FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A practical view of answer-set programming transformations

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Mikołaj Koziarkiewicz
Matrikelnummer 0309169

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer/in: Ao. Univ.-Prof. Dr. techn. Hans Tompits

Wien, 04.03.2011

(Unterschrift Verfasser/in)

(Unterschrift Betreuer/in)

Deutsche Zusammenfassung

Diese Arbeit beschäftigt sich mit der Untersuchung des praktisches Nutzens von formalen Transformationen in der Answer-Set Programmen. Answer-Set Programmierung (ASP) ist ein für deklaratives Problemlösen bestimmter Formalismus, der auf Prinzipien des nichtmonotonen Schließens aufbaut.

Wir beginnen mit einer Übersicht des heutigen Stands in dem Feld. Unsere Untersuchungen basieren auf zwei unterschiedlichen Transformationssysteme – eines, dass auf bestimmten syntaktischen Transformationen basiert; und ein zweites, welches auf semantischen Bedingungen aufbaut und redundante Regeln so wie Literale entfernt.

Anschließend wird eine Diskussion über die Implementierung von Algorithmen zur Bestimmung der Anwendbarkeit von Regeln für diese zwei Systeme durchgeführt. Danach zeigen wir mit Hilfe der Interaktionseigenschaften der Transformationen, wie die Eingabedaten intern effektiv behandelt werden können.

Eine Implementierung in Python wird benutzt, um eine Untersuchung der Anwendbarkeit der Transformationen bezüglich unterschiedlichen Beispielprogrammen durchzuführen. Analysiert werden, unter anderem, Studentenprogramme, zufallsgenerierte Programme, und automatisch generierte Programme erzeugt von verschiedenen Frontends für deklaratives Problemlösen.

Die Resultate zeigen, dass die betrachteten Transformationen nur eine geringe Anwendbarkeit besitzen. Außer für den Fall von Zufallsprogrammem wird nur eine kleine Anzahl von Anwendungsinstanzen gefunden. Folgen von anwendbaren Transformationen sind praktisch nicht existent.

Somit ist der praktische Nutzen für diese in der Literatur diskutierten Transformationen für Online- oder Offline-Optimierung gering.

Abstract

The focus of this thesis is the examination of the practicality of applying known formal transformations of answer-set programs. Answer-set programming (ASP) is a formalism for declarative problem solving, based on the principles of non-monotonic reasoning.

The research is started by constructing an overview of well-defined transformations existing in the scientific literature. We consider two systems of transformations – one composed of certain syntactic transformations; the other of the semantic removal of rules and literals.

This work then discusses algorithms for implementing applicability checking within the two systems, either existing ones from the literature, or new ones. Next, it is shown how to effectively process input data within the defined frameworks, utilizing properties related to the interactions between the transformations.

With the help of a Python implementation, several data sets composed of answer-set programs are checked for transformation applicability, including student-written programs, randomly-generated programs, and programs created by various front-ends for declarative problem solving.

The results obtained show that the utility of applying the considered program transformations is quite low. Barring random input, only a small number of applicable transformation instances is found. Moreover, sequences of transformation applications are almost non-existent.

Hence, there is little practical gain in using the considered transformation for online or offline optimization.

Erklärung zur Verfassung der Arbeit

Mikołaj Koziarkiewicz
ul. Powązkowska 59E/32
01-728 Warszawa
POLAND

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 4. März 2011

Acknowledgements

I would like to thank my supervisor for the endless patience and constructive advice.

Next, many thanks go out to my family and friends for the support they have given.

And last but not least, I would like to extend an additional "thank you" to Karol Czachorowski, who kindly provided me with the server access utilized in the experiments performed for this thesis.

Mikołaj Koziarkiewicz

Contents

Chapter 1

Introduction

Answer-Set programming (ASP) is a formalism that has steadily been gaining in popularity in the recent years. As a declarative logic-oriented programming language, it is especially suited to solving problems related to knowledge representation. It also possesses the advantage of being able to express non-monotonic knowledge.

One of the developments in the area has been the construction of different frameworks for solving reasoning tasks based on various ASP interpreters. Such frameworks usually employ their own language for problem definition (often similar in some way to an ASP variant), translating the obtained specification into a logic program which is then fed to an underlying ASP solver. Finally, the output from such a solver is collected and processed. The described approach has the obvious advantage of limiting the necessary work to problem-specific issues, be it diagnosis [EFLP99], planning [EFL$^+$04, EFL$^+$03], or some other task.

On the other hand, such automatic, generalized compilation from the specification language of the framework to ASP has the inherent danger of creating some sort of suboptimality. However, what actually amounts to being "suboptimal" is ambiguous in this context.

A rational interpretation of optimality would concentrate on redundancy present in the program. In other words, the existence of some shorter program (for example, one with less rules) semantically equivalent to the output of the framework's compiler infers the possibility for improvement. The crux of the matter in both cases is located in the term "semantically equivalent". ASP, due to its non-monotonic nature, has several definitions of equivalence – two programs with the same semantics can have a different meaning with the introduction of additional knowledge, identical in both cases. This issue shall be discussed in the coming sections of this thesis.

The goal of this thesis is to examine the actual practicality of using program transformations on various programs from multiple sources, including those generated by ASP solvers, but also "human-made" data. With regard to the remarks made above, the main benchmark will be the number of rules and the size of their head and body sets. This is because there is a certain problem with a runtime benchmark, apart from the differences by using different solvers.

To illustrate the above points, let us consider the following program:

$$P = \{a \leftarrow\}.$$

For this very basic program, one can very easily find another one that is equivalent, i.e., one that has the same semantics, or informally, "meaning", say:

$$\begin{aligned}\{&a \leftarrow,\\ &c \leftarrow \text{not } a\}.\end{aligned}$$

This trivial example illustrates the fact that each program has a potential for having a large set of other programs equivalent to it (especially for certain kinds of equivalence). It is therefore inefficient, if not futile (given lack of bound on the size of programs), to simply enumerate all equivalent variants. Instead, simplification by executing a sequence of previously defined transformations will be attempted. The main criteria here are the length of the program and the amount of time it takes to verify whether one can apply the transformations to the supplied input.

The thesis is organized as follows. The next chapter introduces the necessary syntactical and semantical notions of answer-set programming. Chapter 3 will provide an overview of the current research results in the field of equivalence-preserving transformations of logic programs under the answer-set semantics. In Chapter 4, the transformations used for the purpose of data analysis are selected, and algorithms for testing of their applicability are developed, researched, and analyzed. Chapter 5 considers the transformations in the context of their respective systems, exploring the properties that emerge from their interaction with one another. Chapter 6 presents the software framework developed for the experiments, and demonstrates the results on various data sets. The final chapter interprets those results and lists the conclusions made, suggesting further research as well.

Chapter 2

Answer-Set Programming

This chapter introduces the basic concepts of answer-set programming [GL88, GL91], needed for the understanding of this thesis. Firstly, we discuss syntactical notions and afterwards semantic ones.

2.1 Syntax

The sets of *constants*, *variables*, and *predicates* are pairwise disjoint subsets of some language $\mathcal{L}$.[1] Additionally, the set of constants, also called *domain*, denoted by $\mathcal{C}$, may be either finite or infinite. A *term* is either a constant (a 0-ary function) or a variable. An *atom* is a predicate symbol, followed by a comma-delimited list of terms with length corresponding to the symbol's arity, in parentheses (an atom with a 0-ary predicate omits those parentheses, for brevity). A *literal* is an atom optionally preceded by $\neg$, the sign of *strong negation*. We actually distinguish between two types of negation: besides $\neg$, we also have *default negation*, denoted by 'not', also referred to as *negation as failure*. A *NAF literal* is a literal optionally preceded by 'not'.

We use $|l|$ to denote the predicate symbol of literal l, and $n(p)$ to denote the arity of a predicate symbol p. The set of all predicate symbols present in atoms of a given construct X (either a rule, program, or a set of atoms) is denoted by $\mathcal{R}_X$.

A rule is a pair of the form

$$a_1 \vee \ldots \vee a_l \leftarrow b_1, \ldots, b_m, \text{not}\, c_1, \ldots, \text{not}\, c_n \ ,$$

where $a_1, \ldots, a_l, b_1, \ldots, b_m, c_1, \ldots, c_n$ are literals.

We call $a_1 \vee \ldots \vee a_l$ the *head* of the rule and $b_1, \ldots, b_m, \text{not}\, c_1, \ldots, \text{not}\, c_n$ the *body of the rule*.

A rule is

[1] By convention, variables start with an uppercase letter or an underscore, whereas constants begin with a lowercase letter or number.

- *basic* if $n = 0$,
- *disjunctive* if $l > 1$,
- *normal* if it is non-disjunctive and every literal is positive, and
- *ground* if every literal is ground, i.e., if it contains no variables.

Additionally, a rule of the form

$$\leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n$$

is called a *constraint*.

A *program* is a finite set of rules. The above taxonomy of rules applies to programs for which every member rule possesses the classifying property.

Some definitions regarding rules need to be fixed for a simpler specification of several concepts later on. For any given rule r:

- $H(r)$ is the set of all literals occurring in the head of r,
- $B^+(r)$ is the *positive body* of r, i.e., the set of all literals occurring in the body of r which are not preceded by 'not',
- $B^-(r)$ is the *negative body* of r, i.e., the set of all default negated literals occurring in the body of r with 'not' removed, and
- $B(r) = B^+(r) \cup \text{not } B^-(r)$, where, for a given set of literals S, $\text{not } S = \{\text{not } a : a \in S\}$.

Therefore, a given rule might also be represented in set notation as

$$H(r) \leftarrow B^+(r), \text{not } B^-(r).$$

The final three syntactical concepts, the notions of *safety*, *grounding*, and *head-cycle freeness*, possess a large significance to semantics.

A rule r is *safe* if every variable in all literals contained in $H(r)$ and $B^-(r)$ is also contained in a literal in $B^+(r)$. A program is *safe* if all its rules are safe. This thesis deals only with safe programs.

For non-ground programs, a *variable assignment* ϑ is a function assigning a symbol from the set $\mathcal{V}$ of variables to a constant from the domain $\mathcal{C}$. A *grounding* of an atom is generated by replacing all variable symbols by the corresponding constant symbols in the atom. Groundings for a literal, rule, or a program are defined analogously. In general, for a variable assignment ϑ, the grounding of an object a (whether an atom, literal, rule, program, or a set of these) with respect to ϑ is denoted as $a\vartheta$.

Given a ground program P, the *positive dependency graph of* P, G_P, is a directed graph defined as

$$G_P = (V_P, E_P),$$

where V_P is the set of all possible atoms generated from $\mathcal{R}_P$ and the constants in P, and

$$E_P = \{(a, b) : a \in H(r), b \in B^+(r), \text{ and } r \in P\}.$$

A rule r in a ground program P is *head-cycle free* (HCF) iff, for all pairs of distinct atoms $a, b \in H(r)$, no cycle containing both a and b exists in G_P.

A rule r in any program P is *HCF under* $\mathcal{C}$, iff, for each $C \subseteq \mathcal{C}$, the grounding of r with respect to C is HCF in the grounding of P with respect to C [EFT$^+$06].

To specify some of the declarative algorithms introduced in this thesis, the notion of *aggregates* will be introduced. An aggregate is a special construct enabling a potentially higher expressiveness of a program [FLPP04]. To begin with, by a *symbolic set* we understand an expression of the form $\{\mathit{Vars} : \mathit{Conj}\}$, where *Vars* is a tuple of variables and *Conj* is a conjunction of literals. Furthermore, a *ground set* is a set of pairs of the form $(t_1, \ldots, t_n : \mathit{Conj})$, where $t_1, \ldots, t_n$ is a list of constants and *Conj* is a ground conjunction of NAF literals. Then, an *aggregate function* is an expression of the form

$$fS,$$

where f is the name of the function and S is either a symbolic set or a ground set (in the latter case, fS is a *ground aggregate function*).

An *aggregate atom* has the form

$$fS \prec n,$$

where $\prec \in \{=, <, \leq, >, \geq\}$, fS is an aggregate function, and n, called *right guard*, is a positive number. Aggregate atoms may appear in bodies of rules but not in heads.

By a *local variable* of a rule r we understand a variable which appears solely in an aggregate function in r. A variable which is not local is *global*. The notion of safety can be extended to rules with aggregates thus: we call a rule r safe if (i) each global variable of r appears in a positive literal of the body of r which itself does not occur in an aggregate atom; (ii) each local variable of r that appears in a symbolic set $\{\mathit{Vars} : \mathit{Conj}\}$ also appears in a positive literal in *Conj*. Finally, a program is safe if all of its rules are safe.

2.2 Semantics

To begin with, we need to introduce the notion of an *interpretation* – it is a consistent set of literals, i.e., one which does not contain an atom p and its negation $\neg p$. Furthermore, for a ground program P, an interpretation I is a *model* of P if, for all $r \in P$,

$$B^+(r) \subseteq I \text{ and } B^-(r) \cap I = \emptyset \text{ implies } H(r) \cap I \neq \emptyset.$$

A rule that fulfills the antecedent of the implication is *applicable* under I, whereas a rule for which additionally the consequent is true is *applied* under I. Note that a constraint can never fulfill the consequent, so if it is applicable under an interpretation, that interpretation is not a model for the specified program.

For a ground basic program P, an interpretation I is an *answer set* of P if it is minimal among the set of models of P.

Example 1. *Consider*

$$P = \{a \vee b \vee c \leftarrow\}.$$

Since the single rule of P is applicable under any interpretation (it is a disjunctive fact), then, the set of models of P is

$$\{\{a, b, c\}, \{a, c\}, \{a, b\}, \{b, c\}, \{a\}, \{b\}, \{c\}\}.$$

However, due to the minimality condition only $\{a\}, \{b\}$, and $\{c\}$ are answer sets of P.

Having defined answer sets for ground basic programs, let us next consider the case of arbitrary ground programs. To this end, we define the *reduct* of a ground program P under an interpretation I, denoted by P^I, as follows:

1. Remove any rule r for which $B^-(r) \cap I \neq \emptyset$, and
2. for all remaining rules, remove all default negated literals from their bodies.

The result is a ground basic program. Then, an interpretation I of a program P is an *answer set* of P if it is an answer set of P^I. Answer sets are, informally, "results" of the program. The set of all answer sets for P is denoted by $\mathcal{AS}(P)$.

The concepts of a model and an answer set of a non-ground program are defined analogously by applying them to a program created from all possible rule groundings.

Defining the semantics of programs allows for considering the equivalence between a pair of them. This is not as simple as, for example, equivalence of formulas under zero- or first-order logic, due to the aforementioned non-monotonicity of ASP. In what follows, three common notions of equivalence are introduced:

- two programs P and S are *ordinarily equivalent* equivalent iff $\mathcal{AS}(P) = \mathcal{AS}(S)$; denoted by $P \equiv_o S$,
- two programs P and S are *strongly equivalent* iff $\mathcal{AS}(P \cup R) = \mathcal{AS}(S \cup R)$, for any program R, denoted by $P \equiv_s S$ [LPV01],
- two programs P and S are *uniformly equivalent* iff $\mathcal{AS}(P \cup F) = \mathcal{AS}(S \cup F)$, for any set of ground facts F, denoted by $P \equiv_u S$ [Mah88, EF03].

As one can observe from the above definitions, equivalence in ASP is often determined not only by the current semantics, but also by adding new information to the program. The relevance of this phenomenon to the research conducted within the thesis shall be discussed later on.

Two problems related to determining $\mathcal{AS}(P)$ given P are of interest for our purposes as well; namely, those of *brave* and *cautious* reasoning:

- brave reasoning – given a program P and a set Q of NAF literals, determine whether there exists an $A \in \mathcal{AS}(P)$ such that $\leftarrow Q$ is applicable under A,
- cautious reasoning – given a program P and a set Q of NAF literals, determine whether, for all $A \in \mathcal{AS}(P)$, $\leftarrow Q$ is applicable under A.

For both reasoning modes, $\leftarrow Q$ is called a *query*.

Example 2. *Consider the program from Example 1. Taking into account its answer sets, the query* $\leftarrow$ *a succeeds under brave reasoning* (*due to the answer set* $\{a\}$)*, but fails under cautious reasoning* (*due to the answer sets* $\{b\}$ *and* $\{c\}$)*.*

The significance of the reasoning problems appears in the context of algorithms for detecting and applying transformations, specifically those that use a translation to a logic program to obtain the solution – in several cases throughout this thesis, the brave and cautious reasoning facility of *DLV* [LPF+06] is used, respectively.

The description of the extension of the semantics with respect to aggregate atoms will now follow, in accordance to the definitions laid out by Dell'Armi et al. [DFI+03].

For a given symbolic set S, by a *local assignment* we understand a variable assignment ϑ_l of the local variables in S. The *grounding* of a symbolic set $S = \{\, Vars : Conj\}$ without global variables is given as the following ground set:

$$\{(\, Vars\vartheta_l : Conj\vartheta_l) : \vartheta_l \text{ is a local assignment for } S\}.$$

For S with global variables, given an assignment ϑ_g only for its global variables (a *global variable assignment*), the grounding is constructed by first setting the global variable values in correspondence to ϑ_g, and then proceeding in the way described in the preceding paragraph. The *value* of a ground set S under an interpretation I is the multiset

$$\{\{t_1 : (t_1, \ldots, t_n) \in S_I\}\},$$

where

$$S_I := \{(t_1, \ldots, t_n) : (t_1, \ldots, t_n : Conj) \in S \text{ and } Conj \text{ is true with respect to I } \}\ .$$

The value of a ground aggregate function fS under I is the result of applying f to the value of S under I.

A ground aggregate atom $A = fS \prec n$ is *true* if both of the following holds:

- the value of S under I is in the domain of f, and
- the relationship between the value of $f(S)$ under I and n, with respect to $\prec$, holds.

If A is not true, then it is *false*.

The definitions of applicability, models, answer sets, etc., are analogous to the ones applying to the semantics of disjunctive programs without aggregates.

In this thesis, only one aggregate function is utilized: *#count*. Given the set $\mathcal{C}$ of constants, the domain of *#count* is the set of all possible multisets generated from $\mathcal{C}$, and it returns the cardinality of its argument.

There is one final, unassociated definition that needs to be clarified: a *renaming* is a bijective function assigning a symbol from a given set of variables (or constants) to a set of variables (or constants).

Chapter 3

A basic overview of equivalence-preserving transformations

3.1 General definitions

Research in the field of program transformations has been conducted since at least 14 years from the time of writing of this thesis [BD95], and generally falls into two categories. Firstly, such transformations are utilized to assist in interpreting the semantics of logic programs [BZF96]. The second category of literature concentrates chiefly on simplification of programs [ONG01, EFT$^+$06, CPV07, Hei07, ONA01, Pea04]. It is the latter that will serve as the focus of this chapter. However, note that the two sets of articles overlap, as some simplification remains often a desired process in semantic interpretation.

Regardless of the different uses listed above, the authors of the publications mentioned above usually want the found transformations to possess some useful property, specifically to preserve the semantics of the original program in some way. Most attention has been given to uniform and strong equivalence, as such transformations would have more use in the context of, for example, refactoring of ASP programs.

Now, in order to correctly approach the process of examination of the intermediary code, it is necessary to restrict the attention only to transformations which preserve the level of equivalence necessary in the analyzed case. However, the choice in this situation is actually only virtual. This is because all knowledge is not contained in the ASP program *per se*, but in the contents of the problem specification provided to a framework utilizing an ASP solver. The intermediary ASP code forms only a semantically equivalent representation of the original knowledge, and it is regenerated every time said knowledge becomes modified. Therefore, it follows that here one can adopt the most general type of equivalence, i.e., ordinary equivalence.

This provides an advantage, because it allows for a larger set of optimizations to be considered. Indeed, in the literature, there exist cases of transformations where only and exclusively ordinary equivalence is preserved.

During research for this thesis, two approaches for defining (formalizing) transformations of logic programs have been found. The first, utilized extensively, considers a transformation as nothing else but a relation from the set of ASP programs to itself (sometimes, with subclass restrictions). This straightforward variant possesses two advantages apart from its simplicity:

- one can easily introduce in the context such concepts as transitive closures, useful e.g., when defining end results of transformations, and
- the creation of non-deterministic transformations is straightforward.

Another way to define transformations has been provided by Eiter et al. [EFT$^+$06]. Here, the concept of a specific transformation instance (a *replacement*) and a generalized transformation scheme (a *replacement schema*) have been differentiated and strictly formalized. This allows for a simpler definition of properties and the creation of appropriate proofs, especially in the context of semantic equivalence.

On the other hand, the level of sophistication contained in the definitions given by Eiter et al. [EFT$^+$06] is more than needed for this thesis. Therefore, it was decided to utilize the simplified notation variant.

Definition 1. *A* transformation *is a binary relation on the set of all possible programs.*

A transformation T is applicable *to a program P, or P is* eligible *for T, iff $(P, P') \in T$, where P' is some program. An element $(P, P') \in T$ is also called an* application *of T.*

A transformation T is $\equiv_X$-preserving *iff, for all $(P, P') \in T$, $P' \equiv_X P$.*

We also call rules $r_1, \ldots, r_n$ eligible for a transformation T iff there exists some $(P, P') \in T$ such that $r_1, \ldots, r_n \in P$ and P' results from P by replacing $r_1, \ldots, r_n$ by rules $r'_1, \ldots, r'_m$.

Definition 2. *A* transformation system *is a set of transformations.*

The two concepts will be required later on, when discussing the choice of transformations and their interactions.

For the purpose of displaying multiple sequences efficiently, transformations are presented in the form of trees. Each node in such a tree denotes a program, and each vertex is labeled with a transformation. Each successor node, then, denotes the program of the predecessor node changed by the transformation specified in the vertex label.

Additionally, a vertex label may be suffixed with "(n/a)". This signifies a non-applicable transformation that has been displayed regardless because the information it conveys is crucial to the understanding of the accompanying text. In such a case, the successor and predecessor node are the same programs.

3.2 Specific transformations

In what follows, we present an overview of transformations found in the literature. The formalization of transformations (replacements schemata) is done according to the definitions proposed in the preceding section.

Since the "real-life" programs we use for the examination of the transformations are finite-domain only, this thesis requires a consideration of a finite domain exclusively, which, as shown by Fink et al. [FPTW07], puts forth different requirements than an infinite one. Therefore, only those transformations which possess a finite-domain version should be considered for the comparison later on.

3.2.1 *FD-TAUT*, *FD-CONTRA*, and *FD-LC$_{0\text{-}1\text{-}0}$*

Definition 3 ([EFT+06]). *FD-TAUT allows to eliminate a rule of a program P providing that, for every $\vartheta : \mathcal{V}_r \to \mathcal{C}$, $H(r\vartheta) \cap B^+(r\vartheta) \neq \emptyset$.*[1]

Definition 4 ([EFT+06]). *FD-CONTRA allows to eliminate a rule of a program P providing that, for every $\vartheta : \mathcal{V}_r \to \mathcal{C}$, $B^+(r\vartheta) \cap B^-(r\vartheta) \neq \emptyset$.*

Definition 5 ([EFT+06, LC07]). *FD-LC$_{0\text{-}1\text{-}0}$ allows to eliminate a rule of a program P providing that, for every $\vartheta : \mathcal{V}_r \to \mathcal{C}$,*

$$B^+(r\vartheta) \cap (H(r\vartheta) \cup B^-(r\vartheta)) \neq \emptyset. \tag{3.1}$$

As shown by Eiter et al. [EFT+06], *FD-LC$_{0\text{-}1\text{-}0}$* encompasses both *FD-TAUT* and *FD-CONTRA* (i.e., all programs or rules eligible for the last two are also eligible for the former). Additionally, *FD-LC$_{0\text{-}1\text{-}0}$* is $\equiv_s$-preserving, as demonstrated by Lin and Chen [LC07].

Informally speaking, the rules which can be removed by these three transformations provide no additional knowledge compared to the remaining rules of the program they are contained in. Specifically, there exists a strong relationship between their applicability and application under any interpretation.

Example 3. *Consider the rule*

$$r = p(X) \leftarrow p(X) \ .$$

It is obvious that this rule is applicable if and only if it is applied. Hence, it has absolutely no impact on the answer sets of any program containing it. This type of rule is a tautology, and may be removed both by FD-TAUT and FD-LC$_{0\text{-}1\text{-}0}$.

[1] *FD* stands for "finite domain".

Example 4. *Consider the rule*

$$r = g(X) \leftarrow p(X), \text{not}\, p(X)\ .$$

Analogously to the previous example, this rule is not applicable if and only if it is not applied, and therefore again cannot influence a program's answer sets. This type of rule may be removed by both FD-CONTRA and FD-LC$_{0\text{-}1\text{-}0}$.

An interesting fact to note is the synergistic property of *FD-LC$_{0\text{-}1\text{-}0}$*. Not only can it be applied to any rule that *FD-TAUT* or *FD-CONTRA* are applicable to, it also detects an additional redundancy-producing class.

Example 5. *Consider the rule*

$$r = p(1) \leftarrow p(X), q(X), \text{not}\, q(0)$$

and the domain $\mathcal{C}_r = \{0, 1\}$. The following groundings can be obtained, using all possible variable substitutions:

$$p(1) \leftarrow p(0), q(0), \text{not}\, q(0);$$
$$p(1) \leftarrow p(1), q(1), \text{not}\, q(0)\ .$$

Clearly, both rules satisfy Condition (3.1) in the proviso of FD-LC$_{0\text{-}1\text{-}0}$, and so FD-LC$_{0\text{-}1\text{-}0}$ is applicable to r. However, note that one could not attain this result by checking p and q separately, and by extension, neither FD-TAUT nor FD-CONTRA can be applied to r autonomously.

3.2.2 *RED$^-$*, *S-IMPL*, *NONMIN*, and *SUB*

Definition 6 ([EFT$^+$06])**.** *RED$^-$ allows to eliminate a rule r in a program P providing there is a rule s for which $H(s) \subseteq B^-(r)$ and $B(s) = \emptyset$ holds.*

Definition 7 ([EFT$^+$06])**.** *S-IMPL allows to remove a rule r in a program P providing there exists a rule $s \in P$ and an $A \subseteq B^-(r)$ such that for every $\vartheta : \mathcal{V}_P \to \mathcal{V}_r \cup \mathcal{C}$:*

1. *$B^+(s\vartheta) \subseteq B^+(r\vartheta)$,*
2. *$H(s\vartheta) \subseteq H(r\vartheta) \cup A\vartheta$, and*
3. *$B^-(s\vartheta) \subseteq B^-(r\vartheta) \setminus A\vartheta$.*

Definition 8 ([EFT$^+$06])**.** *NONMIN allows to remove a rule r in a program P providing there exists a rule $s \in P$ such that for every $\vartheta : \mathcal{V}_P \to \mathcal{V}_r \cup \mathcal{C}$:*

1. *$B^+(s\vartheta) \subseteq B^+(r\vartheta)$ and*
2. *$H(s\vartheta) \subseteq H(r\vartheta)$.*

Definition 9 ([EFT$^+$06])**.** *SUB allows to remove a rule r in a program P providing there exists a rule $s \in P$ such that for every $\vartheta : \mathcal{V}_P \to \mathcal{V}_r \cup \mathcal{C}$:*

1. $B^+(s\vartheta) \subseteq B^+(r\vartheta)$,
2. $H(s\vartheta) \subseteq H(r) \cup B^-(r\vartheta)$, *and*
3. $B^-(s\vartheta) \subseteq B^-(r\vartheta)$.

We say that a rule s satisfying the condition in the above definition *subsumes* rule r.

SUB is applicable whenever any of the transformations *RED*$^-$, *S-IMPL*, or *NONMIN* are applicable. Additionally, all the above transformations preserve strong equivalence [EFT$^+$06].

Informally, *SUB* permits to eliminate those rules that are either

- "subsumed" by another rule – meaning that, even though they may be applied in a model, any alternatives to those in the head of the "subsumer" rule would cause that such a model would fail the minimality criterion, or
- "precluded" by another rule – this may happen if the rule is applicable whenever another rule is applicable, the latter having an atom in the head which is also contained in the head of the former, leading to a situation where the eligible rule may never be applicable.[2]

Example 6. *Consider a program consisting of the rules*

$$\begin{aligned} &r(X) \leftarrow s(X), \\ &r(X) \vee g(X) \leftarrow s(X), l(X), \\ &m(x) \leftarrow s(X), \mathrm{not}\, r(X) \end{aligned}$$

over the domain $\mathcal{C} = \{0, 1\}$. Then, after exhaustively applying SUB, only the rule

$$r(X) \leftarrow s(X)$$

would be left. Also, the second rule could be removed by NONMIN, and the third one by S-IMPL.

[2] Of course, the other head atoms of the "precluding" rule must be contained either in the head or also in the negative body of the "precluded" rule.

3.2.3 *DSuc* and variants

Definition 10 ([ONG01])**.** *DSuc allows to remove a ground atom a from each rule that contains it in its positive body if there exists a fact of the form $a \leftarrow$ in the program.*

A possible variation of *DSuc* is the following transformation:

Definition 11. *FD-DSuc allows to remove an atom l from any rule in a program P that contains it in its positive body, if, for all $\vartheta : \mathcal{V}_r \rightarrow \mathcal{C}$, there exists a ground fact of the form $a \leftarrow$ in P such that $l\vartheta = a$, as long as the removal of l would not cause r to become non-safe.*

A similar schema to *DSuc* and *FD-DSuc* could be constructed for eliminating rules that contain a default negated body literal corresponding to a fact (or a set of facts, in case of non-ground literals). However, this would really just be a subcase of *SUB*.

A problem with *DSuc* and *FD-DSuc* exists, however. For positive programs, it collapses the entire structure of such programs and yields the set of facts corresponding to the minimal model. This is not desired for program analysis.

Example 7. *Consider a program consisting of the rules*

$$\begin{aligned} &r(1) \leftarrow, \\ &g(1) \leftarrow, \text{ and} \\ &h(X) \vee s(X) \leftarrow r(X), g(X) \end{aligned}$$

over the domain $\mathcal{C} = \{1\}$. While applying DSuc, the first two rules remain unchanged, while the third rule turns to either

$$h(X) \vee s(X) \leftarrow r(X)$$

or

$$h(X) \vee s(X) \leftarrow g(X) \ .$$

Take note that $r(X)$ and $g(X)$ cannot be both removed by DSuc as this would violate the safety conditions; additionally, the obtained program is somewhat shorter, and this type of simplification makes structural analysis somewhat easier.

3.2.4 RED^+ and *FOLD*

Definition 12 ([EFT$^+$06])**.** *RED^+ allows to convert a rule r to a rule t of the form $H(r) = H(t)$, $B(r) = B(t) \cup \{\text{not}\, a\}$ providing, for every $u \in P$, every $b \in H(u)$, every $\vartheta_a : V_a \rightarrow C$, and every $\vartheta_b : V_b \rightarrow C$, it holds that $a\vartheta_a \neq b\vartheta_b$.*

The peculiarity of RED^+ among the transformations discussed so far is that it preserves only ordinary equivalence [EFT$^+$06]. Also, it is one of the few transformations that specifically remove literals.

Example 8. *Consider the simple program*

$$P = \{c(1) \leftarrow;\ a(X) \vee b(X) \leftarrow c(X), \text{not}\, d(X)\}$$

under the domain $\mathcal{C} = \{1\}$.

Here is the result of applying RED^+*:*

$$P' = \{c(1) \leftarrow; a(X) \vee b(X) \leftarrow c(X), \text{not}\, d(X)\}\ .$$

The lack of the ability to preserve even uniform equivalence is quite apparent in this example. It becomes obvious with a simple addition of the fact $d(1) \leftarrow$*. Then,*

$$\mathcal{AS}(P \cup \{d(1)\}) = \{\{c(1)\}\}$$

but

$$\mathcal{AS}(P' \cup \{d(1)\}) = \{\{c(1), a(1)\}, \{c(1), b(1)\}\}\ .$$

Definition 13 ([EFT$^+$06])**.** *FOLD allows to convert two rules* r *and* s *into a single rule* t *providing there exists a renaming* δ *and an atom* $a \in B^-(r\delta) \cap B^+(s)$ *for which*

- $H(r\delta) = H(s) = H(t)$,
- $B^+(r\delta) = B^+(s) \setminus \{a\} = B^+(t)$,
- $B^-(r\delta) \setminus \{a\} = B^-(s) = B^-(t)$,

and, for every $u \in P$*, every* $b \in H(u)$*, every* $\vartheta_a : V_a \to C$*, and* $\vartheta_b : V_b \to C$*, it holds that* $a\vartheta_a \neq b\vartheta_b$.

FOLD preserves uniform equivalence, as shown by Eiter et al. [EFT$^+$06].

Example 9. *Consider the program consisting of the rules*

$$\begin{aligned} &a(X) \vee b(X) \leftarrow c(X), \text{not}\, d(X),\\ &a(X) \vee b(X) \leftarrow c(X), \text{not}\, d(X),\\ &g(X) \leftarrow k(X), b(X),\ \textit{and}\\ &g(X) \leftarrow k(X), \text{not}\, b(X) \end{aligned}$$

over the domain $\mathcal{C} = \{0, 1\}$*. Using FOLD, the first two rules may be combined to*

$$a(X) \vee b(X) \leftarrow c(X)\ .$$

However, the final two rules may not be changed by applying FOLD, as the atom a *from the definition in this case is* $a(X)$*, being also contained in the head of the other rules with respect to grounding.*

3.2.5 *FAILURE*

Definition 14 ([ONG01])**.** *FAILURE allows to remove any rule which in its positive body contains an atom a which is not contained in a head of any rule in the program.*

A finite-domain version of *FAILURE* can be constructed in analogy to *FD-DSuc*. The transformation, in general, preserves only ordinary equivalence [ONG01] .

Example 10. *Consider the following program:*

$$P = \{g \leftarrow; f \leftarrow h\} \ .$$

After applying FAILURE, the following program is obtained:

$$P = \{g \leftarrow\} \ .$$

The lack of uniform (and strong) equivalence preservation may be demonstrated by appending $h \leftarrow$ to each program. Then,

$$\mathcal{AS}(P \cup \{h\}) = \{\{g, f, h\}\}$$

but

$$\mathcal{AS}(P' \cup \{h\}) = \{\{g, h\}\} \ .$$

FAILURE could be interesting in the context of searching for redundant rules.

3.2.6 *LSH*

Definition 15 ([EFT$^+$06])**.** *LSH allows to remove a rule r if r is head-cycle-free in P and, for all $\mathcal{V}_r \to \mathcal{C}, |H(r\vartheta)| = |H(r)| > 1$, replacing it with a set N_r defined as follows:*

$$N_r = \{h \leftarrow B(r), \text{not } H(r) \setminus h \mid h \in H(r)\}.$$

Unlike the other transformations presented, *LSH* performs no reduction and actually increases the number of rules in a program after its application. Additionally, it has been shown to preserve uniform equivalence [EFT$^+$06].

As a matter of fact, we would dispense with this means of removing disjunction since it was shown by Eiter et al. [EFTW04a] that there *always* exists a uniformly (and an ordinarily) equivalent normal program to a given disjunctive one. However, the context of the results obtained in that paper is incompatible with the one in this thesis, as we consider finite domains only.

Example 11. *Consider the program consisting of the rules*

$$a(1) \vee b(1) \leftarrow,$$
$$b(1) \vee c(1) \leftarrow, \text{ and}$$
$$g(X) \vee h(X) \leftarrow k(X)$$

over the domain $\mathcal{C} = \{1\}$*. After using LSH exhaustively, the resulting program contains the following rules:*

$$a(1) \vee b(1) \leftarrow,$$
$$b(1) \vee c(1) \leftarrow,$$
$$g(X) \leftarrow k(X), \text{not } h(X),$$
$$h(X) \leftarrow k(X), \text{not } g(X) \ .$$

The first two rules cannot be modified by LSH, because they share a head atom (namely $b(1)$*), and so are not head-cycle-free in the program. There exists no such problem with the final rule, however, and it becomes converted into two new rules. Finally, note how LSH is unable to render a safe program non-safe – it transfers atoms exclusively between the head and the negative body.*

3.2.7 Miscellaneous other transformations

What follows is a short description of some remaining, potentially interesting transformations, compressed into a summary form:

- *SUPRA* [EFTW04b] – if a fact classically follows from a program, add it to the program. This transformation only preserves ordinary equivalence and may theoretically lead to the applicability of other transformation schemata. The main problem of *SUPRA* is its potential for the same structural destruction as *(FD-)DSuc*.
- *SRR*$\mathcal{E}$, for $\mathcal{E} \in \{O, U, S\}$, the semantic removal of redundant rules under ordinary, uniform, and strong equivalence, respectively [EFTW04b] – simply check whether $P \setminus \{r\} \equiv_e P$, where $e \in \{o, u, s\}$. These rules would be useful to implement, especially in order to compare their effectiveness with syntactic transformations.
- *SRL*$\mathcal{E}$, for $\mathcal{E} \in \{O, U, S\}$, deals with the semantic removal of redundant literals [EFTW04b]. As for the previous rules, analogous observations can be made, with similar utility.
- *GPPE* and *WGPPE* [EFTW04b] – the first allows to remove a rule, while both add several rules in its place. These transformations may cause an exponential explosion of the program. Nevertheless, they are interesting to analyze since they can make other replacement schemata applicable. However, both are only available for the ground case.

- *WGPPE*$^{\oplus}$ and *WGPPE*$^{\ominus}$ [Hei07] – these are non-ground variants of the two previous transformations. Since they will be discussed further, their full definitions will be provided in what follows.

Definition 16 ([Hei07])**.** *For a program P, a domain $\mathcal{C}$, a rule $r \in P$, and an atom $a \in B^+(r)$, WGPPE$^{\oplus}$ allows to transform P into*

$$P \cup \{H(r\theta) \cup (H(r'\theta') \setminus \{a\theta\}) \leftarrow (B^+(r\theta) \setminus \{a\theta\}) \cup B(r'\theta'), \text{not } B^-(r'\theta') : (r, \theta, \theta') \in Q_{\mathcal{C}}\}$$

provided that

$$Q_{\mathcal{C}} \subseteq \{(r, \theta, \theta') : \textit{there is some } b \in H(r) \textit{ such that } a\theta = b\theta', \textit{ for some groundings } \theta, \theta' \textit{ over } \mathcal{C}\}.$$

Definition 17 ([Hei07])**.** *For a program P, a domain $\mathcal{C}$, a rule $r \in P$, and an atom $a \in B^+(r)$, WGPPE$^{\ominus}$ allows to transform P into*

$$P \cup \{H(r'\theta') \setminus \{a\theta\} \leftarrow (B^+(r\theta) \setminus \{a\theta\}) \cup B^+(r'\theta'), \text{not}(B^-(r\theta) \cup H(r\theta) \cup B^-(r'\theta') \setminus \{a\theta'\}) : (r, \theta, \theta') \in Q_{\mathcal{C}}\}$$

provided that

$$Q_{\mathcal{C}} \subseteq \{(r, \theta, \theta') : \textit{there is some } b \in B^-(r) \textit{ such that } a\theta = b\theta', \textit{ for some groundings } \theta, \theta' \textit{ over } \mathcal{C}\}.$$

Chapter 4

Detecting transformation eligibility

4.1 General discussion

We now discuss methods for detecting the applicability of transformations and specifying a suitable transformation system. Several considerations should be taken into account. Of course, transformations that are trivially subsumed by other transformations are completely disregarded as a preliminary measure.

Availability of existing algorithms for determining applicability certainly is a first. Various ones already fulfill this condition but others require further development. However, for some transformations, e.g., for $FD\text{-}LC_{0\text{-}1\text{-}0}$, a solution is easy, as shown further on.

The performance of applicability testing is an important factor as well. For example, while the test for $FD\text{-}LC_{0\text{-}1\text{-}0}$ only requires a check of all literals within a single rule, the one for *SUB* requires an analysis of the entire program to verify the applicability for any candidate.

Furthermore, consequences of interaction with other transformations need to be considered. As an illustration, note that if a rule gets removed by, e.g., $FD\text{-}LC_{0\text{-}1\text{-}0}$, all its head literals cease to exist in the context of the program, and thus this transformation may enable the application of *FOLD*, among others.

Finally, it is advantageous to prioritize those transformation that provide "stronger" equivalence preservation. To preserve strong or even uniform equivalence may prove more beneficial than only preserving ordinary equivalence, mainly because more of the structure may remain intact. In the most extreme case – that is, if a program has no answer sets to begin with –, transformations preserving ordinary equivalence will reduce the program to the empty set. Conversely, using transformations preserving ordinary equivalence may provide an advantage of a faster check and of more room for improvement, as those often subsume one preserving strong or uniform equivalence.

With all the factors in mind, it remains to select the transformations included in the system used for our subsequent experiments.

With regard to the availability of algorithms, *SUB* and *LSH* have theirs defined explicitly [Tra06], whereas *FOLD* is handled implicitly in a reduction proof [EFT+06]. Not only that, but all of these transformations preserve strong equivalence, and they are, for the most part, unique in their effect. Therefore, it was decided to include them in the system.

As already mentioned, another transformation that preserves strong equivalence and would be easy to introduce into the system is *FD-LC*$_{0\text{-}1\text{-}0}$, in view of its dependence on a very simple precondition, implying that every rule may be analyzed in isolation. Due to these facts, it will be made a part of the system as well.

Now, adding *DSuc*, *FAILURE*, and *SUPRA* into the mix may cause some problems. As already mentioned, they either only preserve ordinary equivalence or may destroy the structure of the program. An analogous situation occurs with RED^+. This problem disqualifies those transformations from the final system.

We continue with $WGPPE^{\oplus}$ and $WGPPE^{\ominus}$. Their advantage could lie in the already noted theoretical ability to enable other transformations. However, $WGPPE^{\oplus}$ and $WGPPE^{\ominus}$ not only cause a potentially large increase in the number of rules in the program, they also ground each rule they are applied to. This hinders any attempts at structural analysis. More specifically, of the transformations already selected (apart from the semantic ones), it is readily apparent that the application of either $WGPPE^{\oplus}$ or $WGPPE^{\ominus}$ would be of little advantage: *FD-LC*$_{0\text{-}1\text{-}0}$ would not be affected at all, *SUB* would be enabled only in some specific (and possibly artificial) cases; the same applies to *LSH* and *FOLD*. Furthermore, $WGPPE^{\oplus}$ and $WGPPE^{\ominus}$ are somewhat "non-compatible" with the the other transformations – the two generate ground rules, which the other ones cannot efficiently process. This can be best illustrated with an example:

Example 12. *Consider the program*

$$P = \{g(X) \leftarrow h(X), c(X), \text{not}\, e(X);\\ h(X) \leftarrow k(X), e(X)\}.$$

After applying $WGPPE^{\oplus}$ *in the domain* $\{1, 2, 3\}$ *we obtain the following result:*

$$P' = P \cup \{g(1) \leftarrow c(1), k(1), e(1);\\ g(2) \leftarrow c(2), k(2), e(2);\\ g(3) \leftarrow c(3), k(3), e(3)\}.$$

Not only is the resulting program longer than the original, it is also ground, and therefore less suitable for applying a number of other transformations. Therefore, it is best to leave $WGPPE^{\oplus}$ and $WGPPE^{\ominus}$ out of our area of interest, especially since no algorithm for them has been created to date.

To summarize,

- we use $\{FD\text{-}LC_{0\text{-}1\text{-}0}, SUB, FOLD, LSH\}$ as our primary transformation system, and
- treat *SRRS* and *SRLS* as alternative transformations, to be applied in a separate run.

4.2 *SUB* eligibility detection

4.2.1 Preprocessing

SUB is a difficult transformation to analyze, in the sense that it requires a comparison between each and every rule, with the naive approach having an average-, best-, and worst-case execution time of $O(|P|^2)$. Therefore, it may be prudent to devise a preprocessing step in order to improve performance. For this, the following algorithm has been created.

Definition 18. *$SUBPREPROP(r, s, \mathcal{C}, alg)$ denotes the following algorithm, where r and s are rules, $\mathcal{C}$ is a finite domain, and alg signifies an algorithm for solving SUB, whose arguments comprise r, s, and $\mathcal{C}$:*

1. *If one of the following conditions holds:*
   - $|B^+(r)| < |B^+(s)|$,
   - $|H(r)| + |B^-(r)| < |H(s)|$,
   - $|B^-(r)| < |B^-(s)|$, *or*
   - $\mathcal{R}_s \not\subseteq \mathcal{R}_r$,

   return false.
2. *Otherwise, return $alg(r, s, \mathcal{C})$.*

Theorem 1. *If alg is correct and complete, then so is SUBPREPROP.*

Proof. Assume that *alg* is correct and complete but *SUBPREPROP* is not.

First of all, we note that the first step of *SUBPREPROP* always terminates since we deal with finite rules only. Hence, *SUBPREPROP* terminates iff *alg* terminates.

We distinguish the following cases:

1. Suppose there are rules r and s which are eligible for *SUB* but *SUBPREPROP* returns *false*. This may only happen if *alg* returns *false*. But by the completeness of *alg* it must hold that r and s are not eligible for *SUB*. This is in violation of our assumption and thus whenever rules are eligible for *SUB*, *SUBPREPROP* must return *true* whenever it terminates.

2. Suppose now that there are rules r and s which are not eligible for SUB but $SUBPREPROP$ returns *true*. By construction of $SUBPREPROP$, this means that *alg* returns *true*. Since *alg* is correct, it follows that r and s must be eligible for SUB, a contradiction. Therefore, if rules are not eligible for SUB, $SUBPREPROP$ returns *false* or does not terminate.

□

Of course, even with $SUBPREPROP$, the worst-case time for comparison still reaches $O(|P|^2)$. However, this reduction might quite possibly translate into faster execution during practical usage.

4.2.2 Imperative approach

We first define a straightforward imperative algorithm for solving SUB.

Definition 19. *$SUBIMP(r, s, \mathcal{C})$ is defined as follows, where r and s are rules, and $\mathcal{C}$ is a finite set of constants:*

1. *For each $\vartheta : \mathcal{V}_r \cup \mathcal{V}_s \to \mathcal{C}$, if one of the following conditions does not hold, then return false:*
   - $B^+(s\vartheta) \subseteq B^+(r\vartheta)$,
   - $H(s\vartheta) \subseteq H(r) \cup B^-(r\vartheta)$, *or*
   - $B^-(s\vartheta) \subseteq B^-(r\vartheta)$.
2. *Otherwise, return true.*

Theorem 2. *For any rule r and any finite set $\mathcal{C}$ of constants, $SUBIMP(r, s, \mathcal{C})$ returns true for some rule s iff r is eligible for SUB.*

Proof. This follows directly from the definitions of $SUBIMP$ and SUB. □

This algorithm might seem inefficient at first glance, however it possesses certain advantages. First and foremost, there is a large potential for implementation-specific optimization, such as caching. Not only that, but the algorithm is intended to run with $SUBPREPROP$, possibly significantly cutting down the execution time.

4.2.3 Declarative approach

This approach uses the algorithm presented by Traxler [Tra06]. The basis of the eligibility testing is a reduction to the *Boolean query problem* (BCQ), which is the following task: given a set F of facts and a rule t of the form

$$a \leftarrow b_1, ..., b_n,$$

where $a, b_1, ..., b_n$ are propositional atoms, decide whether there exists a stable model of $F \cup \{t\}$ containing a.

The algorithm itself has the following form, given rules r and s:

1. Replace every symbol in $|B^+(s) \cup B^+(r)| \cap |H(s) \cup (H(r) \cup B^-(r))|$ in such a way that the cardinality of the new set $|B^+(s) \cup B^+(r)|$ is unchanged.
2. Replace every symbol in $|B^-(s) \cup B^-(r)| \cap |H(s) \cup (H(r) \cup B^-(r)) \cup B^+(s) \cup B^+(r)|$ in such a way that the cardinality of the new set $|B^-(s) \cup B^-(r)|$ is unchanged.
3. Define:
   - $A := H(s) \cup B^+(s) \cup B^-(s)$ and
   - $B := (H(r) \cup B^-(r)) \cup B^+(r) \cup B^-(r)$.
4. Replace every variable in B by a new unique constant symbol not occurring in $A \cup B$.
5. Generate a propositional atom b not occurring in $A \cup B$.
6. Solve the BCQ problem for $t = b \leftarrow A$ and $F = B$. If the result is positive, return *true*, else return *false*.

As shown by Traxler [Tra06], the algorithm returns *true* if s subsumes r, and *false* otherwise.

4.3 *LSH* eligibility detection and modification

To solve the problem of testing whether *LSH* may be applied to a given rule, we can again make use of an algorithm due to Traxler [Tra06], defined next.

Given a program P and rule $r \in P$, perform the following steps:

1. Let C_P be the set of constants in P and create a new set D such that the cardinality of D is four times greater than the cardinality of C_P.
2. Generate a relational symbol p not occurring in P.
3. Define $A := \{p(x) : x \in D\}$.

4. Define

$$B := \bigcup_{r \in P} \{\{b \leftarrow a, \{p(x) : x \in V_b\} : a \in H(r), b \in B^+(r)\} : H(r) \neq \emptyset, B^+(r) \neq \emptyset\}.$$

5. For all $s, l \in H(r')$, where r' is a grounding of r with respect to to D:
   - If $s = l$ or else l is a brave reasoning consequence of $A \cup B \cup \{s\}$ and s is a brave reasoning consequence of $A \cup B \cup \{l\}$, return *false*.
   - Otherwise, return *true*.

The algorithm returns *true* if r is eligible for LSH in P, and *false* otherwise.

4.4 *FD-LC$_{0\text{-}1\text{-}0}$* eligibility detection

Before we dwell into the subject of testing *FD-LC$_{0\text{-}1\text{-}0}$* eligibility, some general observations, including those on potential pitfalls, should be made. First of all, unlike its infinite-domain pendant, *FD-LC$_{0\text{-}1\text{-}0}$* has a more complicated verification procedure, in that one must ground the rule in order to complete the check – specifically, one must produce all groundings of that rule with respect to the domain.

Secondly, it is necessary to analyze the groundings of the entire rule and not just of specific predicates, as demonstrated by Example 5. This induces yet another increase of the overhead.

Finally, if we derive the domain from the data supplied, one must be aware of avoiding concentrating solely on the constants within the examined rule. Instead, the extracted domain should be composed of all the constants in the input program.

The three considerations listed above provide necessary hints which have shaped the development of the two algorithms described below.

4.4.1 Imperative Approach

In the first approach, we are concerned with checking all possible rule groundings in order to test *FD-LC$_{0\text{-}1\text{-}0}$* applicability. Of course, explicitly testing *all* groundings of the rule under a given domain is unnecessary, and one can make several improvements. One of these is the observation that only literals with predicate symbols that occur both in $H(r) \cup B^-(r)$ and $B^+(r)$ are relevant. Hence, the algorithm will need to consider subsets of the two sets only.[1] We use the following definition to simplify the algorithm.

[1] Of course, the same improvement may be applied to the declarative approach, described below.

Definition 20. *Let S_1 and S_2 be sets of atoms. Then, $Com(S_1, S_2)$ is given by*

$$\{a : a \in S_1 \text{ and there exists an } a_2 \in S_2 \text{ such that } |a_2| = |a|\}.$$

Furthermore, the following can be used to reduce the required grounding work:

Lemma 1. *Let S_1 and S_2 be two sets of atoms and C a set of constants. Assume there exists a predicate symbol $p \in \mathcal{R}_{S_1 \cup S_2}$ and some i, $1 \leq i \leq n(p)$, such that the i-th variable of every atom $a \in S_1 \cup S_2$ with $|a| = p$ is X, for some variable X. Then, for every $\vartheta : C \cup V_{S_1 \cup S_2} \to C$,*

$$\text{if } \overline{\Gamma}^i(p, \vartheta, S_1) \cap \overline{\Gamma}^i(p, \vartheta, S_2) \neq \emptyset, \text{ then } \Gamma(p, \vartheta, S_1) \cap \Gamma(p, \vartheta, S_2) \neq \emptyset,$$

where

$$\overline{\Gamma}^i(p, \vartheta, S_j) := \{\bar{p}(x_1\vartheta, ..., x_{i-1}\vartheta, x_{i+1}\vartheta, ..., x_m\vartheta) : \\ x_1, ..., x_{i-1}, x_{i+1}, ..., x_m \text{ are variables with } p(x_1, ..., x_m) \in S_j \text{ for some } x_i\}$$

and

$$\Gamma(p, \vartheta, S_j) := \{p(x_1\vartheta, ..., x_m\vartheta) : x_1, ..., x_m \text{ are variables with } p(x_1, ..., x_m) \in S_j\},$$

for $i = 1, 2$, $m = n(p)$, and $\bar{p}$ is a new predicate symbol of arity $m - 1$.

Proof. Let $p \in \mathcal{R}_{S_1 \cup S_2}$ be a predicate symbol such that the ith variable of every atom $a \in S_1 \cup S_2$ with $|a| = p$ is X, for $1 \leq i \leq n(p)$. Assume there exists a $\vartheta : C \cup V_{S_1 \cup S_2} \to C$ such that $\overline{\Gamma}^i(p, \vartheta, S_1) \cap \overline{\Gamma}^i(p, \vartheta, S_2) \neq \emptyset$ but $\Gamma(p, \vartheta, S_1) \cap \Gamma(p, \vartheta, S_2) = \emptyset$. Hence, there must be variables $x_1, \ldots, x_m, x'_1, \ldots, x'_m$ such that

$$\bar{p}(x_1\vartheta, ..., x_{i-1}\vartheta, x_{i+1}\vartheta, ..., x_n\vartheta) = \bar{p}(x'_1\vartheta, ..., x'_{i-1}\vartheta, x'_{i+1}\vartheta, ..., x'_n\vartheta) \tag{4.1}$$

but

$$p(x_1\vartheta, ..., x_n\vartheta) \neq p(x'_1\vartheta, ..., x'_n\vartheta). \tag{4.2}$$

From (4.1), it follows that

$$x_1\vartheta = x'_1\vartheta, ..., x_{i-1}\vartheta = x'_{x-1}\vartheta, x_{i+1}\vartheta = x'_{x+1}\vartheta, ..., x_n\vartheta = x'_n\vartheta.$$

But by (4.2) it can only be that $x_i\vartheta \neq x'_i\vartheta$. Since by hypothesis both $x_i = X$ and $x'_i = X$, we arrive at a contradiction. □

Intuitively, the above lemma allows for "cutting out" the variables that would otherwise have to be fully enumerated.

The algorithm will be constructed from the two components that have been just introduced. The plan is to, firstly, minimize the amount of atoms to be analyzed with the help of *Com*, then use Lemma 1 to decrease the size of the grounding, and only then to check for all the possibilities.

The general algorithm, *LC010IMP*, is as follows:

Definition 21. *Let r be a rule and $\mathcal{C}$ a domain. Then, LC010IMP$(r, \mathcal{C})$ is constructed as follows:*

1. *Set*
   - $Q_1(r) := Com(B^+(r), H(r) \cup B^-(r))$ *and*
   - $Q_2(r) := Com(H(r) \cup B^-(r), B^+(r))$.
2. *If there exist atoms $l_1 \in Q_1(r)$ and $l_2 \in Q_2(r)$, where $l_1 = l_2 = p(c_1, ..., c_n)$, and where $n \geq 0$ and $c_1, ..., c_n$ are constants, return true.*
3. *If there exists a predicate symbol p for which all atoms a in $Q_1(r) \cup Q_2(r)$ with $|a| = p$ contain the same variable symbol at some argument position i, reduce the arity of p by removing the ith argument of every a in $Q_1(r) \cup Q_2(r)$, and go back to Step 2. Otherwise, proceed.*
4. *Construct all possible groundings on $Q_1(r)$ and $Q_2(r)$. If all groundings fulfill Condition (3.1) from the definition of FD-LC$_{0\text{-}1\text{-}0}$, return true, else return false.*

Of course, in terms of complexity, the worst-case time is only negligibly better than in a naive enumeration. However, for most practical cases, the introduced simplifications can provide a noticeable speed-up.

Theorem 3. *If LC010IMP$(r, \mathcal{C})$ returns true, then r is eligible for FD-LC$_{0\text{-}1\text{-}0}$.*

Proof. Assume that the algorithm returns true, but the rule is not eligible for *FD-LC$_{0\text{-}1\text{-}0}$*. There are two cases where the algorithm returns *true*, namely in Step 2 or in Step 4:

1. Assume *true* is returned in Step 2. Then, there exists an $l_1 \in Q_1(r)$ and an $l_2 \in Q_2(r)$ for which $l_1 = l_2 = p(c_1, ..., c_n)$. Therefore, trivially, for all $\mathcal{V}_{Q_1(r) \cup Q_2(r)} \rightarrow \mathcal{C}$,

   $$Q_1(r)\vartheta \cap Q_2(r)\vartheta \neq \emptyset.$$

   Due to the construction of $Q_1(r)$ (being a subset of $B^+(r)$) and of $Q_2(r)$ (being a subset of $H(r) \cup B^-(r)$), as well as by Lemma 1, it follows that, for all $\mathcal{V}_r \rightarrow \mathcal{C}$,

   $$B^+(r\vartheta) \cap (H(r\vartheta) \cup B^-(r\vartheta)) \neq \emptyset,$$

   regardless of the amount of reductions made in Step 3. However, since r is not eligible for *FD-LC$_{0\text{-}1\text{-}0}$* by hypothesis, there must be some $\mathcal{V}_r \rightarrow \mathcal{C}$ for which

   $$B^+(r\vartheta) \cap (H(r\vartheta) \cup B^-(r\vartheta)) = \emptyset,$$

   a contradiction.

2. Assume *true* is returned in Step 4. Then, per definition of the algorithm, for all $\mathcal{V}_{Q_1(r) \cup Q_2(r)} \rightarrow \mathcal{C}$,

   $$Q_1(r)\vartheta \cap Q_2(r)\vartheta \neq \emptyset.$$

   The contradiction can now be reached analogously to Step 2.

Hence, the contradiction is reached in both subcases, and thus for the general case as well. □

Theorem 4. *If r is eligible for FD-LC$_{0\text{-}1\text{-}0}$ in the domain $\mathcal{C}$, then LC010IMP$(r, \mathcal{C})$ returns true.*

Proof. Assume that r is eligible for *FD-LC$_{0\text{-}1\text{-}0}$* in $\mathcal{C}$ but *LC010IMP*$(r, \mathcal{C})$ returns *false*, or does not terminate.

- If any reductions are performed in Step 3, then, by Lemma 1, they have no impact on the return value of the algorithm. Therefore, the algorithm returns *false* if it fails the check in Step 4, i.e., if there exists a $\vartheta : \mathcal{V}_{Q_1(r) \cup Q_2(r)} \to \mathcal{C}$ for which

  $$Q_1(r)\vartheta \cap Q_2(r)\vartheta \neq \emptyset.$$

  Because, per definition, $Q_1(r)$ and $Q_2(r)$ contain only those predicates whose symbols are contained in both $B^+(r)$ and $H(r) \cup B^-(r)$, this implies that such a ϑ satisfies

  $$B^+(r\vartheta) \cap (H(r\vartheta) \cup B^-(r\vartheta)) = \emptyset.$$

  However, this yields a contradiction to the assumption that r is eligible for *FD-LC$_{0\text{-}1\text{-}0}$*.

- The execution of each step is bounded by

  1. the number of literals in r,
  2. the number of literals in $Q_1(r)$ and $Q_2(r)$, where each cannot, by definition of Com, be greater in size than r,
  3. the number of literals in r together with the arities of those literals, and
  4. the number of elements in $\mathcal{C}$.

  Because

  - rules, by definition, are of finite size,
  - literals, by definition, have a finite arity, and
  - supplied domains can only be finite by definition of the algorithm,

  LC010IMP must always terminate.

Therefore, the assumption leads to a contradiction in either case. □

4.4.2 Declarative approach

We now discuss an approach to reduce the problem of testing whether a rule is eligible for *FD-LC$_{0\text{-}1\text{-}0}$*to brave reasoning over the answer sets of a program.

An idea for a solution handling this issue could be to have the algorithm represent all possible groundings through answer sets of a specially crafted program. Then, produce a query to verify the *FD-LC$_{0\text{-}1\text{-}0}$* applicability condition.

A problem arises here – the query would have to ask about the existence of a common element, from potentially many alternatives. However, reasoning queries in ASP are by nature conjunctive, not disjunctive. Fortunately, there is a simple workaround for this problem. One may verify by the query whether no common element exist, and if it is true for at least one answer set (which, as already said, represents a single grounding), then *FD-LC$_{0\text{-}1\text{-}0}$* would not be applicable for the considered rule.

Definition 22. *The algorithm, LC010DEC$(r, \mathcal{C})$, where r denotes a rule and $\mathcal{C}$ a domain, has the following form:*

1. *Create set* $C := \begin{cases} \mathcal{C} & \text{if } \mathcal{C} \neq \emptyset; \\ \{c\} & \text{otherwise.} \end{cases}$
2. *Define a bijective function f_r with the domain V_r and the co-domain being the set of all possible constant symbols, excluding elements of C.*
3. *Construct the program*

$$\begin{aligned} O_r = &\{var(f_r(x)) : x \in \mathcal{V}_r\} \\ &\cup \{\bigvee_{c \in C} inter(c, X) \leftarrow var(X)\} \\ &\cup \{l \leftarrow \{inter(v, f_r(v)) : v \text{ a variable in } l\} : l \in H(r) \cup B^-(r)\}, \end{aligned}$$

   where var and inter are predicate symbols not occurring in r.

4. *Perform brave reasoning over the answer sets of O_r with the query*

$$\{inter(v, f_r(v)) : v \in \mathcal{V}_r\}, \text{not } B^+(r).$$

   If the query evaluates to false, then return true, otherwise return false.

We note that O_r provides all possible groundings of the variables in r, under $\mathcal{C}$, via the disjunctive rules generating one grounding per answer set. The query checks whether there is an answer set (corresponding to an interpretation) such that the set of the grounded atoms from $H(r) \cup B^-(r)$ is disjoint with the set of grounded atoms from $B^+(r)$. If there exists such an answer set, then the application of the query returns *true*, which means that the transformation cannot be applied. Otherwise, in each interpretation, the two sets possess at least one common element, and the transformation is applicable.

Example 13. *Applying the algorithm to the rule*

$$r_1 = p(1) \leftarrow p(X), q(X), \text{not } q(0)$$

over the domain $\{0, 1\}$ *we obtain*

$$\begin{aligned} O_r = \{&var(constForX) \leftarrow; \\ &inter(0, X) \vee inter(1, X) \leftarrow var(X); \\ &p(1) \leftarrow; \\ &q(0) \leftarrow\}, \end{aligned}$$

where $constForX$ *is a constant assigned to* X*.*

The query then is

$$inter(X, constForX), \text{not } p(X), \text{not } q(X).$$

The program possesses two answer sets:

$$\{var(constForX), p(1), q(0), inter(1, constForX)\}$$

and

$$\{var(constForX), p(1), q(0), inter(0, constForX)\}.$$

Evidently, the query yields false for both answer sets. Hence, it is also bravely false, and, according to the algorithm, the rule is eligible for the transformation.

Example 14. *Applying the algorithm to the rule*

$$r_2 = f(X, 1, Z) \leftarrow f(X, Y, Z)$$

over the domain $\{0, 1\}$*, we obtain*

$$\begin{aligned} O_r = \{&var(constForX) \leftarrow; \\ &var(constForY) \leftarrow; \\ &var(constForZ) \leftarrow; \\ &inter(0, X) \vee inter(1, X) \leftarrow var(X); \\ &f(X, 1, Z) \leftarrow inter(X, constForX), inter(Z, constForZ)\}, \end{aligned}$$

where $constForX$*,* $constForY$*, and* $constForZ$ *are constants assigned to* X*,* Y*, and* Z*, respectively.*

With the query

$$inter(X, constForX), inter(Y, constForY), inter(Z, constForZ), not f(X, Y, Z)$$

the program has eight ($= 2^3$*, three variables with two possible values each*) *answer sets:*

$$\{var(constForX), var(constForY), var(constForZ), inter(1, constForX), inter(1, constForY), inter(1, constForZ), f(1,1,1)\},$$
$$\{var(constForX), var(constForY), var(constForZ), inter(1, constForX), inter(0, constForY), inter(1, constForZ), f(1,1,1)\},$$
$$\{var(constForX), var(constForY), var(constForZ), inter(0, constForX), inter(1, constForY), inter(1, constForZ), f(0,1,1)\},$$
$$\{var(constForX), var(constForY), var(constForZ), inter(0, constForX), inter(0, constForY), inter(1, constForZ), f(0,1,1)\},$$
$$\{var(constForX), var(constForY), var(constForZ), inter(1, constForX), inter(1, constForY), inter(0, constForZ), f(1,1,0)\},$$
$$\{var(constForX), var(constForY), var(constForZ), inter(0, constForX), inter(1, constForY), inter(0, constForZ), f(0,1,0)\},$$
$$\{var(constForX), var(constForY), var(constForZ), inter(1, constForX), inter(0, constForY), inter(0, constForZ), f(1,1,0)\},$$
$$\{var(constForX), var(constForY), var(constForZ), inter(0, constForX), inter(0, constForY), inter(0, constForZ), f(0,1,0)\}.$$

The query is bravely true as evidenced by several answer sets, the first one being an example. Therefore, the examined rule is neither a tautology nor a contradiction under the given domain.

In what follows we analyze the adequacy of the algorithm.

The program always generates at least one answer set, since it is basic. Furthermore, the only rule that may induce multiple answer sets in O_r is

$$\bigvee_{c \in C} inter(c, X) \leftarrow var(X),$$

since all other rules of O_r are normal. The rule induces k answer sets for every atom with the predicate symbol var, where $k = |C|$. Hence, the total number of answer sets is equal to $k^{|V_r|} = |C|^{|V_r|}$, as there are exactly $|V_r|$ atoms with the predicate symbol var in O_r. The number corresponds to the amount of possible groundings of r under C.

Lemma 2. *For a program O_r for a given rule r, under a given nonempty domain $\mathcal{C}$, there is a one-to-one correspondence between the answer sets of O_r and the possible groundings of r under $\mathcal{C}$. More specifically,*

1. *for each grounding $\vartheta : \mathcal{V}_r \to \mathcal{C}$ of r, there exists exactly one answer set of O_r containing, for every $v \in \mathcal{V}_r$, exactly one literal of the form $inter(v\vartheta, f_r(v))$, and*

2. *for each answer set A of O_r, there exists exactly one valid grounding $\vartheta : \mathcal{V}_r \to \mathcal{C}$ such that for every*

$$inter(x, y) \in A,$$

$v\vartheta = x$ and $f_r(v) = y$, for some $v \in \mathcal{V}_r$.

Proof. Firstly, the only rules that may induce atoms with the predicate symbol *inter* are the facts in

$$\{var(f_r(x)) : x \in \mathcal{V}_r\}$$

and the rules

$$\bigvee_{c \in C} inter(c, X) \leftarrow var(X),$$

where $C = \mathcal{C}$. By definition, each answer set of O_r contains exactly one atom of the form $var(f_r(x))$, where each $f_r(x)$ corresponds to a variable in $\mathcal{V}_r$. Conditions 1 and 2 of the lemma are now shown as follows:

1. a) Assume there exists a valid grounding $\mathcal{G}_r$ but no answer set whose subset consisting of all atoms with the predicate symbol *inter* is of the form

$$\{inter(v\vartheta, f_r(v),) : v \in \mathcal{V}_r\}.$$

Then, at least one of the following two conditions must hold true:

- There exists a variable in $\mathcal{V}_r$ for which there is no corresponding *inter* atom. However, by the definition of O_r, there is always exactly one fact of the form $var(x)$ for each variable in $\mathcal{V}_r$, and this causes also to exist a corresponding atom with the predicate symbol *inter* due to the second component of O_r.
- There exists an atom with the predicate symbol *inter* with the first term being outside of C. This is not possible due to the definition of the second component of O_r.

Therefore, this assumption cannot hold.

b) Assume there exists a valid grounding $\mathcal{G}_r$ but there is more than one answer set whose subset consisting of all atoms with the predicate symbol *inter* is of the form

$$\{inter(v\vartheta, f_r(v),) : v \in \mathcal{V}_r\}.$$

From the definition of the algorithm, every answer set generated, and hence the two or more fulfilling the condition of our assumption, consist of three parts:

- the subset that contains all the atoms with the predicate symbol *var* – by the definition of the algorithm, this is identical for every answer set, since f_r is fixed,

- the subset that contains all the atoms with the predicate symbol *inter* – by our assumption, this is identical in all the regarded answer sets,
- the optional subset containing either l or nothing. Because all the parts that contain the atoms with predicate symbol *inter* are identical in all regarded answer sets, and the applicability of the rule generating l is only dependent on those atoms, this part also has to be identical in all regarded answer sets.

Therefore, all answer sets containing a fixed subset of atoms with the predicate symbol *inter* would be exactly the same set. Hence, the assumption cannot hold.

2. a) Assume there exists an answer set with a subset of the form

$$\{inter(v\vartheta, f_r(v)) : v \in \mathcal{V}_r\}$$

but there is no possible corresponding grounding $\mathcal{G}_r$ of r under $\mathcal{C}$. For there to be no corresponding grounding, at least one of the following four conditions would have to occur:

- There exists a $v \in \mathcal{V}_r$ with no corresponding atom with the predicate symbol *inter*. This is not possible, as demonstrated above.
- The answer set contains an atom $inter(h, c)$ with the first term such that $f_r^{-1}(h) \notin \mathcal{V}_r$. In order for this to be, $var(h)$ must also be present in the answer set. By the definition of O_r, this may not occur since the only rules which induce the presence of instances of *var* are facts from the first component set, and these only include terms t such that $f_r^{-1}(t) \in \mathcal{V}_r$.
- In the considered answer set, there exist two atoms, $inter(h_1, c_1)$ and $inter(h_2, c_2)$, for which $h_1 = h_2$ and $c_1 \neq c_2$. However, there always exists only one rule in O_r that may induce atoms with the predicate symbol *inter* in an answer set, and that rule is a disjunctive one, for which the second terms of the head atoms are pairwise unequal. Furthermore, f_r assigns a unique constant to each $v \in \mathcal{V}_r$, by its definition.
- An atom belonging to the answer set has the form $inter(h, c)$, where $f_r^{-1}(h) \in \mathcal{V}_r$ and $c \notin \mathcal{C}$. But, per the definition of the second component of O_r, any atom having the predicate symbol *inter* may only possess a second term t fulfilling the constraint $t \in C$. In accordance with the lemma's premises, $\mathcal{C}$ is non-empty, and, per definition of C, $C = \mathcal{C}$. Hence, for any atom of the form $inter(h, c)$ in the answer set, it must be the case that $c \in \mathcal{C}$. This assumption cannot hold as well.

b) Assume there exists an answer set with a subset of the form

$$\{inter(v\vartheta, f_r(v)) : v \in \mathcal{V}_r\}$$

but there exists more than one possible corresponding grounding $\mathcal{G}_r$ of $\mathcal{V}_r$ under $\mathcal{C}$. Since such a subset uniquely associates every variable $v \in \mathcal{V}_r$ with exactly one constant each, this means that such an answer set corresponds to at most one grounding in the way described in Part 2 of the Lemma. Therefore, this assumption also cannot hold.

Because both components of the lemma have been proven to hold, the lemma holds as well. □

We note that the final component of O_r, i.e., the block

$$\{l \leftarrow \{inter(v, f_r(v)) : v \text{ a variable in } l\} : l \in H(r) \cup B^-(r)\},$$

causes the grounding of all literals in $H(r) \cup B^-(r)$.

Generally, an answer set of O_r has the following form:

$$\begin{aligned}\{var(f_r(x)) : x \in \mathcal{V}_r\} \cup \{inter(c, f_r(x)) : x \in \mathcal{V}_r, c \in C\} \cup \\ \{p(c_1, ..., c_n) : p(x_1, ..., x_n) \in H(r) \cup B^-(r), x_i = c_i \text{ if } c_i \in C, \text{ or} \\ c_i = k \text{ such that there is an } inter(k, f_r(x_i)) \text{ otherwise}\}.\end{aligned}$$

One can equivalently, per Lemma 2, rewrite the third subset to

$$\bigcup_{l \in H(r) \cup B^-(r)} \{l\vartheta : \text{for each } \vartheta : \mathcal{V}_r \to \mathcal{C}\} \tag{4.3}$$

for some $\vartheta : \mathcal{V}_r \to C$.

Preparatorily for the next result we consider an analysis of the answer to the brave reasoning query: Regardless of the situation, the positive part of the query is always satisfied under an answer set of any O_r, due to the first subset described in the general answer set characteristic given above.

1. The query returns *false*. This means that, for every $\vartheta : \mathcal{V}_r \to \mathcal{C}$, there exists a negative body atom a of the query and an $l \in H(r) \cup B^-(r)$ such that $a\vartheta = l\vartheta$.
2. The query returns *true*. This means that there exists a $\vartheta : \mathcal{V}_r \to \mathcal{C}$ for which, for all a, there is no l with $a\vartheta = l\vartheta$.

Theorem 5. *$LC010DEC(r, \mathcal{C})$ returns true iff r is eligible for FD-$LC_{0\text{-}1\text{-}0}$.*

Proof. Recall that a rule is eligible for *FD-$LC_{0\text{-}1\text{-}0}$* iff, for every $\vartheta : \mathcal{V}_r \to \mathcal{C}$, $B^+(r\vartheta) \cap (H(r\vartheta) \cup B^-(r\vartheta)) \neq \emptyset$.

The algorithm returns true only if the brave query returns *false*. It only does so in the situation described in Point 1 of our analysis. Since an atom a is contained in the negative body part of the query if and only if $a \in B^+(r)$, it follows from the definition of *FD-$LC_{0\text{-}1\text{-}0}$* that r is eligible for it.

Conversely, assume that r is eligible for *FD-LC$_{0\text{-}1\text{-}0}$*but the algorithm returns *false*. Then, the result of the query on O_r is *true*. By Point 2 of our analysis above, there exists a $\vartheta : \mathcal{V}_r \to \mathcal{C}$ for which, for all a, there is no l with $a\vartheta = l\vartheta$. But that means, by definition, that the rule is not eligible for *FD-LC$_{0\text{-}1\text{-}0}$*, which is a contradiction to our assumption. □

4.5 *FOLD* eligibility detection

As with *FD-LC$_{0\text{-}1\text{-}0}$*, for *FOLD*, there are several points to consider:

- the algorithm has to return not only the result of the applicability test, but also at least the literal to be reduced.
- from the definition, there is an additional test which requires the examination of *all* rules in the program.

This time, for the sake of brevity, only one method will be presented, namely a declarative one.

However, first an auxiliary definition shall be introduced for describing a set containing the valid candidates for the *FOLD* reduction.

Definition 23. *Let P be a program P and a an atom. Then, the following sets are introduced:*

$$E_a := \{(r_1, r_2, l_1, l_2) : r_1, r_2 \in P, l_1 \in B^+(r_1), l_2 \in B^-(r_2), |a| = |l_1| = |l_2|\},$$
$$E'_a := \{(r_1, r_2, l_1, l_2) : (r_1, r_2, l_1, l_2) \in E_a, \text{ for each } \vartheta \text{ and } b \in \textstyle\bigcup_{r \in P} H(r), l_1\vartheta \neq b\vartheta \text{ and } l_2\vartheta \neq b\vartheta\}, \text{ and}$$
$$E''_a := \{(r_1, r'_2, l_1, l_2) : (r_1, r_2, l_1, l_2) \in E'_a, r'_2 = H(r_2) \leftarrow B^+(r_2) \cup \{l_2\}, \text{not}(B^-(r_2) \backslash \{l_2\})\}.$$

Theorem 6. *Given a program P and rules $r_1, r_2 \in P$, if r_1 and r_2 are eligible for FOLD, then, for some a, l_1, and l_2, $(r_1, r'_2, l_1, l_2) \in E''_a$, where $r'_2 = H(r_2) \leftarrow B^+(r_2) \cup \{l_2\}$, and vice versa.*

Proof. This follows directly from the definition of *FOLD*. The generation of E_a and E''_a corresponds to the three conditions for the structure of the pairs of rules, and the generation of E'_a corresponds to the final condition. □

The first idea in designing the algorithm might be to create a fully declarative approach, by checking the entire program for *FOLD* applicability simultaneously. On the other hand, E''_a already offers an efficient way to find candidate rules, without resorting to a full-blown, potentially bloated logic program. A hybrid approach, i.e., utilizing an

imperative mechanism for calculation of preliminary data and creating a logic program for specific, pre-selected candidates, creates arguably a more sensible solution.

However, first, we have to define and evaluate the declarative algorithm itself. The following part of the section is devoted to this task.

Definition 24. *FOLDDECCHECK*$((r_1, r_2', l_1, l_2), \mathcal{C})$*, where* (r_1, r_2', l_1, l_2) *is a tuple from the previously defined* E_a'' *runs in the following way:*

1. *For the given* (r_1, r_2', l_1, l_2)*:*

   a) *Rewrite each constant in such a way that, if one rewrites the variable symbols in* r_2' *and* r_1*, the set* $\mathcal{C}_\mathcal{V}$ *encompassing them will be mutually disjoint with the set* $\mathcal{C}$*.*

   b) *Replace each anonymous variable with a new variable symbol.*

   c) *Reify the variables present in each rule, by creating facts of the following form:*

   $$var(\mathit{Type}, \mathit{VarName}) \leftarrow \ ,$$

   where 'Type' is one if the literal encompassing the variable belongs to r_1*, or 'two' otherwise, and 'VarName' is the variable symbol with the first character made lowercase.*

   d) *Define* D_v *as the set of all facts created during the process described above.*

   e) *Reify the constants present in both rules by creating facts of the following form:*

   $$const(\mathit{ConstName}) \leftarrow \ ,$$

   where 'ConstName' is the constant symbol.

   f) *Define* D_c *as the set of all facts created during the process described above.*

   g) *Reify each literal from the rules, with facts of the following form:*

   $$pred(\mathit{Type}, \mathit{Place}, \mathit{Name}, \mathit{Num}, \mathit{Pos}, \mathit{Term}) \leftarrow \ ,$$

   where:

   - *'Type' is 'one' if the literal belongs to* r_1*, and 'two' otherwise,*
   - *'Place' is either 'head', 'bodyp', or 'bodyn',*
   - *'Name' is the predicate symbol of the literals,*
   - *'Num' is a unique counter given to each literal with the same predicate symbol in a given rule (see discussion below),*
   - *'Pos' is the position within the arity structure of the literal, and*
   - *'Term' is the variable or constant symbol for the given position.*

h) Define D_p as the set of all facts created during the process described above.

i) Define D_{guess} as:

$$\begin{aligned}\{&rename(X,X) \leftarrow const(X),\\ &rename(X,Y) \vee \neg rename(X,Y) \leftarrow var(one,X), var(two,Y),\\ &\leftarrow rename(X,Y), rename(X,Z), Y! = Z,\\ &\leftarrow rename(X,Y), rename(Z,Y), X! = Z,\\ &\leftarrow var(one,X), \#\text{count}\{Y : rename(X,Y)\} = 0,\\ &\leftarrow var(two,X), \#\text{count}\{Y : rename(Y,X)\} = 0\}.\end{aligned}$$

j) Define D_{check} as:

$$\begin{aligned}\{ &\leftarrow pred(one, Place, Name, Num, Pos, Term),\\ &\#\text{count}\{AnyNum : pred(two, Place, Name, AnyNum, Pos, Term2)\} = X,\\ &\#\text{count}\{AnyNum : pred(one, Place, Name, AnyNum, Pos, Term)\} = Y,\\ &rename(Term, Term2), X! = Y,\\ &\leftarrow pred(two, Place, Name, Num, Pos, Term),\\ &\#\text{count}\{AnyNum : pred(one, Place, Name, AnyNum, Pos, Term2)\} = X,\\ &\#\text{count}\{AnyNum : pred(two, Place, Name, AnyNum, Pos, Term)\} = Y,\\ &rename(Term2, Term), X! = Y\}.\end{aligned}$$

k) Define

$$O = D_v \cup D_c \cup D_p \cup D_{guess} \cup D_{check}.$$

2. *Evaluate O. If O possesses at least one answer set, output a random one, filtered for the rename predicate. Otherwise, return false.*

Informally speaking, the algorithm constructs a program which "guesses" all the possible renamings and checks whether each one provides an identity between the rules. The special #count predicate and the *Num* element are needed because it is entirely possible for there to be more than one literal with the same predicate symbol in the given segment of a particular rule.

A more formal analysis will now follow:

The deciding factor of the program's output is the answer set of O. Quite obviously, it will contain all sets of facts D_v, D_c, and D_p. Since D_{check} requires an instance of *rename* for any of its rules to be applicable, D_{guess} must be considered first.

The first rule of D_{guess} ensures that constants are detected by the constraints of D_{guess}.

The second rule of D_{guess} creates an atom with the predicate symbol $\neg rename$ or *rename* for every pair in $\mathcal{V}_{r_1} \times \mathcal{V}_{r_2}$. All such possible pairs are created, represented by distinct answer sets. The other rules remove answer sets for which one of the following is true:

- containing two positive atoms with the predicate symbol *rename* for a $v \in \mathcal{V}_{r_2}$,
- containing two positive atoms with the predicate symbol *rename* for a $v \in \mathcal{V}_{r_1}$,
- containing no positive atoms with the predicate symbol *rename* for a $v \in \mathcal{V}_{r_1}$,
- containing no positive atoms with the predicate symbol *rename* for a $v \in \mathcal{V}_{r_2}$.

In summary, the four conditions imply that for every $v_1 \in \mathcal{V}_{r_1}$ there exists exactly one $v_2 \in \mathcal{V}_{r_2}$, and vice versa.

Since the first rule generates answer sets which correspond to any possible relation between the elements of sets $\mathcal{V}_{r_1}$ and $\mathcal{V}_{r_2}$, the conditions described above ensure that only and exactly those answer sets which represent all possible bijections remain.

D_{check} introduces two more constraints, which remove answer sets for which the following holds:

- For any given atom with the predicate symbol *pred* corresponding to the first rule, the number of such atoms with the same predicate name, component of the rule, position in the predicate, and the given symbol s_1, must be equal to the number of atoms with the predicate symbol *pred* in the second rule with the same predicate name, component of the rule, position in the predicate, and the symbol s_2 for which there exists an atom $rename(s_1, s_2)$ in the given answer set.[2]
- For any given atom with the predicate symbol *pred* corresponding to the second rule, the number of atoms with the predicate symbol *pred* with the same predicate name, component of the rule, position in the predicate, and the given symbol s_2, must be equal to the number of *pred* instances in the first rule with the same predicate name, component of the rule, position in the predicate, and the symbol s_1 for which there exists an instance of $rename(s_1, s_2)$ in the given answer set.

Theorem 7. *Let r_1 and r_2 be rules, a an atom, and $\mathcal{C}$ a domain. Then, for any literals l_1 and l_2 such that $(r_1, r_2', l_1, l_2) \in E_a''$, if $FOLDDECCHECK((r_1, r_2', l_1, l_2), \mathcal{C})$ returns a renaming, then r_1 and r_2 are eligible for FOLD.*

Proof. Assume the contrary, i.e., that $FOLDDECCHECK((r_1, r_2', l_1, l_2), \mathcal{C})$ returns a renaming, for some literals l_1 and l_2 with $(r_1, r_2', l_1, l_2) \in E_a''$, but r_1 and r_2 are not reducible under *FOLD*.

There are two possible subcases:

- There exists only an invalid renaming (which is not a bijection) for the variables of r_1 and r_2. But then, it follows from the analysis that no answer set is generated in this case, and hence, from the last step of the algorithm, that no renaming is generated.

[2]Note that if there exists no possible valid bijection due to, e.g., the difference of cardinality between the two sets of variables, then no answer sets are generated at all.

- There exists a valid renaming, but there is a literal l in r_1 for which there is no corresponding literal k in r_2 with $l\delta = \text{not}\, k\delta$. But then, there must exist a position i in l whose symbol has no unique corresponding symbol in any literal in the other rule. But then:
  - if the symbol is a constant, then the corresponding answer set would be removed by a constraint in D_{check}, due to the first rule of D_{guess},
  - if the symbol is a variable, then the corresponding answer set would also be removed by a constraint in D_{check}, due to the remaining rules of D_{guess}.

  In both cases, the renaming that would have to be returned for the antecedent of the assumption to hold, could not possibly be represented by any answer set of O, and hence could not actually be returned by the algorithm.

A contradiction is therefore reached in all subcases. □

Theorem 8. *Let r_1 and r_2 be rules, a an atom, and $\mathcal{C}$ a domain. Then, for any literals l_1 and l_2 such that $(r_1, r_2', l_1, l_2) \in E_a''$, if r_1 and r_2 are eligible for FOLD, then $FOLDDECCHECK((r_1, r_2', l_1, l_2), \mathcal{C})$ returns a renaming.*

Proof. Assume that r_1 and r_2 are reducible under *FOLD* but $FOLDDECCHECK((r_1, r_2', l_1, l_2), \mathcal{C})$ returns no renaming, for some literals l_1 and l_2 with $(r_1, r_2', l_1, l_2) \in E_a''$.

Analogously to the above proof, if r_1 and r_2 are reducible under *FOLD*, then there must exist a valid renaming (bijection) for which every atom $l \in (B(r_1) \cup H(r_1)) \setminus (B(r_2) \cup H(r_2))$ has a corresponding atom $k \in (B(r_2) \cup H(r_2)) \setminus (B(r_1) \cup H(r_1))$ such that $l\delta = k\delta$. Therefore, for all such positions i in l, there is a unique corresponding symbol in a literal in the other rule. We can distinguish the following two cases:

- If the symbol is a constant, then none of the constraints of D_{check} are applicable, because of the first rule in D_{guess}.
- If the symbol is a variable, then there exists at least one answer set for which none of the constraints of D_{check} are applicable for this symbol, because of the other rules in D_{guess}.

Because all symbols share this property, as r_1 and r_2 are by hypothesis eligible for *FOLD*, O must have at least one answer set for the given input rules (recall from the analysis that any bijection can be potentially created by the algorithm). However, this is a contradiction with the consequent of the assumption, since the algorithm then must return the renaming associated with this answer set. □

We can now proceed with the overall algorithm for testing *FOLD* eligibility:

Definition 25. *$FOLDDEC(P, \mathcal{C})$, where P is a program and $\mathcal{C}$ is a domain, runs in the following way:*

1. *Define $S := \emptyset$.*

2. *For every* $r, s \in P$*:*

   - *for every predicate symbol* a *that occurs in either* r *or* s*, define* E''_a*, and run FOLDDECCHECK for every element of* E''_a*. If FOLDDECCHECK returns a renaming at least once, add* (r, s, a) *to* O*.*

3. *Return* S*.*

Theorem 9. *Let* P *be a program and* $\mathcal{C}$ *a domain. Then, for all rules* $r, s \in P$*,* r *and* s *are eligible for FOLD iff* (r, s, a) *is a member of the output set* S *of* $FOLDDEC(P, \mathcal{C})$*, for some* a*.*

Proof. This follows directly from Theorems 6, 7, and 8. □

Chapter 5

Combining transformations

Recall that, in previous chapters, we have narrowed down the choice of transformations to be used in our experiments to two transformation systems:

- $\{FD\text{-}LC_{0\text{-}1\text{-}0}, SUB, FOLD, LSH\}$, which contains purely syntactical rules, and
- $\{SRRS, SRLS\}$, which contains purely semantical rules.

Having chosen the systems, one must determine a meta-plan for their application. The first and fundamental step is to study the interaction between the selected transformations. This is the focus of the present chapter.

5.1 General concepts

We impose the following partial preordering on the set of programs:

Definition 26. *Given programs P and P', we say that P is* shorter *than P' if either*

- *the cardinality of P is lower than the cardinality of P', or*
- *the cardinality of P is equal to the cardinality of P', but the sum of the cardinalities of the sets $H(r) \cup B(r)$, for every $r \in P$, is lower than the sum of the cardinalities of $H(r) \cup B(r)$, for every $r \in P'$.*

A rule r_1 is shorter *than a rule r_2 iff the cardinality of the set $H(r_1) \cup B(r_1)$ is lower than the cardinality of the set $H(r_2) \cup B(r_2)$.*

For both of the above notions, *longer* denotes the inverse relation to shorter.

These orderings allow us to quantify the results of applying several transformations, one after the other. Thus, the goal of the following discussion shall be to find out what types of transformation sequences produce the best results in this context. For that, we need some additional definitions.

Definition 27. *Given a transformation system S and a program P, a* transformation tree *is a tree with labeled edges constructed as follows:*

1. *define P as the root node of the tree,*
2. *let $N := \{P\}$,*
3. *choose one element from N, remove it from N, and define it as O,*
4. *for every applicable transformation $T \in S$:*
   a) *apply the transformation on O,*
   b) *define the resulting program O' as a new node, and (O, O') a new unidirectional edge labeled with T,*
   c) *let $N := N \cup \{O'\}$,*
5. *if $N = \emptyset$ terminate, else go to Step 3.*

Definition 28. *Let S be a transformation system, P a program, and $\mathcal{T}$ the transformation tree for P and S.*

1. *A* transformation sequence *is any path in $\mathcal{T}$.*
2. *A* transformation branch *is a transformation sequence that starts at the root node and terminates at a leaf node of the transformation tree.*
3. *A* proper transformation sequence *is a transformation sequence in which a transformation application can only be preceded by an application of a different transformation type, or by a sequence of applications of the same type only if no such other sequence occurs previously.*
4. *A* maximal reduction *is a node for which there is no shorter program in $\mathcal{T}$.*

Because, to the best of our knowledge, no research on the sequencing of the transformations discussed in this thesis has been performed, a proper investigation must be conducted here.

5.2 Syntactic transformations

The best situation would be if all transformations could invariably be aligned in a predetermined sequence. Sadly, as it will be shown, this is impossible in general.

However, this general observation does not imply the non-existence of any ordering. Indeed, one transformation can always be placed in a predetermined order - namely, $FD\text{-}LC_{0\text{-}1\text{-}0}$. More specifically, the following property holds:

Lemma 3. *If $FD\text{-}LC_{0\text{-}1\text{-}0}$ is not applicable to a program P, then it is not applicable to any program P' resulting from P by applying any transformation rule from the system $\{SUB, FOLD, LSH\}$.*

Proof. Assume that *FD-LC*$_{0\text{-}1\text{-}0}$ is not applicable to P and let P' be a program resulting from P by applying some transformation T from $\{SUB, FOLD, LSH\}$. We distinguish three cases depending on which transformation was applied.

1. Case $T = SUB$: Since *SUB* only removes rules, *FD-LC*$_{0\text{-}1\text{-}0}$ clearly cannot become applicable to P', because otherwise P would be already eligible for it.
2. Case $T = FOLD$: Since, similar to *SUB*, *FOLD* only removes rules and atoms, we obtain likewise that P' cannot be eligible for *FD-LC*$_{0\text{-}1\text{-}0}$.
3. Case $T = LSH$: According to its definition, *LSH* replaces a rule r by the collection $N_r = \{h \leftarrow B(r), \text{not } H(r) \setminus h \mid h \in H(r)\}$ of rules. Now, for each $s \in N_r$, it clearly holds that $B^+(r) = B^+(s)$ and $H(r) \cup B^-(r) = H(r) \cup B^-(r)$. Hence, since no rule in P satisfies the precondition of *FD-LC*$_{0\text{-}1\text{-}0}$, none of the rules introduced by *LSH* in P' satisfies the preconditions of *FD-LC*$_{0\text{-}1\text{-}0}$ either. Therefore, P' is not eligible for *FD-LC*$_{0\text{-}1\text{-}0}$.

□

Consequently, in the transformation system $\{$*FD-LC*$_{0\text{-}1\text{-}0}, SUB, FOLD, LSH\}$, applications of *FD-LC*$_{0\text{-}1\text{-}0}$ must always precede applications of *SUB*, *FOLD*, and *LSH*. This in turn implies the following result:

Theorem 10. *In the transformation system $\{$FD-LC$_{0\text{-}1\text{-}0}, SUB, FOLD, LSH\}$, a maximal reduction is always guaranteed when applying FD-LC$_{0\text{-}1\text{-}0}$ at precisely the start of a transformation sequence.*

Unfortunately, other transformations do not share the same feature. The following property is a case in point.

Theorem 11. *In the transformation system $\{SUB, FOLD, LSH\}$, there is no unique proper transformation sequence providing a maximal reduction for any input program.*

Proof. We show the result by providing two programs, P and P', possessing different proper transformation sequences yielding a maximal reduction.

The two programs are as follows:

$$P = \{a \vee c \leftarrow g,\quad a \leftarrow \text{not } g, \text{not } c,\quad a \leftarrow h, \text{not } c\};$$

$$P' = \{\leftarrow z,\quad k \leftarrow c, z,\quad a \vee c \leftarrow g, k,\quad a \leftarrow k, \text{not } g, \text{not } c,\quad c \leftarrow k, \text{not } g, \text{not } a\}.$$

Then, all derivation possibilities for P are given in Figure 5.1 while Figure 5.2 depicts the same for P'.

Now, the optimal sequence for P is *LSH*, *FOLD*, *SUB*, whereas in the case of P', it is *SUB*, *LSH*,*FOLD*. □

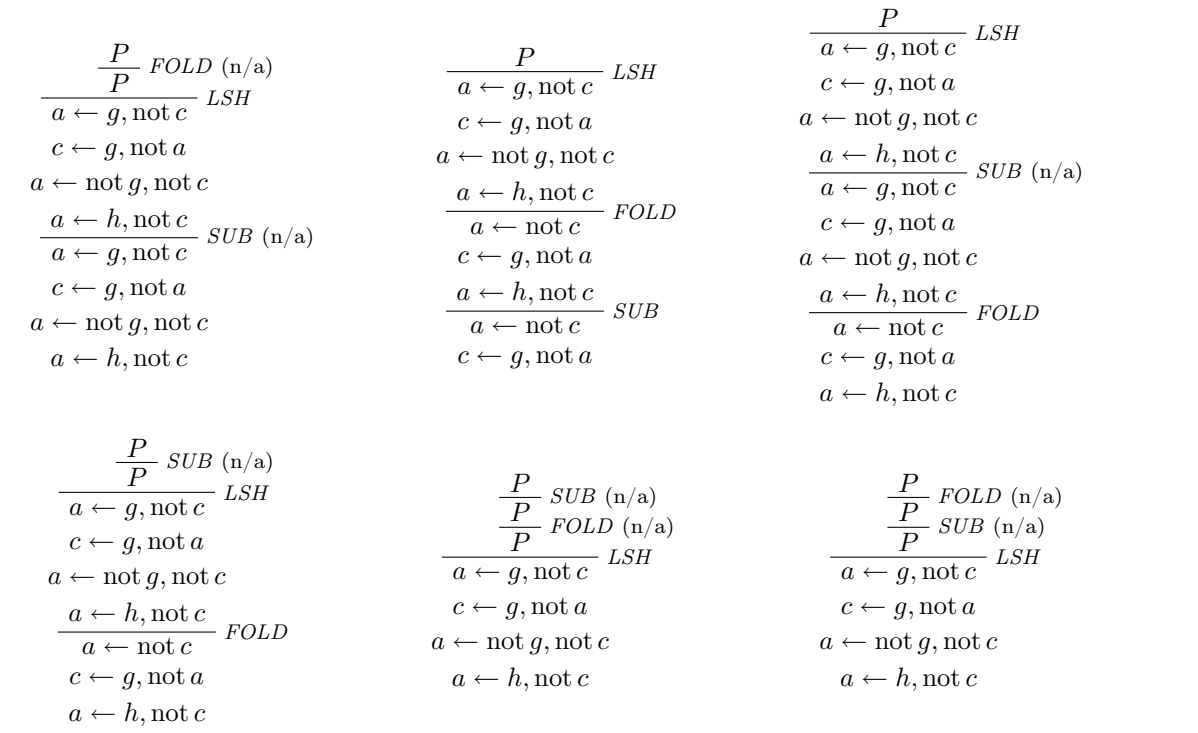

Figure 5.1: The transformation tree for P.

The above is an unfortunate result, since it demonstrates that the search tree for the "best" transformation sequence may potentially be quite large.

Delving deeper into the matter, even more problems can be encountered, as witnessed by the next result.

Theorem 12. *In the transformation system $\{SUB, FOLD, LSH\}$, applying only one kind of transformation in a proper transformation sequence does not guarantee a maximal reduction for every program.*

Proof. Table 5.1 gives programs P_T, for $T \in \{SUB, FOLD, LSH\}$, witnessing the result. In all three cases, the optimal (and, in fact, the only) sequence is one with alternating applications of the relevant transformations. Note that in Table 5.1, branches of the search tree in which no eligibility occurs are disregarded, and all derivations are presented as a single tree. □

The preceding results dealt with transformation sequences. Next, we study questions concerning the permutability of transformation.

Theorem 13. *LSH, FOLD, and SUB are in general not pairwise permutable.*

Proof. Examples showing the non-permutability of the respective transformations are depicted in Figure 5.3. □

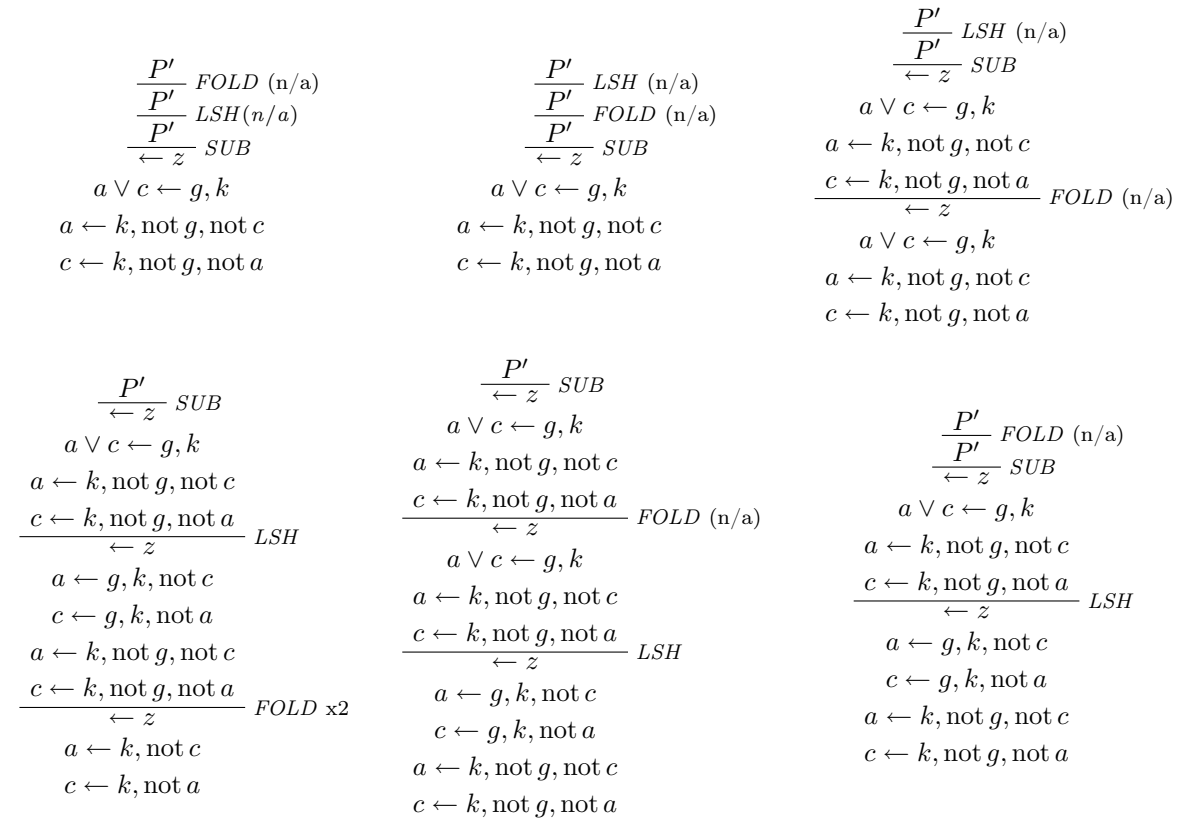

Figure 5.2: The transformation tree for P'.

The results listed above do not exhaust the set of possible local interactions. Specifically, the permutability of transformations with themselves might also possess useful applications. We start with some useful ancillary properties.

Sublemma 1. *Let S be either the transformation system $\{SUB\}$ or $\{LSH\}$. For a program P with n application possibilities, an application of a transformation reduces the number of application possibilities in the resulting program P' to $n-1$.*

Proof. For $S = \{SUB\}$, the result is an immediate consequence of the fact that an application of SUB only removes one rule without modifying any other rules.

In case of $S = \{LSH\}$, since LSH removes a disjunctive rule in P and replaces it with non-disjunctive ones, it follows that the program P' obtained after an application of LSH remains head-cycle-free and the number of possibilities for applying LSH in P' is reduced by one. □

From the above result, we immediately get the following:

Lemma 4. *Let S be either the transformation system $\{SUB\}$ or $\{LSH\}$. Then, given a program P with n application opportunities, every transformation branch has length n.*

T	SUB	$FOLD$	LSH
P_T	$\{a \leftarrow b,$ $\leftarrow b,$ $g \leftarrow c, \text{not } a,$ $g \leftarrow c, a$ $f \vee g \leftarrow c, k\}$	$\{\leftarrow a, b, \text{not } c,$ $\leftarrow a, b, c,$ $d \leftarrow a, b, f,$ $g \leftarrow f, a, d,$ $g \leftarrow f, a, \text{not } d\}$	$\{a \vee b \leftarrow g$ $k \vee a \leftarrow g, h, \text{not } b,$ $h \vee l \vee m \leftarrow k,$ $\leftarrow \text{not } l,$ $\leftarrow \text{not } m\}$
Transformation Tree	$\dfrac{P_{SUB}}{\leftarrow b}\ SUB$ $g \leftarrow c, \text{not } a$ $g \leftarrow c, a$ $\dfrac{f \vee g \leftarrow c, k}{\leftarrow b}\ FOLD$ $g \leftarrow c$ $\dfrac{f \vee g \leftarrow c, k}{\leftarrow b}\ SUB$ $g \leftarrow c$	$\dfrac{P_{FOLD}}{\leftarrow a, b}\ FOLD$ $d \leftarrow a, b, f$ $g \leftarrow f, a, d$ $\dfrac{g \leftarrow f, a, \text{not } d}{\leftarrow a, b}\ SUB$ $g \leftarrow f, a, d$ $\dfrac{g \leftarrow f, a, \text{not } d}{\leftarrow a, b}\ FOLD$ $g \leftarrow f, a, d$	$\dfrac{P_{LSH}}{a \leftarrow g, \text{not } b}\ LSH$ $b \leftarrow g, \text{not } a$ $k \vee a \leftarrow g, h, \text{not } b$ $h \vee l \vee m \leftarrow k$ $\leftarrow \text{not } l$ $\dfrac{\leftarrow \text{not } m}{a \leftarrow g, \text{not } b}\ SUB$ $b \leftarrow g, \text{not } a$ $h \vee l \vee m \leftarrow k$ $\leftarrow \text{not } l$ $\dfrac{\leftarrow \text{not } m}{a \leftarrow g, \text{not } b}\ LSH$ $b \leftarrow g, \text{not } a$ $k \vee a \leftarrow g, h, \text{not } b$ $h \leftarrow k, \text{not } l, \text{not } m$ $l \leftarrow k, \text{not } h, \text{not } m$ $m \leftarrow k, \text{not } l, \text{not } h$ $\leftarrow \text{not } l$ $\dfrac{\leftarrow \text{not } m}{a \leftarrow g, \text{not } b}\ SUB$ $b \leftarrow g, \text{not } a$ $k \vee a \leftarrow g, h, \text{not } b$ $l \leftarrow k, \text{not } h, \text{not } m$ $\leftarrow \text{not } l$ $\dfrac{\leftarrow \text{not } m}{a \leftarrow g, \text{not } b}\ SUB$ $b \leftarrow g, \text{not } a$ $k \vee a \leftarrow g, h, \text{not } b$ $\leftarrow \text{not } l$ $\leftarrow \text{not } m$

Table 5.1: Programs and transformation trees for Theorem 12.

This lemma is needed for the following theorem.

Theorem 14. *For the transformation system S being either $\{SUB\}$ or $\{LSH\}$, each transformation branch generates the same program.*

Proof. By Lemma 4, each transformation branch has the same length. Furthermore, any

- Non-permutability of *LSH* and *SUB*:

$$\dfrac{\dfrac{\dfrac{a \vee b \leftarrow c}{\leftarrow c, \text{not } b}\ \textit{SUB}\ \text{(n/a)}}{a \vee b \leftarrow c}}{\leftarrow c, \text{not } b}\ \textit{LSH} \qquad \dfrac{\dfrac{\dfrac{a \vee b \leftarrow c}{\leftarrow c, \text{not } b}\ \textit{LSH}}{a \leftarrow c, \text{not } b}}{b \leftarrow c, \text{not } a}}{\leftarrow c, \text{not } b}\ \textit{SUB}$$

- Non-permutability of *FOLD* and *SUB*:

$$\dfrac{\dfrac{\dfrac{g \leftarrow a}{g \leftarrow \text{not } a}}{b \leftarrow c}}{\dfrac{a \vee b \leftarrow c}{g \leftarrow a}\ \textit{FOLD}\ \text{(n/a)}}}{g \leftarrow \text{not } a}}{b \leftarrow c}}{\dfrac{a \vee b \leftarrow c}{g \leftarrow a}\ \textit{SUB}}}{g \leftarrow \text{not } a}}{b \leftarrow c} \qquad \dfrac{\dfrac{\dfrac{g \leftarrow a}{g \leftarrow \text{not } a}}{b \leftarrow c}}{\dfrac{a \vee b \leftarrow c}{g \leftarrow a}\ \textit{SUB}}}{g \leftarrow \text{not } a}}{\dfrac{b \leftarrow c}{g \leftarrow}\ \textit{FOLD}}}{a \vee b \leftarrow}$$

- Non-permutability of *FOLD* and *LSH*:

$$\dfrac{\dfrac{\dfrac{a \vee c \leftarrow g}{a \leftarrow \text{not } g, \text{not } c}\ \textit{FOLD}\ \text{(n/a)}}{a \vee c \leftarrow g}}{\dfrac{a \leftarrow \text{not } g, \text{not } c}{a \leftarrow g, \text{not } c}\ \textit{LSH}}}{c \leftarrow g, \text{not } a}}{a \leftarrow \text{not } g, \text{not } c} \qquad \dfrac{\dfrac{\dfrac{a \vee c \leftarrow g}{a \leftarrow \text{not } g, \text{not } c}\ \textit{LSH}}{a \leftarrow g, \text{not } c}}{c \leftarrow g, \text{not } a}}{\dfrac{a \leftarrow \text{not } g, \text{not } c}{c \leftarrow g, \text{not } a}\ \textit{FOLD}}}{a \leftarrow \text{not } c}$$

Figure 5.3: Counterexamples for Theorem 13.

two instances of either *SUB* or *LSH* are clearly permutable, hence the result follows. □

However, the last property is not attainable in general for other transformation systems, as illustrated next.

Theorem 15. *Given a program P and the transformation system $S = \{FOLD\}$, two transformation branches in the transformation tree for P and S may generate different end programs.*

Proof. Consider the program P, comprising the following rules:

$$a \leftarrow g, h; \qquad a \leftarrow h, \text{not } g; \qquad \text{and} \qquad a \leftarrow g, \text{not } h.$$

It is obvious that, in this case, *FOLD* can be applied in two different manners. However, once the application has been made, one cannot proceed further, and the end results differ. □

One final theorem in this section provides an important result, in terms of relevancy to the choice of search strategy.

Theorem 16. *Any transformation sequence in the transformation tree for a program under the transformation system* $S = \{FD\text{-}LC_{0\text{-}1\text{-}0}, SUB, FOLD, LSH\}$ *is finite.*

Proof. We note the following facts about S:

1. Among the transformations in S, only *LSH* increases the number of rules and literals.
2. *LSH* may only be applicable if there are disjunctive rules present.
3. No transformation in S can create new disjunctive rules, only remove them.
4. None of $FD\text{-}LC_{0\text{-}1\text{-}0}$, *SUB*, or *FOLD* may have a non-negative effect on the number of rules in the program – all of them have to remove a rule when applied.
5. We only deal with finite programs.

This implies that *LSH* may be applied only a finite amount of times. Regardless of the sequence we follow, after a finite amount of steps (due to Facts 2 and 5), we reach a point where no application of *LSH* will be possible. We are then only left with $FD\text{-}LC_{0\text{-}1\text{-}0}$, *SUB*, or *FOLD*, which may, at best, reduce the program to an empty program. Because the program is finite, and due to Fact 4, this can be done only in a finite amount of steps. Therefore, each transformation sequence in the system of $\{FD\text{-}LC_{0\text{-}1\text{-}0}, SUB, FOLD, LSH\}$ is finite. □

5.3 Semantic transformations

We now address issues of the transformation system $\{SRRS, SRLS\}$. An initial idea would be to always place rule elimination first. However, this should not be done, as shown below.

Theorem 17. *In the transformation system* $S = \{SRRS, SRLS\}$*, the sequence SRRS, SRLS does not always guarantee maximal reduction.*

Proof. Consider the program consisting of the following rules:

$$a \leftarrow; \qquad b \leftarrow; \qquad d \leftarrow; \qquad e \leftarrow; \qquad l \vee c \leftarrow a; \quad \text{and} \quad c \leftarrow b, d, e.$$

For this program, the only way to obtain a maximal reduction is to apply *SRLS* by removing l, and then *SRRS* by removing the last rule. □

On the other hand, there may be a number of cases where *SRLS* emulates *SRRS*, i.e., instead of one rule elimination, there are multiple literal eliminations. Being not efficient, this can be avoided by utilizing a "soft" hierarchy, namely:

1. Apply rule elimination first,
2. apply literal elimination on the results of rule elimination, or on the original results if none of the latter have been obtained, and
3. if no rule or literal eliminations have been found during this phase, terminate. Otherwise, go back to Step 1,

We chose this strategy for our test setup, as discussed in the next chapter.

5.4 Summary

To recapitulate, the following results have been obtained:

- *FD-LC*$_{0\text{-}1\text{-}0}$ is always to be used at the start of any transformation sequence;
- permutations of transformation sequences of type *LSH* and *SUB* produce the same output,
- there is no risk of an infinitely-long branch of the transformation tree, and
- a "soft" applicability testing hierarchy has been chosen for the semantic system.

In summary, the exhaustive search tree (which must be constructed, because there is no optimal strategy) may have a fair size. Fortunately, the analysis and optimization conducted as part of this thesis is considered to be strictly offline, providing some leeway.

Nevertheless, the only option seems to be an explicit, uninformed tree search. Moreover, it *must* expand the entire structure, as there exists no information on the parameters on the goal. Unlike in searching, e.g., for a specific element or a minimal path, the optimal solution can be singled out only by comparison with its alternatives, by itself having no defined utility.

Theoretically, in the transformation system chosen, it is entirely possible for two sequences to have one or more "common points", i.e., places along the execution path where the intermediate program is identical. Here lies the first clue towards a more efficient analytical system.

The idea is to construct an efficient look-up table (for example, by providing hash indices). Its goal would be to "glue together" branches of the sequence tree. Such an improvement could decrease run-time significantly, depending on the program as input, simply by avoiding duplicate calculations.

Finally, an important choice to be made is one between two main types of tree search:

- breadth-first-search (BFS) – where, upon entering a node, all of its children nodes are checked, before moving to the further successor nodes, or
- depth-first-search (DFS) – where, upon entering a node, each child's descendants are checked first and foremost, and only then the next child is processed.

The choice is only illusory – whichever algorithm will be utilized, it will make the information on the duplicate available immediately, making them equally effective in the context of generating the complete tree.

To conclude, the system will generate unidirectional graphs representing the exhaustive search tree, with nodes referring to programs, and vertices to transformations and associated operations. The invariant is that a program occurs only once in the graph. This graph will be generated either by BFS or DFS, as the utility of both of them remains the same, under the given conditions.

Chapter 6

Experimental results

6.1 The analysis and optimization system

We start with deciding on the general structure of the system. One potential problem that might be encountered relates to the lack of knowledge of the kind of results that might be obtained. It remains entirely within the realm of possibility that the set of data attributes chosen for analysis before the experiment is not sufficient to describe the product. Therefore, effort should be directed into generating an output as generic as possible – in other words, for the data to be relatively "raw". Figure 6.1 illustrates the application of this principle. The system is divided into two portions, one producing data as generic as possible (dubbed the *experimentation framework*, or just *framework*), the other, completely independent and interchangeable, putting that data into analysis (referred to as the *analyzer*).

We now discuss the details of the structure of the experimentation framework. Since the expectation is that there will be multiple samples to test, the framework should be prepared to accept a set of input programs. This implies the approach of the main loop, as shown in Figure 6.1. Each experimentation step is split into two phases, delegated to separate components – the *parser* and the sequence *tree builder*.

The role of the parser is converting the input program into a data structure appropriate for the tree builder to process. As foreshadowed in the previous chapter, since the programs that are written in practice are usually quite small, of the order of dozens of rules (especially versus thousands of lines of code in the case of most imperative programs), focus has been placed on providing as much of a memory-time trade-off as possible. For example, unit-time access to internal representations of constants and variables has been provided at program- and rule-level.

The tree builder receives the data structure mirroring the input, and constructs the exhaustive sequence tree. This process is illustrated by Figure 6.2. The builder runs the applicability test for each transformation in turn, generating new nodes whenever applicability has been detected. If the program generated after applying the current transformation is unique within the set of programs generated previously, a new node is made, with the node itself representing the program, and the vertex standing for the

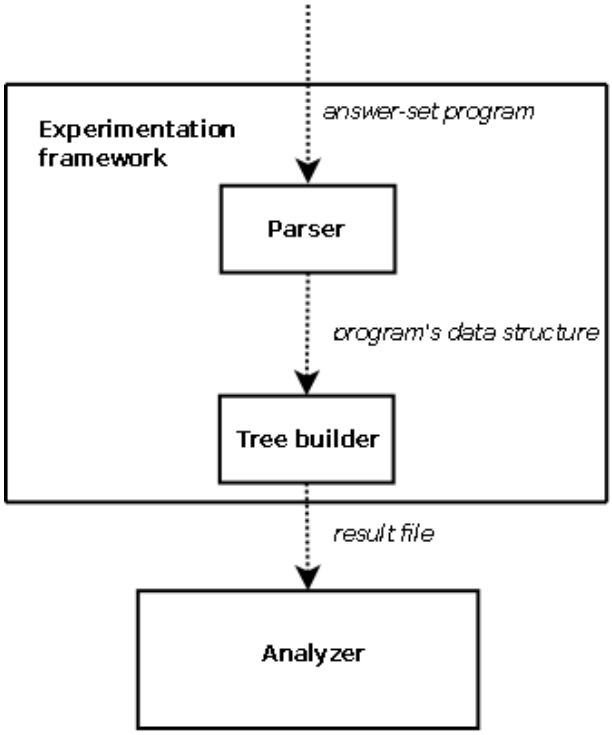

Figure 6.1: The structure of the system: dashed lines indicate data flow.

transformation that was required to convert the predecessor program. However, if the resultant program has already occurred, only the appropriate vertex is created, therefore avoiding duplication.

Additionally, two limiting parameters may be delivered at start-up: the maximal depth of the transformation tree for a program, and the timeout, also on the basis of programs per input. These may become useful in the case of programs which would otherwise take a prohibitively long amount of time to verify.

Care was taken to keep the transformation-definition component generic, in order to make an easy extension of the system possible. All a developer needs in order to add a new transformation is to extend a specific class in the API and follow some basic conventions. Specifically, each transformation class extends the class "AbstractTransformation", and contains the following methods:

- the constructor, which accepts and stores two arguments – the input program and the domain (this is already implemented in the abstract class),
- *generateCandidates*(), which, quite appropriately, generates a candidate tuple (this can be, for example, be a pair of rules) constituting a meaningful transformation candidate,

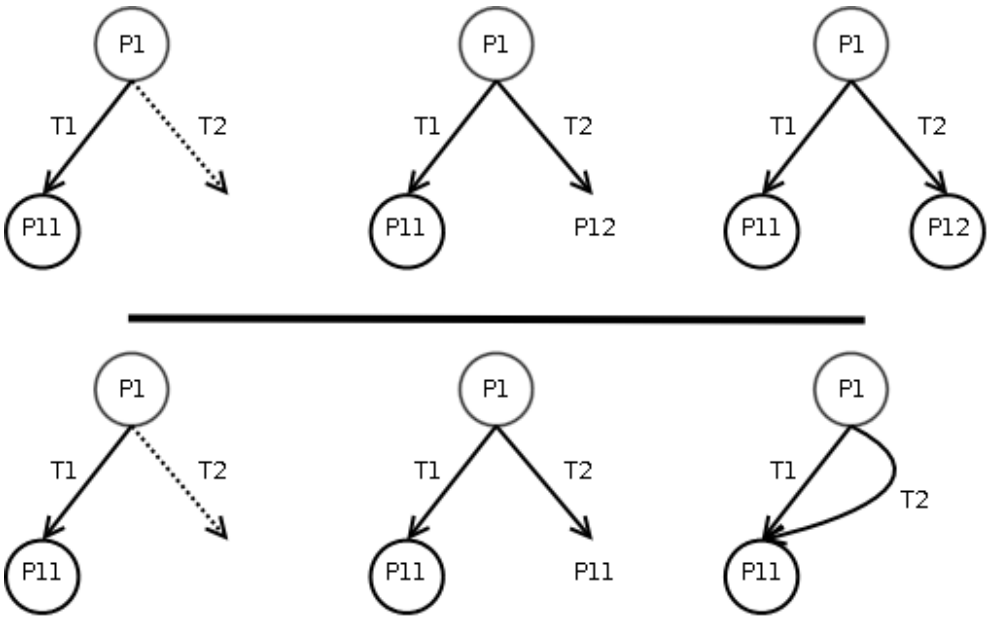

Figure 6.2: The functionality of the tree builder. The first row shows what happens when the generated program is globally unique in the context of the already created transformation sequence. The second row demonstrates the alternative case.

- one or more methods with the name scheme of $algoXXXX$, representing algorithmic variants for the transformation, and run by accepting a candidate and returning either $None$[1] (if the transformation is not applicable for the given candidate tuple), or a list of operations corresponding to a valid application of the transformation, based on the mentioned candidate tuple, and
- *applyAlgo*, again, already implemented in the abstract class, which runs the appropriate algorithm through the generated candidates.

Additionally, each transformation possesses the following attributes:

- its priority relative to the other transformations (transformations having the same priority are understood to be applied interchangeably), and
- whether the transformation is idempotent or not.

All transformation classes are placed in a single directory, which is scanned at start-up by the program, in order to generate the appropriate run-time options.

Because the programs used in practice are not particularly large as already pointed out, it was decided to store the entire transformation tree, along with the following data:

- the total time it took to complete the transformation analysis, summed up as well as categorized per transformation type,

[1]The equivalent of *null* in other languages.

- information whether any of the termination conditions were reached (i.e., whether time-out has been reached or the maximal depth was exceeded),
- the complete program corresponding to each node,
- the transformations used for the transition between the nodes, including the operations (literal addition or removal, etc.) corresponding to those transformation instances.

This solution enables an exhaustive offline examination of the transformation process. The data format is simply a serialized data structure, which allows for an easy and straightforward construction of analyzers – one needs to simply unserialize the data stored in the `.done` file in order to obtain the complete, easy-to-traverse data structure.

As with any application, one of the fundamental choices facing a programmer implementing a system is one of what programming language to use. In this case, attention has been turned to Python[2], which is a high-level, object-oriented language with dynamic typing. It is chiefly an interpreted language (although the interpreter creates a form of byte code á la Java on the fly, to facilitate faster repeated executions), sacrificing some efficiency for rapid prototyping and portability.

The main advantage of the language is, however, its design – not only in a "batteries included" philosophy which netted in a very comprehensive standard library, but in the syntax of the language itself – sequences (arrays, dictionaries, hash sets, etc.) may be manipulated to a quite large degree of complexity with very little code, thanks to such features as list comprehension. For example, the mathematical statement

$$A := \{a^2 : a \in B, a > 3\}$$

and the code

```
A = [a**2 for a in B if a > 3]
```

share a correspondence in the language. The ability to naturally express set-based mathematical formulas, and other features already mentioned in the current section, make Python well-suited for scientific application, as long as minimizing runtime is not the absolute priority. These qualities of the language in our context have indeed been noted in several publications [Hin02, Fan04].

For the parser component, ANTLR was utilized. ANTLR is a parser generator capable of generating output in several languages, Python included. The syntactical structure of the generated program model was constructed "by hand", however, in order to provide the transformation algorithms with efficient access to program components (such as variables, constants, and predicates), at the cost of somewhat increased memory usage and slightly slower transformation application time.

[2] http://www.python.org/

6.2 Test setup

The primary goal here is to determine the practicability of using the chosen transformations for offline optimization of programs. Therefore, the tests concentrate on the number of reductions performed.

The results of the analysis will be viewed through the following criteria, in particular:

- the number of non timed-out analysis runs,
- the average, maximal, minimal, and median time for non timed-out analysis runs,
- the maximal depth of the transformation tree,
- the number of literals and rules in the input programs, and
- the histograms of performable reductions and runtime of the analysis system.

For syntactic transformations, the declarative algorithms have been used, whenever applicable. The time-out has been set to 60 minutes for ensuring a complete result.

All experiments have been performed on a machine with a quad-core Intel Core i7 CPU 920 having a 2.67GHz processor frequency and 8 MB L3 Cache, 12 GB of RAM, and two 1.5 TB HDDs with a 7200 RPM rotation speed, set up in a RAID 1 configuration. As solver we used *DLV*; the specific version was the Oct 2007 release for Linux, `build BEN/Oct 11 2007 gcc 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)`.

As already discussed, the transformation systems underlying our experiments are, on the one hand, $\{FD\text{-}LC_{0\text{-}1\text{-}0}, SUB, FOLD, LSH\}$ (in which case, the imperative algorithm was used for $FD\text{-}LC_{0\text{-}1\text{-}0}$, and the declarative ones for the others) and, on the other hand, $\{SRRS, SRLS\}$.

6.3 Data Sets

Next, we briefly describe the the data sets used for our experiments.

6.3.1 Student data

The first benchmark class is comprised of a small selection of programs written by students of the Vienna University of Technology for a laboratory course on logic programming. The programs were supposed to solve a problem of assigning papers to reviewers, with some constraints given.

Note that, although the programs are correct in the programming language sense (they are interpretable), they do not necessarily provide correct solutions to the problem. This

is because they are unfiltered submissions, and as such, should provide more diversity to the examination.

Although this set is rather small, it nevertheless possesses a high relevance as it represents a sample of "pure ASP code" (i.e., no front-ends used) written by non-experts. Therefore, the results obtained here will be indicative of using the transformations within a possible assistance tool for those learning answer-set programming.

6.3.2 Random programs

The second benchmark set was created using the `ccT-gen` program for testing the `ccT` tool for equivalence checking of answer-set programs [OSTW06]. The tool is a generator for random programs and is available from the `ccT` Web page.[3] It has a number of options, but relevant for our context are the following ones:

- v – determines the number literals,
- p – determines the number of disjunctive rules, and
- q – gives the maximal number of literal instances in the non-disjunctive rules.

In general, the tool generates propositional programs according to the supplied parameters that dictate the complexity of the output. In our experiments, two sets were made: a simpler one (produced with `ccT-gen -v 4 -p 4`) and a more complex one (generated with `ccT-gen -v 50 -p 50 -q 50`).

The purpose of this data sample is twofold: Firstly, efficacy on purely propositional programs was tested. Secondly, random data has a tendency to often highlight interesting anomalies in the results.

6.3.3 *PLP*

PLP [DST01] is an implementation of an approach due to Delgrande, Schaub, and Tompits [DST03] for enhancing answer-set programs by the possibility of specifying preferences between rules. Such an approach allows to influence the number of resultant answer sets, by giving preference among multiple alternatives in case of conflicts. The specific method underlying the general preference method is to compile a logic program with preferences into a standard answer-set program.

For our experiments, we used the compilations of the logic programs with preferences taken from the *PLP* Web page.[4] They thus comprise one of the benchmark collections exploiting programs from an ASP front-end; the second such collection is described next.

[3] `http://www.kr.tuwien.ac.at/research/systems/eq/index.html`.
[4] `http://www.cs.uni-potsdam.de/~torsten/plp/`.

6.3.4 DLV^K

DLV^K [EFL+04, EFL+03] is an integrated front-end for DLV used for specifying and solving problems related to planning. Using the extension, one may define a planning scenario, including invariants, a description of the initial state, and possible actions. One can also set additional parameters for planning, the most prevalent of which is the concurrency mode in which two or more simultaneous actions may be conducted in a single step. Like in the PLP approach, a planning specification in the DLV^K language is compiled into standard answer-set programs which can then be processed by DLV.

The data used for the experiments are, as in the first benchmark collection, the compilations of programs written by students as part of their assignments for a laboratory course on knowledge-based systems at the Vienna University of Technology. The task of the students required to formalize within the DLV^K language the specification of a particular blocks-world scenario as typically used for illustrating planning problems in the AI literature.

6.3.5 Diagnosis front-end

The final data set comprises compilations of programs from the diagnosis front-end of DLV [EFLP99]. The programs again stem from student programs taken from the above mentioned laboratory course on knowledge-based systems.

The diagnosis front-end is based on the principles of model-based diagnosis and allows to define

- the correct behavior of a system to be diagnosed (here in terms of a logic program),
- the observations, i.e., measurements on the system revealing the malfunction of the system, and
- a set of possible defective components.

On the basis of this specification, possible reasons for the observed malfunction are computed by DLV, by selecting a subset of the possible defective components. As for the other front-ends, this computation is done by compiling the diagnosis specification into a standard answer-set program.

In the laboratory course, the task of the students was to diagnose a simple air-conditioning system.

6.4 Results

The presentation of the results is is done in two ways – textual and graphical. The former is achieved in terms of two tables; the latter in terms of pairs of graphs, one per data set and per transformation system.

Before presenting the results, we give a general description of the format and content of the tables. The tables contain the following entries:

Set: listing of all data sets in columns.

Total amount: the sum of all programs that have been read successfully.

Total time: the time it took to analyze the programs.

Total amount: the amount of programs that have not timed out.

Maximal time: the largest time needed for a single program to be analyzed.

Minimum time: the smallest time needed for a single program to be analyzed.

Average time: the average time needed for a single program to be analyzed.

Median time: the median time needed for a single program to be analyzed.

Average maximal tree depth: average length of the longest branch in the program's transformation tree.

Average number of literals: average number of literal instances in a program.

Average maximal literal reduction: the difference between the number of literal instances in the input and the maximal reduction.

Total number of applications per transformation: the sum of all edges labeled with the transformation in all transformation trees.

Total time per transformation: the total time the transformation took.

The graphs, in turn, are histograms and should be understood in the following manner:

- The graphs on the left-hand side of the figures describe the difference of the number of literal instances between the input and the maximal reduction, for each transformation system. The abscissa gives the value of the difference and the ordinate shows the number of programs in the data sets that are distinguished by having the given difference.
- The graphs on the right-hand side of the figures describe the runtime for each program in the data set, again for each transformation system. The abscissa is the runtime in seconds; the ordinate is the number of programs that required the given runtime to be processed.

Table 6.1 gives the results for the syntactic transformations while Table 6.2 gives the corresponding results for the semantic transformations. The diagrams for the data sets under the syntactic transformations are depicted in Figures 6.3 to 6.8 and those for the data sets under the semantic transformations are given in Figures 6.9 to 6.13.

6.5 Observations

The results for the syntactic and the semantic transformations are somewhat different in detail, hence they shall be described separately.

In the case of the syntactic transformations, most analyzing runs terminated before the time out. An exception is the DLV^K case, where only 2% of the input programs did not time out. Fortunately, the system faired much better for the other data sets.

The student data displays the most variety in terms of applicable transformations, as it contains even the sole application of $FD\text{-}LC_{0\text{-}1\text{-}0}$. Still, the programs here seem to be mostly optimal, which is surprising given that they were written by persons who presumably had their first encounter with answer-set programming.

The random programs give a little better result, especially for the first, simpler set. However, the specifics of the types of applicable transformations make it apparent that those programs have a similar structure, and hence are not really truly random. Nevertheless, since the programs were not intended to be used to solve a specific problem, these data sets remain usable as a control group.

The *PLP* and DLV^K data sets both yield zero transformation possibilities. This is to be expected in the former case – the example programs were written by a professional, and are exemplary to the front-end converter in question. On the other hand, there was little possibility for improvement in the DLV^K set, since many programs timed out – that is also a meaningful result, however.

What is surprising is the amount of transformations possible in the diagnosis front-end. This may, however, be on account that this front-end needs little conversion in comparison with DLV^K, expressing the structure of the original program (a student submission) more faithfully. Still, the 2.15 average reduction of literal instances, although the greatest in all the data sets, is nothing remarkable.

The semantic results have the general characteristic of having a higher number of reductions, but also a higher time-out rate.

The results for the student programs are consistent with the syntactic case. There is a considerably larger amount of transformations to be applied, however the amount is still too small to be of any relevance (1.11 per average).

Set:	Student	Random 1	Random 2	PLP	DLV^K	Diagnosis
Criterion	**Result**					
Total amount:	51	100	100	25	147	755
Statistics for not timed-out programs:						
Total time:	17.22min	0.2min	3.99min	24.18min	0.8min	150.11min
Total amount:	51	100	100	25	3	755
Maximum time:	2.99min	0.0min	0.05min	16.8min	0.31min	0.97min
Minimum time:	0.0min	0.0min	0.03min	0.0min	0.24min	0.01min
Average time:	0.34min	0.0min	0.04min	0.97min	0.27min	0.2min
Median time:	0.17min	0.0min	0.04min	0.01min	0.25min	0.12min
Avg. max. tree depth:	0.94	1.3	1.0	0.0	0.0	1.14
Avg. no. of literals:	106.55	18.39	2075.75	245.96	120.67	376.94
Avg. max. literal red.:	0.25	1.05	0.0	0.0	0.0	2.15
Total number of applications per transformation:						
FOLD:	0	0	100	0	0	0
LSH:	46	130	0	0	0	859
FD-LC$_{0\text{-}1\text{-}0}$:	1	0	0	0	0	0
SUB:	3	60	0	0	0	208
Total time per transformation:						
FOLD:	4.92min	1.94min	0.0min	24.0min	0.0min	0.33min
LSH:	12.02min	0.03min	0.12min	0.0min	0.26min	6.18min
FD-LC$_{0\text{-}1\text{-}0}$:	0.15min	0.9min	0.01min	0.01min	0.01min	1.17min
SUB:	0.11min	0.61min	0.04min	0.16min	0.53min	140.75min

Table 6.1: Results for the syntactic transformations.

Set:	Student	Random 1	PLP	DLV^K	Diagnosis
Criterion	**Result**				
Total amount:	49	100	18	146	115
Statistics for not timed-out programs:					
Total time:	315.73min	23.72min	167.57min	1.37min	675.23min
Total amount:	44	100	7	2	67
Maximum time:	59.55min	0.37min	53.99min	0.69min	59.67min
Minimum time:	0.05min	0.09min	6.6min	0.68min	0.01min
Average time:	7.18min	0.24min	23.94min	0.69min	10.08min
Median time:	0.23min	0.24min	23.51min	0.69min	5.0min
Avg. max. tree depth:	0.5	6.84	8.0	0.0	0.75
Avg. no. of literals:	109.09	18.39	72.43	145.0	251.7
Avg. max. literal red.:	1.11	13.65	8.86	0.0	2.15
Total number of applications per transformation:					
SRRS:	1	20493	6928	0	108
SRLS:	2327	2487	5	0	120
Total time per transformation:					
SRRS:	56.32min	20.67min	161.86min	0.44min	537.1min
SRLS:	258.98min	1.86min	2.75min	0.93min	137.92min

Table 6.2: Results for the semantic transformations.

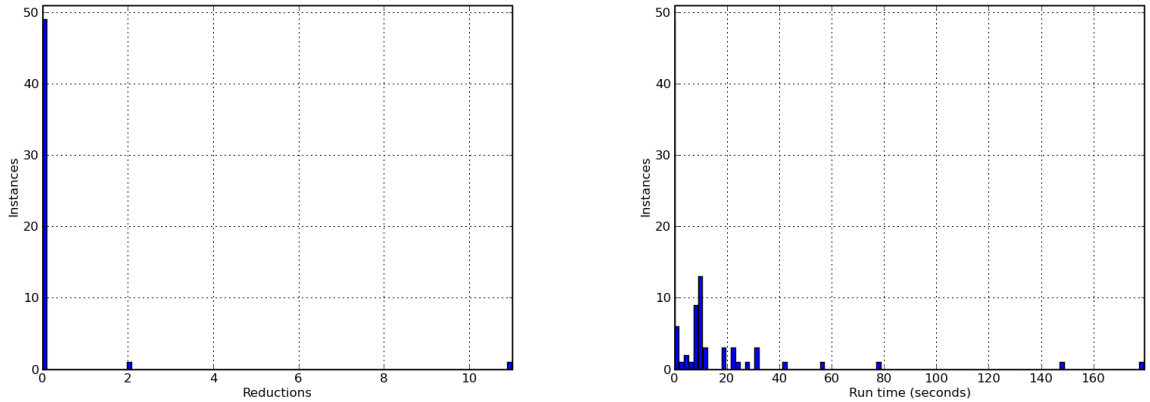

Figure 6.3: Histograms for the Student test set under syntactic transformations.

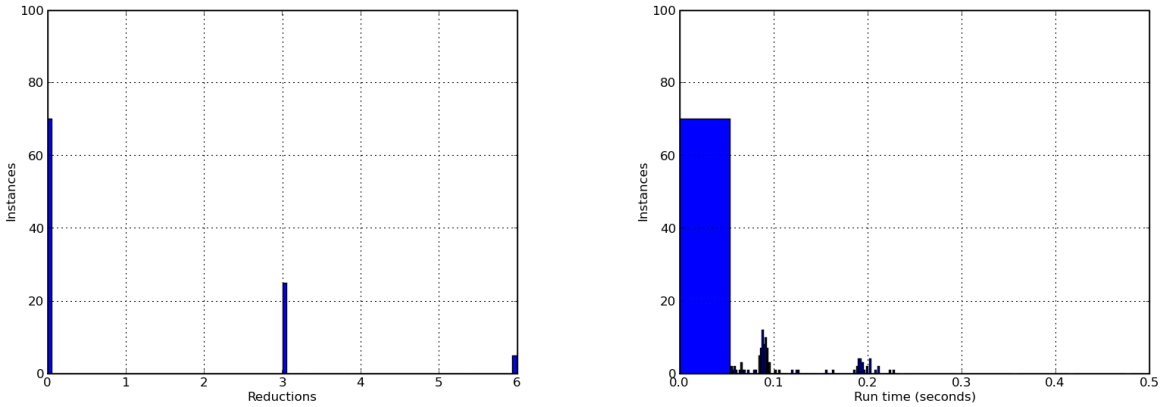

Figure 6.4: Histograms for the Random 1 test set under syntactic transformations.

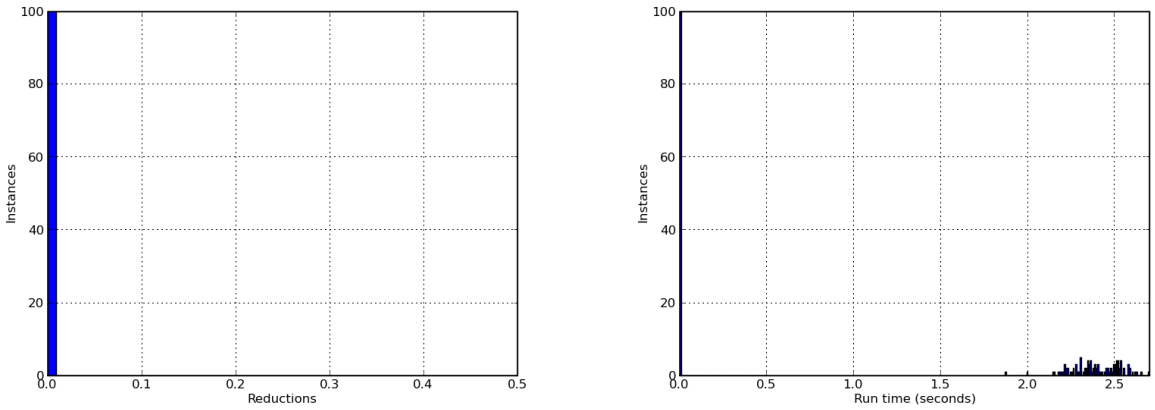

Figure 6.5: Histograms for the Random 2 test set under syntactic transformations.

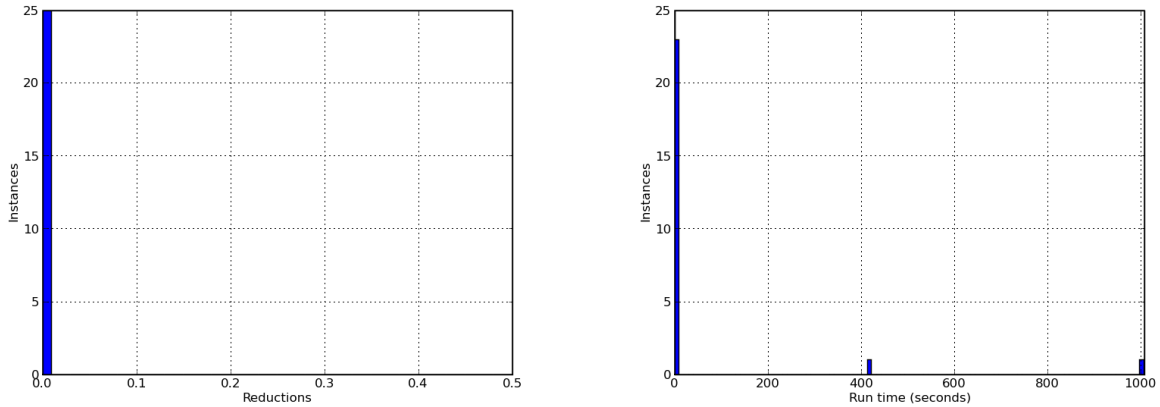

Figure 6.6: Histograms for the *PLP* test set under syntactic transformations.

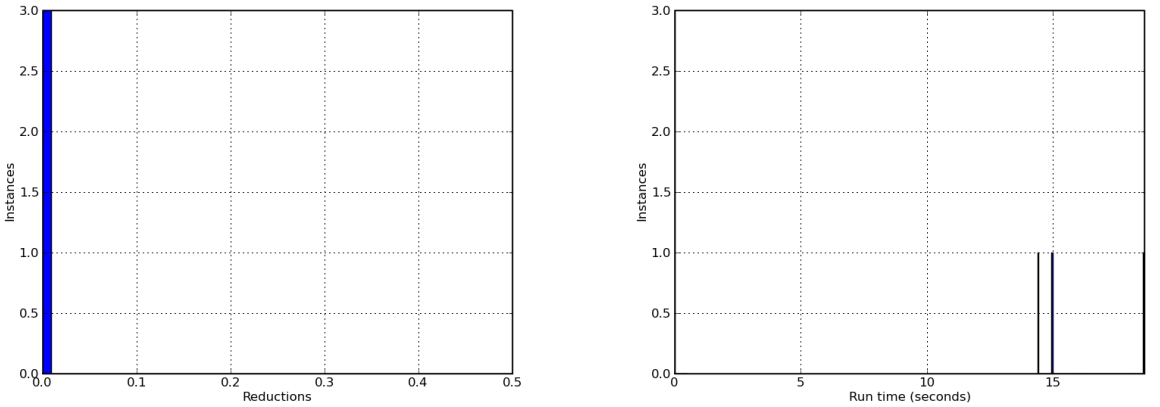

Figure 6.7: Histograms for the DLV^K test set under syntactic transformations.

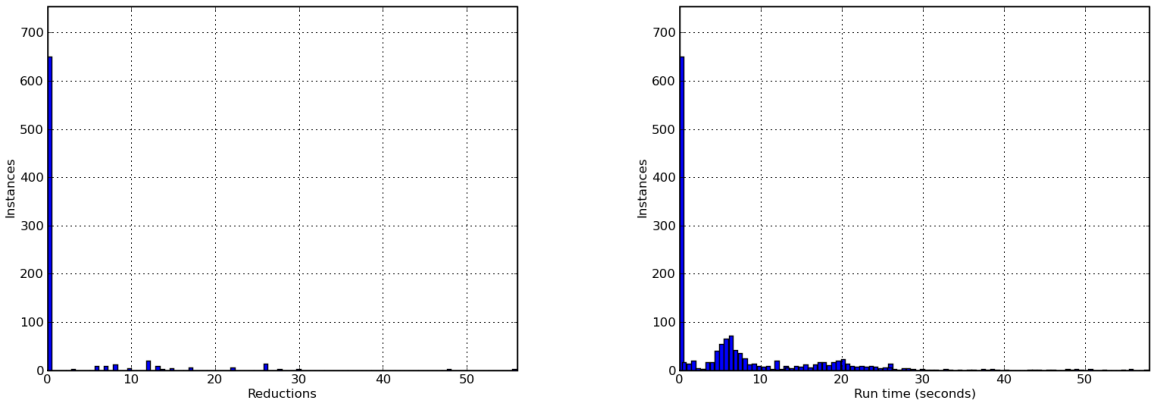

Figure 6.8: Histograms for the Diagnosis test set under syntactic transformations.

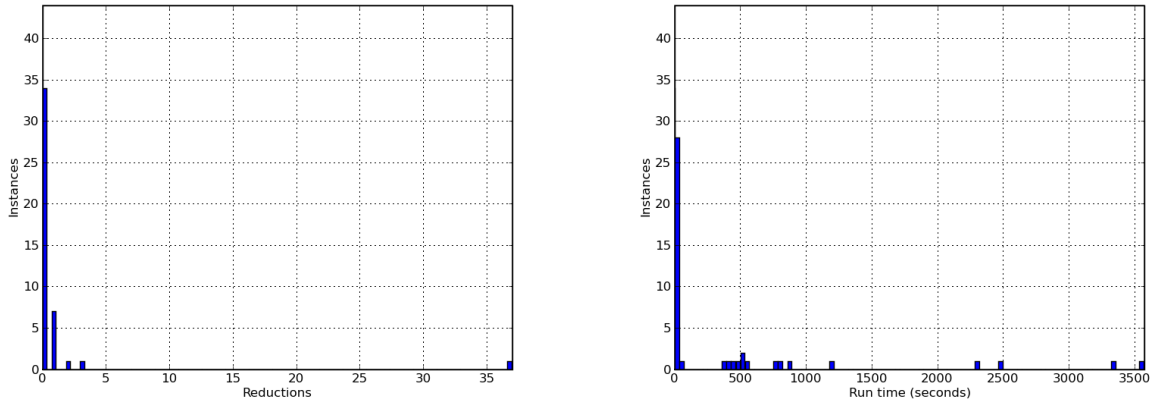

Figure 6.9: Histograms for the Student test set under semantic transformations.

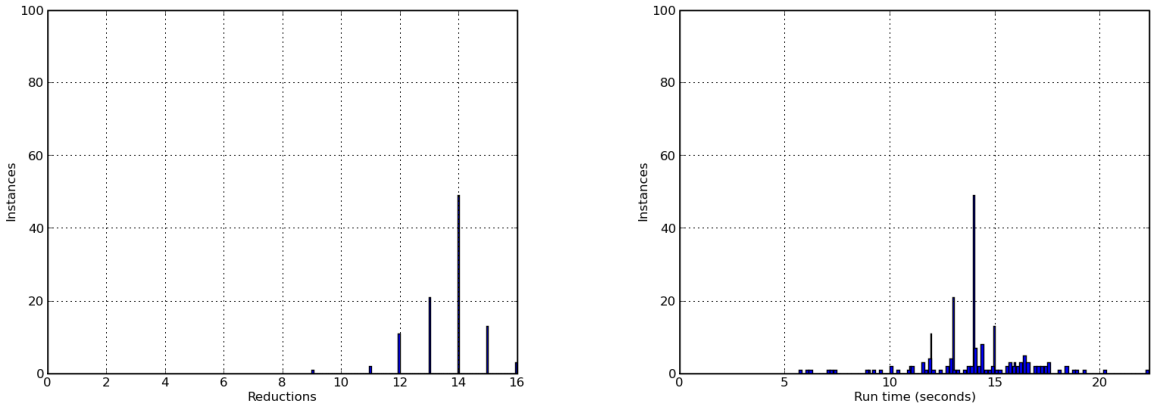

Figure 6.10: Histograms for the Random 1 test set under semantic transformations.

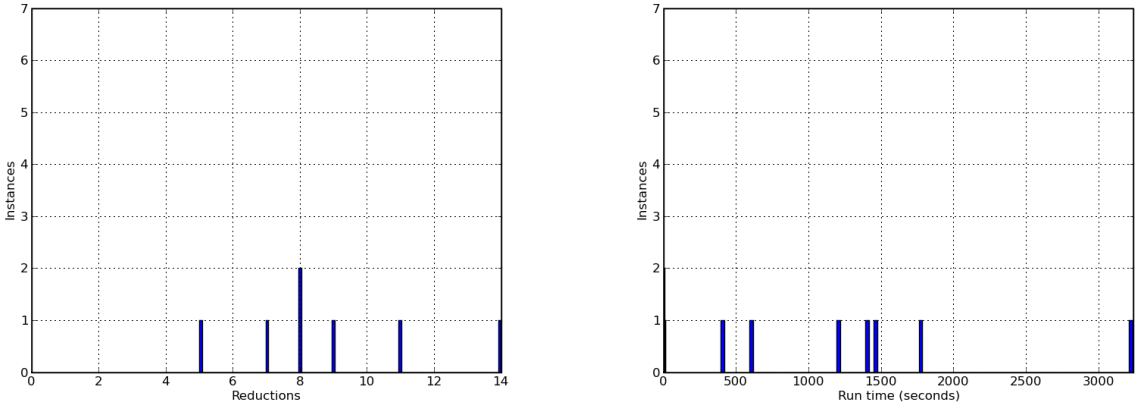

Figure 6.11: Histograms for the *PLP* test set under semantic transformations.

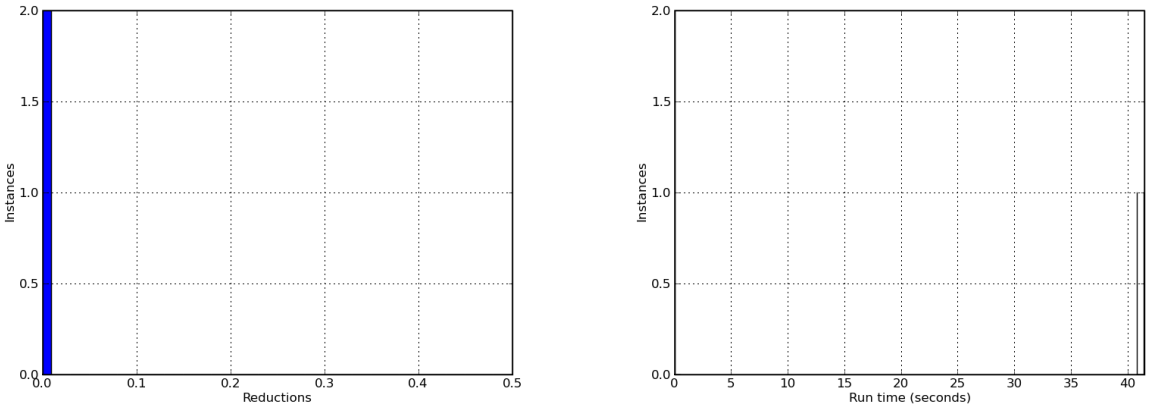

Figure 6.12: Histograms for the DLV^K test set under semantic transformations.

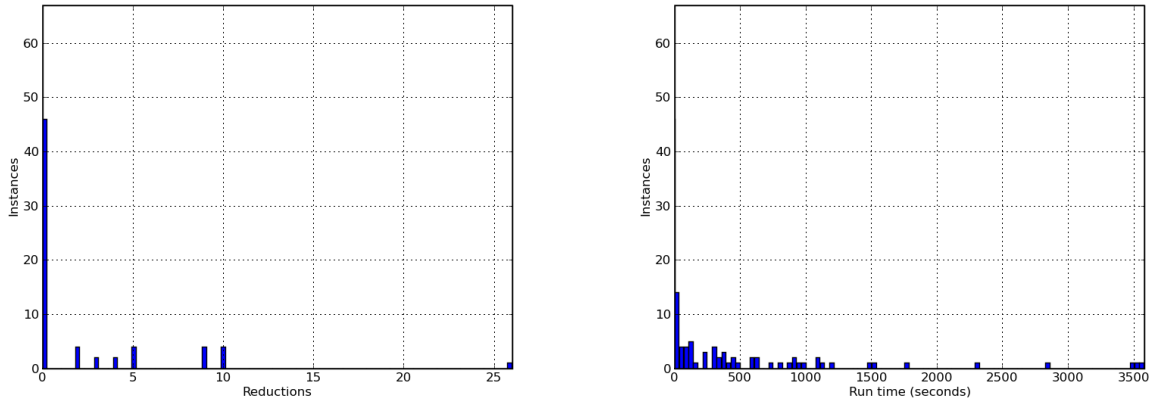

Figure 6.13: Histograms for the Diagnosis test set under semantic transformations.

The random programs give a mixed result. The first set is the most promising overall, with a meaningful number of reductions (over 14 per average). This is probably due to the semantic transformations being structure-agnostic.

Surprisingly, the second-best performance is achieved on *PLP* programs. However, almost two-thirds of the programs timed out, and the ones that did not time out took almost three hours to be checked. The *DLV* front-ends netted a similar result as in the syntactic case, although the time-out rate for diagnosis is again much higher.

In general, the statistics do not paint an encouraging picture on practical applications of answer-set program transformations. To begin with, the execution times for a number of transformations, mostly the semantic ones, are prohibitively large. For most applications, such as IDE integration, they just take too much to operate within the desired time frame. Normally, for such purposes, it would be desired for the execution time to stay under two seconds, which is just not the case, except for $FD\text{-}LC_{0\text{-}1\text{-}0}$, and occasionally

Algorithm	Time (seconds)
SUBDEC	6836
SUBIMP	2991
LCLC010DEC	25719
LCLC010IMP	452

Table 6.3: Results for comparing declarative vs. imperative eligibility-testing algorithms.

for *SUB* as well as for *FOLD* if considered in isolation.

The applicability of transformations is also disappointing. In all the data sets, with the exception of *PLP* and random programs, the number of applications was found to be quite low. This especially applies to syntactic transformations.

Similar can be said about transformation chains. It turns out that, for actual data, the transformation search trees are relatively flat. There is really no net gain in the exploration of possible sequences – or even worse, it generates additional overhead in most cases, as it at least doubles (needlessly) the time needed for the checker to run. For syntactic transformations, that is coupled with the fact that *LSH*, which was to be an "enabling" transformation, contributed massively to the time overhead.

Indeed, the only data set that provides any encouraging outcome is the random one. However, this only highlights that the way humans write programs is quite specific and structurally organized.

There was some hope of positive results for the simple student data, unfortunately to no avail. It seems that, per average, students are able to intrinsically grasp the basic concepts of ASP optimization, and supply relatively good code.

6.6 Declarative vs. imperative algorithms

As an addendum, the effectiveness of the declarative vs. the imperative variants for eligibility testing will be briefly discussed here. For this, the environment was set up in the following way:

- all programs from all data sets were added into the data set for this analysis,
- the transformation system was set to only one transformation per run, either $\{SUB\}$ or $\{LSH\}$, and
- there was one run per algorithm variant, giving four runs in total.

The results are given in Table 6.3.

Although the *SUB* result is inconclusive and might be influenced by the solver used, it is immediately apparent that in the case of *FD-LC*$_{0\text{-}1\text{-}0}$ the implementations of declarative algorithms take far longer to process (well over 10 times). This suggests that there is little use in utilizing ASP for solving tasks that can be relatively easily expressed and processed using other programming paradigms. In addition, there is a large overhead due to the need to encode the problem into answer-set programming, which contributes further to the slowdown, especially in case of non-experts.

Chapter 7

Conclusion and Outlook

This thesis examined the viability of several transformation rules studied in the literature for answer-set programs in a practical setting. An overview of the most prevalent ones has been presented. A selection has been made by constructing two independent transformation systems for the examinations: $\{FD\text{-}LC_{0\text{-}1\text{-}0}, SUB, FOLD, LSH\}$ for syntactical analysis and $\{SRRS, SRLS\}$ for semantical elimination.

As a consequence, algorithms have been researched and developed, both imperative and declarative, in order to automate the transformation applicability checking. After this, the two systems have then been examined for their properties, and methods have been found to apply them. Using the thus gained knowledge, a test setup has been coded in Python, for the purpose of experimenting with a number of data sources – both from front-ends as well as other data, in particular, student-created programs and random programs.

The results show that, at least in the case of the transformations used, their utility on actual programs is severely limited. This is for two reasons.

Firstly, the time it takes for the analysis to finish is, in a number of cases, prohibitively long. This delay prevents a number of the transformations from being used in optimization, such as an assistance tool for an Integrated Development Environment.

The second reason is the simple case of effectiveness – or rather, the lack thereof. For most data sets, the number of transformations actually detected to be applicable and performed is low to non-existent. The analysis of the programs written by students has given especially disappointing results, since it was hoped that beginners to ASP would simply make less efficient programs, yet it does not seem so.

One must keep in mind, however, that the data supplied in the latter case are final submissions. It would perhaps be interesting to check whether transformation applicability detection can provide help during the initial program writing process, of course among the beginners' group. This presents itself as a viable avenue for further research.

Bibliography

[BD95] Stefan Brass and Jürgen Dix. Characterizations of the stable semantics by partial evaluation. In *Proceedings of the 3rd International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 1995*), volume 928 of *Lecture Notes in Computer Science*, pages 85–98. Springer, 1995.

[BZF96] Stefan Brass, Ulrich Zukowski, and Burkhard Freitag. Transformation-based bottom-up computation of the well-founded model. In *Proceedings of the 6th Workshop on Non-Monotonic Extensions of Logic Programming* (*NMELP 1996*)*, Selected Papers*, volume 1216 of *Lecture Notes in Computer Science*, pages 171–201. Springer, 1996.

[CPV07] Pedro Cabalar, David Pearce, and Agustín Valverde. Minimal logic programs. In *Proceedings of the 23rd International Conference on Logic Programming* (*ICLP 2007*), volume 4670 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2007.

[DFI+03] Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerald Pfeifer. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in DLV. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence* (*IJCAI 2003*), pages 847–852. Morgan Kaufmann, 2003.

[DST01] James P. Delgrande, Torsten Schaub, and Hans Tompits. plp: A generic compiler for ordered logic programs. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2001*), volume Volume 2173 of *Lecture Notes in Computer Science*, pages 411–415. Springer, 2001.

[DST03] James P. Delgrande, Torsten Schaub, and Hans Tompits. A framework for compiling preferences in logic programs. *Theory and Practice of Logic Programming*, 3(2):129–187, 2003.

[EF03] Thomas Eiter and Michael Fink. Uniform equivalence of logic programs under the stable model semantics. In *Proceedings of the 19th International Conference on Logic Programming* (*ICLP 2003*), volume 2916 of *Lecture Notes in Computer Science*, pages 224–238. Springer, 2003.

[EFL+03] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A logic programming approach to knowledge-state planning, II: The DLV^K system. *Artificial Intelligence*, 144(1-2):157–211, 2003.

[EFL+04] Eiter, Faber, Leone, Pfeifer, and Polleres. A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Transactions on Computational Logic*, 5:206–263, 2004.

[EFLP99] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. The diagnosis frontend of the dlv system. *AI Communications*, 12(1-2):99–111, 1999.

[EFT+06] Thomas Eiter, Michael Fink, Hans Tompits, Patrick Traxler, and Stefan Woltran. Replacements in non-ground answer-set programming. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning* (*KR 2006*), pages 340–351, 2006.

[EFTW04a] Thomas Eiter, Michael Fink, Hans Tompits, and Stefan Woltran. On eliminating disjunctions in stable logic programming. In *Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning* (*KR2004*), pages 447–458. AAAI Press, 2004.

[EFTW04b] Thomas Eiter, Michael Fink, Hans Tompits, and Stefan Woltran. Simplifying logic programs under uniform and strong equivalence. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2004*), volume 2923 of *Lecture Notes in Computer Science*, pages 87–99. Springer, 2004.

[Fan04] Hans Fangohr. A comparison of C, Matlab and Python as teaching languages in engineering. In *Proceedings of the 4th International Conference on Computational Science* (*ICCS 2004*), volume 3039 of *Lecture Notes in Computer Science*, pages 1210–1217. Springer, 2004.

[FLPP04] Wolfgang Faber, Nicola Leone, Simona Perri, and Gerald Pfeifer. System description: DLV with aggregates. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2004*), volume 2923 of *Lecture Notes in Computer Science*, pages 326–330. Springer, 2004.

[FPTW07] Michael Fink, Reinhard Pichler, Hans Tompits, and Stefan Woltran. Complexity of rule redundancy in non-ground answer-set programming over finite domains. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2007*), volume 4483 of *Lecture Notes in Computer Science*, pages 123–135. Springer, 2007.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and*

Symposium on Logic Programming (*ICLP 1988*), pages 1070–1080. MIT Press, 1988.

[GL91] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[Hei07] Andreas Heindl. On replacements in answer-set programming based on partial evaluation. Master's thesis, Vienna University of Technology, Institute of Information Systems, 2007.

[Hin02] Konrad Hinsen. High-level scientific programming with python. In *Proceedings of the 2nd International Conference on Computational Science* (*ICCS 2002*)*, Part III*, volume 2331 of *Lecture Notes in Computer Science*, pages 691–700, 2002.

[LC07] Fangzhen Lin and Yin Chen. Discovering classes of strongly equivalent logic programs. *Journal of Artificial Intelligence Research*, 28:431–451, 2007.

[LPF+06] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[LPV01] Vladimir Lifschitz, David Pearce, and Agustín Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.

[Mah88] Michael J. Maher. Equivalences of logic programs. In *Foundations of Deductive Databases and Logic Programming*, pages 627–658. Morgan Kaufmann, 1988.

[ONA01] Mauricio Osorio, Juan A. Navarro, and José Arrazola. Equivalence in answer set programming. In *Proceedings of the 11th International Workshop on Logic Based Program Synthesis and Transformation* (*LOPSTR 2001*), volume 2372 of *Lecture Notes in Computer Science*, pages 57–75. Springer, 2001.

[ONG01] Mauricio Osorio, Juan Carlos Nieves, and Chris Giannella. Useful transformations in answer set programming. In *Proceedings of the 1st International Workshop on Answer Set Programming* (*ASP 2001*), 2001.

[OSTW06] Johannes Oetsch, Martina Seidl, Hans Tompits, and Stefan Woltran. ccT: A correspondence-checking tool for logic programs under the answer-set semantics. In *Proceedings of the 10th European Conference, on Logics in Artificial Intelligence* (*JELIA 2006*), volume 4160 of *Lecture Notes in Computer Science*, pages 502–505. Springer, 2006.

[Pea04] David Pearce. Simplifying logic programs under answer set semantics. In *Proceedings of the 20th International Conference on Logic Programming (ICLP 2004)*, volume 3132 of *Lecture Notes in Computer Science*, pages 210–224. Springer, 2004.

[Tra06] Patrick Traxler. Techniques for simplification of disjunctive datalog programs with negation. Master's thesis, Vienna University of Technology, Institute of Information Systems, 2006.