Program-Rewriting Transformations for Datalog

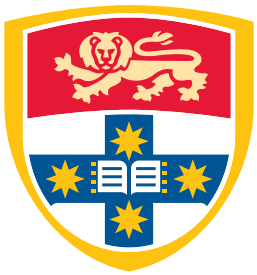
ABDUL ZREIKA
SID: 440206640

Supervisor: Dr. Bernhard Scholz

This thesis is submitted in partial fulfillment of
the requirements for the degree of
Bachelor of Science (Advanced Mathematics) (Honours)

School of Information Technologies
The University of Sydney
Australia

28 June 2019



THE UNIVERSITY OF SYDNEY

Student Plagiarism: Compliance Statement

I certify that:

I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure;

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to the University commencing proceedings against me for potential student misconduct under Chapter 8 of the University of Sydney By-Law 1999 (as amended);

This Work is substantially my own, and to the extent that any part of this Work is not my own I have indicated that it is not my own by Acknowledging the Source of that part or those parts of the Work.

Name: Abdul Zreika

Signature:  **Date**: 06/11/18

Abstract

Datalog is a declarative language that has seen a recent resurgence in various fields of computer science, including static program analysis, security, and network analysis. The new application domains of Datalog require the computation of billions of tuples, and so the scalability of modern Datalog programs is paramount. Datalog is a decidable fragment of logic for which there exist efficient evaluation strategies. To achieve scalability for very large Datalog instances, however, programmers must provide "performance hints" to the current generation of Datalog engines. Hints are non-declarative features in the Datalog code that guide the evaluation of a Datalog engine. For example, a hint can be a reordering of atoms in a body to suggest a different execution order by rewriting the Datalog code, or an annotation that pre-empts performance decisions made by the Datalog engine. Providing effective hints is a time-consuming activity for the programmer, and requires a deep understanding of the internals of a Datalog engine. There is, therefore, the question of whether the next generation of Datalog engines can find performance hints automatically, making the programmer more productive.

In this thesis, we introduce a series of high-level program-rewriting transformations to exploit performance opportunities in Datalog programs. The transformations generate hints, producing optimised code that executes faster on a bottom-up Datalog engine. These program-rewriting transformations work more effectively in concert. To exploit the synergy between the transformations, we introduce a transformation pipeline to coordinate the effects of each individual transformer. We measured the performance benefits of individual transformations and the transformer pipeline for synthetic and real-world benchmarks. Some of the transformations enabled the execution of larger instances, i.e. without transformations the instance would not terminate its execution. On real-world benchmarks, such as the Amazon AWS industrial application for network-security analysis, the effect of individual transformers reduced evaluation time and memory usage by up to 68% and 84% respectively. The synergy of transformers improved performance on real-world industrial benchmarks further, with a 98% reduction in evaluation time, running up to 57 times faster.

Acknowledgements

First and foremost, a huge thank you to my supervisor, Bernhard Scholz, for being possibly one of the greatest supervisors in existence. I sincerely appreciate your helpfulness, patience, and invaluable discussions throughout this year. The breadth of your knowledge has been equal parts inspiring and frightening.

Thank you to everyone else in the programming languages research group for your wisdom and support: David, Martin, Rachel, Brody, Lexi, Lyndon, Patrick, and past members. The weekly meetings have been illuminating and so much fun, despite being more or less set at dawn.

Thank you also to the people at Amazon, including Sean McLaughlin and Byron Cook, for providing us with a set of practical examples for this thesis.

Thank you to all my honours peers: Jonathan, Daniel, Vincent, and especially Michael Rizzuto, who has been both the main protagonist and antagonist in this story.

Finally, a special thank you to my family and friends, including Courtney, David, Martin, Tuba, Derrick, Noor, Vincey, my students, and everyone else for their honours support packages and snacks. Had I known this was in store, I would have pretended to be writing a thesis years ago.

CONTENTS

List of Figures

List of Tables

CHAPTER 1

Introduction

Datalog has seen a recent resurgence in various fields of computer science as a domain-specific logic-programming language. The ability to represent programs as concise logical specifications makes it naturally applicable to many classes of problems, such as static program analysis (Smaragdakis and Bravenboer, 2011; Bravenboer and Smaragdakis, 2009), deductive databases (Minker, 2014), network analysis (Loo et al., 2005b,a; Abiteboul et al., 2005), and security (Jim, 2001; Marczak et al., 2010). The declarative nature of Datalog programs allows programmers to focus on the formal specifications of the problem, rather than the process towards it; that is, *what* to do, rather than *how* to do it. The result is reduced complexities in program development and maintenance, and more natural techniques for program verification. Datalog becomes a tool of choice for rapid-prototyping and the concise description of problems.

The reluctance to accept Datalog as a mainstream industrial tool in the past stemmed from its initial inability to perform to the same standards as manually developed tools (Scholz et al., 2016). Several application domains appropriate for Datalog, such as program analysis, typically involve the computation of billions of tuples (Antoniadis et al., 2017). The problem of scalability and efficiency is tied to the evaluation strategies of logical programs. Due to the inherently imperative nature of current computational technology, Datalog engines must transform these declarative programs into an equivalent imperative form through the use of evaluation strategies. Typical evaluation strategies either evaluate a goal query from the facts to the goal, known as bottom-up evaluation, or from the goal to the facts, known as top-down evaluation (Abiteboul et al., 1995). Alongside choosing the basic strategy for evaluation, there are several decisions to be made by a Datalog engine which will determine the performance of a program.

The evaluation strategy and its concrete implementation therefore directly impact the scalability of Datalog programs. In general, the current generation of Datalog engines have suboptimal evaluation strategies. The strategies are too generic, and cannot harvest the optimisation potential of a Datalog program automatically. To fine-tune the evaluation strategy for a given Datalog program, programmers provide "performance hints" to the Datalog engine.

A hint is a non-declarative feature of the logic program that guides the evaluation process of the underlying engine for a given program. Hints can be manual annotations by the programmer to pre-empt performance decisions. Alternatively, a hint could be a rewriting of the Datalog program into a new semantically equivalent form that performs better on the engine, due to a differing imperative translation. By writing a Datalog program in a particular way, the underlying engine can evaluate a program more optimally using these hints without altering program output. Note that, although two Datalog programs can be identical in what they represent, evaluation performance could vary dramatically (Sereni et al., 2008; Ullman, 1989; Tekle and Liu, 2011).

In the world of declarative programming, providing hints to find a more suitable evaluation strategy is a detriment to the actual goal of being "declarative". The programmer should be shielded from the burden of imperative programming, including performance considerations in evaluation. However, the current generation of Datalog engines, such as SOUFFLÉ (Jordan et al., 2016), LogicBlox (Aref et al., 2015), and bddbddb (Whaley, 2006), require hints to meet the scalability demands of real-world applications. They all employ a wide range of highly-efficient data-structures and fine-tuned bottom-up evaluation strategies to effectively evaluate Datalog programs. Neverthless, they can only compete with state-of-the-art hand-written tools if hints are provided (Bancilhon et al., 1985; Jordan et al., 2016; Tekle and Liu, 2011; Sagiv, 1988; Balbin et al., 1991).

Unfortunately, providing hints that improve performance can be a very time-consuming task for programmers, and requires a deep understanding of the internals of the Datalog engine in question. Implementing Datalog program transformations that generate these hints for the backend evaluation system therefore has significant potential for the next generation of Datalog engines.

In this thesis, we introduce a series of high-level program-rewriting transformations to exploit performance opportunities in Datalog programs. The aim of these transformations is to make user hints superfluous, i.e., the transformations find hints to improve the evaluation strategy fully automatically. The

transformations were implemented in the open-source SOUFFLÉ Datalog engine (soufflé, 2018). SOUFFLÉ is a high-performing state-of-the-art bottom-up Datalog engine implementing low-level techniques in the backend to far exceed the scalability and efficiency of other existing translators (Scholz et al., 2016; Jordan et al., 2016). Despite its capabilities, a major limitation of SOUFFLÉ is that it does not provide automated rewriting transformations to adapt the evaluation strategy for a given input program.

The program-rewriting transformations can often also have complementary effects (Ullman, 1989). One transformation can reveal an optimisation potential that can be used by subsequent transformations. To coordinate the synergy between the program-rewriting transformations, we introduce a configurable transformation pipeline. We conduct a wide range of experiments, including on industrial-strength applications deployed in industry, such as AWS. In our experiments, we show the performance benefits of the individual transformations on both synthesised and real-world benchmarks, especially when working in concert, dealing with relations containing billions of tuples. With the new set of transformations, we reduce the reliance on programmer-provided hints to obtain scalability.

The work we present is unique, as program-rewriting techniques for logic programs have rarely been studied, bar standard rewriting transformations (Bancilhon et al., 1985) mainly aimed at using Datalog as a database query language. The work on transformer interactions is particularly limited. We present a new class of rewriting transformations, particularly useful for the new application domains of Datalog programs, including security, network analysis, and static program analysis. These domains exhibit programs with hundreds of relations and rules, and go beyond the classical use-case of Datalog programming.

1.1 Contributions

In this thesis, we present the following contributions:

- We develop a series of high-level program-rewriting transformations.
  - `InlineRelations`: Relations can be chosen to be inlined by the user, reducing the overhead of constructing intermediate non-output relations.
  - `ReduceExistentials`: Existential relations are transformed into propositional, non-recursive counterparts.

  - `ReplaceSingletonVariables`: Singleton variables are replaced with unnamed variables to indicate to the engine that an existential check suffices, rather than the typical full relation scan.
  - `PartitionBodyLiterals`: Disconnected components in clauses can be extracted out into separate propositional relations.
  - `ReorderLiterals`: Literals in rule bodies can be reordered based on a given heuristic.
  - `MinimiseProgram`: Redundant clauses semantically equivalent to other existing clauses in the program are identified and removed.
- We develop a theoretically justified transformation pipeline, based on transformer dependencies.
- We implement the transformations and the pipeline concept into the existing SOUFFLÉ Datalog engine.
- We conduct a range of experiments on large-scale programs, including both synthesised and real-world benchmarks, to evaluate the impact of the transformers individually and in concert.

Our rewriting transformations have a direct impact on real-world benchmarks. In particular, we evaluated our contributions on an industrial application used by Amazon to check the security properties of AWS software-defined networks. Our transformations obtain a speedup of 57x (98%), reducing the runtime from 285 seconds to just 5 seconds.

1.2 Organisation

We begin in Chapter 2, with some background information and a review of existing related work. Chapter 3 introduces each developed program-rewriting transformation. Each transformation is motivated, and includes an algorithm, necessary transformation conditions, and an outline of its implementation in SOUFFLÉ. In Chapter 4, we develop a new pipeline transformation framework. The framework is used to develop a transformation pipeline for SOUFFLÉ by analysing transformer dependencies. In Chapter 5, we conduct a range of experiments to test the effectiveness of our contributions on synthesised and real-world benchmarks, and discuss the results. We discuss future work in Chapter 6, and conclude in Chapter 7.

CHAPTER 2

Background and Literature Review

In this chapter, we provide an overview of the Datalog language and existing work in the area of Datalog program optimisations. We begin with an introduction to Datalog syntax and semantics, followed by popular techniques for program evaluation. We then review current work in the field of Datalog optimisations, discussing both high-level program-rewriting strategies and backend evaluation strategies. We conclude with a brief description of SOUFFLÉ: a state-of-the-art Datalog evaluation engine with a high-performing backend, but a poor framework for program-rewriting optimisations.

2.1 Datalog as a Fragment of Logic

2.1.1 Datalog Syntax and Semantics

Datalog is a domain-specific decidable fragment of logic. It is syntactically a subset of the general-purpose logic programming language, Prolog. As with Prolog, Datalog programs are expressed as a set of statements, called *rules*. Each rule is a *Horn clause*, written in the form:

$$p_0(x_{0_1}, x_{0_2}, \ldots, x_{0_{k_0}}) \leftarrow p_1(x_{1_1}, x_{1_2}, \ldots, x_{1_{k_1}}), \ldots, p_n(x_{n_1}, x_{n_2}, \ldots, x_{n_{k_n}}).$$

Each symbol p_i represents a *predicate* (or, in database terms, a *relation*) with arity k_i. An *atom* is a predicate p_i supplied with k_i arguments, called *terms*. Terms are comprised of constants and variables. The *head* of a rule is the atom $p_0(x_{0_1}, x_{0_2}, \ldots, x_{0_{k_0}})$ appearing on the left hand side of the arrow. For notational brevity, we will use $a(\vec{x})$ to represent an atom $a(x_1, \ldots, x_n)$ with its arguments. An atom is *grounded* if none of its terms contains variables.

In terms of semantics, an atom $p_i(x_{i_1}, x_{i_2}, \ldots, x_{i_{k_i}})$ is said to be *true* if the tuple $(x_{i_1}, x_{i_2}, \ldots, x_{i_{k_i}})$ exists in the relation p_i. If all atoms in the body of a rule are true for a given assignment of variables,

then the atom in the head of the rule must also be assigned a true value for that assignment. In the case that the body is empty, the head of the rule is immediately true, and we call it a *fact* in the database. The set of all facts is called the *extensional database* (EDB), and the complement is called the *intensional database* (IDB).

Datalog usually also allows atoms to be negated. A concrete semantic definition for negation is difficult (Greco and Molinaro, 2015). Datalog typically operates under the Closed-World Assumption: all things that are true in the universe can be proven to be true by the supplied database; that is, the input database is complete. Intuitively, $\neg p(X)$ is therefore said to be true if and only if we cannot prove $p(X)$ to be true with the given IDB and EDB. A *literal* refers to any atom or its negation.

For any given Datalog program, we can supply a query of the form $\leftarrow p$, which outputs all tuples $X = (x_1, \ldots, x_k)$ where $p(X)$ is true given a set of IDB and EDB relations. Computation in a Datalog program is the discovery of the minimal set of tuples for a given query such that all rules in the program hold.

2.1.2 Datalog Restrictions

The primary purpose of Datalog programming over Prolog is the guaranteed termination of programs when operating over a finite universe. To provide this, certain semantic restrictions must be imposed on the Datalog program that guarantee decidability, at the cost of expressiveness.

The most fundamental semantic restriction is that all variables in the head atom must appear in some body atom. This means that all tuples proven have only constant terms; they are independent of any variables in the program. The Datalog program can then be evaluated using a simple fixpoint procedure, based on the concept of *bottom-up evaluation* discussed in the next section. Essentially, starting with the set of facts, all rules are applied repeatedly until no new tuples can be derived.

Pure Datalog is the most restricted form of Datalog programming, whereby all atoms must be positive, and all terms are strictly either constants or variables. Since all terms are constants or variables, no new constants can be introduced outside of those already present in the EDB, and so only a finite number of assignments are possible. As a result, all queries are guaranteed to terminate. In fact, pure Datalog programs run in polynomial time (McAllester, 1999).

In many applications, allowing more complex terms through the use of *functors* is necessary. Functors are functions of a fixed arity k that take in k elements from the universe and return a single element. For example, the addition operator $+$ is a binary functor over the universe of real numbers. Some problems, such as computing the sum of weights along a path in a weighted directed acyclic graph, require functors to be expressed. Most Datalog implementations hence extend pure Datalog with a notion of built-in functors. To preserve the fixpoint evaluation strategy, all variables appearing in functors must also appear functorless in some body atom. Complex terms are then only evaluated after all constituent variables have been bound by their functorless appearance (Greco and Molinaro, 2015; Abiteboul et al., 1995). Due to the violation of the finite universe assumption, however, adding functors shifts Datalog programs from the decidable world into undecidability, as unrestricted constants may be generated from functors.

Finally, many logical statements can only be expressed using the concept of negation. As stated previously, $\neg p(X)$ is defined to be true if and only if we cannot prove $p(X)$ to be true with the given IDB and EDB. To enact this semantic meaning in practice, whilst also maintaining the termination guarantee and fixpoint evaluation, we restrict negation such that our program can be *stratified*. A program is stratified if the dependency graph between predicates does not contain a negative cycle. That is, there cannot be a sequence of predicates $(p_1, p_2, \ldots, p_n, p_1)$ such that p_{i+1} appears negated in a rule of p_i. If a program can be stratified, then we can divide the program into a sequence of *strata* $(S_1, \ldots, S_n)$ such that all predicates p appearing negated in some rule in stratum S_i have already been fully computed in a previous stratum. Thus, we can compute the fixpoint of each stratum in sequence, piping in the results of the previous stratum as facts into the next. Although a single program can admit many stratifications, the resulting fixpoint is independent of the choice in strata (Greco and Molinaro, 2015; Abiteboul et al., 1995).

Checking whether the negation of a grounded atom is true can always be performed with a simple lookup in stratified programs, as the associated relation must have been fully computed in a previous stratum. By restricting negation further, such that all variables appearing in negated atoms also appear in a positive atom in the rule, we can treat negation as a constraint: once the variables in the negated atom are bound to a value, the now grounded negated atom can be checked.

Datalog with negation semantics is called *normal Datalog*. When discussing Datalog throughout, we assume the semantics of normal Datalog with functors, unless otherwise specified.

The code shown in Listing 2.1 is an example of a program written in Datalog that outputs all edges in the transitive closure of a directed graph with a given set of edges. Note that the arrow, ←, is dropped when the body of a rule is empty. A rule with no head atom indicates the relation we wish to query.

LISTING 2.1. Transitive closure in Datalog

```
1  edge(1,2).
2  edge(1,4).
3  edge(2,3).
4
5  transitive_closure(x,y) ← edge(x,y).
6  transitive_closure(x,y) ← edge(x,z), transitive_closure(z,y).
7
8  ← transitive_closure
```

2.2 Datalog Evaluation

As computers in the real-world are inherently procedural, Datalog programs must be translated into an imperative style to be evaluated. There are several techniques devised for the imperative evaluation of Datalog programs, typically classified as either top-down or bottom-up evaluation strategies.

2.2.1 Bottom-Up Evaluation

Bottom-up evaluation strategies begin from the set of facts and work towards the query.

The most basic form of bottom-up evaluation is an iterative fixpoint procedure, termed *naïve evaluation* (Bancilhon and Ramakrishnan, 1988). The idea is to start with an empty database, and repeatedly apply all rules until no new tuples can be generated. The procedure is summarised in Algorithm 1.

Algorithm 1 Naïve evaluation of a Datalog program

```
1:  procedure EVALUATEDATALOGPROGRAM(EDB, IDB)
2:      database ← ∅
3:      δ ← EDB
4:      while δ ≠ ∅ do
5:          database ← database ∪ δ
6:          δ ← ∅
7:          for each rule r ∈ IDB do
8:              derivedTuples ← APPLYRULE(r,database)
9:              newTuples ← derivedTuples \ database
10:             δ ← δ ∪ newTuples
11:     return database
```

The derived fixpoint is unique and correct (Greco and Molinaro, 2015; Liu and Stoller, 2009). Naïve evaluation is the most widely used and described Datalog evaluation strategy (Bancilhon and Ramakrishnan, 1988) due to the ease of implementation and the natural mapping from the declarative program. The iterative non-goal-oriented nature of naïve evaluation means we are likely to constantly produce irrelevant facts that do not contribute to the given query (Bancilhon et al., 1985). In particular, relations are always fully computed, even if only a small subset is used, and duplicate tuples are likely to be generated.

2.2.2 Top-Down Evaluation

Top-down evaluation strategies are a goal-directed alternative to bottom-up strategies, where evaluation begins from the query and works towards the facts instead. The idea is that the query can only be true when all atoms in the body can be satisfied. Each atom a in the body can in turn only be true when all atoms in the body of at least one rule of a can be satisfied. For a given body atom, we can hence try out a rule by substituting in its body. When substituting, we may be forced to bind certain variables to each other or to constants. The substitution process is called *unification*. We continuously expand and unify the body literals in this manner until we reach an EDB fact, of when unification is impossible. If a particular path fails, we backtrack and try another rule substitution, until all rule possibilities have been tested.

Prolog uses a top-down evaluation strategy called *SLD resolution*. SLD resolution defines a mechanical approach to top-down evaluation: atoms in queries are expanded left to right, choosing the next rule to substitute from top to bottom, based on the source code.

Although (functorless) Datalog programs always admit a finite minimal model (Greco and Molinaro, 2015; Abiteboul et al., 1995), top-down evaluation methods are not guaranteed to terminate (Bancilhon et al., 1985). Moreover, top-down evaluation strategies tend to repeatedly generate the same result when expanding body atoms. The need for recomputation can be mitigated through tabling techniques, such as variant or subsumptive tabling, where the truth values of certain forms of atoms are stored and reused (Bancilhon et al., 1985; Chen and Warren, 1996; Tekle and Liu, 2011).

2.2.3 Bottom-Up vs. Top-Down

Top-down evaluation strategies avoid the computation of irrelevant tuples inherent to bottom-up methods. By deriving cost-functions based on different variants of top-down and bottom-up strategies on several simple Datalog programs, it has been shown that neither evaluation strategy class is superior to the other in all cases (Bancilhon and Ramakrishnan, 1988).

Interestingly, however, all Datalog programs P can be rewritten into a semantically equivalent form P' such that evaluating P' using semi-naïve evaluation, discussed in the next section, is always better than or equal to evaluating P top-down when considering the number of rule firings (Ullman, 1989). Bottom-up strategies can therefore always trump top-down methods, provided that the original program is first translated appropriately. Unless explicitly stated, we assume a bottom-up strategy when discussing Datalog evaluation engines.

2.3 Optimising Program Evaluation

The development of a Datalog engine scalable and efficient enough to meet the demands of industrial applications has often been met with skepticism (Scholz et al., 2016; Neuhold et al., 1988; Ceri et al., 1989). In particular, an unsophisticated approach to program evaluation does not lend promising results, especially when compared with tools developed in other languages (Scholz et al., 2016). The most important optimisations to meet this baseline lie in the use of effective evaluation strategies, and the development of program transformations that rewrite the original Datalog code into a more imperatively efficient form.

2.3.1 Evaluation Strategies

As discussed in the previous section, the evaluation strategy used can have a significant impact on program efficiency, and even termination. The choice of an appropriate evaluation strategy is hence critical in the development of a Datalog engine fit for more general use.

Bottom-up evaluation can be optimised in practice by using incrementally maintained variables to represent the current environment, without computing expensive set relations (Liu and Stoller, 2009). Moreover, several efficient data-structures specialised for fixpoint computations have been developed in the past (Jordan et al., 2016; Liu and Stoller, 2009).

The naïve approach shown in Algorithm 1 can also be extended to reduce the number of duplicate tuples generated (Bancilhon and Ramakrishnan, 1988). The idea is based on the fact that, in each iteration, at least one newly-generated tuple from the previous level must be used. The resulting new approach is called *semi-naïve* evaluation. Using semi-naïve evaluation for general Datalog programs is only a recent advancement, but has shown very promising results in practice (Scholz et al., 2016; Ramakrishnan et al., 1994; Whaley et al., 2005).

Although the choice of evaluation strategy plays a significant role in program optimisation, previous work showing the superiority of bottom-up evaluation strategies given 'sufficiently good' program rewrites indicates that program-rewriting transformations may be a more valuable avenue in maximising evaluation efficiency.

2.3.2 Program-Rewriting Strategies

Program-rewriting optimisations involve reformulating a given Datalog program in a semantically equivalent manner to provide hints to the backend evaluation system, based on an analysis of the program. Here, semantic equivalence means the output of the new program is identical to the output of the original. That is, the tuples in query relations must remain unchanged, while intermediate computations may differ. Existing program-rewriting transformations typically operate on Datalog programs with restrictions.

2.3.2.1 The Magic-Set Transformation

The most well-studied program-rewriting transformation is the Magic-Set Transformation (MST). Recall that the main issue with bottom-up evaluation strategies was the generation of irrelevant tuples (Bancilhon et al., 1985). The issue is seemingly inherent to bottom-up methods due to the non-goal-oriented nature of the technique. The MST aims to reduce wasted computations through complex rule specialisations.

The initial MST technique was devised only for pure Datalog programs (Bancilhon et al., 1985). The analysis stage involved discovering the information flow between variables in each individual rule. As Datalog programs are declarative, and information flow is a strictly imperative concept, the ordering of atom execution in each rule is made deterministic through a *Sideways Information Passing Strategy* (SIPS). The SIPS is a heuristic used to determine which atom should be chosen next, typically based on a boundness analysis. An *adornment* is calculated for each atom in each rule, representing which of its arguments are bound to a value at execution time. Based on these adornments, we can predict at compile-time a restricted set of values that a variable can take. Consequently, we can add specialising predicates to the start of each rule that enable this restriction. The result is likely fewer irrelevant computations per rule. The work was later extended to work on functorless normal Datalog programs (Balbin et al., 1991).

The effectiveness of the MST in practice is debated (Ullman, 1989; Sereni et al., 2008; Bancilhon et al., 1985; Tekle and Liu, 2011). In his paper on bottom-up evaluation, Ullman (Ullman, 1989) shows an example of a program performing poorly after the MST is applied. Bancilhon and Ramakrishnan (Bancilhon and Ramakrishnan, 1988) show analytically that the performance of the MST varies even with simple programs. Sereni et al. (Sereni et al., 2008) finally experimentally showed that, although program evaluation can be significantly faster when the MST is applied, there are cases where speed degrades drastically. The authors ran 111 standard Datalog programs with and without MST, and showed that, while approximately 30% had up to a 4x increase in speed, 10% were more than 3x slower.

The SIPS should be chosen in such a way that each successive body maximises the cut down of tuples to consider. Two natural approaches are to always execute the maximally bound atom (Balbin et al., 1991) or to always execute the atom for which the predicted relation size is minimal (Sereni et al., 2008). The latter is significantly more complex to implement, though it has been shown to improve evaluation time in many cases (Sereni et al., 2008).

2.3.2.2 Program Minimisation

The presence of redundant clauses and atoms in Datalog programs can also lead to unnecessary recomputations. For a Datalog program P, program minimisation removes these redundancies by returning the minimal equivalent subset P' of P. Unfortunately, checking the general equivalence of two programs is undecidable (Sagiv, 1988), due to the undecidability of general clausal equivalence (Shmueli, 1993; Ramakrishnan et al., 1988). *Uniform equivalence* is a stronger form of equivalence, where IDB relations can be given input facts. Uniform equivalence is a decidable property for functorless Datalog (Sagiv, 1988). Uniform equivalence can be tested by running a simpler Datalog program with a particular set of inputs (Sagiv, 1988). Unfortunately, the process cannot be immediately extended to work with functors.

2.3.2.3 Existential Queries

In some Datalog programs, certain arguments of a relation may only be used existentially. For example, consider the following snippet of a Datalog program, assuming that `b` and `c` have been previously defined:

```
1   a(x,y) ← b(x,z), c(z,y).
2   query(x) ← a(x,y).
3   ← query.
```

Clearly, the value of the variable `y` on line 2 does not matter, as it is not used elsewhere. We call a variable that appears a single time in a rule a *singleton*. Singleton variables can be represented by unnamed variables in Datalog, typically written as underscores '_'. Assuming that the relation `a` is not used elsewhere, not all possible values for the second argument need to be computed; we only want to know if any such value exists, given a fixed first argument `x`. A paper by (Ramakrishnan et al., 1988) rewrites such programs more optimally by first adorning rules, and then deducing which arguments can be deleted. The adornment process is similar to that in the magic-set transformation earlier, except based on existential arguments, rather than boundness. The removal of arguments may cause clauses to become equivalent. The authors use uniform equivalence in an attempt to minimise the program, while noting that minimising under general equivalence is undecidable. The importance of an interaction between transformers is therefore implied. The paper also briefly discusses removing disconnected literals in clauses into a single separate propositional relation to simulate the *cut* operator in Prolog. No experimental justification is given for the discussed transformations, however.

2.3.2.4 Other Techniques

Other techniques for program-rewriting optimisations have been researched to a much lesser degree.

The most commonly discussed alternative optimisation, called *counting*, is in fact based on the MST itself (Bancilhon et al., 1985). The counting method applies to the very restricted class of Datalog programs containing only linear acyclic relations. The restriction offers further avenues for minimising the number of generated tuples. A complexity analysis shows that the counting method can be superior to the MST in some cases.

The remaining literature on optimisations is very limited. In particular, there are several undiscussed program-rewriting optimisations that can be derived from typical imperative approaches to optimised compiler development. For example, constant propagation, statement reordering, and relation inlining all seem to be promising optimisations that have not yet been considered.

2.3.2.5 Multi-Optimisational Systems

Many papers discuss the impacts of individual optimisations relative to each other, both mathematically (Bancilhon et al., 1985; McAllester, 1999; Liu and Stoller, 2009) and empirically (Jordan et al., 2016; Sereni et al., 2008; Antoniadis et al., 2017). Conversely, there has been very limited work comparing their impacts when run together or in sequence (Ceri et al., 1989). Ullman (Ullman, 1989) discovers the effectiveness of bottom-up evaluation only through a composition of multiple transformations, and so this largely untapped field is promising.

2.4 SOUFFLÉ

SOUFFLÉ is a modern open-source Datalog engine primarily designed for large-scale applications (Scholz et al., 2016). The intricate low-level optimisation techniques used in the backend evaluation strategy allow SOUFFLÉ to far exceed the scalability and performance of other existing translators (Scholz et al., 2016; Jordan et al., 2016; Antoniadis et al., 2017), such as bddbddb (Whaley, 2006) and μZ (Hoder et al., 2011). In fact, SOUFFLÉ achieves results on par with manually developed tools (Jordan et al., 2016). Nevertheless, the lack of non-trivial program-rewriting optimisations, which have been shown to have enormous impacts on program efficiency, means there are several untapped avenues for optimisation. SOUFFLÉ hence serves as a suitable vehicle for understanding and extending such optimisations,

and analysing their impacts on the performance of Datalog systems, especially for large-scale real-world applications.

2.4.1 SOUFFLÉ Syntax

SOUFFLÉ is an extension of normal Datalog with functors. Predicate arguments are typed as either `symbol` (a string) or `number`. SOUFFLÉ also allows the use of aggregators, components, and records to improve expressiveness. To limit our scope and maintain the applicability of our transformers to general Datalog, we omit further discussion of such constructs. In SOUFFLÉ, predicates must be declared. Output and input relations can be specified with a declarative. Listing 2.2 shows an implementation of the transitive closure Datalog example from Listing 2.1 in SOUFFLÉ.

LISTING 2.2. Transitive closure in SOUFFLÉ

```
1  .decl edge(x:number, y:number)
2  .input edge(filename="edge.facts")
3  edge(1,2).
4  edge(1,4).
5  edge(2,3).
6
7  .decl transitive_closure(x:number, y:number)
8  transitive_closure(x,y) :- edge(x,y).
9  transitive_closure(x,y) :- edge(x,z), transitive_closure(z,y).
10
11 .output transitive_closure
```

The `.decl` keyword declares a new predicate, assigning each argument a type. The `.output` declarative is used to specify one or more query relations, the contents of which will be the output of the program. The `.input` declarative reads in facts from a fact file, in this case specified to be `"edge.facts"`. SOUFFLÉ also offers notation for disjunctions, through the use of a semicolon (`;`). For example, the rule:

```
1  a(x) :- (b(x),c(x)) ; (d(x),e(x)).
```

is equivalent to:

```
1  a(x) :- b(x), c(x).
```

```
2    a(x) :- d(x), e(x).
```

Note that disjunctions are only syntactic sugar, and are handled entirely by the parser.

2.4.2 SOUFFLÉ Evaluation

To evaluate an input Datalog program, SOUFFLÉ uses the compilation pipeline shown in Figure 2.1 (Scholz et al., 2016). SOUFFLÉ uses semi-naïve evaluation as the backend evaluation strategy.

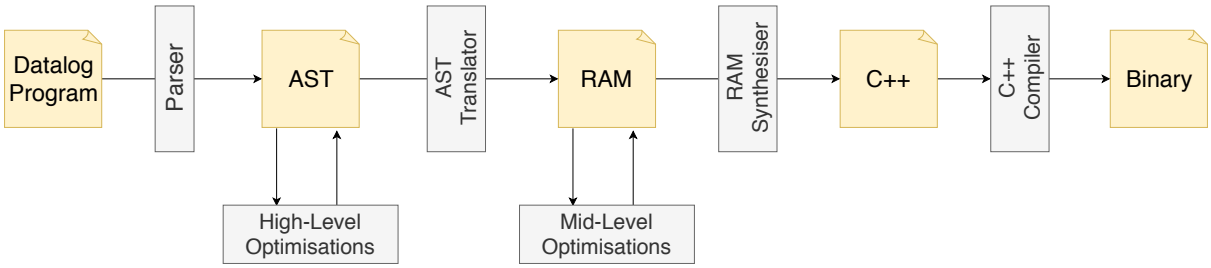


FIGURE 2.1. SOUFFLÉ compilation pipeline

As is the norm with compilers, the Datalog program is fed into the parser, which translates it into an Abstract Syntax Tree (AST). The AST format is a very high-level representation of the input program, and can be thought of as identical to the Datalog language, but easier to manipulate. Once the AST level is reached, SOUFFLÉ performs some basic high-level program-rewriting optimisations, discussed in detail in Chapter 4. These optimisations are an AST-to-AST function, and the result remains a valid Datalog program. The AST is then fed into a translator to convert it into Relational Algebra Machine (RAM) code. RAM code is a very high-level imperative language, mostly comprised of nested loops. For example, consider the following Datalog rule in SOUFFLÉ:

```
1    a(x,y) :- b(x,z), c(z,y), d(z).
```

The rule will be converted into the following nested for loops, called a *loop chunk*, by the AST-to-RAM translator, as part of the bottom-up evaluation process:

```
1    FOR EACH TUPLE (x,z) IN RELATION b:
2       FOR EACH TUPLE (z,y) IN RELATION c:
3          IF (z) IN RELATION d:
4             ADD (x,y) TO RELATION a
```

The correspondence is straightforward. The Datalog rule tells us that, if `(x,z)` $\in$ `b`, `(z,y)` $\in$ `c`, and `(z)` $\in$ `d` for any values of `x`, `y`, and `z`, then `(x,y)` $\in$ `a` as well. The RAM code emulates this check imperatively. We begin by scanning every tuple in `b` to ground variables `x` and `z` to instantiated values. We then ground the value of `y` by checking for all tuples in `c` with the instantiated value of `z` as the first argument. Finally, since `z` is already grounded to a value, we can just check if it exists in `d`, and we are done.

Notice that such a computation of `a` depends on the relations `b`, `c`, and `d` already being computed, so that tuple existence can be checked. SOUFFLÉ generates a precedence graph for each input program to determine the order of computation when translated to RAM code. The precedence graph encodes the dependencies between relations. In the event of mutual recursion, SOUFFLÉ condenses the strongly-connected subgraphs in the precedence graph into single nodes to form the Strongly-Connected Component (SCC) graph. As all strongly-connected components are condensed, the SCC graph contains no cycles, and so provides a strict topological ordering of the nodes. All relations in the same node in the SCC graph are computed simultaneously until a fixpoint is reached, before moving onto the next node in the topological order. For example, consider the following Datalog code written in SOUFFLÉ:

```
1   .decl a(x:number)
2   a(0).
3   a(x) :- c(x), c(x+1).
4
5   .decl b(x:number,y:number)
6   b(x,y) :- a(x), a(y).
7
8   .decl c(x:number)
9   c(x) :- b(x,x).
10
11  .decl d(x:number)
12  d(x) :- c(x), a(x).
13
14  .decl e(x:number, y:number)
15  e(x,y) :- c(x), d(y).
16
17  .output e
```

The precedence graph is shown in Figure 2.2, while the corresponding SCC graph is shown in Figure 2.3.

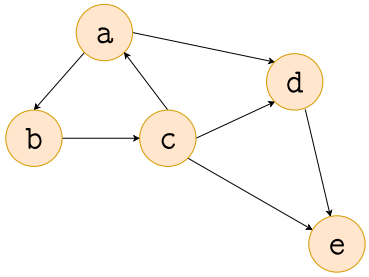


FIGURE 2.2. Precedence graph corresponding to the example program

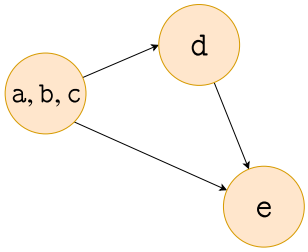


FIGURE 2.3. SCC graph corresponding to precedence graph in Figure 2.2

As this thesis is primarily on high-level program-rewriting transformations, our focus will be on the AST level of the compilation pipeline. The effectiveness of such transformations typically relies on the imperative code produced, so we will sometimes venture into the sphere of RAM code to motivate AST optimisations.

CHAPTER 3

Transformations

In this chapter, we devise a series of program-rewriting transformations aimed at improving the performance of Datalog programs. For each transformation, we motivate its existence and present an algorithm for applying it on a generic implementation of the Datalog language. The necessary pre-conditions on the input program for the transformation algorithm to be valid are detailed. We also briefly discuss our implementation in the aforementioned Datalog engine, SOUFFLÉ.

3.1 `InlineRelations`: Inlining

The idea of inlining functions is exceptionally common in modern programming languages, relied upon heavily by compilers both declarative and imperative (Chang and Hwu, 1989; Ayers et al., 1997). The idea is to expand function applications directly when they appear in the source code in an attempt at reducing function overhead. Expanding a function call in such languages typically results in less time and memory being used to copy arguments and manipulate the registers and stack. Moreover, the inlining process may present new avenues for optimisation in the context of the calling code, due to additional constraints imposed by the surrounding source code. Such optimisations include improved register allocation, code compaction, common subexpression elimination, and constant propagation (Chang and Hwu, 1989).

The concept of inlining has a natural counterpart in the logic-programming universe. The overhead generated by fully precomputing all tuples in a relation can be mitigated by instead replacing the 'calls' to the relation with the rules that define it. Whenever a lookup of the relation is performed, the evaluation engine now checks the body of each rule defined for the relation on the fly instead. The effect is similar to one iteration of a top-down evaluation strategy, where queries are replaced with equivalent subgoals. The benefit is that the appearance of the relation is likely constrained, and so not all tuples in the relation are actually relevant. Conversely, evaluating tuple-containment on the fly can lead to repeated

computations if the same lookup is performed several times. Similar to other programming languages, however, inlining here can significantly reduce unnecessary overhead, while also possibly introducing new opportunities for later optimisations, such as the elimination of repeated body atoms.

Despite the potentials of such an optimisation, inlining has not yet been researched from a logic-programming perspective. Here, we introduce a high-level program-rewriting transformation for inlining, and the conditions necessary for it to occur. As with several popular languages, such as `C`, a new `inline` keyword was added to SOUFFLÉ, which can be used by programmers as a hint to the compiler that certain relations should be inlined using the transformative procedure. The inlining feature is hence a semi-automated transformation driven by manual annotations. Data from previous runs, analyses of program structure, and domain-dependent programmer intuition can all provide insights into when on-the-fly evaluation is more appropriate than a full precomputation of the relation.

3.1.1 Motivation

Consider the following Datalog program P:

```
1  .decl natural(x:number)
2  natural(0).
3  natural(x+1) ← natural(x), x < N.
4
5  .decl natural_pair(x:number, y:number)
6  natural_pair(x,y) ← natural(x), natural(y).
7
8  .decl query(x:number, y:number)
9  query(x,y) ← natural_pair(x,y), y = x * x.
10
11 .output query
```

The program P outputs all pairs of natural numbers (x, x^2) for $x \leq N$, where N is some predefined variable. The recursive `natural` relation is a common design pattern for working with number ranges in Datalog. The sole output relation, `query`, is dependent on the `natural_pair` relation, which in turn is dependent on the `natural` relation. As no relations in P are mutually recursive, the SCC graph will look as shown in Figure 3.1.

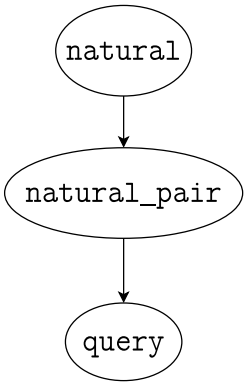


FIGURE 3.1. SCC graph corresponding to the example program P

A bottom-up Datalog evaluation engine like SOUFFLÉ will hence begin by precomputing all tuples in the `natural` relation, followed by all tuples in the `natural_pair`, before finally using the result to evaluate the output. The linearity of the ordering of relation evaluation means an analysis is simple. Each of the N elements of the `natural` relation is computed in sequence, resulting in $O(N)$ firings of its recursive rule. The `natural_pair` relation consequently generates $O(N^2)$ tuples by computing all pairs (x, y), with $x, y \in$ `natural`. Finally, we add a constraint in the query relation to return only those pairs (x, y) where the second element y is the square of the first. Notice that x^2 is unique for each $x \in$ `natural`, and so the constraint results in only $O(N)$ output tuples being produced, despite having to iterate through all $O(N^2)$ elements of `natural_pair`. There is no way for the evaluation strategy to anticipate the constraint when computing the `natural_pair` relation, as each relation is computed independently in sequence.

Rather than precomputing all tuples in the `natural_pair` relation, and then filtering out the valid pairs by applying the constraint, it is beneficial to only generate the tuples as needed, immediately taking into consideration the constraint. Using this 'on-the-fly' evaluation technique, we bypass the construction of `natural_pair`. Instead, we iterate through all elements of `natural` just once, only forming the pair (x, x^2) if $x^2 \in$ `natural_pair` as well. The net result is a generation of only $O(N)$ tuples in total throughout the entire program, rather than $O(N^2)$. Both evaluation time and memory usage have therefore benefited by skipping the construction of an intermediate relation.

On-the-fly evaluation can be emulated by removing the `natural_pair` relation, and instead embedding its rule in the query directly. The following program P' is produced in this case:

```
1    .decl natural(x:number)
```

```
2    natural(0).
3    natural(x+1) ← natural(x), x < N.
4
5    .decl query(x:number, y:number)
6    query(x,y) ← natural(x), natural(y), y = x * x.
7
8    .output query
```

Since the new conjunction in the query `natural(x), natural(y)` is true if and only if the original atom `natural_number(x,y)` was true, the output of P is identical to that of P'. The two programs P and P' are therefore semantically equivalent. However, the constraint appearing in the query of the program P has reduced the number of viable tuples in the `natural_pair` relation significantly. The embedding of the rule defining `natural_pair` directly into the query allows the evaluation mechanism in the backend to benefit as a result.

Thus, computational efforts for some non-query relations can be reduced by determining tuple containment on the fly, instead of forcing a full precomputation of all tuples. In particular, the computation of large relations where only a small portion of the result is used in later parts of the program, such as `natural_pair` in the above example, can be heavily dominated by the overhead of discovering and storing all tuples, rather than its use, in terms of both time and memory consumption.

Notice that, by replacing the appearances of a literal with the bodies of its associated rules to check the truth value, rather than precomputing the full relation, inlining in Datalog essentially emulates one iteration of a top-down evaluation strategy. With top-down evaluation, such as the SLD resolution technique used by Prolog, body literals in the query are iteratively replaced with equivalent subgoals until a true or false value can be guaranteed.

It is important to note that the inlining technique may not apply in certain cases, stated and explained in the conditions section below. Moreover, applying inlining may mean that some computations are repeated, since the tuple containment $\vec{y} \in R$ may need to be checked multiple times through the program. Nevertheless, many programs can largely benefit, such as the program P discussed above, as the number of tuple repetitions encountered may be insignificant when compared to the overhead of full relation computation. In general, inlining is most appropriate when used for large relations where it is not beneficial to compute and cache all tuples beforehand, such as when dependent relations impose sufficiently

strong constraints. Relations containing only a small number of rules, and with few appearances in other rules, are also more ideal for inlining, as a blowup in the number of rules in the program is avoided.

3.1.2 Conditions

Let P be the original program, and $I \subseteq \texttt{relations}(P)$ be the set of relations chosen to be inlined. For inlining to be possible such that semantic equivalence is maintained, we require some restrictions on P and I.

Output Relations

Condition 1: For all $R \in I$, R cannot be declared an output relation in P.

Justification: After embedding all appearances of R in P, the original relation R and its rules will all be deleted. Output relations must be returned at program termination, however, and so cannot be deleted. Moreover, the purpose of inlining is to avoid the complete computation of the given relation in a program. Inlining output relations is hence nonsensical, as all tuples must be discovered.

Input Relations

Condition 2: For all $R \in I$, R cannot be declared an input relation in P.

Justification: As R is an input relation, additional facts associated with the relation may be added at run-time, after all high-level transformations have been performed. To remove all appearances of R in the program, however, the inlining transformation must be aware of all rules associated with R. Input relations therefore cannot be inlined.

Recursion

Condition 3: Suppose G is the precedence graph of the Datalog program P. Let G_I be the subset of G containing only the relations $R \in I$. Then, G_I cannot contain a cycle.

Justification:

To get an intuition for the issue underlying this condition, consider first the following program P_1:

```
1   .decl a(x:number) inline
2
3   a(x) ← b(x), c(x).
4   a(x) ← a(x), d(x).
```

```
5
6     query(x) ← a(x), e(x).
7
8     .output query
```

Here, the set of relations to inline is $I = \{\texttt{a}\}$. Note that `query` is not in I, and so all rules where it appears in the head will remain in the program after the inlining process is complete. However, all appearances of the relation `a` must be inlined. Thus, the appearance `a(x)` in the single rule of the `query` relation must be inlined. The process will replace the appearance `a(x)` with a disjunction of the two rules for a. This gives us the following program:

```
1     .decl a(x:number) inline
2
3     a(x) ← b(x), c(x).
4     a(x) ← a(x), d(x).
5
6     query(x) ← ((b(x), c(x)) ; (a(x), d(x))) , e(x).
7
8     .output query
```

which is expanded out to the form:

```
1     .decl a(x:number) inline
2
3     a(x) ← b(x), c(x).
4     a(x) ← a(x), d(x).
5
6     query(x) ← b(x), c(x), e(x).
7     query(x) ← a(x), d(x), e(x).
8
9     .output query
```

Since the relation `a` is directly recursive, it reappears in the query rule after inlining. The new appearance must also be removed, requiring another step of inlining. The relation `a` will again reappear in a query

rule due to the recursive nature of `a`, compelling a further round of inlining, and so on. The program-rewriting process to produce the inlined program will hence never terminate. Figure 3.2 shows the first few steps of the inlining procedure on the rewriting of the query rule. The right-most branch will grow indefinitely.

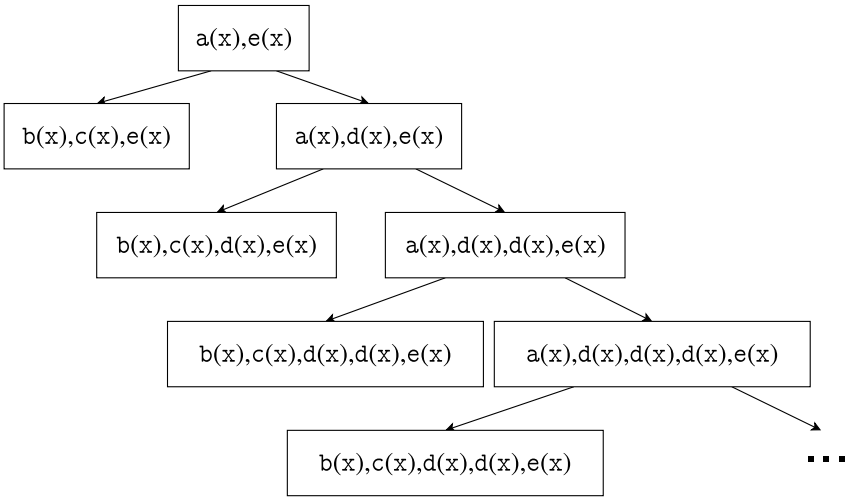


FIGURE 3.2. Inlining process when inlining the recursive relation `a` in `query` in program P_1

The problem also emerges in the more general case, where we have a cycle $C = (a_1, ..., a_n)$ in the dependency graph of P, where all the relations a_i in C are chosen to be inlined. That is, a_2 uses a_1 in at least one of its rules, a_3 uses a_2, and so on. As C is a cycle, a_1 will also use a_n in at least one of its rules. Now, suppose a_i appears in some non-inlined relation R. As a_i must be inlined, the appearance of a_i will be replaced with the bodies of its rules, as performed above. a_i uses a_{i-1} in at least one of its rules, however, and so a_{i-1} will now appear in a rule of R. a_{i-1} itself must be inlined, and so the process repeats, with a_{i-2} now appearing in one of the rules of R. Ultimately, after n iterations, a_i will reappear in a rule of R, due to the cyclic dependency chain C. The entire process was initiated to eliminate an occurence of a_i, however, and so the process will not terminate.

Program P_2 shows an example of the more general case.

```
1  .decl a(x:number) inline
2  .decl b(x:number) inline
3  .decl c(x:number) inline
4
5  a(x) ← b(x).
```

```
6    b(x) ← c(x).
7    c(x) ← a(x), d(x).
8
9    query(x) ← a(x), e(x).
10
11   .output query
```

Again, the result is an endless sequence of rewriting applications on the query rule, as shown in Figure 3.3.

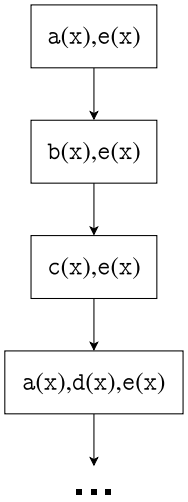


FIGURE 3.3. Inlining process when inlining the recursive set $I = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ on `query` in program P_2

Negation

Inlining negated appearances of relations is particularly interesting. Due to stratification requirements, negation in the Datalog world is rooted in the idea of fully precomputing a relation beforehand, and then performing a lookup on the resultant table to check the existence of a tuple.

For example, consider the evaluation of a Datalog rule for a relation R containing the body literal $\neg a(\vec{x})$. The relation a would first have to be fully computed prior to the evaluation of the relation R. Moreover, all variables in $\vec{x}$ must be grounded, that is, they must appear in some positive body literal in the body of the same rule. The groundedness property means that the variables in $\vec{x}$ can be instantiated with actual

values, after which a simple lookup for the resultant tuple can be performed on the already computed relation a.

As stated earlier, however, inlining replaces full precomputation with on-the-fly evaluation. The form of rules allowed for negated relations to be inlined is hence limited to those which can be checked as we go along, forcing the following conditions.

Condition 4: Suppose P contains a rule r for the relation R such that r introduces a variable in its body that does not appear in the head. Suppose further that R appears negated in some other rule in P. Then, R cannot be inlined.

Justification: Consider the fragment of Datalog code below, where $I = \{\texttt{a}\}$:

```
1   a(x) ← b(x,y), c(y).
2   d(x) ← e(x), ¬a(x).
```

Inlining `a` in the rule for `d` results in the rewrite sequence shown in Figure 3.4.

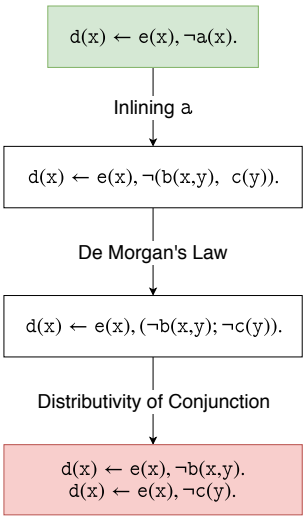


FIGURE 3.4. Inlining a negated relation that introduces a new ungrounded variable, leading to a semantic error

The variable y introduced in the new rules only appears negated, and so cannot be grounded. Consequently, a Datalog semantic restriction is broken, and so the produced program is invalid, despite the semantic validity of the original rule.

In the general case, suppose we wish to inline a relation a in a rule r such that:

- a appears negated in r, in the form $\neg a(\vec{x})$
- a introduces a new variable v in one of its rules r_a, such that v does not appear in the head of r_a

Inlining a in r means a new rule will be created such that the appearance of a is replaced with the body of r_a. Since the negated appearance $\neg a(\vec{x})$ of a in r must satisfy the groundedness requirements of Datalog, all variables in $\vec{x}$ must already be grounded in r. Since r_a introduces a new variable v, however, v does not appear elsewhere in the rule, and so must be ungrounded. The rule produced must therefore be invalid.

Condition 5 : Suppose R appears negated in some rule in P, in the form $\neg R(\vec{y})$, such that the k^{th} argument y_k is an unnamed variable. Moreover, suppose that there exists a rule $r_i : R(\vec{x}) \leftarrow B_i[\vec{x}]$ such that the k^{th} argument x_k of the head atom appears more than once in the body $B_i[\vec{x}]$. Then, R cannot be inlined.

Justification: Consider the fragment of Datalog code below, where $I = \{\texttt{a}\}$:

```
1  a(x) ← b(x,x).
2  c(x) ← d(x), ¬a(_).
```

Note that the unnamed variable in the appearance of `a` in `c` must immediately become named, as it is used multiple times within the rule defined for `a`. The program will hence be rewritten into the form:

```
1  c(x) ← d(x), ¬b(y,y).
```

The variable y does not appear in a positive body literal, and so it is ungrounded. The issue leading to the invalidation of the Datalog program arises for a similar reason as the previous negation condition, in the sense that new variables are being introduced which cannot be grounded.

3.1.3 Algorithm

3.1.3.1 α-reductions and Unification

As exemplary motivation for the chosen algorithmic design, consider the following fragment of a Datalog program P, and let the set of inlined relations be $I = \{a\}$:

```
.decl a(x,y) inline
a(x,y) ← d(x,x), e(y), f(v,v,y).
a(x,y) ← g(y,x).
...
b(x) ← c(x,z), b(y), a(y,z), d(v,v).
...
```

`a` appears in a non-inlined rule `b`, and so the appearance must be replaced with the disjuncted combination of its two rules. However, note that `a` appears in the form `a(y,z)`, while the rules have head `a(x,y)`. Thus, the variables in the two rules associated with a must first be renamed, as follows:

```
.decl a(x,y) inline
a(y,z) ← d(y,y), e(z), f(v,v,z).
a(y,z) ← g(z,y).
...
b(x) ← c(x,z), b(y), a(y,z), d(v,v).
...
```

Suppose we immediately embed the rules into `b` without further adjustment. The final program is then:

```
b(x) ← c(x,z), b(y), (d(y,y), e(z), f(v,v,z) ; g(z,y)), d(v,v).
```

Expanding the disjunction, we now have:

```
b(x) ← c(x,z), b(y), g(z,y), d(v,v).
b(x) ← c(x,z), b(y), d(y,y), e(z), f(v,v,z), d(v,v).
```

Notice that the variable `v` introduced in the second rule has now been implicitly equated with the `v` already existing in the rule prior to the inlining round. The semantics of the rule have hence changed, as these two variables were distinct, since they appeared in different rules. This issue with boundedness motivates the need for complete variable renaming prior to inlining, such that all variables in the rules of inlined relations are given unique names. The process of renaming is called an α-reduction.

Recall the original program P:

```
1  .decl a(x,y) inline
2  a(x,y) ← d(x,x), e(y), f(v,v,y).
3  a(x,y) ← g(y,x).
4  ...
5  b(x) ← c(x,z), b(y), a(y,z), d(v,v).
6  ...
```

Performing an α-reduction on the two rules for `a` results in the equivalent program:

```
1  .decl a(x,y) inline
2  a(x1,y1) ← d(x1,x1), e(y1), f(v1,v1,y1).
3  a(x2,y2) ← g(y2,x2).
4  ...
5  b(x) ← c(x,z), b(y), a(y,z), d(v,v).
6  ...
```

However, the appearance of `a` is still in the form `a(y,z)`, and so argument matching must occur. The process of matching arguments in the head of a rule with the appearance of a literal is called *unification*. Normally, unification in a logic-programming engine is performed through an iterative variable-matching algorithm (Martelli and Montanari, 1982). However, as we are already in a logic-programming environment, and variables are now uniquely named in the rules we wish to inline, we can avoid the intricacies of such an implementation through the use of constraints. In particular, we can replace the appearance of the relation with the body of the rule directly, and then add in equality constraints to unify the head and the appearance. The obligation of unification has hence been pushed from the transformation to the underlying Datalog system.

For instance, in the above example, we can form the appropriate unified version of the second rule for `a` by adding the equality constraint $x_2 = y, y_2 = z$. The program produced, after disjunctions have been expanded, is then:

```
1  b(x) ← c(x,z), b(y), d(x1,x1), e(y1), f(v1,v1,y1), x1 = y, y1 = z,
       d(v,v).
2  b(x) ← c(x,z), b(y), g(y2,x2), x2 = y, y2 = z, d(v,v).
```

which satisfies our expectations.

3.1.3.2 The Inlining Algorithm

The inlining algorithm designed repeatedly applies the literal-replacement procedure shown in the previous example until a fixpoint is reached, taking into account necessary unifications and α-reductions. Algorithm 2 performs one iteration of the literal-replacement procedure on a clause given an embedded literal to inline. Algorithm 3 then uses the clause inlining algorithm to perform the complete inlining transformation.

Algorithm 2 Inlining a given literal in a clause

Given: a clause c and a literal ℓ in c to inline
Output: the clause c' with ℓ inlined

1: **procedure** INLINECLAUSELITERAL(c, ℓ)
2: $result \leftarrow$ new empty rule
3: SETHEAD($result$) $\leftarrow$ CLONEHEAD(c)
4: SETBODY($result$) $\leftarrow$ CLONEBODY(c) without ℓ
5: $addedDisjunction \leftarrow \emptyset$
6:
7: ▷ if negated, get the associated positive atom
8: $R(\vec{y}) \leftarrow$ GETATOM(ℓ)
9:
10: ▷ add each rule to the disjunction, adding unification constraints
11: **for each** rule r of the form $R(\vec{x_i}) \leftarrow B_i[\vec{x_i}]$ **do**
12: $addedDisjunction \leftarrow addedDisjunction \vee (B_i[\vec{x_i}], \vec{x_i} = \vec{y})$
13:
14: ▷ add the disjunction to the clause
15: **if** $R(\vec{y})$ was negated **then**
16: $result$.ADDTOBODY($\neg addedDisjunction$)
17: **else**
18: $result$.ADDTOBODY($addedDisjunction$)
19: **return** $result$

Algorithm 3 Inlining algorithm

Given: a Datalog program P and a set of relations I to inline s.t. P and I satisfy the above conditions
Output: transforms P into a semantically equivalent program P' s.t. all relations in I are completely inlined

procedure INLINE(P, I)
  ▷ perform an α-reduction on all inlined relations
  for each relation $R \in I$ **do**
    for each rule $r \in R$ **do**
      Rename all variables in r to a program-wide unique variable name

  ▷ inline literals until a fixpoint is reached
  $inliningPerformed \leftarrow true$
  while $inliningPerformed$ **do**
    $clausesToRemove \leftarrow \emptyset$
    for each clause $c \in P$ s.t. GETRELATION(c) $\notin I$ **do**
      if GETBODY(c) contains a literal ℓ s.t. GETRELATION(ℓ) $\in I$ **then**
        $inliningPerformed \leftarrow true$
        add c to $clausesToRemove$
        $c' \leftarrow$ INLINECLAUSELITERAL(c, ℓ)
        add c' to P
    for each clause $c \in clausesToRemove$ **do**
      remove c from P

  ▷ remove no longer necessary relations
  for each relation $R \in I$ **do**
    remove R from P

3.1.4 Implementation

The inlining algorithm was implemented as a new AST transformation in the SOUFFLÉ Datalog evaluation engine. The transformation is called the `InlineRelationsTransformer`. The programmer can denote the set of inlined relations I using the newly added keyword, `inline`, during relation declaration. For example, the following is now a valid program:

```
1  .decl natural(x:number)
2  natural(0).
3  natural(x+1) ← natural(x), x < 1000.
4
5  .decl good_pair(x:number, y:number) inline
6  good_pair(x,y) ← natural(x), natural(y).
7
8  .decl bad_pair(x:number, y:number) inline
```

```
9    bad_pair(x,x) ← natural(x), natural(x).
10
11   .decl query(x:number, y:number)
12   query(x,y) ← good_pair(x,y), ¬bad_pair(x,y), x < 50.
13
14   .output query
```

Here, I is chosen by the programmer to be the set {`good_pair, bad_pair`}. The actual implementation expands the disjunctions produced in rules using the usual distributive property of the operation so that the output program only contains conjunctions in the rule bodies.

Note that programs using the `inline` declarative are always semantically equivalent to the same program with the declarative removed, and so the output is identical. The new feature hence constitutes no semantic changes, but provides a hint to the evaluation mechanism in the backend that rewriting some rules in a certain equivalent manner may provide benefits. Unlike some programming languages, such as `C`, the SOUFFLÉ engine currently always inlines a relation marked with the `inline` keyword, provided that the aforementioned conditions are met, without an analysis of potential efficiency gains. The inlining functionality added to SOUFFLÉ is hence only semi-automatic, in that, although the transformation process is fully automated, the optimisation is manually activated through user annotations.

3.2 `ReduceExistentials`: Reducing Existential Relations

When defining relations in Datalog programs, the evaluation engine discovers all possible tuples over the universe that can fit the established criteria. In certain cases, the actual tuples generated are irrelevant to the remainder of the program, and the only concern is whether any valid tuple at all can be discovered for the relation. The desire is then for an existential check, rather than a full computation.

We define a *singleton* variable v to be any variable in a rule r such that v appears exactly once in r. The instantiated value of that variable is irrelevant, as it has no consequence on any other value in the rule. Now, consider an appearance $R(\vec{x})$ of a relation R in the body of a rule r such that all arguments in the vector $\vec{x}$ are singletons in the rule r. We say that the appearance is then an *existential appearance* of the relation R. If all appearances of R in the program are existential, bar recursive appearances in the rules of R itself, then we call R an *existential relation*. As the instantiated values of singleton variables have

no consequence, the actual tuples appearing in an existential relation are unnecessary; the only required check is that any tuple exists in the relation at all.

An immediate corollary is that the arguments of existential relations do not matter. The program needs to only keep track of whether any tuple can be generated for the relation. The relation can therefore be transformed into a proposition; that is, a relation with no arguments. Unlike relations with non-zero arity, propositions can essentially take on only two values, true or false, depending on the satisfiability of its rules. The simplicity of propositional relations allows the backend system to take advantage of special optimisation opportunities. For example, once a rule for the proposition is found to be satisfiable, rule processing can be stopped entirely for the relation. Unfortunately, SOUFFLÉ is limited in the area of propositional optimisations. Nevertheless, transforming relations with non-zero arity into propositions means less storage is required for the relation, as no values must be stored beyond a boolean result for emptiness.

Our focus is on program-rewriting transformations, independent of the evaluation mechanism used, rather than the propositional optimisations in the backend. We therefore shift our attention to a more high-level view of the Datalog code. In particular, consider any existential relation R with a directly recursive rule r_{rec}. Due to the restrictions of program stratification, the recursive appearance of R in r_{rec} must be positive. Therefore, r_{rec} will only produce a new tuple if the appearance of R can be satisfied. The appearance can only be satisfied if R already contains a tuple. However, since R is only used existentially, the only concern is whether any tuple at all can be generated for the relation. The recursive rules for R are therefore redundant, and their computation unnecessary. Recursive rules for existential relations can therefore be removed from the program without effect. Recursive rules typically require continuous iterative refirings, and so the gains may be dramatic. As an extreme example, some relations may potentially have infinitely recursing rules, despite only appearing existentially. The removal of the recursive rules for such relations can hence transport the program from the realm of non-termination to termiation, without compromising program semantics from a declarative perspective.

Some work has been done in the past on eliminating existential arguments of relations (Ramakrishnan et al., 1988). However, the special case of existential relations lends new optimisation opportunities that have not yet been discussed. Moreover, the work relies on uniform equivalence for rule deletion, which is undecidable in Datalog with functors (Sagiv, 1988). The recursive rule deletion discussed for existential relations here does not rely on a separate program-minimising transformer, and applies to general Datalog programs with functors.

Significant performance benefits can therefore be gained with respect to both time and memory by discovering existential relations, and transforming them in such a way into non-recursive propositional relations. We call such a transformation of an existential relation *existential reduction*. As explained, the change into a propositional relation and the deletion of recursive clauses both provide separate advantages. Both changes clearly always improve program efficiency. The development of a transformation to deal with existential relations hence has great potential.

3.2.1 Motivation

Consider the following Datalog program P:

```
1  .decl natural(x:number)
2  natural(0).
3  natural(x+1) ← natural(x), x < N.
4
5  .decl query()
6  query() ← natural(_).
7
8  .output query
```

P outputs the value of the `query` proposition, which is true if and only if the single body atom `natural(_)` can be satisfied. The underscore in Datalog programs is used to represent singleton variables; essentially, variables that are not used anywhere else in the rule, and so do not need to be named. The atom can therefore be satisfied by any tuple in the `natural` relation, if any exist. `natural` is hence an existential relation in P; all uses of the relation (outside of its own recursive rules) only require a check that it is non-empty. Due to the semantics of bottom-up evaluation, the `natural` relation will nevertheless be fully computed beforehand, producing $O(N)$ tuples. The query then just checks that the produced set is non-empty, taking $O(1)$ time. The program hence takes $O(N)$ time to evaluate.

The computation of the `natural` relation is extremely wasteful. Computing and storing the full set of $O(N)$ tuples beforehand, to only just check whether the set is empty, is extremely inefficient. Since the `natural` relation is existential, we can begin optimising by noticing that its recursive rule is redundant; the recursive rule only holds if the body atom `natural(x)` holds for some `x`. Thus, the recursive rule

will discover atoms in the `natural` relation only if `natural` already contains at least one value. Since we are only concerned in whether the set is non-empty, the recursive rule can be deleted, as follows:

```
1  .decl natural(x:number)
2  natural(0).
3
4  .decl query()
5  query() ← natural(_).
6
7  .output query
```

The existentiality of the `natural` relation means that its arguments never matter, and so do not need to be stored. Consequently, the relation can also be transformed into its propositional counterpart by eliminating its arguments in all its appearances, to produce the following program P':

```
1  .decl natural()
2  natural().
3
4  .decl query()
5  query() ← natural().
6
7  .output query
```

Interestingly, due to the presence of a fact for `natural` in the original rule set, the remaining rule immediately sets the proposition to be true with no further computation. Notice further that P' is now entirely independent of N. As the recursive rule was the root cause of evaluation overhead for the relation, the advantages are significant in both time and memory. The net result is that program evaluation has now dropped from $O(N)$ to $O(1)$ time and storage.

Both transformations were justified by the fact that the original `natural` relation was solely used in an existential sense. Had the `natural` relation been used in any other rule in a non-existential fashion, then the actual values of the tuples in `natural` would be important. The transformations would then not be valid. We are relying on the use of unnamed variables to denote singletons, as is convention in the Datalog world. Programmers may at times neglect to use unnamed variables for singletons, however.

We will rely on another transformation, discussed in Section 3.3, to find and unname such variables, as such a transformation may provide other benefits.

3.2.2 Conditions

For the transformation to be applied to a given relation R, R must be an existential relation. That is, for all appearances $R(\vec{x})$ of R (outside of its own recursive rules), all arguments in $\vec{x}$ must be unnamed variables. A separate transformation, discussed in Section 3.3, can be used to transform all named singleton variables into unnamed variables beforehand.

Moreover, we also impose the restriction that the existential relation R cannot be an input or output relations. Input relations may have some facts not available at compile-time, which are given in a fixed format depending on the arguments in the relation declaration. Thus, input relations might not be able to be propositionalised. On the other hand, output relations must always be fully computed, regardless of how they are used in the remainder of the program.

3.2.3 Algorithm

The overall transformation algorithm is divided into two major steps: (1) discovering all existential relations in the program suitable for transformation, and (2) existentially reducing these relations.

3.2.3.1 Discovering Existentially Reducible Relations

The first stage of the algorithm is the discovery of all existential relations. Determining whether a relation is existential involves checking whether all appearances outside its own recursive rules contain only unnamed variables. Note, however, that an iteration of existential reduction may lead to the deletion of clauses. As these deleted clauses potentially contain the only non-existential appearances of some relations, it may be the case that a new set of relations are existential after an iteration of the transformation. To see the issue in practice, consider the following program P:

```
1  .decl a(x:number)
2  a(x) ← b(_), c(x), d(x).
3  a(x+1) ← b(x), a(x).
4
5  .decl b(x:number)
```

```
6    b(x) ← e(x), f(x).
7
8    .decl query()
9    query() ← a(_).
10
11   .output query
```

a is a typical existential relation. b is not, as it appears non-existentially in a rule of a. As a is existential, we can reduce the program as discussed earlier to the following form, transforming a into a propositional non-recursive relation:

```
1    .decl a()
2    a() ← b(_), c(x), d(x).
3
4    .decl b(x:number)
5    b(x) ← e(x), f(x).
6
7    .decl query()
8    query() ← a().
9
10   .output query
```

The recursive rule of a was deleted. The rule was the only non-existential appearance of b, and so b is now existential. The transformation can therefore be applied again, producing:

```
1    .decl a()
2    a() ← b(), c(x), d(x).
3
4    .decl b()
5    b() ← e(x), f(x).
6
7    .decl query()
8    query() ← a().
9
10   .output query
```

No remaining relations are existential, and so the transformation no longer applies.

Rather than working towards a fixpoint in the general case, we can use a graph structure to find all relations to transform in a single iteration. Consider a graph G where the nodes are the relations of the program, and an edge (a, b) exists in G if and only if b appears non-existentially in a directly recursive rule belonging to a. We say that a relation R is *eventually existentially reducible* in a program P if, after a finite number of applications of existential reduction on P, R becomes existentially reducible. We claim that, ignoring input and output relations, R is eventually existentially reducible if and only if:

(1) all ancestors of R in G are also eventually existentially reducible
(2) all appearances of R outside of recursive rules are existential

PROOF. We begin by proving the reverse implication: if R satisfies the two conditions, then it is eventually existentially reducible. Since all ancestors of R are eventually existentially reducible, after a certain number of iterations of the transformer, say k, all recursive relations belonging to the ancestors of R will be deleted. By the definition of G, the edge (a, b) exists if and only if b appears non-existentially in a recursive rule belonging to a. Therefore, no other relations use R non-existentially in their recursive rules. We can conclude that R appears existentially in all recursive rules. The second condition tells us that all other appearances of R (outside of the recursive rules defined for R itself) are also existential. By the definition of an existential relation, R must therefore be existential after the k^{th} iteration, and so can be reduced in the next. R is hence eventually existentially reducible.

For the forward implication, suppose R is eventually existentially reducible. The second condition must hold immediately, as non-recursive rules can never be removed by existential reduction. Suppose now that the relation S is an ancestor of R in G. By definition of G, R appears non-existentially in a directly recursive rule belonging to S. But R is eventually existentially reducible, and so after some number of iterations, say k, it must be existential. The non-existential appearance therefore cannot exist after k iterations. The only possibility is that the recursive rule of S must have been deleted, and so S must be eventually existentially reducible as well. Consquently, all ancestors of R in G must also be eventually existentially reducible. The two conditions hold, and so both sides of the implication are true. □

Notice that we can just reduce all eventually existentially reducible relations in the first iteration, rather than repeatedly applying the procedure. Thus, we can use Algorithm 4 to find the set of all relations to existentially reduce. The algorithm finds all relations that are fundamentally irreducible: input/output

relations, and relations breaking condition two. By our proof above, the only other relations that are not reducible are descendents of these.

Algorithm 4 Discovering existentially reducible relations

Given: a Datalog program P
Output: the set of relations E that are existentially reducible

```
1:  procedure FINDEXISTENTIALRELATIONS(P)
2:      G ← an empty graph
3:
4:      ▷ create the nodes in the graph
5:      for each relation R ∈ P do
6:          add R as a node to G
7:
8:      ▷ baseIrreducibles are fundamentally irreducible relations
9:      baseIrreducibles ← ∅
10:
11:     for each input/output relation R ∈ P do
12:         ▷ IO relations are not reducible
13:         add R to baseIrreducibles
14:     for each relation R ∈ P do
15:         for each rule r ∈ R do
16:             for each body atom a ∈ r do
17:                 if a contains any non-unnamed variables then
18:                     A ← GETRELATION(a)
19:                     if r is directly recursive then
20:                         ▷ A is conditionally existential
21:                         add the edge (R, A) to G
22:                     else
23:                         ▷ A breaks condition two
24:                         add A to baseIrreducibles
25:
26:     ▷ find relations that hence do not satisfy the first condition
27:     allIrreducibles ← ∅
28:     for each relation r ∈ baseIrreducibles do
29:         newIrreducibles ← relations reachable from r in G through a DFS
30:         allIrreducibles ← allIrreducibles ∪ newIrreducibles
31:
32:     ▷ all other relations are existentially reducible
33:     E ← ∅
34:     for each relation R ∈ P do
35:         if R ∉ allIrreducibles then
36:             add R to E
37:     return E
```

3.2.3.2 Transforming Existentially Reducible Relations

Once all existentially reducible relations have been found, the transformation step can be applied to each of the relations to produce the final program. The process is shown in Algorithm 5.

Algorithm 5 Transforming existentially reducible relations

Given: a Datalog program P
Output: transforms P into a semantically equivalent program P' s.t. all existentially reducible relations have been reduced

```
1: procedure REDUCEEXISTENTIALRELATIONS(P)
2:     E ← FINDEXISTENTIALRELATIONS(P)
3:     for each relation R ∈ E do
4:         create a new propositional relation exists_R, taking in no arguments
5:         copy over all non-recursive rules for R to exists_R
6:         replace all occurrences of R in the program with exists_R
7:         delete the relation R from P, along with its rules
8:         add exists_R to P
```

3.2.4 Implementation

The algorithm was implemented as a new AST transformation in the SOUFFLÉ Datalog evaluation engine. The transformation is implemented as the `ReduceExistentialsTransformer` class. As mentioned previously, we assume that all singleton variables have already been set as unnamed variables in the input program. Another transformer, called the `ReplaceSingletonVariables` transformer, will be discussed in Section 3.3. The transformer unnames singleton variables, hence making the `ReduceExistentials` transformer more effective.

As the discovery and transformation steps are fully automated, input Datalog programs remain the same, and programmers can use the language exactly as before.

Unfortunately, propositions are not effectively optimised in SOUFFLÉ. Further gains can therefore be exploited from this transformation by optimising the use of propositions in the backend evaluation system.

3.3 `ReplaceSingletonVariables`: Singleton Removal

A singleton variable in a given rule is any variable used exactly once in that rule. That is, the variable is not constrained, and so can take on any value without affecting the rest of the rule. As they are only

used once, singleton variables do not need to be referred to again, and so can remain unnamed. The convention for programs in common logic languages is to represent unnamed variables with a different symbol, almost always an underscore character ('_').

Knowing which variables are singletons opens new avenues for optimisation for both program-rewriting transformations and the backend evaluation strategies. From the perspective of evaluation, consider the appearance of an atom $\texttt{a}(\vec{x})$ in the body of some rule being evaluated. When the program is compiled into imperative code, the usual technique is to transform the atom into a scan over all items in the relation `a`. The process is hence linear in the size of `a`. On the other hand, if all arguments to `a` in such an appearance are known to be singletons, then the compiler can instead transform this into a constant-time existential check: does any tuple exist in `a`? When instead considering program-rewriting transformations, the presence of singletons may lead to satisfying the program conditions of other transformers. One example is the reduction of existential relations, described in Section 3.2, where the presence of singleton variables is the main trigger for the transformation.

It is therefore of interest to discover and mark all singleton variables, so that both the evaluation strategy and future transformations are aware of their use. The ideal means for marking singletons is by denoting them as unnamed variables throughout. Although programmers are able to use unnamed variables directly, some singletons may not be marked. One possible reason for neglecting the use of unnamed variables for singletons is the programmers themselves. Programmers may not notice that a variable is a singleton, may not be aware that unnamed variables create greater potential for program optimisation, or may simply prefer the clarity provided by naming singletons in certain contexts. On the other hand, certain variables may only become singletons after another program-rewriting transformation. Thus, a built-in automated transformation for discovering and unnaming singleton variables is a useful addition to a transformation pipeline.

3.3.1 Motivation

Consider the following very simple program P:

```
1  .decl natural(x:number)
2  natural(0).
3  natural(x+1) ← natural(x), x < N.
4
5  .decl a(x:number)
```

```
6    a(0) ← natural(z).
7
8    .decl query(x:number)
9    query(x) ← a(x).
10
11   .output query
```

The query outputs all values in the relation `a`. The sole rule of `a` is satisfied if there is a value `z` $\in$ `natural`. The variable `z` does not appear elsewhere in the rule, and so is a singleton. `z` can therefore be replaced with an unnamed variable, '_', passing on that it is a singleton to both the evaluation engine in the backend and future transformations. The result is the following program P':

```
1    .decl natural(x:number)
2    natural(0).
3    natural(x+1) ← natural(x), x < N.
4
5    .decl a(x:number)
6    a(0) ← natural(_).
7
8    .decl query(x:number)
9    query(x) ← a(x).
10
11   .output query
```

Though P and P' appear extremely similar, the implications of the change are significant.

To observe the impact on evaluation strategy, we will observe the produced imperative code for each program. First, consider the `query` relation common to both P and P'. All values in `a` are copied into `query`, and so a full scan of the relation `a` is necessary. The code produced for the `query` rule is hence of the form:

```
1    FOR EACH TUPLE (x) IN RELATION a:
2       ADD (x) TO RELATION query
```

Let us now shift our focus to the single rule for `a` in the original program P. As the evaluation strategy is unaware that `x` is a singleton in the rule, it must similarly scan through all values in `natural` to guarantee a check of all possible instantiations. The complete scan is essential in the general case, in case different instantiations of the variable `x` produce different results. Thus, the compiled code for the relation `a` will be of the form:

```
1   FOR EACH TUPLE (x) IN RELATION natural:
2      ADD (0) TO RELATION a
```

The produced code hence takes $O(N)$ time to evaluate, as `natural` contains N tuples. On the other hand, the use of an unnamed variable in the rule in P' means that the evaluation strategy is now aware that the instantiated value of `x` is irrelevant, as it is not used elsewhere in the rule. Thus, the compiled program will instead be of the form:

```
1   IF ∃ TUPLE (x) IN RELATION natural:
2      ADD (0) TO RELATION a
```

The complete scan of `natural` is replaced with a single existential check of the relation `natural`; on an imperative level, a loop through a relation has been replaced with a simple conditional. The result is constant evaluation time, rather than a linear scan.

Note that, due to the recursive nature of the `natural` relation, the total time is still linear, as `natural` must compute all N tuples. However, the mutation of the singleton into an unnamed variable has now also opened up a new avenue for optimisation, namely through existential reduction. The `natural` relation now appears only existentially in the program P', and so the transformation discussed in Section 3.2 can now be applied to produce the following program P''.

```
1   .decl natural()
2   natural().
3
4   .decl a(x:number)
5   a(0) ← natural().
6
7   .decl query(x:number)
8   query(x) ← a(x).
```

```
9
10    .output query
```

Notice that the program P'' is now entirely independent of N, and evaluation now takes constant time due to the interaction of the two transformers. The transformation is hence motivated from the perspective of both evaluation strategies and program-rewriting optimisations. The interaction of interdependent transformers is also motivated, and will be further discussed in Chapter 4, where a transformation pipeline is constructed.

As an aside, it is important to note that the programmer may not be able to unname certain variables immediately in the original program. Unnaming opportunities sometimes arise only as a result of other transformations. For example, consider the simple datalog rule:

```
1    c(x) ← b(y), b(y), d(x).
```

The variable `y` cannot be immediately unnamed by the programmer, as it appears twice in the same rule. However, the second appearance of the atom `b(y)` is redundant due to the idempotency of conjunction in logical calculus. One appearance of `b(y)` can hence possibly be removed through a high-level transformation to produce the following modified rule:

```
1    c(x) ← b(y), d(x).
```

The variable `y` is now a singleton, and can be safely unnamed.

```
1    c(x) ← b(_), d(x).
```

Future transformers can then possibly further optimise the resultant program.

3.3.2 Conditions

The transformation works on general Datalog programs. All singleton variables can be safely replaced with unnamed variables, as program semantics do not change. The process is completely automated and does not rely on programmer annotations.

3.3.3 Algorithm

As the singleton property of a variable is a property local to each rule, the transformation can be performed by iterating through each rule and discovering all singletons. The algorithm is shown in Algorithm 6.

Algorithm 6 Unnaming named singleton variables

Given: a Datalog program P
Output: transforms P into a semantically equivalent program P' s.t. all named singleton variables have been replaced with unnamed variables

```
1:  procedure REPLACESINGLETONVARIABLES(P)
2:      for each rule r ∈ P do
3:          variables ← ∅
4:          nonsingletons ← ∅
5:          for each variable v ∈ r do
6:              if v ∈ variables then
7:                  add v to nonsingletons
8:              else
9:                  add v to variables
10:         singletons ← variables \ nonsingletons
11:         for each variable v ∈ r do
12:             if v ∈ singletons then
13:                 replace v with an unnamed variable
```

3.3.4 Implementation

Singleton removal was implemented as a new AST transformation in the SOUFFLÉ Datalog evaluation engine. The transformation is called the `RemoveSingletonVariablesTransformer`, and works on general SOUFFLÉ programs.

3.4 `PartitionBodyLiterals`: Partitioning Rule Bodies

Propositions are predicates with arity zero. When evaluating the rules of propositional relations, no information must be stored on which variable instantiations satisfied the rule. As discussed in Section 3.2, once any instantiation of variables is found to satisfy the rule of a proposition, no other instantiations must be checked. The proposition then takes on a value of either true or false, based on the satisfiability of its rules, acting as a 'constant' truth value for the remainder of the program. The binary state of the proposition means the truth value of its appearances once it is computed can always be evaluated in constant time. Placing propositions at the start of rules can therefore allow a rule to be skipped entirely,

as discussed in Section 3.5. Due to such unique properties, the use of propositions can be especially optimised in the backend evaluation system.

Suppose a propositional relation is inlined. The first impact is that the backend evaluation system can no longer take advantage of such properties of propositions. Moreover, when embedded in the rules the proposition appears in, the lack of arguments means the introduced body literals will always be independent of the remainder of the rule. In particular, no new constraints are introduced. Inlining propositional relations therefore offers no benefits in reducing the number of possible satisfying instantiations of variables in the rule. The truth value of the proposition must also now be recomputed at each appearance, despite always producing the same single result. We therefore gain no benefits offered by the inlining transformation, while suffering the potential unnecessary repeated computations. In some cases, the truth value may even be recomputed multiple times within the same rule. For example, consider the following fragment of a Datalog program P:

```
1  .decl Prop() inline
2  Prop() ← a(z), b(z), c(z).
3
4  .decl R(x:number)
5  R(x) ← d(x), Prop(), e(x).
```

The propositional relation `Prop` is inlined into the rule for `R` to produce the output program P':

```
1  .decl R(x:number)
2  R(x) ← d(x), a(z), b(z), c(z), e(x).
```

Suppose that the relation `d` in P is large, say of size N. Moreover, suppose that the rule for the relation `Prop` evaluates to true, while the relation `e` is empty. In the original program P, the satisfiability of the single rule for `Prop` will only be checked once. `Prop()` will then act as a true value for the remainder of the program, without needed to be rechecked. On the other hand, embedding the rule into the clause for `R` in P' means that the satisfiability of the rule for `Prop` will necessarily be checked N times, as we fail and backtrack at `e(x)` for all values of `x` $\in$ `d`. In the situation where the relation `a` is large for this program, the increase in computation necessary can be significant.

The only scenario where it is better to inline a propositional relation is when the value of the proposition need not be computed at all in any rule of the program. For example, in the above program, if the relation

`d` is empty, then the value of the proposition is never used, and so is unnecessary. In a typical bottom-up computation of P, however, `Prop` will be evaluated beforehand, since `R` is dependent on `Prop`. In P', `Prop` does not exist as a separate relation, and so the relations `a`, `b`, and `c` will not be scanned if `d` is empty. Therefore, unless a check for the value of the propositional relation will never be triggered, inlining is a disadvantage to the efficiency of the program in a bottom-up evaluation engine.

By considering the converse of the situation, we hence propose the idea of 'reverse-inlining' or 'outlining' propositions. If some subset of literals in a rule body are entirely independent of the variables in the head of the rule, then the literals act as a propositional check independent of the remainder of the rule. Consequently, the literals can be pushed out (or *outlined*) into separate propositional relations, and replaced with a single propositional check. Needless recomputations of the satisfiability of a subset of the rule are hence reduced in many cases.

A natural conceptual extension is that rule bodies may be partitioned such that the variables used in partition p are entirely independent of the variables in partition q for all distinct partitions p and q. That is, changing the instantiated value of a variable in a literal in p has no consequence, indirect or otherwise, on the truth result of a literal in partition q. The partitions that the head atom depends on must remain in the clause, while all others may be pushed out into separate independent propositional relations, and replaced with an atomic check on the satisfiability of the proposition. Up to a reordering of body literals, inlining the produced propositional relations will return the original program, without the discussed benefits of using propositions in Datalog programs. Extraneous sections of clauses can therefore be extracted into separate relations, potentially reducing needless recomputations. The net benefit is reduced evaluation time, especially for computationally complex rule sets.

Similar work has been touched on briefly in the past (Ramakrishnan et al., 1988), but is not discussed in depth nor experimentally justified. The paper also does not discuss the finer partitioning of rules into multiple propositions based on variable interdependencies, nor the link with inlining. Moreover, the importance of placing the generated propositions first in the rule, as discussed in the motivation below, is not mentioned, but is critical for optimal performance. We discuss the importance of literal ordering further in Section 3.5.

3.4.1 Motivation

Consider the definition of the following relation `worried`:

```
.decl worried(x:string)
worried(x) ← person(x), ¬jailed(x), thief(y), ¬jailed(y).

.output worried
```

The relation `worried` will output the set of all unjailed people if there is a thief `y` that is not jailed. If the rule was directly evaluated, the produced imperative code would be of the following form:

```
FOR EACH TUPLE (x) IN RELATION person:
   IF (x) IS NOT IN jailed:
      FOR EACH TUPLE (y) IN RELATION thief:
         IF (y) IS NOT IN jailed:
            ADD (x) TO RELATION worried
```

The code takes $O(|\texttt{person}| \cdot |\texttt{thief}|)$ time. Notice that we perform a full scan of the relation `thief` for each possible instantiation of the variable `x` in `person`. The result of the scan never changes based on the instantiation of `x`, however, and so is unnecessary. Looking solely at the imperative code produced, we hence have an immediate natural optimisation, as follows:

```
unjailed_thief_exists := FALSE

FOR EACH TUPLE (y) IN RELATION thief:
   IF (y) IS NOT IN jailed:
      unjailed_thief_exists = TRUE

FOR EACH TUPLE (x) IN RELATION person:
   IF (x) IS NOT IN jailed:
      IF unjailed_thief_exists:
         ADD (x) TO RELATION worried
```

Essentially, we have pulled out the check that the inner loop performs into a separate external loop. The code now takes $O(|\texttt{thief}| + |\texttt{person}|)$ time, rather than $O(|\texttt{person}| \cdot |\texttt{thief}|)$ time, and so there is a clear benefit. Notice the negative effect in the corner-case that `person` is empty, as mentioned earlier.

Recall that our focus is only on high-level program-rewriting transformations, however. We wish to emulate the above change in imperative code through such a transformation. Notice how the inner loop has simply been replaced with a propositional check: does an unjailed thief exist? The value of the proposition itself is checked and stored separately prior to the scan of the `person` relation. The natural conclusion is hence to pull out the independent section of the rule into a new propositional relation, producing the following transformed program:

```
1  .decl worried(x:string)
2  worried(x) ← person(x), ¬jailed(x), unjailed_thief_exists().
3
4  .decl unjailed_thief_exists()
5  unjailed_thief_exists() ← thief(y), ¬jailed(y).
6
7  .output worried
```

The produced imperative code is then very similar to the one above, with the same improvement in time complexity:

```
1  FOR EACH TUPLE (y) IN RELATION thief:
2     IF (y) IS NOT IN jailed:
3        ADD () TO RELATION unjailed_thief_exists
4
5  FOR EACH TUPLE (x) IN RELATION person:
6     IF (x) IS NOT IN jailed:
7           IF ∃ TUPLE () IN RELATION unjailed_thief_exists:
8              ADD (x) TO RELATION thief
```

Consider now the following more complex rule:

```
1  a(x) ← b(x,y), c(y), d(z), e(z), f(w,v), g(v).
```

The produced imperative code without any transformations will be of the form:

```
1  FOR EACH TUPLE (x,y) IN RELATION b:
2     IF (y) IS IN RELATION c:
3        FOR EACH TUPLE (z) IN RELATION d:
```

```
4              IF (z) IS IN RELATION e:
5                 FOR EACH TUPLE (w,v) IS IN RELATION f:
6                    IF (v) IS IN RELATION g:
7                       ADD (x) TO RELATION a
```

The code takes $O(|\mathtt{b}| \cdot |\mathtt{d}| \cdot |\mathtt{f}|)$ time. Notice that the first check, `IF (y) IS IN RELATION c`, cannot be extracted out, as `x` must potentially be added to the relation `a`, and `y` is dependent on the value of `x`. However, all other inner loops and checks are independent of their values, and so can be extracted out as before:

```
1    extracted_section := FALSE
2
3    FOR EACH TUPLE (z) IN RELATION d:
4       IF (z) IS IN RELATION e:
5          FOR EACH TUPLE (w,v) IS IN RELATION f:
6             IF (v) IS IN RELATION g:
7                extracted_section = TRUE
8
9    FOR EACH TUPLE (x,y) IN RELATION b:
10      IF (y) IS IN RELATION c:
11         IF extracted_section:
12            ADD (x) TO RELATION a
```

The code now takes $O(|\mathtt{d}| \cdot |\mathtt{f}| + |\mathtt{b}|)$ time, and so we have improved our check. Again, the above can be emulated by the following program:

```
1    newrel0() ← d(z), e(z), f(w,v), g(v).
2    a(x) ← b(x,y), c(y), newrel0().
```

The produced code is then actually as follows, with the same improved time complexity:

```
1    FOR EACH TUPLE (z) IN RELATION d:
2       IF (z) IS IN RELATION e:
3          FOR EACH TUPLE (w,v) IS IN RELATION f:
4             IF (v) IS IN RELATION g:
```

```
5                ADD () TO RELATION newrel0
6
7     FOR EACH TUPLE (x,y) IN RELATION b:
8        IF (y) IS IN RELATION c:
9           IF ∃ TUPLE () IN RELATION newrel0:
10             ADD (x) TO RELATION a
```

However, notice that in our propositional check, represented by the first loop chunk, the inner loop and the associated check involving the variables `w` and `v` are independent of the outer loop and check, which use only the variable `z`. The inner loop and check can hence be pushed out even further to produce the following code:

```
1     extracted_rel := FALSE
2
3     FOR EACH TUPLE (w,v) IS IN RELATION f:
4        IF (v) IS IN RELATION g:
5           extracted_rel = TRUE
6
7     FOR EACH TUPLE (z) IN RELATION d:
8        IF (z) IS IN RELATION e:
9           IF extracted_rel:
10             ADD () TO RELATION newrel0
11
12    FOR EACH TUPLE (x,y) IN RELATION b:
13       IF (y) IS IN RELATION c:
14          IF ∃ TUPLE () IN RELATION newrel0:
15             ADD (x) TO RELATION a
```

The produced code is now further improved, taking $O(|\mathtt{f}|+|\mathtt{d}|+|\mathtt{b}|)$ time. The change can be emulated in a very similar manner as done previously, by pushing out the independent sequence of body literals in the new proposition formed into another separate propositional relation:

```
1     newrel1() ← d(z), e(z).
2     newrel0() ← f(w,v), g(v), newrel1().
3     a(x) ← b(x,y), c(y), newrel0().
```

Rather than recursively breaking up propositions formed, the literals in the original rule can be partitioned directly based on the variables used and their interdependencies. Each complete subset of body literals that is entirely independent of the literals in the remainder of the rule can be pushed out into a separate proposition. For the above example, this produces the following equivalent program:

```
1  newrel0() ← f(w,v), g(v).
2  newrel1() ← d(z), e(z).
3  a(x) ← b(x,y), c(y), newrel0(), newrel1().
```

The produced code is then as follows, with the same time complexity as the imperative program above:

```
1  FOR EACH TUPLE (w,v) IS IN RELATION f:
2     IF (v) IS IN RELATION g:
3        ADD () TO RELATION newrel0
4
5  FOR EACH TUPLE (z) IN RELATION d:
6     IF (z) IS IN RELATION e:
7        ADD () TO RELATION newrel1
8
9  FOR EACH TUPLE (x,y) IN RELATION b:
10    IF (y) IS IN RELATION c:
11       IF ∃ TUPLE () IN RELATION newrel0:
12          IF ∃ TUPLE() IN RELATION newrel1:
13             ADD (x) TO RELATION a
```

Partitioning body literals in such a way hence provides major benefits in terms of time complexity. Note, however, that the inner `IF` statements are still checked once for each element of the outer `FOR`-loops. Due to the lack of variables used in the `IF` statements in the final loop, the inner-conditionals are entirely independent of the value of the variables instantiated. Consequently, we can improve performance by reordering the statements, without compromising correctness. In particular, the propositional statements (corresponding to the conditionals) should be reordered to appear prior to other predicates (corresponding to the loops). Applying the principle to the Datalog program produces the following equivalent form:

```
1  newrel0() ← f(w,v), g(v).
```

```
2   newrel1() ← d(z), e(z).
3   a(x) ← newrel0(), newrel1(), b(x,y), c(y).
```

The transformation results in the following imperative code:

```
1   FOR EACH TUPLE (w,v) IS IN RELATION f:
2      IF (v) IS IN RELATION g:
3         ADD () TO RELATION newrel0
4
5   FOR EACH TUPLE (z) IN RELATION d:
6      IF (z) IS IN RELATION e:
7         ADD () TO RELATION newrel1
8
9   IF ∃ TUPLE () IN RELATION newrel0:
10     IF ∃ TUPLE () IN RELATION newrel1:
11        FOR EACH TUPLE (x,y) IN RELATION b:
12           IF (y) IS IN RELATION c:
13              ADD (x) TO RELATION a
```

Although the worst-case time complexity remains unchanged, the scan of relation `b` will now only occur if the propositional statements pass, which are $O(1)$ checks. The idea of atom-reordering to achieve similar gains is highlighted further in Section 3.5. Extracting disconnected body partitions, and replacing them with early propositional checks, can hence produce much more optimal code in the general case.

3.4.2 Conditions

The transformation works on general Datalog programs with no restrictions. Detection of body literals to push out into separate propositional relations can be done automatically. The transformation is hence entirely automated, and requires no user annotations. User annotations may be beneficial in the case where the programmer does not wish for extraction to occur, however.

3.4.3 Algorithm

The essence of the transformation is to partition rule bodies based on variable-usage, and extracting out partitions independent of the variables in the head. The transformation is hence local to each rule, and can be divided into an analysis stage and an extraction stage.

3.4.3.1 Stage 1: Variable Dependency Analysis

The analysis stage begins the process by partitioning the set of variables in a rule based on variable dependencies. The instantiation of certain variables may only affect or be dependent on a subset of other variables in the rule, while being completely independent of the others.

For instance, consider the following rule `Q`:

```
1   Q(x,y) ← A(x,t), B(t,s), C(s), D(x,y), E(u,v), B(u,r), F(r), G(k),
        H(k).
```

We say that two variables v and w are *directly connected*, written $v \sim_{weak} w$, if they appear in the same atom. Clearly, $\sim_{weak}$ is a reflexive and symmetric relationship. Directly connected variables are interdependent. For example, as `t` and `x` appear in the same atom `A(t,x)`, `t` $\sim_{weak}$ `x`. Changing the value of `x` may affect what values `t` may take on and vice-versa, based on the contents of `A`.

Note that $\sim_{weak}$ does not fully capture the dependencies between variables. For example, since `x` $\sim_{weak}$ `t`, and `t` $\sim_{weak}$ `s`, the instantiated value of `x` will affect what values `t` may take on, which in turn may affect what values `s` takes on, despite `x` $\not\sim_{weak}$ `s`. It is hence of interest to consider the transitive closure of the $\sim_{weak}$ relation.

Let $\sim$ be the transitive closure of $\sim_{weak}$. Clearly, $\sim$ is an equivalence relation. As described, if $x \sim y$, then the instantiation of either variable may affect the value of the other. Conversely, if $x \not\sim y$, then there is no path of predicates connecting the two variables, and so the instantiation of one will not affect the other. $\sim$ hence appropriately captures the dependency relationship between variables. Since $\sim$ is an equivalence relation, the equivalence classes of $\sim$ naturally partition the set of variables in a rule based on variable interdependencies.

To find these equivalence relations, we can begin by creating a graph modelling the relationship $\sim_{weak}$, which involves a simple traversal of the literals in the rule. The nodes in the graph represent the variables

in the rule, while an (undirected) edge exists between two variables if and only if they appear in the same literal. Since $\sim$ is the transitive closure of $\sim_{weak}$, the equivalence class of a variable v over $\sim$ is exactly those variables reachable from v in the graph. Partitioning over the equivalence classes of $\sim$ is hence equivalent to the discovery of all the connected components of a graph.

For example, in the rule defined above for `Q`, the graph in Figure 3.5 represents the $\sim_{weak}$ relationship between variables.

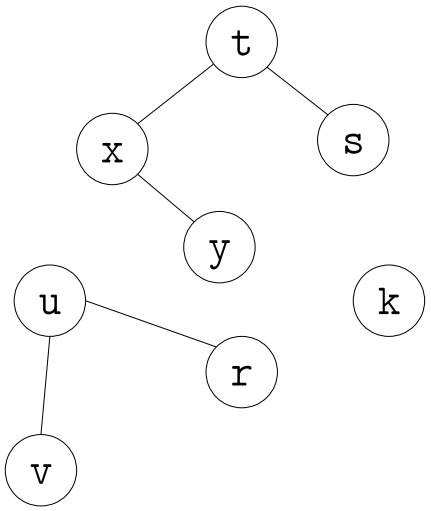


FIGURE 3.5. Graph representing the $\sim_{weak}$ relation for the rule defined for `Q`

The connected components are then exactly the equivalence classes over $\sim$, shown highlighted in different colours in Figure 3.6.

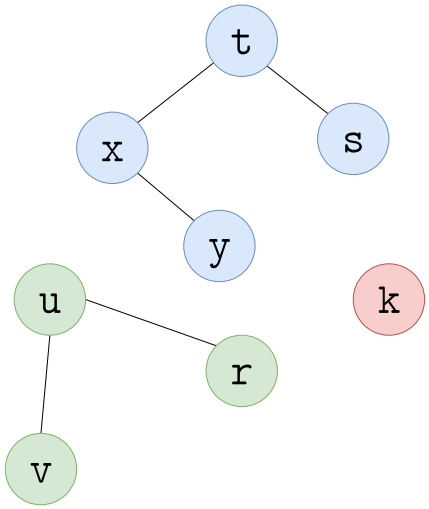


FIGURE 3.6. Graph showing each equivalence class of dependent variables over the $\sim$ equivalence relation in a different colour

In this case, the set of equivalence classes is $\{\{\mathtt{x}, \mathtt{y}, \mathtt{t}, \mathtt{s}\}, \{\mathtt{r}, \mathtt{u}, \mathtt{v}\}, \{\mathtt{k}\}\}$.

Algorithm 7 describes the process for a general Datalog rule R.

Algorithm 7 Partitioning rule variables based on interdependencies

Given: a Datalog rule R
Output: the set of equivalence classes over $\sim$

procedure PARTITIONRULEVARIABLES(R)
  $G \leftarrow$ an empty graph
  ▷ create the nodes of graph G
  for each variable $v \in R$ **do**
    add v as a node to G
  ▷ create the edges representing $\sim_{weak}$
  for each literal $\ell \in R$ **do**
    if ℓ contains any variables **then**
      $w \leftarrow$ an arbitrary variable in ℓ
      for each variable $v \neq w \in \ell$ **do**
        add the edge (v, w) to G
  ▷ find the connected components of G
  $seenNodes \leftarrow \emptyset$
  $connectedComponents \leftarrow \emptyset$
  for each variable $v \in R$ **do**
    if $v \in seenNodes$ **then**
      continue
    $currentComponent \leftarrow$ all variables reached from a DFS on G from v
    for each variable $w \in currentComponent$ **do**
      add w to $seenNodes$
    add $currentComponent$ to $connectedComponents$
  ▷ the connected components are the desired equivalence classes
  return $connectedComponents$

3.4.3.2 Stage 2: Proposition Extraction

The second stage is the extraction of body literals into propositions based on the variable partitions. The variable partitioning of each rule from the previous stage induces a natural partitioning of the literals in the rule. Each variable partition has an associated set of literals, namely the set of literals that use any variables within that partition.

To see this in practice, consider again the rule from above.

```
1   Q(x,y) ← A(x,t), B(t,s), C(s), D(x,y), E(u,v), B(u,r), F(r), G(k),
        H(k).
```

The set of equivalence classes was found to be $V = \{\{\texttt{x}, \texttt{y}, \texttt{t}, \texttt{s}\}, \{\texttt{r}, \texttt{u}, \texttt{v}\}, \{\texttt{k}\}\}$. For the equivalence class {x, y, t, s}, the set of associated literals (that is, literals in the rule using any variable in the

equivalence class) is {`Q(x,y)`, `A(x,t)`, `B(t,s)`, `C(s)`, `D(x,y)`}. Similarly, the set of associated literals for {`r`, `u`, `v`} and {`k`} are {`E(u,v)`, `B(u,r)`, `F(r)`} and {`G(k)`, `H(k)`} respectively.

Notice that (1) each set of literals is disjoint, and (2) the union produces all literals in the original rule. (1) is clearly true in the general case; if they share one variable, then they share all variables in the partition. In particular, the head atom of the clause appears in at most one such set. (2), on the other hand, is only true for all literals that use at least one variable. Propositions in the original rule can be placed in any of these sets. For simplicity, add the propositions to the set of literals containing the head atom if such a set exists; otherwise, create a new set for such literals, which will include the head atom. With this extension, the sets of literals form a partitioning of the literals in the clause.

As each literal partition uses a different set of variables by construction, the instantiations of variables in one partition will never affect, or be affected by, the values of variables in another partition. Thus, with the exception of the partition involving the head atom, all other partitions can be extracted out into separate propositional relations: the value of the variables does not matter for the remainder of the clause, only that the subclauses are satisfiable.

To illustrate, consider again the set of literal partitions {{`Q(x,y)`, `A(x,t)`, `B(t,s)`, `C(s)`, `D(x,y)`}, {`E(u,v)`, `B(u,r)`, `F(r)`}, {`G(k)`, `H(k)`}} for the above example. For the third partition, for instance, we need to check that `G(k)` $\wedge$ `H(k)` is true for some instantiation of `k`. By construction of the partitions, `k` will not be used elsewhere, in any other partition. Thus, the actual instantiated value is unnecessary outside the scope of these two literals. The literals can hence be safely extracted out into a separate propositional relation. The approach can similarly be performed for the second partition, {`E(u,v)`, `B(u,r)`, `F(r)`}. The partition containing the head atom, {`Q(x,y)`, `A(x,t)`, `B(t,s)`, `C(s)`, `D(x,y)`}, must of course remain in the clause, however, as the instantiated values must be stored in the head relation. The original rule can hence be transformed into the following fragment of Datalog.

```
1  .decl newrel0()
2  newrel0() ← G(k), H(k).
3
4  .decl newrel1()
5  newrel1() ← E(u,v), B(u,r), F(r).
6
7  // original rule
8  Q(x,y) ← newrel0(), newrel1(), A(x,t), B(t,s), C(s), D(x,y).
```

As propositions are $O(1)$ checks, it is beneficial to place them as the first atoms in the rule, as explained in Section 3.5.

Algorithm 8 describes extraction process for the transformer.

Algorithm 8 Partitioning rule bodies

Given: a Datalog program P
Output: transforms P into a semantically equivalent program P' s.t. independent body literals are extracted into propositions in all rules

```
1:  procedure PARTITIONRULEBODIES(P)
2:      for each rule r ∈ P do
3:          varPartitions ← PARTITIONRULEVARIABLES(r)
4:          if varPartitions has only one partition then
5:              ▷ no need to keep going, no splitting possible
6:              continue
7:          replacementClause ← empty rule
8:          SETHEAD(replacementClause) ← CLONEHEAD(r)
9:          ▷ deal with each disconnected set of literals
10:         for each partition part ∈ varPartitions do
11:             if part is the head partition then
12:                 continue
13:             extractedClause ← empty clause with extractedRelation as the head
14:             for each literal ℓ ∈ r do
15:                 if ℓ has no variables then
16:                     continue
17:                 v ← arbitrary variable in ℓ
18:                 if v ∈ part then
19:                     add ℓ to extractedClause
20:             add extractedClause to extractedRelation
21:             add extractedRelation to P
22:             replacementClause.ADDTOBODY(extractedRelation)
23:         ▷ deal with propositions and head-dependent literals
24:         if GETHEAD(r) has variables then
25:             v ← arbitrary variable in GETHEAD(r)
26:             headPartition ← partition in varPartitions containing v
27:         else
28:             headPartition ← emptyset
29:         for each literal ℓ ∈ GETBODY(r) do
30:             if ℓ has no variables then
31:                 replacementClause.ADDTOBODY(ℓ)
32:             else
33:                 v ← arbitrary variable in ℓ
34:                 if v ∈ headPartition then
35:                     replacementClause.ADDTOBODY(ℓ)
36:         remove r from P
37:         add replacementClause to P
```

3.4.4 Implementation

The transformation was implemented as a new AST transformation in the SOUFFLÉ Datalog evaluation engine. The transformation is called the `PartitionBodyLiteralsTransfomer`. The process is fully automated, and added to the main transformation pipeline.

3.5 `ReorderLiterals`: Literal Permutations

The declarative nature of the Datalog language means the ordering of statements does not impact program semantics. In particular, rearranging the literals within the body of clauses should have no semantic consequences; the body of a clause is a conjunction of literals, and conjunction is a commutative operation. As discussed previously, however, Datalog cannot be purely declarative, as it must be transformed into imperative form to be run.

Each body atom is typically translated into a full scan of the relation, through a for loop, with each consecutive atom being nested one level deeper. For example, a simple rule like the following:

```
a(x,y) ← b(x,z), c(z,w), d(w,y).
```

would be translated into:

```
FOR EACH TUPLE (x,z) IN RELATION b:
   FOR EACH TUPLE (z,w) IN RELATION c:
      FOR EACH TUPLE (w,y) IN RELATION d:
         ADD (x,y) TO RELATION a
```

Rearranging the atoms would hence cause the nesting levels to be rearranged. These nested levels may interact with previous levels through conditions, variable name bindings, and so on. The ordering can hence affect speed significantly. If the outer loops are relatively constraining and smaller in size, then nested levels may not be searched as deeply or as much. On the other hand, if the outer loops involve scanning large relations, or do not constrain each other, then the loop chunk may be unnecessarily inefficient.

For similar reasons, query scheduling is an important well-studied area in the database world. In the same way that conjunctions are core to Datalog programs, joins are core to relational database languages, performing, essentially, the same idea of connecting different relations. Abstracting away the typically non-recursive style of conventional database languages, the goal for database query schedulers is ultimately to devise an optimal ordering during a sequence of multi-table join operation (Iyer and Swami, 1994). The difficulty for such schedulers comes in from (1) deciding on a suitable cost-metric, and (2) the factorial number of possible orderings for each query. The latter may make it infeasible to gurantee the most optimal ordering regardless of the cost-metric. The cost-metric is usually based on relation size, as smaller relations result in a smaller cross-product. However, the distribution of data both within a single relation and between multiple relations can have a large effect on the size after a join is complete, and may be difficult to efficiently predict. For example, fixing a particular column in one relation may fix other columns. The use of heuristics has been shown to be promising in countering the issues faced (Swami, 1989; Swami and Gupta, 1988).

In the Datalog world, however, the constructed 'databases' are not usually persistent like conventional database systems. Instead, a different set of facts is given at the start of each run, and the database is created purely to compute a fixed query. Consequently, relation sizes and data distribution are not known statically, at the time the Datalog code is written. Thus, it may be more beneficial for query scheduling to be handled dynamically at runtime by the evaluation engine itself. Nevertheless, the variable-binding information inherently present in Datalog clauses, as well as user-defined domain-specific knowledge and information from previous runs on different datasets, may provide hints that allow us to work towards a more optimal ordering of body atoms.

We introduce a static query scheduling optimisation based on these principles. The query scheduler is a user-driven static transformation on the Datalog program that reorders body atoms in clauses based on a given heuristic. Several binding-based heuristics are provided as an option, each based on a different cost-metric. The heuristics greedily choose the next atom to scan in the conjunction based on the priority imposed by the cost-metric. The process hence emulates the choices typically done at runtime by conventional database query optimisers. Though dynamic query schedulers have an obvious advantage by having extra information at hand, we hope to gain similar improvements in some cases, despite cruder predictions.

3.5.1 Motivation

We begin with a simple teaser for the potential of a static query scheduler. Consider the following rule defined for a relation `R`:

```
1  R(x,y) ← A(x,z), B(z,y), P().
```

The corresponding imperative code is:

```
1  FOR EACH TUPLE (x,z) IN RELATION A:
2     FOR EACH TUPLE (z,y) IN RELATION B:
3        IF ∃ TUPLE () IN RELATION P:
4           ADD (x,y) TO RELATION R
```

Notice that the `IF` statement nested at the deepest level in the loop chunk is entirely independent of the other loops. The situation arises whenever a proposition is used. In particular, if the proposition does not hold, then the outer scans of `A` and `B` will be fully performed, despite having no effect. The program can easily be optimised by pushing out the proposition into the highest level, prior to any scan.

```
1  IF ∃ TUPLE () IN RELATION P:
2     FOR EACH TUPLE (x,z) IN RELATION A:
3        FOR EACH TUPLE (z,y) IN RELATION B:
4           ADD (x,y) TO RELATION R
```

The imperative transformation is the equivalent of moving the proposition to the beginning of the rule in the Datalog code:

```
1  R(x,y) ← P(), A(x,z), B(z,y).
```

If the proposition does not hold, then no loops will be executed at all. The number of steps in this case is reduced from $O(|\texttt{A}| \cdot |\texttt{B}|)$ to $O(1)$. If the proposition does hold, then there is no impact on efficiency. Therefore, pulling out propositions to the beginning of rules will always be either as good or better than the original rule.

The situation is not limited to propositions, however. Suppose, instead, the rule was defined as:

```
1   R(x,y) ← A(x,z), B(z,y), Q(1,2).
```

Then the program will become:

```
1   FOR EACH TUPLE (x,z) IN RELATION A:
2      FOR EACH TUPLE (z,y) IN RELATION B:
3         IF ∃ TUPLE (1,2) IN RELATION Q:
4            ADD (x,y) TO RELATION R
```

Again, the inner `IF` statement is independent of all other scans, and so can be moved to the beginning of the rule with the same gains as earlier. The ability to push out the atom to the front with a net positive impact was a simple consequence of the fact that the tuple being checked in `Q` was constant, and so can be checked in constant time. Computing the result as early as possible is hence advantageous.

We generalise the approach even further by considering an example with no constants.

```
1   R(x,y) ← A(x,z), B(z,w), C(w,y), E(x,w).
```

The corresponding imperative code is:

```
1   FOR EACH TUPLE (x,z) IN RELATION A:
2      FOR EACH TUPLE (z,w) IN RELATION B:
3         FOR EACH TUPLE (w,y) IN RELATION C:
4            FOR EACH TUPLE (x,w) IN RELATION E:
5               ADD (x,y) TO RELATION R
```

Observe that the outermost loop, scanning relation `A`, fixes the values of `x` and `z` for the inner loops it contains. Thus, the values of `y` and `z` are, for all intents and purposes, 'constant' for the inner loops. Consider the scan for tuples `(x,w)` in relation `E` on line 4. Notice that the value of `x` is fixed by the scan of `A` in the outermost loop, while the value of `w` is fixed by the scan of relation `B`. The for loop is therefore actually a check for a fixed tuple `(x,w)` in `E`. The imperative code is hence equivalent to:

```
1   FOR EACH TUPLE (x,z) IN RELATION A:
2      FOR EACH TUPLE (z,w) IN RELATION B:
3         FOR EACH TUPLE (w,y) IN RELATION C:
```

```
4              IF ∃ (x,w) IN RELATION E:
5                 ADD (x,y) TO RELATION R
```

Although w is fixed, the same cannot be said for the scan over relation C, as y is still free to take on any value. Note that relation C nevertheless does not introduce any new variables that E depends on. While it does use w, w is introduced and bound to a fixed value earlier on, by B. Hence, we can swap the conditional check on E and the scan over C to produce:

```
1   FOR EACH TUPLE (x,z) IN RELATION A:
2      FOR EACH TUPLE (z,w) IN RELATION B:
3         IF ∃ (x,w) IN RELATION E:
4            FOR EACH TUPLE (w,y) IN RELATION C:
5               ADD (x,y) TO RELATION R
```

Again, this is always a beneficial transformation; the conditional takes constant time, while the scan is linear in C in the worst-case. Bubbling the conditional upwards does not affect the binding of later variables either, as no new variables are introduced.

We therefore have a transformation guaranteed to always improve a program: if an atom in a rule has all its variables already bound to a value, then it should be pushed to the earliest point in the rule such that its variables remain bounded. In other words, execute the atom just after the last of its variables is bound.

The technique exploited variable binding information to extrapolate information about the size of certain scans; namely, if all arguments of an atom are bound, then the effective scan size is 1 - an existential check - regardless of actual relation size. Without knowledge of relation sizes and data distribution both within and between relations, however, no such guarantee of optimality can be obtained for general rules. The use of heuristics to estimate the optimality of a join order in such instances, as done in the database world, is hence beneficial to produce a more generally applicable transformer. Note that the applicability of the transformer is not limited to the situation where the user provides a non-optimal literal ordering. Some optimisations, such as inlining, can manipulate existing rules by adding, removing, or altering body literals. An automated scheduling transformer can hence cater for changes introduced by program transformations.

3.5.2 Conditions

The reordering transformation applies to all valid Datalog programs. The transformation is only semi-automatic, however, as a heuristic must be chosen. Note that functors in Datalog rules are typically replaced with new variables in practice, constrained to the value of the functor. Functors therefore do not need to be treated differently to other terms.

3.5.3 Algorithm

For the algorithm, we assume that we have a defined cost-metric h_{cost}, based on some choice of heuristic. h_{cost} takes in an arbitrary atom and the current environment of bound variables, and returns an integer representing the associated cost of choosing that atom next in our join. Lower costs imply a higher priority. Possible heuristics are described in the next subsection.

For simplicity, we take a greedy approach to determining the 'optimal' join order. The algorithm adds in the atoms to the clause one by one, always choosing the atom that minimises the cost at that point given the current environment of bound variables.

The algorithm is a local transformation on each rule. The only change is the ordering of atoms within each clause.

Algorithm 9 Applying a scheduling heuristic to a program

Given: a Datalog program P, and a cost-metric h_{cost} defined on all possible atoms
Output: transforms P into a semantically equivalent program P' with the body literals of each clause rearranged based on the heuristic

```
1:  procedure REORDERBODYATOMS(P,h_cost)
2:      for each relation R ∈ P do
3:          for each clause C ∈ R do
4:              atoms ← getBodyAtoms(C)
5:              remove all body atoms from C
6:              boundVariables ← ∅
7:              while atoms is not empty do
8:                  a ← atom in atoms that minimises h_cost(a, boundVariables)
9:                  add a to C
10:                 add variable arguments in a to boundVariables
11:                 remove a from atoms
```

3.5.4 Proposed Heuristics

We now propose a number of possible heuristics to use, and justify why the use of each heuristic may (or may not) be appropriate when working towards an optimal join order. Each heuristic is described by a cost-metric, defined on each possible atom appearance. As described in the algorithms section above, the heuristic prioritises atoms by minimising the cost-value of the next atom in a greedy fashion, taking into account the variables bound so far.

If two atoms have equal priority for a given metric, then the left-most atom in the original program will be chosen. The ordering provided by the user in the original program is hence still taken into consideration as the default option when applying the transformer.

For each heuristic, we will define the cost-metric on an arbitrary atom $a(x_1, \ldots, x_n)$, given a set of bound variables B.

3.5.4.1 `null`: Left-to-Right

The `null` transformer prioritises atoms entirely based on the order already given in the input program. The cost of an atom under the `null` heuristic is hence:

$$h_{\texttt{null}}(a(x_1, \ldots, x_n), B) := 0$$

Each atom is given equal weight. As the original order is prioritised in the case of equal cost, the heuristic will not affect the program.

3.5.4.2 `all-bound`: Completely Bound Atoms

The `all-bound` transformer prioritises atoms with all arguments bound. Recall from the motivation section above that any atom with all its arguments bound requires constant time to process, and that prioritising such atoms will always result in a better or equally efficient program. The cost-metric is hence defined by:

$$h_{\texttt{all-bound}}(a(x_1, \ldots, x_n), B) := \begin{cases} 0 & x_i \in B,\ \forall i \in \{1, \ldots, n\} \\ 1 & otherwise \end{cases}$$

Thus, all completely bound atoms are given a bottom (equal) cost of 0, hence having top priority, while other atoms are given a higher cost of 1.

3.5.4.3 `max-bound`: Maximal Number of Bound Arguments

The `max-bound` transformer prioritises atoms with the maximal number of bound aguments. The metric is motivated by the likely correlation between the number of bound arguments and a decrease in relation size: the more arguments that are fixed for an atom, the more the relation size is cut down, and so the smaller the resultant join. Thus, the assumption of this heuristic is that the atoms with the most bound arguments have been cut down the most, and so are most suitable as the next choice.

Note that the produced 'optimal' join based on this metric is not actually optimal in all cases. As a simple example, consider the heuristic applied to the following rule:

```
1   R(x,y) ← A(x,z), P(), B(z,y).
```

All atoms in the rule initially have the same number of bound arguments (zero), and so the original left-most atom, `A(x,z)`, is chosen by default as the first atom in the resultant ordering. `x` and `z` are now bound, so `P()` has zero bound arguments, while `B(z,y)` has one. Thus, `B(z,y)` is chosen as the next atom. The resultant rule after reordering is hence:

```
1   R(x,y) ← A(x,z), B(z,y), P().
```

But above we have shown that propositions - or, more generally, atoms with all arguments bound - should always be chosen first. Although this indicates an issue with the nature of the heuristic itself, we tweak the metric slightly to take into consideration the case where all arguments are bound.

The corresponding cost-metric is hence:

$$h_{\texttt{max-bound}}(a(x_1,\ldots,x_n),B) := \begin{cases} -\infty & x_i \in B,\ \forall i \in \{1,\ldots,n\} \\ -k & otherwise \end{cases}$$

where k is the number of bound arguments in $a(x_1,\ldots,x_n)$. The higher the number of bound arguments k, the lower the cost $-k$ is, the more priority it is given.

3.5.4.4 `max-ratio`: Maximal Ratio of Bound Arguments

The issue discussed with the previous `max-bound` metric hints that just the number of bound arguments in an atom may not be an accurate measure of relation size. Nevertheless, we know that atoms with all arguments bound should always have maximal priority. On the other hand, atoms with only unbound arguments are not constrained in size by the bound variables so far, and so, knowing nothing else of the relations, are unlikely to be a good candidate.

We hence alter the metric to take into account the *proportion* of bound arguments, relative to the number of arguments the atom actually has. Thus, atoms with *most* of their arguments bound are likely to be good next candidates; most arguments being bound implies most fields have been constrained for the atom.

We hence define the metric as:

$$h_{\texttt{max-ratio}}(a(x_1, \ldots, x_n), B) := -\frac{k}{n}$$

where k is the number of bound arguments in $a(x_1, \ldots, x_n)$. The higher the proportion $\frac{k}{n}$ of bound atoms, the lower the cost $-\frac{k}{n}$, the higher the priority of the atom. Note that maximising the ratio of bound arguments is equivalent to minimising the ratio of free arguments.

Again, though more natural than the previous metric, this heuristic is not guaranteed to give the optimal ordering in all situations. For example, consider the following program:

```
1   R(x,y) ← A(x,z), C(z,y,y,y), B(z,y).
```

The reordering strategy will transform the program into:

```
1   R(x,y) ← A(x,z), B(z,y), C(z,y,y,y).
```

Suppose, however, that the tuples $(\alpha, \beta, \gamma, \delta)$ in relation `C` are distributed such that, given the first column α, the triple (β, γ, δ) is uniquely identified. That is, `C` is a unique function of its first argument. Suppose further that there are hundreds of possible values for the second argument given the first argument for each tuple in `B`. Then, joining on `B` before `C` will involve scanning hundreds of possible values for `y` before performing an existential check on `C`. On the other hand, joining on `C` before `B` means only one

value for `y` will need to be checked, and then an existential check is performed on `B`. The first ordering is hence more optimal in this scenario. Note, of course, that this is entirely dependent on the data distribution in a particular run; in another situation, the produced ordering may be significantly more effective than the initial. The example highlights the importance of understanding the data distribution in the choice of a heuristic.

3.5.4.5 `least-free`: Minimal Number of Bound Arguments

Our final heuristic shifts the perspective from bound arguments to free arguments - that is, arguments not yet bound by a previous atom. The idea is to minimise the number of free arguments introduced by our next atom. To motivate this, observe that each introduced variable adds a new dimension to our search space. Consequently, the more free arguments introduced by our chosen atom, the more degrees of freedom the search space has. By minimising the number of free arguments, we hence minimise the dimensionality of our search.

The corresponding cost-metric is:

$$h_{\texttt{max-ratio}}(a(x_1, \ldots, x_n), B) := f$$

where f is the number of free arguments: arguments x_i in $a(x_1, \ldots, x_n)$ with $x_i \notin B$.

Unlike the `max-bound` heuristic above, the `least-free` heuristic encapsulates the `all-bound` principle that fully bound atoms should be prioritised over all others. First, note that $f \geq 0$ in all cases, as we cannot have a negative number of free arguments. Now, $f = 0$ if and only if there are no free variables, which is true if and only if all arguments are bound. Thus, the lowest cost is obtained if and only if the atom is maximally bound. Maximally bound atoms are hence prioritised, as needed.

Regardless, this heuristic may again not be ideal in certain cases, due to a lack of complete knowledge of the domain. In particular, the counterexample shown in the `max-ratio` subsection above still holds for the same reasons.

3.5.5 Implementation

The transformation was implemented as a new semi-automatic AST transformation in the SOUFFLÉ Datalog evauation engine. The transformer is called the `ReorderLiteralsTransformer`. Users can define the heuristic to use by passing in a 'SIPS' (*Sideways Information-Passing Strategy*) option to the compiler with the appropriate value, using the key-value pragma construct implemented in SOUFFLÉ. The term SIPS is an homage to the magic-set transformation (Bancilhon et al., 1985), discussed in the background section. If a SIPS is not provided, then SOUFFLÉ should default to the `all-bound` heuristic, as it is guaranteed to be as good as or better than the input ordering.

3.6 `MinimiseProgram`: Clausal Bijective Equivalence

A primary source of inefficiency in the execution of a bottom-up Datalog program is the computation of unnecessary tuples (Bancilhon et al., 1985). Tuples generated in an iteration of the bottom-up procedure are deemed unnecessary if they do not contribute to the query, or they have already been generated earlier in the program. While transformations like magic-set or existential reductions aim to eliminate non-contributing tuples, redundant tuple generation can remain a significant issue.

One major contributing factor to repeated tuple generation is the existence of redundancy in the formulated program. For example, a subset of clauses defined for a relation may actually be semantically subsumed by another clause. In such cases, the redundant clauses are unnecessary, and will not contribute to the output. Nevertheless, the redundant clauses will be evaluated, hence serving only as a hindrance to evaluation efficiency. It is therefore of interest to eliminate such redundancies in programs.

Redundancies in Datalog programs can be eliminated through the development of a program-minimising transformation. The scope of the transformation can be quite general, potentially aimed at removing superfluous:

- atoms, which are equivalent to other atoms within a clause
- clauses, which are equivalent to other clauses in the same relation
- relations, which are equivalent to other relations

There are therefore benefits in developing a program-minimising transformation. Our focus in this section will be on eliminating redundant clauses. Previous work has shown, however, that minimising programs under general equivalence is undecidable (Shmueli, 1993; Sagiv, 1988). Moreover, the fallback equivalence definition, uniform equivalence (Sagiv, 1988), also becomes undecidable in the presence of functors. Despite the theoretical roadblock, we push onwards by reducing our requirement of clausal equivalence to a much stricter form: *clausal bijective equivalence*.

We define clausal bijective equivalence as follows:

DEFINITION. *Let C_1 and C_2 be two distinct clauses of the form:*

$$C_1 : A(\vec{x_0}) \leftarrow R_1(\vec{x_1}), \ldots, R_n(\vec{x_n}).$$
$$C_2 : B(\vec{y_0}) \leftarrow S_1(\vec{y_1}), \ldots, S_m(\vec{y_m}).$$

We say that C_1 and C_2 are bijectively equivalent if and only if there exists:

- *a permutation $\pi : \{1, \ldots, n\} \rightarrow \{1, \ldots, m\}$, and*
- *a bijection $\alpha : Var(C_1) \rightarrow Var(C_2)$, where $Var(C_i)$ is the set of variables in C_i*

such that applying π and α to the body literals and variables of C_2 respectively syntactically produces C_1. That is,

$$B(\alpha(\vec{y_0})) \leftarrow S_{\pi^{-1}(1)}(\alpha(y_{\vec{\pi^{-1}(1)}})), \ldots, S_{\pi^{-1}(m)}(\alpha(y_{\vec{\pi^{-1}(m)}})) \equiv C_1, \textit{ syntactically}$$

Informally, two clauses C_1 and C_2 are said to be bijectively equivalent if we can reorder the body literals of C_2 and rename its variables such that it looks identical to C_1. As there are only a finite number of body-literal permutations and variable renamings possible, a check for bijective equivalence is trivially decidable. To achieve decidability, the generality of the equivalence check is sacrificed; some clauses not bijectively equivalent are still equivalent in the general sense. In essence, clausal bijective equivalence produces cruder equivalence classes; as we refine our notion of equivalence, these equivalence classes begin to merge.

Checking the bijective equivalence of clauses within a given relation hence provides a basis for program minimisation techniques aimed at reducing program redundancies. Given a means for checking

clausal equivalence, program-minimisation can be extended to check for redundant relations, or even intraclausal redundancies (that is, superfluous body literals within a clause). We defer such extensions to future work, detailed in Chapter 6.

3.6.1 Motivation

Consider the definition of the following relation `a` in a Datalog program:

```
1  .decl a(x:number, y:number)
2  a(x,y) ← b(x,z), c(z,y).
3  a(x,y) ← b(x,z), c(z,y).
4  a(x,y) ← d(x,y).
```

Note that the first rule is repeated, and is hence redundant. The impact is immediately illustrated when the produced imperative code is analysed.

```
1  FOR EACH TUPLE (x,z) IN RELATION b:
2     FOR EACH TUPLE (z,y) IN RELATION c:
3        ADD (x,y) TO RELATION a
4
5  FOR EACH TUPLE (x,z) IN RELATION b:
6     FOR EACH TUPLE (z,y) IN RELATION c:
7        ADD (x,y) TO RELATION a
8
9  FOR EACH TUPLE (x,y) IN RELATION d:
10    ADD (x,y) TO RELATION a
```

The total number of tuples scanned is $2|b||c| + |d|$. The relation `a` is a *set* of tuples, however, and so the second for loop is entirely useless. Eliminating it reduces the number of tuples scanned to $|b||c| + |d|$. Thus, deleting the second rule in the original definition of the relation `a` can have a significant effect on computational efficiency.

The repeated rule may not be syntactically identical, however. For example, consider the following modified definition of the relation `a`:

```
1  .decl a(x:number, y:number)
```

```
2   a(x,y) ← b(x,z), c(z,y).
3   a(x,y) ← c(z,y), b(x,z).
4   a(x,y) ← d(x,y).
```

Declaratively, reordering body atoms has no effect on the satisfiability of the body of the clause. Consequently, the first two rules are still equivalent. The redundancy of the second clause can also be seen through the produced imperative code:

```
1   FOR EACH TUPLE (x,z) IN RELATION b:
2      FOR EACH TUPLE (z,y) IN RELATION c:
3         ADD (x,y) TO RELATION a
4
5   FOR EACH TUPLE (z,y) IN RELATION c:
6      FOR EACH TUPLE (x,z) IN RELATION b:
7         ADD (x,y) TO RELATION a
8
9   FOR EACH TUPLE (x,y) IN RELATION d:
10     ADD (x,y) TO RELATION a
```

The sole difference between the first two loop chunks is the inverted order of the for loops. Swapping the order of the for loops in a loop chunk will clearly not affect the tuples added into `a`, however; statements are only executed in the deepest level of the loop chunk. We hence achieve the same gains eliminating the second rule as we did in the first case. As shown in Section 3.5, however, different orderings can have significant impacts on evaluation efficiency, despite declarative equivalence. Consequently, care must be taken in choosing the correct clause to eliminate, or another program-rewriting transformation must be run to mitigate the issue.

Repeated rules can be even more complex to identify. For example, consider a further adjustment to the definition of `a`:

```
1   .decl a(x:number, y:number)
2   a(x,y) ← b(x,z), c(z,y).
3   a(u,v) ← c(k,v), b(u,k).
4   a(x,y) ← d(x,y).
```

Again, permutation is inconsequential in the declarative world, and so the program is semantically equivalent to:

```
1  .decl a(x:number, y:number)
2  a(x,y) ← b(x,z), c(z,y).
3  a(u,v) ← b(u,k), c(k,v).
4  a(x,y) ← d(x,y).
```

The first two rules are still not syntactically identical. They can be made to be, however, through a renaming of variables in the second rule. In particular, renaming `u` and `v` to `x` and `y` respectively transforms the program into:

```
1  .decl a(x:number, y:number)
2  a(x,y) ← b(x,z), c(z,y).
3  a(x,y) ← b(x,z), c(z,y).
4  a(x,y) ← d(x,y).
```

We have not altered the variable bindings in any way, only the name used to identify them. As the names used are irrelevant, provided that they are distinct, program semantics have again not changed. Imperatively, the loop chunk is transformed from:

```
1  FOR EACH TUPLE (u,k) IN RELATION b:
2     FOR EACH TUPLE (k,v) IN RELATION c:
3        ADD (x,y) TO RELATION a
```

to this:

```
1  FOR EACH TUPLE (x,z) IN RELATION b:
2     FOR EACH TUPLE (z,y) IN RELATION c:
3        ADD (x,y) TO RELATION a
```

Variable renamings are similarly fine in the imperative world. The second rule can hence again be eliminated, producing the same efficiency gains.

Therefore, by finding clauses that are bijectively equivalent - that is, syntactically identical up to body-literal permutation and variable renaming - we can eliminate large loop chunks, leading to potentially significant efficiency gains, particular in evaluation time.

3.6.2 Conditions

The bijective equivalence check applies to pairs of clauses within the same relation. For simplicity, we limit the equivalence check to clauses with no negation or constraints. In the presence of constraints, particularly symmetric constraints like equality, the bijective equivalence check becomes more nuanced. Thus, the focus of the transformation is only on clauses with positive body atoms, whose arguments are either constants or variables.

3.6.3 Algorithm

The transformation will iterate through all relations. For each relation R, the rules defined for R will be partitioned into equivalence classes over $\sim$, where the equivalence relation $\sim$ is defined to be the bijective equivalence notion described earlier. That is, $C_1 \sim C_2$ if and only if there exists a permutation of the body atoms of C_2 such that C_2 is syntactically identical to C_1, up to variable renaming. Once the clauses of the relation have been partitioned in this way, only one rule needs to remain per class, as all clauses in the same class compute the same tuples. Consequently, all but one can be deleted from the program.

The core step of the process is checking clausal bijective equivalence: given two clauses C_1 and C_2, is C_1 equivalent to C_2 under bijective equivalence? To check this, we need to find:

- A permutation π of the body atoms of C_2, and
- A variable renaming function α on the variables in C_2

such that, after applying both π and α to C_2, $C_1 \equiv C_2$ syntactically.

Recall that we have restricted the clauses we consider to those containing only positive atoms with no constraints. Thus, the two clauses will be of the form:

$$C_1 : A(\vec{x_0}) \leftarrow R_1(\vec{x_1}), \ldots, R_n(\vec{x_n}).$$

$$C_2 : B(\vec{y_0}) \leftarrow S_1(\vec{y_1}), \ldots, S_m(\vec{y_m}).$$

As a running example, we consider the following two rules r_1 and r_2, defined for the relation `R`:

```
r1 : R(x) ← a(x), a(z), b(x,z).
r2 : R(k) ← a(w), b(k,w), a(k).
```

3.6.3.1 π: Enumerating Valid Permutations

We begin by traversing the search space of the body-atom permutation, π. Note first that it is necessary that the two clauses have the same length; that is, the same number of body literals. Otherwise, there is no way to form C_1 from C_2 through only permutations and variable renamings. Therefore, $n = m$ is a necessary condition.

For simplicity, number the literals in each rule from 0 to n, going left to right from the head atom.

$$C_1 : A(\vec{x_0}) \leftarrow R_1(\vec{x_1}), \ldots, R_n(\vec{x_n}).$$
$$C_2 : B(\vec{y_0}) \leftarrow S_1(\vec{y_1}), \ldots, S_n(\vec{y_n}).$$

Each possible permutation is a bijection $\pi : \{0, 1, \ldots, n\} \to \{0, 1, \ldots, n\}$ on the literals of C_2, where $\pi(p) = q$ means that the literal in position p moves to position q. We can also intuitively think of the assignment $\pi(p) = q$ as mapping the atom in position p in C_2 to the atom in position q in C_2. There are $(n + 1)!$ such permutations, however, so, although rule sizes are typically small in Datalog programs, we aim to substantially restrict the search space.

First, the head atoms A and B are semantically unique: the evaluation engine will place all generated tuples into those relations once the body is satisfied. As a result, no other literal can take the position of the head atom. Head atoms are hence fixed, so $\pi(0) := 0$ is necessary.

Now, relation names are static throughout the equivalence check. For the permutation to be valid, we require that $\pi(p) = q \Rightarrow \text{relation\_name}(S_p) = \text{relation\_name}(R_q)$. This is because the permuted atom in C_2 must map to the atom in the corresponding position in C_1 when checking for syntactic equivalence at the end. As $\pi(0) = 0$ necessarily, the head atoms must have the same relation name. We hence require $B = A$ for the two clauses. The constraint is satisfied when only checking for bijective equivalence of

clauses defined for the same relation. Moreover, we can now restrict our valid permutation space to those which map atoms in C_2 to same-relation atoms in C_1.

To find all such permutations, we define a two-dimensional $(n+1) \times (n+1)$ permutation matrix M as follows:

$$\forall_{0 \leq p \leq n} \forall_{0 \leq q \leq n}, M[p,q] := \begin{cases} 1 & p = q = 0 \\ 1 & \text{relation\_name}(S_p) = \text{relation\_name}(R_q),\ p \neq 0,\ q \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

Essentially, the row 0 and column 0 are entirely zeroes, except for the shared cell $M[0,0]$. This is because $\pi(0) = 0$, which means that 0 can map to nothing else (hence, the rest of row 0 is filled with zeroes), and that 0 can be mapped to by nothing else, as π is a bijection (hence, the rest of column 0 is filled with zeroes).

We will now construct the permutation matrix M for our running example above. We are checking for valid permutations of the literals in r_2 into the form of r_1. The first row and first column are immediate, as the head atom is fixed. Now, row p looks at where the p^{th} literal in r_2 can move to; that is, the positions j where the j^{th} literal in r_1 has the same relation name as the p^{th} literal in r_2. For row 1, the atom we are considering is `a(w)`. We hence need to find all positions in r_1 where the atom uses the same relation `a`. In this case, positions 1 and 2 in r_1 use the relation `a`, and so $M[1,1] = M[1,2] = 1$, while $M[1,3] = 0$. Similarly for row 3. Row 2, on the other hand, looks at relations in r_1 using the relation `b`. So, $M[2,3] = 1$, while $M[2,1] = M[2,2] = 0$.

Thus, the permutation matrix for the running example is as follows:

$$M_{r_1,r_2} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

Algorithm 10 describes the process of generating a permutation matrix for a given pair of clauses.

Algorithm 10 Generating a permutation matrix for a pair of clauses

Given: Two clauses C_1 and C_2 with the same number of body atoms
Output: The corresponding permutation matrix

```
1:  procedure CREATEPERMUTATIONMATRIX(C_1, C_2)
2:      N ← number of body literals in C_1 (and C_2)
3:      permutationMatrix ← n × n matrix filled with 0s
4:      permutationMatrix[0, 0] ← 1
5:      for each i ← 1 to N do
6:          atom_{C_2} ← i^th atom in C_2
7:          for each j ← 1 to N do
8:              atom_{C_1} ← j^th atom in C_1
9:              if GETRELATIONNAME(atom_{C_1}) == GETRELATIONNAME(atom_{C_2}) then
10:                 permutationMatrix[i, j] ← 1
11:     return permutationMatrix
```

The matrix hence encodes which transpositions are valid in the definition of the permutation. The issue is now extracting the valid permutations from the matrix.

A permutation π is possible through the matrix if and only if $\pi(p) = q \Rightarrow M[p, q] = 1$ for all rows $p \in \{0, 1, \ldots, n\}$. To derive all such permutations, we begin with an empty permutation, and construct each possible complete permutation by filling in the blanks from left to right. To demonstrate the concept, we will apply it to the permutation matrix M_{r_1,r_2} of the running example.

We begin with an empty permutation, represented as an empty array of the size of the permutation, $[\square, \square, \square, \square]$. The element in index i represents which position item i will be moved to. For example, the permutation $\pi = [1, 2, 0]$ applied to the list $\ell = [a, b, c]$ will result in $\pi(\ell) = [c, a, b]$. The permutation matrix gives us the set of possible values for each index. Looking at the the 0^{th} row of M_{r_1,r_2}, the only possible value for index 0 is 0. For index 1, the possibilities are 1 and 2, and so on. A valid permutation π is then any possible array $[p_0, p_1, p_2, p_3]$ such that p_i is a valid move for index i according to the permutation matrix, with the added constraint that it is a bijection from $\{0, 1, 2, 3\}$ into $\{0, 1, 2, 3\}$, by definition of a permutation. Since the domain and codomain are the same, and have finite size, surjectivity and injectivity imply each other. It is hence sufficient (and necessary) to force each p_i to be distinct, as this gives us injectivity.

Starting with the empty permutation, we can choose a possibility for the leftmost index, necessarily 0 in this case, giving us the permutation $[0, \square, \square, \square]$. We now need to choose a move for index 1. There are two possibilities, 1 and 2. Since all possibilities must be traversed, we need to choose this non-deterministically. The result is that we now branch out to two possible permutation prefixes: $[0, 1, \square, \square]$

and $[0, 2, \square, \square]$. The process then continues on each permutation independently. The technique used can be naturally modelled through a depth-first graph traversal. Due to the constraint that 0 always maps to 0, we can always begin at a root node 0. For each possible move for the next index based on the permutation matrix, we branch out to a separate child node, each representing a different move. This gives us the following tree T_{r_1,r_2} for the matrix M_{r_1,r_2}:

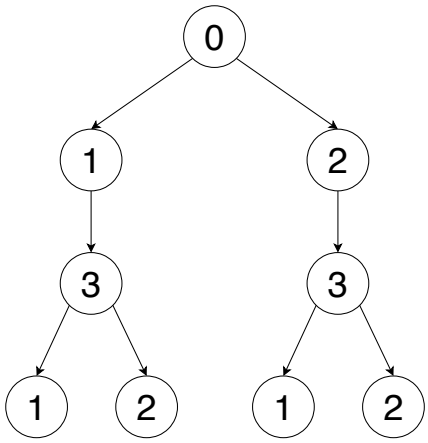


FIGURE 3.7. Tree representation T_{r_1,r_2} of permutation matrix M_{r_1,r_2}

Each path down the tree now represents a possible move combination. In this case, we get the arrays $[0, 1, 3, 1]$, $[0, 1, 3, 2]$, $[0, 2, 3, 1]$, and $[0, 2, 3, 2]$. Figure 3.8 illustrates the path that gives us the permutation $[0, 1, 3, 2]$.

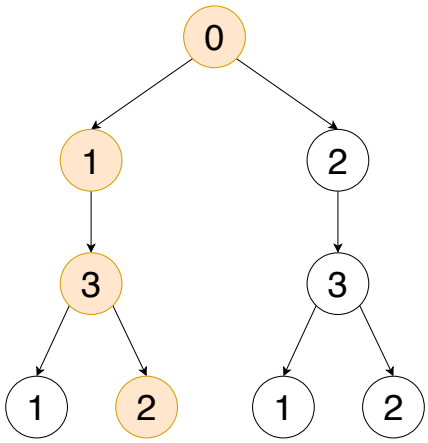


FIGURE 3.8. Path in T_{r_1,r_2} that produces the permutation $[0, 1, 3, 2]$ from M_{r_1,r_2}

We have not yet considered the constraint that each value in the array must be unique. We can alter our depth-first graph search to discard a path we are travelling on as soon as a repeated index is found.

Figure 3.9 illustrates the branch pruning that can be performed to discard unwanted permutations. The possible permutations are then exactly $[0, 1, 3, 2]$ and $[0, 2, 3, 1]$.

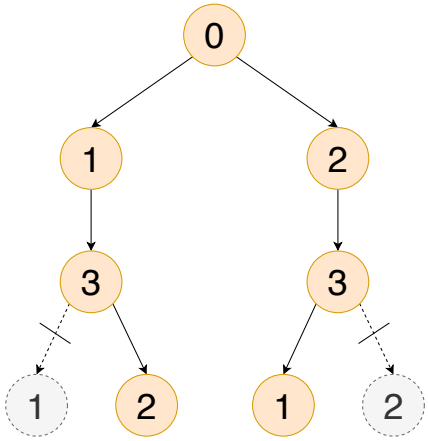


FIGURE 3.9. Possible paths down T_{r_1,r_2} after pruning

To avoid the exponential branching out factor due to the non-determinism, we can merge the identical nodes on each level. Each level i then only contains m_i nodes, where $m_i \leq length(\pi)$ is the number of possible moves for index i.

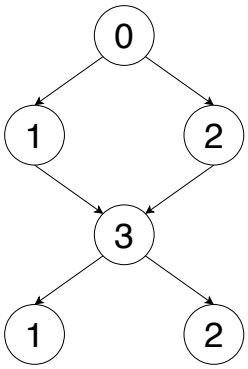


FIGURE 3.10. Tree representation of M_{r_1,r_2} after merging identical nodes

The extraction problem is then reduced to finding paths from the root to the leaves in such a graph, with the added uniqueness constraint. The devised algorithm, shown in Algorithm 11, uses the core idea of a stack from the usual depth-first algorithm to create each possible path, backtracking when a repeated number is found during the traversal.

Algorithm 11 Generating valid literal permutations

Given: Two clauses C_1 and C_2 with the same number of body atoms
Output: A set of valid permutations of the body atoms of C_2

```
1:  procedure GENERATEPERMUTATIONS(C_1, C_2)
2:      permutationMatrix ← CREATEPERMUTATIONMATRIX(C_1, C_2)
3:      ▷ create the list of valid moves
4:      validMoves ← ∅
5:      for each i ← 0 to n do
6:          for each j ← 0 to n do
7:              if permutationMatrix[i, j] == 1 then
8:                  append j to validMoves[i]
9:      ▷ begin extracting the valid permutations
10:     currentPermutation ← []
11:     seenPermutations ← ∅
12:     currentIdx ← 0
13:     S ← empty stack
14:     push validMoves[0] onto S
15:     while S is not empty do
16:         if currentIdx == n then
17:             ▷ permutation done, add it to list of valid permutations
18:             add currentPermutation to seenPermutations
19:             remove last element of currentPermutation
20:             decrement currentIdx
21:             continue
22:         possibleMoves ← pop from S
23:         if possibleMoves is empty then
24:             ▷ backtrack!
25:             decrement currentIdx
26:             remove last element of currentPermutation
27:             continue
28:         ▷ try the next possibility
29:         nextPossibleMove ← possibleMoves[0]
30:         push possibleMoves[1 :] to S
31:         if nextPossibleMove is already in currentPermutation then
32:             ▷ can't repeat values; skip!
33:             continue
34:         ▷ otherwise, everything is fine
35:         add nextPossibleMove to the end of currentPermutation
36:         increment currentIdx
37:         if currentIdx != clauseSize then
38:             ▷ add the set of valid moves for the next step
39:             push validMoves[currentIdx] to the stack
40:     return seenPermutations
```

We hence have an algorithm for generating a strict subset of all possible atom permutations. Although the permutation list may still contain false-positives (that is, permutations that cannot be used to show

equivalence), there are no false-negatives: if a permutation is not in the generated permutation set, then it is impossible for it to be used to show bijective equivalence as defined above. Moreover, since it is relatively rare for Datalog rules to contain several repeated body atoms in practice, the search space is significantly reduced. Even with the simple running example above, where a relation name *has* been repeated in the body, the number of valid permutations has been reduced from $4! = 24$ to only 2.

3.6.3.2 α: Finding a Corresponding Variable Assignment

The next stage of the process is to determine if a given permutation π on the body atoms of C_2 has an associated variable renaming function α on the variables of C_2 such that the rules are syntactically identical.

As the permutation π is given, we have a fixed ordering for how the literals will appear in C_1 and C_2. For example, consider again the running example from above. We found two valid permutations, $\pi_1 = [0, 1, 3, 2]$, and $\pi_2 = [0, 2, 3, 1]$. Recall that $\pi(p) = q$ means that the atom in position p will be moved to position q in the clause. If the first permutation π_1 is applied to clause C_2, we get the following new pair of clauses:

```
1   r1 : R(x) ← a(x), a(z), b(x,z).
2   r2 : R(k) ← a(w), a(k), b(k,w).
```

Naturally, by construction of the permutation through the above process, atom relation names must match up across the two clauses. We need to now only check that a valid variable renaming function exists. Since the permutation fixes the ordering for the bijection check, variables appearing in the same position in both clauses must be 'equated'. In the head atom above, for instance, `k` must necessarily be renamed to `x`. Moving on right in a similar manner, `w` must match up with `x`, `k` with `z`, `k` with `x`, and `w` with `z`. Notice, however, that `k` must map to both `x` and `z`. But `x` and `z` are necessarily distinct variables. We hence have a contradiction. Given this permutation, no variable renaming function is hence possible to transform r_2 to r_1. We can therefore discard the permutation $\pi_1 = [0, 1, 3, 2]$ as a potential permutation in the bijective equivalence check.

On the other hand, consider the second permutation, $\pi_2 = [0, 2, 3, 1]$. The formed pair of clauses is then:

```
1   r1 : R(x) ← a(x), a(z), b(x,z).
2   r2 : R(k) ← a(k), a(w), b(k,w).
```

The variable renaming approach, going left to right, then forces us to rename $\mathtt{k}$ to $\mathtt{x}$, $\mathtt{k}$ to $\mathtt{x}$, $\mathtt{w}$ to $\mathtt{z}$, $\mathtt{k}$ to $\mathtt{x}$, and $\mathtt{w}$ to $\mathtt{z}$. This time, we have a bijection with the variable renaming strategy from the variables in r_2 to the variables in r_1 that preserves the bindings:

$$\alpha_2 : \{\mathtt{k}, \mathtt{w}\} \to \{\mathtt{x}, \mathtt{z}\}$$

with $\alpha_2[\mathtt{k}] = \mathtt{x}$ and $\alpha_2[\mathtt{w}] = \mathtt{z}$.

A valid variable renaming function α_2 hence exists for the given permutation π_2. Similarly, constants must match up with equal constants; a variable cannot be mapped to a constant, and vice versa, as they cannot be syntactically identical.

Algorithm 12 determines if a valid variable-renaming function exists for a given permutation.

Algorithm 12 Checking whether a valid variable-renaming bijection exists

Given: Two clauses C_1 and C_2, and a permutation π on the atoms in C_2
Output: Whether a variable-renaming bijection exists that transforms C_2 into C_1

```
1:  procedure VARIABLERENAMEREXISTS(C_1, C_2, π)
2:      reorder the literals in C_2 based on the permutation π
3:      variableMap ← empty map
4:      for each variable v ∈ C_2 do
5:          variableMap[v] ← null
6:      for each i ← 0 to n do
7:          atom_{C_1} ← i^th atom in C_1
8:          atom_{C_2} ← i^th atom in C_2
9:          arity ← arity(atom_{C_1})
10:         for each j ← 0 to arity do
11:             arg_{C_1} ← j^th argument of atom_{C_1}
12:             arg_{C_2} ← j^th argument of atom_{C_2}
13:             if both arg_{C_1} and arg_{C_2} are constants then
14:                 if arg_{C_1} == arg_{C_2} then
15:                     return true
16:             else if both arg_{C_1} and arg_{C_2} are variables then
17:                 existingMapping ← variableMap[arg_{C_2}]
18:                 if existingMapping == null then
19:                     ▷ not set yet
20:                     variableMap[arg_{C_2}] = arg_{C_1}
21:                 else if existingMapping == arg_{C_1} then
22:                     continue
23:                 else
24:                     ▷ contradictory mapping
25:                     return false
26:             else
27:                 return false
28:     ▷ check if everything is covered
29:     for each (in, out) ∈ variableMap do
30:         if out == null then return false
31:     ▷ everything matches
32:     return true
```

3.6.3.3 $\sim$: Checking Clausal Bijective Equivalence

Algorithm 11 finds all possible permutations of the literals in C_2, while Algorithm 12 checks whether a variable-renaming bijection exists corresponding to that permutation. Combining these two, we hence have an algorithm for checking clausal bijective equivalence, shown in Algorithm 13.

Algorithm 13 Checking clausal bijective equivalence

Given: Two clauses C_1 and C_2
Output: Whether the clauses are bijectively equivalent

```
1: procedure AREBIJECTIVELYEQUIVALENT(C1, C2)
2:     if C1 and C2 do not belong to the same relation then return false
3:     if C1 and C2 have a different number of body atoms then return false
4:     possiblePermutations ← GENERATEPERMUTATIONS(C1, C2)
5:     for each π ∈ possiblePermutations do
6:         if VARIABLERENAMEREXISTS(C1,C2,π) then return true
       return false
```

For our running example, we have found a valid permutation and corresponding variable-renaming bijection transforming the second clause into the first. Thus, the two clauses are bijectively equivalent.

3.6.3.4 Minimising a Program under Clausal Bijective Equivalence

The clausal bijective equivalence check can then be used to devise a transformation that works towards minimising a program by deleting redundant clauses. To do this, we iterate through all clauses of all relations, and create equivalence classes based on bijective equivalence. As all but one clause of each equivalence class can be deleted, we need to only keep track of the representative of each equivalence class as we go along. The algorithm is shown in Algorithm 14.

Algorithm 14 Minimising a program under clausal bijective equivalence

Given: a Datalog program P
Output: transforms P into a semantically equivalent program P' with all clauses redundant under bijective equivalence deleted

```
1: procedure MINIMISEPROGRAM(P)
2:     for each relation R ∈ P do
3:         equivalenceReps ← ∅
4:         for each clause C ∈ R do
5:             added ← false
6:             for each C_eqClass ∈ equivalenceReps do
7:                 if AREBIJECTIVELYEQUIVALENT(C, C_eqClass) then
8:                     added ← true
9:                     delete C from P
10:                    break
11:            if not added then
12:                add C to equivalenceReps
```

3.6.4 Implementation

The transformation was implemented as a new AST transformation in the SOUFFLÉ Datalog evaluation engine. The transformation is called the `MinimiseProgramTransformer`. The process is fully automated, and was added to the main transformation pipeline.

Although the transformation was restricted to pure positive Datalog with no constraints, SOUFFLÉ has many extra features, including functors and negation. Currently, the transformation ignores the check for clauses not meeting the conditions. In the future, the algorithm can be extended to consider such features.

CHAPTER 4

The Transformation Pipeline

Thus far, all transformations have been discussed as standalone rewrite procedures on an input Datalog program. Although effective when individually applied, the potential benefits gained from their interaction can already be seen from the previous chapter; the whole is greater than the sum of the parts. For example, the singleton replacement procedure can lead to more opportunities for existential reduction, while a program-minimising transformer may be able to eliminate redundant rules formed by the body partitioner, and so on. The transformations may also be intertwined in more complex ways, or dependent on volatile conditions for application. Devising a solid framework and pipeline for transformer application is hence critical for maximising the gains of each transformation procedure when used in concert.

In this chapter, we will describe the original transformation framework built into SOUFFLÉ and its limitations. The discussion motivates the development of a meta-language for transformer application, where the implemented transformers can be wrapped into so-called *meta-transformers* that allow more complex interactions between transformations. Finally, we discuss the choices made in our application of the meta-transformer concept into SOUFFLÉ to build a theoretically justified transformation pipeline.

4.1 SOUFFLÉ: The Existing Transformation Framework

In the background section, we mention that SOUFFLÉ has a very minimal transformation framework. There are very few transformations, and all are just applied one after the other in sequence, exactly once each. The bulk of the transformations deal with resolving syntactic sugar so that the next stage of the process, translation into code for the Relational Algebra Machine (RAM), can occur. Such transformations are hence essential for code correctness. Most deal with complex constructs in SOUFFLÉ, however, that do not appear in regular Datalog code, such as aggregates, records, and components. As our research focuses on conventional Datalog, we will not discuss such transformations, and instead will abstract

them away into an overarching transformer henceforth dubbed the `ResolveComplexConstructs` transformer. `ResolveComplexConstructs` will only need to be run once as soon as the Abstract Syntax Tree (AST) is constructed for the input program.

There are a few relevant transformers in the original framework that will not be abstracted away: `ResolveAliases`, `RemoveEmptyRelations`, `RemoveRedundantRelations`, and `RemoveRelationCopies`.

4.1.1 Existing SOUFFLÉ Transformations

4.1.1.1 `ResolveAliases`

The `ResolveAliases` transformer resolves certain important constructs, such as solving equality constraints among variables and constants, and replacing each functor appearance in an atom with a variable and a corresponding equality constraint.

Although dealing more with syntactic sugar than optimality, we consider the transformer separate from `ResolveComplexConstructs` above, as it deals with primitive Datalog constructs seen in most dialects, such as equality constraints, that must still be resolved for the RAM stage to occur smoothly. Alias resolution may be expected by later transformations, which themselves may produce a program that requires the transformer to be run again, by, for example, introducing a new equality constraint. `ResolveAliases` hence serves as a good example of a critical transformation that may need to be rerun multiple times in a transformation pipeline, despite not directly optimising the program. The original SOUFFLÉ transformation framework was very basic, however, and so only required it to be run once, just after `ResolveComplexConstructs`.

4.1.1.2 `RemoveEmptyRelations`

As the name suggests, the `RemoveEmptyRelations` transformer removes relations containing no rules or facts. Such relations contain no tuples, and so are trivially always ‘false’ values. The relations, and all rules in which they appear in, can therefore be removed without consequence.

Since removing such relations may delete clauses, the transformation may have a cascading effect: deleting empty relation A may cause the relation B, which uses A, to be empty, and so on. Interestingly, however, the original SOUFFLÉ transformation framework ran the transformer only once. Resulting

programs could therefore still contain empty relations. It is more appropriate to iteratively apply the transformer until a fixpoint is reached; that is, until no more relations are deleted.

4.1.1.3 `RemoveRedundantRelations`

The `RemoveRedundantRelations` transformer removes all relations that do not contribute to the output of the program. In effect, all relations that an output query does not directly or indirectly depend on are deleted from the program. The transformation is a typical compiler optimisation in most languages. Only one pass is needed at a time, as deleting relations independent of an output query does not affect any relations the output relations actually depend on.

4.1.1.4 `RemoveRelationCopies`

Finally, the `RemoveRelationCopies` transformer considers all relations R containing a single clause, where the clause has the form:

$$R(x) \leftarrow S(x).$$

for some other relation S. The relation R is hence just a copy of the relation S, and performs no useful computations. All appearances of R can then be replaced by S, and R can be deleted from the program. Again, only one pass is needed at a time, as relation replacement cannot trigger the same condition for a new relation.

4.1.2 The Original SOUFFLÉ Transformation Pipeline

As stated earlier, the SOUFFLÉ transformation pipeline involved simply running each transformation exactly once in sequence. Since the transformers mostly did not perform complex program rewrites, the resolving transformations, such as `ResolveAliases`, did not need to be rerun. Correctness was therefore preserved in the output.

Program-optimising transformations did not flourish under such a regime, however. The limitations are evident in even this simple set of pre-existing transformations. As mentioned in the `RemoveEmptyRelations` subsection above, for example, there are some cases where relevant optimisation opportunities are missed, simply because the transformation is only run once.

Moreover, the original SOUFFLÉ framework was acceptable only due to the simple nature of the transformations. There were very few of them, each was fairly standalone, and all should be applied to any input program. It was natural to have a fixed order for the transformations, and execute them one at a time just after parsing.

With the newly introduced transformations, however, there are no longer any guarantees of simplicity. Complex transformations now exist, which can interact with each other in important ways. For example, singleton replacement (`ReplaceSingletonVariables`) can allow some relations to be reduced existentially (`ReduceExistentials`). Transformations may also now introduce new equality constraints, such as `InlineRelations`, which the `ResolveAliases` transformer must deal with before moving onto the next transformer. If not handled appropriately, such constraints can lead to a software-engineering nightmare, especially when new transformations are added, as well as a potential for missed optimisation opportunities.

For usability and extension purposes, the added complexity hence motivates the introduction of a meta-language to handle transformations on the AST level.

4.2 The Meta-Transformer Language

We now introduce a new, separate class of transformers, called *meta-transformers*. Rather than being standalone transformers that execute a single transformation pass when applied to a program, meta-transformers do not represent any new optimisation technique. Instead, a meta-transformer is comprised of subtransformers, which are handled in a certain way depending on its semantics. To avoid confusion, we will call the transformers discussed earlier *base-transformers*.

To allow a comfortable level of control over the base-transformers, we will devise several forms of meta-transformers that may be useful in a general framework of program-rewriting transformations. The main meta-transformers we will discuss are the `PIPELINE` transformer, the `CONDITIONAL` transformer, the `WHILE` transformer, and the `FIXPOINT` transformer. We will also introduce the `NULL` meta-transformer, which can be of aid when deleting or interacting with complex transformer chains.

The `PIPELINE` Transformer: The `PIPELINE` transformer holds a vector of subtransformations. When a `PIPELINE` transformer is executed, all subtransformers are executed in sequence. The subtransformers can be base-transformers, or meta-transformers themselves. An overarching main `PIPELINE`

transformer can be used to contain all transformation runs that need to be executed on the program. Therefore, in the end, only this transformation must be applied directly on an input program.

The `CONDITIONAL` Transformer: The `CONDITIONAL` transformer takes in a single subtransformation and a condition. If the condition holds when the `CONDITIONAL` transformer is executed on the program, then the subtransformation is applied. Otherwise, no transformation is performed. The subtransformer can again be either a base-transformer or a meta-transformer. An entire pipeline of transformers can hence be conditionally executed with a simple interface.

The `WHILE` Transformer: The `WHILE` transformer is similar to the `CONDITIONAL` transformer, except the transformation is reapplied to the program as long as the given condition holds.

The `FIXPOINT` Transformer: The `FIXPOINT` transformer takes in a single subtransformer, and executes it until the program no longer changes. The subtransformer can again be either a base-transformer or meta-transformer, and so entire series of optimising transformations can be executed in a looped sequence until all optimisation opportunities for these transformers have been consumed.

The `NULL` Transformer: Finally, the `NULL` transformer is a trivial meta-transformer that takes in no subtransformers, and performs no changes to the input program. The transformer can make interactions with the pipeline more flexible, especially when a transformer should be replaced in-place without affecting other parts of the chain.

The benefits of such a meta-transformation system can immediately be seen. For instance, the `RemoveEmptyRelations` transformer can now be wrapped in a `FIXPOINT` meta-transformer to overcome the issue of missed optimisation opportunities noted earlier. Moreover, the `InlineRelations` transformer can be wrapped in a pipeline which holds the inlining transformer, followed by the `ResolveAliases` transformer, to avoid potential RAM issues caused when introducing new equalities. New transformations, possibly composed of several subtransformers, can also be added easily to the main pipeline, or a subcomponent of the pipeline. Such transformers may be easier to implement. For example, rather than implementing the full inlining algorithm, a transformation that performs a single iteration could be wrapped in a fixpoint transformer.

Notice that the meta-transformers emulate constructs in common languages, such as conditionals and loops. By combining the meta-transformers, more intricate control can be gained over the collection of transformations. For example, wrapping a sequence of transformers in a pipeline allows us to reuse that

sequence whenever needed, similar to the idea of a callable subroutine. The meta-transformers are also beneficial in the sense that they add a layer of abstraction over the backend implementation, namely the C++ backend in this case. Recall that we did not want to restrict our transformations to any specific implementation of Datalog. The abstraction hence makes the transformation pipeline more portable to other languages.

We have hence developed a language of meta-transformers that wrap around our set of base-transformers. The meta-transformers were implemented as a separate class of transformers in the SOUFFLÉ system, allowing us to replace the existing framework with a more powerful fine-tuned transformation interface.

4.3 Constructing the SOUFFLÉ Transformation Pipeline

The original SOUFFLÉ transformation pipeline applied the following in sequence:

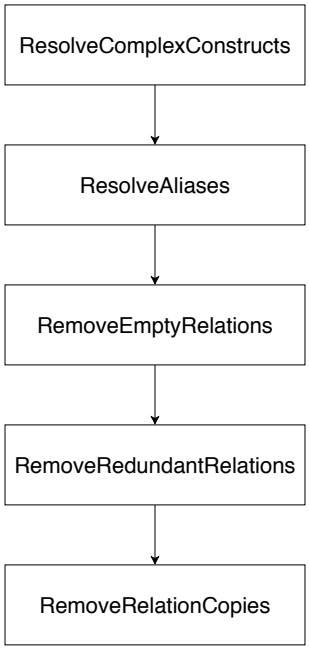


FIGURE 4.1. Original SOUFFLÉ transformation pipeline

We wish to now develop an effective pipeline incorporating the newly implemented transformations, in such a way that the maximal set of optimisation opportunities are taken. To justify the developed pipeline, the interactions between all transformers must be analysed.

4.3.1 Transformer Dependencies

In this section, we investigate transformer dependencies by analysing each base-transformer individually, noting which other transformers may gain optimisation opportunities. We begin with the pre-existing transformers.

`ResolveComplexConstructs` [COMPLEX]: This transformer must occur first, and does not need to be applied again. We can safely place the transformer at the beginning of the pipeline, and move on.

`ResolveAliases` [ALIAS]: Alias resolution works only on the interior of clauses, and does not remove clauses or otherwise alter inter-relational interactions. Due to the handling of equivalence by the current `MinimiseProgram` transformer, however, reducing the number of constraints might trigger bijective equivalence between two clauses. As alias resolution may remove existing equality constraints, the transformer should precede `MinimiseProgram`. Other transformations will not gain further optimisation opportunities.

`RemoveEmptyRelations` [EMPTY]: The transformer removes relations that are empty, along with all clauses in which the empty relations appear. As removing clauses may cause other to be empty, the transformer must be wrapped in a fixpoint meta-transformer. Removing empty relations may remove clauses from other relations, so introduces new potential for:

- `RemoveRelationCopies`: certain relations may now only have one clause, of the form $a(x) \leftarrow b(x)$.
- `ReduceExistentials`: certain relations may have only appeared non-existentially within a clause that is now deleted, and so can now be reduced.

`RemoveEmptyRelations` must hence precede these two transformations.

`RemoveRedundantRelations` [REDUN]: The transformer removes entire relations that are not needed to compute the query. As removed relations are entirely independent of the query, the remaining relations are not affected, and no further opportunities for optimisation are introduced.

`RemoveRelationCopies` [COPY]: The transformer removes 'middle-man' relations, and does not introduce new avenues for optimisation.

We now move on to the transformers introduced in Chapter 3.

`ReorderLiterals` [PERM]: Literal reordering only rearranges atoms within clauses. No other transformation relies on literal ordering, so no further optimisation opportunities arise. However, literal reordering must occur whenever clauses are changed or added. The transformer should hence be executed as the final transformation stage, and will not be discussed further.

`InlineRelations` [INLINE]: Inlining typically introduces equality constraints, and so must be followed by `ResolveAliases`. Inlining may also cause clauses to become equivalent, and so `MinimiseProgram` must eventually follow it. In case the user inlines a propositional relation, `PartitionBodyLiterals` should also be run after inlining.

`ReplaceSingletonVariables` [SING]: Singleton variables may cause certain relations to become existential, creating new opportunities for `ReduceExistentials`. `ReplaceSingletonVariables` must therefore precede `ReduceExistentials`.

`ReduceExistentials` [EXIST]: Reducing existentials can only remove recursive rules. In the rare event that a relation only contains recursive rules, `RemoveEmptyRelations` should be executed afterwards. The same points mentioned in `RemoveEmptyRelations` also apply: relations may now be copies of other relations, and the sole non-existential appearances of some relations may have been removed. New optimisation opportunities may hence be introduced for `RemoveRelationCopies`, as well as `ReduceExistentials` itself.

`PartitionBodyLiterals` [PART]: Partitioning body literals may result in the generation of several equivalent relations. Optimising out such redundancies has not yet been implemented. The transformer hence does not introduce any new optimisation opportunities for other existing optimisations.

`MinimiseProgram` [MIN]: Finally, `MinimiseProgram` reduces the set of clauses, but only removes redundant clauses. `RemoveRelationCopies` is the only transformer that may now have new opportunities, as it relies on the number of clauses defined for a relation.

The dependencies between the transformers are illustrated in the graph in Figure 4.2. The nodes represent the transformations. An edge from a to b means transformation a may introduce new optimisation opportunities for b, and so b should occur at some point after a in the pipeline. The dependencies therefore impose ordering constraints on the constructed pipeline. Transformers `ResolveComplexConstructs` and `ReorderLiterals` are not shown, as they will always appear first and last in the pipeline respectively, as mentioned above.

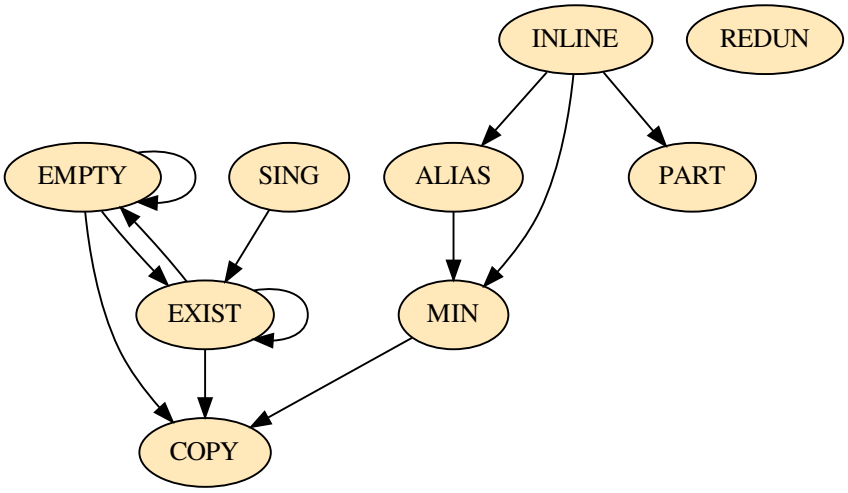


FIGURE 4.2. Dependencies between AST transformations implemented in SOUFFLÉ

As new transformers come into play or old transformers evolve in complexity with future work, the dependencies will likely be affected. For example, if `MinimiseProgram` is extended to cater for equivalent relations, then `PartitionBodyLiterals` will introduce for it a relevant optimisation opportunity. In such a case, a new edge should be inserted from `PartitionBodyLiterals` to `MinimiseProgram` in the dependency graph.

The edges in the graph already hint at the importance of considering transformer interactions when developing an optimal framework of transformations. It is important to note that a lack of edges between nodes does not at all indicate that the transformers are independent, however. For example, `PartitionBodyLiterals` will extract subclauses into new relations. The new extracted relations may benefit more thoroughly by singleton replacement, using `ReplaceSingletonVariables`, than the original program, due to a shift in the structure of the clauses. Nevertheless, it does not matter if singleton removal occurs before or after body partitioning, and so a dependency edge is not required; it is only important that both transformers are run at some point.

4.3.2 The Final Pipeline

Using the graph G in Figure 4.2, we can begin to devise a transformation pipeline for SOUFFLÉ. The goal is a pipeline that produces a program such that no more optimisation opportunities exist for any implemented transformation. We therefore base pipeline construction on the following conditions:

- All transformers should be applied at least once

- If a transformer a is applied and makes a change to the program, then all transformations b where the edge (a, b) is in the dependency graph G must eventually be applied

In an acyclic graph, pipeline construction is simple: a topological sorting algorithm on the graph will produce an appropriate ordering of transformers. Unfortunately, the dependency graph G contains directed cycles. For example, `RemoveEmptyRelations` contains a self-loop, and so must continuously be applied. Once a program no longer changes after the application of a given transformer, however, further applications are unnecessary. A natural consequence is to wrap such cycles in `FIXPOINT` meta-transformers, such that they are followed until there is no longer an impact on the program.

To construct the pipeline, we therefore begin by reducing all strongly-connected components in the dependency graph into nodes that represent a wrapping fixpoint transformation. The idea is akin to the construction of the Strongly-Connected Component (SCC) graphs developed by SOUFFLÉ to divide programs into strata. The dependency graph after the reduction of strongly-connected components is shown in Figure 4.3. Nodes with a bold outline indicate a component originating from a directed cycle, which should therefore be wrapped in a fixpoint transformer.

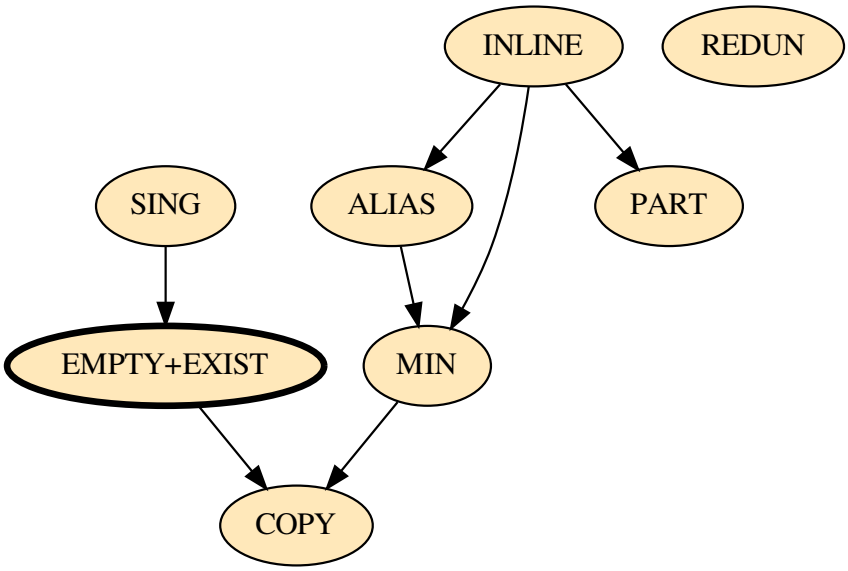


FIGURE 4.3. SCC graph corresponding to dependency graph in Figure 4.2

The result is a directed acyclic graph, with possible fixpoint nodes. The reduction means we can now apply a simple topological sorting algorithm on the graph. Any solution corresponds to a valid transformation pipeline of the shown transformations. One such solution is shown in Figure 4.4. Node numberings indicate the order of application.

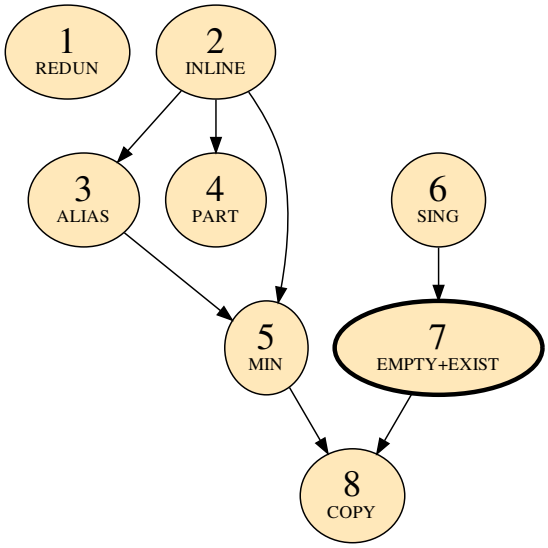


FIGURE 4.4. SCC graph with nodes numbered to show a valid topological ordering

Adding in `ResolveComplexConstructs` and `ReorderLiterals` at the start and end respectively, and wrapping `RemoveEmptyRelations` and `ReduceExistentialRelations` in a common fixpoint transformer, we have a valid transformation pipeline for SOUFFLÉ. The final pipeline is shown in Figure 4.5.

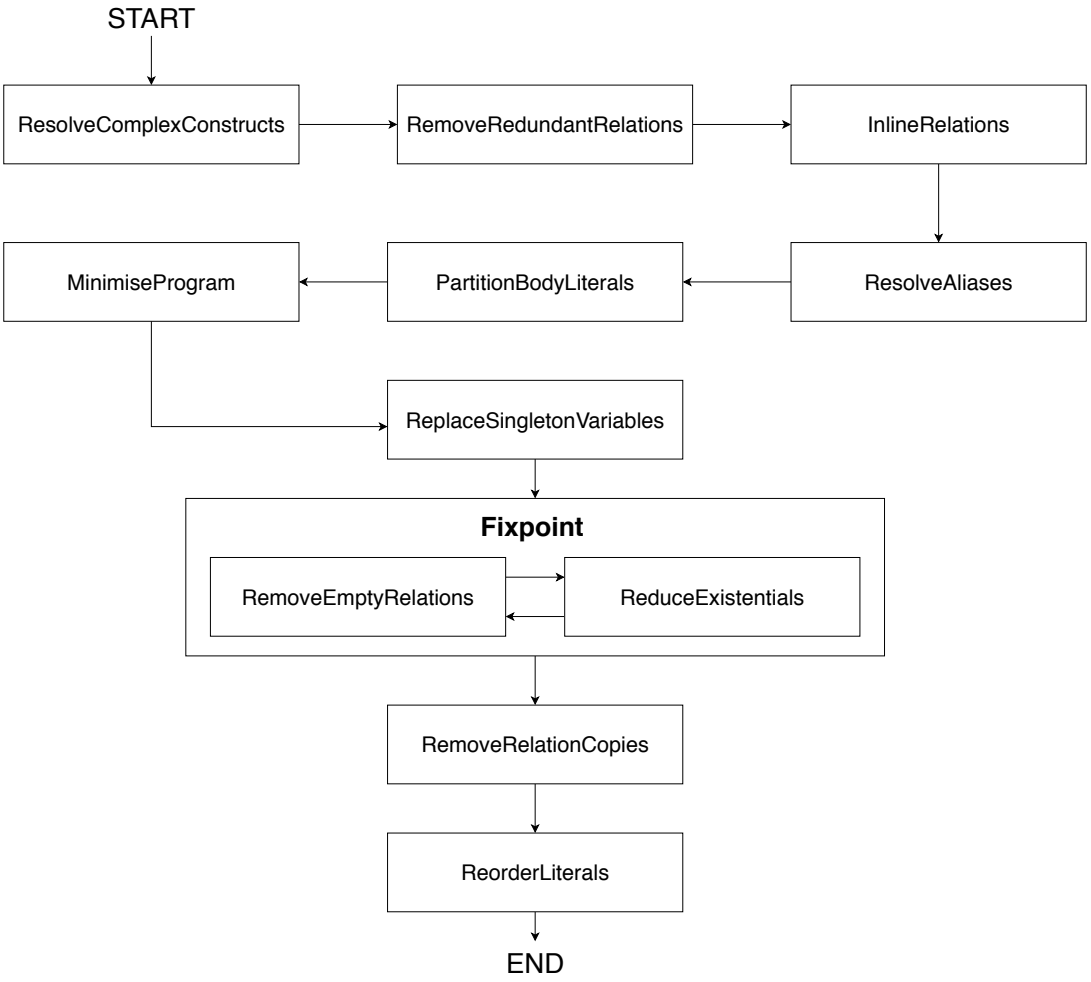


FIGURE 4.5. Final SOUFFLÉ transformation pipeline, based on the dependency analysis

CHAPTER 5

Experiments and Results

The transformations have until this point only been justified from a theoretical standpoint. In this chapter, we move on to demonstrating the impact of each of these transformations on both synthesised and real-world Datalog programs and fact sets. We aim to achieve an understanding of the impact of each transformation, when run both individually and in concert, by answering the following experimental questions.

(Q1) What is the impact of each transformer on runtime and memory when applied on:
  (i) synthesised programs?
  (ii) real-world programs?

(Q2) How do these impacts scale with larger input datasets?

(Q3) How do the efficiency impacts of the automated transformations compare to semi-automated transformations requiring manual annotations?

(Q4) How do the transformers perform when working together, rather than as standalone transformations?

For our experimental setup, we use an Intel Xeon Gold 6130 CPU @ 2.10GHz server running Fedora 27 with 192 GB of memory. To compile the SOUFFLÉ C++ executables, we use GCC 7.3.1 with the `-O3` flag. For each experiment, evaluation time in seconds and memory usage in megabytes is recorded. A timeout of 15 minutes was chosen for all experiments. We use maximum resident set size to measure memory. A limitation for resident set size is that it overapproximates for low memory usage, and so a plateau for lower memory may occur. As our focus is on the trend for larger input sizes, such differences can be safely ignored.

5.1 Synthesised Benchmarks

We begin with an empirical investigation of the effects of our transformations on hand-crafted Datalog programs. For each transformation, we present a simple Datalog program with an optimisation opportunity specially targeted to the transformer in question. The programs will have a variable fact set, allowing tests to range in size. Throughout this section, N denotes the size of the input fact set. We will demonstrate the performance of each transformation on its corresponding Datalog program for input fact sizes of increasing orders of magnitude. Results in this section will be shown on a logarithmic scale.

Recall that some transformations require manual user annotations to be enabled. We therefore split the analysis up into automated transformations - `PartitionBodyLiterals`, `ReduceExistentials`, `ReplaceSingletonVariables`, and `MinimiseProgram` - and semi-automated transformations - `ReorderLiterals` and `InlineRelations`. When testing each transformation, all others will be disabled to illustrate individual performance benefits. We conclude the section by investigating the effects of the transformations in concert on a separate benchmark.

5.1.1 Automated Transformations

5.1.1.1 `PartitionBodyLiterals`

The synthesised program is as follows:

```
1   .decl jailed(x:number)
2   .input jailed
3
4   .decl thief(x:number)
5   .input thief
6
7   .decl person(x:number)
8   .input person
9
10  .decl worried(x:number)
11  worried(x) :- person(x), !jailed(x), thief(y), !jailed(y).
12
13  .output worried
```

The optimised program has the form:

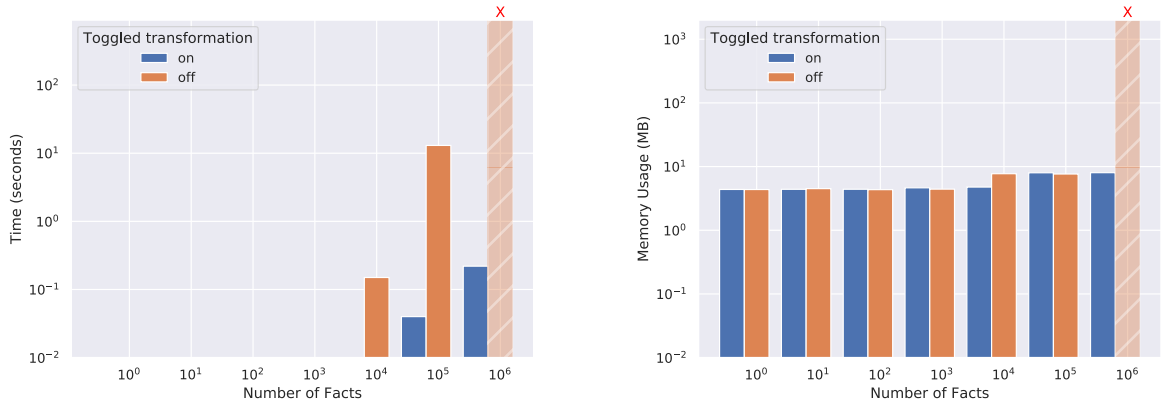
```
.decl jailed(x:number)
.input jailed

.decl thief(x:number)
.input thief

.decl person(x:number)
.input person

.decl exists_thief()
exists_thief() :- thief(y), !jailed(y).

.decl worried(x:number)
worried(x) :- exists_thief(), person(x), !jailed(x).

.output worried
```

With regards to input facts:

- $INPUT(\texttt{person}) = \{1, \ldots, N\}$
- $INPUT(\texttt{thief}) = \{x \in \{1, \ldots, N\} \mid x \equiv 0 \mod 3\}$
- $INPUT(\texttt{jailed}) = \{x \in \{1, \ldots, N\} \mid x \equiv 0 \mod 6\}$

For each possible unjailed person, the unoptimised program must redundantly recompute the value of `exists_thief`, which scans through every element in the `thief` relation - $O(N)$ time. Thus, $O(N)$ extra time is spent per unjailed person. While the optimised program will scale linearly, the optimised program will hence scale quadratically with the input size. As the same amount of information is stored per relation (ignoring the possible single extra tuple stored in `exists_thief`) and the same atoms are being checked, the memory usage is expected to be similar for both programs.

The performance of the Datalog program when run on SOUFFLÉ is shown in Figure 5.1.



(A) Evaluation time

(B) Memory usage

FIGURE 5.1. SOUFFLÉ performance on synthesised Datalog program with the `PartitionBodyLiterals` transformer toggled on or off

As expected, the optimised program scales much better. The evaluation time of the program is near instantaneous for both optimised and unoptimised versions up until $N = 10^4$. While evaluation time remains at a subsecond level when the transformer is toggled on, the time shoots up for the unoptimised version, before timing out at the $N = 10^6$ point. Memory is fairly constant across all fact sizes where the unoptimised version does not time out, except for the $N = 10^4$ case, where the optimised version uses around 23% less memory. We speculate that this is a consequence of memory management techniques for large-size relations in the optimised data-structures in the backend.

5.1.1.2 `ReduceExistentials`

The synthesised program is as follows:

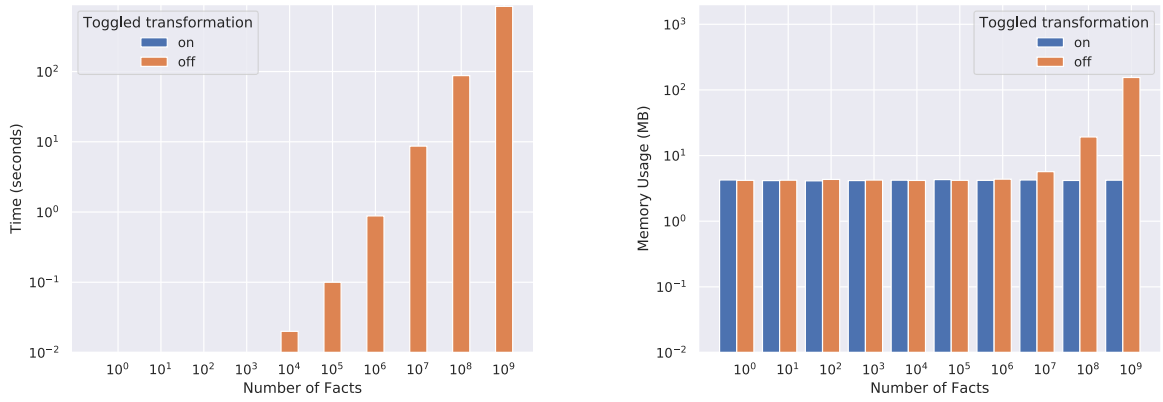
```
1  .decl natural(x:number)
2  natural(0).
3  natural(x+1) :- natural(x), x < $N$.
4
5  .decl query()
6  query() :- natural(_).
7
8  .output query
```

The optimised program has the form:

```
1   .decl natural(x:number)
2   natural(0).
3
4   .decl query()
5   query() :- natural(_).
6
7   .output query
```

It is clear that the original program will likely scale linearly with the input fact size, while the optimised program will take constant time and memory, as it is independent of N.

The performance of the Datalog program when run on SOUFFLÉ is shown in Figure 5.2.



(A) Evaluation time

(B) Memory usage

FIGURE 5.2. SOUFFLÉ performance on synthesised Datalog program with the `ReduceExistentials` transformer toggled on or off

As expected, time and memory were constant and relatively extremely low for the optimised program. Conversely, the unoptimised program seems to scale roughly linearly with input size for both time and memory usage. A dramatic constrast between the two programs is hence increasingly evident as input size increases.

5.1.1.3 `ReplaceSingletonVariables`

The synthesised program is as follows:

```
1   .decl natural(x:number)
2   natural(0).
3   natural(x+1) :- natural(x), x < $N$.
4
5   .decl a(x:number)
6   a(0) :- natural(x), natural(y).
7
8   .decl query(x:number)
9   query(x) :- a(x).
10
11  .output query()
```

The optimised program has the form:

```
1   .decl natural(x:number)
2   natural(0).
3   natural(x+1) :- natural(x), x < $N$.
4
5   .decl a(x:number)
6   a(0) :- natural(_), natural(_).
7
8   .decl query(x:number)
9   query(x) :- a(x).
10
11  .output query()
```

The imperative code for relation `a` in the original program will consist of a nested for loop:

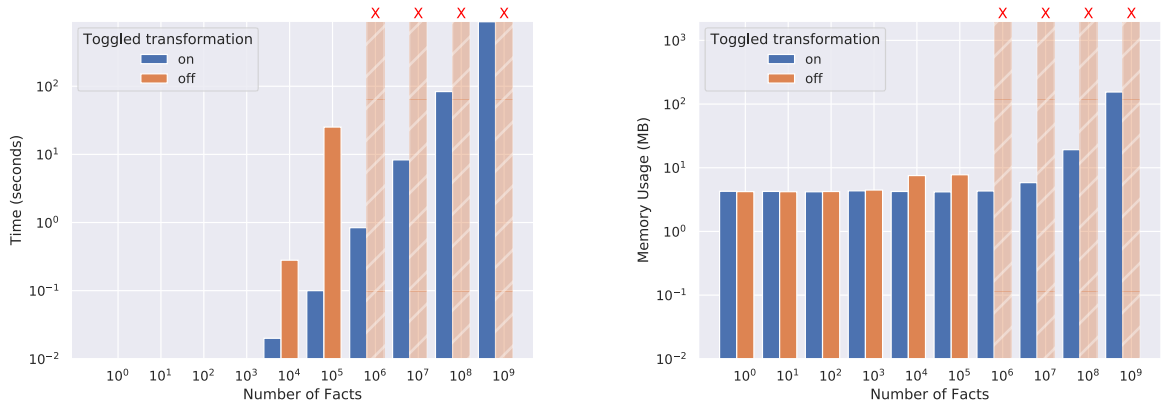
```
1   FOR EACH TUPLE (x) IN RELATION natural:
2      FOR EACH TUPLE (y) IN RELATION natural:
3         ADD (0) TO RELATION a
```

while the optimised version will have the form:

```
1   IF ∃ TUPLE IN RELATION natural:
2       IF ∃ TUPLE IN RELATION natural:
3           ADD (0) TO RELATION a
```

As `natural` has size N, the evaluation time of `a` will hence likely remain constant for the optimised program, while scaling quadratically for the unoptimised. Memory should be similar for both, as no significant extra storage is required, though the added nested loop may introduce some memory overhead. Note that the `natural` relation itself must still be computed for both, taking $O(N)$ time and space, so the optimised program should still scale linearly in size and memory with input size.

The performance of the Datalog program when run on SOUFFLÉ is shown in Figure 5.3.



(A) Evaluation time

(B) Memory usage

FIGURE 5.3. SOUFFLÉ performance on synthesised Datalog program with the `ReplaceSingletonVariables` transformer toggled on or off

As expected, the optimised program scales linearly, while the unoptimised blows up, before timing out at the $N = 10^6$ stage. Memory is fairly constant and equal for both program, until the $N = 10^4$ point, where some extra overhead seems to come into play. The overhead is similar to that seen in the results for `PartitionBodyLiterals` at the same point, and we again speculate that this is a consequence of memory management techniques from the optimised data-structures used. Note that, after the $N = 10^6$ point, the memory usage of the program increases; this is likely a result of the input facts beginning to dominate memory usage, rather than the computation itself.

5.1.1.4 `MinimiseProgram`

The synthesised program is as follows:

```
1   .decl a(x:number,y:number)
2   .input a
3
4   .decl b(x:number,y:number)
5   .input b
6
7   .decl c(x:number, y:number)
8   c(x,y) :- a(x,z), b(z,y).
9   c(s,t) :- b(u,t), a(s,u).
10
11  .output c
```

The optimised program has the form:

```
1   .decl a(x:number,y:number)
2   .input a
3
4   .decl b(x:number,y:number)
5   .input b
6
7   .decl c(x:number, y:number)
8   c(x,y) :- a(x,z), b(z,y).
9
10  .output c
```

The input relations this time take in a pair, rather than a single element. The input facts for a size N test will simply be the set $\{(x, y) \mid x \in \{1, \ldots, N\}, y \in \{1, \ldots, N\}\}$ for both relations.

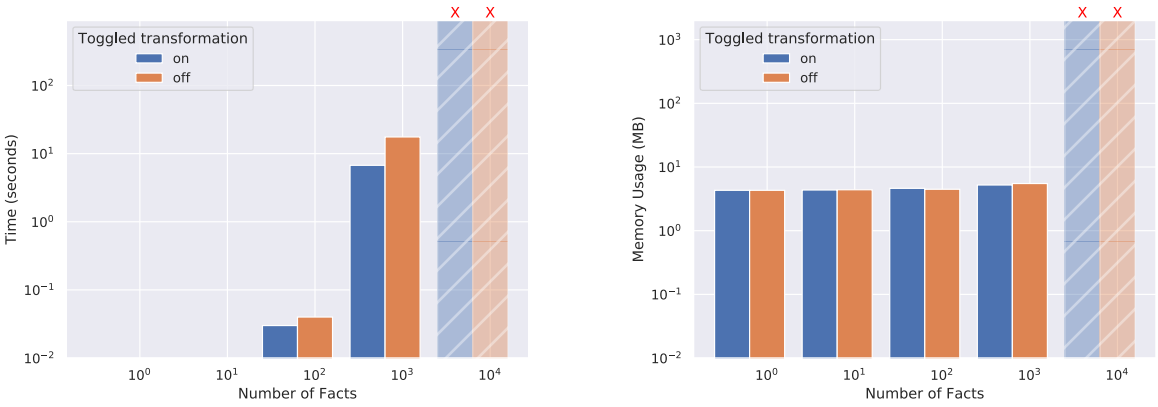
The only difference between the two programs is the duplicated rule, which has the following form in the translated imperative code:

```
1   FOR EACH TUPLE (x,z) IN RELATION a:
```

```
2       FOR EACH TUPLE (z,y) IN RELATION b:
3           ADD (x,y) TO RELATION c
```

The outer loop iterates through N^2 elements, while the inner loop iterates through an extra N per tuple, to ground the value of `y`. The extra loop hence takes $O(N^3)$ time. The loop appears in both programs, however, but is duplicated in the original. As the loop dominates the computation, it is expected that the optimised program will hence take roughly half the time as the unoptimised. Memory should remain constant for both, however, as no memory overhead is added or removed; the loop is simply duplicated in the original.

The performance of the Datalog program when run on SOUFFLÉ is shown in Figure 5.4.



(A) Evaluation time

(B) Memory usage

FIGURE 5.4. SOUFFLÉ performance on synthesised Datalog program with the `MinimiseProgram` transformer toggled on or off

The results match expectations fairly consistently. Both tests increase in time fairly rapidly, and both time out at the same order of magnitude, but have similar memory usage throughout. Nevertheless, on the largest input size prior to timeout, the time taken for the unoptimised program is over 2x the time for the optimised version, as expected.

5.1.2 Semi-Automated Transformations

5.1.2.1 Inlining

The synthesised program is:

```
1   .decl natural(x:number)
2   natural(0).
3   natural(x+1) :- natural(x), x < SIZE.
4
5   .decl natural_pair(x:number, y:number) inline
6   natural_pair(x,y) :- natural(x), natural(y).
7
8   .decl query(x:number, y:number)
9   query(x,y) :- natural_pair(x,y), y = x * x.
10
11  .output query
```

The imperative version, ignoring the recursive `natural` relation computed as part of the first stratum, is:

```
1   FOR EACH TUPLE (x) IN RELATION natural:
2      FOR EACH TUPLE (y) IN RELATION natural:
3         ADD (x,y) TO RELATION natural_pair
4
5   FOR EACH TUPLE (x,x*x) IN RELATION natural_pair:
6      ADD (x,x*x) TO RELATION query
```

Note that the `natural_pair` relation is marked to be inlined. The optimised program, after inlining and resolution of equality constraints, therefore has the form:

```
1   .decl natural(x:number)
2   natural(0).
3   natural(x+1) :- natural(x), x < SIZE.
4
5   .decl query(x:number, y:number)
```

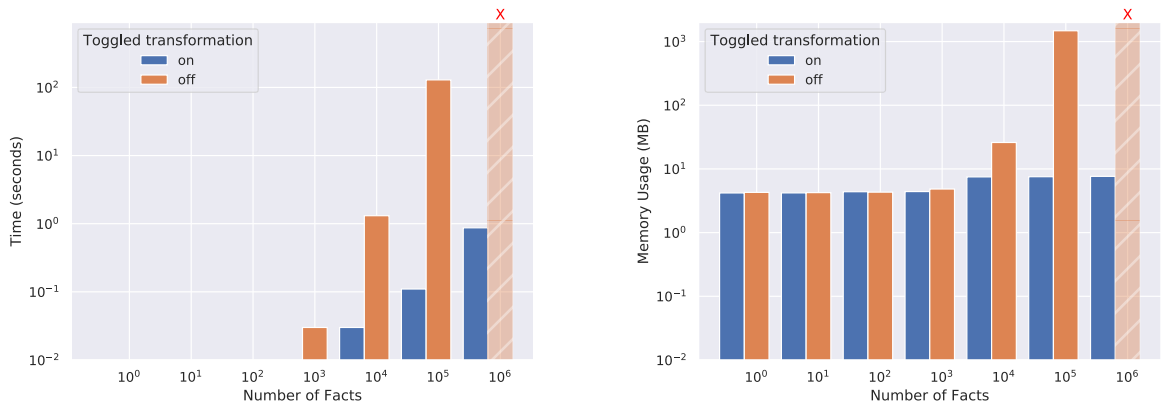
```
6    query(x,x*x) :- natural(x), natural(x*x).
7
8    .output query()
```

The imperative version, again ignoring the `natural` relation, is:

```
1    FOR EACH TUPLE (x) IN RELATION natural:
2       IF (x*x) ∈ RELATION natural:
3          ADD (x,x*x) TO RELATION query
```

Notice that the original program takes $O(N^2)$ time, as it must scan the `natural` relation twice in a nested for loop. On the other hand, the optimised program only takes $O(N)$ time, as the variable `x` becomes bounded, meaning a simple existential check is enough rather than a full second scan. Furthermore, the `natural_pair` relation no longer needs to be stored. As the relation stored $O(N^2)$ tuples, and no new relation is required to take its place, we should also see significant improvements in memory usage.

The performance of the Datalog program when run on SOUFFLÉ is shown in Figure 5.5.



(A) Evaluation time

(B) Memory usage

FIGURE 5.5. SOUFFLÉ performance on synthesised Datalog program with the `InlineRelations` transformer toggled on or off

Time and memory were improved extremely by inlining the relation. While the optimised program is still at a subsecond level for an input of size $N = 10^6$, the unoptimised program times out. Memory

is also more or less consistent across input sizes for the optimised program, while the original begins to increase in memory rapidly as input size increases, especially once the quadratic factor of the `natural_pair` relation begins to dominate memory usage. A similar slight bump in memory usage is observed for at $N = 10^4$ for the optimised version as seen in `PartitionBodyLiterals` and `ReplaceSingletonVariables`.

5.1.2.2 Reordering

The synthesised program is:

```
1   .decl a(x:number)
2   .input a
3
4   .decl b(x:number)
5   .input b
6
7   .decl bad(x:number)
8   bad(0).
9   bad(x+1) :- bad(x), x < 10.
10
11  .decl query(x:number,y:number)
12  query(x,y) :- a(x), b(y), bad(100).
13
14  .output query
```

The output imperative version of the single `query` clause is:

```
1  FOR EACH TUPLE (x) IN RELATION a:
2     FOR EACH TUPLE (y) IN RELATION b:
3        IF (100) ∈ RELATION bad:
4           ADD (x,y) TO RELATION query
```

The optimised program result with no SIPS (that is, no heuristic is used, and the original ordering of rules is preserved) will not change the program. With the transformation toggled on with any given SIPS option, however, the optimised program in this case will look as follows:

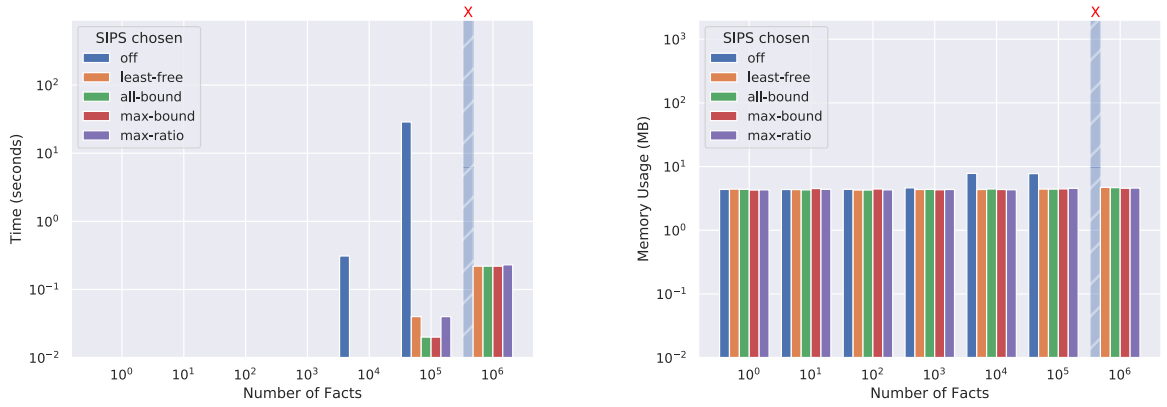
```
1   .decl a(x:number)
2   .input a
3
4   .decl b(x:number)
5   .input b
6
7   .decl bad(x:number)
8   bad(0).
9   bad(x+1) :- bad(x), x < 10.
10
11  .decl query(x:number,y:number)
12  query(x,y) :- bad(100), a(x), b(y).
13
14  .output query()
```

The only change is the position of the fully bound atom, `bad(100)`, in the query clause. As all heuristics developed prioritise such atoms, and the remaining atoms all have independent free variables as their arguments, the result is consistent across SIPS options. The imperative version is hence just a rearrangement of the statements, as follows:

```
1   IF (100) ∈ RELATION bad:
2      FOR EACH TUPLE (x) IN RELATION a:
3         FOR EACH TUPLE (y) IN RELATION b:
4            ADD (x,y) TO RELATION query
```

Notice from the program that the tuple `(100)` can never be in the relation `bad`. Therefore, the conditional statement will fail in both cases. In the unoptimised version, we perform a double-nested for loop scan of both relations either way, however, leading to an $O(N^2)$ time approach. Conversely, the optimised program will fail the check immediately, skipping the entire loop chunk, hence taking constant time. As only the order of atoms has changed, we expect memory to remain roughly the same.

The performance of the Datalog program when run on SOUFFLÉ is shown in Figure 5.6.



(A) Evaluation time

(B) Memory usage

FIGURE 5.6. SOUFFLÉ performance on synthesised Datalog program with the `ReorderLiterals` transformer toggled on or off, with each possible SIPS option

As expected, the program scales significantly worse for the unoptimised program, with proportional gains similar to those observed with `InlineRelations` and `PartitionBodyLiterals`. There is some slight variability in the data at the $N = 10^5$ time analysis, though this is negligible due to the small scale. Again, we see a slight bump in memory at the $N = 10^4$ range, which is speculated to have the same causes as the similar bumps discussed fo the `InlineRelations`, `PartitionBodyLiterals`, and `ReplaceSingletonVariables` experimental results.

5.1.3 Transformer Synergy

Thus far, the results have shown promising results for all transformations when acting individually on simple synthesised programs. We still wish to observe whether the whole is greater than the sum of its parts. That is, does the interaction of transformers aid in our optimisation pipeline? To investigate, we write a new Datalog program aimed at taking advantage of the optimisation opportunities introduced by each transformer, discussed in Chapter 4.

```
1    .decl a(x:number,y:number)
2    .input a
3
4    .decl b(x:number,y:number)
5    .input b
6
```

```
7   .decl natural(x:number)
8   natural(0).
9   natural(x+1) :- natural(x), x<1000000.
10
11  .decl natural_pair(x:number,y:number) inline
12  natural_pair(x,y) :- natural(x), natural(y).
13
14  .decl c(x:number,y:number)
15  c(x,y) :- a(x,z), b(z,y), natural_pair(_,_).
16  c(s,t) :- b(u,t), a(s,u), natural_pair(x,y).
17  c(a,b) :- a(a,y), b(y,b), natural_pair(_,_).
18
19  .decl query(x:number)
20  query(x) :- c(x,_).
21
22  .output query
```

Running the program through the pipeline, as dictated in Chapter 4, the transformations produce the following final program:

```
1   .decl a(x:number,y:number)
2   .input a
3
4   .decl b(x:number,y:number)
5   .input b
6
7   .decl exists_natural()
8   exists_natural().
9
10  .decl c(x:number,y:number)
11  c(a,b) :- exists_natural(), exists_natural(), a(x,z), b(z,y).
12
13  .decl query(x:number)
14  query(x) :- c(x,_).
15
```

```
16    .output query
```

A combination of transformations are needed as part of certain stages of the transformation. For example, the relation `c` cannot be reduced to one clause by the `MinimiseProgram` transformer without first replacing all singleton variables. We would still get some benefit without `ReplaceSingletonVariables` however, as `c` could be reduced to two clauses regardless. Similar interactions occur for other combinations of transformers. We hence expect that disabling larger sets of transformers will result in a significant degradation of performance.

The original program was run with every possible combination of transformers, and the results are summarised in Figure 5.7 (evaluation time) and Figure 5.8 (memory usage). The input fact size was $N = 1000$,

$$INPUT(\texttt{a}) = INPUT(\texttt{b}) = \{(x, y) \mid x \in \{1, \ldots, N\}, y \in \{1, \ldots, N\}\}$$

Each node represents a subset of the transformers. The label on the node corresponds to which transformers were disabled: M means `MinimiseProgram`, P means `PartitionBodyLiterals`, E means `ReduceExistentials`, S means `ReplaceSingletonVariables`, and I means `InlineRelations`. As we go up the lattice, a larger set of transformers are disabled. We mark each cluster of similar results with the same colour, green indicating the best group, orange indicating the middle group, and black indicating a timeout.

Note that the final transformation, `ReorderLiterals`, was not shown to have an impact regardless of SIPS chosen (including being disabled), and so is not shown in the figures to reduce clutter. It is likely because of a lack of optimisation opportunities present due to the simplicity of the program and data distribution of input facts. Moreover, the implementation of `PartitionBodyLiterals` already prepends added propositions to the beginning of clauses, as discussed in Chapter 3, and so the potential benefit of `ReorderLiterals` in the presence of propositions has already been applied.

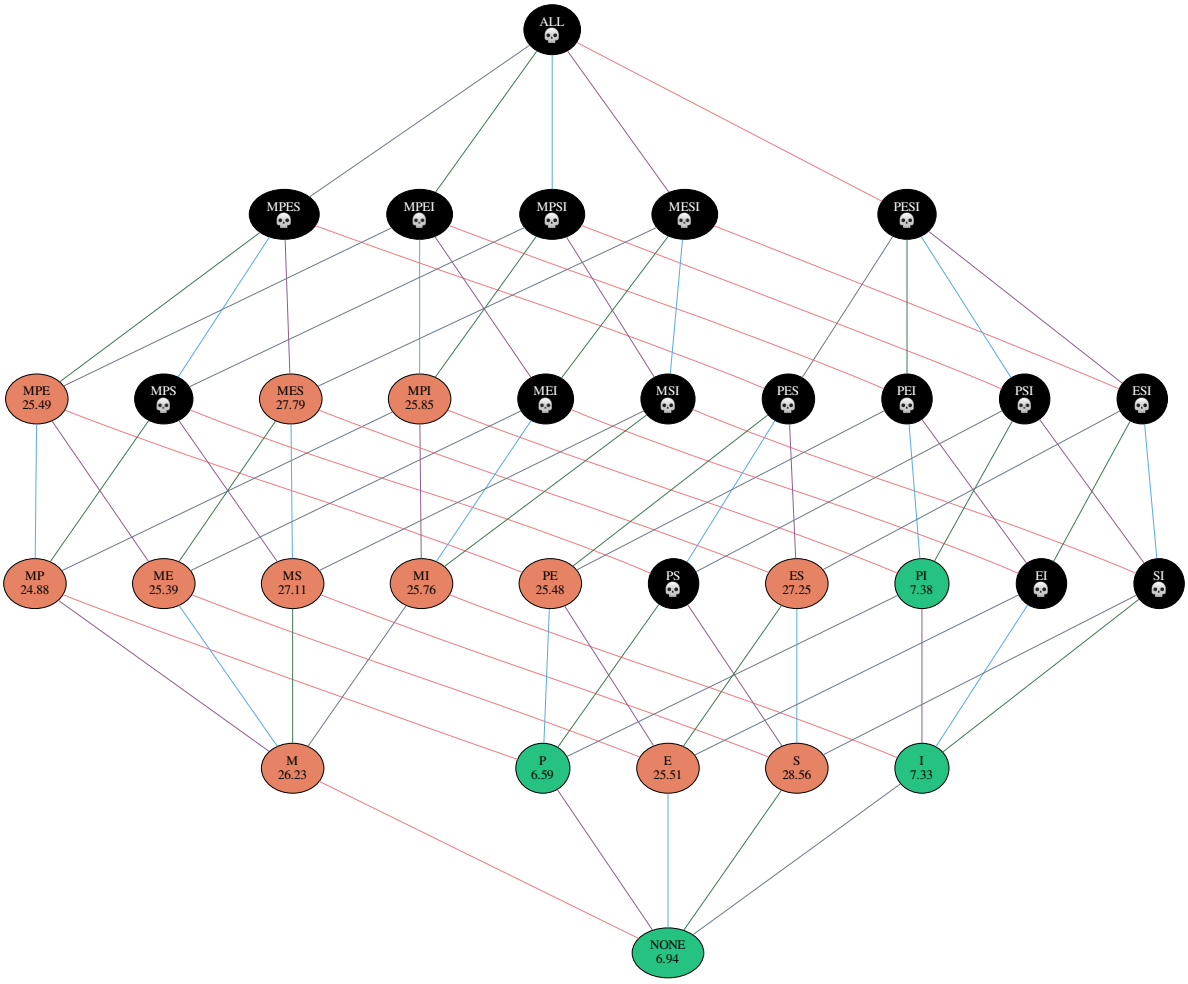


FIGURE 5.7. SOUFFLÉ evaluation time in seconds on synthesised Datalog program testing all combinations of disabled transformers

Notice that, as we climb up the time lattice in Figure 5.7, disabling more and more transformations, evaluation time increases. In particular, as soon as a program times out when a set T_1 of transformers are disabled, then it also times out for any set of disabled transformers $T_2 \supseteq T_1$. Interestingly, when only one transformer remains, regardless of which transformer it is, the program will always time out. Moreover, disabling only one transformer does not always produce significant degradations in performance. The scale of performance improvements is dramatic itself: when only two transformers are enabled, the 15 minute timeout limit is always reached; as soon as a third transformer is enabled, performance either

remains at the timeout limit, or drops to less than half a minute. When transformers interact with each other well, they interact with each other *very* well.

As a specific example, consider node P on the first level, where `PartitionBodyLiterals` was disabled. Although not affecting performance when disabled on its own, as soon as `RemoveSingletonVariable` is disabled as well, the program times out. The transformer hence has more of a background effect on the program here, acting as a fallback in the case that other transformers are not present. At least one of the combination is hence necessary for the program to not time out. The interaction is very interesting, as it highlights the note mentioned when analysing transformer dependencies in Chapter 4: despite not introducing new opportunities for one another, the two transformers interact in a way necessary for maximal efficiency. Nevertheless, if these two transformers remain, while all others are disabled, we still end up with a timeout. The transformers are hence necessary, but not sufficient. A similar effect occurs with nodes EI and SI.

Another interesting observation is the node PI. Although green, indicating top performance, the program times out as soon as any other transformer is disabled. Therefore, despite not leading to a significant decrease in performance when disabled on their own or together on this particular program, they together provide an important framework of redundancy for other transformations to fall back on.

The transformations offer a contrasting impact on memory usage, as shown in Figure 5.7.

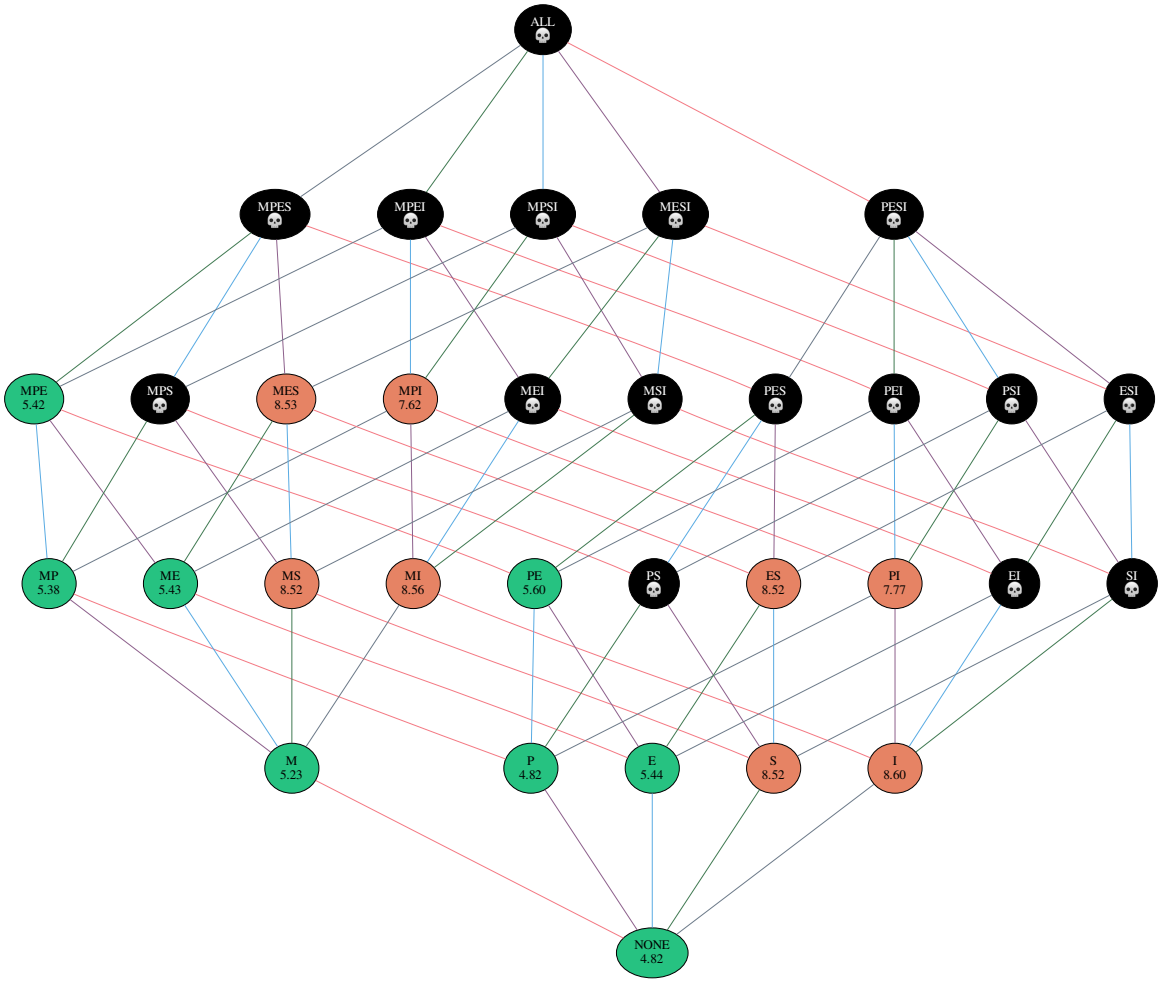


FIGURE 5.8. SOUFFLÉ memory usage in megabytes on synthesised Datalog program testing all combinations of disabled transformers

As we climb the lattice, memory usage increases, as expected. However, the effect of each transformation set in the memory lattice is very different to the effects observed in the time lattice. For instance, `MinimiseProgram` no longer has the dramatic effect it does on time as it does on memory usage, whilst inlining becomes a core factor. Such changes are expected: some transformations lead to memory improvements, while others provide only temporal gains. Inlining removes intermediate relations, leading to lower storage overhead. On the other hand, `MinimiseProgram` only removes redundant clauses, which would store no extra tuples anyway.

Interestingly, the intersection of green nodes in Figure 5.7 and Figure 5.8 is the set of exactly two nodes: the node where no transformation is disabled, and the node where only `PartitionBodyLiterals` is disabled. When looked at in isolation, it may therefore seem as if the partitioning transformer is not critical for performance gains. However, even in the simple synthesised program, it is an important fallback when other transformations are not used. In other programs, other transformations may take this role.

The interaction between the optimising program-rewrites is hence an important consideration when developing a solid transformation framework. The work of multiple transformers in concert is critical for efficient program execution.

From the above experiments, we have found that both the automated and semi-automated transformations can lead to incredible gains in performance. The gains in scalability seen are exciting; although SOUFFLÉ is already praised for its scalability in the Datalog community, the results show a significant improvement for each transformation. In particular, the original SOUFFLÉ engine was often observed to time out after the 15 minute limit, while the optimised version was typically below a minute, and sometimes almost instantaneous. Memory usage also scaled well, especially for inlining. Despite being the state-of-the-art in scalability, SOUFFLÉ therefore still has several potential avenues for improvement in this area. When working in concert, the transformations also showed gains well beyond the power of the work of any individual transformer. These results already illustrate the potentials of a solid high-level program-rewriting optimisation framework, especially with the interaction of several transformers.

5.2 Real-World Benchmarks

The transformations, both automated and semi-automated, have illustrated a dramatic impact thus far on the synthesised benchmarks. It is important that the optimisations devised in this thesis can also be effectual when applied on real-world benchmarks, namely SOUFFLÉ Datalog benchmarks that are used in practice. We therefore shift our attention now to two existing large-scale benchmark suites: the publically available DOOP Dacapo benchmarks for static program analysis (Antoniadis et al., 2017; Bravenboer and Smaragdakis, 2009), and private Amazon network-security benchmarks.

The analyses crafted for the DOOP Dacapo benchmarks are extremely well-written hand-optimised code, expertly made to explicitly take advantage of the evaluation system behind SOUFFLÉ. Consequently, very little improvement is expected with respect to both time and memory for these benchmarks, as related manual optimisations have likely already been made.

The Amazon network-security benchmarks are not handled in the same way. Amazon analyses are generated from a domain-specific language to ease the writing procedure. As a result, the performance may have been sacrificed, with few manual optimisations performed. We hence expect some visible improvement in performance.

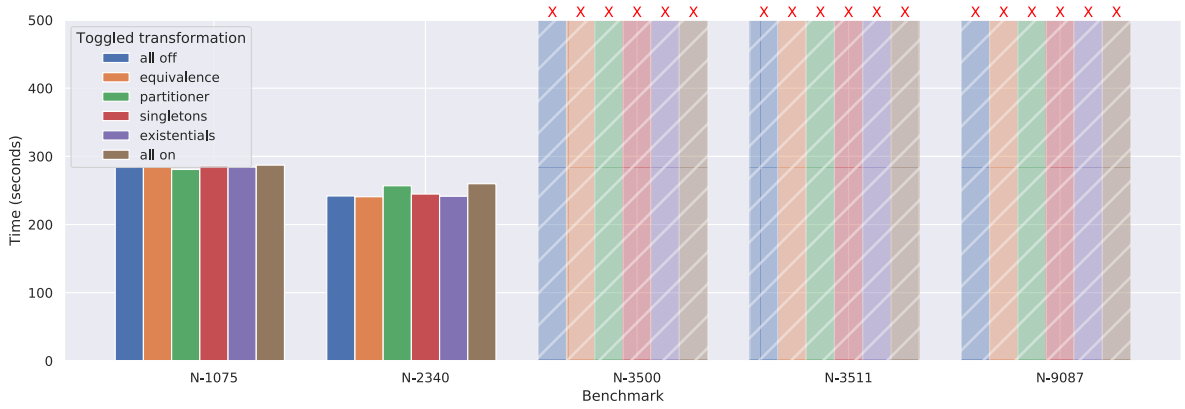
Observing the impact on both benchmarks will hence provide unique insights on performance in the real-world. Note that, regardless of the benchmark suite chosen, real-world static-analysis programs in Datalog tend to be more streamlined than the simple programs shown in the previous section on synthesised benchmarks, despite being considerably larger. That is, recursion is not very common, and relations are typically expressed to only compute what they need to. We therefore do not expect improvements to the same level as those shown earlier, but hope to see changes nonetheless.

Again, we divide the experimental analysis into automated and semi-automated transformations. As semi-automated transformations are more fine-tuned to the program being analysed, we expect their effect to be more significant than that of fully automated transformers. We conclude with a look at the complementary effects of the optimisations when applied in concert.

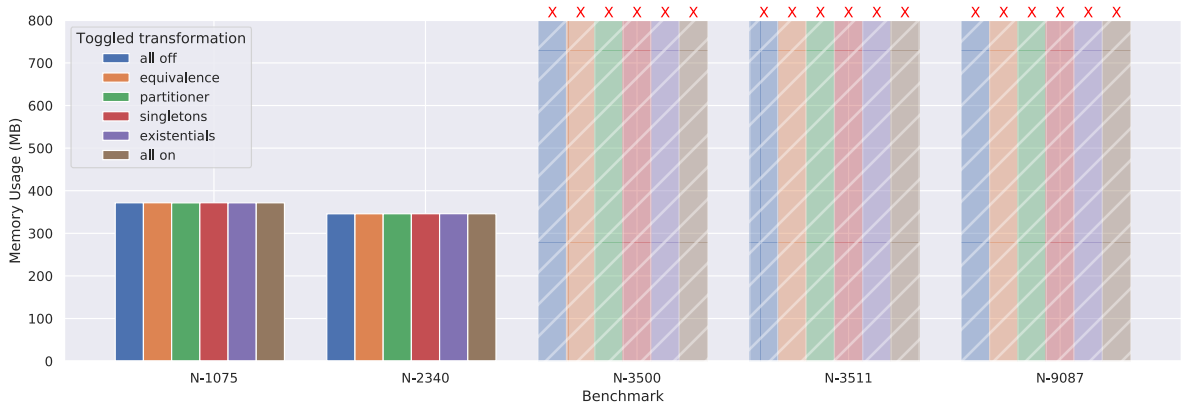
5.2.1 Automated Transformations

There are four automated transformers: `MinimiseProgram`, `PartitionBodyLiterals`, `ReplaceSingletonVariables`, and `ReduceExistentials`. For each benchmark, we evaluate the performance with respect to both memory and time, based on five conditions: all transformers disabled, each transformer enabled on its own, and all transformers enabled.

Figure 5.9 shows the effect of each condition on the Amazon benchmarks, running the sec1 analysis.



(A) Evaluation time

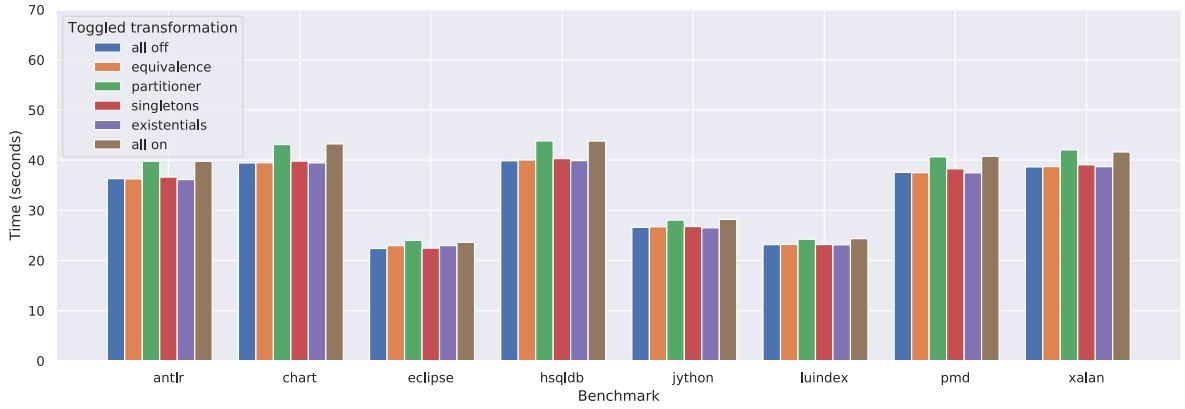


(B) Memory usage

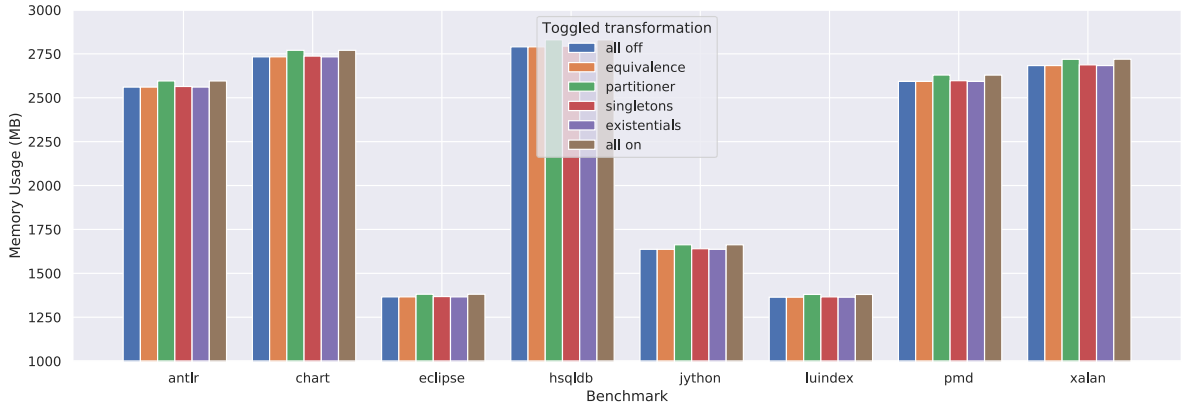
FIGURE 5.9. Performance of SOUFFLÉ on the sec1 analysis with and without the new automated transformations on Amazon benchmarks

The transformations do not seem to have a significant effect on time nor memory. In fact, partitioning body literals seems to have a negative effect on time. The effect is tiny: only a 3% increase in time taken. The negative effect is likely due to the added relational overhead by the transformer being greater than the benefits obtained by extracting disconnected clause components.

A very similar effect can be seen on the DOOP benchmarks in Figure 5.10, running the context-insensitive analysis.



(A) Evaluation time



(B) Memory usage

FIGURE 5.10. SOUFFLÉ performance of context-insensitive analysis with and without the new automated transformations on DOOP Dacapo benchmarks

As before, no significant impact can be seen on time nor memory, and partitioning seems to have a small negative impact on evaluation time.

The automated transformations therefore do not currently improve the investigated real-world Datalog programs. The results starkly contrast those of the synthesised benchmarks, where certain programs were made instantaneous even in the presence of just one transformer, and negative effects were not observed. It may hence be beneficial to more closely analyse the true inefficiencies in the programs using manual human observations. Although the DOOP benchmarks are not expected to have many such inefficiencies, due to already hand-optimised code, Amazon code may have yet undiscovered optimisation opportunities.

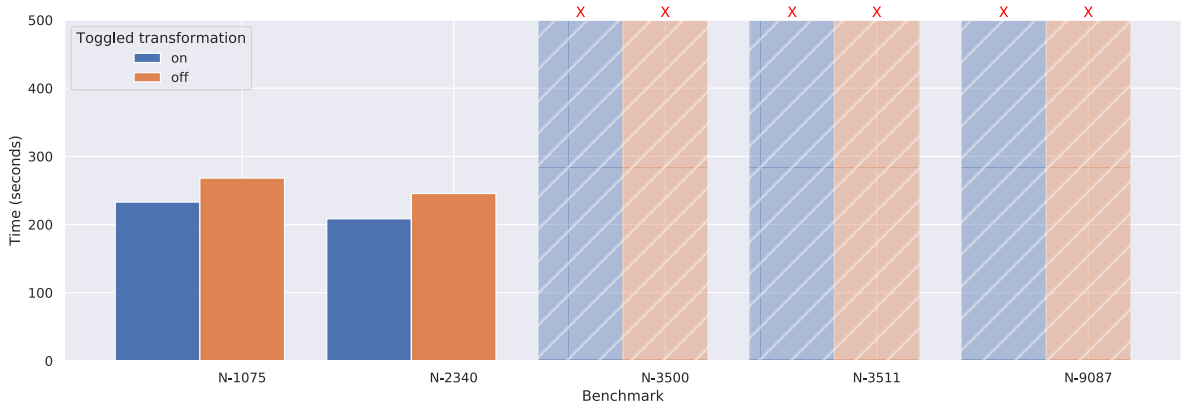
5.2.2 Semi-Automated Transformations

We now move on to the semi-automated transformations, `InlineRelations` and `ReorderLiterals`. As mentioned above, semi-automated transformations rely on manual user annotations, and so are more fine-tuned to the program being analysed. The transformational effects are hence expected to be more significant than the automated transformers.

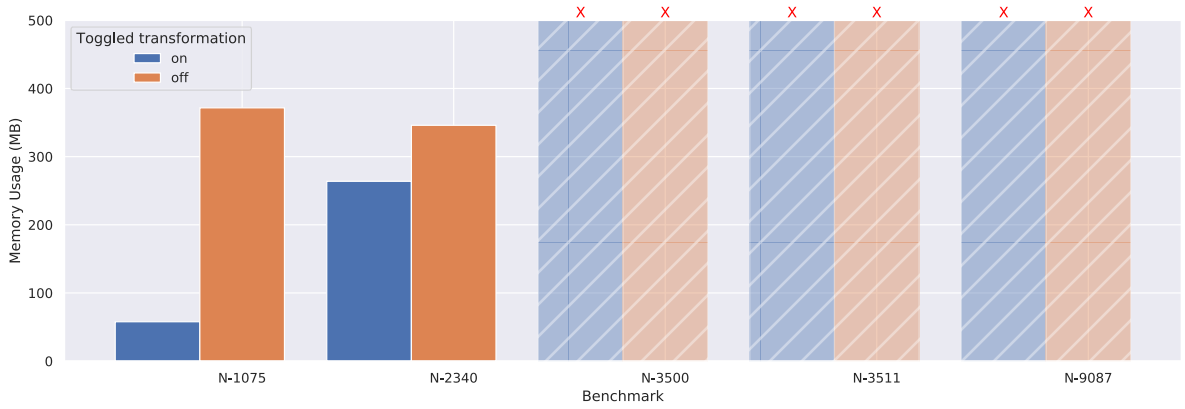
5.2.2.1 Inlining

For the inlining transformation, relations must be chosen to be inlined. As there are 277 relations in the Amazon analysis, and 429 relations in the DOOP analysis, it is infeasible to evaluate each possible combination of inlined relations. Recall from Section 3.1 that it is best to inline large relations that are not used many times. We employ the profiler built into SOUFFLÉ to find such relations. As only a crude heuristic and intuition is involved in selecting relations to inline at this stage, there may exist a more effective combination of relations to inline.

Figure 5.11 shows the results when the sec1 analysis is run on each Amazon benchmark.



(A) Evaluation time



(B) Memory usage

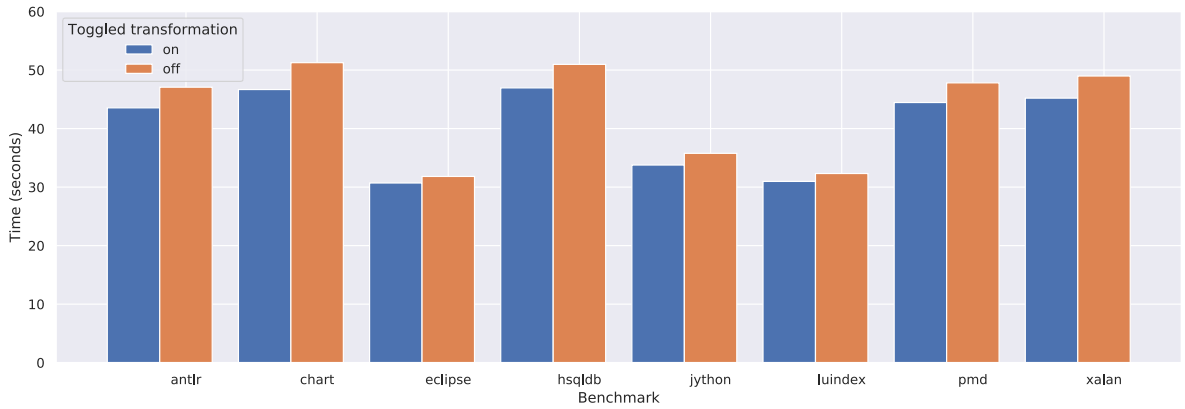
FIGURE 5.11. SOUFFLÉ performance of sec1 analysis with and without selective inlining on Amazon benchmarks

A considerable improvement in evaluation time was found for all benchmarks that did not time out. However, the effect on memory usage is exceptional: the first benchmark reduced memory usage by approximately 85%, while the second was reduced by almost 25%. Table 5.1 summarises the raw results and percentage improvement of inlining on the Amazon benchmarks that did not time out, N-1075 and N-2340.

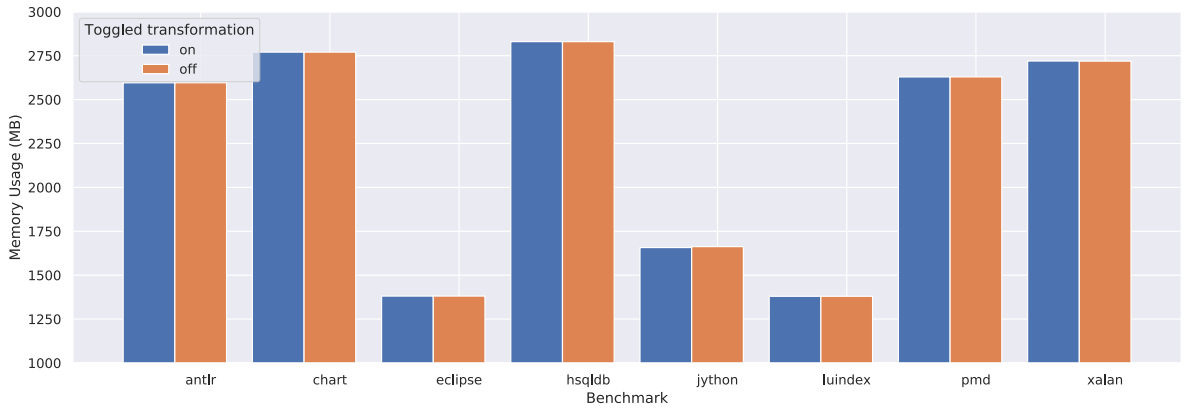
		Inlining		
		Off	**On**	**Improvement**
N-1075	**Time (s)**	268.13	232.92	13.13%
	Memory (MB)	371.59	57.74	84.46%
N-2340	**Time (s)**	245.52	208.4	15.12%
	Memory (MB)	346.00	263.81	23.75%

TABLE 5.1. Performance impact of inlining running sec1 analysis on Amazon benchmarks

Time improvements can surprisingly also be seen in the DOOP benchmarks, despite no memory improvement, as shown in Figure 5.12. The lack of improvement in memory usage is likely due to the same non-inlined relations in the program dominating memory space.



(A) Evaluation time



(B) Memory usage

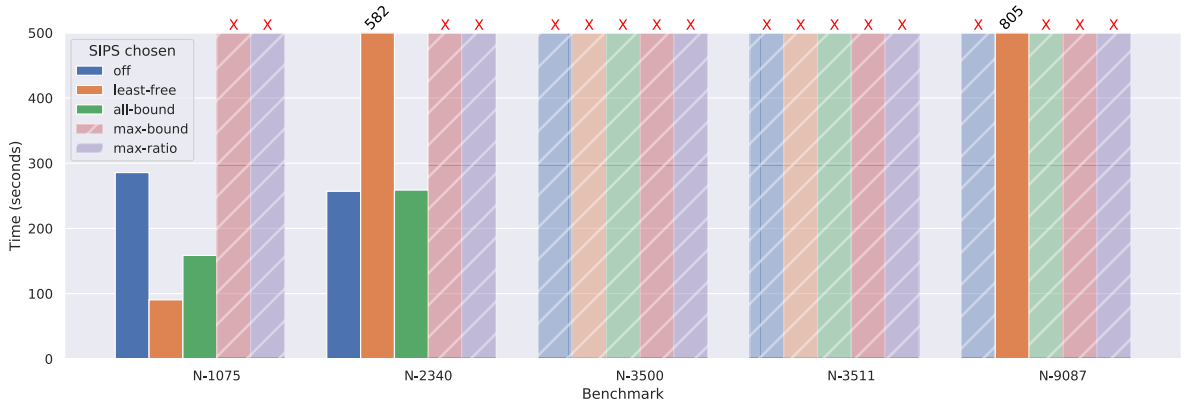
FIGURE 5.12. SOUFFLÉ performance of context-insensitive analysis with and without selective inlining on DOOP Dacapo benchmarks

The effects of inlining already show promising results for the semi-automated transformations, when compared to the automated transformations. With a more fine-tuned profile analysis, along with possible rounds of trial and error on different datasets, an even more effective choice of relations to inline could be found.

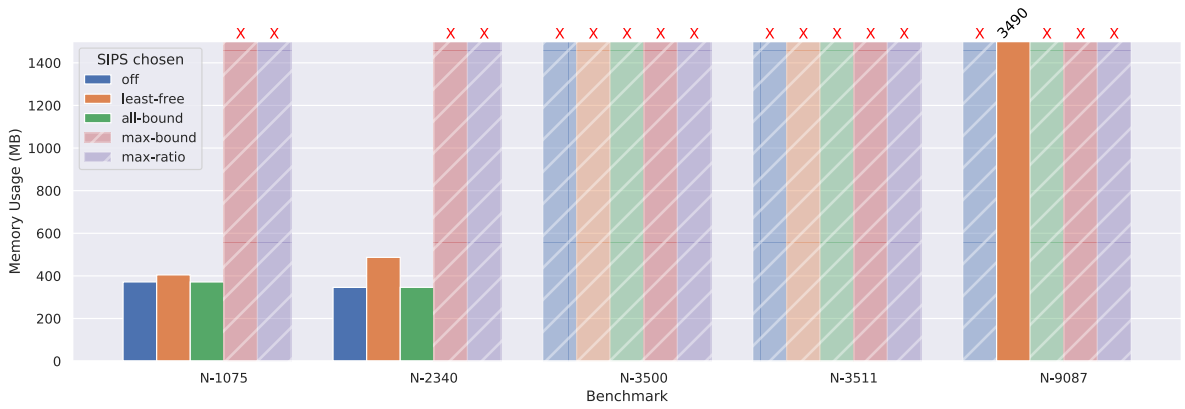
5.2.2.2 Reordering

For the literal-reordering transformer, the atom-prioritising heuristic (called the SIPS in our implementation) must be chosen. We test each SIPS possibility: disabled, least-free, all-bound, max-bound, and max-ratio.

Figure 5.13 shows the effect of the reordering transformer on the sec1 analysis when run on the Amazon benchmarks.



(A) Evaluation time



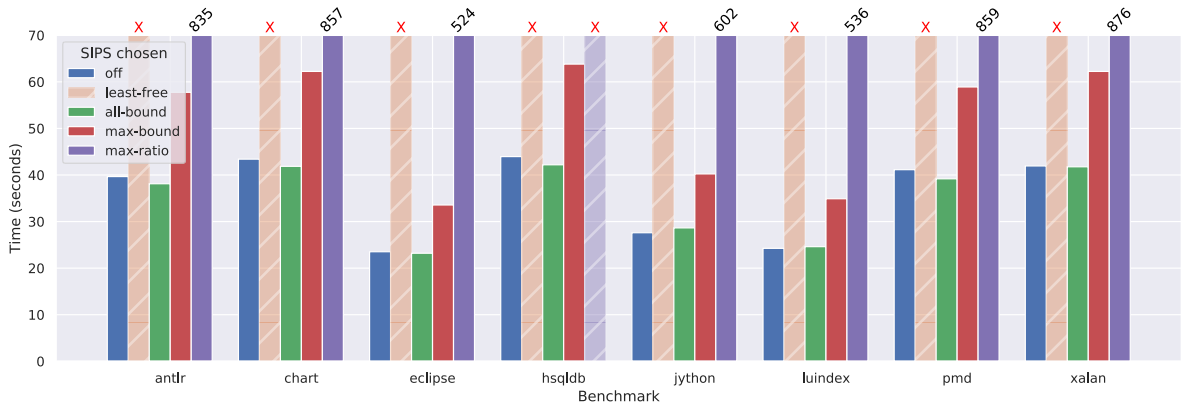
(B) Memory usage

FIGURE 5.13. SOUFFLÉ performance of sec1 analysis with varying SIPS on Amazon security benchmarks

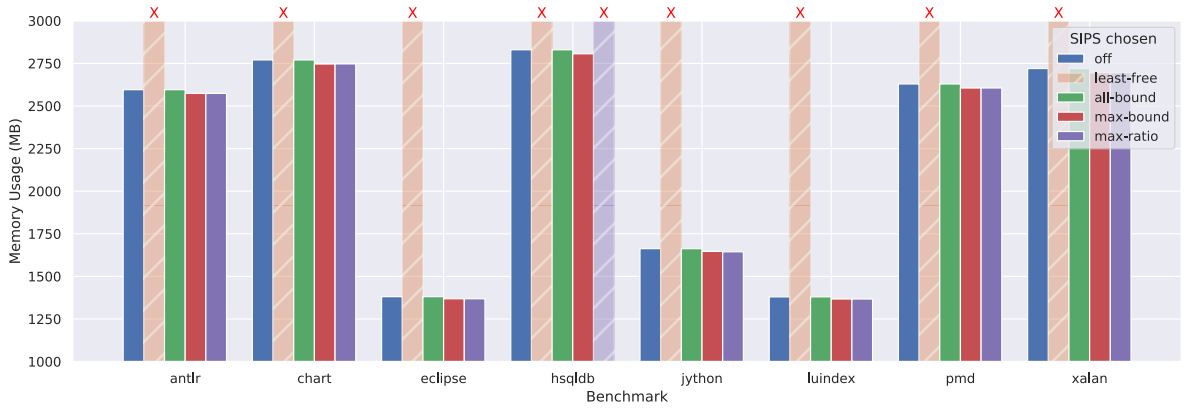
The impact on time is the best seen so far on the first benchmark, with an improvement of approximately 68% when the least-free heuristic is chosen, and of 45% when the all-bound heuristic is used. The least-free heuristic also finally shifted the final benchmark outside the range of a timeout for the first time. Interestingly, however, the least-free heuristic almost doubled evaluation time for the second benchmark. This shows the importance of the input fact-set, particularly the data distribution, on how a reordering strategy impacts performance. Nevertheless, the all-bound heuristic is guaranteed to never worsen performance, as shown in Section 3.5, and can already be seen as an effective default SIPS option. The max-bound and max-ratio heuristics, on the other hand, both resulted in a timeout in all cases, and so may be less suited for the given analysis. Memory remained fairly consistent on reorderings that

did not time out, though least-free had a noticeable increase in memory usage for the second benchmark. We speculate that the spike is due to increased loop overhead with the new ordering.

The same analysis was run on the DOOP benchmarks, shown in Figure 5.14.



(A) Evaluation time



(B) Memory usage

FIGURE 5.14. SOUFFLÉ performance of context-insensitive analysis with varying SIPS on DOOP Dacapo benchmarks

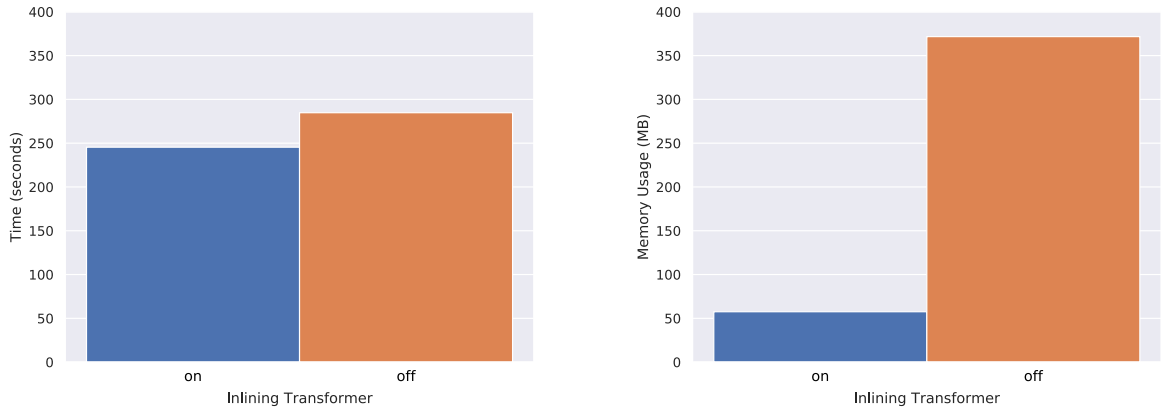
A stark contrast is seen when compared to the Amazon benchmarks. Most noticeably, the only timeout that occurs this time is with the least-free heuristic, which showed the most promising gains for Amazon. Marginal evaluation time improvements are seen when the all-bound heuristic is chosen, though not very significant. Interestingly, despite being less theoretically justified in Section 3.5, max-ratio preforms worse than max-bound on these benchmarks. The difference is likely a consequence of the max-bound

heuristic matching the data distribution better in this case; the number of bound arguments is a better estimate of relation size rather than the ratio.

It is therefore evident that the reordering transformation can have a substantial impact on evaluation time, even on real-world benchmarks. The choice of SIPS is critical in ensuring performance gains, however. An analysis of data distribution, possibly with the aid of an extended profiler for SOUFFLÉ, may aid in the choice. Regardless, the all-bound heuristic is appropriate as a default SIPS option, as it is both theoretically and empirically found to not worsen program performance. The major impact of the transformation indicates that query scheduling could be the next big frontier in optimising Datalog program evaluation.

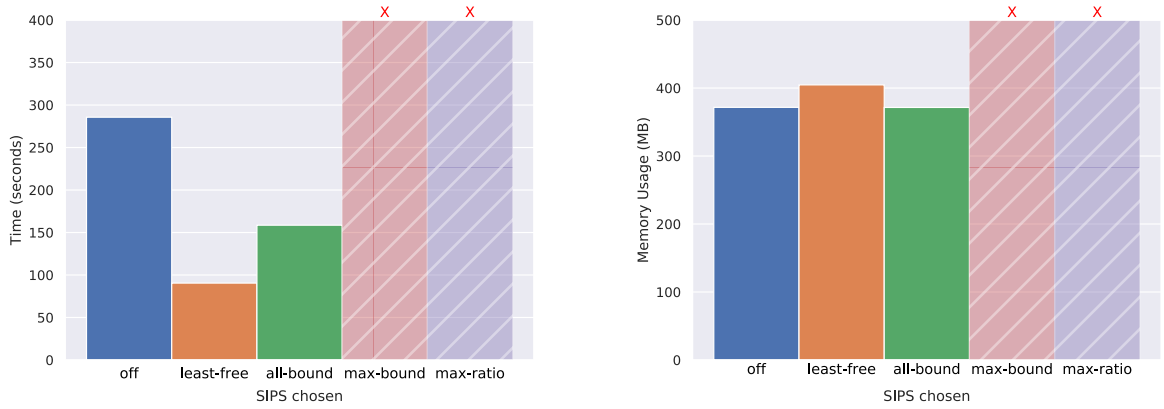
5.2.3 Transformer Synergy

In the synthesised benchmarks section, we found that an interaction between transformers can produce gains greater than the sum of its parts. To observe whether similar effects occur on real-world benchmarks, we will use the sec1 analysis on the Amazon benchmark, N-1075, as a case-study. Recall that the automated transformers did not have an impact on evaluation time or memory. However, both the inlining, shown in Figure 5.15, and reordering, shown in Figure 5.16, semi-automated transformers had a dramatic effect, as discussed earlier.



(A) Evaluation time (B) Memory usage

FIGURE 5.15. SOUFFLÉ performance of sec1 analysis on Amazon N-1075 benchmark with and without selective inlining

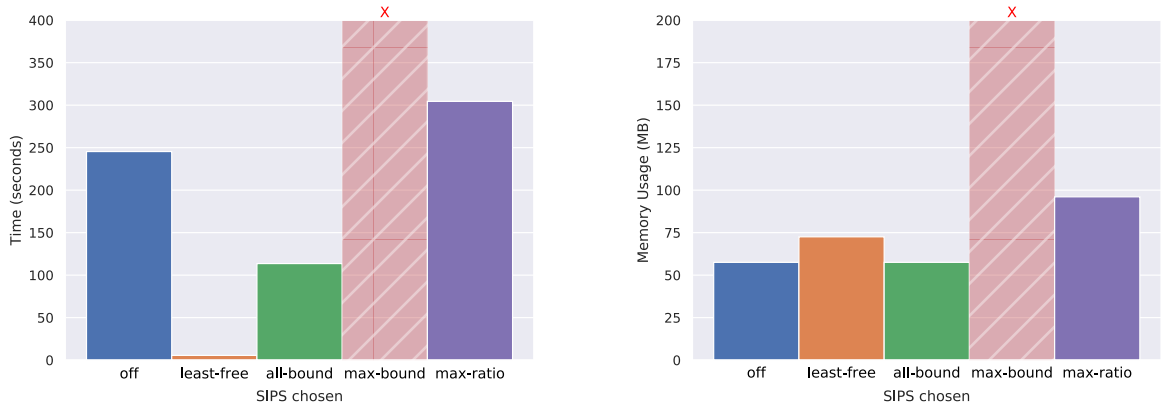


(A) Evaluation time

(B) Memory usage

FIGURE 5.16. SOUFFLÉ performance of sec1 analysis on Amazon N-1075 benchmark with varying SIPS

The original program takes 285 seconds. Inlining alone brings this down to 245 seconds (14% improvement), while reordering alone reduces evaluation time to 90 seconds (68% improvement). Running the automated transformations, even in conjunction with the semi-automated techniques, did not have a notable effect. However, once we run the reordering transformer in addition to the inlining transformer, the result is an enormous improvement in time, as shown in Figure 5.17. In particular, though the all-bound heuristic more than halves evaluation time, the least-free heuristic causes evaluation time to plunge from the original 285 seconds to 5 seconds - a 98% improvement.



(A) Evaluation time

(B) Memory usage

FIGURE 5.17. SOUFFLÉ performance of sec1 analysis on already inlined Amazon N-1075 benchmark with varying SIPS

Table 5.2 summarises the raw results for each combination of transformation. The results for the reordering transformation reflect performance using the best-performing SIPS on this analysis: least-free.

	Transformers Enabled			
	None	**Inlining**	**Reordering**	**Both**
Time (s)	284.77	245.41	90.48	5.56
Memory (MB)	371.67	57.52	404.64	72.54

TABLE 5.2. Effect of each possible combination of semi-automated transformers (sec1 analysis, Amazon N-1075 benchmark, least-free heuristic)

The percentage improvement is shown in Table 5.3.

	Transformers Enabled		
	Inlining	**Reordering**	**Both**
Time (%)	13.82	68.22	98.05
Memory (%)	84.52	-8.87	80.48

TABLE 5.3. Percentage improvement over the original program of each possible combination of semi-automated transformers (sec1 analysis, Amazon N-1075 benchmark, least-free heuristic)

The benefits provided when used in combination is greater than the sum of the individual benefits. The substantial impact of transformer interaction hence remains exceptionally evident, even with real-world benchmarks.

Therefore, although the automated transformations did not have a significant impact, semi-automated optimisations had an incredible effect on both time and memory. The effect was further compounded when the transformations worked in concert: while already impactful on their own, their synergy enabled optimisations greater than the sum of their parts. The lack of existing work on transformer interactions is hence a surprising issue. The performance benefits of a well-designed transformation framework can therefore push the capabilities of SOUFFLÉ even further, regardless of its highly-optimised backend evaluation strategy. More work could be done manually investigating real-world benchmarks to find new avenues in the realm of fully-automated transformations for such programs.

CHAPTER 6

Future Work

The work presented in this thesis presents a broad range of transformers. Each transformer aims to exploit different features of a typical bottom-up evaluation strategy to improve performance. The transformers implemented can be extended to further improve performance, or to have a wider range of applicability. New transformers tackling other inefficiencies in Datalog code could also be introduced. In this section, we discuss potential future work in these areas. The future work is motivated by our study of each transformer and their interdependencies, as well as the experiments and discussion from the previous chapter.

6.1 Extending `InlineRelations`

6.1.1 Choosing Relations to Inline

The `InlineRelations` transformer is manually activated, and the user must make a choice on which relations to inline. The choice is not trivial: the effectiveness of inlining a relation depends on its arity, how often it is used, and so on. As mentioned in the experiments chapter, trial and error is usually not an option, due to the sheer size of Datalog programs in the real-world. Moreover, poor decisions can lead to decreased performance due to blowups in program size or unnecessarily repeated computations.

Developing a set of heuristics for automatically marking relations to inline is therefore of potential benefit. In particular, as the choice is dependent on the size of the relation, and so on, a profile-driven analysis may be beneficial. Profiling data may be derived from a previous run on similarly distributed data. In the imperative world, a profile-guided approach alongside user directives has been shown to yield significant performance gains (Ayers et al., 1997).

6.1.2 Memoisation

The main drawback when inlining relations is the increased potential for repeated computation in certain cases. The problem arises as the tuples computed for the relation are no longer stored in memory, but instead generated on-the-fly, and discarded after use in a rule. However, on-the-fly computation is also the primary benefit of inlining, as relations do not need to be fully computed. A compromise can be found in the form of *memoisation*, similar to the tabling strategies used for top-down evaluation (Chen and Warren, 1996; Tekle and Liu, 2011). Rather than purely generating tuples for an inlined relation on the fly, and then discarding results after use, we can remember where they have been inlined, and keep track of the generated tuples for the relation as we discover them. When a portion of a clause from an inlined relation must be evaluated, we can check whether an appropriate tuple has been generated already. The memoisation technique may improve evaluation time for certain inlined relations by reducing redundant recomputations, at the cost of increased overhead.

6.2 Extending `ReorderLiterals`

6.2.1 Dynamic Query Scheduling

Our experimental results have shown that ordering literals effectively has one of the most profound impacts on real-world Datalog instances. We choose an ordering based on heuristics aimed at minimising the cost of a join. Such heuristics are severely limited by their assumptions on the relationship between a boundness analysis and true data distribution. Note that the true data distribution of the relations in a join can be known at runtime, hence removing the dependency on such assumptions. Therefore, we anticipate that the development of a dynamic query scheduler that reorders atoms in a rule at runtime instead could dramatically improve performance, especially considering the gains already seen with the static scheduler. A dynamic query scheduler is more a backend optimisation, and no longer requires user input.

6.2.2 Profile-Guided Heuristics

An alternative technique for reducing the dependency on a pure boundness analysis for the existing static scheduler is the use of profile data. SOUFFLÉ offers a built-in profiler that offers a range of statistical information for a particular program run. In particular, the size of each relation is recorded.

Such information provides insights into the distribution of data among relations, and may provide a more fine-tuned mechanism for join-cost estimation. The SOUFFLÉ profiler could be extended to also store information on the data dependencies between arguments within a single relation. For example, a binary relation $a(x, y)$ may only contain one value of y for every value x; therefore, given x, y is now fixed. Such statistics could be of further use when determining atom priorities. Therefore, static query scheduling can be improved if profile data on previous runs with similarly distributed input data is taken into account.

6.2.3 Improved Static Query Scheduling

The developed static query scheduler takes a greedy approach. Given a cost-metric, the current least expensive atom is chosen next at each iteration. It may be useful to consider global heuristics that, for instance, maximise the number of bound arguments across the entire rule. Further work can also be done on developing useful heuristics that work well across a wider range of databases, and perhaps determining the most appropriate ordering for a given set of rules based on similar programs in the past.

6.3 Extending `MinimiseProgram`

The program-minimising transformer currently only considers bijectively equivalent clauses. To further the reach of the transformer, relations can also be checked to be bijectively equivalent. That is, for a given pair of relations R and S, there exists a bijectively equivalent clause in S for every clause in R, and vice-versa. In this way, several relations can be merged into one. The extension may be particularly useful when interacting with the `PartitionBodyLiterals` transformer, which may generate several equivalent propositional relations. Extending bijective equivalence to find redundant atoms in clauses while still maintaining decidability in the presence of functors would also be interesting.

The equivalence check can also be extended to eliminate subsumed clauses within relations. For example, consider the following snippet of Datalog code:

```
1   a(x) ← b(x).
2   a(x) ← b(x), c(x).
```

The second rule is subsumed by the first: if a variable `x` satisfies the second, then it must also satisfy the first. Thus, the second rule can be eliminated without consequence, always producing a better program.

From a more technical viewpoint, we limited our scope to clauses without negation or constraints. Negation is likely a simple extension, similar to the current atomic check. Adding symmetric constraints in particular, such as equality, would add a layer of complexity to the equivalence check, however, as the terms on either side can be swapped. Continuing the development of the `MinimiseProgram` could allow it to become a very useful component of the transformation pipeline, eliminating many redundancies introduced by both the original program and other intermediate transformations.

6.4 Developing Further Transformations

From our observations of transformer synergy, it is evident that a broad range of distinct transformers can induce significant benefits in program efficiency. For real-world benchmarks, although semi-automated transformations had a dramatic effect on both evaluation time and memory, fully automated strategies were not as productive. This implies that manually-driven optimisations may be a more interesting endeavour. Transformers that perform well on real-world benchmarks may be developed through a closer manual analysis of the set of rules and the usual distribution of input data to find common inefficiencies. Since Amazon benchmarks are not as carefully hand-optimised as DOOP, Amazon code may have yet undiscovered optimisation opportunities.

CHAPTER 7

Conclusion

Datalog has resurfaced as a domain-specific programming language in many new application domains in recent years. The rise is primarily due to the development of high-performance scalable modern Datalog engines, such as SOUFFLÉ, with well-optimised backend evaluation strategies. Nevertheless, SOUFFLÉ and other Datalog engines in popular use typically have a poor framework for high-level program-rewriting optimisations. Previous work in the literature has highlighted the benefits of such optimisations, whilst also neglecting to discuss the potential benefits of interacting transformations.

In this thesis, we introduce a series of high-level program-rewriting transformations. The transformations are aimed at providing hints to the imperative evaluation mechanism to further optimise the evaluation process. Each transformation is justified by a distinct impact on the produced code, given a non-specific bottom-up evaluation approach. We also developed a novel framework for transformation pipelines on Datalog code to coordinate the effects of each transformation. The transformations and pipeline concept were all implemented into SOUFFLÉ.

The impact of such a transformation framework was demonstrated on a broad range of Datalog programs. The increase in performance on large data-sets for the already scalable SOUFFLÉ Datalog engine was evident on both synthesised and real-world benchmarks. The synthesised benchmarks illustrated the raw potential of the transformers, while the real-world benchmarks reflected the translation of these performance gains when used in practice. In several cases, the transformations enabled an efficient execution of programs that would otherwise time out. The semi-automatic transformations also had a sizeable impact on real-world benchmarks, including an industrial security-analysis tool used by Amazon, reducing time and memory requirements by up to 68% and 85% respectively on terminating programs when transformers are run individually. The benefits were further escalated to a 98% improvement in evaluation time when the optimisations were used in concert, speeding up the performance of the original program by up to 57 times. An even higher degree of improvement was seen for synthesised

benchmarks, where a lack of transformer interaction resulted in programs consistently timing out. The field of transformer interactions is hence an area of untapped potential, and should be further explored.

Possible future work involves the extension of each transformer to be more applicable on Datalog programs. For example, program minimisation could be extended to consider equivalent relations, as well as clauses. Future research could also more closely observe real-world benchmarks and associated profiling information manually to discover the current bottlenecks in evaluation arising due to poorly written Datalog code. The analysis could likely prompt new avenues for transformer development and extension. From our experiments, effective query scheduling is likely the next big step in Datalog program optimisations, especially when performed dynamically at runtime.

The transformation framework we have developed in this thesis has pushed the boundaries of scalable Datalog evaluation to new heights. By extending the work on transformer development, pipeline analysis, and the synergy of transformations, we believe that Datalog will become a more widespread tool from both an academic and industrial standpoint.

Bibliography

Serge Abiteboul, Zoë Abrams, Stefan Haar, and Tova Milo. 2005. Diagnosis of asynchronous discrete event systems: datalog to the rescue! In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 358–367. ACM.

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc.

Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. Porting doop to soufflé: a tale of inter-engine portability for datalog-based analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 25–30. ACM.

Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. 2015. Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1371–1382. ACM.

Andrew Ayers, Richard Schooler, and Robert Gottlieb. 1997. Aggressive inlining. In *ACM SIGPLAN Notices*, volume 32, pages 134–145. ACM.

Isaac Balbin, Graeme S. Port, Kotagiri Ramamohanarao, and Krishnamurthy Meenakshi. 1991. Efficient bottom-up computation of queries on stratified databases. *The Journal of logic programming*, 11(3-4):295–344.

Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. 1985. Magic sets and other strange ways to implement logic programs. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15. ACM.

Francois Bancilhon and Raghu Ramakrishnan. 1988. An amateur's introduction to recursive query processing strategies. In *Readings in Artificial Intelligence and Databases*, pages 376–430. Elsevier.

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. *ACM SIGPLAN Notices*, 44(10):243–262.

Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What you always wanted to know about datalog (and never dared to ask). *IEEE transactions on knowledge and data engineering*, 1(1):146–166.

Pohua P Chang and W-W Hwu. 1989. Inline function expansion for compiling c programs. In *ACM SIGPLAN Notices*, volume 24, pages 246–257. ACM.

Weidong Chen and David S Warren. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM (JACM)*, 43(1):20–74.

S. Greco and C. Molinaro. 2015. *Datalog and Logic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers.

Kryštof Hoder, Nikolaj Bjørner, and Leonardo De Moura. 2011. μz–an efficient engine for fixed points with constraints. In *International Conference on Computer Aided Verification*, pages 457–462. Springer.

Balakrishna R Iyer and Arun N Swami. 1994. Method for optimizing processing of join queries by determining optimal processing order and assigning optimal join methods to each of the join operations. US Patent 5,345,585.

Trevor Jim. 2001. Sd3: A trust management system with certified evaluation. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 106–115. IEEE.

Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: on synthesis of program analyzers. In *International Conference on Computer Aided Verification*, pages 422–430. Springer.

Yanhong A Liu and Scott D Stoller. 2009. From datalog rules to efficient programs with time and space guarantees. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(6):21.

Boon Thau Loo, Tyson Condie, Joseph M Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. 2005a. Implementing declarative overlays. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 75–90. ACM.

Boon Thau Loo, Joseph M Hellerstein, Ion Stoica, and Raghu Ramakrishnan. 2005b. Declarative routing: extensible routing with declarative queries. In *ACM SIGCOMM Computer Communication Review*, volume 35, pages 289–300. ACM.

William R Marczak, Shan Shan Huang, Martin Bravenboer, Micah Sherr, Boon Thau Loo, and Molham Aref. 2010. Secureblox: customizable secure distributed data processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 723–734. ACM.

Alberto Martelli and Ugo Montanari. 1982. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282.

David McAllester. 1999. On the complexity analysis of static analyses. In *International Static Analysis Symposium*, pages 312–329. Springer.

Jack Minker. 2014. *Foundations of deductive databases and logic programming*. Morgan Kaufmann.

Erich J Neuhold, Michael Stonebraker, et al. 1988. *Future directions in DBMS research*. International Computer Science Institute.

Raghu Ramakrishnan, Catriel Beeri, and Ravi Krishnamurthy. 1988. Optimizing existential datalog queries. In *Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 89–102. ACM.

Raghu Ramakrishnan, Divesh Srivastava, S Sudarshan, and Praveen Seshadri. 1994. The coral deductive system. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 3(2):161–210.

Yehoshua Sagiv. 1988. Optimizing datalog programs. In *Foundations of deductive databases and logic programming*, pages 659–698. Elsevier.

Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 196–206. ACM.

Damien Sereni, Pavel Avgustinov, and Oege De Moor. 2008. Adding magic to an optimising datalog compiler. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 553–566. ACM.

Oded Shmueli. 1993. Equivalence of datalog queries is undecidable. *The Journal of Logic Programming*, 15(3):231–241.

Yannis Smaragdakis and Martin Bravenboer. 2011. Using datalog for fast and easy program analysis. In *Datalog Reloaded*, pages 245–251. Springer.

soufflé. 2018. The soufflé project. `https://github.com/souffle-lang/souffle`.

Arun Swami. 1989. Optimization of large join queries: combining heuristics and combinatorial techniques. In *ACM SIGMOD Record*, volume 18, pages 367–376. ACM.

Arun Swami and Anoop Gupta. 1988. *Optimization of large join queries*, volume 17. ACM.

K Tuncay Tekle and Yanhong A Liu. 2011. More efficient datalog queries: subsumptive tabling beats magic sets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 661–672. ACM.

Jeffrey D Ullman. 1989. Bottom-up beats top-down for datalog. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 140–149. ACM.

John Whaley. 2006. bddbddb {P} roject.

John Whaley, Dzintars Avots, Michael Carbin, and Monica S Lam. 2005. Using datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems*, pages 97–118. Springer.