Furthering Datalog

in the pursuit of program analysis

Joseph Owen Sutherland Isaacs

Sidney Sussex

This thesis is submitted on July, 2023 for the degree of Doctor of Philosophy
Declaration

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. It is not substantially the same as any that I have submitted, or am concurrently submitting, for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or is being concurrently submitted, for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. This dissertation does not exceed the prescribed limit of 60,000 words.

Joseph Owen Sutherland Isaacs
Wednesday 19th July, 2023
Abstract

This thesis focuses on specifying compiler analyses in a declarative language, namely Datalog. This work contains three contributions: expanding the number of programs Datalog, specifically SOUFFLÉ (a superset of Datalog), can support; providing a declarative implementation of a well-known analysis; and introducing a new program analysis.

First, a new construct is introduced into SOUFFLÉ allowing consistent negation of relations used to infer facts about that same negated relation. This can be seen as a join point in inference when a rule partially pauses inference until the negated relation is in a consistent state, and it is safe to evaluate the negation. SOUFFLÉ, with consistent negation, is then used to specify a natural encoding of a depth-first graph exploration using a visited set and the sum of affine multivariable polynomials with partially disjoint variable domains. The technique allows previously difficult-to-implement problems to be implemented with asymptotically better performance, in a more natural and expressive fashion.

Second, a declarative implementation of global value numbering (GVN) is presented—a common program analysis to find equivalent program variables—called DGVN. This implementation breaks a data dependence throughout the whole GVN algorithm that previously limited its parallelism. The SOUFFLÉ implementation also makes use of the previously described new negation-aiding construct to increase performance and to allow a natural expression of the algorithm. DGVN is complete enough to be evaluated on benchmarks from SPECint2006.

Third, a definition of pairwise program commutativity is presented, useful to automatic parallelising compilers that transform data dependency graphs. This definition is then lifted into the set domain, allowing for interface-boundary respecting commutativity specifications using semantic program properties, as source-code inline annotations. This system is then used to encode commutative functions from libc and other common data structures and employed to parallelise several benchmark applications. The whole system, including the annotations, is implemented in SOUFFLÉ. The analysis allows transformation of the evaluated programs, leading to between a 1.5x and 3x performance increase (decrease in runtime).
I want to thank my primary supervisor Timothy M. Jones for working with me over the years of my research and supporting me through the writing of this thesis. I want to thank Alan Mycroft for providing invaluable feedback. This research would not have been possible without funding from the EPSRC and ARM® and Ali Zaidi (Arm) who provided insight into the applicability of this research in industry.

I want to thank my parents for their tireless work helping me review and edit the thesis.

I want to thank my partner for understanding the sacrifices made through the pursuit of this thesis.
# Contents

1 Introduction .................................................. 25
  1.1 Hypothesis .................................................. 26
  1.2 Datalog ..................................................... 26
  1.3 Contributions .............................................. 27

2 Background .................................................. 29
  2.1 Compiler components ..................................... 29
    2.1.1 Program intermediate representation (IR) .......... 29
    2.1.2 Compiler optimisation ................................ 30
  2.2 Declarative languages .................................... 31
    2.2.1 Datalog ................................................ 31
      2.2.1.1 Comparison of Datalog to Prolog ............... 32
      2.2.1.2 Herbrand interpretation .......................... 34
      2.2.1.3 Bottom-up Datalog evaluation .................... 34
      2.2.1.4 Semi-naïve evaluation ............................ 35
      2.2.1.5 Semi-positive Datalog ............................. 36
      2.2.1.6 Precedence graph ................................ 37
      2.2.1.7 Stratification ...................................... 37
      2.2.1.8 Forward chaining semantics of Datalog .......... 38
      2.2.1.9 Perfect models and local stratification ......... 38
    2.2.2 Soufflé ................................................. 40
      2.2.2.1 Language overview ................................. 40
      2.2.2.2 Relational algebra machine overview ............ 42
      2.2.2.3 Semi-naïve evaluation in Soufflé ............... 43
  2.3 Global value numbering overview ....................... 44

3 Related work ................................................ 49
  3.1 Datalog and Soufflé ....................................... 49
  3.2 Global value numbering ................................... 51
  3.3 Programming approaches for parallelisation ............ 53
# Dynamic stratification in **SOUFLÉ**

4.1 Motivating example .................................................. 56
4.2 Semantics ............................................................... 60
   4.2.1 Language extensions ........................................... 60
      4.2.1.1 Stabilising relations to enforce safety ............. 62
4.3 Formal Semantics ................................................... 62
   4.3.1 Dynamically stratified Datalog .............................. 64
   4.3.2 Stable semantics in the presence of cyclic negation .... 66
4.4 Implementation ....................................................... 67
4.5 Evaluation ............................................................. 68
   4.5.1 Polynomial expression program ............................. 68
   4.5.2 Graph search with visited set .............................. 71
4.6 Summary ............................................................... 77

# Declarative global value numbering

5.1 Motivating example .................................................. 80
5.2 Implementation in **SOUFLÉ**\textsuperscript{stable} .................. 83
   5.2.1 Analysis relations ............................................ 83
   5.2.2 Value-number encoding and numbering ...................... 83
   5.2.3 Available value-number intersection ....................... 85
   5.2.4 Handling cycles in the control flow graph ............... 86
   5.2.5 VPF .............................................................. 88
   5.2.6 Redundancies ................................................... 93
5.3 Evaluation ............................................................. 94
   5.3.1 Equivalences found by DGVN and missed by LGVN .......... 96
   5.3.2 Execution time .................................................. 97
5.4 Summary ............................................................... 98

# Semantic set annotation analysis

6.1 Motivating example .................................................. 102
6.2 Implementation in **SOUFLÉ** ....................................... 104
   6.2.1 Key concepts ................................................... 105
      6.2.1.1 Syntactic analysis variables ........................... 105
      6.2.1.2 Abstract memory locations ............................. 106
      6.2.1.3 Call-graph-sensitive context ........................... 106
      6.2.1.4 Loop-iteration context ................................... 107
      6.2.1.5 Contextual constructs ................................... 107
      6.2.1.6 Lifting binary relations ................................. 107
      6.2.1.7 Commutativity as set membership ....................... 108
6.2.1.8 Function effects ........................................... 108
6.2.2 Set annotation language ..................................... 109
   6.2.2.1 Embedded annotation clause generation .............. 113
   6.2.2.2 Built-in sets ............................................ 113
6.3 SEMSET analysis inference ..................................... 114
   6.3.1 Building a dependency graph ............................... 114
   6.3.2 Semantic-set-membership propagation ..................... 117
   6.3.3 Checking program dependency graph edges ................ 118
6.4 Case studies ..................................................... 118
   6.4.1 Md5sum ..................................................... 119
      6.4.1.1 Inference result ...................................... 122
      6.4.1.2 Transformation ........................................ 122
   6.4.2 Dedup ..................................................... 125
      6.4.2.1 Inference result ...................................... 128
      6.4.2.2 Transformation ........................................ 129
   6.4.3 Potrace .................................................... 130
      6.4.3.1 Inference result ...................................... 130
      6.4.3.2 Transformation ........................................ 130
6.5 Evaluation ....................................................... 132
   6.5.1 Evaluation environment .................................... 136
6.6 Summary ........................................................ 136

7 Conclusion ........................................................ 137
   7.1 Future work .................................................... 138

References .......................................................... 139

A RAM code for the polynomial summation program .............. 145

B Passes run before GVN by LLVM ................................. 147
List of Tables

4.1 Runtime of depth first search (DFS) programs ................. 75
5.1 Shows the size of each program in the evaluation ............... 94
5.2 GVN partition size evaluation ................................... 94
5.3 GVN runtime comparison ........................................... 98
6.1 Annotations used in case study ................................... 133
6.2 Count of the annotations used in the evaluation ............... 133
6.3 Dependencies found in evaluation .................................. 134
6.4 Program speed-up in evaluation ................................... 134
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Edge reachability program in Datalog</td>
<td>32</td>
</tr>
<tr>
<td>2.2</td>
<td>Naïve edge reachability program in Datalog</td>
<td>35</td>
</tr>
<tr>
<td>2.3</td>
<td>Precedence graph of edge reachability program</td>
<td>37</td>
</tr>
<tr>
<td>2.4</td>
<td>An unsatisfiable Datalog program, which has perfect model semantics.</td>
<td>39</td>
</tr>
<tr>
<td>2.5</td>
<td>Precedence graph of edge reachability program</td>
<td>40</td>
</tr>
<tr>
<td>2.6</td>
<td>The architecture of the Soufflé compiler</td>
<td>40</td>
</tr>
<tr>
<td>2.7</td>
<td>The constructs in a Soufflé program</td>
<td>41</td>
</tr>
<tr>
<td>2.8</td>
<td>The relational algebra machine (RAM) language grammar</td>
<td>43</td>
</tr>
<tr>
<td>2.9</td>
<td>Semi-naïve evaluation of the edge-reachability program, duplicated from section 2.2.1.4.</td>
<td>44</td>
</tr>
<tr>
<td>2.10</td>
<td>The RAM code generated for the naïve edge reachability program</td>
<td>45</td>
</tr>
<tr>
<td>2.11</td>
<td>odd-even Datalog program</td>
<td>46</td>
</tr>
<tr>
<td>2.12</td>
<td>Loop statement of the odd-even program</td>
<td>46</td>
</tr>
<tr>
<td>2.13</td>
<td>The data-flow equation to find the equivalence classes of a program</td>
<td>47</td>
</tr>
<tr>
<td>2.14</td>
<td>An example program and the associated equivalence classes</td>
<td>48</td>
</tr>
<tr>
<td>3.1</td>
<td>An example program and the associated equivalence classes</td>
<td>52</td>
</tr>
<tr>
<td>4.1</td>
<td>Set C specified by set difference in Datalog</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>A possible pure-Datalog implementation of the polynomial summation problem 2.</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>Polynomial expression rules</td>
<td>59</td>
</tr>
<tr>
<td>4.4</td>
<td>stable declaration syntax</td>
<td>60</td>
</tr>
<tr>
<td>4.5</td>
<td>Set Difference in Soufflé using a stable companion relation</td>
<td>62</td>
</tr>
<tr>
<td>4.6</td>
<td>Dynamic precedence graph (DPG) node definition</td>
<td>65</td>
</tr>
<tr>
<td>4.7</td>
<td>The edges in a DPG</td>
<td>65</td>
</tr>
<tr>
<td>4.8</td>
<td>Simplified dynamic precedence graph (DPG) definition</td>
<td>66</td>
</tr>
<tr>
<td>4.9</td>
<td>The Soufflé program that encodes problem 3</td>
<td>67</td>
</tr>
<tr>
<td>4.10</td>
<td>Polynomial sum using Soufflé*stable</td>
<td>69</td>
</tr>
<tr>
<td>4.11</td>
<td>Precedence graph (PG) of polynomial sum program</td>
<td>69</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.10</td>
<td>The contents of the annotation-only header file <code>_anno_fs.h</code></td>
<td>121</td>
</tr>
<tr>
<td>6.11</td>
<td>The annotations attached to relevant functions in the <code>stdio.h</code> header file</td>
<td>123</td>
</tr>
<tr>
<td>6.12</td>
<td>The annotations attached to the <code>fcntl.h</code> header file</td>
<td>124</td>
</tr>
<tr>
<td>6.13</td>
<td>Annotations that are attached to functions in the <code>string.h</code> header file</td>
<td>124</td>
</tr>
<tr>
<td>6.14</td>
<td>The DEDUP program’s main loop, extracted from the <code>Decode</code> function</td>
<td>126</td>
</tr>
<tr>
<td>6.15</td>
<td>Annotations attached to the <code>stdlib.h</code> header file</td>
<td>127</td>
</tr>
<tr>
<td>6.16</td>
<td>The annotations attached to the annotation-only header file <code>_anno unordered_exit.h</code></td>
<td>127</td>
</tr>
<tr>
<td>6.17</td>
<td>Annotations applied to functions in the header files <code>openssl/sha.h</code> and</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td><code>zlib.h</code></td>
<td></td>
</tr>
<tr>
<td>6.18</td>
<td>The ad-hoc annotations included in DEDUP</td>
<td>129</td>
</tr>
<tr>
<td>6.19</td>
<td>The POTRACE main loop</td>
<td>131</td>
</tr>
<tr>
<td>6.20</td>
<td>The annotations attached to functions in the <code>math.h</code> header file</td>
<td>131</td>
</tr>
<tr>
<td>6.21</td>
<td>Annotations attached to the function-call sites used in POTRACE</td>
<td>132</td>
</tr>
<tr>
<td>6.22</td>
<td>The speed-up of programs relative to their sequential performance</td>
<td>135</td>
</tr>
</tbody>
</table>
Glossary

abstract syntax tree  the initial structure that a compiler generates in its parsing stage.

ADT  algebraic data type.

algebraic data type  a data type, common to functional and declarative programming languages, which represents one of multiple tagged compound data types.

AST  abstract syntax tree.

basic block  a grouping of instructions of which the final instruction is the only branch instruction.

CFG  control flow graph.

clang  the C/C++ compiler which is part of the LLVM project.

common subexpression elimination  a compiler transformation used to remove expressions which have been previously computed and are currently available.

control flow graph  a graph, specific to a procedure $p$, with nodes being basic blocks of $p$ and edges existing from basic blocks with a branch instruction to any target basic block from that instruction.

CSE  common subexpression elimination.

Datalog  a declarative programming language, considered a subset of Prolog, where rules are specified as horn clauses; the language is used for database queries and program analysis.

DCE  dead code elimination.

dead-code elimination  an optimisation procedure aiming to remove unnecessary computation and reduce outputted code size by removing instructions which write to a register that is not later read from.
**declarative global value numbering** the name of the GVN implementation presented in this thesis.

**depth first search** a graph search scheme where exploration occurs as far as possible along a single branch before backtracking to search other branches.

**DFS** depth first search.

**DGVN** declarative global value numbering.

**domain specific language** a programming language suited to a specific task, which it usually excels at.

**DPG** dynamic precedence graph.

**DSL** domain specific language.

**dynamic precedence graph** the data structure used to reason about the correctness of programs containing stable-companion guarded negations—constructed from program traces.

**EDB** extensional database.

**extensional database** facts present before program inference begins.

**fixpoint** when applying the same function successively to an input, a fixed point is found once another application of the function does not change output.

**global value numbering** an algorithm tasked with finding redundancies between variables in a procedure.

**GVN** global value numbering.

**horn clause** a clause that is a disjunction of predicate symbols with at most one positive literal.

**IDB** intensional database.

**intensional database** facts in a relation which are inferred throughout the program’s execution.

**intermediate representation** a language use by a compiler or virtual machine to represent a program.

**IR** intermediate representation.
**k** key stable a relation \( R \), with a stable-companion relation \( R_c(k) \), cannot infer new facts with the key \( k \).

least model the smallest complete solution to a Datalog program.

LLVM low level virtual machine.

**Low Level Virtual Machine** a compiler library and a virtual machine, which is targeted by the IR used by this compiler.

**magic-set transformation** a Datalog program transformation to reduce the number of unnecessary intermediate facts inferred.

**OFD** open file descriptor.

open file descriptor a handle used to access a file which the operating system has marked as open, hence storing the file offset; multiple different open file descriptors might exist for the same file on disk.

**PDG** program dependency graph.

**PG** precedence graph.

precedence graph a labelled graph whose nodes are the IDB relations of a Datalog program, and edges are between relations in the body of a rule and the relations at its head; edges can be positive or negative, depending on whether the body relation is in positive or negative form.

program dependency graph a directed graph with nodes as instructions and edges representing a partial execution ordering that must be satisfied to guarantee program safety.

**Prolog** a declarative logic programming language where rules are specified as horn clauses; the language uses a top-down evaluation schema with backtracking.

**RAM** relational algebra machine.

relation series of attribute-typed elements; the type constrains the element to a specific data domain.

**relational algebra** an algebraic theory used to both model data and define queries on that data.
relational algebra machine  the IR used in the SOUFLÉ compiler, inspired by relational algebra.

rule  a horn clause that has one predicate (or more) as a head, and one or more literals in the body.

scalar evolution an algorithm which finds closed mathematical expressions of scalar expressions.

SCC  strongly connected component.

SE  scalar evolution.

semi-naïve evaluation an efficient evaluation scheme for Datalog where updates to relations are captured in Delta (Δ) relations which are used to produce later updates reducing redundant computation.

semi-positive Datalog a form of Datalog where only EDB relations can be negated.

SEMSET  the inference and annotation system presented in this thesis.

SOUFLÉ  a dialect of Datalog, used extensively in this thesis.

SOUFLÉstable  the SOUFLÉ language with the stable-companion set extension presented in this thesis.

SPECINT2006 a benchmarking suite common in industry.

SSA  static single assignment.

stable-companion relation  a relation, attached to another relation $R$, used to inform the runtime that $R$ has a stable key, with an unchanging set of values.

static single-assignment  a language usually in TAC form where a register cannot be reassigned to; a $\phi$-instruction type is introduced to allow a variable to be the choice of many predecessor variables.

stratifiable  a Datalog program that has a stratification.

stratifiable negation constraint  a constraint, on a Datalog program, that there are no cycles in the program’s precedence graph with a negative edge.

stratification  a stratification mapping for a program which follows a series of properties dependent on that program.

stratification level  the natural number assigned to a relation stratification mapping.
stratification mapping  a mapping from relations to a finite continuous series of natural numbers representing the stratification levels and satisfying the stratification constraints for a specific Datalog program partition.

stratum  a partition of a larger Datalog program satisfying the stratification constraints in relation to a stratification mapping and the other strata making up the larger program.

strongly connected component  an element contained in a set of nodes in a directed graph where each node is reachable from all other nodes.

TAC  three address code.

three-address code  an instruction form using three or fewer operands to compute a simple expression; when multiple codes are combined, in sequence, more complex expressions can be represented.

Turing-complete  a system of computation which can simulate a Turing machine.

value $\phi$-function  the data structure used to find possible redundancies in previous basic blocks used in the GVN algorithm.

variable liveness  a variable is live at a program point if that variable is later used in that program.

VPF  value phi function.
CHAPTER 1

Introduction

Compilers are tasked with transforming programs written in a source language into a target language where both languages have properties favourable to different parties. Compilers try to optimise the input program by applying transformations to enhance characteristics better suited to the target system, while preserving program semantics. These optimising transformations can be unsafe in isolation and require program analyses to provide safety guarantees. This makes program analysis crucial to optimising compilers, specifically that the analysis must be sound and as complete as can be reasonably achieved. A sound analysis only allows program transformations that do not alter the intended semantics of the program, while a more complete and precise analysis provides greater opportunities for safe transformations.

A compiler optimisation pass comprises a transformation along with a set of analyses, where each pass is tasked with performing a usually small but well-defined task. Take, for example, an optimisation such as dead code elimination (DCE). DCE tries to remove operations that write to unused registers; the analysis to verify that this transformation is sound is an assertion that the register assigned to by the operation is never read from again. This property, variable liveness, is calculated by tracing the variables used, and the variables assigned to at each line of a program. Variable liveness is specified in terms of logical statements, which are connected to capture these properties throughout a program, and this is true of many other useful program analyses.

The programming language focused on in this thesis is Datalog—considered part of the declarative programming paradigm.\(^1\) This programming paradigm focuses on writing a series of logical statements about a problem domain. These are called clauses and can be in the form of facts or rules. Rules use the currently known facts to infer new facts in an iterative process until no new facts can be deduced, and a least model is found. There is a large gap between how a program is written in an imperative language and Datalog; the former is concerned with the how and what, whereas the latter is only concerned

\(^1\)Datalog is sometimes referred to as being part of the logical programming paradigm.
with what the program must achieve. This means that imperative languages must specify exactly how to enforce each rule, whereas in a logical language the rules can merely be stated.

Analyses are usually written in an imperative paradigm, which can hide the logical nature of the underlying analysis. Necessary performance optimisations can further detract from the clarity of the analysis and possibly allow for bugs to be introduced, as noted by Knuth [33]. Many program analyses are specified as data-flow equations that match up well with a logical programming paradigm. An imperative approach to finding a solution to a set of data-flow equations would be using a fixed-point iteration algorithm that necessarily chooses a fixed search strategy, whereas a logical programming paradigm is free to choose the best search strategy considering the specific set of data-flow equations.

Program analyses take a much larger amount of total engineering effort than transformations in modern compilers and are usually more complex. One analysis (A) can be used by many compiler transformations and even other analyses (B). This means an increase in precision of analysis A can be amplified many times over by allowing more applications of program transformations and more precise B analysis results for the users of the improved analysis A.

1.1 Hypothesis

This thesis explores the thesis that Datalog allows declarative formulations of compiler analyses. Initially program analysis is commonly devised as a series of mathematical equations, which is amenable to proving correctness guarantees, then this analysis is non-trivially transformed into an algorithm that tries to compute an efficient solution to the already defined equations. This creates a greater amount of work for the analysis creator and introduces the issue of ensuring the algorithm implemented simulates the equations devised. This thesis will explore the practicality of implementing these compiler analyses in a language that is as close as possible to the mathematical equations used to initially derive the analysis. This will reduce the work required to implement these analyses since the gap between the mathematical equation and code implementation is smaller.

1.2 Datalog

The declarative programming language used throughout this thesis is SOUFLÉ, a Datalog superset. The extensions to Datalog in SOUFLÉ include arithmetic expressions and unbounded recursive data types, which both remove the termination guarantees in Datalog. SOUFLÉ uses an evaluation technique called semi-naïve evaluation and can produce an
optimised binary by translating the clauses in the program to a compiled programming language and also by applying transformations borrowed from database-query optimisation literature. The SOUFLÉ runtime has enough freedom in how it chooses to find a solution to the program to allow the runtime to choose an efficient-evaluation system, such as a high-performance parallel-execution engine.

Negation and aggregation of relations in a clause were initially not present in Datalog and were added as a series of extensions: semi-positive Datalog and stratifiable negation. Semi-positive Datalog allows negation of input clauses, stratifiable negation improves on this by breaking up a program into statically assigned strata and allowing negation of relations that are part of a lower stratum. Perfect model semantics [16], an extension without a practical implementation, allows negation of facts in a relation, even in the same statically assigned stratum, so long as the order that facts are considered for inference are non-cyclic.

This thesis presents a practical Datalog system, named SOUFLÉ\textsuperscript{stable}, where relations can be examined in negated form once parts of them are stabilised (and cannot be altered)—ensuring a perfect model. The assertion of being stabilised can alter the semantics of a program stopping unwanted inference, which can prove very useful, as is discussed in chapter 4. SOUFLÉ\textsuperscript{stable} can overcome the limitations of stratifiable negation efficiently and safely. The stable-companion approach, defined shortly, allows a small isolated section of the program to ‘break-out’ of the stratification constraints otherwise imposed while not having the larger runtime overhead present from the whole-program dependency tracking, in the technique developed by Ross [48].

1.3 Contributions

This thesis has three contributions. Firstly, an extension to a declarative programming language named SOUFLÉ is presented (called SOUFLÉ\textsuperscript{stable}), which can allow more succinct implementations of common compiler analyses such as graph search with a visited set or scalar evolution (SE). SOUFLÉ\textsuperscript{stable} allows the programmer to guarantee to the SOUFLÉ runtime that a section of computation is finished; this happens once all the facts of a given form have been inferred. The fact’s form is a specific pattern match of the facts in a relation. Once a section of computation is finished the results of that section can be examined (e.g. with negation) in a way that would be unsafe if the computation of that section had not yet finished. This is performed through a new language construct called a stable-companion relation associated with an existing relation. The stable companion acts to safely restrict cyclic inference with negation/aggregation to stabilised parts of the existing relation. The stable-companion relation can lead to asymptotically better performance when used to implement common programs whilst also increasing their clarity.
Secondly, a novel declarative implementation of global value numbering is developed, namely DGVN, which is adapted from existing works by Gulwani et al. and Pai et al. [29, 38]. The algorithm uses the stable-companion relation construct presented in the previous chapter, in conjunction with tracking the fully computed basic blocks and examining them in stable negated form, when required, in successive basic blocks. DGVN is implemented using SOUFFLÉ\textsuperscript{stable}, which allows a performant and easy-to-comprehend program when compared to induction over an ordered domain\textsuperscript{2}. The value numbers are allocated without in-expressible side effects, which is crucial for a Datalog implementation. This is enabled by using the equivalence of ADTs instead of a side-effecting counter for value number generation. The implementation relaxes the order in which program-variable equivalence partitions are considered from a total ordering to a partial ordering. The implementation is compared to LLVM’s GVN implementation using the benchmarks from SPECint2006.

Finally, a new program analysis to find commutativity between program instructions is presented. Commutativity can enable transformations to better exploit parallel hardware. A known happens-before dependency on a pair of instructions can be reversed if found to be commutative, which leads to a greater choice of instruction-execution orders and to transformations that create less-constrained safe parallel programs. The analysis operates on data dependencies (edges in a data dependency graph) attempting to mark edges as commutative, then merging commutativity of pairwise data dependencies to find pairwise instruction commutativity. The commutativity is inferred conditionally on domain-specific reusable program properties, declared by a programmer in the form of annotations. The whole inference engine and annotation language to enable this is written in SOUFFLÉ. The analysis is used to transform existing programs leading to notable performance increase.

\textsuperscript{2}Ordered domains can be complex for a programmer to specify and can lead to inefficient programs which must sequentially check if a property holds for each element of the set before it can be concluded for hold for the whole set.
CHAPTER 2

Background

This chapter provides an overview of the core concepts that this thesis is built on. There is an overview of an optimising compiler (section 2.1), including the intermediate representation (IR) they operate on, and the analyses and transformations of which they are comprised. The Datalog language section 2.2.1 and semantics are presented and compared to Prolog, along with the approaches taken to introduce negation in Datalog. Section 2.2.2 is completed by a closer look at SOUFFLÉ—a Datalog superset used extensively in this thesis.

2.1 Compiler components

Compilers are comprised of multiple passes: a parsing stage, transformation of parsed grammar to an IR, optimisation of the IR and finally transformation of the IR to the target language. The IR and optimisations are of special interest in this thesis.

2.1.1 Program intermediate representation (IR)

In the process of compiling a program from a source language to a target language modern compilers usually make use of an IR. This provides a multitude of benefits including having a language amenable to analysis and transformation, allowing for an m-to-n connection between many input languages and many output languages, and providing a simple, easily reasoned and well-defined language to aid compiler correctness.

While this thesis focuses on LLVM IR, there are many other IRs used in other compilers, including: gcc’s GIMPLE [27] or JVM bytecode [36]. GIMPLE and LLVM IR are ostensibly three address codes (TACs), however there are instructions with two operands and others with more than three. The idea of TACs is that they try to express a more complex computation in terms of multiple simpler expressions using registers as intermediates.

LLVM IR comprises instructions (e.g. \%var1 = add i32 4, \%var) that exist in the
scope of a basic block. A procedure contains many basic blocks; there is an entry basic block for each procedure. Basic blocks themselves contain branch instructions that can be used to derive possible paths of execution through the IR. A simple analysis pass can be used to create a control flow graph (CFG)—a directed graph with basic blocks as nodes and paths (branches, br i1 %cond, label %IfEqual, label %IfUnequal) between basic blocks as edges.

LLVM IR is in static single assignment (SSA) form, meaning that registers are only assigned to once. To deal with the same variable being written to from multiple-predecessor basic blocks a new $\phi$ instruction is introduced that represents the choice of one of many variables, each from different predecessor basic blocks. The variable value at runtime, a copy of one of the predecessor variables, is decided by which predecessor edge is taken when entering the basic block containing the $\phi$ instruction.

Common analyses, part of LLVM, search for useful IR patterns such as dominators and loops. These derivable properties of a program’s IR are used throughout this thesis. A dominator is a control-flow property of two basic blocks $d, n$ computed using a CFG. Block $d$ dominates $n$ if for any path from the entry of the procedure to $n$ the path contains $d$. The immediate dominator $i$ is the dominator $d$ closest to $n$ such that there are no other dominators $d'$ that dominate $n$ and are dominated by $i$. A loop\(^1\) is a set of basic blocks in a strongly connected component (SCC) with a header block that dominates all basic blocks in the loop, meaning the header is accessible from basic blocks in the loop and the basic blocks in the loop are accessible from the header.

### 2.1.2 Compiler optimisation

Compiler optimisations are used to try to modify programs to exhibit desirable properties—for example, a smaller runtime or a smaller binary size. Optimisations are comprised of analyses and transformations, whereby the analyses are used to check that subsequent transformations are safe to apply. Optimisations usually operate on the compiler’s IR. Many optimisations are applied successively to a program and finding an efficient ordering of these transformations can be difficult.

Program analysis can be categorised as either interprocedural or intraprocedural, where the latter is an analysis that considers a single procedure in isolation and the former considers the whole program beyond a procedures’ boundary. Analysis can also focus on control flow or data flow. Control-flow analyses compute properties of the order of program constructs, such as a function call graph or a basic-block dominator graph. Data-flow analyses instead consider properties of values of program variables, for example, global value numbering (GVN), variable-may-alias analysis, or available-expression analysis.

\(^1\)In a reducible CFG.
A common compiler optimisation is to remove duplicated expressions, and replace all instances apart from the first with a copy of that first-evaluated expression. One transformation to achieve this is common subexpression elimination (CSE). This transformation requires input parameters to inform the transformation which expressions are equivalent. The transformation also requires a cost model to decide if the series of register copies will ‘cost less’ (by the metric that this analysis is trying to reduce) than the original computation. The commonly used analysis to detect program equivalence is GVN. GVN uses successive iterations of a data-flow analysis to find partitions of the program variables that are equivalent, described in detail in chapter 5. The cost model used by the CSE transformation will estimate the cost of the original series of expressions against the cost to copy the redundant expressions. The cost model considered, usually and throughout this thesis, is runtime however other metrics such as code size or memory/register use have been considered. The CSE transformation can increase register pressure since post transformation extra registers must be used to store redundant expressions as temporary variables. If too many registers are used then this transformation might need to be undone or the registers, introduced by the CSE transformation might need to be temporarily stored on the call stack (spilled to memory).

2.2 Declarative languages

There are many useful declarative languages such as markup languages (config files, HTML, CSS), query-languages (SQL and Datalog) and general-purpose languages (Prolog). Datalog is of most interest in this thesis and are discussed in this section. Datalog is compared against the more well-known Prolog language.

2.2.1 Datalog

Datalog is a declarative/logic programming language considered to be a subset of Prolog. Prolog has similar syntax but a different evaluation scheme, which is observable in the language. Datalog has been used widely in many fields including as a query language for deductive databases, data integration, information extraction, networking, security, and cloud computing [32, 28]. Datalog has been used to great effect particularly in the area of program analysis [51, 8, 6, 2].

Datalog programs are a series of horn clauses, with variable and ground term tuples $\tilde{X}$:

\[
\forall \tilde{X}_0, \ldots, \tilde{X}_n, \tilde{X} (\neg R_0(\tilde{X}_0) \lor \cdots \lor \neg R_n(\tilde{X}_n) \lor R(\tilde{X})) \equiv \forall \tilde{X}_0, \tilde{X}_n, \tilde{X}. (R_0(\tilde{X}_0) \land \cdots \land R_n(\tilde{X}_n) \Rightarrow R(\tilde{X})).
\]

Each variable in both equivalent clauses is universally quantified implicitly, meaning the
program need not include the universal quantification since it is implicitly included.

Clauses are comprised of a head atom and a possibly empty sequence of literals known as the body. A clause with an empty body is called a fact and all variables must be positively bound in the body.

The Datalog program in figure 2.1 contains two facts (lines 1–2) and two rules (lines 4–5). The two facts include two tuples into the edge relation (0, 1) and (1, 2), which are then included into the path relation at line 4. Then on line 5 the tuple (0, 2) will be included into path since \( x = 0, y = 1, z = 2 \) is a satisfying configuration of the clause variables.

### 2.2.1.1 Comparison of Datalog to Prolog

Datalog and Prolog are two of the most widely used logical programming languages. They have similar syntax, yet different semantics. Datalog finds the set of all solutions to a program whereas Prolog finds a sequence of solutions—unifications of variables, possibly with ground terms. Prolog carries out a depth-first search of the program’s clauses, unifying variables with clauses in the order they exist in the file. Datalog finds a unique least model that can be found using a memoized\(^2\) top-down breadth- or depth-first search or a bottom-up iterative approach—a unique least model is guaranteed to be finite if the program contains a fixed set of ground atoms. This fixed set is guaranteed if no new atoms can be produced; if numbers are allowed in the program then unbounded arithmetic cannot be allowed if termination is to be guaranteed.

The program in figure 2.1 with the goal \(?- \text{path}(X, Y)\) in Prolog would produce the sequence delimited by ‘;’:

\[
X=0,Y=1 \ ; \ X=1,Y=2 \ ; \ X=0,Y=2
\]

whereas Datalog would produce a set of all these results. Here the Prolog sequence in its entirety matches results from Datalog.

\(^2\)Only recording each new fact once.
The difference in semantics becomes more apparent when facts creating a cyclic graph are used:

1. `edge("A", "B").`
2. `edge("B", "A").`

These clauses interpreted, as a Datalog program, would result in the fact set

```
path(A,B) path(B,A) path(A,A) path(B,B)
```

whereas a Prolog program would result in the infinite series:

```
X = "A", Y = "B" ; X = "B", Y = "A" ; X = Y = "A" ; X = Y = "B" ; ...
```

This series is infinite since the Prolog program keep recursing on the second rule (figure 2.1, line 5) with no bound. However, if the program at line 5 instead has the clause

```
path(X, Z) :- edge(X,Y), path(Y,Z).
```

then the Datalog program’s output would be unchanged, but the Prolog program’s resulting series would instead be:

```
X = "A", Y = "B" ; X = "B", Y = "A" ; X = Y = "A" ;
X = "A", Y = "B" ; X = Y = "A" ; ...
```

note that this misses the `X=Y="B"` substitution. This tail of the output is another infinite series of `X = "A", Y = "B" ; X = Y = "A" ;...`. This happens since the greedy DFS evaluation used in Prolog never tries `edge("B", "A")` in the line 5 clause (figure 2.1), instead the search continues finding an infinite number of solution to `edge("A", ?Y)`, namely `Y = "B"; Y = "A"`.

Datalog and Prolog also handle negation differently—Datalog, commonly, uses stratified negation, described in section 2.2.1.7, whereas Prolog uses Negation As Failure (NAF). NAF for a literal `q :- not(p)` must exhaustively find no proof of `p` before `q` can be assumed. NAF is more powerful than Datalog’s stratifiable negation constraint, but can be logically unsound.

1. `member_of("A", "x")`
2. `member_of("B", "y")`
3. `member_of("C", X) :- member_of("A", X), not(member_of("B", X)).`
   // `member_of("B", X) :- member_of("C", X).`
This program is forbidden as it is unstratifiable in Datalog, section 2.2.1.7, but can be expressed in Prolog. The goal, \texttt{?- member_of("C", X)}, finds \texttt{X="x"}, since \texttt{member_of("B", "x")} cannot be proved. If line 5 was included, however, this program would not terminate in Prolog since the runtime would get stuck trying to prove or disprove \texttt{member_of("B", X)}. Chapter 4 explores SOUFFLÉ and enhances it to give a system whereby this program could be solved by a Datalog system—SOUFFLÉstable.

### 2.2.1.2 Herbrand interpretation

Given a Datalog program, the Herbrand Universe is defined as the possibly infinite set of all ground formulae created from the constants which appear in a program \(P\) and from infinite applications of functors applied to all applicable elements already in the universe. A Herbrand Base is a subset of the Herbrand Universe. The extensional database (EDB) (written \(\text{edb}(P)\)), intensional database (IDB) (written \(\text{idb}(P)\)) and schema (written \(\text{sch}(P)\)) are all Herbrand Bases of a program \(P\). The EDB consists of all ground formulae obtained from considering only facts—clauses without a body. The intensional database (IDB) consists of all ground formulae obtained from only considering facts generated from inferences using rules—clauses with a body—starting with the programs EDB. The schema, the union of the EDB and IDB (\(\text{sch}(P) = \text{edb}(P) \cup \text{idb}(P)\)), is the set of all ground formulae derived from the clauses of a program \(P\).

### 2.2.1.3 Bottom-up Datalog evaluation

The operational semantics of Datalog are defined by finding the smallest fixpoint of the immediate consequence operator \(\Gamma_P(I)\), defined shortly. Let \(P\) be a Datalog program, let \(\text{sch}(P)\) be the program’s schema—the set of all possibly inferable facts—then let \(I\) be an instance of the program \(P\)’s schema \(I \subseteq \text{sch}(P)\). A fact \(t\) is an immediate consequence for \(I\) and \(P\): if \(t \in I(R)\), for some EDB relation\(^3\) \(R\) or if it is generated by an instantiation of a rule \(t \leftarrow t_1, \ldots, t_n\) in \(P\) and each \(t_i\) is in \(I\).

\[
\Gamma_P(I) = I \cup \{t \mid t_1, \ldots, t_n\text{ is the instantiation of a clause in } P \text{ and each } t_i \in I\}
\]

The immediate consequence operator \(\Gamma_P(I) \subseteq \mathcal{P}(\text{sch}(P)) \times \mathcal{P}(\text{sch}(P))\) is a function from a subset of \(\text{sch}(P)\) to a subset of \(\text{sch}(P)\), that consists of all the facts that are immediate consequences for \(I\) and \(P\). The consequence operator can be used to build an algorithm (see algorithm 1) to find the least model of a Datalog program \(P\). The algorithm starts with an initial instance \(I\) of the program’s schema \(\text{sch}(P)\) and then iteratively computes the immediate consequence operator \(\Gamma_P(I)\) until the result of the computation is equal.

\(^3\)Where \(I(R)\) is a set of all facts in the relation \(R\) contained in \(I\).

\(^4\)With no clauses in the form of rules.
Algorithm 1 Naive fixpoint algorithm using the consequence operator $\Gamma_P(I)$.

1: procedure Fixpoint($P$, $I$) ▷ The fixpoint of a program $P$ with the EDB $I$
2: while $I \neq \Gamma_P(I)$ do ▷ Check for convergence
3: $I \leftarrow \Gamma_P(I)$ ▷ Compute new updates
4: end while
5: return $I$ ▷ Least fixpoint of $P$
6: end procedure

1 path($x$, $y$) :- edge($x$, $y$).
2 path($x$, $z$) :- path($x$, $y$), path($y$, $z$).

Figure 2.2: A program for computing the transitive closure of the edge relation in a non-standard way. Note, it would be preferable to use path($x$, $z$) :- path($x$, $y$), edge($y$, $z$) on line 2.

to the previous result. Line 2 checks for the fix point condition, and line 3 computes the new updates to the scheme. They both use the immediate consequence operator. This algorithm however is inefficient since it will compute each derivation multiple times, at least once for each iteration after it is first discovered. The algorithm also compute the immediate consequence operator one extra time just to check that no we facts could be inferred.

The semi-naive evaluation algorithm in the next section address these inefficiencies by introducing $\Delta$-relations which all empty signal the algorithms’ completion. These relations also focus the algorithm on considering only newly inferred facts.

2.2.1.4 Semi-naive evaluation

The semi-naive evaluation system is a bottom-up technique used to evaluate Datalog programs. The transitive closure of the edge relation, in figure 2.2, is used as an example program to explain this evaluation scheme.

The semi-naive transformation of a program $P$ requires the definition of two new relation types $R^i$ and $\Delta^i_R$ for each relation $R$. $R^i$ contains all the facts inferred by a relation $R$ up to the $i$th iteration. The relation $\Delta^i_R$ contains all the facts inferred only in the $i$th iteration and no iteration before this. The two relations have the following properties: $i \neq j, \Delta^i \cap \Delta^j = \emptyset$, $R^i = \bigcup_{i \in [1,n]} \Delta^i_R$ and $\forall i.R^i \subseteq R^{i+1}$. The program in figure 2.1 is transformed into semi-naive form:

\[
\begin{align*}
\Delta^1_{\text{path}}(x, y) & :\text{ edge}(x, y). \\
\Delta^{i+1}_{\text{path}}(x, z) & :\Delta_i^\text{path}(x, y), \text{ path}^i(y, z). \\
\Delta^{i+1}_{\text{path}}(x, z) & :\text{ path}^i(x, y), \Delta^i_{\text{path}}(y, z).
\end{align*}
\]
This can be written instead as an update loop, using Datalog rules inference (\(\ :-\)) and imperative assignment (\(\ :=\));

\[
\Delta_{\text{path}}^1(x, y) := \text{edge}(x, y). \tag{2.1}
\]

\[
\text{path}^1 := \Delta_{\text{path}}^1 \tag{2.2}
\]

\[
\text{temp}_{\text{path}}^{i+1}(x, z) := -\Delta_i^i(x, y), \text{path}(y, z). \tag{2.3}
\]

\[
\text{temp}_{\text{path}}^{i+1}(x, z) := \text{path}(x, y), \Delta_i^i(y, z). \tag{2.4}
\]

\[
\Delta_{\text{path}}^{i+1} := \text{temp}_{\text{path}}^{i+1} - \text{path}^i \tag{2.5}
\]

\[
\text{path}^{i+1} := \text{path}^i \cup \Delta_{\text{path}}^{i+1} \tag{2.6}
\]

Equations (2.1) and (2.2) copy the \textit{edge} relation’s facts into the \textit{path} relation. Equations (2.3) and (2.4) compute the new updates for the \textit{path} relation caused by newly inferred \(\Delta\) facts from the previous iteration. Finally, equations (2.5) and (2.6) compute the next iteration’s \(\Delta_{\text{path}}^{i+1}\) and \textit{path}^{i+1} relations. The equations (2.1)–(2.4) are applied for ascending \(i\) values until no new facts are inferred \(\text{path}^i = \text{path}^{i+1}\)—a fix point is found—this happens once all the delta relations \(\Delta^i_R\) are empty and all equations have been applied at least once.

The semi-naïve evaluation scheme can be defined in terms of an extension to the consequence operator, previously defined in section 2.2.1.3. The semi-naïve consequence operator \(\Gamma_P(\Delta, I)\) takes two arguments: \(I \subseteq \text{sch}(P)\) an instance the program’s schema \(\text{sch}(P)\) and an update relation \(\Delta \subseteq \text{sch}(P)\). \(\Delta\) contains all the facts inferred from the latest application of the consequence operator. The operator computes the immediate consequence using only the preceding immediately inferred facts:

\[
\Gamma_P(\Delta, I) = I \cup (\{ t \mid t \leftarrow t_1, \ldots, t_k \text{ is a rule instantiation with each } t_i \in I \land \exists t_j \in \Delta \})
\]

The operator’s fixpoint is found, and that is the least model of a program \(P\). The fixpoint can be computed using algorithm 2. This approach is used in many Datalog systems, including \textsc{Soufflé}.

2.2.1.5 Semi-positive Datalog

Datalog with negation has ill-defined fixpoint semantics [1]. When adding negation to Datalog one approach is semi-positive Datalog. In semi-positive Datalog only EDB relations \((R(x))\) can be in negated form—that is relation which cannot change throughout program evaluation. This is allowed because the negated relation \((\neg R(x))\) is replaced with the
Algorithm 2  Semi-naive fixpoint procedure using the semi-naive consequence operator \( \Gamma_P(\Delta, I) \).

1: procedure FP-SEMI-NAIVE\((\textbf{P}, \textbf{I})\) \hspace{1em} \triangleright \text{The fixpoint of a program } \textbf{P} \text{ with the EDB } \textbf{I} \\
2: \hspace{1em} \Delta \leftarrow \textbf{I} \hspace{1em} \triangleright \text{Check for convergence} \\
3: \hspace{1em} \textbf{while } \Delta \neq \emptyset \textbf{ do} \hspace{1em} \triangleright \text{Compute new updates} \\
4: \hspace{1em} \textbf{I}' \leftarrow \Gamma_P(\Delta, \textbf{I}) \hspace{1em} \triangleright \text{Compute } \Delta \text{ relations} \\
5: \hspace{1em} \Delta \leftarrow \textbf{I}' \setminus \textbf{I} \hspace{1em} \triangleright \text{Compute } \Delta \text{ relations} \\
6: \hspace{1em} \textbf{I} \leftarrow \textbf{I}' \hspace{1em} \triangleright \text{Least fixpoint of } \textbf{P} \\
7: \hspace{1em} \textbf{end while} \\
8: \hspace{1em} \textbf{return } \textbf{I} \\
9: \textbf{end procedure}

\[ \begin{array}{c}
E \hspace{1em} + \rightarrow \hspace{1em} P \\
\end{array} \]

Figure 2.3: Shows the precedence graph (PG) for the program in figure 2.2, where the letters stand for the relations: \( P = \text{path}, E = \text{edge} \).

complemented relation \((\bar{R}(x))\). The complement of a relation in a clause is defined in terms of the domain of the other variables from other literals in that clause that are mentioned in the negated literal. E.g. the domain of the clause body \( R(x, y), \neg R'(x), R''(y, x) \), is \( \{ x \mid \exists y. (x, y) \in R \land (y, x) \in R'' \} \). The evaluation proceeds using the instance of the complemented relation, \( \bar{R}(x) \), in the clause body.

2.2.1.6 Precedence graph

A precedence graph (PG) \( G_P \) can be found for a Datalog program \( P \). The nodes are all relations in \( P \). The edges in \( G_P \) are found by considering each rule:

- \( R : \ldots, R'(v) \) there is a positive edge from \( R' \) to \( R \) ((\( R', R \)).
- \( R : \ldots, \neg R'(v) \) there is a negative edge from \( R' \) to \( R \) ((\( R', R \)).

The PG of the program in figure 2.2 is shown in figure 2.3.

2.2.1.7 Stratification

Semi-positive Datalog can be extended allowing the negation of input relations at a lower stratification level, explained shortly, than the relations currently being evaluated. A (relational-symbol) stratification [1] of a Datalog program \( P \) is a sequence of Datalog programs \( P_1, \ldots, P_n \) such that for some mapping \( \sigma \) from IDB relations to \([0, n] \) satisfying the following properties:

- The programs \( \{P^1, \ldots, P^n\} \) form a disjoint partition of program \( P \).
• Each relation \( R \) has all its defining rules in the same partition \( P^{\sigma(R)} \).

• Rules \( (R(u) :- ..., R'(v), ...) \), with two relations \( R \) and \( R' \) find \( R' \) is in a lower (or equal) stratification level than \( R \) \( \sigma(R') \leq \sigma(R) \).

• Rules \( (R(u) :- ..., !R'(v), ...) \), with two relations \( R \) and \( R' \) find \( R' \) is in a strictly lower stratification level than \( R \) \( \sigma(R') < \sigma(R) \) where \( R' \) is in the body in negative form and \( R \) in the head.

\( P^i \) is called a stratum and \( \sigma \) is called the stratification mapping. Programs that have a stratification are called stratifiable.

**Theorem 1.** A Datalog program \( P \) is stratifiable if its PG \( G_P \) has no cycle containing a negative edge \([1]\).

The program in figure 2.2 will have two following stratification levels, one containing the edge relation and any associated rules and the other containing the path relations and its associated rules.

### 2.2.1.8 Forward chaining semantics of Datalog

The forward chaining semantics \([57]\) of Datalog is a procedural semantics instead of a declarative semantics in section 2.2.1.2, for pure Datalog with no negation these procedural and declarative semantics coincide, however for Datalog with negation the two semantics differ. The forward chaining semantics applies the consequence operator to all clauses in the program at once until a fixpoint is reached. This however can difficult to comprehend programs since the number of steps to infer a fact can affect the inference of another fact is the former is used in negated form in the latter.

### 2.2.1.9 Perfect models and local stratification

Local stratification is a generalisation of stratification, as defined by Ceri et al. \([16]\). Consider the rule \( R \) of the form \( t_0 \leftarrow t_1, ..., t_n \), new ground instances of \( R \) can be found \( t_0 \) by instantiation of all the terms with all already inferred facts \( t_1, ..., t_n \). For a Datalog program \( P \) the ground expansion \( P^o \) of \( P \) is all the ground instances of rules in \( P \).

Each program \( P \) induces a depends-on relation \( \leftrightarrow \) on the program’s schema \( \text{sch}(P) \)—the possibly infinite set of all possible ground terms. Let \( A \) and \( B \) be ground terms \( A, B \in \text{sch}(P) \). Define the depends-on relation \( A \leftrightarrow B \) iff there is a rule \( R \) in \( P^o \) such that \( A \) occurs in positive or negative form in the body of \( R \) and \( B \) is in its head. The depends-on relation is used to define a second relation priority \( (\succ) \), \( A \succ B \) iff there is a finite sequence of depends-on relationships \( (\leftrightarrow) \) holds \( A = N_0 \leftrightarrow N_1 \leftrightarrow N_2, ..., N_{k-1} \leftrightarrow N_k = B \), where \( N_i \in \text{sch}(P) \) and there is at least one \( 0 \leq i < k \) such that \( N_i \) occurs in negated form.
\[ P(X) :- B(X) \\
\[ P(X) :- E(X, Y, Z), \neg P(Z), P(Y) \]

Figure 2.4: An unsatisfiable Datalog program, which has perfect model semantics.

in a rule of \( P^\omega \) and the head is equal to \( N_{i+1} \). This means that the evaluation process should consider \( A \) before \( B \) since a derivation attempt of \( B \) requires knowledge of whether \( A \) (or \( \neg A \)) is derivable from \( P \). If \( A > B \) then \( A \) has a higher priority than \( B \).

Let \( I_1 \) and \( I_2 \) be two schema instances \( I_1, I_2 \subseteq \text{sch}(P) \). \( I_1 \) is preferable to \( I_2 \) iff for each element \( B \in I_1 \setminus I_2 \) there exists an element \( A \in I_2 \setminus I_1 \) such that \( A \) has a higher priority than \( B \), \( A > B \). Intuitively this means that \( I_1 \) is preferable to \( I_2 \) iff \( I_1 \) can be obtained from \( I_2 \) by removing zero or more higher priority ground atoms and by adding zero or more lower priority ground atoms.

Relational-symbol stratification (section 2.2.1.7) of a program is sufficient but not necessary for the existence of a perfect model, defined shortly. If the ordering of stratification is instead defined in terms of ground formulae\(^5\) and if the ground-formulae stratification conditions\(^6\) are satisfied the program is ground-formulae stratifiable (see theorem 1) then the program has a perfect model. A program \( P \) is a unique perfect model iff the program schema can be partitioned into a finite number of schema strata \( S_1, S_2, \ldots, S_n \) such that all ground atoms \( A \) and \( B \) of \( S \), meet the following two conditions:

- If \( A \) depends on \( B \) (\( A \hookrightarrow B \)), then the stratum of \( A \) is at most as high as the stratum of \( B \).

- If \( A \) has a higher priority than \( B \) (\( A > B \)) then \( A \) belongs to a lower stratum than \( B \).

The difference between stratification and local stratification can be seen using the program in figure 2.4. The precedence graph of this program is shown in figure 2.5, and from the negative cycle this graph it can be concluded that there is no possible stratification of the program. The perfect model semantics can however be seen for the following EDB \( \text{edb}(P_r) = B(2), E(6, 3, 2) \). This gives the following relations of ground terms \( P(2) \hookrightarrow B(2), P(6) \hookrightarrow P(2), P(6) \hookrightarrow \neg P(3), P(6) \hookrightarrow E(6, 3, 2), P(6) > B(2), P(6) > \neg P(3) \). This a (non-unique) local stratification \( S_l \) of the program can be constructed \( S_l = \{\{B(2), P(2), E(6, 3, 2), \neg P(3)\}, \{P(6)\}\} \). This shows that \( P(6) \) can be soundly derived from the execution of the program \( P_r \) with \( \text{edb}(P_r) \). The semantics provides a basis for the ideas presented in chapter 4

\(^5\)Elements of the program’s schema \( \text{sch}(P) \).

\(^6\)As compared to the relational-symbol stratification conditions.
Figure 2.5: Shows the precedence graph (PG) for the program in figure 2.2, where the letters stand for the relations: $P = \text{path}$, $E = \text{edge}$.

![Precedence Graph](image)

Figure 2.6: Shows the precedence graph (PG) for the program in figure 2.2, where the letters stand for the relations: $P = \text{path}$, $E = \text{edge}$.

2.2.2 SOUFFLÉ

SOUFFLÉ [50] is a Datalog variant with multiple extensions with unbounded arithmetic, recursive data structures, non-deterministic choice [30], and relation subsumption [34]. These additional features have increased the number of useful and understandable programs expressible in SOUFFLÉ. SOUFFLÉ uses an intermediate representation named RAM to represent the relational algebra that a Datalog program written for SOUFFLÉ is translated to. SOUFFLÉ uses the semi-naïve evaluation technique (outlined in sections 2.2.1.4 and 2.2.2.3) used to translate input from programs from their AST representation to RAM.

The overall structure of the SOUFFLÉ compiler can be seen in figure 2.6. The abstract syntax tree (AST) and RAM transformations apply semantic and performance transformations. The ast2ram pass converts the AST to RAM. The final output can either be the relations inferred by the execution of the program (using the interpreter) or a C++ program (using the c++ synthesiser phase) which can be run to find the inferred relations.

2.2.2.1 Language overview

SOUFFLÉ programs have four top-level constructs: relations, types, clauses and subsumption meta-clauses.

Relation declarations, figure 2.7(a), introduce new relations accessible throughout the
(a) Relation declaration, named $R$, with an attribute tuple $t = x_1 : T_1, \ldots, x_n : T_n$. Attribute comprises a name and an already declared type.

\begin{align*}
\text{.decl } & \quad R \text{ (name)} \\
& \quad \text{attribute tuple} \\
& \quad \text{choice-domain (x_1, x_2)}
\end{align*}

\[ R \quad (\text{attribute tuple}) \]

(b) Subtype, union type and algebraic data type declarations, the two primitive types are \textbf{symbol} and \textbf{number}.

\begin{align*}
\text{.type } & \quad A \quad <: \quad B \\
& \quad \text{supertype type} \\
\text{.type } & \quad A \quad = A_1 \mid \ldots \mid A_n \\
& \quad \text{union list} \\
\text{.type } & \quad A \quad = \quad \{ B \{ x_1 : T_1, \ldots, x_n : T_n \} \mid \ldots \mid \{ C \{ \} \} \} \\
& \quad \text{empty branch}
\end{align*}

(c) The two types of clauses and the different literals: atoms, constraints and negations (of an atom).

\begin{align*}
R(\text{a}, \text{1}). \\
R(x, z) \quad :- \quad R(x, y), \quad z = y + 1, \quad !R1(z).
\end{align*}

(d) The subsumption meta-rule, checked all pairs of tuples.

\begin{align*}
R(x, y) \quad \leftarrow \quad R(x, z) \quad :- \quad y < z.
\end{align*}

Figure 2.7: The constructs in a SOUFFLÉ program.
program. The declarations contain a name, a possibly empty attribute tuple \( t \) (pairs of a tuple-unique name \( x \) and declared type \( T \)) and a possibly empty list of qualifiers. The choice-domain qualifier imposes a functional dependency on the domain \( d = \{ x_1, x_2 \} \) and the co-domain \( c = t \setminus d \). The functional dependency ensures that the facts in the relation are unique up to the domain, meaning the domain’s value uniquely defines the co-domain’s value.

The three type declarations, figure 2.7(b), are: (strict) subset declarations, union types, and ADTs definitions allowing tagged unions which can be pattern-matched at runtime. The subtypes must stem from either of the two primitive types, symbol and number. Clauses (figure 2.7(c)) can either be facts or rules, where facts have no body and rules must have a body. Variables in a clause head must be bound inside the clause body, since all variables in the clause body are implicitly universally quantified. The subsumption meta-rule (figure 2.7(d)) allows the declaration of a subsumption relation \( I \sqsubseteq I' \), defined over a pair of schemas \( I, I' \), a program specific preferability relation. For example if there was a relation \( D(n, l) \) where \( n \) is a node of a graph and \( l \) is the length from a root node then the subsumption relation for finding the shortest distance to \( n \) could be defined as \( D(n, l) \sqsubseteq D(n, l') \leftarrow l' \leq l \). This has the mean that for any fact \( D(n, l) \) if there is a fact \( D(n, l') \) such that \( l' \leq l \). The meta-rule can be used to allow the shortest distance algorithm to be implemented in Datalog and applied to a graph with cycles without the need to apply other cycle detection. This subsumption relation \( \sqsubseteq \) must be monotonic with respect to the consequence operator (\( \Gamma_P(I) \)), see section 2.2.1.3. That means that for a pair of schema instances \( I, I' \) the instances after an application of the consequence operator must also follow the preferability of that program \( I \sqsubseteq I' \implies \Gamma_P(t) \sqsubseteq \Gamma_P(t') \). This monotonicity property is dependent on the specific program and must be asserted by the program writer. Once facts are shown to be subsumed those facts are removed from their relation.

### 2.2.2.2 Relational algebra machine overview

The relevant RAM grammar, first presented by Scholz et al. [50], is presented in figure 2.8 and explained in this section. The AST, some of which is omitted, is split into the following three groups: statements \( s \), operations \( o \) and conditions \( c \). The tuples \( t \), tuple elements \( t.i \) and relations \( R \) also are part of the grammar. The statements each have the following semantics: QUERY used to contain an operation, the LOOP executes the statement \( s \) repeatedly until a corresponding, contained, EXIT statement finds its condition \( c \) true and the loop is exited. The SWAP statement operates on relations with the same attribute domain, swapping the contents of both relations. The CLEAR statement deletes all elements in the relation \( R \).

---

7However the grammar used by SOUFFLÉ today has changed since this work was published.
Some available RAM operations are: scan, filter and insert. The scan operation is used for iterating over each tuple in the relation $R$, capturing the tuple in the variable $t$, and making it accessible to nested operations, similar to a for loop in an imperative programming language. There is also an index scan operation which has the same form as scan but with an index-check appended to the scan operation. This is a more efficient primitive that has the same semantics as a scan followed by a filter. The filter operation IF is used to skip iterating over any $\neg c$ configurations. The INSERT operation creates a new tuple $(t.1, \ldots, t.n)$ from any elements available from parents operations and inserts this into $R$. The condition syntax contains basic-Boolean operations along with a membership check and a relation-is-empty check.

2.2.2.3 Semi-naïve evaluation in SOUFFLE

The program presented in figure 2.2 is used as an example to illustrate the semi-naïve evaluation system expressed in the RAM IR used by SOUFFLE. The update loop, in section 2.2.1.4, is duplicated in figure 2.9.

The RAM program, in figure 2.10, implements the edge-reachability program, in figure 2.2. The recursive relation path is computed in a RAM IR program using three relations: path, @new_path, and @delta_path where, for each step in the loop $i > 1$, $\Delta_{i+1}^{i+1}(x,y) \equiv @new_path(x,y)$ and $\Delta_{i}^{i+1}(x,y) \equiv @delta_path(x,y)$. The queries at lines 2–10 in the RAM program are responsible for equations (2.7) (2.8). The equations (2.9), (2.10) and (2.11) find the $\Delta_{i+1}^{i+1}$ relation for all $i > 1$ the RAM queries, lines 13–28, and

---

Figure 2.8: The RAM language grammar. The grammar contains statements $s$, operations $o$, conditions $c$, tuples $t$, tuple elements $t.i$ and relations $R$. There is a special tuple element named UNDEF matching any element.
\[ \Delta^1_{\text{path}}(x, y) :\text{edge}(x, y). \]  
\[ \text{path}^1 := \Delta^1_{\text{path}} \]  

(2.7)  

(2.8)  

\[ \text{temp}^{i+1}_{\text{path}}(x, z) := \Delta^i_{\text{path}}(x, y), \text{path}(y, z). \]  
\[ \text{temp}^{i+1}_{\text{path}}(x, z) := \text{path}(x, y), \Delta^i_{\text{path}}(y, z). \]  
\[ \Delta^{i+1}_{\text{path}} := \text{temp}^{i+1}_{\text{path}} - \text{path}^i \]  
\[ \text{path}^{i+1} := \text{path}^i \cup \Delta^{i+1}_{\text{path}} \]  

(2.9)  

(2.10)  

(2.11)  

(2.12)  

Figure 2.9: Semi-naïve evaluation of the edge-reachability program, duplicated from section 2.2.1.4.

conduct the same computation adding the results to a relation named @new_path. This relation is cleared at line 30, meaning at every iteration @new_path is the equivalent to \( \Delta^i_{\text{path}} \). The computation of \( \text{temp}^i_{\text{path}} \) and \( \Delta^i_{\text{path}} \) are folded into one query using the IF ((NOT (t0.0, t1.1) IN PATH)... condition in both queries to subtract \( \text{path}^i \). Then the equation (2.12) is simulated by the query on lines 30–33. The swap at the end of the loop, on line 34, moves \( \Delta^{i+1}_{\text{path}} \) relation into @delta_path.

The odd-even program in figure 2.11 is used to illustrate how the semi-naïve evaluation algorithm works in SOUFFLÉ when two relations in the same stratum have mutually recursive clauses. The RAM code generated can be seen in figure 2.12. There are two delta relations @new_odd and @new_even that are successively updated in the same loop. The loop will be entered with the relation @delta_even containing only the fact \( \text{(NUMBER(0))} \) and after each iteration either a new odd or even number will be discovered. This shows that the semi-naïve evaluation scheme independently computes all updates, for a step, for all relations in a stratum inside the loop body.

### 2.3 Global value numbering overview

This section gives an overview and simple data-flow equations to find the global value numbering (GVN) of a program (defined shortly). This section assumes the input program is already in static single assignment (SSA) form. The GVN algorithm tries to assign all the variables \( x, y, \ldots \) in a program to equivalence classes, called (equivalence) partitions, at a specific program point. These classes contain a unique-value number \( v_i \), for some \( i \in \mathbb{N} \) and can also contain variables, value expressions, and binary operations with value-number operands (e.g. \( v_2 + v_6 \) or \( v_3 \times v_5 \)). The value numbers characterise a partition. For example

---

\(^8\)Lines 14–20, in fact, compute \( \text{temp}^{i+1}_{\text{path}}(x, z) := \Delta^i_{\text{path}}(x, y), \text{path}^{i-1}(y, z) \), which is more efficient yet equivalent to equation (2.9).
// path(x,y) :- edge(x,y).
QUERY
  IF (NOT ISEMPTY(edge))
    FOR t0 IN edge
      INSERT (t0.0, t0.1) INTO path
END QUERY
QUERY
  FOR t0 IN path
    INSERT (t0.0, t0.1) INTO @delta_path
END QUERY
LOOP
// path(x,z) :- @delta_path(x,y), path(y,z).
QUERY
  IF (NOT ISEMPTY(@delta_path)) AND (NOT ISEMPTY(path))
    FOR t0 IN @delta_path
      FOR t1 IN path ON INDEX t1.0 = t0.1
        IF (NOT (t0.0,t1.1) IN path) AND
           (NOT (t0.1,t1.1) IN @delta_path)
          INSERT (t0.0, t1.1) INTO @new_path
    END QUERY
// path(x,z) :- path(x,y), @delta_path(y,z).
QUERY
  IF (NOT ISEMPTY(path)) AND (NOT ISEMPTY(@delta_path))
    FOR t0 IN path
      FOR t1 IN @delta_path ON INDEX t1.0 = t0.1
        IF NOT (t0.0,t1.1) IN path
          INSERT (t0.0, t1.1) INTO @new_path
    END QUERY
EXIT ISEMPTY(@new_path)
QUERY
  FOR t0 IN @new_path
    INSERT (t0.0, t0.1) INTO path
END QUERY
SWAP (@delta_path, @new_path)
CLEAR @new_path
END LOOP

Figure 2.10: The RAM code generated for the naïve edge reachability program, in figure 2.2.
.decl odd(n: number)
.decl even(n: number)
.decl request(r: number)

request(range(0,10,1)).
even(0).
odd(n+1) :- request(n), even(n).
even(n+1) :- request(n), odd(n).

Figure 2.11: A mutually recursive odd-even program, for naively checking if an integer is odd or even. The range functor populates the request relation with the values in the set \{0,1,\ldots,10\}.

LOOP
//even((n+1)) :- request(n), @delta_odd(n).
QUERY
   IF (NOT ISEMPTY(request)) AND (NOT ISEMPTY(@delta_odd))
      FOR t0 IN request
         IF ((t0.0) IN @delta_odd AND (NOT ((t0.0+NUMBER(1))) IN even))
            INSERT (t0.0+NUMBER(1)) INTO @new_even
      END QUERY
// odd((n+1)) :- request(n), @delta_even(n).
QUERY
   ....... // similar to even
      INSERT (t0.0+NUMBER(1)) INTO @new_odd
END QUERY
EXIT (ISEMPTY(@new_even) AND ISEMPTY(@new_odd))
QUERY
   FOR t0 IN @new_even
      INSERT (t0.0) INTO even
END QUERY
SWAP (@delta_even, @new_even)
CLEAR @new_even
QUERY
   FOR t0 IN @new_odd
      INSERT (t0.0) INTO odd
END QUERY
SWAP (@delta_odd, @new_odd)
CLEAR @new_odd
END LOOP

Figure 2.12: Loop statement of the odd-even program in figure 2.11.
Figure 2.13: The data-flow equation to find the equivalence classes of a program.

\[ gvn(i) = \text{def} \left( i, \bigcap_{p \in \text{pred}(i)} gvn(p) \right) \]

The program \( x = 1 + 1; \ y = 1 + 1; \ z = x + y \) would have three equivalence partitions containing

- a constant 1 and its value number \( v_0 \)
- the equivalence between the variables \( x \) and \( y \), their value expression \( v_0 + v_0 \) and their value number \( v_1 \), and
- a variable \( z \) with the value expression \( v_1 + v_1 \) and its value number \( v_2 \).

The partition set is written as \( \{ v_0, 1 \mid v_1, x, y, v_0 + v_0 \mid v_2, z, v_1 + v_1 \} \), where the symbol (|) denotes separate partitions, and \( v_0, v_1 \), and \( v_2 \) are arbitrary value numbers that characterise each partition.

The GVN of a program is, commonly, found using a data-flow analysis where the approximations to the equivalence partitions are refined until they are unchanged between iterations. An optimistic algorithm will start with all variables assigned to a single partition and apply a meet operation, at any join points, to the equivalence partitions. The meet operation will be set intersection. The algorithm uses the current set of equivalence partitions to find a new value expression for each instruction successively, updating the current set of equivalence partitions. Define \( x \) to be the unique value number of \( x \) taken from the equivalence class that \( x \) is in. The program expression \( v = x + y \) is added to the equivalence class with the expression \( x + y \), or a new one is created if no expression of that form exists in any current equivalence class. The data-flow equations capturing the GVN algorithm use a binary relation between an instruction \( i \) and a set of equivalence partitions \( C \) where \( C \in gvn(i) \). The definition of \( gvn(i) \) can be seen in figure 2.13. This relation is defined using a binary auxiliary function \( \text{def}(i, C) = C' \). The function \( \text{def} \) has two arguments: an instruction \( i \) and the current equivalence partitions \( C \). The result \( C' \) is the next set of equivalence partitions including the equivalence discovered at \( i \), where the new expression from \( i \) either extends an existing class in \( C \) or creates a new class containing the equivalence inferred at \( i \) and adds it to \( C \), both giving \( C' \). The \( p \in \text{pred}(i) \) relation contains all the predecessor instructions \( p \) of an instruction \( i \), which is usually the previous instruction in its basic block, but for the first instruction of a basic block this relation could contain more than one value—the predecessor basic blocks.

The program in figure 2.14 is used to illustrate the execution of a simple GVN algorithm. The partitions derived from the meet operation on predecessor basic blocks are underlined.
Figure 2.14: An example program and the associated equivalence classes.

The equivalence set at $E_1$ is formed of three partitions, the first two assign an arbitrary value number to the constant value 1 and the effectively constant value $\text{arg}$, then the expression $\text{arg} + 1$ is considered, and their value number are substituted for their variables giving $v_2 + v_1$. The set $P_{E_2}$ is built from $P_{E_1}$ since $E_1$ is the only predecessor of $E_2$, then the value expression for $y$ is derived from the substitution of value numbers for variable applied to $x+1$. The set $P_{E_3}$ is formed from the set intersection of $P_{E_1}$ and $P_{E_2}$ (which is $P_{E_1}$). The partition $v_3$ is then extended with $z$ as it is found to have the same value expression as $v_3$. The variables $z$ and $x$ are equivalent. This fact could be used to replace all further instances of $z$ with $x$. 
CHAPTER 3

Related work

This chapter is split into three sections with work relating to the three contributions of this thesis: Datalog and **SOUFLÉ** (section 3.1), GVN (section 3.2) and programming approaches for parallelisation (section 3.3).

3.1 Datalog and **SOUFLÉ**

The **SOUFLÉ** language, first presented by Scholz et al. [50], is a high-performance Datalog engine using staged-compilation techniques and specialisation of the underlying relational data structure to specific Datalog programs. The engine outputs ‘highly efficient monolithic C++’ [50, p. 1] for an input Datalog program. The staged compilation contains front-end optimisation passes such as constant propagation and rule elimination. The front-end AST is then transformed into RAM code, optimised further and a specialisation transformation results in C++ code.

There have been multiple Datalog extensions published, some of which are already in **SOUFLÉ**: Köstler et al. [34] and Hu et al. [30]; others by Ross [48, 49] could be. Köstler et al. [34] extends Datalog with a new Datalog subsumption meta-rule and a semi-naïve evaluation alteration called ‘delta-iteration with subsumption’. The subsumption meta-rule compares two facts from the same relation with an ordering supplied as a body literal. The evaluation alteration delta-iteration with subsumption prunes the inferred facts with applications of the subsumption meta-rule, leading to the deletion of subsumed facts. This allows for search programs, where an unbounded number of superfluous facts could be inferred, to execute and terminate once only superfluous (subsumed) facts are being inferred. This can allow programs to conduct a large amount of unnecessary work before a minimal model is found. Hu et al. [30] introduces a new relational qualifier signifying the presence of a functional dependency between a complete partition of the elements in the relation, of size two, namely the domain and the co-domain. Then only one fact inferred for each element of the domain will be included in the relation, with ties broken.
non-deterministically, and all others inferred will be excluded since they would violate the functional dependency. This construct can allow efficient representation and evaluation of useful Datalog programs such as finding a spanning tree; however this introduces non-determinism into the language, which may or may not be acceptable.

Ross [48] presents a system that allows unstratifiable negation in Datalog. The system tracks dependencies of the negation and ensures that all the possible dependencies have been satisfied once before the negation is evaluated. The system must track whole program dependencies; creating a large amount of overhead. This is somewhat alleviated by a magic-set transformation (MST). The magic-set transformation introduces a set of new relations, at most one for each existing relation, which are included in the existing clauses and used to stop unnecessary inference. If a query clause \( q(Y) : - b("const", Y). \) was included in a program \( b(X,Y) : - c(X, Y), d(X, Y). \) then magic set transformation on the relation \( b \), creating \( m_b \) could be used to transform the program into \( m_b("const") \).

\( b(X,Y) : - m_b(X), c(X, Y), d(X, Y). \) This would not affect the fact set inferred in \( q \), while not inferring unwanted facts where \( X \neq "const" \) in \( b \). The dependencies of each relation are filtered by its magic set. The magic sets are built starting from the goal and working backwards. This forms a bidirectional top-down and bottom-up search with the magic set being a top-down search and the original encoding a bottom-up search. This approach requires all components, minimal mutually recursive relation sets, to be inferred in a natural order—ground terms are used for inference only in a non-cyclic order. The natural order is required for completeness of inference. This technique has a similar non-cyclic condition as the stable-companion contribution in section 4.

Ross then presents work [47] for re-writing a subclass of these Datalog programs, if they are modularly stratifiable, to allow for bottom up evaluation. Ross [49] continues the previous work via the usage of a constraint language. The constraint language allows equality, partial ordering, monotonicity or arithmetic constraints, applied by the user to extensional database (EDB) relations. The constraints are projected onto relevant program rules and used to check if the cyclic negation is acceptable. This technique allows enforceable and ‘natural’ relational constraints to be used to verify the correctness of programs. The inference system is intentionally incomplete, due to the alternative’s performance cost. The enforcement of EDB constraints can also be expensive—for example checking the monotonicity of all the facts in a relation.

The MST inspired the demand transformation (DT) [52, 10]—a simple transformation, by K. Tekle and Y. Liu, that can be applied to any Datalog program that leads to precise, known and preferable space and time complexity relative to MST. The further work by Bogaerts et al. applies the DT to stratified Datalog programs, which can usually lead to unstratifiable programs being created, however this work uses a new relation to represent the negated relation, but in positive form. That relation is filled by the negation of the
original relation. This work allows the DT to be applied to a larger amount of stratifiable programs, however still requires the input program to be stratifiable.

There has also been work to find stable models of unstratifiable Datalog programs, using execution techniques other than semi-naïve evaluation, by Van Gelder [25]. The work uses two sets for each negated relation: the usual positive relation and also the negation of the relation. The evaluation starts by initializing the positive and negative relations in the program. The evaluation process then alternates between deriving new facts for the positive and negative relations based on the newly derived facts from the other. This process continues until no new facts can be derived, at which point the alternating fixpoint is considered to be reached, and the stable model of the program is obtained. However, this model-finding system is not as efficient as semi-naïve evaluation, since lost facts are rederived in each iteration.

The Datalog language was extended, by Eiter et al. [20] and further by Leone et al. [37], to include disjunction and disjunction with nested rules which can be used to express some otherwise unstratifiable Datalog programs. These works do not however provide a scalable enough implementation to be competitive with semi-naïve evaluation systems, such as SOUFFLÉ. The disjunctive inference of the form \( r_1 : R_1 \lor R_2 \leftarrow R_3 \) allows from this clause the inference of \( R_1 \) or \( R_2 \) from \( R_3 \), but not both. The nested rules \( r_2 : R_1 \lor (R_2 \leftarrow R') \leftarrow R_3 \) allow for local constraints head disjuncts, here \( R_2 \) relies on \( R' \) and \( R_3 \) whereas \( R_1 \) only relies on \( R_3 \). If \( R_2 \) and \( R' \) are the same then the rules \((r_3 : R_1 \lor (R_2 \leftarrow R_2) \leftarrow R_3) \) can be inferred if \( R_3 \) and not \( R_2 \), which is unstratifiable in regular Datalog and therefore inexpressible in this form. The difference between the clauses \( r_1 \) and \( r_3 \) is that \( r_3 \) put an ordering on the inference, namely that \( R_1 \) can only be inferred after \( R_2 \) cannot be inferred, whereas \( r_1 \) does not.

**Dedalus** [40] builds upon a semi-positive Datalog, extending it to support temporal reasoning, through a successor relation. The goal of the project is not to answer questions about a static database of facts, but rather to program and reason about the evolution of a distributed system over time. This implementation allows negation over time, where negation between instances of the same relation at different time steps is allowed, even though this would be unstratifiable if applied on the same time step. The program in a single time step must be stratifiable (locally), but not globally and therefore any guarantee about termination is lost, however for this use case, a long-running distributed system, this is not a problem.

### 3.2 Global value numbering

Gulwani et al. [29] presents a polynomial-time value-partition intersection algorithm by limiting the number of partitions considered for intersection at a basic-block join-point.
Figure 3.1: An example program and the associated equivalence classes. The class would be constructed if a value φ-function was used in the value number association algorithm.

The ordering of the intersection is based on the distance of the expression from partitions containing an in-scope after-intersection variable. The algorithm sacrifices completeness for performance.

Gargi [24] presents a balanced algorithm, which has speed and completeness characteristics between optimistic (slower and more complete) and pessimistic (faster and less complete). The algorithm uses global reassociation, predicate inference, and value inference to find matching value-partition expressions, choosing a preferable expression as the partition leader and comparing new canonicalised expressions only to the leaders of each partition. This technique is used in modern compilers such as LLVM.

Pai [38] presents value phi functions (VPFs), a data structure, to express unexplored-equivalence-partition intersections created when partitions are operands in φ instructions. They are used lazily to search for partition equivalence. This technique can increase the precision of the algorithm without computing the intersection of partitions that are not later used by successive instructions—partitions are instead computed on demand. This technique can be combined with the approaches of the previous works, gaining the benefits of both, and which has been implemented by the LLVM compiler in the pass NewGVN (further referred to as LGVN). The usage of VPFs can be seen in the example in figure 3.1. The set $P_{E_1}$ contains all the values from an (omitted) previous basic block which are propagated at a fork point. The partitions with value numbers $v_5$ and $v_6$ are derived using value number substitution. $P_{E_2}$ is derived similarly to $P_{E_1}$. $P_{E_3}$ has underlined partitions computed from the intersection of $P_{E_2}$ and $P_{E_3}$. Then the IR φ-nodes causes $z_2$ to be added to the partition with value number $v_8$ and value-expression (vpf) $φ(v_1, v_2)$ representing the predecessor dependent choice of the values 1 or 2. The addition of the vpf allows later steps in the algorithm to search in predecessor basic blocks. The previously used alternatives were to ignore equivalence hidden by a join point or to create the set of all of them and propagate this throughout the program, if this was done there would be a partition which represents the predecessor dependent values of...
c+3: \( E_1 \) or c+2: \( E_2 \), this partition would be created and would cause later partitions to be created, all of which might never be used—this causes a large amount of wasted work. The variable \( z_1 \) and vpf \( \phi(v_5, v_7) \) are added to the partition with value number \( v_9 \). Then the expression assigned to \( z_3 \) is considered, this is seen as \( v_4 + v_8 \), then the vpf search is conducted for \( v_4 + \phi(v_1, v_2) \). This expression is equal to the previous expression, but with value numbers being replaced with their vpf expression if one exists. Then the value expressions \( v_4 + v_1 : E_1 \) and \( v_4 + v_2 : E_2 \) are searched for in the respective partition. Since both values exist there is an equivalence found between \( v_4 + v_8 \) and \( \phi(y_1, y_2) \), they are therefore added to the same partition. The work for checking previous partitions is only done in a demand driven fashion—leading to a large performance increase.

3.3 Programming approaches for parallelisation

The final contribution in this thesis (chapter 6) is an annotation language used to aid automatic parallelisation. This section provides a brief overview of select parallel programming approaches which can be applied to sequential programs, without a change in their semantics. There are broadly two parallelisation approaches: explicit, such as OPENMP [18, 3] and CILK [22], and implicit, such as VELOCITY [12], PARALAX [56] and COMMSET [39].

OPENMP by Dagum et al. [18], is an explicit parallelisation approach that presents a set of annotations, a compiler extension, and a runtime. The annotations can be applied to different programming constructs (using \#pragma omp in C/C++) to instruct the compiler to transform the construct as dictated. A loop, for example, can be marked as \texttt{omp parallel for} which instructs the compiler to run different loop iterations in parallel, thereby asserting that it is safe to do so. OPENMP was extended to include tasks, in The design of OpenMP tasks [3]. Tasks are specified units of work, which OPENMP can choose to run in parallel. A region can be marked as a task using \#pragma omp task; this task will be spawned once the region is entered by another thread. CILK, presented by Frigo et al. [22], is a programming language that ‘generalises the semantics of C by introducing linguistic constructs for parallel control’ [22, p. 1]. These constructs include: \texttt{cilk_spawn}, \texttt{cilk_sync} and \texttt{cilk_for}. The \texttt{cilk_for} construct has similar semantics to \texttt{omp parallel for}. The \texttt{cilk_spawn} and \texttt{cilk_sync} keywords are used to: spawn a new task, which carries out a piece of work and awaits (sync) the response. CILK tasks are similar to OPENMP tasks, however they encapsulate procedures whereas OPENMP tasks encapsulate blocks. CILK provides a runtime to improve the scheduling of these parallel constructs as does OPENMP.

The implicit models such as PARALAX, VELOCITY or COMMSET allow for parallelisation by communicating properties to a parallelisation system. PARALAX by Vandieren-
donck et al. [56], is an auto-parallelisation system targeting imperative programming languages, specifically with ‘irregular pointer-intensive codes’, exposing annotations such as \textbf{COMMUTATIVE}. \textbf{COMMUTATIVE} implicitly encodes the \textit{choice} of execution ordering allowing the parallelisation system more freedom when transforming. VELOCITY, Bridges [12], has a similar \textbf{Commutative} annotation and a \textit{Y-branch} annotation informing a paralleliser that a branch is less frequent and that a speculative execution style parallelisation would be fruitful. COMMSET, Prabhu et al. [39], is an annotation system for exposing commutativity information to a paralleliser, whereby regions of code are added to commutative sets; if two code regions are in a common commutative set their execution order be can be commuted (swapped).

The implicit parallel form aids an auto-parallelisation strategy and is usually designed in conjunction with the programming model. The explicit programming models are commonly more error-prone and difficult to maintain. The explicit model will allow parallelisation of a larger set of programs since the implicit forms can only encode the concepts exposed by that system.

There are also domain specific models which implicitly allow the freedom to create parallel execution namely HALIDE [41, 42], SOUFFLÉ [50]. The HALIDE language defines the data flow of the computation, without any mention of parallelisation, leaving the compiler/runtime to decide the best parallelisation scheme.
Dynamic stratification in SOUFLÉ

SOUFLÉ’s modifications to Datalog make the former Turing-complete. Despite this some programs, involving negation are difficult to express or are under-performant using only the current SOUFLÉ constructs. For example, to check if an element is absent from a known list of elements, a finite-ordered domain must be formed over the elements and then the absence proved inductively. An (arbitrary) ordering could be the order that symbols appear in the program\(^1\). To frame some negations as inductions can be very difficult due to difficulties in finding a finite-ordered domain\(^2\).

The main contribution of this chapter is the safe relaxation of Datalog’s stratifiable program constraint that allows useful programs with negation in the same (relational-symbol) stratum to be accepted and executed in a performant manner. To restore safety, permissive-dynamic checks are introduced to the runtime system analogous to quasi-static (gradual) typing [53]. While, allowing dynamic/ground-formulae stratification checks allows programs that are asymptotically faster than possible when restricted to static/relational-symbol stratification checks. This is enabled by two mutually beneficial extensions to SOUFLÉ taking the form of: a new language construct, a stable companion relation, and an execution-system alteration (ordered-stable-companion updates). These extensions form an extended language named SOUFLÉ\(^\text{stable}\), a product of this thesis. The first extension, the stable companion relation, is a mechanism to inform the runtime that a relation has been partially computed (partially stable). Relations can be considered as having a key domain and a value domain that partition the relation’s attribute set into two disjoint domains \(R(\hat{k}, \hat{v})\). Elements of the key domain \(\hat{k}\) can be marked as partially stable, signifying that all the values in the relation with that key have already been computed. No new tuples with that key will be added to the relation’s database. The elements are marked by their keys \(\hat{k}\) being included in a stable-companion relation \(R_c(\hat{k})\).

\(^1\)This can be implemented using the \texttt{ord} functor in SOUFLÉ

\(^2\)Finding an ordered domain requires finding a starting element, a total ordering on all the elements, and a final element. This domain can be both complex to implement and computationally expensive.
of a relation, which also have the same key, can be negated in a rule that is also used to update the same relation but with a different key $R(\hat{k}',...):=\ldots\!, \neg R(\hat{k},...)$, currently not possible due to the stratifiable negation constraint found in Soufflé. The second extension is a mechanism to ensure that the runtime is informed of a relation’s stable keys by inclusion into the stable-companion relation after the facts with that key are included in the relation’s database.

There has been research into extending the set of stratifiable programs by Ross [49]. Ross’s work includes constraints, which are specified by the user, on the EDB—allowing for certain forms of negation in modularly stratified programs. The work presented in this chapter complements that work by allowing users to specify constraints in pure Datalog, which, when satisfied, allow cyclic-negation, similarly to the explicit constraints in Ross’s work.

This chapter shows that Soufflé increases expressibility while also being comprehensible, which was shown by the case studies using these proposed language constructs. These case studies showed the benefit of using these constructs over what was previously available. The algorithms considered are graph exploration using a visited set and polynomial expression summation.

In summary this chapter provides the following contributions:

- An examination of and augmentation to the handling of rules in the same stratification level, allowing a precise understanding of computation progress.
- A system to partially overcome the stratification negation constraints Soufflé imposes, while still retaining execution-safety guarantees.
- A system to order the inference of related facts between relations.
- A review of the ease of implementation of the extensions.
- A demonstration of the asymptotic speed increase possible when using dynamic stratification conditions.

### 4.1 Motivating example

Many useful Datalog programs can be naturally expressed using negation. The examples presented in this chapter include computing set difference and computing the summation of affine multivariable polynomials.

---

3 Programs that respect the constraints, presented by Ross, on the EDB and IDB relations.

4 See figure 4.9, p.67.
Problem 1. The set difference of two sets $A$ and $B$ to produce a new set $C$, also presented in section 2.2.1.1, is expressed using the set comprehension in equation (4.1).

\[
C = A \setminus B = \{ x \mid x \in A \land x \notin B \}. \tag{4.1}
\]

The program in figure 4.1, is derived from the direct translation of the set comprehension in equation (4.1) (problem 1) for computing the set difference, however it is unstratifiable in SOUFFLÉ. The relation `member_of` stores the set’s elements. The EDB is populated with facts on line 3. The rule on lines 5–7 is a direct encoding of the set comprehension equation 4.1. This program is unstratifiable since the body of the clause contains the same relation as the head of the clause, but in negated form. The program should result in inference of `member_of("C","y")`. If the stratification constraint was not present this program would still be sound and result in the set $C$ being inferred correctly. This would happen since the `member_of` relation has all facts relating to $A$ and $B$ available, stabilised, before the clause on lines 5–7 is considered, at which point it is possible to infer `member_of("C","y")`. The takeaway from this example is that negation is safe\(^5\) once the projection (`member_of("B", _)`) being negated has been fully populated, or stabilised.

Problem 2. The problem of affine multivariable-polynomial summation (and simplification to a canonicalised form) is presented and a natural-SOUFFLÉ implementation is now discussed. This problem is relevant to many compiler analyses including scalar evolution [5, 55]. As an example: the polynomials $P$ and $Q$ can be summed to produce $R$:

\[
P \equiv 1 \cdot x + 3 \cdot y, \quad Q \equiv 4 \cdot y + 2 \cdot z, \quad R \equiv P + Q = 1 \cdot x + 7 \cdot y + 2 \cdot z
\]

These polynomials are stored in a relation `monomial_of` (figure 4.3(a)), where each monomial of a polynomial is stored as a separate fact. For example `monomial_of("P", "x", 1)` `monomial_of("P", "y", 3)` fully represents the polynomial $P$. The definition of a new polynomial, derived from the sum of two polynomials,

\(^5\)Discounting recursively defined polynomials.
Figure 4.2: A possible pure-Datalog implementation of the polynomial summation problem 2.

is defined in the relation sum_polynomial. The above example is specified in the fact sum_polynomial("R", "P", "Q") where the sum of P and Q is a new polynomial, named R. The monomial_of relation cannot have two monomial entries (e.g. (P, x, 3), (P, x, 4)) for a polynomial P, instead the single monomial (P, x, 7) must be included. This problem could be solved in Datalog using the program in figure 4.2. This program has two limitations, firstly this program cannot require that the expr_var_in must be in a lower stratum than expr_var, and secondly the program may carry around a large number of monomials with zero-valued coefficients. The lower stratum could be a limitation for an algorithm such as scalar evolution where the result of one polynomial simplification could require another simplification based on the result of the first. The large number of extra monomials would be a performance problem if there were too many spare polynomials.

The program, the union of all the subfigures in figure 4.3, could, almost be used to represent this problem without the limitations of the previous program, however it is statically unstratifiable.
monomial_of(e, var, c) :-
sum_polynomial(e, l, r),
monomial_of(l, var, c1),
monomial_of(r, var, c2),
c = c1 + c2.

monomial_of(e, var, c) :-
sum_polynomial(e, l, r),
monomial_of(l, var, c),
// unstratifiable negation
!monomial_of(r, var, _).

Figure 4.3: Polynomial expression rules.

Given the following initial database:


the two polynomials \( R \equiv 2 \cdot x \) and \( V \equiv 2 \cdot x + 1 \cdot y \) should be inferred in the form of three facts:

monomial_of("V", "y", 1).

The procedure to find the sum of the two polynomials is to match monomials from both polynomials by variable equivalence—if a variable is absent (from either polynomial) a zero coefficient is assumed—and then their coefficients are summed. The clause to achieve this can be seen in figures 4.3(b) and 4.3(c). The clause 4.3(c) is unstratifiable, however.

If it were possible to know that all the monomials for a polynomial were already in the database then that polynomial could be considered \textit{stable}, and it would be safe for the absence of a specified monomial to be checked for. This would mean that \( R \) must be stable before \( V \) can be fully computed; in fact \texttt{monomial_of("V","y",1)} can only be computed after \( R \) has been stabilised, \texttt{monomial_of("V","x",2)} could be computed earlier.

The stable polynomials could be tracked with a relation, which could be called \texttt{monomial_of_F(e:Polynomial)}. This relation would be initialised with \( P \) and \( Q \). Then after all the monomials of \( R \) are computed \( R \) would be stabilised and added to this relation.
\[ \text{decl } R(k_1 : T_1, \ldots, k_n : T_n, v_1 : T'_1, \ldots, v_m : T'_m) \]

\[ \text{key tuple } k \quad \text{value tuple } \hat{v} \]

\[ \text{key projection} \]

\[ \text{stable } R(k_1, \ldots, k_n) R_c \]

\[ \text{stable-companion relation} \]

Figure 4.4: The \texttt{stable} declaration is used to create a stable-companion relation named \( R_c \) for a previously declared relation \( R \), which can be used in subsequent program rules.

Finally, \( V \) could be added once both of its monomials were computed. This is the structure used to implement a solution (section 4.5.1) to problem 2.

### 4.2 Semantics

This section first presents the syntax, informal semantics of two \texttt{SOUFFLE\textsuperscript{stable}} language extensions, and finally the formal \texttt{SOUFFLE\textsuperscript{stable}} semantics.

#### 4.2.1 Language extensions

The first language extension is a stable-companion declaration, using the syntax \texttt{.stable}, which can be seen in figure 4.4. This declaration creates a new \textit{stable-companion relation} named \( R_c \) with attributes \( \hat{k} = (k_1, \ldots, k_n) \) and values \( \hat{v} = (v_1, \ldots, v_m) \) gathered from the specified keys and values of \( R \). There can be only one companion relation \( R_c \) for each relation \( R \). The new relation \( R_c \) can be used to allow rules to contain, in their bodies, negation of a stable-companion-guarded relation \( R \) updating a head relation \( R_h \), provided \( R \) is guarded by \( R_c \). The stable-companion relation \( R_c(\hat{k}) \) is used to stabilise the relation \( R(\hat{k}, \hat{v}) \) up to some key, meaning no new \( k \) keyed facts can be inferred for \( R^6 \)—this property is later referred to as being \textit{k-key stable}. If a relation \( R \) is k-key-stable, by a stable-companion relation \( R_c \), then any time an inference happens due to a clause with head \( R_h \), the companion relation is consulted as if the literal \( R_c(\hat{k}) \) was in the clause’s body. The head and body relations must be in the same stratum \( \sigma(R) = \sigma(R_h) \), since \( R \) cannot be in a later stratum to \( R_h \) \( \sigma(R) > \sigma(R_h) \) and if \( R \) were in an earlier stratum \( \sigma(R) < \sigma(R_h) \) then the guard is not necessary since this is already expressible using stratifiable negation. The semantics of these guarded relations is further considered in the section 4.3.1 (Dynamically Stratified Datalog).

The second language extension is a runtime-system modification, which orders stable-companion updates, that is used to ensure that adding a key \( \hat{k} \) to a stable-companion

\[ ^6 \text{see section 4.2.1.1.} \]
relation \( R_c(\hat{k}) \) happens after any \( \hat{k} \)-keyed facts currently\(^7\) being inferred by \( R \). This means that stabilising a relation up to a key happens after there is a request and also there are no new updates to that key in a subsequent iteration. Recall the semi-naive evaluation used in Datalog (section 2.2.1.4) where each recursive rule has a \( \Delta_R \) relation which contains all the facts inferred in the previous evaluation step. The \( \Theta_R \) relation is the next iteration’s \( \Delta_R \) relation. The to-be-stabilised relations \((R(\hat{k}, \hat{v}), \Delta_R(\hat{k}, \hat{v}) \) and \( \Theta_R(\hat{k}, \hat{v}) \)) are defined for tuples of size \(|\hat{k}| + |\hat{v}|\) for non-negative integers sizes \(|\hat{k}|\) (the number of key elements) and \(|\hat{v}|\) (the number of values elements). The stable companion relations \((R_i^c(\hat{k}), \Delta_{R_i^c}(\hat{k}) \) and \( \Theta_{R_i^c}(\hat{k}) \)) are defined for tuples of size \(|\hat{k}|\). The notation \( (\_ ) \) is used to mark an ignored variable; one that is equivalent to \( R(\hat{k}, \hat{v}') \) for some fresh tuple variable \( \hat{v}' \). The companion relation is converted, by the compiler, into two relations, opaquely to the user, the input-stable relation \( R_i^c(\hat{k}) \) and stable relation \( R_c(\hat{k}) \). The former relation (input stable \( R_i^c(\hat{k}) \)) is used to request that a key should be stabilised, and the latter (stable) relation is used to indicate that the key has been stabilised. The updates propagate into the stable companion relation when the relation is momentarily stabilised, defined in equations (4.2) and (4.3). A relation is momentarily stabilised when no values for that key were inferred in the last iteration (4.2), this is only checked for input stable keys \((R_i^c(\hat{k}))\). The equation (4.3) ensure that any input stable key (in \( \Delta_{i^c}(\hat{k}) \)) is considered for stabilisation on successive iterations. The compiler converts any mention of \( R_c \) in the head of a rule into \( R_i^c \), any mention of \( R_c \) in the body of a rule is left unchanged, this conversion happens before considering \( \Delta \)-relations. The pair of stable-companion relation clauses \( R_i^c \) and \( R_c \) are linked with the following, ordered-stable-companion updates, rules:

\[
R_c(\hat{k}) : - (R_i^c(\hat{k}), \Delta_R(\hat{k}, \_)) = \emptyset \quad (4.2)
\]

\[
\Theta_i^c(\hat{k}) : - \Delta_{i^c}(\hat{k}), \Delta_R(\hat{k}, \_) \quad (4.3)
\]

Equation (4.2) computes the intersection over the keys of \( R_i^c \) and \( \Delta_R \), checks if this set is empty and if so adds \( \hat{k} \) to \( R_c \) — stabilising \( \hat{k} \) in \( R \).

The set difference, presented in problem 1, can be implemented using the .stable declaration, seen in figure 4.5 line 3. The member_of_f relation is the stable-companion to the member_of relation. \( A \) and \( B \) are considered as stable initially since the program does not modify either set. If the program was modified to cause extra facts to be inferred in either \( A \) or \( B \) then the companion relation inference clause must be modified accordingly. The stabilising of \( C \) happens once \( B \) is stabilised. This inclusion on line 16 opaquely adds \( C \) to the input of the companion relation, then the fact member_of_f("C") is only observable once inference for member_of("C", _) has completed \((\Delta_{\text{member_of}}("C", _) = \emptyset)\). It is assumed that once a key is added to a stable-companion input relation that no new

\(^7\)In the same application of the consequence operator.

\(^8\)previously unused in that clause
.type Set ::= symbol
.decl member_of(set: Set, element: symbol)
.stable member_of(set) member_of_f
member_of("A", "x").
member_of("A", "y").
member_of("B", "x").
member_of_f("A").
member_of_f("B").

member_of("C", e) :-
  member_of("A", e),
  member_of_f("B"),
  !member_of("B", e).

member_of_f("C") :-
  member_of_f("B").

Figure 4.5: Set Difference in SOUFFLÉ using the stable companion relation member_of_f.

inference can happen for this key and this delay ensures that pending inferences are completed before the key is stabilised.

4.2.1.1 Stabilising relations to enforce safety

The safety afforded to (statically) stratifiable Datalog programs is lost when programs have rules with stable-companion guarded negation but an execution of that program does not dynamically stratify—defined in section 4.3.1. These programs can have safety enforced by applying dynamic restrictions to the inference of new facts to any relation with a stable declaration. A restriction to the new inference of facts is enforced by only allowing new facts to be added to a relation $R(\hat{k}, \hat{v})$ if the key is not in $R_c$, checking that $\neg R_c(\hat{k})$ holds. If this assertion is violated the program will either stop execution and report an error or continue execution while reporting a warning of this violation. This check could also be explicitly disabled for performance reasons.

4.3 Formal Semantics

The language extensions presented in this chapter, namely .stable, use a modified consequence operator $\Gamma_P(I)$ (section 2.2.1.3) and semi-naïve consequence operator $\Gamma_P(\Delta, I)$ (section 2.2.1.4) to formally define their semantics. The modifications enforce the immutability of a $\hat{k}$-key-stable relation and require that the order of updates to the stable-companion relation, with a key $\hat{k}$, happens after updates to the $\hat{k}$-key-stable relation have taken place, for any specific key $\hat{k}$. This ensures the $k$-containig tuples are installed before declaring $k$
Consider a relation \( R(\hat{k}, \hat{v}) \) with two tuples \( \hat{k} = (k_1, \ldots, k_n) \) and \( \hat{v} = (v_1, \ldots, v_m) \), with a stable companion relation \( R_c(\hat{k}) \), storing stable keys of the relation \( R \). The domain \( D_a \) of an attribute \( a \) (in this case either \( k_i \) or \( v_j \)) is the set of all possible values that that attribute can take. Define \( M_R = D_{k_1} \times \cdots \times D_{k_n} \times D_{v_1} \times \cdots \times D_{v_m} \) to be the Cartesian product of the attributes in the domain of \( R \). The auxiliary exclusion function, a mathematical function, \( e^{A\Delta k}(A') \) is defined for attribute sets \( A, A' \subseteq M_R, A_c \subseteq M_{R_c} \) and relation \( R \), where \( A \) is the previously inferred fact set and \( A' \) is the newly inferred fact set and \( A_c \) is the previously stable key set of the relation \( R \):

\[
e^{A\Delta k}(A') = A' \setminus \{ t \in A' \mid t \notin A \land t = (\hat{k}, \hat{v}) \land \hat{k} \in A_c \}
\]

This function, as stated informally, enforces the stabilising of a relation \( R \), up to all the values previously inferred for a specific key \( \hat{k} \), before the relation is \( \hat{k}\)-key-stable. \(^9\)

Let \( \text{sch}(P) \) be the schema of \( P \) and let \( I \subseteq \text{sch}(P) \) and \( I' \subseteq \text{sch}(P) \) be program schema instances. The stable-exclusion function \( E^I_P(I') \) is defined in terms of \( e^{A\Delta k}(A') \). Where \( I \) is the previously inferred fact set and \( I' \) is the currently inferred fact set. The correct companion attribute facts \( I(R_c) \) from the previous database \( I \) and relation facts \( I'(R) \) from the current database \( I' \) are projected out and used by \( e^{A\Delta k}(A') \) inside \( E^I_P(I') \) for each companion relation.

Define a new relation type \( \delta_R \) capturing facts inferred by \( R \) between at this step and the last, used shortly. Define a function \( \text{rel}(t) \) to be true if the argument is a relational fact—a fact from a relation \( R \), where \( R \) is not of the form \( \delta_R \)—and otherwise be false. Define \( \text{del}(t) \) similarly but checking \( t \) is a \( \delta \) relational fact, instead—hence for any \( t \) either \( \text{rel}(t) \) or \( \text{del}(t) \) holds true \( \Leftrightarrow \text{del}(t) \lor \text{rel}(t) \). The consequence operator with stable-exclusion is defined as:

\[
\Gamma_P(I) = I \cup E^I_P(\{t \mid t \leftarrow t_1, \ldots, t_k \text{ is a rule instantiation with each } \text{rel}(t_i) \land t_i \in I\})
\]

This update to the consequence operator enforces the stabilising of a relation \( R \), up to a key \( \hat{k} \) if that key is included in \( R_c(\hat{k}) \), for each step of the fixpoint calculation.

The method for ordering stable-companion updates \( (R_c(\hat{k})) \) of newly stabilised keys uses a new opaque (invisible-to-programmer) relation \( R'_c \), defined in the language extensions section 4.2.1. The inclusion is only observable once the relation \( R \) has no immediate updates, to \( R(\hat{k}, \ldots) \) (when \( \delta_R(\hat{k}, \ldots) = \emptyset \)). This clause queries the \( \delta \) relation to ensure that no more inferences for \( k \) are happening or will happen. This is enforced by adding to

\(^9\)Once \( R_c(\hat{k}) \) holds.
the program, *no-immediate-update clauses*, which have the form:

\[ R_c(\hat{k}) : - R'_c(\hat{k}), (\delta_R(\hat{k}, \ldots) = \emptyset) \]

This requires the possibility that clauses contain literals formed from relations \( R \) and delta relations \( \delta_R \). Since semi-naive evaluation cannot be assumed \( \delta_R \) relations are used instead. Let \( D \) be a schema instance, found by taking the difference of the current \( I' \) and previously inferred facts \( I, D \equiv I' \setminus I \). The consequence operator with fix-exclusion and no-immediate-update clauses is defined as:

\[
\Gamma'_p(I) = I \cup E^1_{p}(I') \quad \text{where} \\
I' \equiv \Gamma_p(I) \quad \text{and} \quad D \equiv I' \setminus I \\
I'' \equiv \{ t \mid t \leftarrow t_1, \ldots, t_k \text{ is a rule instantiation with each} \\
(rel(t_i) \land t_i \in I) \lor (del(t_i) \land t_i \in D) \}
\]

This consequence operator is monotonic and therefore has a fixpoint, meaning there exists a least model of \( I = \Gamma'_p(I) \). The monotonicity holds since the changes to the consequence operator, stable-exclusion and \( \delta \)-relations, cannot cause already inferred facts to be discarded. The semi-naive evaluation consequence operator with stable-exclusion is defined as:

\[
\Gamma'_p(\Delta, I) = I \cup E^1_{p}(\{ t \mid t \leftarrow t_1, \ldots, t_k \text{ is a rule instantiation with each} \\
(rel(t_i) \land t_i \in I) \lor (del(t_i) \land t_i \in \Delta) \land \\
\exists t_j \in \Delta \})
\]

This semi naive operator implementation also requires another clause per companion relation that propagates facts \( t \in \Delta_{R_c} \) that are not in \( t \not\in R_c \) to the next iterations \( \Delta_{R_c} \), see section 4.2.1 equation (4.3).

### 4.3.1 Dynamically stratified Datalog

The section presents a method for assessing the correctness of a dynamically stratified Datalog program, if a *dynamic precedence graph* (DPG), defined shortly, can be constructed, then the program is sound. *Dynamic stratification* of Datalog is only relevant for a set of relations at the same stratification level—called a stratum. The partially ordered inference of the stable-companion and guarding of stratifiable negation can allow programs to be executed that originally would have been rejected by SOUFFLE due to not being stratifiable. The program partition at the problematic stratification level would be allowed if they were correctly guarded and would produce correct results if they were in fact dynamically
nodes(R) = \{ R\#\rho_R(\hat{k}) \mid (\hat{k}, \_ ) \in R \}\)

\[
\text{NODES } = \bigcup_{R \in P^i} \text{nodes}(R)
\]

Figure 4.6: The nodes in a DPG for a program partition \( P^i \in P \) with a cycle in its precedence graph (PG) (p. 37) containing a negative edge and a guarded negation.

- \( R(k, u) :- \ldots, R'(k', v), \ldots \) gives
  \[
  \{(e, e', +) \mid e \in \text{edges}(R\#\rho_R(k)) \land e' \in \text{edges}(R'\#\rho_R R'(k'))\}\)

- \( R(k, v) :- \ldots, R_f(k'), !R'(k', v'), \ldots \) gives
  \[
  \{(e, e', -) \mid e \in \text{edges}(R\#\rho_R(k)) \land e' \in \text{edges}(R'\#\rho_R R'(k'))\}\}

Figure 4.7: The edges in a DPG.

stratifiable.

Define the identification list notation (written \( a\#b \)). The notation creates a new name, distinct up to its arguments \( a\#b \neq a'\#b' \iff a \neq a' \lor b \neq b' \). A DPG is similar to a PG in its structure, however it is formed from the trace of a program execution, an example can be seen in figure 4.12. The DPG can be used to assert correctness of a specific program solution under a fixed input. The key ordering relation \( \gamma_R \) from attribute domains to \( \mathbb{N} \) is used similarly to the stratification mapping but for keys instead of relations. An ordering must be chosen to satisfy the dynamic stratification condition presented, shortly, in theorem 2 p. 65. The nodes of the graph are of the form \( R\#\gamma_R(\hat{k}) \) for a relation \( R \) with an attribute tuple split \( \kappa_R = (\hat{k}, \hat{v}) \), the split must be chosen for each relation without a stable-companion.\(^{10}\) A tuple \( t \) in a relation \( t \in R \) can be disjointly partitioned into the key and value parts using \( \kappa_R(t) = (\hat{k}, \hat{v}) \) where \( \hat{k} = \kappa_R(t)_k \) is a key tuple and \( \hat{v} = \kappa_R(t)_v \) is a value tuple.

**Theorem 2.** A program is *dynamically stratifiable* if the DPG has no cycles that contain a negative edge (the dynamic stratification condition).

If a relation has a stable set attached or is in a negative PG-cycle with a relation in the cycle that has a stable set, there will be many nodes derived from this relation, (shown in figure 4.6).

The edges of the DPG are formed by creating positive and negative edges between nodes indexed by their key as found from the key-value-mapper \( \kappa_R \).

\(^{10}\)As one is explicitly chosen in the .stable declaration.
• $R'(\hat{u}) : - \ldots, R(\hat{v}), \ldots$ gives

$$\{(e, e', +) \mid e = R\#\gamma(\hat{k}) \land e' = R'\#\gamma(\hat{k}) \land \hat{k} = \kappa(\hat{u})_k = \kappa(\hat{v})_k\}$$

• $R'(\hat{k}, \hat{v}') : - \ldots, R_f(\hat{k}), !R(\hat{k}, \hat{v}), \ldots$ gives

$$\{(e, e', -) \mid e = R\#\gamma(\hat{k}) \land e' = R'\#\gamma(\hat{k}) \land \hat{k} = \kappa((\hat{k}, \hat{v}))_k = \kappa((\hat{k}, \hat{v}'))_k\}$$

Figure 4.8: The edges in a DPG when all keys in the cycle are the same one key mapping function will suffice.

A program partition is dynamically stratifiable if it satisfies the stratification condition (see theorem 1 p. 38) for a DPG. The conditions require that there are no cycles in the DPG (DG_P) of a program P with a negative edge in the of path of that cycle.

The graph simplifies to having edges shown in figure 4.8 when all keys $\hat{k}$, key partitions $\kappa$, and key mappings $\gamma$ are the same. Dynamic stratification condition (theorem 2), instead, requires that the key domain is explored in a partial order. This ordering ensures that any value inferred for a key $k$ from a key $k'$ is never used to infer a new value for $k'$ (from $k$).

The dynamic precedence graph is isomorphic to a perfect model (section 2.2.1.9) graphed with edges equivalent to the $\hookrightarrow$ relation and the transitive closure of edges $E^+$ equal to the ordering relation $>$. The adjusted consequence operator disallowing inference of a $k$-key-stable relation ensures the required perfect model semantics.

### 4.3.2 Stable semantics in the presence of cyclic negation

The stable extension to SOUFFLÉ allows cyclic relation dependencies to be specified.

**Problem 3.** Find all the models that satisfy the following formulae:

\[
p \leftarrow \neg q \\
q \leftarrow \neg p
\]

There are two models satisfying the formula, in problem 3, $\{p\}$ or $\{q\}$. They both happen to be stable models, defined by Gelfond et al. [26]. The following SOUFFLÉ program with stable negation (in figure 4.9) is a solution to the problem, where either model can be chosen by defining the other relation as stable. This allows the programmer to define which of a multiset of solutions will be found by the program execution. This choice can be seen as an assertion of closed world assumption (CWA) [45] applied to the tuples with keys in a stable relation. That in this example is either including line 7 or line 8 in figure 4.9.
.decl p() .stable p() p_s
.decl q() .stable q() q_s

p() :- q_s(), !q().
q() :- p_s(), !p().

// p_s(). finds the model { q }
// q_s(). finds the model { p }

Figure 4.9: The SOUFFLÉ program that encodes problem 3, where either relation could be stabilised allowing the other to be inferred.

4.4 Implementation

This section describes how the extensions are implemented in the SOUFFLÉ (version 2.2) compiler and runtime.

The changes required to include into SOUFFLÉ the aforementioned language extensions affect the following modules of the SOUFFLÉ system: the parser, AST grammar, AST transformation, AST verification and translation from the AST to the RAM code (ast2ram).

1. The parser is modified to include the new syntax (.stable).
2. The AST is extended to capture the stable set, this is added as a new AST node.
3. A new internal AST node, DeltaImmediatelyEmpty, is added to model the emptiness check of the delta relation ($\Delta_R(\hat{k}, \_\_) = \emptyset$), used to order stable-companion updates.
4. A new AST transformation creates two new relation declarations, for the companion set $R_c$ and companion set input $R^i_c$, declared in the stable relation.
5. A new AST transformation to represent the clause from equation 4.2 using the DeltaImmediatelyEmpty node.
6. An update to the precedence graph and mutual recursive analyses\(^{11}\) to include this new AST node.
7. The stratifiable program check is relaxed to support dynamic stratification for stable-companion guarded negation.
8. An ast2ram change to include a RAM-check that any clauses with a head atom $R(\hat{k}, \_\_)$ with a companion relation $R_c(\hat{k})$ are not $\hat{k}$-key-stable.

\(^{11}\)A simple analysis to discover which set of clauses in a stratum are mutually recursive in their inference.
9. An ast2ram extension to understand the DeltaImmediatelyEmpty node.

10. An extension to ast2ram to capture equation (4.3).

The changes to the generated RAM code, points 8–10, are discussed further. Consider a relation \( R(\hat{k}, \hat{v}) \) with a companion relation \( R_c(\hat{k}) \) and input companion relation \( R^n_c(\hat{k}) \). The tuple \((t.0, \ldots, t.n, t'.0, \ldots, t'.m)\), in RAM code, represents the relations attributes \((\hat{k}, \hat{v})\), with \( t_i \) being key attributes and \( t'_i \) being value attributes. Point 8 includes a new filter (IF) construct, from figure 2.8, to enforce the \( \hat{k} \)-key-stable constraint.

\[
\text{IF NOT } (t.0 \ldots , t.n) R_c \\
\ldots \\
\text{INSERT } (t.0 \ldots , t.n, t'.0, \ldots, t'.m) \text{ IN } R.
\]

Point 9 converts the DeltaImmediatelyEmpty\((R(\hat{k}, \_))\) node to the RAM operation:

\[
\text{NOT } (t.0 \ldots , t.n, \text{UNDEF}_0, \ldots, \text{UNDEF}_m) \text{ IN } @\text{delta}_R.
\]

Point 10 shows the fragment corresponding to equation (4.3).

\[
\text{FOR } t \text{ IN } @\text{delta}_R^n_c \\
\text{IF } ((t.0, \ldots, t.n, \text{UNDEF}_0, \ldots, \text{UNDEF}_m) \text{ IN } @\text{delta}_R) \text{ AND } \\
\text{(NOT } (t.0, \ldots, t.n) \text{ IN } R_c) \\
\text{INSERT } (t.0 \ldots , t.n) \text{ INTO } @\text{new}_R^n_c
\]

Appendix A shows the transformed RAM of the program in figure 4.10.

4.5 Evaluation

The evaluation comprises multiple programs which make use of SOUFLÉ\textsuperscript{stable} as explained previously in this chapter.

4.5.1 Polynomial expression program

A possible solution (figure 4.10) to problem 2 of summation of affine multivariable-polynomial expressions uses the language constructs (.stable) presented in this chapter.

The PG graph is shown in figure 4.11 which contains a cycle with a negative edge between monomial\_of and itself. The two relations monomial\_of and monomial\_of\_f violate the statically computed stratifiable negation constraint and instead are examined using a DPG in figure 4.12. The graph has no negative edge-cycles for the input shown on
\[ \text{decl sum\_polynomial(e: Expr, l: Expr, r: Expr)} \]
\[ \text{sum}\_\text{polynomial("2x", "x", "x"). // 2x = (x) + (x)} \]
\[ \text{sum}\_\text{polynomial("2x+y", "2x", "y"). // 2x + y = (2x) + (y)} \]

\[ \text{decl monomial\_of(e: Expr, var: Variable, c: number)} \]
\[ \text{monomial\_of("x", "x", 1).} \]
\[ \text{monomial\_of("y", "y", 1).} \]

\[ \text{stable monomial\_of(e) monomial\_of\_f} \]

\[ \text{monomial\_of(e, var, cl) :-} \]
\[ \text{sum}\_\text{polynomial(e, l, r)}, \]
\[ \text{monomial\_of(l, var, cl)}, \]
\[ \text{monomial\_of\_f(r)}, \]
\[ \text{!monomial\_of(r, var, _).} \]

\[ \text{monomial\_of(e, var, cr) :-} \]
\[ \text{sum}\_\text{polynomial(e, l, r)}, \]
\[ \text{monomial\_of\_f(l)}, \]
\[ \text{!monomial\_of(l, var, _)}, \]
\[ \text{monomial\_of(r, var, cr).} \]

\[ \text{monomial\_of(e, var, cl + cr) :-} \]
\[ \text{sum}\_\text{polynomial(e, l, r)}, \]
\[ \text{monomial\_of(l, var, cl)}, \]
\[ \text{monomial\_of(r, var, cr).} \]

\[ \text{monomial\_of\_f(e) :-} \]
\[ \text{sum}\_\text{polynomial(e, l, r)}, \]
\[ \text{monomial\_of\_f(l), monomial\_of\_f(r).} \]

Figure 4.10: Polynomial sum using SOUFFLE\text{\_stable}.

Figure 4.11: Shows the PG for the program in figure 4.10, where the letters stand for the relations: \( F = \text{monomial\_of\_f}, M = \text{monomial\_of} \) and \( R = \text{sum\_polynomial} \). The dash line represent the edges included from the opaque companion update clause.
Figure 4.12: The DPG of the polynomial sum program in figure 4.10.
Figure 4.13: An EDB, for the program in figure 4.10, that causes a cyclic summation.

lines 3–6, and in fact for any input which is non-cyclic. The EDB (figure 4.13) contains an ill-defined-cyclic-polynomial definition, from which no new inference would occur when used in conjunction with the program (figure 4.10), with stable semantics and all the facts removed. Despite the rule on lines 23–26 being applicable with the assignment $e = P, \text{var} = y, cr = 1$. The exclusion function, equation (4.4) in section 4.3, stops this inference. There is an implicit dependency to monomial_of from monomial_of_f, due to the DeltaImmediatelyEmpty clause ensuring that a polynomial is only stabilised once all immediate updates to monomials of that polynomial have been applied. This may change the mental model of the programmer, however in this example once a polynomial is found stable then on the next update iteration all polynomials depending on this one will have all the dependencies satisfied, then all pending updates to those polynomials will be noticeable when checking their $\Delta$ relations, meaning that the stabilising of that key can be safely initiated without the possibility of missing monomials. Then once the $\Delta$ relation does not contain that key then the updates will have been applied. This logic applies to this program, if there is the possibility that for a given key new values could be inferred in a different pattern then the stabilising of that key (via inclusion of the key in the stable companion input relation) must reflect that.

This program shows the possibilities these language extensions offer—an alternative to induction over the variables of each polynomial. The stable approach does not require the creation of an ordered domain of variables in each expression and has the benefit that checking for zero valued coefficients in one of the two polynomials is constant time instead of linear time (in the case of an ordered domain). With an ordered domain to check if a variable $w$ is absent from polynomial ($P \equiv x + y + z$), first $w$ is checked against $x$, then $y$ and finally $z$. Since $z$ is the final variable in $P$ once $w$ is shown to not be equal $z$ and all previous variables it is known that $w$ is not in $P$. The process is linear in the number of variables in $P$.

4.5.2 Graph search with visited set

Graph search algorithms are used extensively in computer science, and in compiler analysis algorithms. The specific graph algorithm used in this section is DFS. The problem is
defined in terms of relations representing the cyclic-graph, lines 1–2, and the relation to run the search, lines 4–7, seen in figure 4.14.

There are two implementations of the DFS algorithm in Datalog using different approaches: induction (figure 4.15 $P_I$) and a stable-companion relation (figure 4.16 $P_S$). The DFS implementation of the search goes as far down as possible into one branch and then backtracks to search siblings in an edge-indexed-order.

The input to both programs $P_I$ and $P_S$ is the next_request relation, where candidates for the next node to be explored are proposed. The rules for producing the traversal order are superfluous and therefore omitted, these rules use the visited and skip_node relations to generate new requests—updating the visited relation. The generated request will be at a new step, if the previous node was not visited, or at a new attempt ($\text{try} + 1$) at the current step, if the previous node has already been visited. The nodes are examined to see if they have been visited before. If they have not been they are accepted into the visited set and this is checked using the rules: on lines 18–21 for $P_I$ in figure 4.15 and lines 21–26 for $P_S$ in figure 4.16. If they have been visited before (checked in figure 4.14 on lines 9–12) they are rejected and put into a skip_node relation. The program $P_I$ uses induction to check that each candidate\(^\text{12}\) is not in the list of previously visited nodes (lines 4–16), using the not_visited_upto relation to check each previous step. The program $P_S$ uses a stable relation (visited_response, lines 8–11), with a companion relation (visited_response_f updated on lines 13–15) to allow the examination of the negated visited_response in a cycle only after the request’s step and try have been fully computed. The program $P_S$ will dynamically stratify since only previous steps are examined by the current step. The guard set visited_response_f will only defer the negation check for one step since when every new node is considered (added to next_-

\(^{12}\text{A step, try pair.}\)
Figure 4.15: Graph exploration using induction $P_I$.

request) the previous step has already been considered meaning the not-visited-check will be fully evaluated after a single step. This program looks similar to a program produced by the demand transformation applied to a program checking for a cycle using this check \texttt{prev\_step < step, !visited(n, prev\_step, \_)} replacing lines 19, 21 and 22 in figure 4.16. The benefit of the stable approach $P_S$ is that the demand transformation will require many superfluous relations to track that the possible dependencies, between all previous steps and the current one, are satisfied requiring unnecessary memory and computation. To be more specific, without any domain knowledge each step must be passed to all future steps, since the demand transformation will not be able to determine that the previous step subsumes all steps before that step when discovering if a step has been computed.

Table 4.1 shows the performance of both programs $P_I$ and $P_S$. The graph $G = (N, E)$, used as input, was generated at random for each execution. The graph was generated by taking a random walk, with each step choosing a random unconnected node and adding an edge from a random connected node, starting from a randomly chosen head node. Random edges are then added once each node is accessible from the head resulting in a graph with four times the number of edges than nodes $|E| = 4 \cdot |N|$.

The asymptotic complexity of the algorithm, for a graph $G = (E, V)$, is $O(|E| + |V|)$.
.decl dfs(n: Node, i: Step)
.stable visited_response(i, try) visited_response_f

visited_request(n, prev_step, try) :-
    prev_step = step - 1,
    next_request(step, try, n).

visited_response(step, try, n) :-
    visited_request(n, step, try),
    visited(n, prev_steps, _),
    prev_steps <= step.

visited_response_f(0, 1).
visited_response_f(step, try) :-
    visited_request(_, step, try).

visited(n, step, try),
dfs(step, n) :-
    prev_step = step - 1,
    next_request(step, try, n),
    visited_response_f(prev_step, try),
    !visited_response(prev_step, try, n).

Figure 4.16: Graph exploration using a stable-companion set ($P_s$).
Node Count | Induction $P_I$ | Stable $P_S$ | Ratio (I/F)  
---|---|---|---
50  | 0.045 | 0.044 | 1.02  
500 | 0.059 | 0.049 | 1.20  
2500 | 0.553 | 0.077 | 7.18  
5000 | 2.098 | 0.113 | 18.57  
10000 | 8.527 | 0.116 | 73.51  
20000 | 34.5 | 0.193 | 178.76  
40000 | 136.79 | 0.384 | 356.22  
80000 | 588.91 | 0.683 | 862.24  

Table 4.1: The runtime of the DFS of a randomly generated graph using a visited set in Datalog using both approaches.

$O(n)$ for $n = |E| + |V|$. The runtime of $P_I$ is $O(n^2)$, since $k \cdot n - 1$ nodes are considered and for each step with $k$ attempts\(^{13}\), this happens $n$ times. The complexity of $P_S$ is $O(n \log n)$ since the check that the node has not been visited is logarithmic. However, in practice the runtime is linear due to implementation details (e.g. a larger branching factor in the backing b-tree). The relative speed-up from the stable method $P_S$ over the inductive method $P_I$ can be seen in figure 4.17(c). This is a linear speed-up in $n$ showing that for these random inputs $P_S$ gives a linear speed-up relative to $P_I$.

The complexity of the $P_S$ approach does not have the overhead of $j$ extra steps of the semi-naïve evaluation loop, whereas $P_I$ does. The loop is used for computing the $j$th DFS iteration step, since the computation of the response relation only requires one step in the evaluation loop.

The program $P_S$ has a considerably lower runtime compared to $P_I$, with around an 800x speed-up, for the largest input size and this difference will of-course be larger for bigger input sizes. The input size for each graph $G = (E, V)$ is taken as $n = E + v$. The median of ten runs was taken for each node count. The possibility to have multiple nodes considered in each step would be much easier to implement using a stable-companion relation than an inductive solution, since in an inductive solution the new nodes considered at step $i$ must be given an order (after being filtered by not being visited before) and then induced over. The stable approach however would be similar to the solution for one node per step presented here, since each set of nodes visited at a single step would all be considered in one LOOP-step.

The common graph search algorithm used in Datalog is breadth-first search (BFS), which is better suited to the parallel memorized evaluation of Datalog, however if non-unique a node metric was being computed (e.g. the shortest distance to a node) then cycle detection would be required as in the DFS implementation, leading to the same inductive

\(^{13}\)This analysis assumes that $k$ is small and constant, which will not necessarily be true for a highly connected graph.
Figure 4.17: The runtime of both DFS implementations and the relative runtime in subfigure (c). Note the different input ranges and the scaling of the input sizes: subfigure (a) has 10,000 and subfigure (b) has 1000.
vs stable implementation performance. The DFS algorithm was chosen for analysis since the GVN algorithm in the next chapter has a similar graph exploration structure. The next section uses the SOUFFLÉ\textsuperscript{stable} extension to achieve desirable performance. The poor performance of the inductive implementation would be amplified if applied to the GVN algorithm and is therefore not presented.

4.6 Summary

Adding dynamic stratification to SOUFFLÉ allows programs previously forbidden, such as the program to efficiently compute the summation of affine multivariable-polynomial expressions (problem 2, p. 57) in a single stratum. This pure-Datalog solution (figure 4.2) required the input to be in a lower stratum. Dynamic stratification is supported by the stable-companion (section 4.2.1) technique. The technique also allows programs to be implemented in a more performant manner. The dynamic stratification condition allows programs to provide the same safety guarantees in SOUFFLÉ\textsuperscript{stable} that are present in SOUFFLÉ. The programs used to solve the problem presented in this chapter can be implemented in a succinct way due to the stable-companion extension. For example, the polynomial summation problem would require an arbitrary finite-ordered domain to be created for all the variables of each polynomial such that they can be summed using an inductive approach. This polynomial summation program is a building block for a ubiquitous program analysis scalar evolution which would be made more performant and succinct in its declarative implementation when using the SOUFFLÉ\textsuperscript{stable} extension.

The stable-companion language extension SOUFFLÉ\textsuperscript{stable} is used in the next chapter to implement a common compiler analysis, namely GVN.
GVN is a fundamental program analysis tasked with finding redundant expressions in procedures that can be eliminated by later program transformations, leading to smaller generated programs with fewer instructions necessarily executed. This chapter demonstrates the power of declarative languages when implementing complex and useful compiler analyses, with GVN as the running example, while showing the added benefit of using the stable-companion-relation language extension (SOUFFLÉ\textsuperscript{stable}) explained in the previous chapter. This results in a program capable of analysing select SPECint2006 benchmarks.

GVN assigns each register to a partition denoted by a unique value number with equivalent registers sharing the same value number (see section 2.3). My research results in a program named DGVN, building on work by Gulwani et al. [29] and Pai et al. [38] where their approaches are adapted to a declarative language instead of the imperative algorithm previously used. Gulwani et al.’s work constitutes a polynomial-time GVN algorithm making use of expressions as uninterpreted functions in each value partition. These expressions are used by DGVN as a starting point for the value expressions and their relation to value numbers. Pai et al.’s work consists of a novel way to delay the basic-block join-intersection computation using value phi functions (VPFs) (a data structure defined in section 5.1) along with a search procedure to find relevant value numbers. This technique is used in DGVN. LLVM uses a GVN algorithm building on work by Pai et al. and Gargi [24]. LLVM uses a sparse representation of value partitions, as in Gargi’s work, operating on a def-use relation\textsuperscript{1} to track which variables are affected by changes to a value partition, meaning partitions may span multiple basic blocks. The value numbering also makes use of predicated constant folding, algebraic simplification and unreachable-code analysis.

Many program analyses have been specified as Datalog programs [11, 51, 8, 2]. The authors of these programs concluded that their declarative nature expressed the essence of the algorithm. The declarative nature of DGVN results in a complete and clear overview of all important aspects of this algorithm (section 5.2). This work exemplifies a common

\footnotesize{\textsuperscript{1}A relation tracking which variables use a defined variable.}
declarative pattern, the join-intersection computation, that can be reused in other meet-based fixed-point iteration algorithms, which are difficult to implement in Datalog due to its inability to remove facts inferred in recursive clauses from its database\(^2\). The DGVN implementation uses SOUFFLÉ\(^\text{stable}\), the stable-companion-relation language extensions defined in the previous chapter, which allows the efficient implementation of the DGVN algorithm. This work also solves the problem of creating new distinct value numbers, previously a side-effecting counter, by using relevant expressions to derive a value number. The DGVN approach can make use of SOUFFLÉ’s implicitly parallel execution model to concurrently explore independent value partitions. DGVN is evaluated, in section 5.3, against LLVM’s GVN implementation, referred to as LGVN. The evaluation includes a comparison of DGVN against LGVN using industrial-strength benchmark code from SPECint2006 and PARSEC-3.0 [9].

5.1 Motivating example

The GVN algorithm partitions all register variables by equivalence, allowing further transformations access to this equivalence information. The join points in the CFG present a performance (or completeness) challenge when implementing this algorithm, since either many partitions from each predecessor basic block must be aggregated or partition information must be discarded. Pai et al. [38] presents value phi functions (VPFs), a remedy to this problem. A VPF expression, stores a list of values for each partition, one from each predecessor basic block. If all the value numbers were \(\phi\)-merged\(^3\), they would be equivalent to the value expression in this partition.

Pai’s GVN algorithm has two main inhibitors impeding a declarative implementation (in Datalog): the side-effecting function to generate new value numbers and the search procedure to find new VPFs.

Partitions are characterised by value numbers \(v\). When a new partition is required a side-effecting function is called to produce a new value number. This function is likely a counter, which inhibits parallelism, especially in a declarative system where this counter must be propagated throughout the rules. Partition value-expressions are instead used to generate a value number based on their uniqueness. Possible expressions are extended to include expressions generated from basic-block joins as value-expression \(\phi\)-nodes.

When a new IR expression is considered, a procedure searches for its, possibly null, VPF from the elements in previous basic blocks. This VPF is necessary to assign the expression a value partition. For this procedure to find a VPF to be null, it must be checked that there is no expression or VPFs matching any predecessor variables in that

\(^2\)Computing the complement of this relation—variable inequivalence—is not computationally tractable.

\(^3\)The VPF \(\phi(v_1 : bb_1, \ldots, v_n : bb_n)\) is dynamically (at runtime) equivalent to the value number \(v_i\) if the predecessor basic block was \(bb_i\). The static equivalence will be discussed in section 5.2.
Figure 5.1: the CFG of a program, from a function $F$, with only a partial redundancy between $x_4 \sim y_3$ in $\phi_{2,1}$. The redundancy is partial since there is no partition in $\phi_{2,2}$ equivalent to $x_4$, however there is one in $\phi_{2,1}$.

proposed VPF. For this to happen all the expressions in that basic block and all relevant predecessor basic blocks must be fully evaluated, at analysis time. This requires tracking the progress\(^4\) of the evaluation of basic blocks to decide if a value number is present or absent from specific basic blocks, which would cause a non-null or null VPF to be inferred.

The procedure for finding a VPF will use SOUFFLE\(^\text{stable}\), presented in the previous chapter, to allow examination of the previous basic blocks with negation allowing for an efficient check of value-number absence. The difference between finding a VPF with and without SOUFFLE\(^\text{stable}\) will be illustrated at the end of this section after the search procedure is described. The equivalence of two value-number expressions $e_1$ and $e_2$ can be written as $e_1 \sim e_2$. The program seen in figure 5.1 is an excerpt from a function with an argument $\text{arg}$. There is a redundancy between $x_4 \sim y_3$ in $\phi_{2,1}$, but not between $x_4$ and $y_5$.

\(^4\)Which expressions have been evaluated, and which partitions’ sets have been intersected.
This is found by noting that the VPFs of both operands of the addition assigned to \( x_4 \) are \( y_6 = \phi_1(y_2, y_4) \) and \( x_3 = \phi_2(x_1, x_2) \). The equivalence between \( x_4 \) and \( \phi_2(y_3, y_4 + x_3) \) is clarified by the equational reasoning:

\[
x_4 \sim \phi_1(y_2, y_4) + x_3 \sim \phi_1(y_2 + x_3, y_4 + x_3) \sim \phi_2(y_3, y_4 + x_3).
\]  

(5.1)

However, since \( y_4 + x_3 \) is not available in \( \phi_{2,2} \) this is a partial redundance and not useful.

The VPF search procedure looks at all basic blocks that VPFs were created in, excluding any VPFs that are dominated by another. In this example \( \phi_1 \) dominates \( \phi_2 \) meaning that the VPF \( \phi_1 \) is ignored and instead \( x_3 \) is treated as a value number with a null VPF. The search for the VPF then proceeds by looking for a VPF of the form \( \phi(y_2, y_4) + x_3 \), which is equivalent to \( \phi(y_2 + x_3, y_4 + x_3) \). The search then looks for \( \phi(y_2 + x_3, y_4 + x_3) \) in \( \phi_2 \), which leads to searching for both \( y_2 + x_3 \) and \( y_4 + x_3 \) in \( \phi_{2,1} \) and \( \phi_{2,2} \) respectively. The search would find \( y_2 + x_3 \) in \( \phi_{2,1} \) but would not find \( y_4 + x_3 \) in \( \phi_{2,2} \). The expression \( y_2 + x_3 \) in \( \phi_{2,1} \) is present, which would just require that the value number for the expression is available in the basic block. The proof that \( \phi_{2,2} \) doesn’t contain the value number with the expression \( y_4 + x_3 \) requires that this expression can never be inferred in that basic block or be found redundant in previous basic blocks. This example presents two problems for a VPF-based GVN algorithm implemented in Datalog: firstly discovering that a value number is not (and cannot be) present in a basic block, and secondly ignoring VPFs from any basic blocks that dominate the chosen basic block.

The VPF search procedure using an ordered domain \( OD \) or \textsc{Soufflé}$^\text{stable}$ is now discussed, using value number associated to \( x_4 \) as the example. The search tries to find \( S \equiv v_{10} + v_7 \) in \( \phi_{2,2} \), which we can see is not available. The \( OD \) search must assert that no value-expression of the form \( S \) is available, this is done by checking that \( S \) is not initially defined in \( \phi_{2,2} \) or \( \phi_1 \) or \( E \) or either one of \( \phi_{1,1} \) or \( \phi_{1,2} \). \textsc{Soufflé}$^\text{stable}$ can be used to assert that a specific basic block has had all values already computed. This allows absence check of \( S \) in \( \phi_{2,2} \) to be done with one negation of a relation storing all available value-expressions in \( \phi_{2,2} \). This difference between \( OD \) and \textsc{Soufflé}$^\text{stable}$ in runtime is therefore drastic where the former is \( O(n) \) for \( n \) being the height of the CFG and the latter is \( O(1) \). In fact since there are multiple iterations required until the algorithm stabilises, let \( i \) be the maximum number for a given program, then the runtime of \( OD \) is \( O(n \cdot i) \), while \textsc{Soufflé}$^\text{stable}$ is unaffected by the iteration count.

This large difference in runtime means that an \( OD \) based GVN algorithm would be impractical for large programs and so is not provided in this thesis. This chapter solves these problems using techniques common in \textsc{Soufflé} and techniques presented in this thesis.
5.2 Implementation in SOUFLÉ\textsuperscript{stable}

The GVN algorithm is used to compute redundancies between variables. The algorithm has the following components:

- A system generating value numbers unique up to matching expressions.
- A system for generating value expressions from IR expressions and assigning variables to partitions, that are characterised by value-numbers.
- A system for finding which value numbers are available at program points.
- A system that uses successive iterations to converge at a fixed variable-partition assignment.
- A system to look at previous basic blocks to find expressions that are missed by the system that is used to join value-number assignments at basic-block join points. This system is intentionally conservative.

These systems are explained throughout this section.

5.2.1 Analysis relations

The main Datalog relations used throughout this analysis are shown in figure 5.3. The relations \texttt{operand\_has\_vn} (and \texttt{operand\_has\_vn\_prop}) store a mapping (and proposed mapping) from each operand to a value-number expression at a specific computation step. The \texttt{vn\_expression\_allocation} relation stores the value number assigned to each expression. Many expressions can be assigned the same value number and the choice-domain qualifier ensures one is arbitrarily picked. The relation \texttt{bb\_vn\_entry} stores value numbers that are in scope when entering a basic block; similarly the relation \texttt{bb\_vn\_exit} stores value numbers available when exiting a basic block.

The GVN algorithm will operate on IR\textsuperscript{5}; each program’s IR will be encoded in the Datalog relations in figure 5.2(a).

5.2.2 Value-number encoding and numbering

Value numbers, which define equivalent sets, are derived from the ADTs shown in figure 5.4. Value numbers are used to characterise equivalent partitions, if variables are assigned to the same value number they are in the same equivalence partition. The injective functor \texttt{ord} built into SOUFLÉ is used to look up the integer that is used to identify a specific ADT. The equivalence of value numbers is, therefore, seen globally throughout inference,
The basic blocks in a function
bb_in_fn(bb: BasicBlock, fn: Function).

The instructions in a basic block
inst_in_bb(inst: Instruction, bb: BasicBlock).

The operands in a binary instruction
inst_opr(inst: Instruction, ei: ExpressionIndex, opr: Operand).

The opcode of the binary instruction
inst_opcode(inst: Instruction, opcode: Opcode).

The variable a binary instruction assign to
inst_assigns_to(inst: Instruction, v: Variable).

The indexed predecessor pbb of a basic block bb
bb_pred(bb: BasicBlock, index: PhiIndex, pbb: BasicBlock).

(a) The Datalog relation declarations used throughout this chapter.

bb_in_fn(E, F).
bb_in_fn(ϕ₁,₁, F).
inst_in_bb(1, E).
inst_in_bb(2, ϕ₁,₁).
inst_opr(1, 1, arg).
inst_opr(1, 2, 1).
instr_opr(2, 1, x).
instr_opr(2, 2, 1).
inst_opcode(1, "+").
instr_opcode(2, "+")
inst_assigns_to(1, x).
instr_assigns_to(2, x₁).
bb_pred(ϕ₁,₁, 1, E).

(b) The EDB facts which would be generated from the first two basic blocks E, ϕ₁,₁ of the program in figure 5.1.

Figure 5.2: The Datalog relations that are populated with facts that represent the IR being analysed.

.decl operand_has_vn(
    o: Operand bb: BasicBlock, i: Iteration, vn: ValueNumber)
.decl expression_variable_prop(
    expr: Expression, o: Operand bb: BasicBlock, i: Iteration)
.decl vn_expression_allocation(vn: ValueNumber, expr: Expression)
    choice-domain(expr)
.decl bb_vn_entry(bb: BasicBlock, i: Iteration, vn: ValueNumber)
.decl bb_vn_exit(bb: BasicBlock, i: Iteration, vn: ValueNumber)

Figure 5.3: Relations used throughout the GVN computation.
Figure 5.4: The types of the expressions used that characterise a value number and thereby a register partition, expressed as a pair of SOUFFLÉ ADTs.

meaning two equal constants in different functions are given the same value number. The definitions of all types apart from PhiNode are self-explanatory. The PhiNode contains an ordered list of value numbers, one for each predecessor basic block, and the basic block it was created at. Value numbers derived from value expressions, of type PhiNode, represent the merger of many operands each coming from a previous basic block. The basic-block entry ensures that the assigned value number respects the unknown path taken to this basic block. As an explanation, consider two basic blocks bb1 and bb2 with an identical phi instruction $\phi(1,2)$ assigning to $x_1$ in bb1 and $x_2$ bb2. Then $x_1$ and $x_2$ may still not be equivalent since the path taken to bb1 might cause $x_1 = 2$ and path taken to bb2 might cause $x_2 = 1$.

Value expressions of type Expression are computed via the examination of IR-instructions, seen in figure 5.5. The relation inst_opr_vn(inst, idx, vn, iter) for each instruction (inst) stores the value assigned to each operand at the current context (inst $\in$ bb.iter), the operand is referenced by index (idx). The assignment ($t_1 = i1 t < 5$) to variable $t_1$ would produce the value expression ($\$\text{BinaryNode}(\langle t \rangle, \langle 5 \rangle, \"i32 i32 < i1\")$) inferred from the rule in figure 5.5. The notation $\langle var \rangle$ represents the value number that var is assigned to. The opcode is type sensitive⁶.

### 5.2.3 Available value-number intersection

The value numbers available at a basic block are important to the VPF search procedure mentioned shortly. The available value numbers at a basic block are added to when a variable value-number assignment at a basic block is inferred. The available value-number set is propagated to a basic block if all predecessors agree. The algorithm for available value-number rules can be seen in figure 5.6. The notation $\forall(\hat{x}) : R(\hat{x}, \hat{y}) \rightarrow \hat{R}$ used by

---

⁶Including the expression’s return type.
expression_variable_prop(var, bb, iter, expr) :-
  inst_assigns_to(inst, var),
  inst_opr_vn(inst, 1, vn1, itr),
  inst_opr_vn(inst, 2, vn2, itr),
  inst_opcode(inst, opcode),
  inst_in_bb(inst, bb),
  vn = $Bin(vn1, vn2, opcode).

Figure 5.5: The Datalog rule responsible for creating a binary value expression from an IR expression using the previously defined value numbers assigned to the operands of the instruction inst being considered. For example this rule will discover expression_variable_prop(x, E, 1, v₀ + v₁), when considering figure 5.1.

G. Balatsouras et al. [6] is syntactic sugar, asserting that all instances of the variables \( \hat{x} \), and previously bound variable \( \hat{y} \) that satisfy \( R(\hat{x}, \hat{y}) \) that \( \hat{R} \) holds. In the specific example that is asserting that all predecessor basic blocks pbb and for the same (or previous\(^7\)) iteration piter that operand_has_vn is satisfied. This is implemented in the Soufflé as three relations check each instance of \( \hat{x} \) hold. This happens by first checking the first instance in the ordered domain hold, then checking \( i + 1 \)th instance assuming that the \( i \)th instance holds, and finally checking that list instances of \( \hat{x} \) hold. Therefore, each instance holds.

5.2.4 Handling cycles in the control flow graph

The approach taken in this work to handle any cycles in the control flow graph is to apply the inference rules for multiple iterations successively until the partitions between successive iterations are unchanged. Any cycle in a CFG must contain one or more back edges, which, if removed, would break the cycle. A back edge can be found by looking at the program’s dominator graph.

A relation basicblock_transfer is used to track which basic-block edges change, or do not change, the iteration index depending on if it is a back edge or not.

The initial configuration used at iteration zero assigns each variable to the same partition, named top \( \top \). This causes a default variable assignment for every back edge entering any basic-block in the first iteration. The \( \top \) variable is included via the extension to the value-expression types, seen in figure 5.8. The intersection of basic blocks in figure 5.6(b) is modified to consider variables being transferred across a back edge at the first iteration to match any \( vn \). The intersection of distinct value numbers with a shared variable will also allow variables assigned to the partition \( \top \) to be included in this newly created value expression with a PhiExprList\( T \) entry.

\(^7\)If basic block is accessible from a backedge.
bb_vn_entry(bb, iter, vn) :-
    operand_has_vn(var, vn, iter),
    inst_assigns_to(inst, var),
    inst_in_bb(inst, bb).

(a) Value-number availability introduction. To explain the motivating example from figure 5.1 is considered, ME: bb_vn_entry(E, 1, v4), since the variable is linked to the basic block and operand_has_vn(x, v4, 1).

bb_vn_entry(bb, iter, vn) :-
    basicblock_iter(bb, iter),
    (∀(pbb, piter):
        bb_transfer(bb, iter, pbb, piter) →
        bb_vn_exit(pbb, piter, vn)).

(b) Value-number availability intersection. ME: bb_vn_entry(ϕ1, 1, v4), since (pbb, piter) range over (ϕ1, 1) and (ϕ1, 2, 1) and also v4 is in both of those predecessors, encoded in bb_vn_exit.

bb_entry_var_fold(bb, iter, var, index, vpf) :-
    bb_pred_count(bb, index),
    bb_var_vn_entry_index(bb, iter, var, vn, index),
    vpf = $PhiExpreListN(vn, $PhiExprListE()).

bb_entry_var_fold(bb, iter, var, nindex, vpf) :-
    nindex = index - 1,
    bb_entry_var_fold(bb, iter, var, index, rest),
    bb_var_vn_entry_index(bb, iter, var, vn, nindex),
    vpf = $PhiExpreListN(vn, rest).

bb_vn_entry(bb, iter, vn) :-
    bb_entry_var_fold(bb, iter, var, 1, vpf),
    phi = $VPhiNode(vpf, bb).

(c) Value number availability intersection when a variable is a member of different partitions at basic-block predecessors. The value-number variable assignment for each predecessor by index is stored in the bb_var_vn_entry_index relation. ME: bb_vn_entry(ϕ2, 1, v12) since bb_entry_var_fold(ϕ2, 1, y6, 2, φ(v4)), and then bb_entry_var_fold(ϕ2, 1, y6, 1, φ(v2, v4)) hold.

Figure 5.6: Handling value-number availability.
```prolog
decl basicblock_transfer(bb: BasicBlock, iter: Iteration, 
    from_bb: BasicBlock, from_iter: Iteration)

basicblock_transfer(bb, iter, pbb, iter) :-
    bb_pred(bb, pbb),
    !back_edge(bb, pbb),
    bb_iteration(bb, iter).

basicblock_transfer(bb, iter, pbb, piter) :-
    bb_pred(bb, pbb),
    back_edge(bb, pbb),
    bb_in_fn(bb, func),
    next_iteration(func, piter, iter).
```

Figure 5.7: The rules for computing iteration transfer between basic blocks.

```prolog
.type PhiExprList =
    ...
    PhiExprListT { rest: PhiExprList } |
    ...
```

Figure 5.8: The alteration of the core types, in figure 5.4, to include $\top$ in a $\phi$-list.

5.2.5 VPF

The algorithm for value assignment, as currently stated in this section up to this point, would miss equivalences that happen as a result of $\phi$-instructions merging variables that are dependents of matching expressions in predecessor basic blocks, seen in figure 5.9(a). The dashed lines (in figure 5.9(b)) show an expression calculation that could be replaced with a copy (in the form of a $\phi$-instruction), specifically $z_2 + z_1 \equiv \phi_{bb3}(x_1, x_2)$.

The VPF search tries to answer the following question: does this expression have a related VPF-expression matching or not? This is answered in two relations `expr_vpf` and `expr_vpf_null`, which are then used to assign value numbers to variables in a specific context, see figure 5.10.

The search procedure presented shortly is a declarative adaptation of work by Pai et al. [38]. This original procedure’s pseudocode (summarized in figure 5.11) omits similar cases to aid clarity, but pays extra attention to which basic block the procedure is considering at each point, which was overlooked by Pai et al.’s presentation of the algorithm.

The declarative implementation computes the same VPF for an input expression as the imperative search (figure 5.11), an example of this search can be seen in figure 5.9.

---

The process of assigning variables and value expressions a value number.
(a) A simple example IR program with a redundancy between $z_3$ and $\phi(x_1, x_2)$.

$$
\left\{ z_3, z_2 + z_1, \phi_{bb_3}(x_1, x_2) \right\}
$$

(b) The value-partition graph of the example in figure (a). The dashed edges represent VPF links and the solid lines represent value-expression links.

(c) Search for VPF of $z_2 + z_1$.

(d) Search for VPF of $z_2 + 1$.

Figure 5.9: A simple program and its value-partition-graph.
operand_has_vn(var, bb, iter, vn) :-
expr_vpf_null(expr, bb, iter),
expression_variable_prop(var, bb, iter, expr),
vn_expression_allocation(vn, expr).

operand_has_vn(var, bb, iter, vn) :-
expr_vpf(expr, bb, iter, vpf),
expression_variable_prop(var, bb, iter, expr),
vn_expression_allocation(vn, vpf).

Figure 5.10: Allocating a variable to a value-number partition, depending on if the expression has a null or non-null VPF.

def vpf-search(e, bb, i):
    (e', bb_f, i_f) = replace(e, bb, i)
    # (1) The \( \phi_{bb_f} \) is a vpf created at \( bb_f \)
    if e' if of the form \( \phi_{bb_f}(e_{l1}, e_{r1}) \odot \phi_{bb_f}(e_{l2}, e_{r2}) \):
        e'_l := e_{l1} \odot e_{l2}
        # \( (bb_l, i_l) \) is left predecessor of \( (bb_f, i_f) \)
        vn_1 := search for \( e'_l \) in \( (bb_l, i_l) \)
        if vn_1 is null :
            #\( (bb_l, i_l) \ predecessor of \( (bb_f, i_f) \)
            vpf_1 = vpf-search(e'_l, bb_l, i_l)
            vn_1 := search for \( vpf_1 \) in \( (bb_l, i_l) \)
            same for \( e'_r := e_{r1} \odot e_{r2} \) as \( e'_l \)
            return \( \phi_{bb_f}(vn'_l, vn'_r) \) if both are none null otherwise return null
        else if e' is of a different form containing at one vpf :
            similar to above case
        else : no vpf in either expression operand :
            return null vpf

Figure 5.11: Imperative VPF search procedure. Line 3 specifies that both VPFs must have been created at the same basic block; if this is not the case then pick the dominated VPF.
expr_vpf_null(expr, bb, itr) :-
expr_not_in_scope(expr, bb, itr),
expr_index_all_null(expr, bb, itr).

(a) Checks if the expression is not in scope and all its operands have null VPFs. ME: expr_vpf_null(v_{10} + v_7, \phi_{2,2}, 1), since neither v_7 nor v_{10} have a VPFs (expr_index_all_null) and the expression is not in scope.

expr_vpf_null(expr, bb, itr) :-
expr_vpf_prop(expr, vpf, bb, itr),
vpf_null(vpf, bb, itr).

(b) Propagates null-ness from further back in the search. ME: expr_vpf_null(v_{12} + v_7, F, 1) would be inferred since v_{12} + v_7 and \phi(v_{13}, v_{?1}), (v_{?1} = v_{10} + v_7), are linked by (expr_vpf_prop) and v_{?1} is null so \phi(v_{13}, v_{?1}) is too, and vpf_null(\phi(v_{13}, v_{?1}), F, 1). This is further explained in equation (5.1) p. 82.

Figure 5.12: The rules for inferring an expression with a null VPF. The vpf_null relation tracks proposed VPFs that are, in fact, invalid.

The structure of the declarative implementation, however, uses two interacting paths of computation: for non-null- and null-VPF expressions, each step of the path is at a search context. The context comprises a basic-block iteration pair.

The search for null VPF expressions, figure 5.12, looks for, at a search context:

- Expressions that are both not available at the context and have only operands that have null VPFs, figure 5.12(a).

- Expressions with any operand with a non-null VPF, where the search path for that VPF expression at its relevant predecessor basic block returns null. See figure 5.12(b).

The search for non-null VPF expressions (figure 5.13) looks for expressions that are either in scope or have all the search VPFs in scope. The search for VPFs is conducted by distributing the expression operation \odot over the \phi-operation (see figure 5.11), this is captured by the clause expr_vpf_prop.

The decision to search for a previous VPF, analogous to recursively calling vpf-search (figure 5.11) is made when the expression under consideration is not in scope at the current configuration.

The declarative implementation must, among other things, answer the question: ‘Is a value number not present in a basic block?’. DGVN answers this using the same techniques shown in the previous chapter, namely induction and stable-companion bounded negation. The first approach was shown, in the previous chapter 4 section 4.5.2 p. 76, to have a considerably larger runtime overhead and so an implementation will not be provided. The link between the DGVN implementation can be seen as a depth-first search with the next relation being the successor basic blocks and the visited check being the assertion of
vpf_index_inscope(\phi, \text{orig}_bb, \text{orig}_itr, \text{index}) :-

expr_inscope(expr, bb, itr),
expr_from_vpf(expr, \phi, index, bb, itr),
expr_vpf_prop(expr, \phi, bb, itr),
expr_vpf_find_bb(expr, \phi, bb, itr, orig_bb, orig_itr).

expr_vpf(expr, bb, itr, \phi) :-
expr_vpf_prop(expr, \phi, bb, itr),
\forall (\text{index}) : \text{vpf_index}(\text{vpf}, \text{index}) \rightarrow
\text{vpf_index_inscope}(\phi, \text{orig}_bb, \text{orig}_itr, \text{index}).

Figure 5.13: The rules for inferring expressions with non-null VPFs. The first clause checks if the expression is in scope, if so, then \text{expr_from_vpf} looks up the VPF containing that expression and the VPF in the originator basic block (from \text{expr_vpf_find_bb}) is marked as having a vpf for that index, as in figure 5.11. The second clause finds an expression has a VPF if all vpf operands are in scope. ME: The fact \text{vpf_index_inscope}(\phi(v_9, v_{11}), F, 1, 1) is inferred since \( v_8 + v_7 \) is in \( \phi_{2,1} \), and is the first operand of \( \phi(v_9, v_{11}) \), the \text{expr_vpf_find_bb} looks up the VPF’s originator basic block (\( F \)). However, \text{vpf_index_inscope}(\phi(v_9, v_{11}), F, 1, 1) cannot be inferred and so the second clause doesn’t find \text{expr_vpf}(V_{12} + v_7, F, 1).

\begin{verbatim}
.stable bb_vn_exit(bb, i)
bb_vn_exit_f
bb_vn_exit_f(bb, iter) :-
bb_vn_entry_done(bb, iter),
\forall (pbb, piter) : bb_transfer(bb, iter, pbb, piter) \rightarrow
bb_vn_exit_f(pbb, piter).

bb_vn_entry_done(bb, iter) :-
basicblock_iter(bb, iter),
bb_vn_inst_done(bb, iter),
bb_vn_entry_done(bb, iter).
\end{verbatim}

(a) Tracking if a basic block has been fully evaluated.  (b) The rule for tracking if all basic-block predecessors are fully evaluated.

Figure 5.14: The rules for tracking value-number computation throughout the program.

value number absence. This is further discussed in section 5.3. The stable-companion-relation approach keeps track of which instructions (\text{bb_vn_inst_done}) and basic blocks (\text{bb_vn_entry_done}) have been fully evaluated and hence the value numbers for those variables have been recorded. These are recorded in \text{bb_vn_exit_f}, seen in figure 5.14(a). Intersections of value numbers between basic blocks are also similarly tracked by the rule in figure 5.14(b). The \text{bb_vn_exit_f} relation can be used to mark the stable basic-block value number assignments (\text{bb_vn_exit}) relation allowing previous basic blocks to be checked for unavailability of a known value number, figure 5.15.

The VPF search procedure can find expressions that contain operands that have value numbers assigned at different basic blocks, noted in figure 5.11 line 3. The rule to find the
expr_not_in_scope(expr, bb, iter) :-
expr_request_vn_in_scope(bb, iter, vn, expr),
bb_vn_exit_f(bb, iter),
!bb_vn_exit(bb, iter, vn).

Figure 5.15: Checking if a value number vn matching an expression expr is not available at a context—a basic-block iteration pair. ME: The clause will find expr_not_in_scope(v_{10} + v_7, \phi_{2,2}, 1), since bb_vn_exit doesn’t contain the value for the expression. The bb_vn_exit relation is updated with (\phi_{2,2}, 1) once all value numbers were chosen in all predecessor basic blocks and also all instructions have had their value numbers assigned, in \phi_{2,2}, figure 5.14.

eexpr_vpf_find_bb(expr, bb, iter, fbb, fiter) :-
expression_request(expr, fbb, fiter),
expression_index_vn(expr, _, e),
vn_vpf(e, __bb, fiter, $VPhiNode(phi, bb)).
.plan 1:(2,1)

eexpr_vpf_find_bb(expr, bb, iter, fbb, fiter) <=
eexpr_vpf_find_bb(expr, bb2, iter, fbb, fiter) :-
bb != bb2,
bb_dominates_bb(bb, bb2).

Figure 5.16: The subsumption rule applied to enforce the dominator-based VPF look up. The first rule finds all VPFs for any operand of the expression expr, then the second subsumption rule removes any basic block bb which dominates another basic block bb2. ME: The first clause infers both eexpr_vpf_find_bb(v_{12} + v_7, F, 1, \phi_{2,1}) and eexpr_vpf_find_bb(v_{12} + v_7, F, 1, \phi_{1,1}). The second clause will remove eexpr_vpf_find_bb(v_{12} + v_7, F, 1, \phi_{1,1}), since \phi_1 dominates \phi_2.

“most” dominated basic block can be seen in figure 5.16. The subsumption rule \textit{R <: R} removes any tuple from the eexpr_vpf_find_bb relation with a basic-block element that is dominated by another, when both tuples share the stable expression and originator context (fbb, fiter).

The VPF search procedure is run for each new expression being considered; under certain conditions the search result has already been computed\textsuperscript{9} and is instead looked up.

\textbf{5.2.6 Redundancies}

Redundancies are computed between a pair of variables \(v\) and \(v_r\). The redundant variable \(v_r\) can instead be replaced by a copy of \(v\)\textsuperscript{10} which removes the need for the computation of the value that is assigned to \(v_r\) and possibly intermediate operations used in computing \(v_r\). Redundancies exist in two forms: variables in the same value partition, or variables

\textsuperscript{9}And memoised by the \textsc{Soufflé} runtime.

\textsuperscript{10}Since LLVM does not have copy instructions, any reference to \(v_r\) will be replaced with \(v\).
### Table 5.1: The table shows the size of each program in the evaluation.

<table>
<thead>
<tr>
<th>Program</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGVN (NewGVN.cpp)</td>
<td>4,302</td>
</tr>
<tr>
<td>DGVNtotal</td>
<td>3410</td>
</tr>
<tr>
<td>DGVNgvn logic</td>
<td>2373</td>
</tr>
</tbody>
</table>

The DGVNgvn logic omits code which defines instruction type and other compiler logic which is not present in NewGVN.cpp.

| Program                   | $|L|$ | $|L - D|$ | $|L_{no-mem} - D|$ | $|D - L|$ |
|---------------------------|-----|--------|--------------|----------|
| parsec/blackscholes       | 45  | 20     | 0            | 0        |
| spec/429.mcf              | 91  | 62     | 0            | 0        |
| spec/473.astar            | 167 | 70     | 0            | 0        |
| spec/458.sjeng            | 1880| 1126   | 0            | 0        |
| spec/456.hmmer            | 2775| 1752   | 0            | 0        |

Table 5.2: The table compares the partition sizes found by LGVN ($L$) and DGVN ($D$). The difference columns, for example $L - D$, look for variables in partitions of LGVN that are not found in DGVN. The $L_{no-mem}$ set contains value-number variables not generated by memory accesses or their results.

5.3 Evaluation

The DGVN implementation presented in this work is compared to the scalar transformation library in LLVM\textsuperscript{11} named NewGVN (Transforms/Scalar/NewGVN.h), referred to in this thesis as LGVN. The comparison between both implementations will consider the value partitions. The comparison, however, is difficult due to differences in the resulting partitions from both implementations and the extra features present in LGVN. The size of both implementations can be seen in figure 5.1.

The purpose of the chapter was to demonstrate the effectiveness of a Datalog GVN implementation. Due to the vast performance difference between an ordered domain and \textsc{Soufflé}\textsuperscript{stable} implementation of the graph search in chapter 4 an ordered domain value number absence implementation is omitted. The performance difference would likely be even more pronounced since the absence of value number implementation, must check each instruction in all predecessor basic blocks and previous iterations for a value number assignment, instead of the single check possible in the \textsc{Soufflé}\textsuperscript{stable} implementation.

\textsuperscript{11}Git tag llvmorg-8.0.1.
LGVN is more fully featured than DGVN, since it is an industry-strength compiler developed over many years, and, while it would be possible, it would not be feasible to include all the necessary features in DGVN to catch up with LGVN. The features missing in DGVN are a more complete handling of memory accesses and a symbolic reduction of expressions present in value partitions. DGVN currently, pessimistically, assumes that each memory read instruction results in a variable assigned to a singleton partition. This is since there was no openly available must-alias analysis in Datalog for low level virtual machine (LLVM) IR and the alias analysis in LLVM could not be exported ahead of time since the set of pointers queried in the DGVN analysis cannot be enumerated before execution. The LGVN algorithm uses alias analysis and a simple data-flow analysis to assign accesses to the same partition if possible. The LGVN implementation checks that both memory accesses must point to the same location and have no possible interleaving write between them in execution order to that location. The symbolic reduction allows some equivalent, but syntactically distinct, expressions to be found, which allows the partition containing those expressions to be merged, possibly leading to more redundancies being discovered.

LGVN uses a sparse representation of the value partitions whereas DGVN uses a dense representation, which makes a partition comparison difficult. Table 5.2 shows the difference in value partitions found when examining the same program with LGVN and DGVN, and the difference when excluding memory accesses. Memory accesses are excluded by
\[ y_1 + 1 : \text{if.en9} \]
\[ = \phi(\text{add : if.tn}, \text{add}_4 : \text{if.tn3}, \text{add}_7 : \text{if.el6}) + 1 \]
\[ = \phi(\text{add} + 1 : \text{if.tn}, \text{add}_4 + 1 : \text{if.tn3}, \text{add}_7 + 1 : \text{if.el6}) \]
\[ = \phi(\text{add}_1 : \text{if.tn}, \text{add}_5 : \text{if.tn3}, \text{add}_8 : \text{if.el6}) \]
\[ = \phi(\text{add}_1 : \text{if.tn}, z_0 : \text{if.en}) \]
\[ = z_1 : \text{if.en9} \]

Figure 5.18: The equivalence between \( y \cdot 1 + 1 \) and \( z \cdot 1 \).

excluding variables in a set; the set is computed by ignoring all variables in partitions that result from a load or call instruction or from instructions using a variable from that set. The sum of extra elements in a partition is calculated by taking the sum of the size of each partition while excluding one element from each partition. The partitions in LGVN are compared to, possibly, many partitions taken from the DGVN result, which are merged by considering computed redundancies to allow a comparison between a dense and sparse GVN-partition representation.

### 5.3.1 Equivalences found by DGVN and missed by LGVN

The unit tests used to check the correctness of DGVN were also checked by LGVN and the resulting value partitions from both were compared. The DGVN analysis finds all the redundancies present in LGVN, however some equivalent variables were found by DGVN that were not found by LGVN. They are named: triple-join and follow-loop. They are not found by LGVN due to the depth-bounded VPF search procedure used. The depth bound must have been set to zero in the LGVN implementation, since the
equivalences can be seen after a single step backwards in the VPF search. This bound stops the search altogether which appears to be incorrect since a large portion of the LGVN implementation is the VPF search procedure.

The triple-join program can be seen in figure 5.17, which has a partition containing $\text{add}_{10}$ and $z_1$; this follows from the equivalence presented in figure 5.18. This analysis result allows the replacement of any use of $\text{add}_{10}$ with a use of $z_1$, hence leaving the unnecessary $z_1 + 1$ computation as dead code.

The follow-loop has two induction variables that are always one value apart. DGVN finds that $\text{inc}$ and $x_0$ are in the same value partition, whereas LGVN does not. This result allows a CSE transformation to remove the expression $\text{inc} = i_0 + 1$ and replace $\text{inc}$ uses with $x_0$ uses.

### 5.3.2 Execution time

The programs compared are taken from PARSEC-3.0 [9] and SPECint2006. There is also a suite of unit-tests omitted from these comparisons but considered later in the evaluation.

The runtime of both passes is compared for each program considered. The runtime of LLVM’s GVN (LGVN) is taken from five runs of the `opt` tool with the `-time-passes` flag\(^{12}\). The runtime of the DGVN implementation was measured from the median of five runs of the interpreter by taking the difference of the time before and after the program was evaluated. The measurements were taken using `std::chrono::system_clock::now`. This shows that the DGVN implementation is slower than LGVN, however the latter technique uses a more complex, and less complete algorithm. The order of magnitude slow-down in figure 5.3 is due to a number of factors: the dense representation of value partitions, the overhead of non-GVN related rules in DGVN (that are not in LGVN) and the exact evaluation scheme of SOUFFLÉ where all the work is carried out in one big semi-naïve evaluation loop. The non-GVN rules include simple control-flow analyses and the time taken to load relations from disk, since the work required to fill the relations from the in-memory LLVM IR data structure was not undertaken in the DGVN proof of concept. These factors can all be considered in future work.

\(^{12}\)The whole command was `opt -S -disable-basicaa -newgvn -time-passes`.
<table>
<thead>
<tr>
<th>Program</th>
<th>LoC</th>
<th>LGVN</th>
<th>DGVN</th>
<th>Relative runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>parsec/blackscholes</td>
<td>509</td>
<td>0.02</td>
<td>0.11</td>
<td>5.5</td>
</tr>
<tr>
<td>spec/429.mcf</td>
<td>2,938</td>
<td>0.08</td>
<td>0.75</td>
<td>9.4</td>
</tr>
<tr>
<td>spec/473.astar</td>
<td>5,842</td>
<td>0.16</td>
<td>1.53</td>
<td>9.6</td>
</tr>
<tr>
<td>spec/458.sjeng</td>
<td>13,847</td>
<td>0.53</td>
<td>7.90</td>
<td>14.9</td>
</tr>
<tr>
<td>spec/456.hmmer</td>
<td>35,992</td>
<td>1.56</td>
<td>19.95</td>
<td>12.8</td>
</tr>
</tbody>
</table>

Table 5.3: The table compares the runtime of both techniques, in seconds. The passes which run when using the flag `-O3`, were run before either GVN pass was run. These passes can be seen in appendix B.

5.4 Summary

This chapter presents a declarative implementation of the GVN algorithm, requiring considerable modifications from previous work, using new language extensions to make this performant. The chapter shows the real-world usage of SOUFFLÉ\textsuperscript{stable} and its ease of use in this larger program. DGVN finds redundancies missed by LLVM—an industry-leading compiler. This implementation shows only an order of magnitude decrease in performance which is acceptable for the proof of concept and when compared to the mature LLVM optimisation system.

The succinctness of the declarative implementation (DGVN) is made possible by the SOUFFLÉ\textsuperscript{stable} extension; without this DGVN would need to be structured differently to avoid the larger performance penalty that comes with an inductive implementation. This would require changes in structure that would necessarily lead to a loss of succinctness relative to previously published work.

The succinctness of Datalog becomes ever more apparent in the next chapter where it is embedded into an annotation language, forming a technique called SEMSET. The terseness of the syntax makes Datalog the ideal candidate to be embedded.
Semantic set annotation analysis

Transforming program statements to exploit parallel hardware is a well-researched topic and the option to commute\(^{1}\) instructions can offer greater flexibility in transformation. This can vastly improve the efficacy of parallelising transformations.

Commutativity is a symmetric non-transitive relation operating on pairs of instructions or blocks of instructions. Automatic-parallelism transformations allocate program sections to different processor cores using a communication primitive to impose a correctness-preserving dependency, ordering execution where necessary. Commutativity can be relied on to guarantee safety when swapping the execution order of two instructions leading to a greater choice of execution orders and potentially to more parallelism being exploited. Commutativity, in this thesis, is safe if the swapping of pairs of instructions leads to no externally observable side effects.

This work develops an annotation language and analysis-time-inference system named SemSet, which is used to expose commutativity to the compiler, among other things. The SemSet annotations provide a user-controlled mechanism to use existing alias analysis and customisable domain-specific data-flow analyses to infer commutativity. A motivation for SemSet is presented in section 6.1, the implementation details are in sections 6.2 and 6.3, the annotations are used on multiple worked examples in section 6.4, evaluated in section 6.5 and a novel program speed-up enabled by this technique is described in the summary in section 6.6. Annotations enrich the program dependency graph (PDG) [21] edges that allows for composable and reusable annotations, which can be safely applied in a gradual manner. The annotations can be applied one by one without leading to unsafe analysis results.

Previous techniques that make use of annotations to improve program performance include CommSet [39], Paralax [56] and VELOCITY [12]. The previous works all use different commutativity annotations, however they are applied at the function-definition level, with the semantics of allowing different execution orders of functions in the same

\(^{1}\)Permute the execution order.
commutative grouping. PARALAX and VELOCITY allow only groupings of size one, meaning that only call instructions with the same target function that are marked as (self) commuting can be found to be commutative. PARALAX also defines other annotations: MOD and REF, which exist in this work as the annotation function_access (in figure 6.3). COMMSET allows for code sections to be included in commutative sets. For any pair of elements in a commutative set it is safe to commute their execution order. Elements can also be included conditionally. VELOCITY is a framework for program parallelisation, which defines a commutative annotation also useful for speculative execution environments.

SEMSET extends commutativity annotations beyond what is possible in either PARALAX or VELOCITY, whereby commutativity conditions are applied at the function definition and also at the function call site where necessary. SEMSET differs from COMMSET [39] by operating at the sub-instruction dependency-pair level, allowing for composable annotations while making use of simple flow analyses to allow for reusable annotations. COMMSET annotations cannot be applied to library function meaning that the annotation cannot be reused between programs, even if they have identical fragments. The commutative set in SEMSET can also be used to capture the semantics of stateless found in PARALAX (defined by Vandierendonck et al. [56] to mean a “function does not maintain internal state”).

This work begins by outlining the possible causes for dependencies between pairs of instructions to motivate the consideration of dependencies, see section 6.1. Dependencies between instructions can be caused by either function arguments aliasing or possible side effect conflicts. These per-argument dependencies can be found to be either: happens-before, commuting or independent. Originally program dependency graphs (PDGs) have unlabelled edges between two nodes that are used to signify a happens-before relation from the source node to the destination—a constraint that the source must execute before the destination to guarantee safety. The independent dependency edge type signifies from this work no dependence in the original formulation. The commuting edge type, a new undirected edge, enforces a choice of either a happens-before relation from the source to the destination or from the destination to the source. The dependencies considered by the SEMSET annotations examine memory locations derived from alias analysis, which allows for a precise tracking of important program properties across whole programs.

The notion of the explicit description of pairwise commutativity, while powerful, requires tight coupling between the statements analysed and requires a quadratic number of annotations to express. Commutativity is instead lifted into sets, using pairwise set membership to signify commutativity—each pair of distinct elements in a (commutative) set is syntactically commutative. Set membership will be predicated on other program

---

2 Requiring that the source be executed before the destination.
3 Requiring that the source or destination can happen in either order, but not in parallel.
4 Requiring no scheduling constraints.
properties, such as variable uniqueness\(^5\). These program properties will also be lifted into sets and a set-membership flow analysis is used to specify each property.

Contexts, used in interprocedural alias analysis as an approximation to program execution, are attached to property and commutative sets to distinguish their members at different program positions. Contexts are also created to capture distinct loop bodies, increasing the precision of alias analysis and commutativity analysis at loops that look amenable to a speed-up from automatic parallelisation.

The output of this analysis is a context-sensitive program dependency graph extended with commutative edges and cross-iteration loop dependencies. The PDG is derived from alias analysis and commutativity analysis merged in this way:

- first, for each pair of instructions all dependencies between a pair of arguments, one from each instruction, is merged,
- then each dependency is aggregated to give a dependency type between those two instructions,
- finally, user-defined functions are summarised at their call sites by a similar aggregation across instruction instead of per instruction argument.

This work provides the following contributions:

- A system of fine-grained commutativity specification and aggregation to infer whole-program commutativity.
- A reusable library of annotations to expose important program commutativity.
- An extensible inference system allowing this framework to infer the commutativity of a wide range of program forms.
- A composable annotation language to express commutativity augmented with alias analysis information.

The \textsc{SemSet} analysis is evaluated by being applied to three worked examples:

- \textsc{md5sum}—a program used to compute and display the MD5 hash of each file given as input.
- \textsc{dedup}—used to reduplicate and decompress deduplicated-compressed files (from \textsc{Parsec-3.0} [9]).
- \textsc{potrace}—used to convert bitmaps to vector graphics.

\(^5\)E.g. a hash map’s put operation commutes with another put operation if both keys are distinct.
All programs are written in C and then translated to LLVM IR [35] for analysis. These programs use various libc functions, which are also analysed using the techniques presented in this chapter. The declarations of the libc functions are annotated and analysis run, finding that large numbers of these programs’ data dependency graph’s edges are commutative. These programs are then manually parallelised\(^6\) using OPENMP [18, 3] annotations and PTHREADS [13], leading to significant performance increases.

### 6.1 Motivating example

Real world programs make use of external procedures that are linked after compilation, inhibiting program analysis from viewing the body of these procedures. This opaque procedure body requires any analysis to assume pessimistically that the procedure will interact with, and therefore not commute with, any other procedure.

The program in figure 6.1 contains well-known libc functions `read` and `open` (labelled as `read1`, `read2`, `open1` and `open2` to aid explanation), which are both called twice. An optimising compiler might wish to try to run these calls in parallel or to commute the sequential order of the calls to better achieve its performance goals. This may or may not be safe depending on the arguments applied (`fd1`, `fd2`, `buf1` and `buf2`) to each function at its call site. Each pair of instructions have different commuting properties:

- The two calls to `open` commute with each other conditional on the oflags that are applied to each function. If both `open` calls have only `O_RDONLY`, `O_WRONLY` or `O_RDWR` flags then they commute. If for example `open1` had the oflags `O_CREAT|O_EXCL` and `open2` had the oflags `O_CREAT` there might be a conflict if their file names resolve, after following symbolic links, to the same name. This conflict inhibits commutativity since `open1` expects the file not to exist, which could not be the case if `open2` runs first.

- The two calls to `read` commute if both `bufs` do not alias and if the file descriptors (`fd1` and `fd2`) are different\(^7\).

- Each pair of calls to `open` and `read` commute if the returned file descriptor from the open call is not the same as the one passed to read, and specific oflags are not passed\(^8\).

These commutative properties could not have been captured by VELOCITY or PARALAX since their annotations cannot express procedures with commutativity annotations predicated on their arguments. This type of commutativity has previously been captured

---

\(^6\)Since this is out of the scope of this work.

\(^7\)Both referencing different open file descriptors.

\(^8\)For example `O_TRUNC` but this is out of the scope of this discussion.
int open(const char *path, int oflag, ...);
ssize_t read(int fd, void *buf, size_t nbyte);

... // buf1 and buf2 do not alias

ℓ₁ fd₁ = open(path₁, O_RDONLY); // open1
ℓ₂ fd₂ = open(path₂, O_RDONLY); // open2
ℓ₃ read(fd₁, buf₁, M); // read1
ℓ₄ read(fd₂, buf₂, N); // read2

Figure 6.1: A program that interleaves the opening of and reads from two files using externally linked libc functions, which have well-defined specifications. This specification is \( fd₁ \neq fd₂ \wedge buf₁ \neq buf₂ \Rightarrow \text{reads are commutative and both oflags are O_RDONLY} \Rightarrow \text{opens are commutative.} \)

in COMMSET by annotating all call sites of the read and open functions with annotations denoting commutativity predicated on context-specific in-scope variables when required. To exploit the commutativity in this example using COMMSET, four commutative-set inclusions would be required and none of these annotations would be transferable to other programs using the same procedures. However, SEMSET overcomes these limitations.

To fully expose the commutativity in the example program (figure 6.1), in a composable and reusable way, all possible dependencies between these pairs of function declarations are considered: \((\text{read}, \text{read}), (\text{open}, \text{open})\) and \((\text{open}, \text{read})\). For \((\text{read}, \text{read})\) the considerations are non-aliasing \textbf{bufs} and the internal state of both reads being consistent irrespective of the order of evaluation, which is true if both \textbf{fds} are distinct. For both \((\text{open}, \text{open})\) and \((\text{open}, \text{read})\) to be commutative they are also both required to have consistent internal state (as in \((\text{read}, \text{read})\)), however this is always the case\(^9\).

The pairs of call instructions \((\text{open}₁, \text{read}₁)\) and \((\text{open}₂, \text{read}₂)\) have register dependencies between the return of each open function and the \textbf{fd} parameter of each read function, which produce happens-before dependencies between those instructions—unlike the commutative dependencies that are inferred between instruction pairs \((\text{open}₁, \text{read}₂)\) and \((\text{open}₂, \text{read}₁)\). Distinct file descriptors can be found by tracking them from creation (at an open call). This property is conveniently captured by assessing membership of a unique file-descriptor set whose membership is propagated throughout the program. These insights justify that the specific function calls \((\text{open}₁, \text{open}₂), (\text{open}₁, \text{read}₂), (\text{open}₂, \text{read}₁)\) and \((\text{read}₁, \text{read}₂)\) commute, while \((\text{open}₁, \text{read}₁)\) and \((\text{open}₂, \text{read}₂)\) call instruction pairs require a happens-before ordering.

The SEMSET annotations presented in this work can be used to express this file-operation commutativity in a composable and reusable way. A simplified version of annotations can be seen in figure 6.2, the full version can be seen in figure 6.12, p. 124.

\(^9\)If specific oflags are not present.
# infer member_at($return, UniqueOFDs, $loc) :- safe_flags(flags).
# infer member(($return, open, $loc), ofd_commute).
int open(const char *pathname, int flags);
# infer member((fd, read, $loc), ofd_commute).
# infer property_presered(fd, UniqueOFD).
ssize_t read(int fd, void *buf, size_t count);

Figure 6.2: The SemSet annotations used to express the commutativity of the open and read function calls in the motivating example.

The annotations, on lines 1-2, are applied to every static instance\(^\text{10}\) of a call of the open function, where parameters, such as $return, are replaced with specific arguments, such as fd1 or fd2. The file descriptor is included in two sets, UniqueOFDs and ofd_commute, the former is a set of all unique file descriptors and the latter is a set which will deduce function commutativity of the function instance (open1 or open2) with another file-open call instruction only if both return variables are distinct and in the UniqueOFDs set. The first annotation has a constraint that the open function must have safe flags, this includes the flag O_RDONLY, in order to have its file descriptor included in the UniqueOFDs set. The annotations on lines 5-6 apply to each read instance (in the above program that is read1 and read2). The inclusions of (f2, read, $loc) in ofd_commute work just as described for open. The property_preseved annotation insures that if the file description fd is in any set of unique open file descriptors before this function is run, this it is also in that set after this function.

6.2 Implementation in SOUFFLÉ

The annotation language, which is the main contribution of this work, requires the definition of many foundational concepts key to understanding the whole language. The overall purpose of the language is to infer commutativity, which is realised as PDG edges. These elements will be explained first, followed by key concepts, then the annotation language will be presented and finally used in a number of case studies in section 6.5.

Program dependency graphs \(G = (N, E)\), in this work, have nodes for each program statement \(N = \text{NODES}\) and zero or more labelled, typed, directed edges \(E = \text{EDGES}\) (from node to node) for each pair of program variables\(^\text{11}\). The set of all edges between two nodes \((n, n' \in \text{NODES})\) is named \(\text{EDGES}_{n,n'}\). The edge types are dependent (\text{dep}), commutative (\text{comm}), independent (\text{ind}) or unknown (\text{unk}). These edge types form a

\(^{10}\)This analysis precision is increase with the usage of call stack contexts, which can partially approximate dynamic call instances.

\(^{11}\)This could be further refined to look at def-use chains.
The lack of an edge between two nodes signifies no dependency and would be equivalent to an independent edge.

Commutativity of program statements $S, S'$ under sequential execution is defined by $S; S' \equiv S'$; $S$, meaning that any starting configuration of memory $\phi$ and application execution of a program $[S; S'](\phi)$ having a resulting configuration $\rho$, would have the same configuration $\rho$ if the same configuration $\phi$ is a starting state and the program $[S'; S](\phi)$ is executed

$$[S; S'](\phi) = \rho \land [S'; S](\phi) = \rho' \Rightarrow \rho = \rho'.$$

Commutativity can be applied multiple times to a program, for example $S; S'; S'' \equiv S'; S; S'' \equiv S'; S''; S'^12$. The atomicity of operations is important when considering commutativity in combination with concurrency transformations. Atomic operations cause a series of externally observable side effects to appear as one indivisible action. This means no other program could observe some but not all of these side effects if run concurrently. This atomicity property cannot always be assumed, leading to a distinction in types of commutativity, and an extra program-dependency type `comm-sync` along with the already mentioned `comm` type (referred to as `comm-nosync` to highlight the distinction). For example a hashmap update operation might not be atomic, so it must be marked, in a dependency graph, as `comm-sync` in relation to another update, not `comm-nosync`, if both updates occur with different keys. `comm-nosync` could be used if the hash map's update implementation was already atomic. All `comm-sync` dependency types are also `comm` types. The extended lattice ($\text{dep} \leq \text{comm-sync} \leq \text{comm-nosync} \leq \ldots$) is used in the merger of dependencies (described in section 6.3.1). A `comm-nosync` label means that operations are already atomic, and it is unnecessary for a transformation to ensure mutual exclusion. This label is further omitted (until the evaluation section 6.5) for simplicity.

### 6.2.1 Key concepts

The key concepts are presented to provide the background needed to define the annotations in figure 6.3.

#### 6.2.1.1 Syntactic analysis variables

The C programming language is augmented with pseudo-variables referenced within program-embedded SemSet annotations. Each call instruction (``call f(V_1, \ldots, V_n)``) will be augmented with another variable $V_f$, which is an analysis-only annotation variable (which can also be accessed in an annotation variable at the function definition or call site, see figure 6.13, in section 6.4.1). The variable $V_f$ is used to capture the possibility of its function $f$ having side effects independent of the arguments passed to $f$, namely

---

12Other equivalences exist.
\((V_1, V_2, \ldots, V_n)\). This can be seen as a single abstract memory location which represents the union of all abstract memory locations accessed inside that function, say \(f\). This could be accessing memory accessible to another function or having an observable side effect shared with another function. The state variable \(V_f\) will cause further data-dependency edges to exist in a data dependency graph stopping the reordering of conflicting function calls. Take, for example the program:

```plaintext
1 f(V);
2 g();
```

with distinct variables \(V\) and state variables \(V_f\) and \(V_g\) for \(f\) and \(g\). The PDG will contain two typed program dependency graph edges from the node at line 1 to the node at line 2 labelled with \((V, V_g)\) and \((V_f, V_g)\). The edge types are dependent on the semantics of \(f\) and \(g\).

### 6.2.1.2 Abstract memory locations

The memory locations accessible via pointer variables are tracked using abstract locations commonly used in may-alias analysis. These locations are created by an alias-analysis pass. For example an \texttt{alloc} instruction in LLVM IR would cause the creation of an abstract location that represents the memory region that would be allocated by that \texttt{alloc} instruction. Pairs of locations are used in the construction of the program’s dependency graph, where two instructions referencing the same location would have a dependency edge, since there is an alias relationship. These abstract memory locations are used throughout the \texttt{SemSet} analysis to track the potential side effects of any function calls with a new state abstract location created at every call site of the function that is pointed to by the state variable associated with that instruction, called \textit{function state allocations}. They are returned, in a contextualised form defined in section 6.2.1.5, when an annotation variable of pointer type is dereferenced (discussed further in section 6.2.2).

### 6.2.1.3 Call-graph-sensitive context

Interprocedural analysis can increase its precision by considering all possible paths in a call graph leading up to the function under consideration. In general, these paths can be unbounded and due to computational constraints finite paths are considered, which are found by truncating the path. Truncation can cause multiple distinct paths with common postfixes to be considered as the same context, leading to a decrease in analysis precision. These contexts are used in \texttt{DOOP} [11, 8, 2] an implementation of alias analysis in Datalog [17, 50], this work is built upon \texttt{CClyzer} the LLVM IR version of \texttt{DOOP}, see section 6.5.1, p. 136.
6.2.1.4 Loop-iteration context

The call graph contexts are extended to understand loop bodies. This is enabled by creating an extra pair of contexts when entering a loop body, which represent two distinct loop-body iterations. For the purpose of context creation the analysis acts as if the loop body is replaced with two function calls to the same body with different names, which would mean these two contexts existed:

\[[\text{main:34, f:loop:2:i, f:52}], [\text{main:34, f:loop:2:j, f:52}]\]

at the entry to a function called at line 52 in \textit{f}, which capture being in different loop iterations. These loop-iteration-context extensions can find memory allocations inside a loop body as non-aliasing and distinguish between the same call instruction dynamically over two distinct loop iterations. The loops to be analysed with the loop iteration context are specified by an analysis parameter.

6.2.1.5 Contextual constructs

\textit{Contextual instructions} are defined as a context and instruction pair \((c, i)\). This is an approximation to a dynamic instance of the instruction at a set of program paths suffixed by a context \(c\). \textit{Contextual basic blocks} are defined similarly to contextual instructions. \textit{Contextual abstract memory locations} \((c_a, a)\) are similarly defined, however the context \(c_a\) is an allocation context, which is created at the point of the allocation as creation by examining the context at that position. These contextualised constructs allow the analysis to approximate dynamic instances of the constructs making a possible distinction between the same instruction accessed by different call locations to the function containing the instruction.

6.2.1.6 Lifting binary relations

\textit{SEMSet} tracks properties\(^{13}\) of memory abstractions, program registers and instructions. Consider the binary relation \(R\) representing uniqueness of program registers \(t, t'\), with registers \((t, t') \in R\) to mean that \(t\) and \(t'\) are unique (not equal). A relation \(R = \{(u, v), \ldots (u', v')\}\) of pairs can instead be encoded as multiple \(R\)-property-sets \(S^R = \{S^R_1, \ldots S^R_n\}\) where all registers in \(R\) satisfy the constraints:

- Each pair in \(S^R_i\) is in \(R\)

\[(u, v) \in S^R_i \Rightarrow (u, v) \in R.\]

\(^{13}\)Which are contextualised.
• For each pair in $R$ both elements are common to at least one set in $S^R$

$$(u,v) \in R \Rightarrow \exists S^R_i \in S^R, (u,v) \in S^R_i.$$

This means each set in $S^R$ forms a clique in a graph with an edge set $R$. This allows for the efficient specification of $R$, if there are $n$ registers in which each pair has a property $R$, then they can be added into a set $S^R$ using $n$ set inclusions instead of $O(n^2)$ pairwise inclusions into $R$. Using set membership it is not necessary for each element to have knowledge about the other elements, which is a benefit for procedure decoupling. This technique can be seen in the header file in figure 6.9, section 6.4.1.

6.2.1.7 Commutativity as set membership

SEMSET exposes a commutative set type comm, allowing for many commutative sets to be declared. This set type is used in section 6.4.1 in figure 6.10 at line 24. These sets encode whole-program commutativity $P$ by lifting this property into many $P$-property sets one for each declared set. These sets are used internally to type the dependency edges in a program’s PDG. The sets can store contextual instructions $i$ or pairs of a contextual instruction and a contextual abstract memory location $(l,i)$. All pairs of elements in these sets commute. Any pair of instruction-location pairs $(l,i), (l',i')$ in a commutative set implies the dependency edge between $(i,i')$ for variables $(l,l')$ will be typed commutative. A pair of instructions in a commutative set implies that an edge $(i,i')$ of type comm in the non-edge-labelled dependency graph overrides edges discovered from summarisation of the memory-location-typed dependency graph. Since these pairs are contextual they approximate dynamic invocations of the instructions and can be used to infer the commutativity of the same instruction on multiple loop iterations.

6.2.1.8 Function effects

Function calls have side effects captured by state variables. When there is a dependency between two functions due to common side effects there must be a mirrored dependency between the state variables of those two function calls. Optional effects are used to capture this. An effect is a nominally equivalent label attached to a function via an annotation. If no effect is assigned to a function it is assumed to clash with all other effects, meaning it has all effect types. Two functions are commutative if their effect set is disjoint. For example, the function fopen and the function time commute since they have non-overlapping effects namely eff(fopen) = \{fs\} and eff(time) = \{time\}. These effects are specified using the annotation function_effect in figure 6.3.
Set annotations are used to provide a user-controlled mechanism to infer commutativity leveraging alias analysis and other domain-specific data-flow analyses. The SEMSET annotations are included in the program code inline and are given in figure 6.3. These annotations are preceded by a hash (#), similarly to C-style pragmas. The specific relations have a shorthand and a relation in brackets, shown in variable deference (\{(xl = \ast x)\} (deref(x, xl))). The example shorthand \{xl = \ast x\} is syntactic sugar for deref(x, xl), meaning that R(\ldots, \ast x, \ldots) is equivalent to deref(x, xl), R(\ldots, xl, \ldots). This
has the meaning that \( R \) operates on the variable pointed to by \( x \), named \( x_d \). The scope, at which a top-level annotation is applicable is shown beside the rule head (e.g. \([S_g]\)); if none is given then the rule is applicable to all scopes. An annotation’s scope can be global \( S_g \), attached to a function declaration \( S_f \) or attached to a statement or block \( S_s \). Some annotations can exist in all scopes \([*]\). The annotation will be attached to the language construct directly below its placement. The three types of annotation scopes are used in the following ways:

- \( S_g \)—global-scoped annotations are used to define sets, types, and set-propagation rules.

- \( S_f \)—function-scoped annotations are used to assert semantic properties of a function, which are used to enrich every call site of this declaration; a common use case would be to insert the annotated function parameters into previously declared sets.

- \( S_s \)—statement-scoped annotations allow infer annotations, which act on statements or blocks and allow access to specific variables.

The brackets \( \{X\} \) and \( \langle X \rangle \) are used to mark an argument as optional or inferred respectively. Inferred arguments are inferred using the placement scope. An annotation attached to a function declaration will apply to each call site of that function in a program, with any function parameters substituted for call-site arguments, enriching the call instruction. An annotation referencing a programming-language variable in a block or at the call site of an annotation-function declaration will be contextualised.

The infer annotation is used to infer new set members using user-defined rules in the form of horn clauses, as used in Datalog with the SOUFFLÉ [50] dialect. The infer annotations will have access to many built-in relations (section 6.2.2.2) and relations created as a side effect of other annotation rules.

The set-type construct declares a family of sets, which the inference system may use to apply rules to sets of a specific type. The set-type name can also be used in the infer rule to look up an existing set (or create a new set) from the input variables supplied (in the \( S_b \) section of grammar). A declared injective set lookup will find the same set only if the input variables are the same. The optional argument \( Opt_T \) can be used to modify the semantics of the set type in the following ways. The propagate_all marker, if applied, would mean the contents of each set of this type is used in a data-flow inference system described in section 6.3.2. The optional input format \((x_0, \ldots, x_i, \ldots, x_n)\) specifies that the members are named tuples (if not included, a single unwrapped tuple is used). The unique\((x_i)\) tag will partition a set with members as tuples \((x_0, \ldots, x_i, \ldots, x_n)\) into all the maximal partitions where all elements are distinct up to \( x_i \). The tag also includes the set in the set_partition relation, relating sets and their maximal partitions.
The set construct declares a globally unique set with name $S$ and type $T$. Sets can be of many types, including built-ins (e.g. pairwise commuting $\text{comm, ind}$).

The constant annotation creates an immutable global variable named $V$ with a type $T_V$. A variable of type ref will make $V$ be a pointer to another abstract memory location accessible via $\texttt{deref}(V)$ or $\ast V$.

The language includes predefined relations that are applicable only to function declarations ($P_f$). The $\texttt{function_accesses}$ annotation informs the analysis of the access specifications of the variable $x$. The $\texttt{function_no_capture}$ annotation is used to assert that the function has no access to $x$ on a subsequent call (the internal state of the function did not keep a reference to $x$). The $\texttt{function_capture_set}$ is used to specify that an argument is captured by the internal state of the function. The set contains all functions that could access the captured argument. If no set is given as an argument then an unnamed singleton set is assumed. The $\texttt{property_preserved}(x, T, \langle f \rangle)$ annotation asserts that a function $f$ does not cause the removal of a variable $(x)$ from the set $(S)$ that is typed $(S : T)$.

The annotation $x \in S$ or $\texttt{member}(x, S)$ is a relation between elements $(x)$ and the sets $(S)$ they are contained in. The annotation $\ast x$ finds zero-or-more abstract locations that $x$ may point to. The $\texttt{set_partition}(S_p, S)$ makes the set partitions $(S_p)$ of the set $(S)$ accessible in this relation. The $\texttt{set_type}$ relation makes accessible the type information of a set, as defined by a $\texttt{set-type}$ annotation.

Name aliasing written $x/y$ means a variable $x$ can also be referred to using the name $y$. The variables available to annotations vary in different scopes:

- $S_f, \texttt{anno-free-vars}(f_{V_f}(V_0, \ldots, V_n)) = \{V_f/\$\texttt{state}, V_0/\$0, \ldots, V_n/\$n, \$\texttt{return}, \$\texttt{args}\}$, $\$\texttt{args}$ is a placeholder for all the function arguments.

- $S_g, S_f, S_s$, all names declared inside a set or constant annotation.

- $S_s$, contextualised variables in scope in a function body after the line that this annotation is attached to.

- $S_f, S_s, \$\texttt{loc}$ is an annotation variable that represents a contextual instruction. In the case of the $S_s$ scope the variable matches the instruction the annotation is attached to. In the case of the $S_f$ scope, the variable matches each enriched call-site instruction that relates to the function declaration the annotation containing this variable was attached to.

All annotations derived from the grammar in figure 6.3, for a whole C program, are translated into a SOUFFLÉ program, which is merged into a larger SOUFFLÉ program, seen in figure 6.5 containing SEMSET’s framework logic which is finally solved. The modules in the framework are outlined in figure 6.4. The translation between annotations.
Figure 6.4: The composition of modules, represented by arrows, used in SEMSET. The Datalog program $D_p$ used to find the PDG of a program $P$. The modules are examined in sections through the chapter: annotations, extraction & contextualisation in section 6.2.2, propagation of annotations in section 6.3.2, the program representation in the previous chapter and PDG merge in section 6.3.1.
presented here, and base relations is not yet implemented. The translation would include, for example, the inference of inferred parameters \((f)\) and the translation of the variable \((\$\text{loc})\) to a scope-dependent-contextualised instruction. All program annotations used in the evaluation (see section 6.5) are specified as \textsc{SOUFLÉ} clauses derivable from the annotations presented in this paper.

### 6.2.2.1 Embedded annotation clause generation

This derivation system is not implemented, however an illustrative example is provided that makes use of the code in figure 6.1 (p. 103) and the annotations in figure 6.2. Function level annotations are applied to each call site with each available call-site context. The \texttt{infer} annotation on line 1 (\texttt{infer member_at($\text{return, UniqueOFDs, \$\text{loc}) :- safe_flags(flags).})\) would be applied to both lines (\texttt{fd1 = open(path1, O_RDONLY)}) and (\texttt{fd2 = open(path2, O_RDONLY)}). The facts inferred for an arbitrary call-site context \texttt{CALL} would be

- \texttt{member_at(fd1, UniqueOFDs, (\ell_1, CALL))}
- \texttt{member_at(fd2, UniqueOFDs, (\ell_2, CALL))}

These facts are inferred since \texttt{O_RDONLY} is in \texttt{safe_flags} and the variable substitutions are

- \{\texttt{fd1/$\text{return, (\ell_1, CALL)}/$\text{loc, O_RDONLY}/flags}\}
- \{\texttt{fd2/$\text{return, (\ell_2, CALL)}/$\text{loc, O_RDONLY}/flags}\}

The facts inferred from the second annotation are \texttt{member(fd1, open::\ell_1, (\ell_1, CALL), ofd_commute)} and \texttt{member(fd2, open::\ell_2, (\ell_2, CALL), ofd_commute)}. The atoms \texttt{open::\ell_1} and \texttt{open::\ell_2} represent the function state variables see section 6.2.1.8.

The evaluation section (section 6.5) shows how these annotations can be used to annotate real-world programs.

### 6.2.2.2 Built-in sets

The \textsc{SEMSet} framework contains many sets that are initialised before the inference is started. All the sets used in this work are defined:

- \texttt{function_state_alloc}—a set containing all the program’s function state allocations.
- \texttt{instruction_in_loop}—a set containing all tuples of (instruction, loop, loop iteration), for each instruction in a loop.
• **entry_location**—a set containing the function that is called to start the program’s execution.

• **call_instruction_fn_target**—a set containing all tuples of a function and a call instruction calling that function.

### 6.3 SEMSET analysis inference

There are two systems that exist alongside the annotations: a dependency-graph summariser and a set-propagation system. These two systems, in conjunction with the annotations, are used to infer the types of dependencies between program instructions. These two systems are described in the rest of this section.

#### 6.3.1 Building a dependency graph

SEMSET’s final output is a typed dependency graph comprised of the commutativity-analysis element of SEMSET, basic register def-use chains and memory alias analysis. All pairs of instructions are classed as either dependent, commutative or independent. Then inter-loop-iteration dependencies are found similarly to COMMSET. Two dependency graphs are built, one with labelled abstract-memory-location edges and another without, with the first being used in the generation of the second; this is shown in the refine action in figure 6.5 and expanded upon in figure 6.6. Instruction pairs that have the following form $V_r = f(V_1, \ldots, V_n)$, $U_r = g(U_1, \ldots, U_n)$ will have analysis variables $A_f = \{v_r, v_f, v_1, \ldots, v_n\}$ and $A_g = \{u_r, u_f, u_1, \ldots, u_n\}$, and variable-labelled dependency edges $D_l = \{(u, v) | u \in A_f \land v \in A_g \land \text{pointer}(u) \land \text{pointer}(v)\}$.

Dependencies $(i, j)$ are typed with labels $E(i, j)$ taken from the partially ordered set $(\text{poset}) (L, \leq) = (\{d, c, i, u\}, u \sqsubseteq d \sqsubseteq c \sqsubseteq i)$ used for edge aggregating $(\text{poset}_{\text{agg}})$, described next. The elements have the following meaning: dependent $d$, commutative $c$, independent $i$ and unknown $u$. The meet $\land$ of two types $a, b \in L$ is given by:

\[
a \land b = \begin{cases} a & \text{if } b \sqsubseteq a \\ b & \text{if } a \sqsubseteq b. \end{cases}
\]

The meet of all elements in a set of types $T = \{t_0, t_1, \ldots, t_n\}$ given by $(\land T)$ is the meet operation $(\land)$ over each element in $T$ $(\land T = t_0 \land t_1 \land \cdots \land t_n)$. Define another $\text{poset}$ $((L, \leq') = (L, i \sqsubseteq c \sqsubseteq u \sqsubseteq d))$ used for merging dependency types between constructs $(\text{poset}_{\text{merge}})$\textsuperscript{14}. There are three relations, that are aggregated to find the static dependency between a pair of analysis variables or a pair of contextualised instructions: memory

\textsuperscript{14}Notice the swapping of $u$ and $d$ in the inversion of $(L, \leq)$.
accesses from the alias analysis, commutativity dependencies and register def-use chains, where the commutativity analysis takes precedence. If the commutativity analysis finds a label other than unknown, this takes precedence over the labels from memory accesses and register def-use.

The analysis can find multiple types for a variable-labelled dependency edge, which are all aggregated using \( \land \), then all the variable-labelled dependencies \( DV_i \) for an instruction pair are merged. This merge uses an unknown propagating lattice arising from the poset \( (L, \leq') \) for merging dependency types between constructs to find the meet of all types \( \land' DV_i \) to give the unlabelled dependency edges between instructions. This edge label is overwritten by any dependency edge already in the unlabelled dependency graph, these
# set comm1 comm
# set ind1 ind
# set ind2 ind
# set dep1 dep

# infer member(x, comm1)
# infer member(x, ind1)
# infer member(f1, dep1)
# infer member($loc, ind2)
f1(x)

# infer member(y, comm1)
# infer member(y, ind1)
# infer member(f2, dep1)
# infer member($loc, ind2)
f2(y)

Figure 6.7: Example program annotations, used to explain dependence merging. These annotations are usually inferred using predicate inference, but here are inferred using inference rules such as in figure 6.10, p. 121.

edges exist from direct assertion of the instruction-pair-dependency type\(^{15}\) possibly from a commuting set with instructions as elements. A dependency between an instruction \(i\) and a call site with a body \(i' = \text{call} f\) is summarised by finding the meet of all instructions in \(i_f \in f\) with \(i, \bigwedge_{i_f \in f} E(i, i_f);\) again, the type of the edge between \(E(i, i')\) would take precedence. A similar aggregation is used for commutativity of basic blocks from instructions and single-entry regions, allowing for ad-hoc commutativity annotations of scopes as used in COMMSets.

Consider two functions \(f_1(x)\) and \(f_2(y)\). Their argument-wise dependencies are written as \((f_1 : x, f_2 : y) : \text{comm}\) to mean the argument \(x\) to \(f_1\) has a commutative dependency\(^{16}\) with the argument \(y\) in \(f_2\). The state variable of a function call \(f\) is written \(f : f\). A call instruction \(f\) without a variable refers to the call instruction itself; \((f, f') : \text{ind}\) means there is an instruction-wise dependency of type independent between \(f\) and \(f'\). These dependencies could have been found from either alias analysis results or from the inclusion of two (or more) variables in a common built-in set (such as a \text{dep} set). Consider the two functions \(f_1(x)\) and \(f_2(y)\) that have the following annotations shown in figure 6.7. These annotations will cause the following dependencies, three labelled and one unlabelled, to be inferred:

\[(f_1 : x, f_2 : y) : \text{comm}, (f_1 : x, f_2 : y) : \text{ind}, (f_1 : f_1, f_2 : f_2) : \text{dep}, (f_1, f_2) : \text{ind}.
\]

\(^{15}\)Meaning assertions about instructions take precedence over assertions about variable dependencies.

\(^{16}\)The effect the functions have on their arguments \(x\) and \(y\), in isolation, are commutative.
The dependency \((f_1 : x, f_2 : y) : \text{comm}\) could arise due to an annotation specifying that the increment operation is commutative and if both \(f_1\) and \(f_2\) increment elements of a shared array. The dependency \((f_1 : x, f_2 : y) : \text{ind}\) could arise due to an annotation specifying that accessing independent elements of an array leads to an independent dependency type if \(f_1\) and \(f_2\) increment different elements of the shared array. The dependency \((f_1 : f_1, f_2 : f_2) : \text{dep}\) could arise due to a false positive from the alias analysis pass if both \(f_1\) and \(f_2\) might write to the same shared variable. The dependency \((f_1, f_2) : \text{ind}\) could arise due to a custom user annotation to encode the fact that \(f_1\) and \(f_2\) never write to any common variables. The first three dependencies exist between one variable from each instruction. The final dependency exists between both instructions. The labelled-argument-wise dependencies will be aggregated using the poset \((L, \leq)\) for edge merging to give:

\[
(f_1 : x, f_2 : y) : \text{ind}, \ (f_1 : f_1, f_2 : f_2) : \text{dep}
\]

and then aggregated using \((L, \leq')\) to give: \((f_1, f_2) : \text{dep}\). The dependency types found for the same pair of instructions \((f_1, f_2) : \text{dep}, (f_1, f_2) : \text{ind}\) are now merged (using \((L, \leq)\)) to give \((f_1, f_2) : \text{ind}\). This result means there is an independent (non-existent) dependency relation between \(f_1\) and \(f_2\).

### 6.3.2 Semantic-set-membership propagation

Some program properties are only valid at specific program points. For example, two abstract locations may be unique (not equal) until one is assigned to the other, at which point they are not unique and both would be excluded from any uniqueness set they are a member of after that instruction.

Sets tracking a property can opt in to using propagation rules to infer these properties at other points using the type annotation `propagate_all`, which constrains sets not to have named tuples. The propagation rules consist of an interprocedural data-flow analysis with the following set-propagation rules:

- For a property of an element to be true at the next instruction (from the current), the element must not be affected by current instruction. This is true if the function called by the instruction does not access the element or, if it does, the function must have been added to the `property_preserved` relation with the argument matching the element being considered. Alias analysis is used to check if any abstract memory locations are accessed by the function.

- For a property of an element to be true on entry to a basic block it must hold at all predecessor basic blocks\(^{17}\).

\(^{17}\)Or if all the dominators agree and the membership is not invalidated by all paths from this basic
• For a property of an element to be true as it enters a function it must hold at the contextualised-call site.

• For a property of an element to be true at the exit of a function it must hold at all the return constructs of that function.

The property tracking is available in the relation \texttt{member\_at}(x, S, i_c) for a variable \(x\) and set \(S\) and a contextualised instruction \(i_c = (c, i)\). Using set membership aids propagation efficiency when compared to the binary-relation encoding, since all pairs would need to be considered instead of each element in isolation. The relation \texttt{member\_at} and set type option \texttt{propagate\_all} is used in figure 6.10.

6.3.3 Checking program dependency graph edges

\texttt{SemSet} was created to offer commutativity information to transformations that are interested. One such transformation would be automatic loop parallelisation, exemplified in works such as \texttt{Helix} [15, 14] and \texttt{DSWP} [44, 31, 43]. The search for commutativity of PDG edges is centred around loops\(^\text{18}\) at a given analysis context \(c\). The \texttt{SemSet} analysis considers all reachable pairs of instructions from the chosen loop bodies. This forms a search space \texttt{space(inst1, ctx1, inst2, ctx2, ctx)}, across contextualised instructions \((\text{inst1}, \text{ctx1})\), \((\text{inst2}, \text{ctx2})\) starting at ctx. All matching inference rules are applied to elements in the search to find possible dependency types.

6.4 Case studies

When evaluating \texttt{SemSet}, three programs were used as case studies: \texttt{MD5SUM} section 6.4.1, \texttt{Dedup} section 6.4.2 and \texttt{Potrace} section 6.4.3.

Annotations were applied to each program to improve the accuracy of the inference of edge-dependency types in their PDGs. These programs were then parallelised manually, for the purpose of evaluation, using techniques from \texttt{DSWP} and \texttt{Helix}. The evaluation uses hand-transformation to parallelise the programs rather than using any automatic techniques. This is due to at the time of evaluation there being no publicly available performant parallelisation compiler passes for LLVM that parallelise using the PDG. The techniques were adapted, specifically for each case study, to operate on the enhanced PDG. The actual transformations for each application are described in the following sections. These transformations were applied manually due to no readily available implementation of either technique being accessible and the adaptation of either technique to use commutativity is block to itself.\(^\text{18}\)

\(^\text{18}\)That are non-reentrant—the function is not reachable from any instruction in its body.
main() {
...  
  for (i = 1; i < argc; i++) MDFile (argv[i]);  
}

static void MDFile (char *filename) {
  FILE *file; MD_CTX context; int len;
  unsigned char buffer[1024], digest[16];
  if ((file = fopen (filename, "rb")) == NULL)
    printf ("err: %s\n", filename);
  else {
    MDInit (&context);
    while (len = fread (buffer, 1, 1024, file))
      MDUpdate (&context, buffer, len);
    MDFinal (digest, &context);
    fclose (file);
    printf ("MD%ld (\%s) = ", MD, filename);
    MDPrint (digest);
    printf ("\n");
  }
}

Figure 6.8: The important loop in MD5SUM.

out of the scope of this work. The important dependencies that are broken and dependency statistics from the PDG are discussed.

The majority of annotations are applied to function declarations written in C header files. The headers, presented shortly, are ordered by first appearance in any of the programs in the case-study. There are annotations which are applied to already existing header and annotation-only headers used to logically group global definitions and global-inference rules. The annotation-header files are summarised in table 6.1 on page 133. Annotations are extracted from every file used in that program and form a larger SOUFFLÉ program that will produce the analysis result.

6.4.1 Md5sum

MD5SUM [46] (figure 6.8) is a program used to compute and display the MD5 hash of the contents of files specified by file names passed to the program as command-line arguments. The program opens a file, reads the contents, computes the hash, closes the file, prints the hash to the screen and repeats for the next file.

The program has many cross-iteration dependencies in the main loop that inhibit parallelising transformations:
1  # set stateless none
2  // make a new commutative set for each stateless function.
3  # infer member(s2, si),
4  # member(s, si) :-
5  # member(s, stateless),
6  # member(s2, function_state_alloc)
7  # si = ind(s,s2).

Figure 6.9: A newly created header file _anno_stateless.h containing only annotations.

- Multiple fread calls at line 13.
- Multiple calls to printf at lines 10, 16, 18 (inside the body of MDPrint) and 19.
- An opaque set of functions to compute an MD5 hash at lines 12, 14 and 15.

The header _anno_stateless.h defines a set named stateless on line 1. When this set is combined with the inference rule on lines 3–7 this guarantees to the compiler that any construct marked as stateless commutes with all other call instructions' internal state. A prerequisite for being a member of the stateless set is that no state is preserved between executions that could conflict with another construct anywhere in this program.

Considering open file descriptor (OFD) [4] uniqueness allows the precise tracking of files and streams, shown by annotations in figure 6.10. Three initial stream OFDs stdout, stderr and stdin are assumed to exist when entering the program and then each call to fopen, annotated in figure 6.11 will create a new unique OFD in the same set as the initial OFDs.

In Linux [54], for example, an OFD is a row in a file-descriptor table, which holds, among other information, a pointer to an inode and an offset in that file. However, this abstraction holds across POSIX [58] compliant systems. Calls to fread or fclose with unique file descriptors modify different rows in a file-descriptor table, making the operations commutative. This is encoded by including both their function-state allocations in a commutative set. In MD5SUM there will be a unique OFD created by each call to fopen, in the function MDFile, one for each unique loop iteration in main (see figure 6.8). This OFD is added to a global set of unique OFDs, then this is used to find calls to fread and fclose commuting across loop iterations, but not in the same iteration. The tracking of the stdout file descriptor (in figure 6.10) allows the analysis to infer that calls

---

19 There are two types: unique open file-descriptor sets and unique file sets. The former is captured in the type UniqueOFD and requires the set to be comprised of different OFDs. The latter, UniqueFile, requires the set to be comprised of file descriptors that reference different files.
20 This is declared using the three annotations including these streams into the UniqueOFDs set at the beginning of the entry function, usually main.
21 Both pointing to different OFDs.
# constant STDOUT ref
# constant STDERR ref
# constant STDIN ref

# set_type UniqueOFD propagate_all
# set UniqueOFDs UniqueOFD

# infer member_at(*STDOUT, UniqueOFDs, loc),
# member_at(*STDOUT, UniqueOFDs, loc),
# member_at(*STDIN, UniqueOFDs, loc) :-
# member(loc, entry_location).

# set_type ofd_commute_b (check, comm_dep, loc)
# set ofd_commute ofd_commute_b
# set_type OFD_Commute_Diff (check, comm_dep) unique(check)

# infer member((check, comm_dep), OFD_Commute_Diff(u)) :-
# member((check, comm_dep, loc), ofd_commute),
# member_at(check, u, loc),
# set_type(u, UniqueOFD).
# infer member(comm_dep, comm(p)) :-
# set_type(s, OFD_Commute_Diff),
# set_partition(p, s),
# member((check, comm_dep), p).

# set_type UniqueFile propagate_all
# set_type uf_commute_b (check, comm_dep, loc)
# set uf_commute uf_commute_b
# set_type File_Commute_Diff (check, comm_dep) unique(check)

...
to `printf` and MDPrint are not commutative, they are dependent\textsuperscript{22}.

The semantics of the header file `_anno_fs.h` is now explained. The annotations on lines 5–6 and 26 create the types for unique OFDs and unique files and an initial OFD set `UniqueOFDs`. The constant annotations on lines 1–3 are used to create global variables of type `ref` for each stream type. These constants are all added to the `UniqueOFDs` set (lines 8–11), which also tracks all OFDs from any function calls that open a file. There is a set type (`ofd_commute_b`) and set (`ofd_commute`) created, on lines 13 and 15, to handle the commutativity of functions that operate on files that are commutative if they are passed different OFDs. This set is a tuple containing three elements: `check`, `comm_dep` and `loc`. The `check` element is a file descriptor, the `comm_dep` element is the variable that could cause a dependency between this file-system operation and others and the variable `loc` is the location of the contextualised call instruction. Lines 17–24 make use of this set, creating a commuting set for all call instructions that are commutative. Lines 17–20 are used to create a set of call instructions that have a file descriptor in a unique OFD set. This set was created with the `unique` option, causing a set of partitions to be created. These partitions are accessible in the `set_partition` relation, which is then used on lines 21–24 to create a commuting set from each partition element containing each `comm_dep` element in that partition. The `infer` rules for the sets defined on lines 28–29 work similarly to the already described rules, on lines 17–24, but instead, acting on file descriptors referencing different files; these are omitted for brevity. These rules allow for overlapping writes as well as reads, used in `fcntl.h`, figure 6.12.

Annotations were attached to the declarations of the four externally declared functions `fopen`, `fread`, `fclose` and `printf` to expose OFD semantics, figure 6.11.

The infer annotations that are included in the header file `_anno_fs.h` (figure 6.10) are reused in `dedup` [9] in section 6.4.2. Inside the MD5* functions there are calls to `strlen` and `strcmp`, which are annotated in figure 6.13.

### 6.4.1.1 Inference result

The SEMSET analysis finds one strongly connected component (SCC) [21] formed from dependent calls to `printf`\textsuperscript{23} (lines 10, 17–19) between loop iterations.

### 6.4.1.2 Transformation

To take advantage of processor parallelism the program was transformed using the OPENMP annotation `omp parallel for ordered`. This annotation was applied to the whole loop with lines 17–19 inside an ordered block.

---

\textsuperscript{22}Since they all affect the same OFD `stdout`.

\textsuperscript{23}All writing to `stdout`.

122
Figure 6.11: The annotations attached to relevant functions in the stdio.h header file. The annotations on file-system calls (lines 1–34) include each read-only file-system function, and the close function into the ofd_commute set meaning that their state variables all commute if they are given unique OFDs. The unordered_exit_function (figure 6.16) annotation is attached to functions (lines 36–41) that exist in those exiting blocks.
int open(const char *pathname, int flags);

ssize_t read(int fd, void *buf, size_t count);

ssize_t write(int fd, const void *buf, size_t count);

int close(int fd);

Figure 6.12: The annotations attached to the fcntl.h header file. The annotations are similar to the fs annotation in stdio figure 6.11, but also make use of the unique file inference set uf_commute set for file-system functions that read to or write from different files. Same flags is the set of safe flags, which include O_RDONLY or O_WRONLY.

int strcmp(const char *s1, const char *s2);

size_t strlen(const char *s);

strncpy, strcpy, strcat

Figure 6.13: The annotations that are attached to functions in the string.h header file used in the worked examples presented. The strcmp function is stateless, read-only and does not capture any parameters.
This transformation respects the dependencies discovered by the analysis, namely each call to `printf` having a dependent (happens-before) relation and all commutative function calls to `read` sharing `comm-sync` dependencies. This is a Helix-style parallelisation transformation with one critical section wrapping the `printf`s.

### 6.4.2 Dedup

The Dedup program (from the Parsec benchmark suite Parsec-3.0 [9]) attempts to compress (encode) a file by breaking it into chunks and replacing any repeated chunks with the SHA1 [19] hash of the chunk. All chunks are themselves compressed after being split. The decode stage must undo this by deduplicating and decompressing. The decode stage first reads in a chunk. If it is a non duplicate then it is decompressed, the hash is computed, and it is put into a hash table of uncompressed chunks. If the chunk is a duplicate then it is looked up from the hash table. Finally, it is written to a file containing the decompressed, deduplicated input. The code taken from the loop of interest inside the function `Decode` in file `decoder.c` is shown in figure 6.14.

The functions `xread` and `xwrite` are wrappers around the libc functions `read` and `write`, which repeatedly call their respective functions until the file read from, or data passed to, the x-version of the function has been completely read from or written to. `read_chunk` allocates memory and makes multiple calls to `xread` to read in a chunk. `uncompress_chunk` decompresses data from chunk->compressed_data into chunk->uncompressed_data, by allocating memory and calling `uncompress`, a zlib [23] library function. SHA1_Digest computes the SHA1 hash of the uncompressed data and writes it to the pointer in chunk->sha1. EXIT_TRACE writes the message argument to `stderr` and then calls `exit(-1)`. hashtable_insert and hashtable_search are thread-safe operations for inserting into and searching a hashtable.

The memory operations at lines 3 and 7 (inside `read_chunk`) are included in the `stateless` set in figure 6.15. The file operations external to the loop (open, close) and the operations contained in the function at lines 7 (`read`) and 33 (`write`) are annotated in the `fcntl.h` file seen in figure 6.12. This uses the file-system inference from the header `_anno_fs.h` in figure 6.10. The calls to `fprintf` and `printf` used inside error procedures are annotated in figure 6.11. Infer annotations are used to define unordered-exit commutativity presented in figure 6.16 and make use of the `unique` set marker. Lines 4–6 find all `unordered_instructions` from any instruction in an `unordered_exit` loop. Lines 13–16 find all `unordered_exit_functions` that are also `unordered_instructions`. Lines 18–25 make commuting sets containing `unordered_exit_instructions` that are in different loop iterations. The exit procedures `exit`, `perror` and `__assert_fail`\(^{24}\) used by the unordered-exit section inference are annotated in figure 6.11. The SHA1* and

---

\(^{24}\)The procedure is called if the assertion is false.
while(TRUE) {
    if(chunk==NULL) {
        chunk = malloc(sizeof(chunk_t));
        if(chunk==NULL) EXIT_TRACE("Memory allocation failed.\n");
    }
    //get input data
    r=read_chunk(fd_in, chunk);
    if(r<0) EXIT_TRACE("error reading from input file")
    else if(r==0) break;
    chunk_t *entry;
    if(!chunk->header.isDuplicate) {
        //We got the compressed data, use it to get original data back
        r=uncompress_chunk(chunk);
        if(r<=0) EXIT_TRACE("error uncompressing data")
        //Compute SHA1 sum and add new chunk with uncompressed data to cache
        SHA1_Digest(chunk->uncompressed_data.ptr,
                    chunk->uncompressed_data.n,
                    chunk->sha1);
        if(hashtable_insert(cache, chunk->sha1, chunk) == 0) {
            EXIT_TRACE("hashtable_insert failed");
        } else {
            entry=chunk;
            chunk=NULL;
        }
    } else {
        //Use a SHA1 key, the unique uncompressed data
        entry=hashtable_search(cache, chunk->sha1);
        if(entry==NULL) {
            EXIT_TRACE("Encountered a duplicate chunk");
        }
    }
    // write uncompressed data to output file
    int written = xwrite(fd_out,
                         entry->uncompressed_data.ptr,
                         entry->uncompressed_data.n)
    if( written < entry->uncompressed_data.n) {
        EXIT_TRACE("error writing to output file");
    }
}

Figure 6.14: The DEDUP program’s main loop, extracted from the Decode function.
# infer member(free, stateless).
# infer function_no_capture(*ptr).
void free(void *ptr);

# infer member(free, malloc).
# infer function_no_capture(*$args).
void *malloc(size_t size);

# infer member(free, realloc).
# infer function_no_capture(*$args).
void *realloc(void *ptr, size_t size);

# infer member(free, stateless).
void * memcpy(void *restrict dst, const void *restrict src, size_t n);

Figure 6.15: Annotations attached to the stdlib.h header file.

# set unordered_instruction none
# set unordered_exit none

# infer member(inst, unordered_instruction) :-
# member(l, unordered_exit),
# member((inst, l, _), instruction_in_loop).

# set unordered_exit_function none

# set error_exit_instruction none

# infer member(inst, error_exit_instruction) :-
# member(((inst, f), call_instruction_fn_target),
# member(f, unordered_exit_function),
# member(inst, unordered_instruction).

# infer member(bb, s),
# member(bb_other, s) :-
# member(inst1, error_exit_instruction),
# member(inst2, error_exit_instruction),
# l_iter1 != l_iter2,
# member((inst1, loop, l_iter1), instruction_in_loop),
# member((inst2, loop, l_iter2), instruction_in_loop),
# s = comm(inst1, inst2).

Figure 6.16: The annotations attached to the annotation-only header file _anno_-_unordered_exit.h.
# infer member(SHA1_Init, stateless).
# infer function_no_capture(*c).
void SHA1_Init (SHA_CTX *c);

# infer member(SHA1_Update, stateless).
# infer function_no_capture(*$args).
# infer function_accesses(*buf, read).
void SHA1_Update (SHA_CTX *context,
    unsigned char const *buf, unsigned len);

# infer member(MD5Final, stateless).
# infer function_no_capture(*$args).
void SHA1_Final (unsigned char *d,
    SHA_CTX *c);

# infer member(uncompress, stateless).
# infer function_no_capture(*$args).
# infer function_accesses(*source, read).
int uncompress (void *dest, size_t *destLen,
    const void *source, size_t sourceLen)

Figure 6.17: The `stateless` annotation that is also attached to the functions SHA1* in the header files `openssl/sha.h`, and the function `uncompress` in the header `zlib.h`.

`uncompress` procedures are annotated as `stateless` in figure 6.17. The unordered-exit loop is marked in figure 6.18.

The main loop of interest is marked as an unordered-exit loop in figure 6.18. The two unique file descriptors (`fd_in`) and (`fd_out`) used on lines 7 and 33 are also annotated in figure 6.18.

6.4.2.1 Inference result

The following dependency types are altered by the annotations:

- **EXIT_TRACE** has a control dependency on the next instruction and also a data dependency on the `stderr` stream. Using the assumption that the ordering of exits across different iterations are irrelevant, it was deduced that the operations related to errors in these basic blocks were commutative across different loop iterations. This was not true for exits in the same loop iteration.

- The `xread` and `xwrite` functions have dependencies since they access the file system. The file descriptors given to these two operations are unique—meaning they point to unrelated files.

- The memory allocations were found to be commutative.
# infer member_at(fd_in, UniqueFile, $loc).
fd_in = open(...);
...
# infer member_at(fd_in, UniqueFile, $loc).
fd_out = open(...);

// marking the whole loop
# infer member($loc, unordered_exit).
while(TRUE) {
  ...
}
}

Figure 6.18: The ad-hoc annotations attached to the body of the loop of interest in DEDUP (figure 6.14), marking that the file descriptors are both pointing to different files. The main loop is marked as having unordered_exits.

The annotations applied to DEDUP result in three SCCs: one of size 1 (at line 33 xwrite), one of size 2 (both the hash-table operations lines 20 and 27) and one of size 4 (for each xread operation all inside line 7).

6.4.2.2 Transformation

DEDUP was parallelised using techniques from DSWP, where all the operations in a SCC are allocated to the same pipeline stage. DEDUP was parallelised using a three-stage pipeline:

- The first stage, lines 2–9, was responsible for reading the file in its compressed form.

- The second stage, lines 14–19, was responsible for decompression and hashing, applied if the header was a duplicate. If the header was not a duplicate this stage would simply pass the data to the final stage.

- The final stage, lines 20–39, was responsible for reordering chunks, deduplication and writing the output to a file.

The three SCCs have their instructions allocated to the first and final stages. The first and final stages are allocated to a single thread each, but the second stage could have been allocated to many threads, in which case these threads would be operating on many loop iterations out-of-order. The parallelisation uses PTHREADS and a thread-safe queue for communication already found in the DEDUP program.
6.4.3 Potrace

POTRACE is a program (with a code excerpt in figure 6.19) used to convert bitmap images into vector graphics. The code structure of POTRACE is similar to MD5SUM, it reads in many files, applies a transformation (converting bitmaps into vector operations), and then writes each vector graphic to a file\textsuperscript{25}.

The procedure \texttt{make_outfilename} makes use of simple string operations, maths operations (annotated in figure 6.20) and the procedures \texttt{my_fopen_read}, \texttt{bm_read} and \texttt{my_fclose}, which are similar to the already-annotated file-system operations. The procedures \texttt{potrace_trace} and \texttt{potrace_state_free} are found in a POTRACE-specific library and are marked as stateless. Due to the call depth of \texttt{calc_dimensions} it was necessary to mark this function as stateless since the alias analysis did not have a deep enough context stack to discover this.

There are three stateful functions (\texttt{init_f}, \texttt{page_f} and \texttt{term_f}, lines 13, 23 and 27) used to write each vector graphic to disk. They store a global reference to the file-descriptor argument. They are all marked, at the call site, with a \texttt{fs} effect and an annotation specifying that they commute with other different file-descriptor functions from the stdio library. They also capture the \texttt{fout} argument and are added to a shared capture set in figure 6.21, causing a dependence between these function calls in and between loop iterations.

6.4.3.1 Inference result

There is one SCC found between the stateful functions on lines 13, 23 and 27. The notable dependencies altered by annotations are:

- Basic blocks that are used to provide an error message and exit the program early were found as commutative.
- The procedures for saving each file were found to be dependent, meaning they must all be ordered.
- \texttt{calc_dimensions} was found to be independent across iterations.

6.4.3.2 Transformation

POTRACE was parallelised using the OPENMP annotation \texttt{omp parallel for ordered}, with an \texttt{omp ordered} section around the stateful procedures used for saving each file—a HELIX style parallelisation with one critical section.

\textsuperscript{25}This differs from MD5SUM.
for (i=0; i<info.infilecount; i++) {
    outfile = make_outfilename(info.infiles[i], b->ext);
    if (!outfile) { // code format of error
        fprintf(stderr, " POTRACE ": %s\n", strerror(errno));
        exit(2);
    }
    fin = my_fopen_read(info.infiles[i]);
    if (!fin) //error ...
    fout = my_fopen_write(outfile);
    if (!fout) //error
    if (b->init_f) {
        TRY(b->init_f(fout));
    }
    //process_file(b, info.infiles[i], outfile, fin, fout);
    r = bm_read(fin, info.blacklevel, &bm);
    // error check ...
    st = potrace_trace(info.param, bm);
    if (!st || st->status != POTRACE_STATUS_OK) //error ...
    r = b->page_f(fout, st->plist, &imginfo);
    if (r) //error
    potrace_state_free(st);
    ...
    if (b->term_f) {
        TRY(b->term_f(fout));
    }
    ...
    my_fclose(fin, info.infiles[i]);
    my_fclose(fout, outfile);
    free(outfile);
}

Figure 6.19: The POTRACE main loop. Code not relevant to this discussion has been omitted; this program has been simplified slightly.

# infer member(fabs, stateless).
double fabs(double x);

//cos, sqrt, sin

Figure 6.20: The annotations, attached to functions in the math.h header file, referenced in this work.
# set b_fd_capture_set none.

# infer function_capture_set(b_fd_capture_set).
# infer property_presered(fd, UniqueOFD).
# infer member((*$0, init_f, $loc), ofd_commute).
 b->init_f(fd)

# infer function_capture_set(b_fd_capture_set).
# infer function_no_capture(*plist).
# infer function_no_capture(imginfo).
# infer function_accesses(*plist, read).
# infer function_accesses(imginfo, read).
# infer property_presered(fd, UniqueOFD).
# infer member((*$0, page_f, $loc), ofd_commute).
 b->page_f(fd, plist, &imginfo)

# infer function_capture_set(b_fd_capture_set).
# infer property_presered(fd, UniqueOFD).
# infer member((*$0, term_f, $loc), ofd_commute).
 b->term_f(fd)

Figure 6.21: Annotations attached to the function-call sites used in POTRACE. They use a capture set (b_fd_capture_set) to model the shared state between the three functions. The annotations are attached here to guide the inference.

### 6.5 Evaluation

- **Inference code**—the number of annotations that are not infer atoms.

- **Library specific**—annotations that are atoms attached to a specific function declaration.

- **Program specific**—annotations that are non-reusable annotation atoms, which denote a specific property of the program they are included in.

One annotation can span multiple lines. The number of annotations required to fully analyse MD5SUM, while large (31), are reusable. There is a large overlap in annotations between each program being considered. For instance, almost every library function annotated for MD5SUM was later reused in POTRACE and all the inference code used in MD5SUM was also used in POTRACE. As more programs are considered the coverage of annotations increase and the number of new annotations required for each new program to be usefully analysed decreases. This was the case for POTRACE after analysing both MD5SUM and DEDUP.
Table 6.1: Summary of the annotations used in all the programs considered in the case studies.

<table>
<thead>
<tr>
<th>Anno. Only</th>
<th>Figure</th>
<th>MD5SUM</th>
<th>DEDUP</th>
<th>POTRACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>_anno_stateless.h</td>
<td>6.9</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_anno_fs.h</td>
<td>6.10</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>__anno_unordered_exit.h</td>
<td>6.16</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>string.h</td>
<td>6.13</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>stdio.h</td>
<td>6.11</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>fcntl.h</td>
<td>6.12</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Anno. Header</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stdlib.h</td>
<td>6.15</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>openssl/sha.h</td>
<td>6.17</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>zlib.h</td>
<td>6.17</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>math.h</td>
<td>6.20</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: This table contains the number of annotations required to safely parallelise each program. The number in brackets is the number of instructions with program-specific annotations attached.

<table>
<thead>
<tr>
<th>Inference code</th>
<th>Library specific</th>
<th>Program specific</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MD5SUM</td>
<td>12</td>
<td>19</td>
</tr>
<tr>
<td>DEDUP</td>
<td>25</td>
<td>52</td>
</tr>
<tr>
<td>POTRACE</td>
<td>18</td>
<td>58</td>
</tr>
<tr>
<td>Overlap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MD5SUM, DEDUP</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>MD5SUM, POTRACE</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>DEDUP, POTRACE</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>Common in at least two programs</td>
<td>18</td>
<td>21</td>
</tr>
</tbody>
</table>
Table 6.3: The cross-iteration dependencies present in each program’s PDG. All comm-sync typed sets were empty for all programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Dependent Comm-NoSync Independent Total SCC instruction size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>w/o w w/o w w w w</td>
</tr>
<tr>
<td>MD5SUM</td>
<td>28 10 0 18 27 27 55 { 4 }</td>
</tr>
<tr>
<td>DEDUP</td>
<td>2214 14 0 1203 271 1268 2485 {1,2,4}</td>
</tr>
<tr>
<td>POTRACE</td>
<td>946 6 0 313 135 762 1081 { 3 }</td>
</tr>
</tbody>
</table>

The number of cross-iteration dependencies found in each program with (w) and without (w/o) SEMSET can be seen in table 6.3. The SCC instruction size column shows the set of sizes of all SCCs with a dependency crossing a loop-iteration boundary. Cross-iteration dependencies are counted by considering every pair of contextualised instructions, each taken from a different loop iteration, meaning that an instruction in one loop iteration could be counted as many dependent instructions if the instruction is dependent on many instructions in the other loop iteration. The label comm-sync requires the marked construct to be placed in a critical section, whereas comm-nosync means that the construct is commutative without any synchronisation.

Each program’s speed-up after parallelisation is compared to its sequential version in table 6.4 and graphed in figure 6.22. MD5SUM shows a linear increase in performance with thread count; this speed-up is highly dependent on the distribution of the size of the input files. This program is also memory-bound for larger thread counts. DEDUP shows a 1.5x speed-up over sequential execution, however the program shows little performance increase when increasing the number of threads in the middle pipeline stage (moving from three to four threads). This is due to the program being memory-bound. POTRACE shows up to a 3.3x speed-up over sequential execution, but after this point there is a slight slow down pointing to the program being IO-bound and having too many threads competing for the resources.

Table 6.4: Speed-up after parallelisation. The program’s execution time is normalised to the sequential (seq) version of the program and the numbers represent the threads allocated to each program.

<table>
<thead>
<tr>
<th>Program</th>
<th>Number of threads</th>
<th>seq</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD5SUM</td>
<td>1 (1.44s)</td>
<td>1.01</td>
<td>1.26</td>
<td>-</td>
<td>1.42</td>
<td>-</td>
<td>1.57</td>
<td>1.66</td>
<td>1.95</td>
<td>2.44</td>
<td>2.82</td>
<td>2.88</td>
<td>3.00</td>
<td></td>
</tr>
<tr>
<td>DEDUP</td>
<td>1 (24.56s)</td>
<td>-</td>
<td>-</td>
<td>1.49</td>
<td>1.48</td>
<td>1.55</td>
<td>1.54</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POTRACE</td>
<td>1 (327.16s)</td>
<td>1.01</td>
<td>0.92</td>
<td>-</td>
<td>1.71</td>
<td>-</td>
<td>2.45</td>
<td>3.10</td>
<td>3.35</td>
<td>3.19</td>
<td>3.19</td>
<td>3.01</td>
<td>3.03</td>
<td></td>
</tr>
</tbody>
</table>

The number of cross-iteration dependencies found in each program with (w) and without (w/o) SEMSET can be seen in table 6.3. The SCC instruction size column shows the set of sizes of all SCCs with a dependency crossing a loop-iteration boundary. Cross-iteration dependencies are counted by considering every pair of contextualised instructions, each taken from a different loop iteration, meaning that an instruction in one loop iteration could be counted as many dependent instructions if the instruction is dependent on many instructions in the other loop iteration. The label comm-sync requires the marked construct to be placed in a critical section, whereas comm-nosync means that the construct is commutative without any synchronisation.

Each program’s speed-up after parallelisation is compared to its sequential version in table 6.4 and graphed in figure 6.22. MD5SUM shows a linear increase in performance with thread count; this speed-up is highly dependent on the distribution of the size of the input files. This program is also memory-bound for larger thread counts. DEDUP shows a 1.5x speed-up over sequential execution, however the program shows little performance increase when increasing the number of threads in the middle pipeline stage (moving from three to four threads). This is due to the program being memory-bound. POTRACE shows up to a 3.3x speed-up over sequential execution, but after this point there is a slight slow down pointing to the program being IO-bound and having too many threads competing for the resources.
Figure 6.22: The speed-up of programs relative to their sequential performance.

(a) Shows the trend of md5sum execution speed-up with extra threads used.

(b) The runtime of dedup when varying the number of threads starting from three; one for each pipeline stage and each thread after that increases the number of instances of stage two.

(c) This shows a linear relationship between number of threads and speed-up, of potrace, until around 10 threads, where there is a slight decrease in performance (shown in the dotted line).
6.5.1 Evaluation environment

SemSet was written in Soufflé (version 2.2), extending CClyzer-Soufflé [8, 7], which was used as a starting point for program analysis and alias analysis of C/C++ programs. LLVM-8.0 was used to compile and extract a program representation compatible with Soufflé, a technique previously used in CClyzer-Soufflé.

The system used to run experiments contained an Intel(R) Xeon(R) W-2195 CPU @ 2.30GHz (18-core) with 24.75 MB of unified last-level cache, 256 GB of DDR4 memory, running Ubuntu 20.04.4 LTS (GNU/Linux 5.11.0-36-generic x86_64). Timing measurements were taken using time and the median of ten measurements are presented.

6.6 Summary

SemSet provides an annotation language and analysis-time inference system used for expressing program commutativity, which is reusable, extensible and composable. For example, the annotations applied to printf in MD5SUM were then used unchanged in DEDUP. The inference rules for OFDs and stateless functions were also shared between all programs. This points to the reusability of the annotations and the framework used to specify those annotations. The ease of defining unordered exits shows the extensibility of these annotations, which are also shared between the programs DEDUP and POTRACE. The isolated treatment of per-variable denoted dependencies, as seen when annotating fread's state and input buffer (inferred from alias analysis) individually implies this work has composable annotations. The custom stateless set shows the abstraction possible when using the inference system.

This annotation framework is shown to be useful in the real world, allowing speed-ups to legacy programs. Using SemSet it is possible to parallelise (and speed-up) MD5SUM without any program-specific knowledge, which as far as can be ascertained has not previously been achieved.

The infer annotations allow the embedding of Soufflé clauses in legacy programs; this is only practical due to the succinct and declarative nature of the Datalog language. Moreover, the implementation of the data-flow analyses used by this approach were implemented succinctly, as noted by Bravenboer et al. [11].
Conclusion

This thesis demonstrated the further applicability of declarative programming languages to compiler analysis in three ways. Firstly, it established how a Datalog-like language \texttt{Soufflé}\textsuperscript{stable} can increase the expressiveness and reduce asymptotic complexity of \texttt{Soufflé} programs, using the ideas of stable-companion relations and dynamic stratification. Secondly, it presented a declarative implementation of a common programming analysis GVN (chapter 5). Finally, it presented a new set of programming-language annotations with an inference system implemented in \texttt{Soufflé}, with program annotations also embedding \texttt{Soufflé} clauses (chapter 6).

\texttt{Soufflé}\textsuperscript{stable} (chapter 4) makes the \texttt{Soufflé} language better-suited still for compiler analysis, allowing for commonly used graph-exploration algorithms to be implemented naturally. The stable-companion-relation extensions compose well with the original \texttt{Soufflé} language since they are well isolated from other clauses, which do not reference a stabilised relation or their companion.

DGVN was created to demonstrate two things: firstly that implementation of a complex set-intersection-based analysis is practical in \texttt{Soufflé} and secondly that the \texttt{Soufflé}\textsuperscript{stable} extension is useful, performant and correct. DGVN uses a stable-companion relation to allow the negation of parts of previously computed basic-block value partitions, vital to the implementation of this algorithm. The DGVN implementation provides a starting point for refinement of the declarative implementation of GVN. DGVN was compared to LLVM’s GVN implementation using multiple benchmarks from SPEC\textsc{int}2006. DGVN was used to find correct redundancies for simple test programs, missed by LLVM.

The \texttt{SemSet} analysis allows for a more complete and reusable commutativity analysis, leading to performance-enhancing transformations of sequential code. The data-flow equations presented in \texttt{SemSet} were naturally specified as a \texttt{Soufflé} program. The annotated set-inference rules, embedded into the source program, were written as well-known Datalog rules, helping with familiarity of the system while not being overly verbose. The analysis achieved this by containing only reusable analysis rules and simple semantic
The reusability of SemSet was displayed by the number of function annotations used by multiple case-study programs, the extensibility was enabled via the succinct inference system borrowed from Soufflé and the composability was demonstrated by use of multiple annotations of different applicabilities acting safely and reasonably on the same instruction.

This thesis concludes that declarative program analysis is promising and worth exploring further. The language extensions presented in this work allow more useful Soufflé programs to be written to aid this exploration.

7.1 Future work

The ideas presented in each chapter all have promising avenues for further exploration.

The Soufflé\textsuperscript{stable} extension is also applicable to statically unstratifiable aggregation, however this was not explicitly discussed throughout this thesis, even though basic aggregation is supported by Soufflé\textsuperscript{stable}. The Soufflé extension to allow dynamic stratification requires specific conditions to be correct for all inputs. This could be verified using an inference system, in some cases with input constraints, using the constraints laid out by Ross [49]. The exact design of this system and how it would interact with the stable-companion set construct would be an important topic to explore.

The DGVN implementation could be extended to support symbolic reduction and memory equivalence. Symbolic reduction could be added using a symbolic reduction system for finding normalised expressions. These normalised expressions may then be merged. Supporting memory equivalence would require a must- and may-alias analysis implementation—used to compute pairs of load instructions which must access the same location. There must be no possibility that any store instruction, on any path between the load instructions, may affect that location. The semi-naïve evaluation scheme used in Soufflé and specifically DGVN creates one big loop for a series of mutually recursive relations, which leads to unnecessary overhead. A more informed order could partially alleviate this. The execution of programs, such as DGVN, could be tuned by adjusting the evaluation order. This could require compiler hints to affect the produced RAM, which would be a topic for future work.

The possibility, in SemSet, to infer program properties, conditional on a variable configuration or branch result, would expand the scope of this technique. The exploration of performance improvements, since inference and summary of successive PDGs requires large amounts of extra computation, would be possible with careful exploration of the PDG.
References


Problem 2 in chapter 4 has the program in figure 4.10 as a solution. This RAM excerpt is from that program. The underlined elements highlight modifications to the SOUFLÉ compiler as part of this thesis.

The polynomial program has the following RAM excerpt, with one clause/QUERY to illustrate each compiler change:

```prolog
1  monomial_of(e,var,cl) :-
2       sum_polynomial(e,l,r),
3       monomial_of(l,var,cl),
4       monomial_of_f(r),
5       !monomial_of(r,var,)._.
6
QUERY
7   IF (((NOT ISEMPTY(@delta_monomial_of)) AND
8        (NOT ISEMPTY(sum_polynomial))) AND
9        (NOT ISEMPTY(monomial_of_f)))
10  FOR t0 IN sum_polynomial
11     IF ((t0.2) IN monomial_of_f AND
12        (NOT (t0.0) IN monomial_of_f))
13     FOR t1 IN @delta_monomial_of ON INDEX t1.0 = t0.1
14       IF ((NOT (t0.0,t1.1,t1.2) IN monomial_of) AND
15          (NOT (t0.2,t1.1,UNDEF) IN monomial_of))
16       INSERT (t0.0, t1.1, t1.2) INTO @new_monomial_of
17   END QUERY
18 QUERY (equation 4.3)
19 IF (NOT ISEMPTY(@delta_monomial_of_f_in))
```

APPENDIX A

RAM code for the polynomial summation program
FOR t0 IN @delta_monomial_of_f_in
    IF ((t0.0,UNDEF,UNDEF) IN @delta_monomial_of AND
        (NOT (t0.0) IN monomial_of_f))
        INSERT (t0.0) INTO @new_monomial_of_f_in
END QUERY

monomial_of_f(e) :-
    monomial_of_f_in(e),
    !\Delta_monomial_of(e,_,_).
QUERY (equation 4.2)
    IF (NOT ISEMPTY(@delta_monomial_of_f_in))
    FOR t0 IN @delta_monomial_of_f_in
        IF ((NOT (t0.0) IN monomial_of_f) AND
            (NOT (t0.0,UNDEF,UNDEF) IN @delta_monomial_of))
            INSERT (t0.0) INTO @new_monomial_of_f
    END QUERY

monomial_of_f(e) :-
    sum_polynomial(e,l,r),
    monomial_of_f(l),
    monomial_of_f(r).
QUERY
    IF (((NOT ISEMPTY(@delta_monomial_of_f)) AND
        (NOT ISEMPTY(sum_polynomial))) AND
        (NOT ISEMPTY(monomial_of_f)))
        FOR t0 IN sum_polynomial
            IF (((t0.2) IN @delta_monomial_of_f AND
                (t0.1) IN monomial_of_f) AND
                (NOT (t0.0) IN @monomial_of_f_in))
                INSERT (t0.0) INTO @new_monomial_of_f_in
        END QUERY
Passes run before GVN by LLVM

The passes run before any GVN by clang with the optimisation flag -O3 turned on are as follows:

- tti -tbaa -scoped-noalias -assumption-cache-tracker
- targetlibinfo -verify -ee-instrument -simplifycfg -domtree
- sroa -early-cse -lower-expect -targetlibinfo -tti -tbaa
- scoped-noalias -assumption-cache-tracker -profile-summary-info
- forceattrs -inferattrs -domtree -callsite-splitting -ipsccp
- called-value-propagation -globalopt -domtree -mem2reg -deadargelim
- domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq
- opt-remark-emitter -instcombine -simplifycfg -basicccg -globalaa -aa
- prune-eh -inline -functionattrs -argpromotion -domtree -sroa
- basicaa -aa -memoryssa -early-cse-memssa -speculative-execution
- basicaa -aa -lazy-value-info -jump-threading -correlated-propagation
- simplifycfg -domtree -aggressive-instcombine -basicaa -aa -loops
- lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine
- libcalls-shrinkwrap -loops -branch-prob -block-freq -lazy-branch-prob
- lazy-block-freq -opt-remark-emitter -pgo-memop-opt -basicaa -aa
- loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter
- tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify
- lcssa-verification -lcssa -basicaa -aa -scalar-evolution -loop-rotate
- licm -loop-unswitch -simplifycfg -domtree -basicaa -aa -loops
- lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine
- loop-simplify -lcssa-verification -lcssa -scalar-evolution
- indvars -loop-idiom -loop-deletion -loop-unroll -mldst-motion
- phi-values -basicaa -aa -memdep -lazy-branch-prob -lazy-block-freq
- opt-remark-emitter