pruning can be achieved.

Now suppose P subsumes H, i.e. P is a generalisation of H. If H is inconsistent, then all of H's generalisations can be pruned, which includes P and its generalisations. Hence if P is only inconsistent then no additional pruning can be achieved, which is exception (ii). If P is (additionally) incomplete, then P's specialisations can be pruned. In addition to H being among P's specialisations, there are also programs incomparable with H among P's specialisations, so more pruning can be achieved.

In the remaining case, where H and P are not related by subsumption, it is immediate that the specialisation/generalisation constraints derived for P prune a distinct part of the hypothesis space, e.g. H's constraints do not prune P. □

5.3 Failure explanation algorithm

We now present a method for identifying failing sub-programs. The approach is based on the observation that branches of an SLD-tree correspond to definite sub-programs. Our algorithm identifies clauses responsible for entailing a negative example. It is when a program fails to prove entailment that our approach distinguishes itself. That is, we also identify literals *within* clauses which cause a positive example to not be entailed. As the presented method relies on SLD-resolution, from this point in this chapter on we assume left-to-right evaluation of literals within clauses, treat our definite logic programs as Prolog programs and use `typewriter` font to highlight this.

5.3.1 SLD-trees

In algorithmic debugging, missing and incorrect answers help characterise which parts of a *debugging tree* are wrong [20]. Debugging trees can be seen as generalising SLD-trees, with the latter representing the search for a refutation [122]. We address the failing sub-programs problem by analysing SLD-trees, only identifying a subset of them. A *branch* in an SLD-tree is a path from the root *goal* to a leaf. Each goal on a branch has a *selected atom*, on which resolution is performed to derive child goals. A branch that ends in an empty leaf is called *successful*, as such a path represents a refutation. Otherwise a branch

```
1 def failing_subprogs⁻(B, H, e⁻):
2     T = SLD-tree of B ∪ H ∪ {¬e⁻}
3     subprogs = {}
4     for every successful branch λ of T:
5         H′ = sub-program of H identified by
6               H's clauses that occur in λ
7         subprogs = subprogs ∪ {H′}
8     return subprogs
```

```
1 def failing_subprogs⁺(B, H, e⁺):
2     T = SLD-tree of B ∪ H ∪ {¬e⁺}
3     subprogs = {}
4     for every failing branch λ of T:
5         H′ = sub-program of H identified by
6               H's literals that occur in λ
7         if SLD-res. fails to prove B ∪ H′ ⊨ e⁺:
8             subprogs = subprogs ∪ {H′}
9     return subprogs
```

Figure 5.1: Identify failing sub-programs from branches in SLD-trees

is *failing*. Note that selected atoms on a branch identify a subset of the literals of a program.

5.3.2 Identifying sub-programs

Let B be a definite program, H be a hypothesis, and e be an atom[2]. The SLD-tree T for $B \cup H \cup \{\neg e\}$, with $\neg e$ as the root, proves $B \cup H \models e$ iff T contains a successful branch. Given a branch λ of T, we define the λ-sub-program of H: a literal L of H occurs in *λ-sub-program* H' if and only if L occurs as a selected atom[3] in λ or L was used to produce a resolvent that occurs in λ. The former case is for literals in the body of clauses and the latter for head literals. Now consider the SLD-tree T' for $B \cup H' \cup \{\neg e\}$ with $\neg e$ as root. As all literals necessary for λ occur in $B \cup H'$, the branch λ must occur in T' as well.

Suppose e^- is an incorrect answer for hypothesis H. Then the SLD-tree for $B \cup H \cup \{\neg e^-\}$ has a successful branch λ. The literals of H necessary for this branch are also present in λ-sub-program H', hence e^- is also an incorrect answer of H'. Now suppose e^+ is a missing answer of H. Let T be the SLD-tree for $B \cup H \cup \{\neg e^+\}$ and λ' be any failing branch of T. The literals of H in λ' are also present in λ'-sub-program H''. While λ' must be a failing branch present in the SLD-tree of $B \cup H'' \cup \{\neg e^+\}$, this is, in general, insufficient for concluding that this SLD-tree has no successful branch. Hence whether e^+ is indeed a missing answer of H'' needs to be verified.

Figure 5.1 shows the procedures for deriving failing sub-programs, in the case of a negative example and a positive example, respectively. Note that hypothesis H can refer to library B but B is not allowed to refer to H. Hence whilst resolving a selected literal of H defined by B with clauses of B we cannot encounter literals of H. Therefore, for failure explanation purposes, we need not inspect the part of the SLD-tree for $B \cup H \cup \{\neg e\}$ that deals with determining whether a literal defined by B holds or not. This is equivalent to viewing B as a (possibly infinite) set of facts, i.e. resolving a selected literal defined by B always returns directly[4]. This is how we will treat resolving literals of B from this point on.

The following example illustrates identifying sub-programs from the SLD-trees of a recursive program.

[2]While in our application to synthesis we only use ground atoms e, the failure explanation algorithm presented in this section also works when e is non-ground.

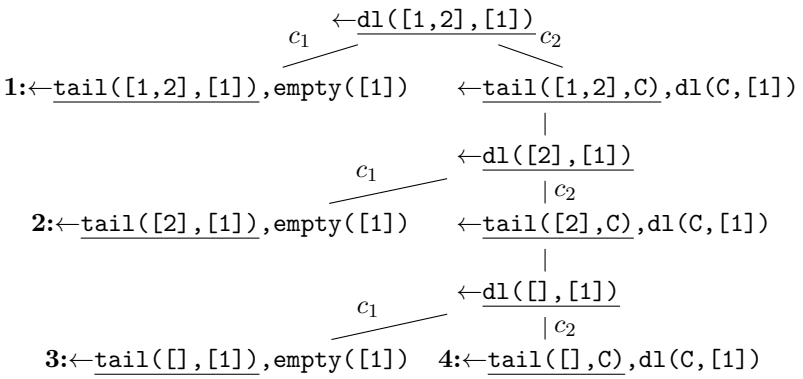
[3]Note that resolution might have unified arguments of L to produce the selected atom.

[4]For a query against B to be guaranteed to return we do need that querying B is at least decidable.

Example 5.3.1. Let H be the following recursive `droplast/2` hypothesis, where the name `droplast` has been shortened to `dl`:

$$c_1 : \texttt{dl(A,B):- tail(A,B),empty(B).}$$
$$c_2 : \texttt{dl(A,B):- tail(A,C),dl(C,B).}$$

Suppose B includes the usual definitions for `tail/2` and `empty/1`. Testing whether $B \cup H \models$ `dl([1,2],[1])` holds is done by SLD-resolution. The SLD-tree for $B \cup H \cup \{\neg$`dl([1,2],[1])`$\}$ is:



Each node is a goal and has its selected literal underlined. The SLD-tree has four branches, each of them failing. The branch marked '**1:**' identifies the sub-program P_1 = { `dl(A,B):- tail(A,B).` } as only clause c_1 is used and only its head and first body literal are evaluated. The branches marked '**2:**' and '**3:**' identify the sub-program P_2 = { `dl(A,B):- tail(A,B).` `dl(A,B):- tail(A,C),dl(C,B).` } as both clauses are used though the second literal of c_1 is never selected while all of the literals of c_2 are. The branch marked '**4:**' never uses clause c_1 and hence identifies sub-program $P_3 = \{c_2\}$. Retesting `dl([1,2],[1])` on these sub-programs confirms that they fail.

Now consider testing for $B \cup H \models$ `dl([1,2],[])`. The SLD-tree for $B \cup H \cup \{\neg$`dl([1,2],[])`$\}$ has failing branches but also a successful one: ←`dl([1,2],[])` $\overset{c_2}{—}$ ←`tail([1,2],C),dl(C,[])` — ←`dl([2],[])` $\overset{c_1}{—}$ ←`tail([2],[]),empty([])` — ←`empty([])` — □. As this branch used all clauses, it identifies H itself as responsible. On the other hand, the SLD-tree for $B \cup H \models$ `dl([1],[])` has a successful branch only using c_1: ←`dl([1],[])` $\overset{c_1}{—}$ ←`tail([1],[]),empty([])` — ←`empty([])` — □. Hence it identifies $P_4 = \{c_1\}$ as the responsible sub-program.

5.4 Conflict-Driven Constraint Learning

We extend the three-stage approach to solving Learning From Failures problems by introducing failure explanation to the test stage. Like the Conflict-Driven ILP systems – IlASP [95], ProSynth [139], and ASPsynth [13] – we make the solver that evaluates our hypotheses identify which parts of a hypothesis caused the failure. Whereas these systems only identify whole clauses as being responsible, we use our preceding failure explanation algorithm to also identify literals within clauses as being responsible. Unlike these systems, we then hand over the failing sub-programs to a component that derives subsumption-based constraints, thereby generalising the learned conflicts.

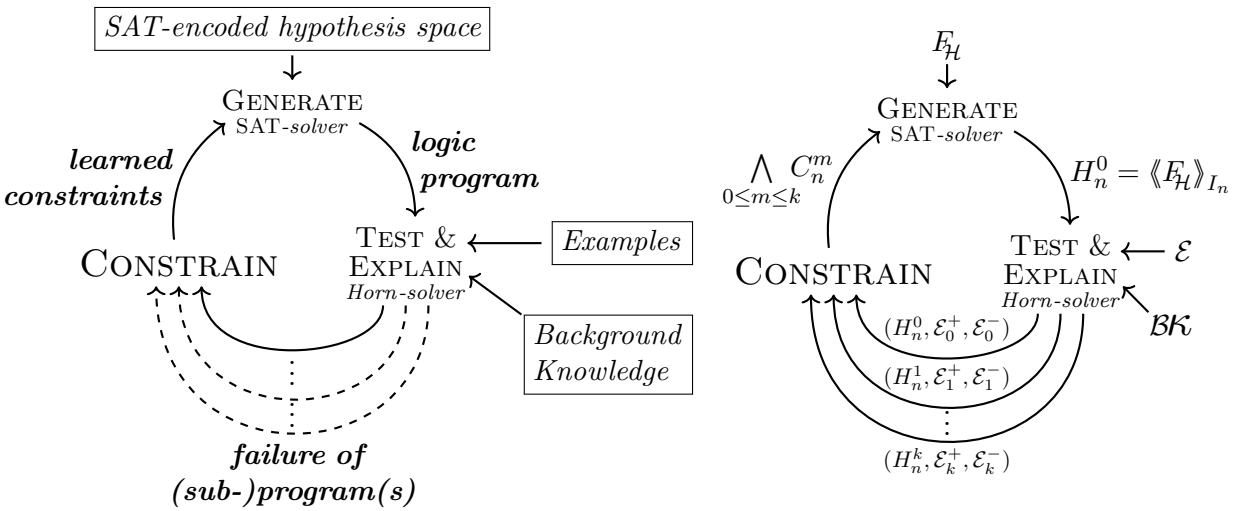


Figure 5.2: The generate, test-and-explain, and constrain loop.

As will become clear in the experimental section, the effectiveness of our approach stems in large part from the ability of the test-and-explain stage to identify failing sub-programs that themselves are not considered well-formed programs. This allows for identifying small program fragments that are not hypotheses that do specialise or generalize many hypotheses. Hence fine-grained failure explanation makes it possible to learn very effective constraints from non-hypothesis programs.

Note that the other CDILP systems do not enjoy the same benefit from their failure explanation. As the existing CDILP systems are rule-selection based and have (close to) all rule combinations count as hypotheses, any identified failing sub-program, i.e. a subset of rules, is just another hypothesis they could have generated instead (of the program they applied failure explanation to) had they had a better search heuristic.

The SMT-perspective

When we view Learning From Failures through the lens of Satisfiability Modulo Theories – see Section 4.2.4 and Section 4.5.2 – we can understand failure explanation methodology as essentially making the *Horn*-solver perform conflict explanation [17], usually using techniques to identify unsat-cores [100]. In normal SMT-solving, such reasoning helps theory solvers identify smaller conflict clauses to inject into the SAT-solver. Instead of injecting these detected conflicts as conflict clauses, we generalize the conflicts to constraints that also prune related (sets of) *Horn*-formulas.

Addressing an LFF-input $(E_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \mathcal{C})$, Figure 5.2 illustrates how the simple three-stage loop from Section 4.4 is expanded. Instead of the test stage returning just the failure of H_n^0, it also returns sub-programs H_n^1 through H_n^k of H_n^0, along with the reason why they failed. Note that $k = 0$ is possible, i.e. no additional (sub-)programs are identified, as it might well be the case that there is no proper sub-program that causes the failure. The constrain stage operates as in Section 4.4, only now it conjoins all the subsumption-based constraints for both H_n^0 and for its failing sub-programs.

5.5 Implementation

Before introducing `Hempel`, our extension of the `Popper` ILP system (Section 4.5), we discuss our implementation of the failure explanation algorithm.

5.5.1 Meta-interpreter for failure explanation

We implement our failure explanation algorithm by a meta-interpreter, $\texttt{mi}_{tr}$, where this meta-interpreter is best understood as instrumenting the program such that executing it keeps track of which parts of the program actually got executed.

Given a program B that serves as a library and an atom G, $\texttt{mi}_{tr}$ keeps track of which literals of a definite program P have been encountered along each branch of the SLD-tree of $B \cup P \cup \{\neg G\}$. For each literal of the hypothesis P being evaluated we keep track of one bit of information: whether this literal[5] has been seen along the current branch or not. $\texttt{mi}_{tr}$ maintains a bitset, which we refer to as a *trace*, containing a unique bit for each literal of the hypothesis.

The meta-interpreter assumes a program transformation $X(\cdot)$ has been applied to the program (where, for notational convenience, clauses are represented by disjunctions):

$$X(P) \quad = \quad \{\bigvee \begin{cases} \neg X(A, C_{idx}, L_{idx}) & \textit{if } L = \neg A \\ X(L, C_{idx}, L_{idx}) & \textit{otherwise} \end{cases} \quad | \; L \in C \wedge C \in P\}$$

Before defining $X(\cdot,\cdot,\cdot)$, we specify how bitsets are derived. C_{idx} and L_{idx} correspond to the index of clause C within P and the index of L within C, respectively. The function $\textit{bitset}(\cdot,\cdot)$ converts a clause index and literal index within that clause to a bitset with a unique bit set for these inputs. $X(L, C_{idx}, L_{idx})$ `:=` $\texttt{mi}(L, \textit{bitset}(C_{idx}, L_{idx}))$, if the predicate of L is defined by P. Otherwise $X(L, C_{idx}, L_{idx})$ `:=` $\texttt{call}(L, \textit{bitset}(C_{idx}, L_{idx}))$, i.e. in the case the predicate of L is defined by the (Background Knowledge) program B.

Figure 5.3 lists the code for meta-interpreter $\texttt{mi}_{tr}$. Given an atom G and program $X(P)$, we can evaluate goal G by invoking our meta-interpreter with $\texttt{mi}_{tr}$`(mi(`G`,0),0,Trace)`, where `0` denotes the empty bitset. When this call succeeds, `Trace` will have become unified with a bitset identifying all literals that occurred on the first successful branch in the SLD-tree of $B \cup P \cup \{\neg G\}$. If evaluation of $\texttt{mi}_{tr}$`(mi(`G`,0),0,Trace)` fails then there is no successful branch in the SLD-tree of $B \cup P \cup \{\neg G\}$. In this case $\texttt{mi}_{tr}$ will have asserted traces for each unsuccessful branch, via a non-logical predicate `assert_failed_trace`[6]. Upon $\texttt{mi}_{tr}$`(mi(`G`,0),0,Trace)` having failed, all these asserted traces can be inspected to obtain the corresponding sub-programs.

Note that $\texttt{mi}_{tr}$ only does a constant number of additional (bitset unioning / logical *or*) operations at every node of the SLD-tree of $B \cup P \cup \{\neg G\}$ involving literals of H (that is, resolving literals defined by B is relegated to the normal interpreter). Hence the SLD-tree of $B \cup X(P) \cup \{\neg\texttt{mi}_{tr}\texttt{(mi(}G\texttt{,0),0,Trace)}\}$ is only a constant factor bigger than the original. It follows that the overhead $\texttt{mi}_{tr}$ incurs from identifying sub-programs is directly proportional to the size of the SLD-tree generated during normal execution, i.e. the algorithm for identifying sub-programs has linear complexity (and leaves the part

[5] Note that the meta-interpreter only keeps track of seen literals of the hypothesis, not of any literals occurring in the background knowledge.

[6] *Asserting a trace* can be done in constant time, e.g. by putting the trace in a hashmap or prepending the trace to the front of a list of failed traces.

```
 1  mi_tr(true,Trace,Trace).
 2  mi_tr((HeadOfBody,TailOfBody),Tr_in,Tr_out) :-
 3      mi_tr(HeadOfBody,Tr_in,Tr_mid),
 4      mi_tr(TailOfBody,Tr_mid,Tr_out).
 5  mi_tr(mi(G,I),Tr_in,Tr_out) :-
 6      clause(mi(G,J),Body),
 7      Tr_head is Tr_in ∨ I ∨ J,
 8      mi_tr(Body,Tr_head,Tr_out).
 9  mi_tr(call(G,I),Tr_in,Tr_out) :-
10      Tr_out is Tr_in ∨ I,
11      (call(G) *-> true ; assert_failed_trace(Tr_out),fail).
```

Figure 5.3: Meta-interpreter $\texttt{mi}_{tr}$. $\texttt{mi}_{tr}$ keeps track of a trace of literal indices encountered along each SLD-branch. The $\vee$ operator takes two bitsets and produces their union (like taking the logical *or* of two integers/bit vectors). `call(G)` just interpreters (complex) term `G` as an atom and evaluates it. The semantics of `G *-> Then ; Else` are that if `G` ever succeeds the entire construct acts as if it were `G,Then`, otherwise it acts as if it just were `Else`. `clause(Head,Body)` unifies with any definite clause the Prolog interpreter knows about. `Body` is a cons-list of atoms which terminates in `true`.

of the SLD-tree which is resolving literals of B with clauses of B untouched, incurring no overhead). This approach does not address non-termination issues of (recursive) programs, i.e. if executing the original program led to an infinite branch in the SLD-tree then executing the meta-interpreter instead will also yield an infinite branch. For sub-programs identified on missing answers, we still need to re-evaluate the sub-programs. If $P = \{C_1, \ldots, C_n\}$, then there are $\prod_{1 \leq i \leq n} \#literals(C_i)$ distinct sub-programs of P, i.e. the possible combinations of prefixes of P's clauses, that could be identified for retesting.

5.5.2 `Hempel`

We now introduce `Hempel`, a Learning From Failures ILP system based on `Popper`. `Hempel` extends the `Popper` generate-test-and-constrain system (Section 4.5) by adding support for failure explanation to the test stage, in accord with Section 5.4.

The generate stage is identical to that of `Popper` and searches for a model of the formula which it converts to a program. In the test stage, a thus generated hypothesis H is tested on positive and negative examples. `Hempel` incorporates Figure 5.1 into the test stage, running it for each tested example. Meta-interpreter $\texttt{mi}_{tr}$ is used to determine clauses and literals that occur along branches responsible for a failure. From this information `Hempel` reconstructs the corresponding sub-programs. If sub-program H' is derived from a branch for a missing answer, H' gets retested, this time using standard SLD-resolution. The test stage tells the constrain stage the number of missing and incorrect answers of a (sub-)program. This determines whether its specialisations and/or generalisations and/or redundant hypotheses should be pruned. For each failed hypothesis and each of its identified failing sub-programs, new hypothesis constraints are added to the formula, eliminating models, thereby pruning the hypothesis space. As in general failing

$$\mathcal{H}_1 = \left\{ \begin{array}{l} \mathtt{h}_1 = \{ \texttt{droplast(A,B):- empty(A),tail(A,B).} \} \\ \mathtt{h}_2 = \{ \texttt{droplast(A,B):- empty(A),cons(C,D,A),tail(D,B).} \} \\ \mathtt{h}_3 = \left\{ \begin{array}{l} \texttt{droplast(A,B):- tail(A,C),tail(C,B).} \\ \texttt{droplast(A,B):- tail(A,B).} \end{array} \right\} \\ \mathtt{h}_4 = \{ \texttt{droplast(A,B):- empty(A),tail(A,B),head(A,C),head(B,C).} \} \\ \mathtt{h}_5 = \left\{ \begin{array}{l} \texttt{droplast(A,B):- tail(A,C),tail(C,B).} \\ \texttt{droplast(A,B):- tail(A,B),tail(B,A).} \end{array} \right\} \\ \mathtt{h}_6 = \left\{ \begin{array}{l} \texttt{droplast(A,B):- tail(A,B),empty(B).} \\ \texttt{droplast(A,B):- cons(C,D,A),droplast(D,E),cons(C,E,B).} \end{array} \right\} \\ \mathtt{h}_7 = \left\{ \begin{array}{l} \texttt{droplast(A,B):- tail(A,C),tail(C,B).} \\ \texttt{droplast(A,B):- tail(A,B).} \\ \texttt{droplast(A,B):- tail(A,C),droplast(C,B).} \end{array} \right\} \end{array} \right\}$$

Figure 5.4: LFF hypothesis space considered in Example 5.5.1.

sub-programs need not be specialisations/generalisations of H, pruning for sub-programs is in addition to the pruning which the constrain stage already does for H in `Popper`. Finally, `Hempel` loops back to the generate stage.

`Hempel` vs. `Popper`

Smaller programs prune more effectively, which is partly why `Popper` and `Hempel` search for hypotheses by increasing size[7] (in terms of number of literals). Yet there are many small programs that `Popper` does not consider well-formed that lead to significant pruning. Consider the sub-program $H'_1 = \{$ *droplast(A,B)* $\leftarrow$ *empty(A)* $\}$ from Example 5.2.4. `Popper` does not generate H'_1 as it does not consider it a well-formed hypothesis (as the head variable B does not occur in the body). Yet, precisely because this sub-program has so few body literals is why it is so effective at pruning specialisations.

The following example demonstrates the loop used by `Hempel` and `Popper`, and how failure explanation can lead to fewer loop iterations.

Example 5.5.1. We illustrate `Hempel`, and how it differs from `Popper`, by running its loop on LFF input $(E_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}^+ \cup \neg\mathcal{E}^-, \mathcal{C} = \emptyset)$ from Example 5.2.1. For demonstration purposes we use the simplified hypothesis space $\mathcal{H}_1 \subset \mathcal{H}$ of Figure 5.4. Our positive examples are $e_1^+ = \mathit{droplast}([1,2,3],[1,2])$ and $e_2^+ = \mathit{droplast}([1,2],[1])$, and our negative example is $e_1^- = \mathit{droplast}([1,2],[])$.

First we induce a program by a generate-test-and-constrain loop *without* failure explanation. This first sequence is representative of `Popper`'s execution:

1. `Popper` starts by generating $\mathtt{h}_1$. $\mathcal{BK} \cup \mathtt{h}_1$ fails to entail e_1^+ and e_2^+ and correctly does not entail e_1^-. Hence only specialisations of $\mathtt{h}_1$ are pruned, namely $\mathtt{h}_4$.
2. `Popper` subsequently generates $\mathtt{h}_2$. $\mathcal{BK} \cup \mathtt{h}_2$ fails to entail e_1^+ and e_2^+ and is correct on e_1^-. Hence specialisations of $\mathtt{h}_2$ are pruned, of which there are none in $\mathcal{H}_1$.
3. `Popper` next generates $\mathtt{h}_3$. $\mathcal{BK} \cup \mathtt{h}_3$ does not entail the positive examples, but does entail negative example e_1^-. Hence specialisations and generalisations of $\mathtt{h}_3$ are

[7] The other reason is to find *optimal* solutions, i.e. those with the minimal number of literals.

pruned, meaning only generalisation $\texttt{h}_7$.

4. `Popper` generates $\texttt{h}_5$. $\mathcal{BK} \cup \texttt{h}_5$ is correct on none of the examples. Hence specialisations and generalisations of $\texttt{h}_5$ are pruned, of which there are none in $\mathcal{H}_1$.
5. `Popper` generates $\texttt{h}_6$. $\mathcal{BK} \cup \texttt{h}_6$ is correct on all the examples and hence $\texttt{h}_6$ is returned.

Now we consider learning by a generate-test-and-constrain loop *with* failure explanation. The following execution sequence is representative of `Hempel`:

1. `Hempel` starts by generating $\texttt{h}_1$. $\mathcal{BK} \cup \texttt{h}_1$ fails to entail e_1^+ and e_2^+ and correctly does not entail e_1^-. The failure explanation algorithm identifies sub-program $\texttt{h}'_1 = \{\texttt{droplast(A,B):- empty(A).}\}$. $\texttt{h}'_1$ fails in the same way as $\texttt{h}_1$. Hence specialisations of both $\texttt{h}_1$ and $\texttt{h}'_1$ get pruned, namely $\texttt{h}_2$ and $\texttt{h}_4$.
2. `Hempel` subsequently generates $\texttt{h}_3$. $\mathcal{BK} \cup \texttt{h}_3$ does not entail the positive examples, but does entail negative example e_1^-. Failure explanation identifies sub-program $\texttt{h}'_3 = \{\texttt{droplast(A,B):- tail(A,C),tail(C,B).}\}$. $\texttt{h}'_3$ fails in the same way as $\texttt{h}_3$. Hence specialisations and generalisations of $\texttt{h}_3$ and $\texttt{h}'_3$ get pruned, meaning $\texttt{h}_5$ and $\texttt{h}_7$.
3. `Hempel` next generates $\texttt{h}_6$. $\mathcal{BK} \cup \texttt{h}_6$ is correct on all the examples and hence $\texttt{h}_6$ is returned.

The difference in these two execution sequences is illustrative of how failure explanation – by way of sub-programs – can help prune away significant parts of the hypothesis space.

5.6 Experiments

We claim that failure explanation can improve learning performance. Our experiments therefore aim to answer the questions:

Q1 Can failure explanation prune more programs?

Q2 Can failure explanation reduce learning times?

Note that an affirmative answer to **Q1** does not imply that **Q2** is the case, as potentially the overhead of failure explanation exceeds the benefits of the pruning it achieves.

To answer **Q1** and **Q2**, we compare `Hempel` against `Popper`. The addition of failure explanation is the only difference between the systems. In each of the experiments, the settings for `Hempel` and `Popper` are identical. Though control over a system's failure explanation capabilities is required to help answer **Q1** and **Q2**, we nevertheless include a comparison against state-of-the-art ILP system Metagol [39] and the classical ILP system Aleph [150].

We run the experiments on a 10-core server (at 2.2GHz) with 30 gigabytes of memory (note that all the systems only run on a single core). When testing individual examples, we use an evaluation timeout of 2 milliseconds.

5.6.1 Experiment 1: robot route planning

We first evaluate the potential performance improvement of failure explanation as a function of target program size. We select a contrived setting where failure explanation ought to be very effective: a basic route planning problem. A robot resides in a grid world and can move in four directions. The robot starts in the lower left corner and needs to move to a position to its right. Unbeknownst to the robot, it has been restricted to a corridor

(dimensions 14×1). In this experiment, failure explanation should determine that any strategy that moves up, down, or starts by moving left can never succeed.

Settings. An example is an atom $f(s_1, s_2)$, with start (s_1) and end (s_2) states. A state is a pair of discrete coordinates (x, y). We provide four dyadic relations as BK: *move_right*, *move_left*, *move_up*, and *move_down*, which change the state, e.g. *move_right((2,2),(3,2))*. We ensure that our hypotheses are forward-chained [76], meaning body literals modify the state one after another. We supply Metagol with the following metarules: $P(A, B) \leftarrow Q(A, B)$ and $P(A, B) \leftarrow Q(A, C), R(C, B)$ and $P(A, B) \leftarrow Q(B, A)$.

Systems. In comparing systems, we try to ensure that hypothesis spaces are as similar as possible. For `Hempel`, `Popper` and Aleph we allow one clause with up to 13 body literals and 14 variables. Metagol is the only system that uses predicate invention, i.e. learns clauses with invented predicate symbols. As reusing invented predicates leads to exponentially shorter programs for this problem, we use both Metagol and a version of Metagol where reuse of invented predicates is disabled: METAGOL$_{\not\in}$.

Method. The start state is $(0, 0)$ and the end state is $(n, 0)$, for n in $1, 2, 3, \ldots, 13$. Each trial has only one (positive) example: $f((0, 0), (n, 0))$. We measure learning times and, for `Popper` and `Hempel`, the number of generated programs. We enforce a timeout of 60 seconds per task. We repeat each experiment 10 times and plot the mean and standard error.

Results. Figure 5.5a shows that `Hempel` substantially outperforms `Popper` in terms of learning time. The reason for the improved learning time is that `Hempel` generates far fewer programs, see Figure 5.5b. For example, upon `Hempel` generating one program that starts by moving left, failure explanation determines any program whose first move is to the left is going to fail and hence all these programs get pruned.

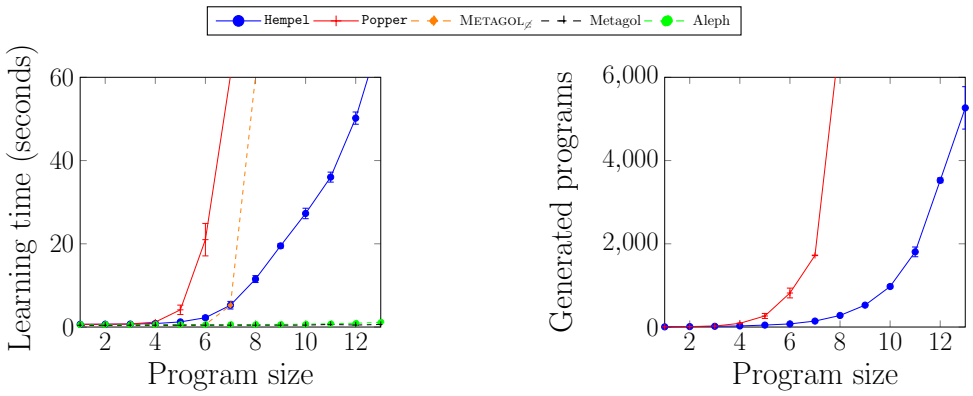
Figure 5.5a also shows that `Hempel` outperforms METAGOL$_{\not\in}$. Due to METAGOL$_{\not\in}$ being example-driven it is effective in pruning programs that try to move out of the corridor, yet, at bigger program sizes its reconsideration of already seen programs is very costly.

Aleph and normal Metagol always find the solution, even at size 13, witin 1.5 seconds. For Metagol, this is due to reusing invented predicates. For example, the size 12 solution that Metagol finds has only eight body literals, versus the 12 that `Hempel` needs. For Aleph, the bottom-clause construction is very effective in only considering moves that are actually allowed. However, the performance of these systems does not have bearing on whether failure explanation is effective or not.

The results from this simple experiment strongly suggest that the answer to questions **Q1** and **Q2** is yes.

5.6.2 Experiment 2: list transformations

This experiment evaluates whether failure explanation can improve performance when learning programs for recursive list problems, which other state-of-the-art ILP systems [95, 49, 76] struggle to solve. We show that `Hempel` can drastically outperform `Popper`,



(a) Learning time.

(b) Number of programs.

Figure 5.5: Results of robot planning experiment. The x-axes denote the number of body literals in the solution, i.e. the number of moves required. Standard error is plotted but is always negligible for `Hempel`.

Metagol and Aleph on the same 10 problems we used to evaluate `Popper` (Section 4.6.3), plus three additional ones: *reverse*, *odd1even2*, *sumlist*.

Settings. We provide as BK the monadic relations *empty*, *zero*, *one*, *even*, *odd*, the dyadic relations *element*, *head*, *tail*, *increment*, *decrement*, *geq*, and the triadic relations *cons*, *snoc*, *sum*. With a single fixed hypothesis space for these problems, `Popper` exhibits significant variance between learning times across problems (ranging from sub-second times for at least four problems to many minutes on others). To control for this variance, we select hypothesis space settings on a per problem basis, such that `Popper` has to do non-trivial search but can still find solutions for each problem within the timeout. See Appendix B.1 for the exact settings.

Systems. For `Hempel` and `Popper`, we provide simple types and mark arguments of predicates as either input or output. For Metagol, we use the same metarules used to evaluate it against `Popper`, listed in Figure 4.15. Because Metagol uses metarules and invented predicates, its hypothesis space is similar but not identical to that of `Hempel` and `Popper`. For Aleph we provide mode declarations and determinations which encode the exact same information made available to `Hempel`. We use the same Aleph settings used to compare it against `Popper`: we set the maximum variable depth and clause length to six and the number of search nodes is limited to 30000.

Method. We generate 10 positive and 10 negative examples per problem. Each example is randomly generated from lists up to length 50, whose integer elements are sampled from 1 to 100. We test on 100 positive and 100 negative randomly sampled examples, giving a default accuracy of 50%. We measure learning time, number of programs generated and predictive accuracy. We also measure the time spent in the three distinct stages of `Popper` and `Hempel`. We repeat each experiment 20 times and record the mean and standard error. We enforce a 60-second timeout.

	Number of programs			Learning time (sec)			Accuracy	
Problem	`Popper`	`Hempel`	**ratio**	`Popper`	`Hempel`	**ratio**	`Popper`	`Hempel`
dropk	2585 ± 184	121 ± 57	**0.05**	35 ± 6	4 ± 2	**0.11**	99 ± 3	99 ± 3
sumlist	2619 ± 23	127 ± 17	**0.05**	47 ± 3	3 ± 0.7	**0.07**	100 ± 0	100 ± 0
len	2826 ± 19	172 ± 18	**0.06**	50 ± 3	3 ± 0.4	**0.06**	100 ± 0	100 ± 0
last	477 ± 91	63 ± 25	**0.13**	13 ± 4	2 ± 0.6	**0.15**	100 ± 0	100 ± 0
droplast	1718 ± 117	242 ± 75	**0.14**	41 ± 8	7 ± 2	**0.18**	100 ± 0	100 ± 0
odd1even2	1324 ± 272	289 ± 98	**0.22**	17 ± 5	4 ± 2	**0.26**	100 ± 0	100 ± 0
member	173 ± 36	64 ± 13	**0.37**	31 ± 10	17 ± 6	**0.54**	100 ± 0	100 ± 0
threesame	136 ± 44	72 ± 41	**0.53**	10 ± 6	5 ± 4	**0.50**	100 ± 0	100 ± 0
finddup	1167 ± 82	653 ± 51	**0.56**	10 ± 1	7 ± 0.6	**0.66**	99 ± 1	99 ± 1
addhead	71 ± 24	41 ± 16	**0.57**	5 ± 2	5 ± 2	**0.98**	100 ± 0	100 ± 0
sorted	861 ± 221	712 ± 148	**0.83**	32 ± 12	28 ± 8	**0.87**	**99 ± 4**	98 ± 5
reverse	1227 ± 424	1025 ± 435	**0.84**	29 ± 8	28 ± 10	**0.97**	100 ± 0	100 ± 0
evens	786 ± 7	754 ± 9	**0.96**	14 ± 0.9	16 ± 0.9	1.14	100 ± 0	100 ± 0

Table 5.1: Results for `Hempel` and `Popper` for Experiment 2. Left, the average number of programs generated by each system. Middle, the (corresponding) average time to find a solution. Right, the average accuracy of solutions. The error is standard error. We round values over one to the nearest integer. Values under one we round to the most significant digit.

	Number of programs			Total time (sec)		
Problem	`Popper`	`Hempel`	**ratio**	`Popper`	`Hempel`	**ratio**
addhead*	42 ± 0.0	25 ± 0.8	**0.58**	5 ± 0.1	4 ± 0.2	**0.87**
reverse*	770 ± 2	539 ± 7	**0.70**	20 ± 0.9	17 ± 0.9	**0.83**
sorted*	599 ± 15	477 ± 9	**0.80**	21 ± 2	18 ± 1	**0.85**

Table 5.2: Selection of programming puzzles for which there was high variance in Table 5.1. Hypotheses spaces for these problems have been pre-pruned of all programs whose size is at least as large as that of the smallest solution. Total time measures the time, in seconds, required to show there is no solution in these hypothesis spaces.

Results. `Hempel`'s accuracy is at least 98% on all problems, see Table 5.1. Both `Hempel` and `Popper` always terminate before the timeout and score 100% on the same ten problems.

Table 5.1 shows the learning times in relation to the number of programs generated. Crucially, it includes the ratio of the mean of `Hempel` over the mean of `Popper`. On these 13 problems, `Hempel` always considers fewer hypotheses than `Popper`. On seven problems less than 50% of the original number of programs is considered while only on three problems over 80% is still needed.

To illustrate why failure explanation is effective, we consider the *dropk* problem. In a particular run, `Popper` generates 471 single-clause programs which have `f(A,B,C):-tail(A,C)` as a sub-program. On the same examples, `Hempel` identifies this as a failing sub-program of the first hypothesis it generates and hence immediately prunes all these specialisations. In total, `Popper` considers 851 programs with `f(A,B,C):-tail(A,C)` as a sub-program, whilst `Hempel` considers just 48.

Failure explanation need not always be effective at pruning. Consider an arbitrary run of the *evens* problem: `Hempel` takes 354 programs before it identifies a sub-program that is not a program it has seen before. In total `Hempel` prunes based on just 19 proper sub-programs. This can be ascribed to `evens(A)` being a monadic predicate: most of the sub-programs that `Hempel` finds are properly formed `Popper` programs that `Hempel` (and

	Learning time (sec)			Accuracy		
Problem	`Hempel`	Aleph	Metagol	`Hempel`	Aleph	Metagol
dropk	4 ± 2	7 ± 18	N/A	**99 ± 2**	50 ± 2	N/A
sumlist	3 ± 0.7	60 ± 0.0	N/A	**100 ± 0**	50 ± 0	N/A
len	3 ± 0.4	60 ± 0.1	60 ± 0.1	**100 ± 0**	50 ± 0	50 ± 0
last	2 ± 0.6	1 ± 0.1	**0.7 ± 0.7**	**100 ± 0**	50 ± 0	**100 ± 0**
droplast	7 ± 2	60 ± 0.0	N/A	**100 ± 0**	50 ± 0	N/A
odd1even2	4 ± 2	56 ± 9	25 ± 25	**100 ± 0**	57 ± 17	85 ± 22
member	17 ± 6	60 ± 0.1	0.3 ± 0.0	**100 ± 0**	50 ± 0	99 ± 0
threesame	**5 ± 4**	55 ± 11	5 ± 12	**100 ± 0**	60 ± 20	**100 ± 0**
finddup	7 ± 0.6	1 ± 0.5	2 ± 2	99 ± 1	50 ± 1	**100 ± 0**
sorted	28 ± 8	0.7 ± 0.1	60 ± 0.1	**98 ± 5**	65 ± 6	50 ± 0
addhead	5 ± 2	58 ± 12	N/A	**100 ± 0**	52 ± 10	N/A
reverse	28 ± 10	36 ± 24	N/A	**100 ± 0**	50 ± 0	N/A
evens	16 ± 0.9	60 ± 0.1	60 ± 0.1	**100 ± 0**	50 ± 0	50 ± 0

Table 5.3: Results for `Hempel`, Aleph and Metagol for Experiment 2. On the left the average time to find a solution. On the right the average accuracy of solutions. The error is standard error. We round values over one to the nearest integer. Values under one we round to the most significant digit.

`Popper`) has already seen and learned constraints from. On a particular run of *reverse*, `Hempel` identifies 135 not-before-seen sub-programs. The first sub-program (of the 5th hypothesis) prunes 112 of `Popper`'s programs, the second sub-program only 26, the third 15, and from the 5th newly identified sub-program on, which already has four literals, only about three additional programs are pruned versus `Popper`. By contrast, the 10th *dropk* sub-program, of size three, still prunes 59 programs relative to `Popper`. The effectiveness of failure-explanation-based pruning appears to be strongly dependent on whether many small sub-programs can be identified.

As seen from the ratio columns of Table 5.1, the number of generated programs correlates strongly with the learning time (0.96 correlation coefficient). Only on one problem is `Hempel` slower than `Popper`. Hence outfitting `Popper` with failure explanation can occasionally affect it negatively, but this result demonstrates that at other times the speed-up can be considerable.

Figure 5.6 shows the relative time spent in each stage of `Hempel` and `Popper`. We can infer the overhead of failure explanation by analysing SLD-trees from this figure. All problems from *odd1even2* to *evens* have `Hempel` spend more time on testing than `Popper`. On *finddup*, *reverse* and *evens*, `Hempel` incurs considerable testing overhead. While for *finddup* this effort translates into more effective pruning constraints, for *sorted* and *evens* this is not the case. Abstracting away from the implementation of failure explanation, we see that `Popper` outfitted with zero-overhead failing sub-program identification would have been strictly faster.

There is considerable variance in the number of generated programs and learning times on three problems. This is in large part due to the solver that is used, Clingo [58], yielding models, i.e. hypotheses, non-deterministically. That is, there is no fixed order in which we see hypotheses, so, by chance, `Hempel` and `Popper` can come across a solution considerably sooner in one trial than in another. As a remedy for this variance, we re-run these three problems with their hypothesis spaces restricted to programs that are strictly smaller than solutions. In this setup, `Hempel` and `Popper` always terminate precisely at the point when they have shown that none of these hypotheses can be a solution. The results, which indeed have less variance, are in Table 5.2.

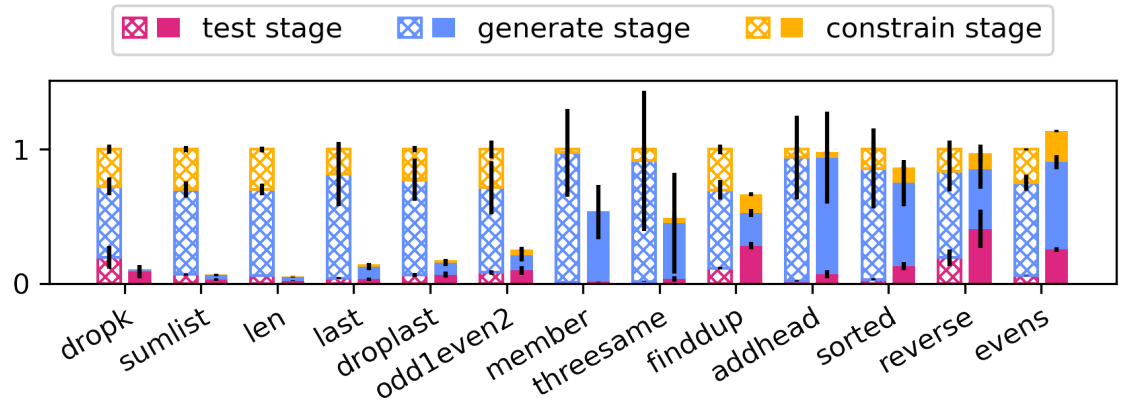


Figure 5.6: Relative time spent in three stages of `Popper`, hatched and on the left, and `Hempel`, on the right. From bottom to top: testing, generating hypotheses, and imposing constraints. Mean times are shown and scaled by the total learning time of `Popper`. Bars are standard error.

Table 5.3 shows the mean accuracy and learning times of Metagol and Aleph versus `Hempel`. Accuracy is below 67% for Aleph on all problems, which can be ascribed to Aleph struggling to learn recursive programs. Metagol cannot find solutions for problems which require arity-three predicates (unless given hand-crafted metarules), which is why 'Not Applicable' is listed for five problems. On another four problems, Metagol returns low accuracy hypotheses. Only on two problems does Metagol outperform `Hempel`. In general, `Hempel` is the more flexible system and outperforms Metagol and Aleph.

Overall, these results strongly suggest that the answer to questions **Q1** and **Q2** is yes.

5.6.3 Experiment 3: IGGP and Michalski trains

For the next experiment, we evaluate `Hempel` on problems where solutions are larger, because they either require many clauses or many literals in a clause. We consider two settings: classification in the form of Michalski train problems [90] and inductive general game playing [33]. The problems in these two settings are sufficiently hard that solutions cannot always be found in a reasonable timeframe, hence we rely on `Hempel`'s anytime capabilities to return the best scoring hypothesis it was able to find upon a timeout.

Michalski train problems concern classifying a train as either eastbound or westbound. The features available for classifying a train's heading are its cars and their features: if a car is long or short, how many wheels the car has, how many loads and which loads it is carrying, and, finally, whether the car's roof is open, closed or flat. The target predicate, `westbound/1`, acts as our classifier and BK predicates allow for inspecting features of the trains to be classified. We consider the same 4 instances considered by Cropper [29]. An example of one of the higher-quality hypotheses for the *trains4* problem is:

```
1 westbound(A):-has_car(A,C),roof_open(C),has_load(C,B),hexagon(B),three_load(B).
2 westbound(A):-has_car(A,B),has_load(B,D),diamond(D),has_load(B,C),rectangle(C).
```

Inductive General Game Playing concerns learning the rules of games from observations of these games being played. The goal is to synthesize a set of rules which are consistent with the *traces* generated by a game from the General Game Playing competition [62]. The four games we consider are: *minimal-decay*, *rock, paper, scissors (rps)*, *buttons* and *coins*. In each case we learn the predicate `next`.

Settings & systems For the trains problems, we provide two dyadic predicates, `has_car` and `has_load`, and 17 monadic predicates which encode features of cars and loads. We provide the types of arguments as well as whether they are inputs or outputs to `Hempel`, `Popper` and Aleph. We allow up to four clauses, and within each clause six variables and up to six body literals. No recursion is allowed. For Metagol we provide the same metarules as in the previous experiment. For Aleph we limit the search nodes to 30000.

For the IGGP problems we provide the monadic, dyadic and triadic predicates that encode the actions and information available to advance the game to the next state. For example, for *rps* we look for a definition of `next_score/3` given predicates `true_score/3`, `succ/2`, `does/3`, `wins/2`, `beats/2`, `different/2`.

Method We use the same instances of the problems considered by Cropper [29]. The four *trains* problems represent progressively harder instances, with *trains1* having a one clause six-literal solution and *trains4* needing 26 literals over four clauses for an optimal solution. Each trains problem has a 1000 examples available, though the distribution between positive and negative varies between tasks. We follow Cropper in that "we randomly sample the examples and split them into 80/20 train/test partitions." The four games are selected as representative instances of the larger IGGP dataset.

We measure learning time and predictive accuracy. We repeat each experiment 10 times and record the mean and standard error. We enforce a 300 second timeout.

	Number of programs			**Learning time** (sec)			**Accuracy**	
Problem	`Popper`	`Hempel`	**ratio**	`Popper`	`Hempel`	**ratio**	`Popper`	`Hempel`
rps	10648 ± 38	250 ± 13	**0.02**	96 ± 2	25 ± 1	**0.26**	100 ± 0	100 ± 0
minimal-decay	23171 ± 1538	1904 ± 76	**0.08**	300 ± 0.0	41 ± 2	**0.14**	94 ± 0	**100 ± 0**
buttons	8022 ± 2265	1073 ± 144	0.13	300 ± 0.2	300 ± 0.0	1.00	90 ± 0	90 ± 0
coins	9458 ± 934	535 ± 151	0.06	300 ± 0.0	300 ± 0.0	1.00	88 ± 3	85 ± 1
trains1	28 ± 0.0	20 ± 0.3	**0.72**	**1.0 ± 0.0**	3 ± 0.0	2.99	100 ± 0	100 ± 0
trains2	9410 ± 6144	306 ± 188	**0.03**	210 ± 137	15 ± 9	**0.07**	91 ± 5	**98 ± 2**
trains4	11223 ± 377	1176 ± 25	0.10	300 ± 0.0	300 ± 0.0	1.00	78 ± 2	**89 ± 1**
trains3	11278 ± 594	1315 ± 23	0.12	300 ± 0.0	300 ± 0.0	1.00	91 ± 2	**96 ± 1**

Table 5.4: Results for `Hempel` and `Popper` for Experiment 3. Left, the average number of programs generated by each system. Middle, the (corresponding) average time to find a solution. Right, the average accuracy of solutions. The error is standard error. We round values over one to the nearest integer. Values under one we round to the most significant digit.

Results Table 5.4 includes the results for `Hempel` and `Popper`. For the IGGP problems, we have that `Hempel` times out on *coins* and *buttons*, while `Popper` additionally times out on *minimal-decay*. On *rps* and *minimal-decay*, `Hempel` is able to find a solution with 100% accuracy. Note how `Hempel` only required around 250 programs for finding a solution for

	Learning time (sec)			Accuracy		
Problem	`Hempel`	Aleph	Metagol	`Hempel`	Aleph	Metagol
rps	25 ± 1	**4 ± 0.1**	N/A	**100 ± 0**	**100 ± 0**	N/A
minimal-decay	41 ± 2	4 ± 0.1	300 ± 0	**100 ± 0**	94 ± 0	88 ± 0
buttons	300 ± 0	137 ± 4	300 ± 0	**90 ± 0**	87 ± 0	80 ± 0
coins	300 ± 0	300 ± 0.0	N/A	**85 ± 1**	82 ± 0	N/A
trains1	3 ± 0.0	**2 ± 0.3**	162 ± 38	100 ± 0	100 ± 0	100 ± 0
trains2	15 ± 9	1 ± 0.1	218 ± 126	98 ± 2	**100 ± 0**	85 ± 6
trains4	300 ± 0	215 ± 4	300 ± 0	89 ± 1	**100 ± 0**	67 ± 0
trains3	300 ± 0	18 ± 0.9	300 ± 0	96 ± 1	**100 ± 0**	20 ± 0

Table 5.5: Results for `Hempel`, Aleph and Metagol for Experiment 3. On the left the average time to find a solution. On the right the average accuracy of solutions. The error is standard error. We round values over one to the nearest integer. Values under one we round to the most significant digit.

rps while `Popper` required over 10,000 programs. For *minimal-decay* `Hempel` needs to consider almost 2000 programs before coming across a solution while `Popper` cannot find one within the time limit.

In Table 5.5 we see the performance of Metagol and Aleph versus `Hempel` on the IGGP problems. As Metagol's metarules do not support arity-three predicates, we have that it is unable to find programs for *rps* and *coins*. On the other two problems, Metagol timeouts and hence achieves the default accuracy for these problems. On *coins*, both `Hempel` and Aleph achieve the default accuracy. On *rps*, Aleph does better than `Hempel` by virtue of its learning time, though `Hempel` still beats Metagol. On the three other games, `Hempel` does better than both Aleph and Metagol.

Referring back to Table 5.4, we see that `Hempel` outperforms `Popper` on the three more difficult trains problems. On *trains1* we see clearly the overhead of failure explanation. Even though `Hempel` requires less programs than `Popper`, testing 800 examples incurs 800 times the linear overhead of failure explanation (with regards to SLD-tree size) plus the cost of retesting failing sub-programs, of which there are more when we are dealing with bigger hypotheses. On the other three problems, the cost of failure explanation is outweighed by the pruning it achieves, with `Hempel` finding more accurate solutions. Not shown in Table 5.4, for the timeouts, `Hempel` spends a greater proportional of time in the test stage than `Popper`, e.g. about two-thirds of the time on *trains4* versus just one-third of the time, respectively. This is likely attributable to the cost of retesting many sub-programs on the high number of examples.

From Table 5.5 we can see that Aleph's bottom clause construction-based learning procedure is quite effective, outperforming `Hempel` on all four trains problems. In turn, `Hempel` outperforms Metagol on all trains problems.

Also for this experiment, the results indicate that the answer to questions **Q1** and **Q2** is yes, though with the note that larger hypotheses and more examples do appear to impact the effectiveness.

5.6.4 Experiment 4: string transformations

We now explore whether failure explanation can improve learning performance on real-world string transformation tasks. We hence restrict ourselves to comparing `Hempel` versus `Popper`. We use a standard dataset [101, 30] formed of 312 tasks, each with 10

input-output pair examples. For example, task 81 has the following two input-output pairs:

Input	Output
"Alex","M",41,74,170	M
"Carly","F",32,70,155	F

Settings. As background knowledge, we give each system the monadic predicates *is_uppercase*, *is_empty*, *is_space*, *is_letter*, *is_number* and dyadic predicates *mk_uppercase*, *mk_lowercase*, *skip1*, *copyskip1*, *copy1*. For each monadic predicate we also provide a predicate that is its negation. We allow up to 3 clauses, with each clauses having a maximum of 4 body literals and up to 5 variables. We extend the test stage with a check whether the generated program is functional or not and prune for any non-functional program.

Method. The dataset has 10 positive examples for each problem. We perform cross validation by selecting 10 distinct subsets of 5 examples for each problem, using the other 5 to test. We measure learning times and number of programs generated. We enforce a timeout of 60 seconds per task. We repeat each experiment 10 times, once for each distinct subset, and record means and standard errors.

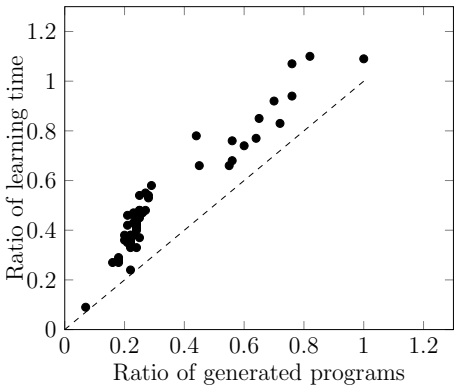


Figure 5.7: String transformation results. The ratio of number of programs that `Hempel` needs versus `Popper` is plotted against the ratio of learning time needed on that problem.

Results. In 132 problems both `Hempel` and `Popper` return programs which have non-zero accuracy on the test set. On 64 tasks `Hempel` scores better than `Popper` versus `Popper` scoring better on 20 tasks. For 54 problems at least one of `Popper` and `Hempel` finds solutions with over 90% mean accuracy. `Hempel` finds solutions[8] with 100% accuracy on 37 tasks, 3 more than `Popper`.

Figure 5.7 plots ratios of generated programs and learning times. Each of the 54 points represents a single problem where either `Hempel` or `Popper` scored over 90% mean accuracy. The x-axis is the ratio of number of programs that `Hempel` generates versus the number of programs that `Popper` generates. The y-value is the ratio of learning time of

[8]Note that these problems are very difficult with many of them not having solutions given only our primitive BK and with the learned program restricted to defining a single predicate. Therefore, absolute performance should be ignored. The important result is the relative performance of the two systems.

`Hempel` versus `Popper`. These ratios are acquired by dividing means, the mean of `Hempel` over that of `Popper`.

Looking at x-axis values, of the 54 problems plotted all require fewer programs when run with `Hempel`. Looking at the y-axis, the learning times of 51 problems are faster for `Hempel`.

Overall, these results show that, compared to `Popper`, `Hempel` typically needs fewer programs and less time to learn programs. This suggests that the answer to questions **Q1** and **Q2** is yes.

Chapter 6

Learning From Failures Modulo Theories

This chapter extends the Learning From Failures approach to Inductive Logic Programming with the notion of over-approximating properties. Given (a conjunction of) atom(s), an over-approximating property is one which holds whenever the atom(s) hold[1]. This connection between atoms and their properties we express with *over-approximating rules*. When provided, over-approximating rules allow us to connect a guess for a partial program to formulas expressing a combined property of that program. We prove that whenever the formulas expressing over-approximating properties of a sub-program are unsatisfiable together, then (each extension of) that program cannot be an optimal solution. By making sure the properties are expressed by formulas from suitable logic fragments, the consistency of these formulas can be decided by a (background) theory solver. We show that reasoning about over-approximating properties leads to learning conflicts without the need for the test stage, and hence without needing to evaluate examples. By revisiting the list experiment, we show that this can lead to significant reductions in number of programs evaluated by the test stage and learning times.

6.1 Introduction

When we write programs, i.e. the task program synthesis is supposed to address, we often take into account properties of the functions we are composing [46]. For instance, when trying to come up with an implementation for a droplast/2 predicate, a programmer might suppose that the tail/2 predicate will be of use. For the tail/2 predicate we know the following property:

"the first argument is always one longer than the second argument."

As for specifying what droplast/2 should do, the programmer can partially capture this with the same property: "the first argument is always one longer than the second argument." Note that by knowing just these properties, our programmer might think that droplast$(A, B) \leftarrow$ tail(A, B) is a solution. This shows why these properties – often expressed in simpler languages than the programs themselves [142] – tend to only partially capture functional correctness and might not be enough to pin down the solutions we

[1] In general, the converse does not hold.

are after. We can see examples as a complementary specification that makes it easy to eliminate non-solutions: even a single example, e.g. droplast($[1,2,3],[1,2]$), suffices to rule out the just considered hypothesis.

Nonetheless, these properties by themselves are very effective at ruling out other programs that we know cannot work. For instance, we know – without evaluating examples – that $\text{droplast}(A,B) \leftarrow \text{tail}(A,C), \text{tail}(C,B)$ occurring as a part of a program is going to be useless, as the property for droplast/2 says the length of A is the length of B plus one while the two tail/2 literals in the body allow us to infer that the difference in length between A and B has to be two. Even without a property for the target predicate, we can infer that there are sets of literals that make a clause useless. For example, if $\text{tail}(A,B)$, $\text{tail}(A,C)$, and $\text{tail}(C,B)$ all occur within one clause, we know – due to the same list length-based reasoning – that this clause (and any extension of it) is useless.

In this chapter we develop the notion of *over-approximating properties* [51] for logic programs. We use rules to connect literals that can occur in programs to formulas expressing their properties, e.g. $\text{tail}(A,B) \rightarrow \text{len}(A) = \text{len}(B)+1$, and formalize when these properties are *over-approximating* with regard to the literals. We use these rules to infer the combined properties of programs. We prove that when the formulas expressing these properties become unsatisfiable together, there is a redundant clause in the program.

This approach to over-approximating properties naturally fits in the Learning From Failures as Satisfiability Modulo Theories framework. In Section 4.2.4, we had that a hypothesis H is a solution when there is an model I of $E_{\mathcal{H}}$ such that $\langle\!\langle E_{\mathcal{H}} \rangle\!\rangle_I^{Horn} = H$ and $\mathcal{BK} \wedge H \wedge \mathcal{E}$ is satisfiable with respect to the theory $\mathcal{T}_{LM(\mathcal{BK} \cup H)}$. With over-approximating rules we have that an interpretation I not only maps to a set of Horn formulas but also to a set of formulas expressing properties, such as relations on lengths. In this extended framework – LFF Modulo Theories – we have that interpretation I can (be extended to) represent an *optimal* solution only when this other set of formulas expressing the over-approximating properties is satisfiable.

For reasoning about over-approximating properties to be effective, the formulas expressing the properties should come from logic fragments which have fast decision procedures [44, 70, 10]. In terms of the LFFMT framework, we abstract from particular fragments. As such, we say that the formulas come from a *Prop* logic fragment and are satisfiable w.r.t. a theory $\mathcal{T}_{Prop}$. Given a solver for deciding satisfiability w.r.t. $\mathcal{T}_{Prop}$, we can extend the three-stage loop approach to LFF.

Figure 6.1 shows how the SAT-solver of the generate stage can communicate formulas enabled by over-approximating rules to a $\mathcal{T}_{Prop}$-solver based on the current (partial) guess for a hypothesis. Whenever this solver detects that the set of formulas it knows about is inconsistent, it generates a conflict – a simple constraint that says that these formulas can never occur together. This conflict prunes the (sub-)program that triggered this set of inconsistent formulas as well as any other hypothesis that triggers the same set of conflicting formulas.

For example, if we have the over-approximating *rules* $\text{tail}(A,B) \rightarrow \text{len}(A) = \text{len}(B)+1$ and $\text{reverse}(A,B) \rightarrow \text{len}(A) \geq 0 \wedge \text{len}(B) \geq 0$, we can eliminate a guess for $\text{droplast}(A,B) \leftarrow \text{tail}(A,C), \text{tail}(C,B)$ being part of the definition of $\text{droplast}(A,B)$, due to to the conflict[2] on the lengths of A and B (droplast's property says that the lengths of A and B differ exactly one while the two tails together imply this difference is two). In con-

[2]As all positive examples of $\text{droplast}(A,B)$ must necessary satisfy its *over-approximating* property, showing such an inconsistency means no positive example can be entailed.

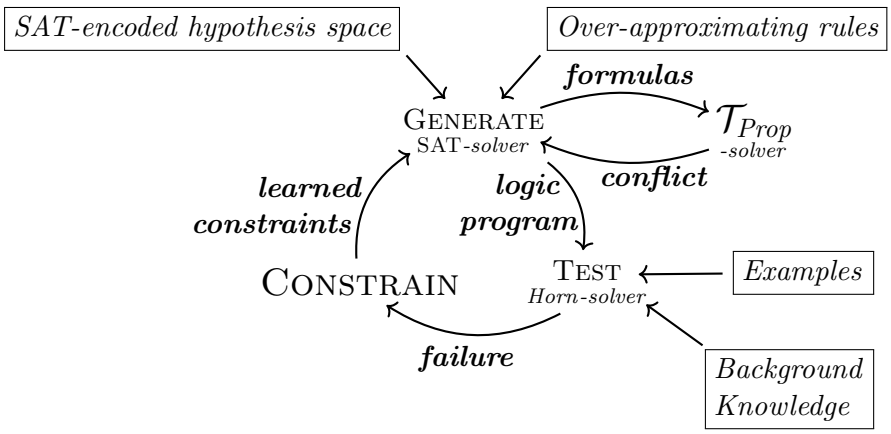


Figure 6.1: LFF's three-staged loop expanded by another loop involving a theory solver that decides satisfiability of formulas expressing over-approximating properties.

trast to functional programming synthesis which seek to establish the validity of a specified property [130, 52, 53, 126, 55, 68], we do not reject a guess like droplast$(A, B) \leftarrow$ reverse(A, C), tail(C, D), reverse(C, B) even though reverse's given property is too weak to prove that droplast's property holds for this implementation.

The contributions of this chapter are:

- We define over-approximating properties for logic programs and introduce rules that connect literals to over-approximating formulas. We use these rules to infer the combined properties of program fragments. We prove that if the inferred combined over-approximating property for a program is unsatisfiable, then the program contains a redundant clause.
- We add over-approximating rules as an input to Learning From Failures. We prove that reasoning with over-approximating properties allows for sound pruning of hypotheses which cannot be optimal solutions, without needing to evaluate examples.
- Given a solver for formulas that express properties, we extend Conflict-Driven ILP with the ability to prune guessed hypotheses before they are evaluated on examples, relying on the SMT-solving capabilities of the solver that selects hypotheses.
- We apply this principle to our Learning From Failures ILP system, `Popper`, making use of its SAT-solver's ability to communicate with a Difference Logic-solver, yielding the `Popper`$[\mathcal{DL}]$ system capable of reasoning about $\mathcal{DL}$-encoded properties.
- We revisit the list transformation experiment to show that reasoning about the satisfiability of over-approximating properties can achieve significant reductions in number of tested hypotheses and learning times.

6.2 Over-approximating properties

We now develop the notion of over-approximating properties for logic programs. Starting from an observation on how body literals cause a clause to be redundant, we lift this reasoning to properties of the body literals. This leads to introducing rules which connect predicates to their properties. We finish off the section by generalizing these rules to state properties of combinations of literals in clauses.

6.2.1 Redundancy due to unsatisfiable bodies

Our point of departure is the observation that some literals of a clause are "obviously" in conflict with one another. For example, if we have both odd(A) and even(A) occur in (the body of) a clause, clearly this clause cannot be used to entail anything.

First we show that a clause – whose variables $\mathbf{v}$ are all universally quantified – within a program having an unsatisfiable body makes that clause redundant (Definition 4.2.13).

Proposition 6.2.1. A clause $C = \forall \mathbf{v}.h \leftarrow b_1, \ldots, b_n$ in program P is redundant when there is no substitution θ such that $P \models (b_1, \ldots, b_n)\theta$.

Proof. Clearly clause C cannot contribute any consequences to P, hence $P \backslash \{ C \} \models P$. □

Note that this also holds in case C is a non-definite Horn clause, i.e. when $h = \bot$.

Example 6.2.2. Suppose we have the following program:

$$P = \left\{ \begin{array}{lcl} \mathrm{p}(A, B) & \leftarrow & \mathrm{add}(A, A, B), \\ \mathrm{p}(A, B) & \leftarrow & \mathrm{mult}(A, A, B), \mathrm{even}(A), \mathrm{odd}(A) \end{array} \right\}$$

Clearly there is no substitution for A and B in the second clause which makes both even(A) and odd(A) true at the same time. Therefore the predicate p, as defined by P, only entails those atoms where B is twice A. Which is exactly the same as the program consisting of just the first clause.

We now generalize the above proposition from programs to interpretations. The reason is two-fold: first, as a programmer we often only know a class of plausible interpretations for our target predicate and would like to know when a clause is useless w.r.t. these interpretations; second, our properties will be expressed in formulas which need to be interpreted to determine their satisfiability.

Definition 6.2.3. A clause $C = \forall \mathbf{v}.h \leftarrow b_1, \ldots, b_n$ is redundant w.r.t. an interpretation I when there is no θ s.t. $[\![(b_1 \wedge \ldots \wedge b_n)\theta]\!]_I = \top$.

To see that this definition encompasses the redundancy characterized by Proposition 6.2.1, let $C = \forall \mathbf{v}.h \leftarrow b_1, \ldots, b_n$ be a clause of program P. If we take $I = LM(P)$ then when there is no θ s.t. $P \models (b_1 \wedge \ldots \wedge b_n)\theta$, C is redundant w.r.t. I but it is also redundant w.r.t. P. We say that rule C has an *unsatisfiable* body w.r.t. I or P, respectively.

Example 6.2.4. Suppose tail(A, B) and droplast(B, A) are in the body of a clause C and interpretation I interprets tail/2 in the usual way. If I interprets droplast/2 such that its first argument is always one longer than its second argument, then C's body is unsatisfiable, meaning C is redundant w.r.t. I and will be redundant in any program which has I as its least model.

6.2.2 Over-approximating properties for predicates

We want to establish a method for reasoning about when the body of a clause is unsatisfiable as that allows us to establish its redundancy. We will do this by reasoning about *over-approximating properties*.

We now define how to connect a predicate to a property of it, introducing a *rule* which is meant to be read as an implication:

Definition 6.2.5. A rule $R = R_{atom} \rightarrow R_{prop}$ is *over-approximating* w.r.t. interpretation I, if for all θ we have that $[\![R_{atom}\theta]\!]_I = \top$ implies $[\![R_{prop}\theta]\!]_I = \top$.

Note that we need interpretation I to both interpret the symbols in formula R_{atom} as well as the symbols in R_{prop}. We then have that if we read the rule like a normal implication, it holds that $[\![(R_{atom} \rightarrow R_{prop})\theta]\!]_I = \top$ for each θ.

We have that $R_{atom} \rightarrow \top$ is always a valid over-approximating rule, no matter the interpretation. Hence we can always use a rule $A \rightarrow \top$ as an over-approximating rule of (the predicate of) A in case a more specific rule is not available.

From this point on we will distinguish a fragment of a program from its properties by denoting them in different colours. We will use blue to highlight formulas representing parts of programs, i.e. from the Horn fragment, and other colours – purple for generic property formulas and red for Difference Logic formulas – for formulas expressing properties. These colours serve just as an aid to identifying which fragments the sub-formulas come from: the colours have no formal meaning and can be elided completely.

Example 6.2.6. If I interprets tail/2 as usual (e.g. because I extends the least model $LM(P)$ of a program P which defines tail/2 as expected) and I interprets the function symbol len/1 as the function returning lengths of lists (and binary addition as usual), then $R = \text{tail}(A, B) \rightarrow \text{len}(A) = \text{len}(B) + 1$ is an over-approximating rule w.r.t. I as no matter which lists l_1 and l_2 we substitute for A and B, whenever $\text{tail}(l_1, l_2)$ holds then $\text{len}(l_1) = \text{len}(l_2) + 1$ is going to hold.

Unsatisfiable combined properties

Over-approximating rules are useful as they allow for reasoning about the (un-)satisfiability of (bodies of) clauses based on their properties. We first prove that the over-approximating properties of predicates lift to over-approximations of the satisfying assignments for clauses:

Proposition 6.2.7. Given clause $C = \forall \mathbf{v}.h \leftarrow b_1, \ldots, b_m$, interpretation I s.t. $[\![C]\!]_I = \top$ and rules $b_1 \rightarrow p_1$, …, $b_m \rightarrow p_m$ and $h \rightarrow p_h$ which are over-approximating w.r.t. I, then whenever there is θ s.t. $[\![(b_1 \wedge \ldots \wedge b_m)\theta]\!]_I = \top$ then $[\![(p_1 \wedge \ldots \wedge p_m \wedge p_h)\theta]\!]_I = \top$.

Proof. As we have that $b_1 \rightarrow p_1$, …, $b_m \rightarrow p_m$ are all over-approximating, whenever we have $[\![(b_1 \wedge \ldots \wedge b_m)\theta]\!]_I = \top$, then $[\![(p_1 \wedge \ldots \wedge p_m)\theta]\!]_I = \top$. We also have that C holds according to I, hence when $[\![(b_1 \wedge \ldots \wedge b_m)\theta]\!]_I = \top$, then $[\![h\theta]\!]_I = \top$ which, per over-approximating rule $h \rightarrow p_h$, we can use to derive that $p_h\theta$ must hold as well. □

Example 6.2.8. If we have $[\![\forall A, B.\text{trimfirstlast}(A, B) \leftarrow \text{tail}(A, C), \text{droplast}(C, B)]\!]_I = \top$ and rules $\text{trimfirstlast}(A, B) \rightarrow \text{len}(A) = \text{len}(B) + 2$, $\text{tail}(A, C) \rightarrow \text{len}(A) = \text{len}(C) + 1$, and $\text{droplast}(C, B) \rightarrow \text{len}(C) = \text{len}(B) + 1$ are over-approximating w.r.t. I, then no matter which lists l_1 and l_2 we replace A and B with, whenever $\text{tail}(A, C), \text{droplast}(C, B)$ holds then the following combined property holds:

$$\text{len}(A) = \text{len}(B) + 2 \wedge \text{len}(A) = \text{len}(C) + 1 \wedge \text{len}(C) = \text{len}(B) + 1$$

A direct corollary of Proposition 6.2.7 is that whenever the combined over-approximation is unsatisfiable, then the clause itself is either inconsistent with the interpretation or its body is unsatisfiable.

Corollary 6.2.9. Given clause $C = \forall \mathbf{v}.h \leftarrow b_1, \ldots, b_m$ and over-approximating rules $b_1 \rightarrow p_1$, …, $b_m \rightarrow p_m$ and $h \rightarrow p_h$ w.r.t. I, if there is no θ s.t. $[\![(p_1 \wedge \ldots p_m \wedge p_h)\theta]\!]_I = \top$, then either $[\![C]\!]_I = \bot$ or there is no ϑ such that $[\![(b_1 \wedge \ldots \wedge b_m)\vartheta]\!]_I = \top$.

Proof. As we are denying the consequent of Proposition 6.2.7, and assume the rules to be actually over-approximating w.r.t. I, the issue must be either with the assumption that clause C holds according to I or that the body of C is never satisfied. □

This result is useful as it allows us to say that, w.r.t. an interpretation I, whenever the combined over-approximating properties of a clause C are unsatisfiable, then clause C cannot meaningfully contribute to a program P whose interpretation is I. This is due to clause C either ensuring that I is not a model of P or else making C redundant in P because of C's unsatisfiable body.

Example 6.2.10. Suppose we have a clause $C = \text{droplast}(A, B) \leftarrow \text{tail}(A, B), \text{tail}(B, A)$ and rules $\text{tail}(A, B) \rightarrow \text{len}(A) = \text{len}(B) + 1$ and $\text{tail}(B, A) \rightarrow \text{len}(B) = \text{len}(A) + 1$ which are over-approximating w.r.t. I. Then $\text{len}(A) = \text{len}(B) + 1 \wedge \text{len}(B) = \text{len}(A) + 1$ is the combined property for C. As this formula is unsatisfiable, we can conclude that clause C is redundant for any program whose least model is I.

Example 6.2.11. Consider clause $C = \text{droplast}(A, B) \leftarrow \text{tail}(A, C), \text{tail}(C, B)$ again now with rules $\text{tail}(A, C) \rightarrow \text{len}(A) = \text{len}(C) + 1$ and $\text{tail}(C, B) \rightarrow \text{len}(C) = \text{len}(B) + 1$ which are over-approximating w.r.t. I. Suppose we also know the property droplast/2 should have, i.e. we have that $\text{droplast}(A, B) \rightarrow \text{len}(A) = \text{len}(B) + 1$ is over-approximating w.r.t. the (intended) interpretation I. Then

$$\text{len}(A) = \text{len}(C) + 1 \wedge \text{len}(C) = \text{len}(A) + 1 \wedge \text{len}(A) = \text{len}(B) + 1$$

is C's combined property. As this formula is unsatisfiable, we can conclude, by Corollary 6.2.9, that $[\![C]\!]_I = \bot$ as the body of C is satisfiable.

6.2.3 Over-approximating properties for clauses

Besides just stating properties for predicates, it can be useful to state a property for a part of a clause. We now expand our definition of over-approximating rules, Definition 6.2.5, to Horn clauses (both definite and not).

As now we are only interested in satisfying assignments for clauses, we modify the main connective ($\rightarrow$) of these rules. We use the double-arrow connective, e.g. in $C \Rightarrow p$, to mean that any assignment that *non-trivially* satisfies C (i.e. all literals of C are satisfied) must satisfy p. The following rules are intended to be read as pattern matches: whenever the Horn clause on the lefthandside occurs as part of a clause in a program then the property must hold.

Definition 6.2.12. A rule $R = (h \leftarrow b_1, \ldots, b_m) \Rightarrow R_{prop}$ $(m \geq 0)$ is *over-approximating* w.r.t. interpretation I, if $[\![(h \wedge b_1 \wedge \ldots \wedge b_m)\theta]\!]_I = \top$ implies $[\![R_{prop}\theta]\!]_I = \top$ for each θ.

Note how the case $m = 0$ gives us back Definition 6.2.5, now only applying if the atom occurs as a head literal. If we leave out the head literal of the clause we have:

Definition 6.2.13. A rule $R = (\leftarrow b_1, \ldots, b_m) \Rightarrow R_{prop}$ is *over-approximating* w.r.t. interpretation I, if for all θ we have that $[\![(b_1 \wedge \ldots \wedge b_m)\theta]\!]_I = \top$ implies $[\![R_{prop}\theta]\!]_I = \top$.

If we set $m = 1$ in this definition, we get a form of Definition 6.2.5 where the property of the predicate only needs to apply if the atom occurs as a body literal.

Example 6.2.14. Depending on the expressivity of the logic fragment for properties, it might be hard to state properties without making use of other literals. For example, in the presence of other predicates we can talk about the relationship of elements of lists, even in a very restricted fragment like Difference Logic:

$$R = (\leftarrow \text{sorted}(A), \text{head}(A, B), \text{tail}(A, C), \text{head}(C, D)) \Rightarrow B \leq D$$

is an over-approximating rule for any interpretation I capturing the usual definitions of sorted/2, head/2 and tail/2. (As the first element of a sorted list A better be at most as big as its second element.)

When we know a property of a predicate we are trying to learn a definition for, it is useful to say a predicate in the head of a clause has a property.

Example 6.2.15. We can use Definition 6.2.12 (in the form that mirrors Definition 6.2.5) to say that when our target predicate occurs in the head of a clause then a property should hold, e.g. $\text{droplast}(A, B) \Rightarrow \text{len}(A) = \text{len}(B) + 1$. We can also say that an occurrence of that predicate in a body, e.g. when it is called recursively, also should have that property, as otherwise the learned definition did not capture any of the interpretations we had in mind: $\leftarrow \text{droplast}(A, B) \Rightarrow \text{len}(A) = \text{len}(B) + 1$, now using Definition 6.2.13.

To know which combined properties hold of a clause, given a set of over-approximating rules, we define when over-approximating rules *apply* to the clause:

Definition 6.2.16. A rule $R = R_{horn} \Rightarrow R_{prop}$ with either $R_{horn} = \leftarrow b_1, \ldots, b_m$ or $R_{horn} = h \leftarrow b_1, \ldots, b_m$ *applies* to a horn clause C when $R_{horn} \subseteq C$.

We will stretch the term *applies* and will say that an (over-approximating) rule $R = R_{horn} \Rightarrow R_{prop}$ also applies when $R_{horn} \preceq C$, i.e. there exists a substitution θ such that $R_{horn}\theta \subseteq C$. This is natural as R and $R\theta = R_{horn}\theta \Rightarrow R_{prop}\theta$ express the same property. When a renaming substitution θ is necessary for a rule R to apply to C, we will implicitly apply this substitution[3] to obtain a rule that formally applies to C, i.e. $R\theta$.

Example 6.2.17. Let's reconsider clause $C = \text{droplast}(A, B) \leftarrow \text{tail}(A, B), \text{tail}(B, A)$ from Example 6.2.10 as well as rules $R^1 = \text{tail}(A, B) \Rightarrow \text{len}(A) = \text{len}(B) + 1$ and $R^2 = \text{tail}(B, A) \Rightarrow \text{len}(B) = \text{len}(A) + 1$. Both R^1 and R^2 apply directly to C. However, as $R^1 = R^2\theta$ for $\theta = \{ A \mapsto B, B \mapsto A \}$, it suffices to just write down R^1 and say that R^1 applies twice to C.

We now introduce notation for applying a rule to a Horn clause:

Definition 6.2.18. Given a rule $R = R_{horn} \Rightarrow R_{prop}$ and a Horn clause C, we have $R(C) = R_{prop}$ if R applies to C else $R(C) = \top$.

We will use the same notation for applying a set of rules $\mathcal{R}$ to a clause C: $\mathcal{R}(C) = \{ R(C) \mid R \in \mathcal{R} \}$. In what follows, when working with a set of rules $\mathcal{R}$, we will assume that such sets are closed under variable renamed versions of the contained rules.

[3] In our implementation, we do explicitly consider all variants of over-approximating clauses by generating versions of them with all relevant variable names, see Section 6.4.

Unsatisfiable combined properties – redux

Now that we have rules that apply to clauses based on subsets of clauses rather than only atoms, we derive the same result as Corollary 6.2.9 in this wider context.

Theorem 6.2.19. Given rules $R^1 = R^1_{horn} \Rightarrow R^1_{prop}, \ldots, R^n = R^n_{horn} \Rightarrow R^n_{prop}$ which are over-approximating w.r.t. I and apply to Horn clause C, whenever $R^1_{prop} \wedge \ldots \wedge R^n_{prop}$ is unsatisfiable w.r.t. I, either I is not a model of C or C is redundant w.r.t. I.

Proof. Assume there is no θ s.t. $[\![(R^1_{prop} \wedge \ldots \wedge R^n_{prop})\theta]\!]_I = \top$. Let $C' = R^1_{horn} \cup \ldots \cup R^n_{horn}$, i.e. the subset of C necessary for the rules to apply, and perform a case analysis on whether C' is definite or not:

1. Suppose $C' = \leftarrow b_1, \ldots, b_m$, then, as $R^1, \ldots, R^n$ are over-approximating, per Definition 6.2.13 there is no ϑ s.t. $[\![(b_1 \wedge \ldots \wedge b_m)\vartheta]\!]_I = \top$. Hence C's body is unsatisfiable as well, meaning C is redundant w.r.t. I.
2. Suppose $C' = h \leftarrow b_1, \ldots, b_m$, then as $R^1, \ldots, R^n$ are over-approximating, per Definition 6.2.12 there is no ϑ s.t. $[\![(h \wedge b_1 \wedge \ldots \wedge b_m)\vartheta]\!]_I = \top$. Hence for each ϑ either $[\![(b_1 \wedge \ldots \wedge b_m)\vartheta]\!]_I = \bot$ or $[\![(h \leftarrow b_1 \wedge \ldots \wedge b_m)\vartheta]\!]_I = \bot$. If there is even one ϑ such that $[\![(h \leftarrow b_1 \wedge \ldots \wedge b_m)\vartheta]\!]_I = \bot$, then I is not a model of C. If for all ϑ we have $[\![(b_1 \wedge \ldots \wedge b_m)\vartheta]\!]_I = \bot$, then C is redundant w.r.t. I.

□

Whenever a clause C is either not modelled by an interpretation I or is redundant w.r.t I based on its (combined) properties, we say C is *rejected* (w.r.t. I) based on its over-approximating properties. In the next section we use over-approximating properties to reject guessed clauses and thereby prune the hypothesis space of programs which have (clauses with) unsatisfiable properties.

6.3 ILP with over-approximating properties

Having developed a notion of over-approximating rules, and how they can show the redundancy of clauses (w.r.t. interpretations), we now make use of them in the setting of Inductive Logic Programming. First, we establish that examples and over-approximating rules are complementary. Next, we generalize the Learning From Failures framework (Section 4.2) to accommodate reasoning with over-approximating properties, which get enabled and disabled based on over-approximating rules. We will focus on over-approximating properties expressed in *background theories* [56]. Reasoning about (the unsatisfiablity of) formulas from these theories can then be dispatched to dedicated solvers. These solvers are responsible for detecting unsatisfiablity of the combined properties and generating a conflict clause (a simple form of constraint), which achieves pruning, whenever this happens.

6.3.1 Over-approximating rules versus examples

When trying to find a solution H^* in a hypothesis space $\mathcal{H}$ such that $\mathcal{BK} \cup H^* \models \mathcal{E}$, we can learn from testing a hypothesis $H \in \mathcal{H}$ against examples $\mathcal{E}$ that a class of programs related to H do not work, see Section 4.3. This is the fundamental mechanism of Learning From Failures. Reasoning with over-approximating properties similarly allows us to consider a

part of a program/clause and learn that a clause is redundant in (or inconsistent with) a whole class of programs.

This means that we now have two ways of identifying that a hypothesis $H \in \mathcal{H}$ is not an (optimal) solution: either we test H on the examples $\mathcal{E}$ or we reason by way of the over-approximating properties that H contains a clause C that either causes inconsistency or has an unsatisfiable body. Furthermore, any other hypothesis containing clause C can be ruled out as an optimal solution as it will have the same issue.

We now compare examples versus over-approximating properties in terms of their capacity for specifying the intended interpretation of which a solution H^* is supposed to be a definition.

Positive examples as finite under-approximations

Positive examples $\mathcal{E}^+$ specify the class of interpretations we are trying to learn a definition for "from below". Positive examples $\mathcal{E}^+$ enforce the constraint that the (Herbrand) interpretation I^* of a solution H^*, i.e. its least model, must include at least each $e^+ \in \mathcal{E}^+$. A rule $R = \mathrm{p}(A_1, \ldots, A_n) \Rightarrow R_{prop}$ is only over-approximating for an interpretation I if for every θ we have that $\mathrm{p}(A_1, \ldots, A_n)\theta$ implies $R_{prop}\theta$. As such, (assuming that the interpretation of the predicate and function symbols in property formulas are already fixed) a rule R sets an upper bound on the interpretation of predicate p/n. An over-approximating rule hence constrains possible interpretations for the predicate we are trying to learn "from above".

Note that, whereas in the Learning From Entailment setting to ILP, we assume $\mathcal{E}^+$ to be finite [136], over-approximating properties state properties in terms of classes of assignments potentially infinite in size (e.g. lengths of all possible lists or for all integers). Hence positive examples and over-approximating properties help us pin down an intended interpretation for a solution in complementary ways.

Example 6.3.1. Suppose $\mathcal{E}^+ = \{\mathrm{droplast}([1], []), \mathrm{droplast}([1,2,3], [1,2])\}$. Then these examples specify that the (least model) interpretation of our program should at least include these two facts. Of course, this can be trivially accomplished by just taking these two atoms and interpret them as clauses (and we can add any other definite clauses doing arbitrary things as well). An over-approximating rule like $R = \mathrm{droplast}(A, B) \Rightarrow \mathrm{len}(A) = \mathrm{len}(B) + 1$ allows us to say that a large number of clauses do not make sense for the program we intend to learn, such as a clause like $\mathrm{droplast}(A, B) \leftarrow \mathrm{tail}(A, C), \mathrm{tail}(C, B)$ (given we also have an over-approximating rule like $\leftarrow \mathrm{tail}(A, B) \Rightarrow \mathrm{len}(A) = \mathrm{len}(B) + 1$).

Expressed succinctly: positive examples serve as under-approximations of the intended interpretation (of a target predicate) and over-approximating properties act as over-approximating bounds on a predicate. Note that each positive example can be expressed as an assignment/substitution for the arguments of the target predicate. Therefore, the substitution that corresponds to a positive example must satisfy the over-approximating property for the target predicate. We will use this fact in our experimental evaluation, see Section 6.5.3.

Negative examples as finite over-approximations

Just like over-approximating properties, negative examples $\mathcal{E}^-$ rule out interpretations "from above". Rather than capturing vast classes of interpretations based on infinite sets

of assignments, negative examples bound the intended interpretation(s) from above by a sparse set of facts, often more specific than over-approximating properties can capture.

We hence see there can be overlap in terms of specifying the intended interpretation of our target predicate using both negative examples and over-approximating properties. In particular, any negative example $e^- \in \mathcal{E}^-$ such that $\mathrm{p}(A_1, \ldots, A_n)\theta = e^-$ is superfluous for specifying p/n's intended interpretation I when $[\![R_{prop}\theta]\!]_I = \bot$ given that the rule $\mathrm{p}(A_1, \ldots, A_n) \Rightarrow R_{prop}$ is over-approximating w.r.t. I.

Example 6.3.2. Suppose $e_1^- = \mathrm{droplast}([1,2,3],[1])$ and $e_2^- = \mathrm{droplast}([1,2,3],[2,3])$ such that $\mathcal{E}^- = \{ e_1^-, e_2^- \}$. If the rule $R = \mathrm{droplast}(A,B) \Rightarrow \mathrm{len}(A) = \mathrm{len}(B) + 1$ is over-approximating w.r.t. the intended interpretation(s) I^*, then we have that negative example e_1^- was superfluous as there is no interpretation I such that both e_1^- is a consequence and rule R is over-approximating. Hence, given that we have R, e_1^- does not help hone in on our intended interpretations. Conversely, e_2^- does say that certain interpretations compatible with R are not our intended interpretation. This is so because the over-approximating property does hold for this example, i.e. $\mathrm{len}([1,2,3]) = \mathrm{len}([2,3]) + 1$, whilst any interpretation that has $\mathrm{droplast}([1,2,3],[2,3])$ as a consequence is ruled out due to e_2^- being classified as a negative example.

6.3.2 Learning from rejections

We now establish what information we can learn from a clause being rejected based on its combined over-approximating properties. In particular, we show that any (part of a) clause which has a combined over-approximating property that is unsatisfiable cannot occur within an optimal solution.

Containing a rejected clause is grounds for rejecting a hypothesis

When clause C of a program P contains a clause that gets rejected w.r.t. an interpretation I, then the entire program can be disregarded. We formalize this as follows:

Proposition 6.3.3. For all interpretations I such that rules $R^1 = R^1_{horn} \Rightarrow R^1_{prop}, \ldots, R^n = R^n_{horn} \Rightarrow R^n_{prop}$ are over-approximating, if definite program P contains a clause C such that $C' \subseteq C$ has $R^1, \ldots, R^n$ apply and $R^1_{prop} \wedge \ldots \wedge R^n_{prop}$ is unsatisfiable w.r.t. I, then either I is not a model of P or I is a model of $P \setminus \{ C \}$ iff I is a model of P.

Proof. Let I be any interpretation such that $R^1, \ldots, R^n$ are over-approximating rules. If $R^1, \ldots, R^n$ apply to C' then they also apply to C. By Theorem 6.2.19, whenever the combined property of a clause C is unsatisfiable w.r.t. I, either I is not a model of C or C is redundant w.r.t. I. In case I is not a model of C, I is also not a model of P. In case C is redundant w.r.t. I, then I is a model of P iff I is a model of $P \setminus \{ C \}$. □

Corollary 6.3.4. Given an I that interprets definite program P's symbols as in P's least model and has rules $R^1 = R^1_{horn} \Rightarrow R^1_{prop}, \ldots, R^n = R^n_{horn} \Rightarrow R^n_{prop}$ be over-approximating, then whenever P contains a clause C such that $C' \subseteq C$ has $R^1, \ldots, R^n$ apply and $R^1_{prop} \wedge \ldots \wedge R^n_{prop}$ is unsatisfiable w.r.t. I, then C is redundant in P.

Proof. By I interpreting P according to P's least model, I is a model of P. Hence, by Proposition 6.3.3, we are forced to conclude that I is a model of $P \setminus \{ C \}$. More importantly, this means we know C's body is unsatisfiable, meaning that C does not add consequences to $P \setminus \{ C \}$. Conclude that C is redundant in P. □

This is the main result we will use to prune programs which are not (optimal) solutions from the hypothesis space. Note as well that as soon as we have established that $R^1_{prop} \wedge \ldots \wedge R^n_{prop}$ is unsatisfiable, we can infer that any hypothesis H containing a clause D such that $D' \subseteq D$ enables the same property formulas $R^1_{prop}, \ldots, R^n_{prop}$, then H is likewise not an optimal solution. This is what the conflict clauses we learn later on in this section will achieve.

Proving inconsistency is not guaranteed

Note that in the following we will use over-approximating properties to prune programs with clauses which have a combined property for which we can prove inconsistency. To illustrate the significance of this, let's say $\text{droplast}(A, B)$ is supposed to have the property $\text{len}(A) = \text{len}(B) + 1$ and that we have a clause

$$C = \text{droplast}(A, B) \leftarrow \text{tail}(A, C), \text{tail}(C, B).$$

As we depend on over-approximating rules being provided, it can be we only have access to a trivial over-approximating rule such as $\leftarrow \text{tail}(A, B) \Rightarrow \top$ that applies to the body literals of C. In this case we get the combined property $\text{len}(A) = \text{len}(B) + 1 \wedge \top \wedge \top$ for clause C, which clearly is satisfiable. As such, our ability to reject clauses based on intended interpretations requires sufficiently "tight" over-approximating properties to establish inconsistency.

Hence negative examples which overlap with over-approximating rules (e.g. negative example e_1^- from Example 6.3.2) can be useful as we are often not certain that all provided properties involved are strong enough to prove inconsistency with the target predicate's property.

6.3.3 Learning From Failures as Satisfiability Modulo Theories – Redux

We now expand the Learning From Failures framework to accommodate using over-approximating properties to prune hypotheses with rejected clauses, thereby rejecting these hypotheses as possible solutions. We do so by allowing over-approximating rules to be provided alongside the hypothesis space, the examples and the Background Knowledge. We build on the Learning From Failures as Satisfiability Modulo Theories perspective developed in Section 4.2.4. To formalize how we are going to reason about the properties, we will require that the formulas expressing the properties come from particular logic fragments.

Properties are satisfiable w.r.t. a theory

For satisfiability of the property formulas to be well-defined, the formulas must come from a logic fragment for which we can fix a theory of interpretations – see Section 3.2. In laying out the *Learning From Failures Modulo Theories* framework, we keep this fragment and theory abstract:

Definition 6.3.5. We let $(\Sigma_{Prop}, \mathcal{F}_{Prop}, \mathcal{T}_{Prop})$ stand for any triple of signature, set of formulas and a theory defining a first-order logic fragment.

Example 6.3.6. The signature Σ_{DL} contains, among others, variables, a binary symbol $>$ and the symbol for conjunction, see Definition 3.2.3. The formula $A > B \wedge B > C \wedge C > A$ belongs to $\mathcal{F}_{DL}$, see Definition 3.2.4. This formula is *not* satisfiable w.r.t. the theory $\mathcal{T}_{DL}$ as $\mathcal{T}_{DL}$ always interprets $>$ as the "greater-than" relation of standard arithmetic – see Definition 3.2.6 – and there is no assignment of the variables A, B and C to integers such that all three conjuncts hold (as there is no integer that is larger than itself).

The logic fragments of interest will be those for which we have solvers available. The perspective that a solver can detect when a combined property is inconsistent – and can thereupon generate a conflict clause – will lead to a modification of the LFF loop.

Satisfiability Modulo Theories as two-level satisfiability

In Section 4.2.4, we established that finding a solution to a LFF-problem $(E_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \mathcal{C} = \emptyset)$ is the same as finding a model (consisting of two interpretations) to the two-level satisfaction problem $E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E}$, see Proposition 4.2.34.

Recap The first satisfaction problem w.r.t. $E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E}$ is to find a constraint satisfying assignment, i.e. a model, of a formula $E_{\mathcal{H}}$ (which is expressed in a constraint language $\mathcal{L}$). A model I of $E_{\mathcal{H}}$ represents a hypothesis that's to be checked. This space of viable hypotheses, represented by $E_{\mathcal{H}}$, gets refined by the (set of constraint) formulas $\mathcal{C}$. That is, $E_{\mathcal{H}} \wedge \mathcal{C}$ has fewer models than $E_{\mathcal{H}}$.

The second level to satisfaction problem $E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E}$ is determined by the interpretation I. We have that $\langle\!\langle E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \rangle\!\rangle_I^{Horn}$ is a *Horn* program intended to be interpreted w.r.t. the theory $\mathcal{T}_{LM(\mathcal{BK} \cup H)}$ where $H = \langle\!\langle E_{\mathcal{H}} \rangle\!\rangle_I$. The corresponding satisfaction problem is whether the (sole) interpretation $LM(\mathcal{BK} \cup H)$ of this theory is a model of $\langle\!\langle E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \rangle\!\rangle_I^{Horn} = \mathcal{BK} \cup \mathcal{H} \cup \mathcal{E}$. When this is the case, all of $\mathcal{E}^+$ is a consequence of $\mathcal{BK} \cup H$ and none of $\mathcal{E}^-$ is – exactly what the test stage is checking.

2nd second level We now expand satisfaction problem $E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E}$ to $E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R}$. The idea is that we maintain the two-level structure, though we now add another constraint satisfaction problem at the second level – a property-checking problem separate from but sitting alongside the example-testing problem. Contrary to $\mathcal{T}_{LM(\mathcal{BK} \cup H)}$ being dependent on a changing formula H, we now stick to the traditional SMT-approach where there is one fixed (background) theory $\mathcal{T}_{Prop}$ which gives interpretations to a fixed set of function and predicate symbols as well as the possible interpretations for first-order variables. These are the symbols we can use in our over-approximating properties.

The idea is that given a first-level interpretation I of $E_{\mathcal{H}}$, we now have two mappings, $\langle\!\langle E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R} \rangle\!\rangle_I^{Horn}$ and $\langle\!\langle E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R} \rangle\!\rangle_I^{Prop}$, yielding two second level satisfaction problems. We have that $\langle\!\langle E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R} \rangle\!\rangle_I^{Prop}$ is intended to be set of over-approximating properties that result from applying $\mathcal{R}$ to $\langle\!\langle E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R} \rangle\!\rangle_I^{Horn}$.

Learning From Failures Modulo Theories definition

We now expand the Learning From Failures framework by allowing for Learning From Failures *Modulo Theories*. As we reason about the satisfiability of the over-approximating properties, it is natural to extend the SMT-presentation of Learning From Failures - which already phrased finding a hypothesis that is correct w.r.t. examples as a question

of satisfiability. In the following definitions and propositions, we carefully identify to which logic fragments the different formulas and interpretations belong.

Definition 6.3.7. Given a LFF-input $(E_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \emptyset)$ represented by $E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E}$ and $\Sigma^{Horn,Prop}$-formulas $\mathcal{R}$, a $\Sigma^{\mathcal{L},Horn,Prop}$-formula $E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R}$ represents a *Learning From Failures Modulo Theories input* when, for each $\Sigma^{\mathcal{L}}$-interpretation I,

1. $\langle\!\langle E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R} \rangle\!\rangle_I^{Horn} = \langle\!\langle E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \rangle\!\rangle_I^{Horn}$
2. $\langle\!\langle E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R} \rangle\!\rangle_I^{Prop} = \bigcup \{ \mathcal{R}(C_i) \mid (\forall \mathbf{v} \colon C_i) \in \langle\!\langle E_{\mathcal{H}} \rangle\!\rangle_I \}$

In essence, this definition just expands Definition 4.2.31 such that $\mathcal{L}$-interpretations map not only to *Horn*-formulas but also to *Prop*-formulas.

The interpretations for LFFMT-formulas come in triples: an interpretation I from $\mathcal{T}_{\mathcal{L}}$ for symbols from $\Sigma^{\mathcal{L}}$, an interpretation J from $\mathcal{T}_{LM(\mathcal{BK} \cup H)}$ (which depends on $H = \langle\!\langle E_{\mathcal{H}} \rangle\!\rangle_I$) for symbols of Σ^{Horn} and an interpretation K from $\mathcal{T}_{Prop}$ for symbols from Σ^{Prop}. It is with respect to these interpretations that we identify solutions to LFFMT-problems.

Definition 6.3.8. Given a $\Sigma^{\mathcal{L},Horn,Prop}$-formula $E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R}$ that represents a LFFMT-input, the corresponding *Learning From Failures Modulo Theories problem* is to find an (intended) interpretation I,J,K which is a model of $E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R}$, in which case we say $\langle\!\langle E_{\mathcal{H}} \rangle\!\rangle_I$ is a solution.

The following two propositions characterise the correctness of extending LFF with over-approximating rules in this way:

Proposition 6.3.9. When interpretation I,J,K is a model of LFFMT-input $E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R}$, then $\langle\!\langle E_{\mathcal{H}} \rangle\!\rangle_I$ is a solution to LFF-problem $(E_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \emptyset)$.

Proof. Let I,J,K be a model of $E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R}$. Then I must be a model of $E_{\mathcal{H}}$ and $\langle\!\langle E_{\mathcal{H}} \rangle\!\rangle_I = H$ for some $H \in \mathcal{H}$. In turn, $\langle\!\langle E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R} \rangle\!\rangle_I^{Horn}$ must then be $\mathcal{BK} \cup H \cup \mathcal{E}$. J must then be the sole intended interpretation from the theory for this formula, namely $LM(\mathcal{BK} \cup H)$. As J is a model of $\mathcal{BK} \cup H \cup \mathcal{E}$, it must be that $\mathcal{E}^+$ is among the consequences of $\mathcal{BK} \cup H$ while no $e^- \in \mathcal{E}^-$ is, which makes H a solution. Whether interpretation K is a model of $\langle\!\langle E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R} \rangle\!\rangle_I^{Prop}$ or not only serves to prune pairs of interpretations I,J and hence does not influence whether $\langle\!\langle E_{\mathcal{H}} \rangle\!\rangle_I$ is a solution or not. □

Conversely, when there is an *optimal* solution, then there is an interpretation that is a model of the LFFMT-formula:

Proposition 6.3.10. When H is an optimal solution to LFF-problem $(E_{\mathcal{H}}, \mathcal{BK}, \mathcal{E}, \emptyset)$, then there exists an interpretation I,J,K which is a model of LFFMT-input $E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R}$ whenever rules $\mathcal{R}$ are over-approximating w.r.t. $LM(\mathcal{BK} \cup H)$.

Proof. Suppose $H \in \mathcal{H}$ is an optimal solution. By the models of $E_{\mathcal{H}}$ representing hypotheses, there exists I s.t. $[\![E_{\mathcal{H}}]\!]_I = \top$ and $\langle\!\langle E_{\mathcal{H}} \rangle\!\rangle_I = H$. We have that $\langle\!\langle E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R} \rangle\!\rangle_I^{Horn} = \mathcal{BK} \cup H \cup \mathcal{E}$. As H is a solution, it must be that the least model of $\mathcal{BK} \cup H$, i.e. $J = LM(\mathcal{BK} \cup H)$, is a model of $\mathcal{BK} \cup H \cup \mathcal{E}$, which is indeed an intended interpretation. By Corollary 6.3.4 we have that whenever $\mathcal{R}$ are over-approximating w.r.t. an interpretation J, K – J interpreting *Horn*-symbols and K interpreting *Prop*-symbols – and $\langle\!\langle E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R} \rangle\!\rangle_I^{Prop}$ is unsatisfiable w.r.t J, K then either J, K is not a model of $\mathcal{BK} \cup H$ or H contains a redundant clause. As J, K extends J it will model $\mathcal{BK} \cup H$ and as H is

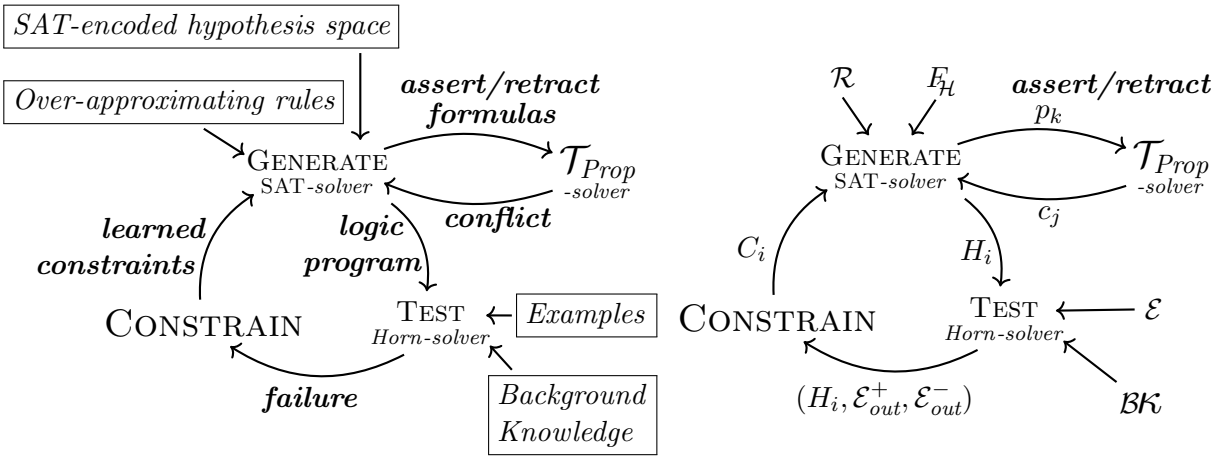


Figure 6.2: In Learning From Failures Modulo Theories, there is an additional loop to communicate with the $\mathcal{T}_{Prop}$-theory solver.

an *optimal* solution, by Assumption 4.2.16 we know H contains no redundant clause. As we assert that the rules $\mathcal{R}$ are over-approximating, this leaves $\langle\!\langle E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R} \rangle\!\rangle_I^{Prop}$ is unsatisfiable w.r.t J, K as the problematic assumption. Conclude that there must be an interpretation K such that $\langle\!\langle E_{\mathcal{H}} \wedge \mathcal{BK} \wedge \mathcal{E} \wedge \mathcal{R} \rangle\!\rangle_I^{Prop}$ is satisfiable, making I,J,K a model of the overall formula. □

6.3.4 Another theory, another stage, another loop

In Section 4.4 we showed that LFF-problems can be solved by way of a three-stage loop. We now expand that approach – consisting of a generate stage, a test stage and a constrain stage – to solving LFFMT problems. Just like the test stage is effectively a solver for *Horn*-formulas, we assume there is a solver available for deciding satisfiability of *Prop*-formulas with respect to a theory $\mathcal{T}_{Prop}$. This solver can then be placed in a separate loop, where it checks for the consistency of the property formulas that arise from rules applying to guesses for clauses. Upon detecting that these formulas are unsatisfiable together, the solver learns a *conflict clause* that get communicated to the generate stage's SAT-solver.

$\mathcal{T}_{Prop}$-solver for rejecting hypotheses

In order to implement a LFFMT system, we now require that whichever concrete logic fragment *Prop* stands for, there is a solver available to determine satisfiability of *Prop*-formulas w.r.t. a theory $\mathcal{T}_{Prop}$.

Propagator We assume a $\mathcal{T}_{Prop}$-solver S is implemented as an *propagator* [70, 56], i.e. a theory solver implementing the standard interface for communicating with a SAT-solver. That is, S takes as inputs *Prop*-formulas, e.g. p_i, though in an incremental fashion, i.e. S accumulates the formulas that get asserted over time. An alternative input is that previously asserted *Prop*-formulas are to be retracted. The idea is that as a SAT-solver is making decisions in an effort to construct a satisfying full assignment, different

background theory formulas become enabled, and as the solver backtracks, they become disabled again.

Conflicts When solver-as-a-propagator S detects that $\bigwedge_{k \in I} p_k$ (for some index set I) is unsatisfiable, S generates a *conflict clause*. For most solvers this is nothing more than a disjunction of the negation of the (sub-)formulas that caused the unsatisfiablity [17], i.e. $c_j = \bigvee_{k \in I} \neg p_k$. (Equivalently, c_j can be presented as a *nogood*[4]: $\bot \leftarrow (\bigwedge_{k \in I} p_k)$.) The way this works in the SAT-solver is that each first-order atom (or compound formula), with free variables, is treated as if it is ground, i.e. it is just treated as a propositional variable which can be used in propositional (conflict) clauses [12].

Upon deriving this conflict clause, S then informs the SAT-solver about this formula. The SAT-solver will henceforth use this clause to elide calls to solver S as it already knows that this combination of formulas is going to be unsatisfiable (in essence, the learned clause serves as a form of caching).

Conflict explanation To make the learned conflict clauses more effective, many background theory solvers implement a form of *conflict explanation* [17]: i.e. identifying which subset of formulas actually caused the conflict (akin to the failure explanation of the previous chapter, c.f. Section 5.4). A solver which does this can, at times, find an index set $J \subseteq I$ such that $c_j = \bigwedge_{k \in J} p_k$ is already unsatisfiable. Just as in the previous chapter, the argument is that these smaller clauses allow the SAT-solver to prune more partial assignments (exactly those that derive the same *Prop*-formulas), thereby pruning the search space more effectively and eliding the work of checking *Prop*-formulas for this part of the search space.

ILPMT In terms of our application of this solver technology, we get that the over-approximating properties for clauses become active as the solver is building up (partially constructed) clauses of hypotheses. Even when the SAT-solver has not made a guess for a full clause yet, and instead has only decided on a couple of literals for a clause, the over-approximating properties for these literals already get communicated to S which can then determine that these properties are already unsatisfiable, irrespective of the context of yet to be added literals. Conflict explanation effectively allows the solver to determine that some of the guessed literals in the clause were not necessary for the conflict, e.g. if the SAT-solver makes guesses for $\text{tail}(A, B)$, $\text{tail}(C, D)$ and $\text{tail}(B, A)$ being part of a clause's body, then conflict explanation could work out that the unsatisfiablity of the combined properties is only due to properties of the first and last literal being active.

Second loop

In Section 4.4 we have that the generate stage's SAT-solver accumulates constraints by keeping track of $\mathcal{C}_{0..i} = C_i \cup \mathcal{C}_{0..i-1}$ where C_i is the constraint learned from the ith iteration of the loop, i.e. from testing the ith hypothesis, H_i.

[4] This representation and terminology is especially common in the Answer Set Programming literature [60].

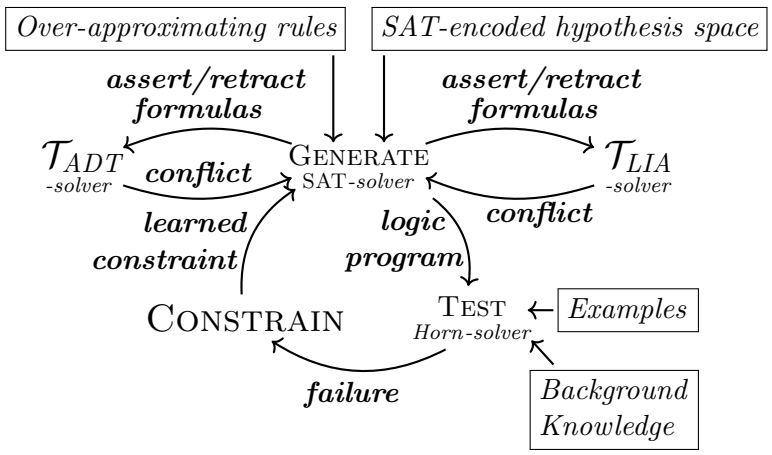


Figure 6.3: The Learning From Failures loop expanded with two theory solvers.

Using a $\mathcal{T}_{Prop}$-solver(-as-a-propagator), we now form a separate loop. See Figure 6.2. While the generate stage's SAT-solver gradually builds up assignments (and backtracks these decisions), the formulas for different over-approximating properties become enabled and disabled. The SAT-solver ensures that the formulas currently enabled for the $\mathcal{T}_{Prop}$-solver correspond to the partial propositional assignment it has built up. Whenever the $\mathcal{T}_{Prop}$-solver determines that the combination of currently asserted *Prop*-formulas is unsatisfiable, it sends a conflict clause c_j to the SAT-solver.

With the two loops present, we hence have that the generate stage's SAT-solver keeps track of two kinds of pruning formulas: the constraints C_i that the constrain stage derives from a hypothesis failing on the examples and the conflict clauses c_j that come from the $\mathcal{T}_{Prop}$-solver. Therefore the SAT-solver now maintains the constrains $\mathcal{C}_{0..i}^{0..j}$ with $\mathcal{C}_{0..0}^{0..0} = \emptyset$ and the recursive equations $C_{0..i}^{0..j} = C_{0..i-1}^{0..j} \cup \{ C_i \}$ and $C_{0..i}^{0..j} = C_{0..i}^{0..j-1} \cup \{ c_j \}$.

Hence with both loops running at the same time we accumulate more constrains on the SAT-encoding of the hypothesis space and hence can narrow down the set of viable hypotheses faster.

More than one properties theory

We are not limited to just one logic fragment to express properties in. For each logic fragment for which we have a theory-solver-as-a-propagator, we can extend the three-stage approach to LFF with another loop.

Figure 6.3 illustrates the expanded loop in case we have a solver both for a Linear Integer Arithmetic logic fragment and for a theory of Algebraic Datatypes. Given over-approximating rules making use of formulas from both fragments, the SAT-solver of the generate stage informs each theory solver of the formulas from its fragment as appropriate, according to the over-approximating rules being enabled and disabled. Each theory solver injects its own conflict clauses, giving potential for even more pruning.

6.4 Implementation

We now present one way of building a system that can tackle Learning From Failures Modulo Theories-problems. For our implementation, we expand the `Popper` system from

Section 4.5 by having the ASP-solver of the generate stage communicate with a background theory solver[5]. Hence we produce a system that prunes hypotheses through two loops, exactly as set out in the previous section.

6.4.1 From Popper using Clingo to Popper[$\mathcal{DL}$] using Clingo[$\mathcal{DL}$]

We introduce the Popper[$\mathcal{DL}$] system that can solve Learning From Entailment-problems when specified as LFF- *and* LFFMT-problems. Popper[$\mathcal{DL}$] is build on Popper and hence we instantiate constraint language $\mathcal{L}$ with Answer Set Programming. As such, the SAT-solver of Popper's generate stage is an ASP-solver, in particular we use Clingo [58].

Clingo as SMT-solver

The Clingo-solver was effectively made into a SMT-solver when it gained support for a propagator interface for interacting with background theory solvers [70, 75]. As of this writing, there are a couple of background theory solvers for Clingo: Clingo[$\mathcal{DL}$] [70], Clingo[$\mathcal{LIA}$] [70], and Clingcon (constraints over finite-domain integers) [9]. We will focus on adding support for the simplest fragment of these: Difference Logic ($\mathcal{DL}$).

Adding support to Popper for $\mathcal{LIA}$ and the Finite-Domain fragments using the existing solvers that interface with Clingo follows the exact same recipe. Hence we can with relative ease also also obtain systems such as Popper[$\mathcal{LIA}$], Popper[$\mathcal{FD}$] and combinations such as Popper[$\mathcal{DL}+\mathcal{LIA}$] and Popper[$\mathcal{LIA}+\mathcal{FD}$] (where $\mathcal{FD}$ denotes constraints over finite domains).

Supported Difference Logic literals

As already discussed in Section 3.2, we have that the atoms of the Difference Logic fragment are restricted to the shape $t_1 - t_2 \leq c$ (also when written in this concrete notation, we will consistently use red colour to indicate symbols and formulas from the $\mathcal{DL}$-fragment). Terms t_1 and t_2 are either integer constants or variables ranging over the integers. The term c always has to be an integer constant.

Note that $\geq$-difference inequalities, like $t_1 - t_2 \geq c$ can be rewritten to $t_2 - t_1 \leq -c$. Similarly equalities of the shape $V = W + c$ and $V = W - c$ can be rewritten as a conjunction of Difference Logic atoms, i.e. $V - W \leq c \wedge W - V \leq -c$ and $V - W \leq -c \wedge W - V \leq c$.

The full set of Difference Logic formulas supported by Clingo[$\mathcal{DL}$], and hence by Popper[$\mathcal{DL}$], are atoms as described above and boolean combinations thereof, i.e. if f_1 and f_2 are $\mathcal{DL}$-formulas then $f_1 \wedge f_2$ and $f_1 \vee f_2$ and $\neg f_1$ are $\mathcal{DL}$-formulas. As such, $\mathcal{DL}$-formulas can be put in conjunctive normal form (CNF).

6.4.2 Background theory atoms and when they are active

Clingo[$\mathcal{DL}$] extends Clingo's ASP-modelling language by additionally allowing atoms of the shape

`&diff{` t_1 `-` t_2 `}` `<=` c.

[5]As discussed in Section 3.3, having a separate solver for background theory reasoning is called the "lazy"-approach to SMT [12]. Whilst we do not make use of it in this thesis, what we present works just as well with logic fragments whose satisfiability problem can be encoded directly in ASP, i.e. the "eager" approach to SMT.

The actual solving components (i.e. the stable models SAT-solver and the $\mathcal{DL}$-solver) require that t_1 and t_2 are (arbitrary) ground terms which the $\mathcal{DL}$-solver then interprets as names of variables (ranging over the integers), in a single global namespace. The term c must be an integer. Hence the above ASP-code is an encoding of the difference-inequality $t_1 - t_2 \leq c$.

Given an inequality like $\text{len}(A) - \text{len}(B) \leq c$, we recognize that $\text{len}(A)$ and $\text{len}(B)$ are of integer type and hence the whole term, e.g. $\text{len}(A)$, can be treated as a $\mathcal{DL}$-variable. If we are in possession of a function *encodeVar*, which turns a (Horn)-variable into a ground term – see Section 4.5.5 – we can write inequalities with len-terms as a Clingo[$\mathcal{DL}$] inequality:

$$\texttt{\&diff\{ len(}\textit{encodeVar}(A)\texttt{) - len(}\textit{encodeVar}(B)\texttt{) \} <= } c$$

where if $\textit{encodeVar}(A) \mapsto 0$ and $\textit{encodeVar}(B) \mapsto 1$ then both $\text{len}(0)$ and $\text{len}(1)$ are interpreted as $\mathcal{DL}$-variables.

Grounding variables in rules with theory atoms

$\mathcal{DL}$-atoms can occur in both the head and body of ASP-rules [70]. All variables in these rules be need to be grounded. Clingo[$\mathcal{DL}$] relies on the Gringo grounder to obtain the relevant ground rules.

Example 6.4.1. We show how the grounder derives the relevant ground rules and how (compound) terms in `&diff`-atoms are interpreted as $\mathcal{DL}$-variables. Let us consider the following toy Clingo[$\mathcal{DL}$] program P:

```
1  happens(event(a)). % event(a) is asserted to be true
2  happens(event(b)).
3  {happens(event(c)).} % choice rule: event(c) can be true or false
4  during(event(a),0,5).
5  during(event(b),5,15).
6  during(event(c),3,5).
7  before(event(a),event(b)).
8  before(event(b),event(c)).
9  &diff{ 0 - time(V) } <= -LB :- happens(event(V)),during(event(V),LB,UB).
10 &diff{ time(V) - 0 } <= UB :- happens(event(V)),during(event(V),LB,UB).
11 &diff{ time(V) - time(W) } <= -1 :-
12   happens(V),happens(W),before(event(V),event(W)).
```

The grounder derives the following ground rules from the three rules which involve ASP-variables.

```
1 &diff{ 0 - time(a) } <= -0.
2 &diff{ 0 - time(b) } <= -5.
3 &diff{ 0 - time(c) } <= -3 :- happens(event(c)).
4 &diff{ time(a) - 0 } <= 10.
5 &diff{ time(b) - 0 } <= 15.
6 &diff{ time(c) - 0 } <= 5 :- happens(event(c)).
7 &diff{ time(a) - time(b) } <= -1.
8 &diff{ time(b) - time(c) } <= -1 :- happens(event(c)).
```

Hence we have that there are 8 literals that Clingo[$\mathcal{DL}$] is informed about. Clingo[$\mathcal{DL}$]'s $\mathcal{DL}$-solver interprets `time(a)`, `time(b)` and `time(c)` as $\mathcal{DL}$-variables. Five of these literals must hold unconditionally (which the grounder already derives by knowing that `happens(event(a))` etc. must hold unconditionally). There are three ground rules conditional on whether `happens(event(c))` is true or not. We see that when `happens(event(c))` is false, a $\mathcal{DL}$-solver can find a satisfying assignment for $\mathcal{DL}$-variables `time(a)` and `time(b)`, e.g. 3 and 7 (and for `time(c)` which is completely unconstrained). Hence

$$I_1 = \left\{ \begin{array}{c} \texttt{happens(event(a))}, \texttt{happens(event(b))}, \\ \texttt{during(event(a),0,5)}, \texttt{during(event(b),5,15)}, \texttt{during(event(c),3,5)}, \\ \texttt{before(event(a),event(b))}, \texttt{before(event(b),event(c))} \end{array} \right\}$$

is an answer set of this (grounded) program, as $\langle\!\langle GRD(P) \rangle\!\rangle_{I_1}^{\mathcal{DL}}$ is satisfiable while

$$I_2 = \left\{ \begin{array}{c} \texttt{happens(event(a))}, \texttt{happens(event(b))}, \texttt{happens(event(c))}, \\ \texttt{during(event(a),0,5)}, \texttt{during(event(b),5,15)}, \texttt{during(event(c),3,5)}, \\ \texttt{before(event(a),event(b))}, \texttt{before(event(b),event(c))} \end{array} \right\}$$

is *not* an answer set because $\langle\!\langle GRD(P) \rangle\!\rangle_{I_2}^{\mathcal{DL}}$ is not satisfiable, due to `time(c)` being constrained to be both at most 5 (`c`'s last possible time) and at least 6 (`c` should come after `b` and `b`'s earliest possible time is 5).

6.4.3 Encoding over-approximating rules with $\mathcal{DL}$-properties

We now show how `Popper`[$\mathcal{DL}$] encodes over-approximating rules with properties which are $\mathcal{DL}$-formulas. As in Section 4.5.5, we will denote ASP-code in bold to highlight the actual code that gets generated.

Our over-approximating rules are of the shape

$$R = \texttt{head} \leftarrow \texttt{body}_1, \ldots, \texttt{body}_n \Rightarrow \mathit{prop}$$

where `head` and $\texttt{body}_1, \ldots, \texttt{body}_n$ are first-order (Horn) atoms containing variables that are also allowed to occur in $\mathcal{DL}$-formula *prop*. Either `head` or $\texttt{body}_1, \ldots, \texttt{body}_n$ is allowed to be omitted (but not all literals). For instance, R could be

$$\leftarrow \text{tail}(A, B) \Rightarrow \text{len}(A) = \text{len}(B) + 1 \wedge \text{len}(B) \geq 0$$

The intention is for such a rule to apply to each clause which contains (the guess of) a body literal with predicate tail/2 (and apply multiple times if there are multiple tail/2 body literals). Just like in Section 4.5.3, we achieve this by replacing variables such as A and B by ASP variables **`V0`** and **`V1`** which in term can be grounded to terms which represent other Horn variables, like C and D.

We translate a rule $R = R_{horn} \Rightarrow R_{\mathcal{DL}}$ by its two constituent parts. First, let $R_{horn} = \texttt{head} \leftarrow \texttt{body}_1, \ldots, \texttt{body}_n$. By using the functions *encodeHead* and *encodeBody* from Section 4.5.5 we have that

$$\begin{array}{l} \mathit{encodeHorn}(R_{horn}) := \\ \quad \mathit{encodeHead}(\texttt{Cl}, \texttt{head}), \mathit{encodeBody}(\texttt{Cl}, \texttt{body}_1), \ldots, \mathit{encodeBody}(\texttt{Cl}, \texttt{body}_n) \end{array}$$

We have that the variables $A, B, \ldots$ of the Horn clause are transformed into ASP variables, by way of $\mathit{encodeVar}(A) = \texttt{V0}$, $\mathit{encodeVar}(B) = \texttt{V1}$, see Section 4.5.5. During grounding,

these variables and ASP-variable `Cl` will be bound to the possible variable ids (0, 1, etc.) and clause ids (0, 1, etc.), respectively.

For translating the $\mathcal{DL}$-property, we introduce the following mapping for $\mathcal{DL}$-atoms:

$$\begin{aligned}&\mathit{encodeDL}(t_1 - t_2 \leq c) :=\\ &\quad \texttt{\&diff\{ var(Cl,}\mathit{encodeTerm}(t_1)\texttt{) - var(Cl,}\mathit{encodeTerm}(t_2)\texttt{) \} <= } c\end{aligned}$$

The function *encodeTerm* keeps compound terms, i.e. (nested) function applications, while mapping $\mathcal{DL}$-variables to ASP variables via *encodeVar*. For example, $\mathit{encodeTerm}(A) =$ `V0` and $\mathit{encodeTerm}(\text{len}(B)) =$ `len(V1)`. We hence have

$$\begin{aligned}&\mathit{encodeDL}(\text{len}(B) - \text{len}(A) \leq -1) =\\ &\quad \texttt{\&diff\{ var(Cl,len(V1)) - var(Cl,len(V0)) \} <= -1}\end{aligned}$$

For translating a generic rule R as a whole, we introduce a mapping that does a case distinction on the CNF-form of R's $\mathcal{DL}$-formula.

$$\begin{aligned}&\mathit{encodeRule}(R_{horn} \Rightarrow f_1 \wedge \ldots \wedge f_k) :=\\ &\quad \mathit{encodeDL}(f_1) \texttt{ :- } \mathit{encodeHorn}(R_{horn})\texttt{.} && \text{if } f_1 \text{ is a } \mathcal{DL}\text{-atom}\\ &\quad \mathit{encodeDisjunction}(R_{horn}, f_1) && \text{if } f_1 \text{ is a disjunction of } \mathcal{DL}\text{-atoms}\\ &\quad \ldots\\ &\quad \mathit{encodeDL}(f_j) \texttt{ :- } \mathit{encodeHorn}(R_{horn})\texttt{.} && \text{if } f_k \text{ is a } \mathcal{DL}\text{-atom}\\ &\quad \mathit{encodeDisjunction}(R_{horn}, f_k) && \text{if } f_k \text{ is a disjunction of } \mathcal{DL}\text{-atoms}\end{aligned}$$

We hence get that every conjunct of a $\mathcal{DL}$-formula gets its own rule when encoded into ASP. Furthermore, when a conjunct consists of a disjunction, we encode that disjunction into multiple rules via the following function:

$$\begin{aligned}&\mathit{encodeDisjunction}(R_{horn}, a_1 \vee \ldots \vee a_m) :=\\ &\quad \mathit{encodeDL}(a_1) \texttt{ :- } \mathit{encodeHorn}(R_{horn})\texttt{,}\\ &\qquad\qquad \texttt{not } \mathit{encodeDL}(a_2)\texttt{,} \ldots \texttt{,not } \mathit{encodeDL}(a_m)\texttt{.}\\ &\quad \mathit{encodeDL}(a_i) \texttt{ :- } \mathit{encodeHorn}(R_{horn})\texttt{,}\\ &\qquad\qquad \texttt{not } \mathit{encodeDL}(a_1)\texttt{,} \ldots \texttt{,not } \mathit{encodeDL}(a_{i-1})\texttt{,}\\ &\qquad\qquad \texttt{not } \mathit{encodeDL}(a_{i+1})\texttt{,} \ldots \texttt{,not } \mathit{encodeDL}(a_m)\texttt{.}\\ &\quad \mathit{encodeDL}(a_m) \texttt{ :- } \mathit{encodeHorn}(R_{horn})\texttt{,}\\ &\qquad\qquad \texttt{not } \mathit{encodeDL}(a_1)\texttt{,} \ldots \texttt{,not } \mathit{encodeDL}(a_{m-1})\texttt{.}\end{aligned}$$

The function *encodeDisjunction* generates a rule for each disjunct. It explicitly says that whenever all other disjuncts are not true, then the remaining disjunct has to have a satisfying assignment.

The following two examples demonstrate this encoding on concrete over-approximating rules:

Example 6.4.2. If $R = \leftarrow \text{tail}(A, B) \Rightarrow \text{len}(A) = \text{len}(B) + 1 \wedge \text{len}(B) \geq 0$ is our over-approximating rule, then its translation, $\mathit{encodeRule}(R)$, is

```
&diff{ var(Cl,len(V0)) - var(Cl,len(V1)) <= 1}:-
    body_literal(Cl,tail,2,(V0,V1)).
&diff{ var(Cl,len(V1)) - var(Cl,len(V0)) <= -1}:-
    body_literal(Cl,tail,2,(V0,V1)).
&diff{ 0 - var(Cl,len(V1)) <= 0}:-
    body_literal(Cl,tail,2,(V0,V1)).
```

These rules become active for each body literal with predicate tail/2 contained in each clause 0, 1, etc. given all the possible clause id assignments for `Cl`. Due to variables `V0` and `V1`, each possible renaming of variables A and B gets detected and leads to its own set of three $\mathcal{DL}$-constraints. The first two capture the equality $\text{len}(A) = \text{len}(B) + 1$, relying on the antisymmetry of $\leq$. The third $\mathcal{DL}$-formula just guarantees that $\text{len}(B) \geq 0$ holds no matter the renaming of B.

Example 6.4.3. If $R = \text{evens}(A) \leftarrow \text{head}(A, B) \Rightarrow B = 0 \vee B \geq 2$ is our over-approximating rule, then its translation, $encodeRule(R)$, is

```
&diff{ 0 - var(Cl,V1) <= 0}:-
    head_literal(Cl,evens,1,(V0)),body_literal(Cl,head,2,(V0,V1)).
&diff{ 0 - var(Cl,V1) <= -2}:-
    not &diff{ var(Cl,V1) - 0 <= 0},
    head_literal(Cl,evens,1,(V0)),body_literal(Cl,head,2,(V0,V1)).
```

The first rule becomes active whenever there is a clause, with id `Cl`, that has a head literal with predicate evens/1 and a body literal with predicate head/2 such that head's first argument is the same as evens' sole argument. Whenever this is the case, the rule asserts that, for all possible substitutions for variable B, i.e. whatever the grounding of `V2`, there is a corresponding $\mathcal{DL}$-variable, by the name `var(Cl,V2)` whose integer assignment must be at least zero. The second rule is active when the same literals are present in clause `Cl` *and* `var(Cl,V2)` is *not* (at most) zero. In that case, `var(Cl,V2)` is forced to be at least 2. Note that the $\mathcal{DL}$-variables are specific to the clause with id `Cl`.

In the experimental section, we will treat this encoding as just an implementation detail and work with the normal first-order logic presentation of hypotheses and over-approximating rules.

6.5 Experimental evaluation

We seek to answer the following questions:

Q1 Can satisfiability checking of over-approximating properties reduce the number of iterations of LFF's three-stage loop?

Q2 Can satisfiability checking of over-approximating properties reduce learning times?

Note that because checking over-approximating properties using solvers takes time, an affirmative answer to **Q1** need not mean that we necessarily see evidence for **Q2** being the case.

For simplicity's sake (as well as time constraints), we re-examine the list transformation tasks and restrict ourselves to Difference Logic properties.

On methodology

The Clingo ASP-solver (i.e. SAT-solver w.r.t. answer sets) employs heuristics to guide the search for models [58]. Across runs, these heuristics cause variance w.r.t. how many candidates are considered before a solution is found – see the experimental sections from the previous chapters. For the following experiments, we focus on finding *all* optimal solutions rather than just the first one. The reason to do so is to force the SAT-solver to explore the same search space in each run. This accomplishes three things:

1. Search heuristics benefit less from randomly choosing one path over another when a whole space (up to the size of the optimal solutions) must be explored.
2. Accounts for search heuristics that are perpetually unlucky, in the sense that the very last candidate they consider (of optimal solution size) might be the solution.
3. Allows for more accurately gauging how effective pruning of the search space is.

In short, by searching for all optimal solutions there should be less noise in our results. Section 6.5.3 contains a discussion regarding this claim based on the experimental results.

Versions of `Popper`[$\mathcal{DL}$]

As explained, we rely on the existing background theory solvers for Clingo.

As it so happens there is Clingo[$\mathcal{DL}$] [74], written in C++ (and communicating with the Clasp solver of Clingo via C++-APIs), as well as a Python version, which we dub Clingo[$\mathcal{DL}_{\text{py}}$] [73]. The latter is easily extensible in Python. This made it possible for us to instrument Clingo[$\mathcal{DL}_{\text{py}}$] with `Popper`'s logging. In the following we use both Clingo[$\mathcal{DL}$], for performant solving of $\mathcal{DL}$-formulas, and Clingo[$\mathcal{DL}_{\text{py}}$], to track how often $\mathcal{DL}$-formulas are checked and how often conflicts get detected and injected.

Alongside the numbers on checks and conflicts, we report the number of programs and time it took to find solutions when using Clingo[$\mathcal{DL}_{\text{py}}$]. However, as Clingo[$\mathcal{DL}_{\text{py}}$] is only used to get representative numbers on the checking of $\mathcal{DL}$-formulas for conflicts, we will show the other numbers in gray. We note now that both systems use the same operating modes (check-on-fixpoint [75]) and find the same conflicts when presented with the same formulas. This is further confirmed by the number of iterations of `Popper`'s loop often being identical across the two solvers (see the tables below).

The previously introduced `Popper`[$\mathcal{DL}$] is the version of `Popper` that uses the C++ $\mathcal{DL}$-solver while we use `Popper`[$\mathcal{DL}_{\text{py}}$] to denote the version of `Popper` making use of Clingo[$\mathcal{DL}_{\text{py}}$]. In the following, we will not disucss the distinction between these two systems any further.

List transformation problems

We take 10 list transformation tasks from Section 4.6.3 and Section 5.6.2. The setup is exactly the same as in these previous sections: we use the same predicates as background knowledge and the learning tasks concern the same predicates as in these earlier sections. The exact settings we use for setting the hypothesis space can be found in Appendix C.1.

6.5.1 List transformations with BK $\mathcal{DL}$-properties

In this first experiment, we assume that the ILP-system user is only able to annotate their Background Knowledge with properties. For each BK predicate from the previous experiments, Section 4.6.3 and Section 5.6.2, it is easy to provide an over-approximating property, even in the very restrictive $\mathcal{DL}$-logic fragment.

In Figure 6.4, we have the set of (sound) over-approximating rules $\mathcal{R}_{\mathcal{BK}}$, with (at least) one rule for each BK predicate. Note that each antecedent is a negative literal, meaning that these literals most occur as body literals (which is the only valid position for BK predicates in LFF hypotheses). The $\mathcal{DL}$-property of sum/3 is the only non-trivial one and hence has been split across two rules.

$$\leftarrow \text{tail}(A,B) \Rightarrow \text{len}(A) = \text{len}(B) + 1 \wedge \text{len}(B) \geq 0$$
$$\leftarrow \text{cons}(A,B,C) \Rightarrow \text{len}(C) = \text{len}(B) + 1 \wedge \text{len}(B) \geq 0$$
$$\leftarrow \text{snoc}(A,B,C) \Rightarrow \text{len}(C) = \text{len}(A) + 1 \wedge \text{len}(B) \geq 0$$

$$\leftarrow \text{empty}(A) \Rightarrow \text{len}(A) = 0 \qquad \leftarrow \text{member}(A,B) \Rightarrow \text{len}(A) \geq 1$$
$$\leftarrow \text{head}(A,B) \Rightarrow \text{len}(A) \geq 1 \qquad \leftarrow \text{geq}(A,B) \Rightarrow A \leq B$$
$$\leftarrow \text{increment}(A,B) \Rightarrow B = A + 1 \qquad \leftarrow \text{decrement}(A,B) \Rightarrow B = A - 1$$
$$\leftarrow \text{even}(A) \Rightarrow A = 0 \vee A \geq 2 \qquad \leftarrow \text{odd}(A) \Rightarrow A = 1 \vee A \geq 3$$
$$\leftarrow \text{zero}(A) \Rightarrow A = 0 \qquad \leftarrow \text{one}(A) \Rightarrow A = 1$$

$$\leftarrow \text{sum}(A,B,C) \Rightarrow (A \geq 0 \wedge B \leq C) \vee (A \leq 0 \wedge C \leq B)$$
$$\leftarrow \text{sum}(A,B,C) \Rightarrow (B \geq 0 \wedge A \leq C) \vee (B \leq 0 \wedge C \leq A)$$

Figure 6.4: Over-approximating rules $\mathcal{R}_{\mathcal{BK}}$ with Difference Logic properties of predicates in the BK for the list transformation learning tasks.

The first rule for $\text{sum}(A,B,C)$ – which only applies when this atom occurs as a body literal – does a case distinction on whether A is non-negative and/or non-positive. In the first case we know that C must certainly be as big as B. In the second case, B gets substracted by the absolute value of A, so C must be at most B. The second over-approximating rule for $\text{sum}(A,B,C)$ does the case distinction on B instead of A.

For details on how these over-approximating rules with $\mathcal{DL}$-properties get translated to concrete ASP code, see Section 6.4.3.

Setup

As before, for each problem we generate 20 positive and 20 negative training examples, and 1000 positive and 1000 negative testing examples. Each list is randomly generated and has a maximum length of 50. We sample the list elements uniformly at random from the set $\{1, 2, \ldots, 100\}$. In addition to the hypothesis space bias for each problem – as specified by Appendix C.1 – we also provide the over-approximating rules $\mathcal{R}_{\mathcal{BK}}$.

We compare a normal version of `Popper` against `Popper`$[\mathcal{DL}]$. The sole distinction is that `Popper`$[\mathcal{DL}]$ is able to reason with over-approximating rules with $\mathcal{DL}$-properties, $\mathcal{R}_{\mathcal{BK}}$ in this case, while `Popper` just ignores them.

We measure the predictive accuracy, learning times, number of hypotheses tested (i.e. programs that reach the test stage), and number of times $\mathcal{DL}$-formulas are evaluated and how often that leads to a detected conflict. We enforce a timeout of 10 minutes per task. We repeat each experiment 20 times and report mean and standard error.

Results

Problem $+\mathcal{R}_{\mathcal{BK}}$	Number of programs			Learning time (sec)			Popper[$\mathcal{DL}_{\text{py}}$]			
	Popper	Popper[$\mathcal{DL}$]	ratio	Popper	Popper[$\mathcal{DL}$]	ratio	#progs	time (sec)	checks	conflicts
addhead	295 ± 0.0	172 ± 0.0	**0.58**	85 ± 29	68 ± 23	**0.80**	172 ± 0.0	75 ± 14	637k ± 42k	336 ± 7
dropk	1548 ± 28	669 ± 18	**0.43**	69 ± 3	28 ± 2	**0.40**	669 ± 18	44 ± 2	24k ± 2540	192 ± 9
droplast	692 ± 0.9	240 ± 0.3	**0.35**	52 ± 7	15 ± 0.8	**0.29**	240 ± 0.3	23 ± 0.8	86k ± 5212	421 ± 23
evens	1327 ± 0.8	1175 ± 0.6	**0.89**	156 ± 16	150 ± 8	**0.96**	1175 ± 0.6	191 ± 8	117k ± 5283	426 ± 10
finddup	1160 ± 109	765 ± 83	**0.66**	53 ± 13	40 ± 8	**0.75**	765 ± 82	58 ± 10	51k ± 11k	182 ± 20
last	1534 ± 57	1125 ± 40	**0.73**	117 ± 14	104 ± 10	**0.89**	1125 ± 40	141 ± 10	96k ± 10k	393 ± 26
len	973 ± 33	756 ± 17	**0.78**	71 ± 9	66 ± 5	**0.93**	757 ± 17	91 ± 5	118k ± 12k	266 ± 22
reverse	4316 ± 3	1945 ± 2	**0.45**	544 ± 19	182 ± 23	**0.33**	1945 ± 2	251 ± 7	224k ± 11k	529 ± 29
sorted	1080 ± 15	559 ± 26	**0.52**	105 ± 8	35 ± 3	**0.33**	559 ± 25	48 ± 3	84k ± 6237	1018 ± 101
sumlist	732 ± 0.9	576 ± 0.7	**0.79**	67 ± 6	59 ± 5	**0.89**	576 ± 0.7	82 ± 3	145k ± 32k	341 ± 15

Table 6.1: Results of list transformation experiment when $\mathcal{R}_{\mathcal{BK}}$ is supplied to both Popper and Popper[$\mathcal{DL}$]. We round times over 1 second to the nearest second. The error is standard error.

Table 6.1 contains the results of this first experiment. First, we note a reduction in the number of programs that get tested by Popper[$\mathcal{DL}$], ranging from needing to consider only 35% to 89% of Popper's hypotheses. We see that this translates to a reduction in learning times as well, going from needing as little as 29% of the time that Popper takes to about 96%. The (ratio of) number of programs and (ratio of) learning times across problems are strongly correlated: the Pearson correlation coefficient is 0.93.

The columns on the right in Table 6.1 report how often a set of $\mathcal{DL}$-formulas was checked, on average, for each problem by Popper[$\mathcal{DL}_{\text{py}}$]. We see that even though many sets of $\mathcal{DL}$-formulas get checked incrementally – as many as over 600,000 for the addhead task – this does not have a severe impact on learning times. While our Horn theory solver, i.e. the test stage, needs up to a millisecond to evaluate a hypothesis, the incremental evaluation of 600,000 checks by the $\mathcal{DL}_{\text{py}}$ solver of Clingo happen within about 6 seconds, meaning an average time of about 1/100th of a millisecond. The number of $\mathcal{DL}$ checks serves as a lower bound on the number of partial programs considered, as only a change in the guess of a partial program can give rise to a change in the enabled $\mathcal{DL}$ formulas[6].

The $\mathcal{DL}$-solvers of Clingo[$\mathcal{DL}$] and Clingo[$\mathcal{DL}_{\text{py}}$] do automatic minimization of a conflict, i.e. detect a subset of the enabled $\mathcal{DL}$-formulas which cause the inconsistency. We see that the number of detected conflicts is on the same order of magnitude as the number of programs that a normal Popper run would try.

In all, these results suggest that the answer to both question **Q1** and **Q2** is yes.

6.5.2 $\mathcal{DL}$-properties on BK and target predicate

For this next experiment, we assume that the user of the ILP system cannot only annotate their background knowledge with (sound) over-approximating rules, $\mathcal{R}_{\mathcal{BK}}$, but is also able to give properties of the predicates that they are trying to learn, a set of rules we will call $\mathcal{R}_f$.

[6]Note that because our approach to SMT-solving involves making many calls the SAT solver, a partial assignment (corresponding to a partial program) can be considered multiple times during the search.

Primitive recursive property

First of, we assert the believe that any program that is provably *not* primitive recursive is not a solution. We do so by a set of rules, $\mathcal{R}_{primrec}$, which contains over-approximating rules like the following, one for each target predicate of the 10 list transformation tasks:

$$\mathcal{R}_{primrec} = \left\{ \begin{array}{rcl} \text{sorted}(A) \leftarrow \text{sorted}(B) & \Rightarrow & \text{len}(A) - \text{len}(B) \leq 1, \\ \text{droplast}(A,B) \leftarrow \text{droplast}(C,D) & \Rightarrow & \text{len}(A) - \text{len}(C) \leq 1, \\ \text{dropk}(A,K,B) \leftarrow \text{dropk}(C,L,D) & \Rightarrow & \text{len}(A) - \text{len}(C) \leq 1 \vee K - L \leq 1, \\ & \ldots & \end{array} \right\}$$

Strictly speaking these rules need not be sound w.r.t. the relation we are trying to learn. Rather they function as a template on the hypotheses we believe can be solutions.

Over-approximating rules per target predicate

For each of the list transformation tasks we can give sound over-approximating rules. The following over-approximating $\mathcal{DL}$-properties are sound w.r.t. to the *intended* relation that we are wanting to learn a *Horn* program definition for. Note that the user does not need to know the exact intended relation in order to give a sound over-approximating rule. Instead it suffices that the specified properties are over-approximating w.r.t. each interpretation/the whole class of interpretations we are willing to considering as possible solutions.

We now list the over-approximating rules we use for each target predicate. As we allow recursive hypotheses for all the tasks, we provide an over-approximating rule both for the case where the target predicate occurs in the head of a clause as well as rules for when the target predicate is used by a body literal.

addhead For the task of learning a definition for $\text{addhead}(A,B)$ – where the initial element of A occurs as the first four elements of B – we have that the lengths of A and B are non-negative and that B is always three longer that A. The following set of over-approximating rules codifies this property, with the first rule applying when $\text{addhead}(A,B)$ occurs as a head literal in a clause and the second applying when $\text{addhead}(A,B)$ occurs as a body literal:

$$\mathcal{R}_{\text{addhead}} = \left\{ \begin{array}{rcl} \text{addhead}(A,B) & \Rightarrow & \text{len}(A) + 3 = \text{len}(B) \wedge \text{len}(A) \geq 0, \\ \leftarrow \text{addhead}(A,B) & \Rightarrow & \text{len}(A) + 3 = \text{len}(B) \wedge \text{len}(A) \geq 0 \end{array} \right\}$$

dropk In considering trying to learn a definition for $\text{dropk}(A,K,B)$, we are able to give the following (Linear Integer Arithmetic) over-approximating rule:

$$\text{dropk}(A,K,B) \Rightarrow \text{len}(A) \geq 0 \wedge \text{len}(B) \geq 0 \wedge \text{len}(A) - \text{len}(B) = K \wedge K \geq 0$$

That is, we know that A and B have (non-negative) lengths and we know these lengths have exactly K difference. In Difference Logic, we cannot have relations on three

variables like this. Instead we rewrite $\text{len}(A) - \text{len}(B) = K$ with $K \geq 0$ to obtain the following over-approximating rules with $\mathcal{DL}$ properties:

$$\mathcal{R}_{\text{dropk}} = \left\{ \begin{array}{rcl} \text{dropk}(A, K, B) & \Rightarrow & \text{len}(A) \geq 0 \wedge \text{len}(B) \geq 0 \wedge \text{len}(A) - \text{len}(B) \geq 0, \\ \leftarrow \text{dropk}(A, K, B) & \Rightarrow & \text{len}(A) \geq 0 \wedge \text{len}(B) \geq 0 \wedge \text{len}(A) - \text{len}(B) \geq 0 \end{array} \right\}$$

droplast The $\mathcal{DL}$-property for $\text{droplast}(A, B)$ has been discussed extensively throughout this chapter:

$$\mathcal{R}_{\text{droplast}} = \left\{ \begin{array}{rcl} \text{droplast}(A, B) & \Rightarrow & \text{len}(A) + 1 = \text{len}(B) \wedge \text{len}(A) \geq 1 \wedge \text{len}(B) \geq 0, \\ \leftarrow \text{droplast}(A, B) & \Rightarrow & \text{len}(A) + 1 = \text{len}(B) \wedge \text{len}(A) \geq 1 \wedge \text{len}(B) \geq 0 \end{array} \right\}$$

evens When learning a definition for $\text{evens}(A)$ we can assert that $\text{len}(A)$ is the non-negative length of A. Additionally we can assert that we want to reject any and all (partial) programs for which we can prove that the first element of A is not even. The following rules capture (a weaker version of) this using $\mathcal{DL}$-properties:

$$\mathcal{R}_{\text{evens}} = \left\{ \begin{array}{rcl} \text{evens}(A) \Rightarrow \text{len}(A) \geq 0, & & \leftarrow \text{evens}(A) \Rightarrow \text{len}(A) \geq 0, \\ \text{evens}(A) \leftarrow \text{head}(A, B) & \Rightarrow & B = 0 \vee B \geq 2, \\ \leftarrow \text{evens}(A), \text{head}(A, B) & \Rightarrow & B = 0 \vee B \geq 2, \\ \text{evens}(A) \leftarrow \text{cons}(B, A, C) & \Rightarrow & B = 0 \vee B \geq 2, \\ \leftarrow \text{evens}(A), \text{cons}(B, A, C) & \Rightarrow & B = 0 \vee B \geq 2 \end{array} \right\}$$

finddup For learning a definition for $\text{finddup}(A, B)$, all we assert is that $\text{len}(A)$ is a variable standing for the length of A, which must necessarily be at least two for there to be a duplicate element.

$$\mathcal{R}_{\text{finddup}} = \left\{ \begin{array}{rcl} \text{finddup}(A, B) & \Rightarrow & \text{len}(A) \geq 2, \\ \leftarrow \text{finddup}(A, B) & \Rightarrow & \text{len}(A) \geq 2 \end{array} \right\}$$

last For $\text{last}(A, B)$, we have a similarly simple property: $\text{len}(A)$ stands for the length of A and is at least one when there is a last element.

$$\mathcal{R}_{\text{last}} = \left\{ \begin{array}{rcl} \text{last}(A, B) & \Rightarrow & \text{len}(A) \geq 1, \\ \leftarrow \text{last}(A, B) & \Rightarrow & \text{len}(A) \geq 1 \end{array} \right\}$$

len When learning a definition for $\text{len}(A, B)$, we let $\text{len}(A)$ be a variable standing for the length of A, while B just reflects the integer B. One property we can express in $\mathcal{DL}$ is that when A is the empty list, then the length is zero. The only alternative case

we encode into the property is that when A is not the empty list, i.e. $\text{len}(A) \geq 1$, then the variable representing the length is also non-zero: $B \geq 1$ must then also hold for the $\mathcal{DL}$-variable that stands in for B.

$$\mathcal{R}_{\text{len}} = \left\{ \begin{array}{rcl} \text{len}(A,B) & \Rightarrow & (\text{len}(A) = 0 \wedge B = 0) \vee (\text{len}(A) \geq 1 \wedge B \geq 1), \\ \leftarrow \text{len}(A,B) & \Rightarrow & (\text{len}(A) = 0 \wedge B = 0) \vee (\text{len}(A) \geq 1 \wedge B \geq 1) \end{array} \right\}$$

reverse For the predicate $\text{reverse}(A,B)$, we posit that any (partial) program which has that the lengths of A and B are provably different cannot be a solution.

$$\mathcal{R}_{\text{reverse}} = \left\{ \begin{array}{rcl} \text{reverse}(A,B) & \Rightarrow & \text{len}(A) = \text{len}(B) \wedge \text{len}(A) \geq 0 \wedge \text{len}(B) \geq 0, \\ \leftarrow \text{reverse}(A,B) & \Rightarrow & \text{len}(A) = \text{len}(B) \wedge \text{len}(A) \geq 0 \wedge \text{len}(B) \geq 0 \end{array} \right\}$$

sorted When learning predicate $\text{sorted}(A)$, we claim that A has a non-negative length and that at least the first two elements occur in ascending order.

$$\mathcal{R}_{\text{sorted}} = \left\{ \begin{array}{rcl} \text{sorted}(A) \Rightarrow \text{len}(A) \geq 0, \quad \leftarrow \text{sorted}(A) & \Rightarrow & \text{len}(A) \geq 0, \\ \text{sorted}(A) \leftarrow \text{cons}(B,A,C), \text{head}(C,D) & \Rightarrow & B \leq D, \\ \leftarrow \text{sorted}(A), \text{cons}(B,A,C), \text{head}(C,D) & \Rightarrow & B \leq D, \\ \text{sorted}(A) \leftarrow \text{head}(A,B), \text{tail}(A,C), \text{head}(C,D) & \Rightarrow & B \leq D, \\ \leftarrow \text{sorted}(A), \text{head}(A,B), \text{tail}(A,C), \text{head}(C,D) & \Rightarrow & B \leq D \end{array} \right\}$$

sumlist For the property of $\text{sumlist}(A,B)$ we use that the list transformation experiments (already in Section 4.6.3 and Section 5.6.2) were set up to only use non-negative integers. For $\text{sumlist}(A,B)$, we assert that both A's length is non-negative and so is the result B, as we are summing non-negative numbers. Additionally, we assert that any partial program that provably has the result be smaller than the first element can be discarded.

$$\mathcal{R}_{\text{sumlist}} = \left\{ \begin{array}{rcl} \text{sumlist}(A,B) & \Rightarrow & \text{len}(A) \geq 0 \wedge B \geq 0, \\ \leftarrow \text{sumlist}(A,B) & \Rightarrow & \text{len}(A) \geq 0 \wedge B \geq 0, \\ \text{sumlist}(A,B) \leftarrow \text{head}(A,C) & \Rightarrow & C \leq B, \\ \leftarrow \text{sumlist}(A,B), \text{head}(A,C) & \Rightarrow & C \leq B, \\ \text{sumlist}(A,B) \leftarrow \text{cons}(C,D,A) & \Rightarrow & C \leq B, \\ \leftarrow \text{sumlist}(A,B), \text{cons}(C,D,A) & \Rightarrow & C \leq B \end{array} \right\}$$

As a convenience, we collect all the above over-approximating rules together:

$$\mathcal{R}_f = \left\{ \begin{array}{l} \mathcal{R}_{primrec} \cup \\ \mathcal{R}_{\text{addhead}} \cup \mathcal{R}_{\text{dropk}} \cup \mathcal{R}_{\text{droplast}} \cup \mathcal{R}_{\text{evens}} \cup \mathcal{R}_{\text{finddup}} \cup \\ \mathcal{R}_{\text{last}} \cup \mathcal{R}_{\text{len}} \cup \mathcal{R}_{\text{reverse}} \cup \mathcal{R}_{\text{sorted}} \cup \mathcal{R}_{\text{sumlist}} \end{array} \right.$$

Problem $+\mathcal{R}_{\mathcal{BK}} \wedge \mathcal{R}_f$	Number of programs			Learning time (sec)			Popper[$\mathcal{DL}_{\text{py}}$]			
	Popper	Popper[$\mathcal{DL}$]	ratio	Popper	Popper[$\mathcal{DL}$]	ratio	#progs	time (sec)	checks	conflicts
addhead	295 ± 0.0	3 ± 0.0	**0.01**	85 ± 29	2 ± 0.0	**0.03**	3 ± 0.0	4 ± 0.1	61k ± 2367	290 ± 9
dropk	1548 ± 28	584 ± 10	**0.38**	69 ± 3	21 ± 1	**0.30**	584 ± 10	36 ± 1	28k ± 1746	315 ± 18
droplast	692 ± 0.9	46 ± 0.0	**0.07**	52 ± 7	2 ± 0.1	**0.03**	46 ± 0.0	3 ± 0.1	16k ± 1370	169 ± 8
evens	1327 ± 0.8	1126 ± 0.4	**0.85**	156 ± 16	151 ± 14	**0.97**	1126 ± 0.4	186 ± 10	121k ± 4728	477 ± 28
finddup	1160 ± 109	728 ± 81	**0.63**	53 ± 13	38 ± 7	**0.72**	727 ± 83	55 ± 11	50k ± 10k	164 ± 17
last	1534 ± 57	1079 ± 40	**0.70**	117 ± 14	97 ± 8	**0.83**	1079 ± 40	134 ± 10	87k ± 9773	364 ± 29
len	973 ± 33	535 ± 15	**0.55**	71 ± 9	39 ± 4	**0.56**	536 ± 15	55 ± 5	96k ± 19k	230 ± 20
reverse	4316 ± 3	127 ± 0.4	**0.03**	544 ± 19	5 ± 0.1	**0.01**	127 ± 0.4	8 ± 0.2	38k ± 3067	206 ± 10
sorted	1080 ± 15	543 ± 25	**0.50**	105 ± 8	35 ± 3	**0.33**	544 ± 26	47 ± 3	82k ± 7319	1076 ± 111
sumlist	732 ± 0.9	411 ± 0.3	**0.56**	67 ± 6	32 ± 4	**0.47**	411 ± 0.3	44 ± 1	99k ± 10k	331 ± 10

Table 6.2: Results of list transformation experiment when $\mathcal{R}_{\mathcal{BK}}$ and $\mathcal{R}_f$ are supplied to both Popper and Popper[$\mathcal{DL}$]. We round times over 1 second to the nearest second. The error is standard error.

Setup

The setup is the exact same as in Section 6.5.2, except that in addition to supplying $\mathcal{R}_{\mathcal{BK}}$ for Popper[$\mathcal{DL}$] to reason about we also supply $\mathcal{R}_f$.

Results

Table 6.2 contains the results of this experiment. Additionally, Table 6.4 lists the learning times and number of tested programs for Popper and Popper[$\mathcal{DL}$] when they are not supplied with over-approximating rules (which is the same as just normal Popper) versus when both systems are supplied with 1) just $\mathcal{R}_{\mathcal{BK}}$ and 2) when supplied with $\mathcal{R}_{\mathcal{BK}} \wedge \mathcal{R}_f$.

For each task we note a decrease in hypotheses tested by the test stage ("Number of programs" in the table), also relative to when just $\mathcal{R}_{\mathcal{BK}}$ was made available to Popper[$\mathcal{DL}$]. The effect of the additional over-approximating rules $\mathcal{R}_f$ varies per task. The tasks where we can state a property where two lengths of lists have a fixed difference – i.e. addhead, droplast, reverse, are the ones for which adding $\mathcal{R}_f$ to $\mathcal{R}_{\mathcal{BK}}$ is very effective. In case of addhead, the total number of programs considered drops from 295 ($\mathcal{R} = \emptyset$) to 172 ($\mathcal{R}_{\mathcal{BK}}$) to just 3 ($\mathcal{R}_{\mathcal{BK}} \wedge \mathcal{R}_f$). From Table 6.4, we can see that even the first tested addhead hypothesis is a solution. Clearly within this hypothesis space there are very few programs (at most the size of a solution) which do not violate the property that their output list is exactly three elements longer than their input list. For droplast and reverse something similar holds: there are very few hypotheses which do not provably violate their supplied length properties.

For the binary predicates finddup and last, we only see minor improvements in terms of pruning with $\mathcal{R}_f$. This can be attributed to these predicates having the simplest over-approximating properties, i.e. they only assert that the length of a single list is of a non-trivial length. The other two tasks that do not benefit much from supplying $\mathcal{R}_f$ to Popper[$\mathcal{DL}$] are evens and sorted, both unary predicates. As these two predicates are unary, e.g. sorted(A), the property we have for A is simply that its length, A, is non-negative. The other over-approximating rules in $\mathcal{R}_{\text{sorted}}$ require specific body literals to occur and hence are only enabled for quite a small proportion of the considered hypotheses.

Again we see that the (ratio of) number of tested hypotheses correlates strongly with the learning time (Pearson's $r \approx 0.97$). For the aforementioned four tasks, the inclusion of $\mathcal{R}_f$ does little for additional pruning, but the additional reasoning incurred by the

Problem $+\mathcal{R}_{\mathcal{BK}} \wedge \mathcal{R}_f \wedge \mathcal{R}_{e^+}$	Number of programs			Learning time (sec)			Popper[$\mathcal{DL}_{\text{py}}$]			
	Popper	Popper[$\mathcal{DL}$]	ratio	Popper	Popper[$\mathcal{DL}$]	ratio	#progs	time (sec)	checks	conflicts
addhead	295 ± 0.0	3 ± 0.0	**0.01**	85 ± 29	5 ± 0.4	**0.06**	3 ± 0.0	5 ± 0.2	56k ± 1587	228 ± 10
dropk	1548 ± 28	126 ± 7	**0.08**	69 ± 3	4 ± 0.4	**0.06**	126 ± 7	7 ± 0.6	7743 ± 345	126 ± 7
droplast	692 ± 0.9	15 ± 0.0	**0.02**	52 ± 7	1 ± 0.1	**0.02**	15 ± 0.0	2 ± 0.0	13k ± 655	183 ± 5
evens	1327 ± 0.8	935 ± 0.8	**0.70**	156 ± 16	130 ± 26	**0.83**	935 ± 0.8	168 ± 23	133k ± 13k	682 ± 33
finddup	1160 ± 109	548 ± 123	**0.47**	53 ± 13	22 ± 8	**0.41**	549 ± 124	34 ± 13	37k ± 10k	170 ± 24
last	1534 ± 57	140 ± 40	**0.09**	117 ± 14	4 ± 2	**0.04**	139 ± 43	7 ± 2	20k ± 4399	184 ± 37
len	973 ± 33	327 ± 23	**0.34**	71 ± 9	16 ± 2	**0.23**	327 ± 23	26 ± 3	55k ± 12k	191 ± 23
reverse	4316 ± 3	112 ± 2	**0.03**	544 ± 19	4 ± 0.2	**0.01**	112 ± 2	7 ± 0.2	34k ± 2608	169 ± 10
sorted	1080 ± 15	494 ± 14	**0.46**	105 ± 8	32 ± 2	**0.30**	494 ± 13	44 ± 2	85k ± 6699	1059 ± 121
sumlist	732 ± 0.9	182 ± 0.3	**0.25**	67 ± 6	9 ± 0.5	**0.13**	182 ± 0.3	16 ± 0.5	49k ± 2862	278 ± 15

Table 6.3: Results of list transformation experiment when $\mathcal{R}_{\mathcal{BK}}$, $\mathcal{R}_f$ and $\mathcal{R}_{e^+}$ are supplied to both Popper and Popper[$\mathcal{DL}$]. We round times over 1 second to the nearest second. The error is standard error.

additional $\mathcal{DL}$ formulas has not hurt learning times versus Popper[$\mathcal{DL}$] with just $\mathcal{R}_{\mathcal{BK}}$. Indeed, while for both evens and sorted a couple thousand more $\mathcal{DL}$ formula checks are performed, the learning time remains close to unchanged.

In all, these results suggest that the answer to both question **Q1** and **Q2** is yes.

6.5.3 $\mathcal{DL}$-properties on BK, target predicate and an example

For the final experiment, we use the observation that over-approximations for the target predicate must hold of the positive examples – see Section 6.3.1. This means we can instantiate the over-approximation with the values for one of the examples.

For each task we select a random positive example e^+ and translate it to a $\mathcal{DL}$-property capturing the actual value of integers and the lengths of lists that occur in the example. This gives us the following set of rules:

$$\mathcal{R}_{e^+} = \left\{ \begin{array}{rcl} \text{droplast}(A,B) & \Rightarrow & \text{len}(A) = 20 \wedge \text{len}(B) = 19, \\ \text{dropk}(A,K,B) & \Rightarrow & \text{len}(A) = 20 \wedge K = 6 \wedge \text{len}(B) = 14, \\ \text{sorted}(A) & \Rightarrow & \text{len}(A) = 36, \\ & \ldots & \end{array} \right\}$$

In general it is not sound to trigger these rules for each clause of a hypothesis as then we would be asserting that the invocations of a rule that happen through recursion also have the same values. Instead we use some helper code to say that the Clingo solver can make a choice as to which clause of a hypothesis a rule from $\mathcal{R}_{e^+}$ applies to. This corresponds to guessing which clause is resolved first in a successful SLD-proof of the example.

Setup

The setup is the exact same as in Section 6.5.2, except that in addition to supplying $\mathcal{R}_{\mathcal{BK}}$ for Popper[$\mathcal{DL}$] to reason about we also supply $\mathcal{R}_f$ and $\mathcal{R}_{e^+}$ (with the code ensuring that an $\mathcal{R}_{e^+}$ rule is triggered just once).

Results

Table 6.3 contains the results of this experiment. Additionally, Table 6.4 lists learning times and number of tested programs for when `Popper` and `Popper`$[\mathcal{DL}]$ are supplied with no rules, or just $\mathcal{R}_{\mathcal{BK}}$, or $\mathcal{R}_{\mathcal{BK}} \wedge \mathcal{R}_f$, or $\mathcal{R}_{\mathcal{BK}} \wedge \mathcal{R}_f \wedge \mathcal{R}_{e^+}$.

For each task we see a reduction in terms of number of tested hypotheses, with respect to $\mathcal{R}_{\mathcal{BK}} \wedge \mathcal{R}_f$. For 6 out of 10 tasks, the improvement is substantial, all the way to `Popper`$[\mathcal{DL}]$ with $\mathcal{R}_{\mathcal{BK}} \wedge \mathcal{R}_f \wedge \mathcal{R}_{e^+}$ needing as little as 13% of the programs that `Popper`$[\mathcal{DL}]$ with $\mathcal{R}_{\mathcal{BK}} \wedge \mathcal{R}_f$ requires, for last. In the case of reverse and addhead, the improvement is small or not there at all. This can be ascribed to the over-approximating rules in $\mathcal{R}_f$ for these predicates already being very effective at pruning hypotheses.

The tasks with least improvement over supplying just $\mathcal{R}_{\mathcal{BK}}$ and $\mathcal{R}_f$ are sorted (91%) and evens (83%). Again, both are monadic tasks which means the $\mathcal{DL}$ property for their examples is less of a constraint as those for binary of ternary target predicates. Most likely knowing that the argument is non-empty just eliminates a body literal like empty(A). We note that the (average) number of $\mathcal{DL}$-formula checks again went up versus the smaller rule sets, and that this again resulted in an improvement in terms of pruning and learning times.

Also these results bear out that the learning time strongly correlates with the number of tested programs: Pearson's $r \approx 0.94$. Five of the tasks now require less than 10% of the learning time versus `Popper` without over-approximating rules.

We conclude these experiments by looking at Table 6.4. Here we see that, essentially, each time we add additional (sound) over-approximating rules – first by adding $\mathcal{R}_{\mathcal{BK}}$, then adding $\mathcal{R}_f$ to that, and finally adding $\mathcal{R}_{e^+}$ – we monotonically reduce the number of programs that still require testing and also reduce learning times. In all, these results suggest that the answer to both question **Q1** and **Q2** is yes.

On methodology

Finally, a note on methodology. Tables 6.1, 6.2 and 6.3 reported the tested programs and time it took to find *all* optimal solutions, i.e. exhausting the search space at the optimal size. The alternative would have been to report these numbers for finding the *first* solution. Table 6.4 does both.

When looking at Table 6.4, we see that variance (i.e. standard error) for finding all optimal solutions is typically either substantially lower or at a similar level to the variance for finding the first optimal solution. For example, for reverse/2, normal `Popper` has a standard error of 505 tested based on 20 runs for finding the first solution but only a standard error of 3 when finding all optimal solutions. For a task like last, the variance in number of programs and learning time is very similar whether searching for the first or all optimal solutions. Only for the task finddup is the variance for finding all solutions notably higher than finding the first solution.

		Number of programs				Learning time (sec)			
Problem	**#sol**	$\emptyset$	$\mathcal{R}_{\mathcal{BK}}$	$\mathcal{R}_{\mathcal{BK}} \wedge \mathcal{R}_f$	$\mathcal{R}_{\mathcal{BK}} \wedge \mathcal{R}_f \wedge \mathcal{R}_{e^+}$	$\emptyset$	$\mathcal{R}_{\mathcal{BK}}$	$\mathcal{R}_{\mathcal{BK}} \wedge \mathcal{R}_f$	$\mathcal{R}_{\mathcal{BK}} \wedge \mathcal{R}_f \wedge \mathcal{R}_{e^+}$
addhead	**1st**	90 ± 37	73 ± 30	1.0 ± 0.0	1.0 ± 0.0	12 ± 6	16 ± 11	1 ± 0.0	1 ± 0.0
	all	295 ± 0.0	172 ± 0.0	3 ± 0.0	3 ± 0.0	85 ± 29	68 ± 23	2 ± 0.0	5 ± 0.4
dropk	**1st**	1348 ± 108	499 ± 42	485 ± 32	27 ± 10	52 ± 9	16 ± 3	14 ± 2	0.7 ± 0.2
	all	1548 ± 28	669 ± 18	584 ± 10	126 ± 7	69 ± 3	28 ± 2	21 ± 1	4 ± 0.4
droplast	**1st**	378 ± 99	145 ± 17	43 ± 1	11 ± 0.7	18 ± 8	7 ± 2	1 ± 0.0	0.8 ± 0.0
	all	692 ± 0.9	240 ± 0.3	46 ± 0.0	15 ± 0.0	52 ± 7	15 ± 0.8	2 ± 0.1	1 ± 0.1
evens	**1st**	1300 ± 10	1168 ± 4	1119 ± 3	931 ± 3	143 ± 16	145 ± 9	145 ± 14	126 ± 27
	all	1327 ± 0.8	1175 ± 0.6	1126 ± 0.4	935 ± 0.8	156 ± 16	150 ± 8	151 ± 14	130 ± 26
finddup	**1st**	1006 ± 38	664 ± 46	628 ± 31	419 ± 41	40 ± 3	31 ± 4	29 ± 3	14 ± 2
	all	1160 ± 109	765 ± 83	728 ± 81	548 ± 123	53 ± 13	40 ± 8	38 ± 7	22 ± 8
last	**1st**	1466 ± 62	1056 ± 36	1041 ± 40	74 ± 38	101 ± 12	88 ± 9	88 ± 8	2 ± 1
	all	1534 ± 57	1125 ± 40	1079 ± 40	140 ± 40	117 ± 14	104 ± 10	97 ± 8	4 ± 2
len	**1st**	806 ± 45	648 ± 31	452 ± 22	233 ± 23	48 ± 6	49 ± 5	30 ± 3	10 ± 2
	all	973 ± 33	756 ± 17	535 ± 15	327 ± 23	71 ± 9	66 ± 5	39 ± 4	16 ± 2
reverse	**1st**	1338 ± 505	629 ± 136	111 ± 9	88 ± 9	86 ± 51	37 ± 10	3 ± 0.7	3 ± 0.6
	all	4316 ± 3	1945 ± 2	127 ± 0.4	112 ± 2	544 ± 19	182 ± 23	5 ± 0.1	4 ± 0.2
sorted	**1st**	712 ± 155	457 ± 84	454 ± 67	393 ± 77	48 ± 18	24 ± 6	24 ± 6	21 ± 6
	all	1080 ± 15	559 ± 26	543 ± 25	494 ± 14	105 ± 8	35 ± 3	35 ± 3	32 ± 2
sumlist	**1st**	591 ± 43	464 ± 36	347 ± 9	116 ± 8	36 ± 8	36 ± 6	23 ± 4	5 ± 0.5
	all	732 ± 0.9	576 ± 0.7	411 ± 0.3	182 ± 0.3	67 ± 6	59 ± 5	32 ± 4	9 ± 0.5

Table 6.4: Results of list transformation experiment when the following rules are supplied to both `Popper` and `Popper`$[\mathcal{DL}]$: $\emptyset$, i.e. no rules, just $\mathcal{R}_{\mathcal{BK}}$, $\mathcal{R}_{\mathcal{BK}} \wedge \mathcal{R}_f$, or $\mathcal{R}_{\mathcal{BK}} \wedge \mathcal{R}_f \wedge \mathcal{R}_{e^+}$. Table includes number of hypotheses tested and learning times for both finding the first (optimal) solution as well as all optimal solutions. We round times over 1 second to the nearest second. The error is standard error.

Chapter 7

Conclusion

To conclude this dissertation, we revisit the main methods and contributions of the preceding chapters and consider how they have substantiated the three claims that form our thesis. Following on, we examine the limitations of our work and list directions for future work.

7.1 In summary: our three claims revisited

This dissertation has introduced the Learning From Failures approach to Inductive Logic Programming. In setting up the LFF framework, we followed the meta-level approach to ILP: we chose to represent the hypothesis space – consisting of entire programs and not just clauses – by a constraint program. The central idea is that we can *refine* the (representation of the) hypothesis space by adding constraints, in particular constraints that prune assignments corresponding to non-solutions. The main mechanism for doing so is to search for an assignment not pruned by the constraints, which corresponds to a currently viable (partial) hypothesis, and check if this program is (or could be extended to) a solution or not. Whenever a thus selected (partial) program fails, we infer that a whole class of related programs are not solutions either. This information we encode into a constraint that prunes assignments that correspond to these programs, thereby reducing the set of hypotheses that are still considered viable.

Claim C1: the SMT-perspective on ILP

Our first claim is that the above guess-and-check methodology is essentially that of Satisfiability Modulo Theories. In particular, having a SAT-solver search for assignments satisfying propositional constraints, with each such assignment triggering first-order formulas which are then checked by another solver is exactly how SMT-solvers work. We demonstrated a way of encoding our constraint-accumulation approach to ILP as an SMT-problem. By carefully interpreting the first-order *Horn*-formulas selected by propositional assignments we get that a theory solver can decide whether the selected formulas, now interpreted as a logic program, satisfy the examples or not. In particular, we showed that the Background Knowledge fixes the interpretation of the (non-hypothesis defined) first-order predicate symbols, effectively making each choice of BK into a separate SMT-background theory. When this *Horn*-theory solver determines there is a *conflict* – between the selected hypothesis and the examples – this leads to learning a constraint that prunes

non-solutions. We identified this SMT-loop as the backbone of Conflict-Driven Inductive Logic Programming.

Claim C2: flexible and performant three-stage implementation

Our second claim is that our generate-test-and-constrain loop provides a flexible framework for addressing ILP problems. We showed that our approach accommodates different hypothesis space encodings, such as synthesis-by-rule-selection, but also our more granular synthesis-by-literal-selection. We argued that this fine-grained encoding allows for constraints on the hypothesis space to be expressed very naturally. In particular, it makes it possible to take a (failed) hypothesis and derive a first-order constraint (that mirrors the hypothesis' syntax) which prunes many related programs based on θ-subsumption. We essentially made the solver used by the test stage into a parameter, allowing us to use a Prolog interpreter for our implementation, `Popper`. In experiments we show that `Popper` can outperform existing ILP systems, especially when it comes to dealing with large hypothesis spaces and large domains. The flexibility of the three-stage loop is further demonstrated by its amenability to extensions.

Claim C3: SMT-based extensions that reduce hypothesis space exploration

Our final claim is that our SMT-perspective facilitates applying techniques from other communities to ILP. Our inclusion of the constrain stage in the (otherwise two-stage) SMT-loop mirrors the technique of dynamic symmetry pruning explored by the constraint solving community. Our fifth and sixth chapters each took another such correspondence and developed it into a general framework applicable to ILP. In both cases, we showed how these frameworks help us to infer more and more effective constraints, meaning that in order to exhaust the same hypothesis space fewer programs need to be tested, which, in our experiments, translated into reduced learning times.

Failure explanation In SMT-solving, a theory solver can choose to expend a bit more effort to identify which of the first-order formulas its been given are causing a conflict, thereby making it possible to derive more effective conflict clauses (i.e. constraints). By the LFF-as-SMT-perspective, we took note of how this corresponds to learning to identify which sub-programs of a hypothesis cause a failure. In terms of the three-stage loop, this led us to incorporating failure explanation into the test stage.

Over-approximating properties Inspired by how SMT-solvers are used to synthesize functional programs which provably possess properties, we developed a framework that allows us to determine which logic programs cannot be (optimal) solutions by reasoning about their *over-approximating* properties. By having another theory solver perform checks on whether (partially) guessed hypotheses are already disqualified based on their properties, we can elide the relatively expensive check of evaluating these hypotheses on examples.

7.2 Limitations & future work

We conclude by touching on multiple significant features and themes not addressed by this dissertation. Some of these limitations have already been addressed to a degree in subsequent work. For others we list new directions to pursue.

7.2.1 Missing ILP features

Predicate invention The version of `Popper` presented in this dissertation does not address predicate invention, i.e. learning programs which involve definitions for auxiliary predicates not occurring in the Background Knowledge or in the examples. In a separate paper [36], we presented an extension of `Popper` that addresses learning this kind of hypothesis. The primary mechanism is to make additional predicate declarations (dynamically) available, such as `head_pred(inv`N`,`M`)` for different N and M. By automatically also inserting the corresponding `body_pred(inv`N`,`M`)` into the declaration bias, we get a hypothesis space which involves programs with invented predicates (in this case, `inv`N). For totally incomplete programs, careful reasoning about redundancy becomes crucial as no hypothesis with invented predicates is *separable*, see Section 4.5.5. In matter of fact, our method for determining which programs are redundant by reasoning about how their clauses dependent on one another was first presented in [36]. We also note here that our techniques for failure explanation and over-approximating properties are applicable without modification to the setting with invented predicates.

Non-observational predicate learning Unlike some ILP systems [118, 78, 99], `Popper` does not support non-observational predicate learning [118], where examples of the target predicates are not directly given (i.e. predicates that occur in the Background Knowledge need to be extended). As far as we can tell, our subsumption-based constraints are still valid in this setting – given that the hypotheses and Background Knowledge are Horn programs – and hence `Popper` should be extendable to this type of learning.

Noise There are a number of ILP systems [95, 135, 118, 137, 16, 150, 140, 1, 49] that can learn programs in the presence of noise, i.e. when there are misclassified examples, typically with the aim of finding the hypothesis with the best fit. In this dissertation, we have restricted ourselves to the noiseless setting. With the SMT-perspective in hand, one approach would be to encode whether examples are covered or not into *weak* constraints [60] and find a hypothesis which incurs the least cost. This effectively turns the turns the SMT-perspective on LFF into a MaxSMT-perspective [124]. A slightly different approach to consider is that of the ILASP systems [97, 92]: we could carefully infer constraints that only prune those programs that are guaranteed to have worse score than the current best-fit program. It would appear that using failure explanation to identify other smaller programs extends to the noisy setting: finding sub-programs of a failed hypothesis with a similarly bad fit should lead to being able to infer stronger constraints.

7.2.2 Logic fragment of hypotheses

In this dissertation, our hypotheses are definite programs. While numerous systems focus on the Datalog fragment – see Section 2.1.5 – we additionally allow arbitrary function

symbols to occur in our programs. We mainly employ this to deal with structured data, such as when learning list transformations. While this makes for a computationally expressive fragment, other fragments are able to express complex concepts using shorter programs.

There is already an extension of `Popper` to allow for learning higher-order programs [134]. Similarly, there is an extension [22] to allow for (a certain form of) normal programs, i.e. definite programs but with negation (as failure) allowed in bodies of clauses.

Learning ASP-programs In principle, our three-stage loop approach is a recipe that also applies to learning Answer Set Programming programs[1], with the SMT-perspective naturally extending to this setting. In Section 4.2.4, we fix the interpretation of a selected hypothesis to be its least model. The natural extension of this is to make the set of intended interpretations of a hypothesis correspond to all its *answer sets* (which in the special case of definite programs gives us back our current definition). By now insisting that one of these interpretations needs to satisfy the examples for there to be an overall model, we have naturally incorporated *brave induction* [143] into our framework (correspondingly, if we take the SyGuS-perspective and insist all answers sets must model the examples we appropriately capture the *cautious induction* framework, c.f. [93]).

The main issue when it comes to inducing ASP programs is that `Popper`'s effectiveness largely depends on how much pruning the learned constraints can achieve. When our programs keep to standard first-order logic semantics, we can use θ-subsumption [129, 109] to infer that large sections of the hypothesis space do not contain solutions. This does *not* hold when we use ASP's stable model semantics, e.g. for programs involving negation (as failure). It is not clear if there are other relations on ASP-programs that allow us to infer constraints from individual failed hypotheses that are (as) effective at pruning the hypothesis space.

Failure explanation As our failure explanation algorithm (Section 5.3) is based on SLD-resolution, it effectively applies just to definite programs. Requiring that our programs are executable by SLD-resolution also leads to imposing a (left-to-right) literal ordering on clauses, which limits which problematic sub-programs we can find. Our algorithm also would require non-trivial modification to apply to the `Popper` extensions that allow learning higher-order programs [134] and learning programs with (a form of) stratified negation [22].

However, the general notion of using failure explanation to identify smaller programs (from which we can derive more effective constraints) can be implemented in myriad ways. The observation that it is often the small sub-programs that themselves are not well-formed programs that achieve the most effective additional pruning is likely to carry over to other logic fragments. The ProSynth [139] and ASPSynth [13] systems use *query provenance* to identify which rules of a Datalog program cause examples to not be covered. In case Learning From Failures would be applied to learning ASP-programs, we could use the work on *justifications* [131] to help identify which ASP sub-programs are responsible for a failure.

[1] In fact, we have experimented with swapping out the test stage with an ASP-solver [58] and had $\mathcal{BK}$ be an ASP-program with our hypotheses being Datalog programs.

Over-approximating properties As in the setting of functional program synthesis, reasoning with (over-approximating) properties also works well when learning higher-order programs. Sticking to the domain of lists, here are two examples of properties that could be employed for very effective pruning of the hypothesis space:

$$\text{map}(F, A, B) \Rightarrow \text{len}(A) = \text{len}(B) \wedge \text{len}(A) \geq 0$$

$$\text{filter}(F, A, B) \Rightarrow \text{len}(A) \geq \text{len}(B) \wedge \text{len}(B) \geq 0$$

It is not clear whether or not our framework extends to hypotheses that involve negated body literals.

7.2.3 Encoding the hypothesis space

We showed that the LFF framework can accommodate different encodings of the hypothesis space, in particular we demonstrated both the popular synthesis-as-rule-selection encoding – with a propositional variable per candidate clause, see Example 4.2.21 – and our own synthesis-as-literal-selection encoding, as used by `Popper`, see Section 4.5.3. Within our framework, other encodings can be explored too. In particular, both the INSPIRE [144] and Zaatar [2] systems make use of an encoding where the predicate symbol of a literal is selected separately from the literal's arguments. As our variable-per-literal encoding managed to capture a lot of the structure of the hypothesis space – making it possible to naturally encode θ-subsumption-based pruning into constraints – an investigation of the pros and cons of a yet more granular encoding is in order.

Ideally we would have an encoding where each full assignment corresponds not just to a syntactically unique program, but a *semantically* unique program. It is not clear for which hypotheses spaces this is possible and what the best encoding strategy would be.

7.2.4 More effective constraints

Hypothesis constraints are central to our idea. `Popper` uses both pre-defined and learned constraints to improve performance. Whereas `Popper`'s pre-defined constraints are helpful in exercising more control over the hypothesis space, e.g. to prune programs from the hypothesis space which cannot possibly work, LFF's learned constraints are crucial to our approach. While our specialisation and generalisation constraints straightforwardly derive from the definition of θ-subsumption, our development of constraints that prune hypotheses with redundancy was more subtle. Hence it is not clear if more kinds of constraints could be deduced from observing a hypothesis fail (on the examples or otherwise).

Types Like many ILP systems [118, 16, 150, 95, 49], Popper supports simple types to prune the hypothesis space. As shown previously [114], polymorphic type checking for programs over structured data is very effective at pruning malformed programs from the hypothesis space. For instance, polymorphic types would allow us to distinguish between using a predicate on a list of integers and on a list of characters. We have yet to work out the best way of encoding polymorphic type checking into constraints.

Theory While refinement operators for clauses [145, 137, 122] and theories, i.e. programs, [122, 109, 8] are well-studied, we have only established basic properties of our constraints. Though we show in Section 4.3 that θ-subsumption-based constraints are sound, we know Popper's constraints are not complete, i.e. they do not prune *every* program related by θ-subsumption to a failed hypothesis. In view of existing results on the complexity of ILP problems [66] as well as the NP-completeness of checking subsumption, it would appear striving for complete constraints might be in vain. It might be fruitful to consider language biases which only allow programs which have a certain structure (e.g. such as having bounded (hyper-)treewidth [65]), as when there is more structure to exploit problems such as subsumption checking can become tractable.

Failure explanation While we showed that learning sub-programs leads to more sound pruning (Theorem 5.2.8), we presented no theory regarding the expected effectiveness of sub-program-based failure explanation. As seen in the experiment of Section 5.6.2, it appears that finding many small sub-programs is key to achieving significant pruning. It should be possible to quantify the (theoretical) effectiveness of sub-program based pruning, e.g. with respect to the size of a sub-program and hypothesis space parameters such as the number of predicates. In general, future work should try to determine characteristics of problems that allow or preclude effective pruning based on failure explanation. The same holds for reasoning about the (expected) effectiveness of enabling reasoning with over-approximating properties.

7.2.5 Further embracing SMT-solving

For `Popper`, we emulate SMT-solving by using multi-shot solving to interleave the search for a satisfying (propositional) assignment with a check for whether the corresponding hypothesis is a solution or not by the test stage, see Section 4.5. Unlike existing SMT-fragments [11], our first-order formulas are Horn programs which we interpret according to their least model semantics[2].

Propagator-based solving In terms of SMT-solving, `Popper` invokes a check by a theory solver when the SAT-solver has constructed *full/total* propositional assignments and each check of the selected first-order formulas runs from scratch. The modern approach to SMT-solving – using the *propagator* interface [125, 70] – allows for tighter integration of the SAT-solver and theory solvers. As described in detail in Section 3.3.4, a SAT-solver can already derive which first-order formulas must hold from *partial* assignments. Many theory solvers hence have the SAT-solver derive and communicate formulas before a full assignment has been found and evaluate if these sets of formulas are already unsatisfiable. Crucially, a theory solver can also keep state between subsequent checks, which is often worthwhile as most of the first-order formulas will remain unchanged. The MonoSynth [13] system introduces a Horn theory solver which can incrementally evaluate Datalog programs as clauses get added and removed, relying on that adding a clause to a Horn program can only add consequences. For `Popper`, we could make the test stage into a proper incremental theory solver as well. Ideally this solver would be incremental with

[2]As we also allow arbitrary function symbols in these programs, in general, this fragment is not decidable. Theory solvers in SMT tend to support just decidable fragments.

respect to the full subsumption lattice, which would also allow (a degree of) incremental evaluation in case just literals are added/removed from clauses.

Conflict minimization Evaluation of partially built-up programs is closely related to failure explanation, as both help identify failing sub-programs. However, incremental evaluation alone does not guarantee that minimal failing sub-programs are found. Hence, ideally we would have an incremental *Horn*-theory solver that maintains state between checks and upon a conflict being encountered also identifies which formulas selected by the partial assignment were not relevant for the failure to occur. This would mirror how the Difference Logic solver of Clingo[$\mathcal{DL}$] works [74].

Variations on the constrain stage As explained in Section 3.3.4, normal theory solvers just return a subset of the formulas they were given as the cause of a conflict. The SAT-solver interprets these formulas as (propositional) conflict clauses, meaning that a re-occurrence of these particular formulas can now be pruned by the SAT-solver without needing to query the theory solver. For `Popper` we go a step further and use θ-subsumption to determine many other formulas, i.e. programs, which must also fail and hence whose corresponding assignments should be pruned. While in the experiments of Chapter 4 we established that the introduction of learned constraints to a version of `Popper` that enumerates programs is effective, we did not evaluate the importance of the variable renaming aspect of the learned constraints. That is, without this symmetry breaking aspect [63], our constraints prune in essentially the same manner as normal conflict clauses. Hence we have yet to properly evaluate the importance of including the constrain stage in the loop[3], see Figure 7.1.

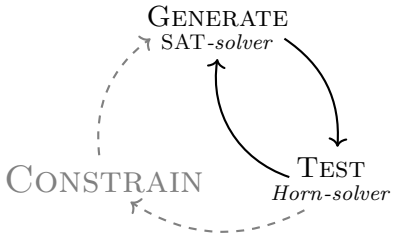


Figure 7.1: LFF's generate and test loop directly corresponds to how SMT-solvers work. The importance of including the constrain stage in this loop is yet to be fully evaluated.

Lazy grounding In `Popper`, the constrain stage first derives a first-order constraint from a failed hypothesis and then immediately fully grounds this constraint such that the resulting propositional clauses can be injected into the SAT-solver. This grounding can be quite expensive, see Table 4.7 in Section 4.6.3. On top of that, it is likely that many of these ground clauses are extraneous, e.g. because they overlap with/are subsumed by previously derived constraints. As SMT-solvers [44, 10] already incorporate techniques for instantiating first-order formulas – in particular, E-matching [43] – it might be possible to have the constrain stage derive first-order constraints and to then directly hand them off

[3]Preliminary experiments indicate that the variable renaming aspect of the learned constraints is very important for effective pruning.

to a SMT-solver. The SMT-solver would then *lazily* ground these formulas if and when it determines that an instantiation is relevant to the current assignment.

7.2.6 Inductive Logic Programming Modulo Theories

Our test stage acts as a theory solver that checks if a set of Horn formulas satisfies given examples. We hence have that determining whether hypotheses fail by testing them on examples scales with the number of examples: it could always be that the last example determines whether the program failed or not[4]. It follows that checks by the test stage can get arbitrarily expensive. In contrast, our approach to checking over-approximating properties is independent of the number of examples.

More theories and more domains In Chapter 6, we viewed the generate stage's SAT-solver as an SMT-solver in its own right, meaning that (theory) solvers other than the test stage can be used to evaluate guesses for hypotheses and reject them. As our implementation of `Popper` relies on the Clingo solver for the generate stage, we are limited to the theory solvers it supports [70, 9]. This made us focus on the Difference Logic fragment [70] and properties that can be expressed in it. The more mainstream SMT-solvers, like Z3 [44] and CVC5 [10], incorporate many other theory solvers for many other first-order fragments [11]. Future work should explore for which domains (besides list transformations) we can express useful over-approximating properties.

Conflict generalization As noted above, the constrain stage infers stronger constraints than the conflict clauses that a theory solver normally derives from a conflict. We could hence explore if the symmetry pruning achieved by the constrain stage is also useful when we use theory solvers other than the test stage – see Figure 7.2.

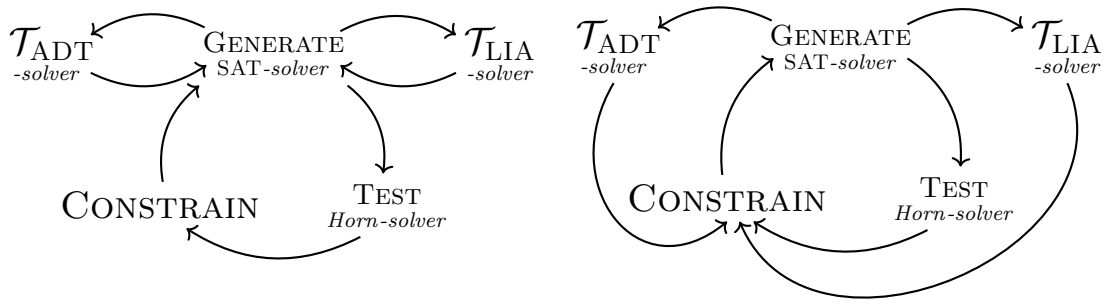
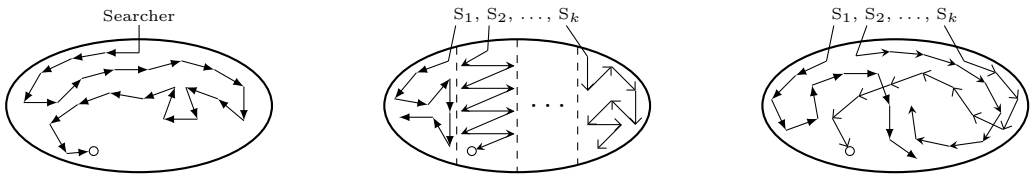


Figure 7.2: When adding theory solvers that check over-approximating properties, we have a choice regarding whether these solvers directly return conflict clauses or pass failing formulas to the constrain stage for it to derive stronger constraints.

Establishing validity In Chapter 6, we use over-approximating properties only to prune programs for which we can prove that they cannot be optimal solutions. By contrast, in functional program synthesis properties are often used to establish that the learned program has some desired property. We can adopt a similar approach to learning logic programs, with the corresponding notion being that of establishing validity of a head literal's property given the properties of the body literals. For example, when trying to

[4]Or that all positive examples failed and that we hence can make use of a redundancy constraint.



Sequential Divide-and-conquer Portfolio

Figure 7.3: One sequential and two parallel search strategies for finding a solution in the same hypothesis space.

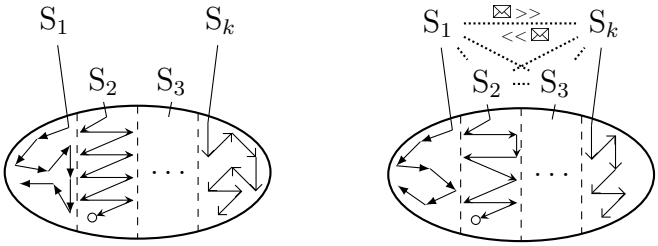
learn a definition for reverse/2, we may want to establish that our returned solution (at least) guarantees that the length of the output is the same as that of the input and use examples to prune hypotheses which do not return the elements in the right order. Like in Syntax-Guided Synthesis [3], we could simply ask the theory solvers to prove the validity of the relevant formulas (typically by asking them to prove that the negation of the formulas is unsatisfiable). As this approach prunes solutions when the given properties are too weak to establish the desired property, this approach is complementary to our usage of over-approximating properties. This hence sets out a way forward for combining deductive and inductive approaches to program synthesis.

Inferring over-approximating properties We have assumed that the user provides all over-approximating properties as part of the specification of the synthesis problem. Future work should explore whether these formulas expressing properties could be inferred, e.g. from the definitions of predicates that make up the Background Knowledge. Properties could also be inferred for the target predicate, making sure that an inferred property at least generalises the positive training examples. Note how the latter suggestion is essentially another inductive synthesis problem though now for a different (and hopefully simpler) logical language.

7.2.7 Search strategies

In delegating the search for viable hypotheses to a SAT-solver, we have imposed just one bias on the search: in order to guarantee that the first solution that `Popper` finds is optimal (i.e. has fewest literals), we impose constraints such that hypotheses are found by increasing program size. We can instead ask the SAT-solver to just search for any solution, whose size we could then use to restrict the search for subsequent solutions, (e.g. as in [29]). In general, there is an entire subfield of search heuristics left to explore.

Parallel search In a follow-up paper [40], we considered how parallel search techniques from the SMT-community can be applied to Learning From Failures. In Figure 7.3, we see how multiple "searchers" can search through the same space concurrently, by either splitting up the search space (the *divide-and-conquer* approach) or having the searchers use different heuristics and compete within the same search space (known as the *portfolio* approach). As each of these searchers is an instance of `Popper` which individually learns constraints, we can hence choose to share the learned constraints, see Figure 7.4. Our preliminary work indicates there are indeed parallel search strategies that can make effective use of scaled-up computational resources.



D&C (without comm.) D&C$_{com}$

Figure 7.4: Divide-and-conquer strategy with and without communication. With communication enabled, learned constraints are passed as messages between different instances of `Popper`.

Hypothesis space revision The above divide-and-conquer approach to splitting up the search space is not restricted to parallel search. For example, we could start with a single humongous hypothesis space and prioritise searching certain parts of it – e.g. if we think we have knowledge about which subspaces are more likely to contain solutions. When we fail to find solutions in the current subspace (or some other metric indicates this subspace is unlikely to yield better programs), we can relax the current bias and search in another area. This kind of methodology can be likened to Kuhn's notion of paradigm shifts [88], namely that the significant advances in science come from being willing to consider wildly different hypotheses from those that were previously deemed appropriate.

Bibliography

[1] John Ahlgren and Shiu Yin Yuen. "Efficient program synthesis using constraint satisfaction in inductive logic programming". In: *Journal of Machine Learning Research* 14.1 (2013), pp. 3649–3682.

[2] Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. "Constraint-Based Synthesis of Datalog Programs". In: *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*. Ed. by J. Christopher Beck. Vol. 10416. Lecture Notes in Computer Science. Springer, 2017, pp. 689–706.

[3] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. "Syntax-Guided Synthesis". In: *Dependable Software Systems Engineering*. Ed. by Maximilian Irlbeck, Doron A. Peled, and Alexander Pretschner. Vol. 40. NATO Science for Peace and Security Series, D: Information and Communication Security. IOS Press, 2015, pp. 1–25.

[4] Mario Alviano, Francesco Calimeri, Carmine Dodaro, Davide Fuscà, Nicola Leone, Simona Perri, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari. "The ASP System DLV2". In: *Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings*. Ed. by Marcello Balduccini and Tomi Janhunen. Vol. 10377. Lecture Notes in Computer Science. Springer, 2017, pp. 215–221.

[5] Krzysztof R. Apt and Roland N. Bol. "Logic Programming and Negation: A Survey". In: *Journal of Logic Programming* (1994).

[6] Duangtida Athakravi, Dalal Alrajeh, Krysia Broda, Alessandra Russo, and Ken Satoh. "Inductive Learning Using Constraint-Driven Bias". In: *Inductive Logic Programming - 24th International Conference, ILP 2014, Nancy, France, September 14-16, 2014, Revised Selected Papers*. Ed. by Jesse Davis and Jan Ramon. Vol. 9046. Lecture Notes in Computer Science. Springer, 2014, pp. 16–32.

[7] Duangtida Athakravi, Domenico Corapi, Krysia Broda, and Alessandra Russo. "Learning Through Hypothesis Refinement Using Answer Set Programming". In: *Inductive Logic Programming - 23rd International Conference, ILP 2013, Rio de Janeiro, Brazil, August 28-30, 2013, Revised Selected Papers*. Ed. by Gerson Zaverucha, Vítor Santos Costa, and Aline Paes. Vol. 8812. Lecture Notes in Computer Science. Springer, 2013, pp. 31–46.

[8] Liviu Badea. “A Refinement Operator for Theories”. In: *Inductive Logic Programming, 11th International Conference, ILP 2001, Strasbourg, France, September 9-11, 2001, Proceedings*. Ed. by Céline Rouveirol and Michèle Sebag. Vol. 2157. Lecture Notes in Computer Science. Springer, 2001, pp. 1–14.

[9] Mutsunori Banbara, Benjamin Kaufmann, Max Ostrowski, and Torsten Schaub. “Clingcon: The next generation”. In: *Theory & Practice of Logic Programming* 17.4 (2017), pp. 408–461.

[10] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. “cvc5: A Versatile and Industrial-Strength SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442.

[11] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. “The smt-lib standard: Version 2.0”. In: *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*. Vol. 13. 2010, p. 14.

[12] Clark Barrett and Cesare Tinelli. *Satisfiability modulo theories*. Springer, 2018.

[13] Aaron Bembenek, Michael Greenberg, and Stephen Chong. “From SMT to ASP: Solver-Based Approaches to Solving Datalog Synthesis-as-Rule-Selection Problems”. In: *Principles of Programming Languages* 7.POPL (2023), pp. 185–217.

[14] Nikolaj Bjørner and Lev Nachmanson. “Navigating the universe of Z3 theory solvers”. In: *Formal Methods: Foundations and Applications: 23rd Brazilian Symposium, SBMF 2020, Ouro Preto, Brazil, November 25–27, 2020, Proceedings 23*. Springer. 2020, pp. 8–24.

[15] Nikolaj S. Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. “Horn Clause Solvers for Program Verification”. In: *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. Ed. by Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte. Vol. 9300. Lecture Notes in Computer Science. Springer, 2015, pp. 24–51.

[16] Hendrik Blockeel and Luc De Raedt. “Top-Down Induction of First-Order Logical Decision Trees”. In: *Artificial Intelligence* 101.1-2 (1998), pp. 285–297.

[17] Maria Paola Bonacina. “On Conflict-Driven Reasoning”. In: *Automated Formal Methods, AFM@NFM 2017, Moffett Field, CA, USA, May 19-20, 2017*. Ed. by Bruno Dutertre and Natarajan Shankar. Vol. 5. Kalpa Publications in Computing. EasyChair, 2017, pp. 31–49.

[18] Ivan Bratko. “Refining Complete Hypotheses in ILP”. In: *Inductive Logic Programming, 9th International Workshop, ILP-99, Bled, Slovenia, June 24-27, 1999, Proceedings*. Ed. by Saso Dzeroski and Peter A. Flach. Vol. 1634. Lecture Notes in Computer Science. Springer, 1999, pp. 44–55.

[19] Alan Bundy and Boris Mitrovic. *Reformation: A domain-independent algorithm for theory repair*. Tech. rep. University of Edinburgh, 2016.

[20] Rafael Caballero, Adrián Riesco, and Josep Silva. "A Survey of Algorithmic Debugging". In: *ACM Computing Surveys* (2017).

[21] Stefano Ceri, Georg Gottlob, and Letizia Tanca. "What you Always Wanted to Know About Datalog (And Never Dared to Ask)". In: *IEEE Transmissions on Knowledge & Data Engineering* 1.1 (1989), pp. 146–166.

[22] David M Cerna and Andrew Cropper. "Generalisation Through Negation and Predicate Invention". In: *arXiv preprint arXiv:2301.07629* (2023).

[23] James Cheney, Laura Chiticariu, and Wang Chiew Tan. "Provenance in Databases: Why, How, and Where". In: *Found. Trends Databases* (2009).

[24] Alonzo Church. "A Note on the Entscheidungsproblem". In: *Journal of Symbolic Logic* 1.1 (1936), pp. 40–41.

[25] Domenico Corapi, Alessandra Russo, and Emil Lupu. "Inductive Logic Programming as Abductive Search". In: *Technical Communications of the 26th International Conference on Logic Programming, ICLP 2010, July 16-19, 2010, Edinburgh, Scotland, UK*. Ed. by Manuel V. Hermenegildo and Torsten Schaub. Vol. 7. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010, pp. 54–63.

[26] Domenico Corapi, Alessandra Russo, and Emil Lupu. "Inductive Logic Programming in Answer Set Programming". In: *Inductive Logic Programming - 21st International Conference, ILP 2011, Windsor Great Park, UK, July 31 - August 3, 2011, Revised Selected Papers*. Ed. by Stephen Muggleton, Alireza Tamaddoni-Nezhad, and Francesca A. Lisi. Vol. 7207. Lecture Notes in Computer Science. Springer, 2011, pp. 91–97.

[27] Patrick Cousot and Radhia Cousot. "Abstract interpretation and application to logic programs". In: *The Journal of Logic Programming* 13.2-3 (1992), pp. 103–179.

[28] Andrew Cropper. "Forgetting to Learn Logic Programs". In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 3676–3683.

[29] Andrew Cropper. "Learning logic programs through divide, constrain, and conquer". In: *AAAI Conference on Artificial Intelligence*. 2022.

[30] Andrew Cropper. "Playgol: Learning Programs Through Play". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*. Ed. by Sarit Kraus. ijcai.org, 2019, pp. 6074–6080.

[31] Andrew Cropper and Sebastijan Dumancic. "Inductive Logic Programming At 30: A New Introduction". In: *Journal of Artificial Intelligence Research* 74 (2022), pp. 765–850.

[32] Andrew Cropper and Sebastijan Dumancic. "Learning Large Logic Programs By Going Beyond Entailment". In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*. Ed. by Christian Bessiere. ijcai.org, 2020, pp. 2073–2079.

[33] Andrew Cropper, Richard Evans, and Mark Law. "Inductive general game playing". In: *Machine Learning* 109.7 (2020), pp. 1393–1434.

[34] Andrew Cropper and Céline Hocquette. "Learning Logic Programs by Discovering Where Not to Search". In: *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*. Ed. by Brian Williams, Yiling Chen, and Jennifer Neville. AAAI Press, 2023, pp. 6289–6296.

[35] Andrew Cropper and Rolf Morel. "Learning programs by learning from failures". In: *Machine Learning* 110.4 (2021), pp. 801–856.

[36] Andrew Cropper and Rolf Morel. "Predicate Invention by Learning From Failures". In: *CoRR* abs/2104.14426 (2021). arXiv: 2104.14426.

[37] Andrew Cropper, Rolf Morel, and Stephen Muggleton. "Learning higher-order logic programs". In: *Machine Learning* 109.7 (2020), pp. 1289–1322.

[38] Andrew Cropper and Stephen H. Muggleton. "Learning Efficient Logical Robot Strategies Involving Composable Objects". In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. Ed. by Qiang Yang and Michael J. Wooldridge. AAAI Press, 2015, pp. 3423–3429.

[39] Andrew Cropper and Stephen H. Muggleton. *Metagol System*. 2016. URL: https://github.com/metagol/metagol.

[40] Andrew Cropper, Oghenejokpeme I. Orhobor, Cristian Dinu, and Rolf Morel. "Parallel Constraint-Driven Inductive Logic Programming". In: *CoRR* abs/2109.07132 (2021). arXiv: 2109.07132.

[41] Andrew Cropper, Alireza Tamaddoni-Nezhad, and Stephen H. Muggleton. "Meta-Interpretive Learning of Data Transformation Programs". In: *Inductive Logic Programming - 25th International Conference, ILP 2015, Kyoto, Japan, August 20-22, 2015, Revised Selected Papers*. Ed. by Katsumi Inoue, Hayato Ohwada, and Akihiro Yamamoto. Vol. 9575. Lecture Notes in Computer Science. Springer, 2015, pp. 46–59.

[42] Andrew Cropper and Sophie Tourret. "Logical reduction of metarules". In: *Machine Learning* 109.7 (2020), pp. 1323–1369.

[43] Leonardo De Moura and Nikolaj Bjørner. "Efficient E-matching for SMT solvers". In: *Automated Deduction–CADE-21: 21st International Conference on Automated Deduction Bremen, Germany, July 17-20, 2007 Proceedings 21*. Springer. 2007, pp. 183–198.

[44] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[45] Jo Devriendt, Bart Bogaerts, Broes De Cat, Marc Denecker, and Christopher Mears. "Symmetry propagation: Improved dynamic symmetry breaking in SAT". In: *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*. Vol. 1. IEEE. 2012, pp. 49–56.

[46] Edsger Dijkstra. “Programming considered as a human activity”. In: *Classics in software engineering*. 1979, pp. 1–9.

[47] Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl. “Conflict-driven ASP solving with external sources”. In: *Theory and Practice of Logic Programming* 12.4-5 (2012), pp. 659–679.

[48] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. *Answer set programming: A primer*. Springer, 2009.

[49] Richard Evans and Edward Grefenstette. “Learning Explanatory Rules from Noisy Data”. In: *Journal of Artificial Intelligence Research* 61 (2018), pp. 1–64.

[50] Richard Evans, José Hernández-Orallo, Johannes Welbl, Pushmeet Kohli, and Marek J. Sergot. “Making sense of sensory input”. In: *CoRR* abs/1910.02227 (2019). arXiv: 1910.02227.

[51] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. “Program synthesis using conflict-driven learning”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Ed. by Jeffrey S. Foster and Dan Grossman. ACM, 2018, pp. 420–435.

[52] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. “Component-based synthesis of table consolidation and transformation tasks from examples”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 422–436.

[53] John K. Feser, Swarat Chaudhuri, and Isil Dillig. “Synthesizing data structure transformations from input-output examples”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Steve Blackburn. ACM, 2015, pp. 229–239.

[54] Johannes Klaus Fichte, Daniel Le Berre, Markus Hecher, and Stefan Szeider. “The Silent (R)evolution of SAT”. In: *Communications of the ACM* 66.6 (2023), pp. 64–72.

[55] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. “Example-directed synthesis: a type-theoretic interpretation”. In: *Principles of Programming Languages* 51.1 (2016), pp. 802–815.

[56] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. “DPLL(T): Fast Decision Procedures”. In: *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*. Ed. by Rajeev Alur and Doron A. Peled. Vol. 3114. Lecture Notes in Computer Science. Springer, 2004, pp. 175–188.

[57] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[58] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. “Clingo = ASP + Control: Preliminary Report”. In: *CoRR* abs/1405.3694 (2014). arXiv: 1405.3694.

[59] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. "Multi-shot ASP solving with clingo". In: *Theory & Practice of Logic Programming* 19.1 (2019), pp. 27–82.

[60] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. "Conflict-driven answer set solving: From theory to practice". In: *Artificial Intelligence* 187 (2012), pp. 52–89.

[61] Michael Gelfond and Vladimir Lifschitz. "The stable model semantics for logic programming." In: *International Conference on Logic Programming/SLP*. Vol. 88. Cambridge, MA. 1988, pp. 1070–1080.

[62] Michael R. Genesereth and Michael Thielscher. *General Game Playing*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2014.

[63] Ian P. Gent, Karen E. Petrie, and Jean-François Puget. "Symmetry in Constraint Programming". In: *Handbook of Constraint Programming*. Ed. by Francesca Rossi, Peter van Beek, and Toby Walsh. Vol. 2. Foundations of Artificial Intelligence. Elsevier, 2006, pp. 329–376.

[64] Peter Godfrey-Smith. *Theory and reality: An introduction to the philosophy of science*. University of Chicago Press, 2009.

[65] Georg Gottlob, Nicola Leone, and Francesco Scarcello. "Hypertree decompositions and tractable queries". In: *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 1999, pp. 21–32.

[66] Georg Gottlob, Nicola Leone, and Francesco Scarcello. "On the Complexity of Some Inductive Logic Programming Problems". In: *New Generation Computing* 17.1 (1999), pp. 53–75.

[67] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. "Program Synthesis". In: *Foundations and Trends in Programming Languages* 4.1-2 (2017), pp. 1–119.

[68] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. "Program synthesis by type-guided abstraction refinement". In: *Principles of Programming Languages* 4.POPL (2020), 12:1–12:28.

[69] Katsumi Inoue. "Meta-Level Abduction". In: *Journal of Logics and their Applications* 3.1 (2016), pp. 7–36.

[70] Tomi Janhunen, Roland Kaminski, Max Ostrowski, Sebastian Schellhorn, Philipp Wanko, and Torsten Schaub. "Clingo goes linear constraints over reals and integers". In: *Theory & Practice of Logic Programming* 17.5-6 (2017), pp. 872–888.

[71] Ranjit Jhala and Niki Vazou. "Refinement types: A tutorial". In: *Foundations and Trends in Programming Languages* 6.3–4 (2021), pp. 159–317.

[72] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. "Soufflé: On Synthesis of Program Analyzers". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 422–430.

[73] Roland Kaminski. *Clingo[DL] Python implementation*. 2018. URL: https://github.com/potassco/clingo/tree/master/examples/clingo/dl.

[74] Roland Kaminski, Max Ostrowski, and Philpp Wanko. *Clingo[DL] C++ implementation*. 2018. URL: https://github.com/potassco/clingo-dl.

[75] Roland Kaminski, Javier Romero, Torsten Schaub, and Philipp Wanko. "How to Build Your Own ASP-based System?!" In: *Theory & Practice of Logic Programming* 23.1 (2023), pp. 299–361.

[76] Tobias Kaminski, Thomas Eiter, and Katsumi Inoue. "Exploiting Answer Set Programming with External Sources for Meta-Interpretive Learning". In: *Theory & Practice of Logic Programming* 18.3-4 (2018), pp. 571–588.

[77] Susumu Katayama. "Systematic search for lambda expressions." In: *Trends in functional programming* 6 (2005), pp. 111–126.

[78] Nikos Katzouris, Alexander Artikis, and Georgios Paliouras. "Online learning of event definitions". In: *Theory & Practice of Logic Programming* 16.5-6 (2016), pp. 817–833.

[79] Frank C Keil and Robert Andrew Wilson. *Explanation and cognition*. MIT press, 2000.

[80] Ross D King, Jem Rowland, Stephen G Oliver, Michael Young, Wayne Aubrey, Emma Byrne, Maria Liakata, Magdalena Markham, Pinar Pir, Larisa N Soldatova, et al. "The automation of science". In: *Science* 324.5923 (2009), pp. 85–89.

[81] Ross D King, Kenneth E Whelan, Ffion M Jones, Philip GK Reiser, Christopher H Bryant, Stephen H Muggleton, Douglas B Kell, and Stephen G Oliver. "Functional genomic hypothesis generation and experimentation by a robot scientist". In: *Nature* 427.6971 (2004), pp. 247–252.

[82] Emanuel Kitzelmann and Ute Schmid. "Inductive Synthesis of Functional Programs: An Explanation Based Generalization Approach". In: *Journal of Machine Learning Research* 7 (2006), pp. 429–454.

[83] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. "Resource-guided program synthesis". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 253–268.

[84] Sven Köhler, Bertram Ludäscher, and Yannis Smaragdakis. "Declarative Datalog Debugging for Mere Mortals". In: *Datalog in Academia and Industry*. 2012.

[85] Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V. Hermenegildo, José F. Morales, Jan Wielemaker, Daniel Diaz, Salvador Abreu, and Giovanni Ciatto. "50 Years of Prolog and Beyond". In: *CoRR* abs/2201.10816 (2022). arXiv: 2201.10816.

[86] Daniel Kroening and Ofer Strichman. *Decision procedures*. Springer, 2016.

[87] Sava Krstic and Amit Goel. "Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL". In: *Frontiers of Combining Systems, 6th International Symposium, FroCoS 2007, Liverpool, UK, September 10-12, 2007, Proceedings*. Ed. by Boris Konev and Frank Wolter. Vol. 4720. Lecture Notes in Computer Science. Springer, 2007, pp. 1–27.

[88] Thomas S Kuhn. *The structure of scientific revolutions*. University of Chicago press, 1962.

[89] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. "Complete functional synthesis". In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. Ed. by Benjamin G. Zorn and Alexander Aiken. ACM, 2010, pp. 316–329.

[90] J. Larson and Ryszard S. Michalski. "Inductive inference of VL decision rules". In: *SIGART Newsletter* 63 (1977), pp. 38–44.

[91] Tessa A. Lau, Steven A. Wolfman, Pedro M. Domingos, and Daniel S. Weld. "Programming by Demonstration Using Version Space Algebra". In: *Machine Learning* 53.1-2 (2003), pp. 111–156.

[92] Mark Law. "Conflict-driven inductive logic programming". In: *Theory and Practice of Logic Programming* 23.2 (2023), pp. 387–414.

[93] Mark Law. "Inductive learning of answer set programs". PhD thesis. Imperial College London, UK, 2018.

[94] Mark Law, Alessandra Russo, Elisa Bertino, Krysia Broda, and Jorge Lobo. "FastLAS: Scalable Inductive Logic Programming Incorporating Domain-Specific Optimisation Criteria". In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 2877–2885.

[95] Mark Law, Alessandra Russo, and Krysia Broda. "Inductive Learning of Answer Set Programs". In: *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*. Ed. by Eduardo Fermé and João Leite. Vol. 8761. Lecture Notes in Computer Science. Springer, 2014, pp. 311–325.

[96] Mark Law, Alessandra Russo, and Krysia Broda. "Iterative Learning of Answer Set Programs from Context Dependent Examples". In: *Theory & Practice of Logic Programming* 16.5-6 (2016), pp. 834–848.

[97] Mark Law, Alessandra Russo, and Krysia Broda. "Learning weak constraints in answer set programming". In: *Theory & Practice of Logic Programming* 15.4-5 (2015), pp. 511–525.

[98] Mark Law, Alessandra Russo, Krysia Broda, and Elisa Bertino. "Scalable Non-observational Predicate Learning in ASP". In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*. Ed. by Zhi-Hua Zhou. ijcai.org, 2021, pp. 1936–1943.

[99] Mark Law, Alessandra Russo, Krysia Broda, and Elisa Bertino. "Scalable non-observational predicate learning in ASP". In: International Joint Conference on Artificial Intelligence. 2021.

[100] Mark H. Liffiton and Karem A. Sakallah. "Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints". In: *Journal of Automated Reasoning* 40.1 (2008), pp. 1–33.

[101] Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B. Tenenbaum, and Stephen Muggleton. "Bias reformulation for one-shot function induction". In: *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*. Ed. by Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan. Vol. 263. Frontiers in Artificial Intelligence and Applications. IOS Press, 2014, pp. 525–530.

[102] John W Lloyd. *Foundations of logic programming*. Springer Science & Business Media, 2012.

[103] David Maier, K. Tuncay Tekle, Michael Kifer, and David Scott Warren. "Datalog: concepts, history, and outlook". In: *Declarative Logic Programming: Theory, Systems, and Applications*. Ed. by Michael Kifer and Yanhong Annie Liu. Vol. 20. ACM Books. ACM / Morgan & Claypool, 2018, pp. 3–100.

[104] Victor W. Marek and Miroslaw Truszczynski. "Stable Models and an Alternative Logic Programming Paradigm". In: *The Logic Programming Paradigm - A 25-Year Perspective*. Ed. by Krzysztof R. Apt, Victor W. Marek, Mirek Truszczynski, and David Scott Warren. Artificial Intelligence. Springer, 1999, pp. 375–398.

[105] João Marques-Silva, Inês Lynce, and Sharad Malik. "Conflict-Driven Clause Learning SAT Solvers". In: *Handbook of Satisfiability - Second Edition*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 133–182.

[106] John McCarthy. "Towards a mathematical science of computation". In: *Program Verification: Fundamental Issues in Computer Science*. Springer, 1993, pp. 35–56.

[107] Jonathan Mendelson, Aaditya Naik, Mukund Raghothaman, and Mayur Naik. "GENSYNTH: Synthesizing Datalog Programs without Language Bias". In: *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 2021, pp. 6444–6453.

[108] Hakan Metin, Souheib Baarir, Maximilien Colange, and Fabrice Kordon. "Cdclsym: Introducing effective symmetry breaking in sat solving". In: *Tools and Algorithms for the Construction and Analysis of Systems: 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I 24*. Springer. 2018, pp. 99–114.

[109] Herman Midelfart. "A Bounded Search Space of Clausal Theories". In: *Inductive Logic Programming, 9th International Workshop, ILP-99, Bled, Slovenia, June 24-27, 1999, Proceedings*. Ed. by Saso Dzeroski and Peter A. Flach. Vol. 1634. Lecture Notes in Computer Science. Springer, 1999, pp. 210–221.

[110] Tom M. Mitchell. "Generalization as Search". In: *Artificial Intelligence* 18.2 (1982), pp. 203–226.

[111] Rolf Morel. "Constraint-Driven Learning of Logic Programs". In: *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI*. 2021.

[112] Rolf Morel. "Refinement Type Directed Search for Meta-Interpretive Learning of Higher-Order Logic Programs". Master's thesis. University of Oxford, 2018.

[113] Rolf Morel and Andrew Cropper. “Learning logic programs by explaining their failures”. In: *Machine Learning* 112.10 (2023), pp. 3917–3943.

[114] Rolf Morel, Andrew Cropper, and C.-H. Luke Ong. “Typed Meta-interpretive Learning of Logic Programs”. In: *Logics in Artificial Intelligence - 16th European Conference, JELIA*. Vol. 11468. 2019, pp. 198–213.

[115] S.H. Muggleton, R.D. King, and M.J.E. Sternberg. “Protein secondary structure prediction using logic-based machine learning”. In: *Protein Engineering* 5.7 (1992), pp. 647–657.

[116] S.H. Muggleton, U. Schmid, C. Zeller, A. Tamaddoni-Nezhad, and T. Besold. “Ultra-Strong Machine Learning - Comprehensibility of Programs Learned with ILP”. In: *Machine Learning* 107 (7 2018), pp. 1119–1140.

[117] Stephen Muggleton. “Inductive Logic Programming”. In: *New Generation Computing* 8.4 (1991), pp. 295–318.

[118] Stephen Muggleton. “Inverse Entailment and Progol”. In: *New Generation Computing* 13.3&4 (1995), pp. 245–286.

[119] Stephen Muggleton, Luc De Raedt, David Poole, Ivan Bratko, Peter A. Flach, Katsumi Inoue, and Ashwin Srinivasan. “ILP turns 20 - Biography and future challenges”. In: *Machine Learning* 86.1 (2012), pp. 3–23.

[120] Stephen H. Muggleton, Dianhuan Lin, Niels Pahlavi, and Alireza Tamaddoni-Nezhad. “Meta-interpretive learning: application to grammatical inference”. In: *Machine Learning* 94.1 (2014), pp. 25–49.

[121] Stephen H. Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. “Meta-interpretive learning of higher-order dyadic Datalog: predicate invention revisited”. In: *Machine Learning* 100.1 (2015), pp. 49–73.

[122] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. *Foundations of Inductive Logic Programming*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1997. ISBN: 3540629270.

[123] Robert Nieuwenhuis and Albert Oliveras. “DPLL(T) with Exhaustive Theory Propagation and Its Application to Difference Logic”. In: *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*. Ed. by Kousha Etessami and Sriram K. Rajamani. Vol. 3576. Lecture Notes in Computer Science. Springer, 2005, pp. 321–334.

[124] Robert Nieuwenhuis and Albert Oliveras. “On SAT Modulo Theories and Optimization Problems”. In: *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*. Ed. by Armin Biere and Carla P. Gomes. Vol. 4121. Lecture Notes in Computer Science. Springer, 2006, pp. 156–169.

[125] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. “Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T)”. In: *Journal of the ACM (JACM)* 53.6 (2006), pp. 937–977.

[126] Peter-Michael Osera and Steve Zdancewic. “Type-and-example-directed program synthesis”. In: *Principles of Programming Languages* 50.6 (2015), pp. 619–630.

[127] Michael J Pazzani and Clifford A Brunk. "Detecting and correcting errors in rule-based expert systems: an integration of empirical and explanation-based learning". In: *Knowledge acquisition* 3.2 (1991), pp. 157–173.

[128] Karen E Petrie, Barbara M Smith, and Neil Yorke-Smith. "Dynamic symmetry breaking in constraint programming and linear programming hybrids". In: *European starting AI researcher symp.* 2004.

[129] G.D. Plotkin. "Automatic Methods of Inductive Inference". PhD thesis. Edinburgh University, Aug. 1971.

[130] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. "Program synthesis from polymorphic refinement types". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. Ed. by Chandra Krintz and Emery Berger. ACM, 2016, pp. 522–538.

[131] Enrico Pontelli, Tran Cao Son, and Omar Elkhatib. "Justifications for logic programs under answer set semantics". In: *Theory and Practice of Logic Programming* 9.1 (2009), pp. 1–56.

[132] K.R. Popper. *Conjectures and Refutations: The Growth of Scientific Knowledge*. Routledge, 1963. ISBN: 9780415285940.

[133] Karl Popper. *The logic of scientific discovery*. Routledge, 1935.

[134] Stanislaw J. Purgal, David M. Cerna, and Cezary Kaliszyk. "Learning Higher-Order Logic Programs From Failures". In: *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*. Ed. by Luc De Raedt. ijcai.org, 2022, pp. 2726–2733.

[135] J. Ross Quinlan. "Learning Logical Definitions from Relations". In: *Machine Learning* 5 (1990), pp. 239–266.

[136] Luc De Raedt. *Logical and relational learning*. Cognitive Technologies. Springer, 2008. ISBN: 978-3-540-20040-6.

[137] Luc De Raedt and Maurice Bruynooghe. "A Theory of Clausal Discovery". In: *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993*. Ed. by Ruzena Bajcsy. Morgan Kaufmann, 1993, pp. 1058–1063.

[138] Luc De Raedt and Maurice Bruynooghe. "Interactive Concept-Learning and Constructive Induction by Analogy". In: *Machine Learning* 8 (1992), pp. 107–150.

[139] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. "Provenance-guided synthesis of Datalog programs". In: *Principles of Programming Languages* 4.POPL (2020), 62:1–62:27.

[140] Oliver Ray. "Non-monotonic abductive inductive learning". In: *Journal of Applied Logic* 7.3 (2009), pp. 329–340.

[141] Bradley L. Richards and Raymond J. Mooney. "Automated Refinement of First-Order Horn-Clause Domain Theories". In: *Mach. Learn.* 19.2 (1995), pp. 95–131. URL: https://%doi.org/10.1007/BF01007461.

[142] Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. "Liquid types". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2008, pp. 159–169.

[143] Chiaki Sakama and Katsumi Inoue. "Brave induction: a logical framework for learning from incomplete information". In: *Machine Learning* 76.1 (2009), pp. 3–35.

[144] Peter Schüller and Mishal Benz. "Best-effort inductive logic programming via fine-grained cost-based hypothesis generation - The inspire system at the inductive logic programming competition". In: *Machine Learning* 107.7 (2018), pp. 1141–1169.

[145] Ehud Y. Shapiro. *Algorithmic Program DeBugging*. Cambridge, MA, USA: MIT Press, 1983. ISBN: 0262192187.

[146] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. "Syntax-guided synthesis of Datalog programs". In: *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT)*. ACM, 2018.

[147] Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. "Synthesizing Datalog Programs using Numerical Relaxation". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*. Ed. by Sarit Kraus. ijcai.org, 2019, pp. 6117–6124.

[148] Armando Solar-Lezama. *MIT 6.S981 Introduction to Program Synthesis*. Fall 2023. URL: https://people.csail.mit.edu/asolar/SynthesisCourse/index.htm.

[149] Armando Solar-Lezama. "Program synthesis by sketching". PhD thesis. University of California, Berkeley, 2008.

[150] A. Srinivasan. "The ALEPH manual". In: *Machine Learning at the Computing Laboratory, Oxford University* (2001).

[151] Ashwin Srinivasan, Stephen H Muggleton, Michael JE Sternberg, and Ross D King. "Theories for mutagenicity: A study in first-order and feature-based induction". In: *Artificial Intelligence* 85.1-2 (1996), pp. 277–299.

[152] Phillip D Summers. "A methodology for LISP program construction from examples". In: *Journal of the ACM (JACM)* 24.1 (1977), pp. 161–175.

[153] Sten-Åke Tärnlund. "Horn Clause Computability". In: *BIT* 17.2 (1977), pp. 215–226.

[154] George Thompson and Allison K. Sullivan. "ProFL: a fault localization framework for Prolog". In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2020.

[155] William Yang Wang, Kathryn Mazaitis, and William W. Cohen. "Structure Learning via Parameter Learning". In: *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014*. Ed. by Jianzhong Li, Xiaoyang Sean Wang, Minos N. Garofalakis, Ian Soboroff, Torsten Suel, and Min Wang. ACM, 2014, pp. 1199–1208.

[156] Stefan Wrobel. "First order theory refinement". In: *Advances in inductive logic programming* 32 (1996), pp. 14–33.

[157] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. "Efficient Conflict Driven Learning in Boolean Satisfiability Solver". In: *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2001, San Jose, CA, USA, November 4-8, 2001*. Ed. by Rolf Ernst. IEEE Computer Society, 2001, pp. 279–285.

Appendix A

Appendix: Learning Programs by Learning From Failures

A.1 Language biases in buttons experiment

A.1.1 ILASP2i and ILASP3

```
#modeh(1,f, (positive)).
#modeb(1,button1, (positive)).
...
#modeb(1,button20, (positive)).
```

A.1.2 Popper and Enumerate

```
max_vars(1).
max_clauses(1).
head_pred(f,1).
body_pred(button1,1).
...
body_pred(button20,1).
```

A.1.3 Aleph

```
:- aleph_set(i,6).
:- aleph_set(clauselength,2).
:- aleph_set(nodes,5000).
:- modeh,f(+var)).
:- modeb(*,button1(+var)).
:- determination(f/1,button1/1).
:- modeb(*,button2(+var)).
...
:- determination(f/1,button20/1).
```

A.1.4 Metagol

```
metarule(unary1, [P,Q], [P,A], [[Q,A]]).
metarule(unary2, [P,Q,R], [P,A], [[Q,A],[R,A]]).
body_pred(button1/1).
...
body_pred(button20/1).
```

A.2 Language biases in robots experiment

A.2.1 ILASP2i and ILASP3

```
#modeh(f(var(state)), (positive)).
#modeh(start_state(var(state)), (positive)).
#modeb(3,move_up(var(state),var(state)), (anti_reflexive,symmetric,positive)).
#modeb(3,move_down(var(state),var(state)), (anti_reflexive,symmetric,positive)).
#modeb(3,move_left(var(state),var(state)), (anti_reflexive,symmetric,positive)).
#modeb(3,move_right(var(state),var(state)), (anti_reflexive,symmetric,positive)).
#modeb(3,at_top(var(state)), (positive)).
#modeb(3,at_bottom(var(state)), (positive)).
#modeb(3,at_left(var(state)), (positive)).
#modeb(3,at_right(var(state)), (positive)).
#modeb(1,start_state(var(state)), (positive)).

#bias(":- occurs(V, X), #false : occurs(V, Y), Y != X.").
#bias("occurs(X, f(X)) :- head(f(X)).").
#bias("occurs(X, start_state(X)) :- head(start_state(X)).").
#bias("occurs(X, start_state(X)) :- body(start_state(X)).").
#bias("occurs(X, at_top(X)) :- body(at_top(X)).").
#bias("occurs(X, at_bottom(X)) :- body(at_bottom(X)).").
#bias("occurs(X, at_left(X)) :- body(at_left(X)).").
#bias("occurs(X, at_right(X)) :- body(at_right(X)).").
#bias("occurs(X, move_up(X, Y)) :- body(move_up(X, Y)).").
#bias("occurs(X, move_left(X, Y)) :- body(move_left(X, Y)).").
#bias("occurs(X, move_right(X, Y)) :- body(move_right(X, Y)).").
#bias("occurs(X, move_down(X, Y)) :- body(move_down(X, Y)).").
#bias("occurs(X, move_up(Y, X)) :- body(move_up(Y, X)).").
#bias("occurs(X, move_left(Y, X)) :- body(move_left(Y, X)).").
#bias("occurs(X, move_right(Y, X)) :- body(move_right(Y, X)).").
#bias("occurs(X, move_down(Y, X)) :- body(move_down(Y, X)).").
```

A.2.2 Popper and Enumerate

```
max_vars(4).
max_body(3).
```

```
max_clauses(3).
head_pred(f,2).
body_pred(f,2).
body_pred(at_top,1).
body_pred(at_bottom,1).
body_pred(at_left,1).
body_pred(at_right,1).
body_pred(move_left,2).
body_pred(move_right,2).
body_pred(move_up,2).
body_pred(move_down,2).
direction(f,0,in).
direction(f,1,out).
direction(move_left,0,in).
direction(move_right,0,in).
direction(move_up,0,in).
direction(move_down,0,in).
direction(move_left,1,out).
direction(move_right,1,out).
direction(move_up,1,out).
direction(move_down,1,out).
direction(at_top,0,in).
direction(at_bottom,0,in).
direction(at_left,0,in).
direction(at_right,0,in).
```

A.2.3 Aleph

```
:- aleph_set(i,6).
:- aleph_set(clauselength,6).
:- aleph_set(nodes,50000).
:- modeh,f(+state,-state)).
:- modeb(*,move_up(+state,-state)).
:- modeb(*,move_down(+state,-state)).
:- modeb(*,move_left(+state,-state)).
:- modeb(*,move_right(+state,-state)).
:- modeb(*,at_top(+state)).
:- modeb(*,at_bottom(+state)).
:- modeb(*,at_left(+state)).
:- modeb(*,at_right(+state)).
:- determination(f/2,move_up/2).
:- determination(f/2,move_down/2).
:- determination(f/2,move_left/2).
:- determination(f/2,move_right/2).
:- determination(f/2,at_top/1).
:- determination(f/2,at_bottom/1).
:- determination(f/2,at_left/1).
```

```
:- determination(f/2,at_right/1).
```

A.2.4 Metagol

```
body_pred(move_right/2).
body_pred(move_left/2).
body_pred(move_up/2).
body_pred(move_down/2).
body_pred(at_top/1).
body_pred(at_bottom/1).
body_pred(at_left/1).
body_pred(at_right/1).
metarule([P,Q], [P,A], [[Q,A]]).
metarule([P,Q], [P,A], [[Q,A]]).
metarule([P,Q,R], [P,A], [[Q,A,B],[R,B]]).
metarule([P,Q], [P,A], [[Q,A,B],[P,B]]).
metarule([P,Q,R], [P,A], [[Q,A,B],[R,A,B]]).
metarule([P,Q], [P,A,B], [[Q,A,B]]).
metarule([P,Q,R], [P,A,B], [[Q,A,B],[R,A,B]]).
metarule([P,Q,R], [P,A,B], [[Q,A],[R,A,B]]).
metarule([P,Q,R], [P,A,B], [[Q,A,B],[R,B]]).
metarule([P,Q,R], [P,A,B], [[Q,A,C],[R,C,B]]).
metarule([P,Q], [P,A,B], [[Q,A,C],[P,C,B]]).
metarule([P,Q], [P,A,B], [[Q,B,A]]).
```

A.3 Language biases in lists experiment

A.3.1 Popper and Enumerate

For each list transformation problem, we have a specific bias to specify the target relations, such as the following bias for the `finddup` problem:

```
head_pred(f,2).
type(f,0,list).
type(f,1,element).
direction(f,0,in).
direction(f,1,out).
body_pred(f,2).
```

For all the problems we include the following biases:

```
max_vars(5).
max_body(5).
max_clauses(2).
body_pred(head,2).
body_pred(tail,2).
body_pred(geq,2).
```

```
body_pred(empty,1).
body_pred(even,1).
body_pred(odd,1).
body_pred(one,1).
body_pred(zero,1).
body_pred(decrement,2).
body_pred(increment,2). % ONLY FOR SORTED
body_pred(element,2). % ONLY FOR FIND DUPLICATE
body_pred(cons,2). % ONLY FOR ADDHEAD, DROPK, DROPLAST
type(cons,0,element).
type(cons,1,list).
type(cons,2,list).
direction(cons,0,in).
direction(cons,1,in).
direction(cons,2,out).
type(head,0,list).
type(head,1,element).
direction(head,0,in).
direction(head,1,out).
type(tail,0,list).
type(tail,1,list).
direction(tail,0,in).
direction(tail,1,out).
type(empty,0,list).
direction(empty,0,in).
type(element,0,list).
type(element,1,element).
direction(element,0,in).
direction(element,1,out).
type(increment,0,element).
type(increment,1,element).
direction(increment,0,in).
direction(increment,1,out).
type(decrement,0,element).
type(decrement,1,element).
direction(decrement,0,in).
direction(decrement,1,out).
type(geq,0,element).
type(geq,1,element).
direction(geq,0,in).
direction(geq,1,in).
type(even,0,element).
direction(even,0,in).
type(odd,0,element).
direction(odd,0,in).
type(one,0,element).
direction(one,0,in).
```

```
type(zero,0,element).
direction(zero,0,out).
```

A.3.2 Aleph

For each list transformation problem, we have a specific bias to specify the target relations, such as the following bias for the `finddup` problem:

```
:- modeh,f(+list,-element)).
:- modeb(*,f(+list,-element)).
```

For all the problems we include the following biases (we we replace `f/2` in the determinations with the correct arity of the target predicate):

Note that `increment` is only given in the `sorted` experiment, `element` is only given in the `finddupli` experiment, and `cons` is only given in the `addhead`, `dropk`, and `droplast` experiments.

```
:- aleph_set(i,6).\\
:- aleph_set(clauselength,6).\\
:- aleph_set(nodes,30000).\\
:- modeb,head(+list,-element)).
:- modeb(*,tail(+list,-list)).
:- modeb(*,geq(+element,+element)).
:- modeb(*,empty(+list)).
:- modeb(*,even(+element)).
:- modeb(*,odd(+element)).
:- modeb(*,one(+element)).
:- modeb(*,zero(-element)).
:- modeb(*,decrement(+element,-element)).
:- modeb(*,increment(+element,-element)).
:- modeb(*,element(+list,-element)).
:- modeb(*,cons(+element,+list,-list)).
```

A.3.3 Metagol

```
body_pred(head/2).
body_pred(tail/2).
body_pred(geq/2).
body_pred(empty/1).
body_pred(even/1).
body_pred(odd/1).
body_pred(one/1).
body_pred(zero/1).
body_pred(decrement/2).
body_pred(increment/2). % ONLY FOR SORTED
body_pred(member/2). % ONLY FOR FIND DUPLICATE
```

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

Appendix B

Appendix: Learning Programs by Explaining Their Failures

B.1 Experiment 2: Hypothesis Space Settings

The following hypothesis space settings were used in the programming puzzles experiment:

problem	max #clauses	max #literals	max #variables	sum/3	cons/3	snoc/3	head/2	tail/2	element/2	decrement/2	increment/2	qeg/2	even/1	odd/1	one/1	zero/1	empty/1
addhead/2	3	7	6	x	x		x	x		x		x	x	x	x	x	x
dropk/3	3	6	5	x	x		x	x		x	x	x	x	x	x	x	x
droplast/2	3	6	5		x		x	x		x	x	x	x	x	x	x	x
evens/1	2	6	5	x			x	x				x	x	x	x	x	x
finddup/2	2	6	5	x			x	x	x	x		x	x	x	x	x	x
last/2	3	7	6				x	x				x	x	x	x	x	x
len/2	2	6	6	x			x	x		x		x	x	x	x	x	x
member/2	3	7	6	x			x	x		x		x	x	x	x	x	x
odd1even2/2	3	6	5	x			x	x				x	x	x	x	x	x
reverse/2	3	5	5			x	x	x		x		x	x	x	x	x	x
sorted/1	3	6	5				x	x		x		x	x	x	x	x	x
sumlist/2	2	6	5	x	x		x	x		x		x	x	x	x	x	x
threesame/1	3	7	6	x			x	x		x		x	x	x	x	x	x

Appendix C

Appendix: Learning From Failures Modulo Theories

C.1 Hypothesis Space Settings in Experiments

The following hypothesis space settings were used in the list transformation experiments:

problem	max #clauses	max #literals	max #variables	sum/3	cons/3	snoc/3	head/2	tail/2	element/2	decrement/2	increment/2	geq/2	even/1	odd/1	one/1	zero/1	empty/1
addhead/2	3	7	6	x	x		x	x		x		x	x	x	x	x	x
dropk/3	2	6	5	x	x		x	x		x		x			x	x	x
droplast/2	3	5	5		x		x	x		x		x	x	x	x	x	x
evens/1	3	6	5	x			x	x				x	x	x	x	x	x
finddup/2	3	6	5				x	x	x			x	x	x	x	x	x
last/2	3	7	6				x	x		x	x	x	x	x	x	x	x
len/2	2	6	6	x			x	x	x		x	x	x	x	x	x	x
reverse/2	2	5	5			x	x	x				x	x	x	x	x	x
sorted/1	3	6	5				x	x		x		x	x	x	x	x	x
sumlist/2	3	5	5	x	x		x	x		x		x	x	x	x	x	x
threesame/1	3	7	6	x			x	x				x	x	x	x	x	x