On the Practice and Application of Context-Free Language Reachability

Nicholas Hollingum

A thesis submitted in fulfilment of the requirements for the degree of
DOCTOR OF PHILOSOPHY,
Faculty of Engineering and IT,
The University of Sydney,
2018

Attribution

This is to certify that to the best of my knowledge, the content of this thesis is my own work. This thesis has not been submitted for any degree or other purposes. This thesis contains material submitted or accepted for publication:

- Chapter 3 of this thesis is published as [27]. I assisted in formalising the algorithmic procedure, and its complexity, as well as designing some parts of the algorithm. Most of the coding on Gigascale itself was completed by Jens Dietrich. I designed and ran the experimental validation, as well as prepared the artefact for evaluation. I contributed a significant portion of the technical writing.
- Chapter 4 of this thesis is published as [35]. I performed the majority of the research and technical writing presented there.
- Chapter 5 of this thesis is submitted for publication as [37]. I performed the majority of the research and technical writing presented there.
- Chapter 6 of this thesis is published as [36]. I performed the majority of the research and technical writing presented there.

I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged. In addition to the statements above, in cases where I am not the corresponding author of a published item, permission to include the published material has been granted by the corresponding author.

Abstract

The Context-Free Language Reachability (CFL-R) formalism relates to some of the most important computational problems facing researchers and industry practitioners. CFL-R is a generalisation of graph reachability and language recognition, such that pairs in a labelled graph are reachable if and only if there is a path between them whose labels, joined together in the order they were encountered, spell a word in a given context-free language. The formalism finds particular use as a vehicle for phrasing and reasoning about program analysis, since complex relationships within the data, logic or structure of computer programs are easily expressed and discovered in CFL-R. Unfortunately, The potential of CFL-R can not be met by state of the art solvers. Current algorithms have scalability and expressibility issues that prevent them from being used on large graph instances or complex grammars.

This work outlines our efforts in understanding the practical concerns surrounding CFL-R, and applying this knowledge to improve the performance of CFL-R applications. We examine the major difficulties with solving CFL-R-based analyses at-scale, via a case-study of points-to analysis as a CFL-R problem. Points-to analysis is fundamentally important to many modern research and industry efforts, and is relevant to optimisation, bug-checking and security technologies. Our understanding of the scalability challenge motivates work in developing practical CFL-R techniques. We present improved evaluation algorithms and declarative optimisation techniques for CFL-R, capitalising on the simplicity of CFL-R to creating fully automatic methodologies. The culmination of our work is a general-purpose and high-performance tool called Cauliflower, a solver-generator for CFL-R problems. We describe Cauliflower and evaluate its performance experimentally, showing significant improvement over alternative general techniques.

Acknowledgements

Foremost, I would like to thank my PHD supervisor, Bernhard Scholz, without whom I could not have undertaken this thesis. Bernhard's tireless efforts in proof reading and correcting my papers and his insights into the theory of my work were the guiding influence throughout my thesis. Bernhard served as my foil, where I would rather write code and run experiments he insisted on proper theoretical tratment, ultimately to the betterment of this thesis. He has been my link to the greater research world, and ultimately made all my publications and progress possible.

I can not go without thanking my long-standing colleague Jens Dietrich. Jens' efforts were instrumental not only in one of our major publications, but also the background and understanding which motivates my bias towards "practical" research, and he has formented my idea of what proper experimental study involves. Thank you Jens, for the visits and for the coffees.

My thanks to Andrew Santosa, my honours supervisor and a long time colleague during my industry and PHD work. Andrew truly deserves most of the credit for turning me from a lazy undergraduate into what I am today. His patience in teaching me what "technical writing" actually is was the important first step on the road that led to this thesis.

I would also like to thank my auxiliary supervisors Julian Mestre and Vincent Gramoli, for their advice on integer programming and prallel analysis.

My thanks to the team at Oracle Labs, Brisbane, for their support during the first few years of my PHD candidacy. Thank you Lian Li, Paddy Krishnan and Cristina Cifuentes for the opportunities you gave me. And thank you to all my friends and colleagues from those days, whose collective names exceed the maximum word count for an achnowledgements section.

A special thanks to my fellow students of the Sydney Programming Languages group, past and present. I could not have asked for a better group.

And finally, my friends and family, for their support and encouragement throughout this process. Words can not express my gratitude.

Contents

1 Introduction **4**
- 1.1 The CFL-R Problem . . . 5
  - 1.1.1 Motivation . . . 6
  - 1.1.2 Running Example . . . 8
  - 1.1.3 Formal Definition . . . 13
- 1.2 Open Issues in CFL-R Research . . . 16
- 1.3 Contributions . . . 17

2 Literature Review **20**
- 2.1 Foundational Work . . . 20
  - 2.1.1 Language Recognition . . . 20
  - 2.1.2 Graph Reachability . . . 22
  - 2.1.3 Datalog . . . 23
- 2.2 Algorithms . . . 24
- 2.3 Applications . . . 25
  - 2.3.1 Points-to Analysis . . . 27

3 Applications **31**
- 3.1 Java Analysis . . . 31
  - 3.1.1 Library Support . . . 33
  - 3.1.2 Limitations . . . 33
- 3.2 Gigascale Analysis . . . 35
  - 3.2.1 Opportunities in Points-To . . . 35
  - 3.2.2 Implementation Optimisations . . . 37
    - 3.2.2.1 Oracle Generation . . . 38
    - 3.2.2.2 Data Structures . . . 41
    - 3.2.2.3 Transitive Closure . . . 42
- 3.3 Effectiveness . . . 43
  - 3.3.1 Experimental Performance . . . 44
  - 3.3.2 Parametric Analysis . . . 48
  - 3.3.3 Generalising . . . 50

4 Algorithmic Improvements **53**
4.1 Opportunities . . . 53
4.2 Algorithms . . . 56
4.2.1 Preamble . . . 56
4.2.2 Fixpoint Solution . . . 57
4.2.3 CFL-R Semi-naïve . . . 58
4.3 Data Structures . . . 62
4.3.1 Quadtrees . . . 62
4.3.1.1 Concise Representation . . . 65
4.3.1.2 Leaf Bitmaps . . . 66
4.3.1.3 Node Squashing . . . 67
4.3.1.4 Strassen's Multiplication . . . 68
4.3.2 Neighbourhood Maps . . . 70
4.4 Experimental Results . . . 71
4.4.1 Methodology . . . 71
4.4.2 Benchmarks . . . 72
4.4.3 Normalisation . . . 72
4.4.4 Performance . . . 74

5 Declarative Optimisation **76**
5.1 Declarative Languages . . . 76
5.1.1 Hints . . . 76
5.2 Executing CFL-R . . . 79
5.2.1 CFL-R Evaluation . . . 79
5.2.2 Matrix-based Solvers . . . 79
5.2.3 Wasted Searches . . . 82
5.2.3.1 An Execution Model . . . 91
5.2.3.2 Approximating Waste . . . 93
5.3 Optimising Execution . . . 95
5.3.1 Feedback . . . 96
5.3.2 Ordering . . . 97
5.3.3 Promoting . . . 99
5.3.4 Filtering . . . 100
5.4 Experiments . . . 101
5.4.1 Setup . . . 101
5.4.2 Results . . . 104

6 CFL-R Tools **108**
6.1 Design Goals . . . 108
6.1.1 Expressibility . . . 109
6.1.2 Performance . . . 109
6.1.3 Convenience . . . 110
6.2 Cauliflower . . . 111
6.2.1 Semantics . . . 112
6.2.1.1 Reversal . . . 112
6.2.1.2 Templating . . . 113

6.2.1.3 Branching 114
6.2.1.4 Disconnection 116
6.2.2 CauliDSL 117
6.2.3 Solver Generator 119
6.2.3.1 Parallelism 122
6.3 Viability 122
6.3.1 Comparison with General Tools 123
6.3.2 Comparison with Specialised Tools 124
6.3.3 Execution Details 125

7 Conclusion **128**
7.1 Future Work 130

Chapter 1

Introduction

Of the many formalisms that have been developed or adopted in the pursuit of software analysis, few manage to balance both *generality* and *practicality*. Some formalisms become highly specialised tools, suited for reasoning about specific, well-understood, and inflexible problem areas. Yet others sacrifice their accuracy, or how closely they model their given problem, in an attempt to empower software analysers with very general, and hopefully therefore also useful, reasoning techniques. In this tradeoff between powerful one-off formalisms, and weaker general models, computer scientists are additionally burdened with the *practical* realities of pursuing research or development in the area of such analyses. In this case, some useful compromise is needed between those frameworks with more expressiveness, which can *encode* hard problems, and those with better practical characteristics, which can *solve* large problems.

No single component of the above four-factor tradeoff can be truly ignored in the pursuit of an effective formalism. Consider the need for *general* frameworks. Software analysis encompasses a vast realm of reasoning/understanding techniques that must be applied to computer programs in an attempt to glean arbitrary information. Without generality, a framework can not even concieve of or express the concepts that it needs to analyse, let alone actually find it. The natural opposition to general approaches is that they lack the *specificity* needed to target problems in a fitting manner, i.e. a manner that is sufficiently close to the given problem that the two are conceptually similar, and an expert in the problem would immediately understand the significance of the formalism. Overly general formulations tend to lack the kind of crucial details which are needed to understand the nature of a problem. Without the right tools to reason about a problem, software analysers are stuck ignoring fundamental components of their analyses or attempting to adapt unwieldy general principles to the task, which hampers understanding.

Parallel to the theoretical concerns of generality and specificity are the practical concerns of expressibility and performance. Software analysis is not a pursuit solely limited to research, and the artefacts and systems generated in this field underpin many modern advances in computing, from performance op-

timisations to system security. On the one hand, these technologies depend on details of arbitrary programs, and therefore require techniques with enough *expressive* power to sufficiently capture their semantics. On the other hand, the arbitrary programs are also large and complex, meaning that in order to gain practical advances, *scalability* is needed, i.e. the ability to continue to produce results on increasingly large problem instances

In designing or adopting a formalism suited to research or develop program analyses, a choice is made which weighs the four factors: specificity, generality, expressibility and scalability. Thus, to manage the conflicting requirements of software analysis, we turn to a mathematical problem which balances practical performance, reasonable expressiveness, conceptual closeness and generality. That problem is known as Context-Free Language Reachability (CFL-R).

1.1 The CFL-R Problem

CFL-R is relevant to understanding and reasoning about computer programs, as it is a generalisation of two fundamental computational problems. Historically, the problem was identified as important by Yannakakis [86], called "L-transitive closure" in that work, the transitive closure over some language. It should be noted, given the relevance of the problem to computer scientists, that even earlier examples can be found in the research literature [28].

Yannakakis was primarily concerned with speeding up a class of common queries in graph databases, known as chain queries. A chain query has the following form (as a Horn clause):

$$H(v_0, v_k) \leftarrow B_1(v_0, v_1), B_2(v_1, v_2), \ldots, B_k(v_{k-1}, v_k).$$

I.e. the "chain" of body predicates whose variables connect left-to-right denotes the head predicate with variables for the start and end of the chain. Note that "adjacent" body terms join each other in a left-to-right fashion, and the head result is composed of the first and last "links" in the chain. Chain rules occur frequently in graph databases, as well as more general logic frameworks such as Datalog, as they are used to encode arbitrary transitions of information across intermediate nodes. Given that all rules are in a chain form, we already know how the predicate variables (the v_i arguments in the perdicates) will match up, and we therefore only care about the order of predicates. Hence it is possible to rewrite any chain rule into a simpler format, familiar as a grammar rule [8], as follows:

$$H \rightarrow B_1 \ B_2 \ldots B_k$$

Based on this connection, Yannakakis recognised three variants of the generalised reachability problem, dependant on the properties of the grammar produced by the chain rules in a given problem instance. The simpler variants, which correspond to the regular grammars and the linear grammars (i.e. left- or right-recursive context-free grammars), permit theoretically faster algorithms.

On the other hand, these variants restrict the allowable grammars, which is unacceptable for those analyses which depend on the full expressive power of context-free languages to encode their chain rules. Hence, in this work, we focus on the generalisation of transitive closure over context-free languages.

The usual presentation of CFL-R is as a constrained reachability problem over edge-labelled graphs. For complete generality, we do not place any restrictions on the kinds of graph under examination, hence we assume cyclic directed graphs are being examined. A graph vertex is **reachable** from another if there exists any (directional) path from the other vertex to it. CFL-R is concerned with reachable pairs, rather than actual paths, as it avoids the complication of enumerating the infinite set of paths that arises in the presence of cycles.

Additionally, the graph in question is edge-labelled, thus any path in the graph can be associated with a string of those labels. We generate the path's string simply by concatenating the labels on the path's edges *in order*. Then CFL-R should be thought of as the restriction of transitive closure to those pairs whose connecting path forms a string which is a member of the context-free language. The most general such restricted-reachability problem is the **all-pairs** problem, namely the problem of enumerating all pairs of vertices that are joined by a path spelling a word in the language, though in the research context, CFL-R is often presented as having other variations:

- **Source-Sink**, where we only care if two given vertices have a CFL-R path between them
- **Source**, where all end-points reachable from a given source are needed
- **Sink**, as above for all start-points to a given sink

The latter three problems can be answered by restricting the output set of the All-Pairs solution, hence it generalises them. Since no algorithm currently is known to make the restricted variants faster [86], this work focuses on the All-Pairs problem exclusively.

1.1.1 Motivation

Whilst the historic purpose of CFL-R research may have been for the purpose of optimising graph databases/logical deduction systems, in the modern context CFL-R is most commonly associated with program analysis. Much of how we deal with and reason about computer systems can be conceptually related to graphs. Consider that most compiler technology relies on converting source programs into one of several kinds of graph (abstract syntax trees, control-flow graphs, call-graphs) simply for the purpose of generating source code. It is unsurprising then that CFL-R lends itself well to the task of automatically inferring complex relationships between different parts of computer programs. We therefore see the increasing need for larger and more scalable software analyses as the primary motivation for continued CFL-R research.

Over its relatively brief history, many analyses have been phrased as, or associated with, CFL-R. Early work was carried out by Dolev et al. using

CFL-R as a means of verifying the security of message passing protocols [28]. Much of the recent popularity of CFL-R can be attributed to Reps, together with Horwitz and Sagiv. CFL-R was used to present an intuitive framework for interprocedural dataflow problems [59]. This work was extended to provide specialised (and more efficient) formulations of three common analysis problems: interprocedural slicing, heap-shape analysis, and points-to analysis [57]. The kinds of shape analysis solved by CFL-R are useful for lisp-like languages [55], though other problems in functional languages have also been explored, such as k-deep control-flow analysis [79]. In a modern context, CFL-R analyses have been favoured because any problem encoded in CFL-R automatically allows for solvers using demand-driven algorithms [85], incremental algorithms [46], or interactive techniques [10].

Points-to analysis alone, as a very important problem which underpins many optimisations and security analyses, comprises a large body of CFL-R research [91, 70, 90]. The need for scalability is paramount in points-to analysis, as codebases continue to grow larger and security concerns mount for increasingly complicated pieces of software. To gain a handle on very large problems even more restricted subclasses of CFL-R have been explored, including the Dyck-Reachability problem (i.e. CFL-R restricted to bracket-matching languages) on graphs/trees with the bi-directional property [88, 89]. We provide a more detailed survey of many relevant analyses in Section 2.3.

Whilst the use of CFL-R is common as a means of phrasing and reasoning about the above software analyses, it is far less common to depend on CFL-R for its *solvers*. In most implementations, CFL-R problems are solved by custom-made hand-optimised tools which are specially tailored to only one CFL-R problem class (i.e. it only solves demand-driven points-to analysis, or interprocedural slicing [57]). This seems initially to be counterproductive: problems phrased in a deductive database system such as Datalog usually use off-the-shelf solvers to evaluate them, so why are CFL-R problems different?

The answer is two-fold, and has to do both with the performance of CFL-R solvers and the limitations of the CFL-R formalism. The standard means of solving CFL-R is a summarisation-based approach [47], which is known to have complexity that is cubic in the size of the graph and grammar. Unfortunately, CFL-R is usually adopted for analyses/tools where runtime performance is a primary concern, in which case even cubic complexity is unacceptably slow. By comparison, the approach usually taken is to hand tailor the evaluation of the problem's solution, which involves tweaking the algorithms in a way that capitalises on **domain-knowledge**, such as skewed data or discrepancies in the time spent exploring different parts of the graph. Further, even the CFL-R formalism is often insufficient for capturing the semantics of a given problem in an efficient way. Techniques for refining languages, such as those used by Sridharan and Bodík [68], and techniques for disconnecting/negating paths, such as those used by Xu et al. [84], are not easily phrased or captured by traditional CFL-R. It is not surprising, then, that researchers in CFL-R-like problems often pursue avenues which render their resulting problems unsolvable by traditional CFL-R solvers alone.

Thus the need for continued research is evident. Firstly, CFL-R is a modern and ubiquitous formalism, which underpins technologies that have become commonplace in modern computing. Secondly, CFL-R solvers are unable to handle the kinds of problems that occur in practice, either by being too inefficient to solve them, or by lacking the kinds of semantics needed to phrase them.

1.1.2 Running Example

Since CFL-R is normally associated with software analysis, it is fitting to draw our example from a very practical software analysis task such as *points-to analysis*, also known as alias-analysis, or simply points-to. Broadly, the analysis attempts to determine a heap invariant for a given input source program. The invariant can be thought of as an approximation of the program's runtime memory configuration, i.e. it details which pointer/reference variables "point-to" which memory locations.

Even in reasonably simple programs, points-to calculations can quickly become infeasible. The number of heap objects that a program will create at runtime is, unfortunately, undecidable, so typically some abstractions and approximations are employed to keep the analysis both decidable and feasible. Andersen's abstraction [4] is commonly employed to finitise the number of memory locations; instead of tracking individual runtime objects, we abstract them to their allocation-site. As a result, all objects that are created at runtime from a specific allocation-site will be treated identically (in fact, treated like a reference to some global "version" of that object). Maintaining an accurate memory model, even an abstract one, is still difficult, so different kinds of points-to formulations have been proposed with varying **precision**. Precision refers to the degree of over-approximation, such that an imprecise analysis gives worse upper-bounds on the memory configuration, where a precise one gives a tighter bound. Usually points-to analyses return possibilities, rather than definite answers (i.e. they say that a variable "may point to" a heap object), so the more often an analysis returns the "maybe" answer instead of a definite "yes" or "no", the more imprecise the analysis is. Three common classes of technique for improving the analysis's precision at the expense of increased runtimes are:

- **field-sensitivity**, where the analysis treats each memory object as a collection of disjoint fields. These fields could represent member variables of a class, or array indices. Importantly, when different fields are accessed the analysis is able to distinguish them, whereas a field-insensitive analysis would simply assume all field accesses refer to the same heap location.

- **context-sensitivity**, where the analysis respects calling semantics. Context-insensitive analyses tend to treat all calls to a specific method as though they happen together. This allows for information to flow along impossible call paths, such as entering a method at one call-site, then being returned to a completely different part of the program. Understandably, tracking context information can be very expensive, as a complete call hierarchy is

```
1  public class Obj{
2    public Object x;
3    public Object y;
4    public Obj copyIn(Obj other){
5      Object tx = this.x;
6      other.x = tx;
7      Object ty = this.y;
8      other.y = ty;
9      return other;
10   }
11 }
12 public static void
       main(String[] args){
13   Obj a = new Obj();//H1
14     Object td=new Double(4.3);//H2
15     a.x = td;
16     Obj b = new Obj();//H3
17     Object ti=new Integer(7);//H4
18     b.y = ti;
19     Obj th5 = new Obj();//H5
20     Obj a2 = a.copyIn(th5);
21     Obj th6 = new Obj();//H6
22     Obj b2 = b.copyIn(th6);
23     assert(b2.x != a2.x);
24     assert(a2.y == null);
25     assert(b2.y != null);
26   }
```

Figure 1.1: A snippet of source code in a Java-like language. For simplicity, statements in this language are restricted to avoid the use of temporary registers.

usually exponential in the size of the program, and can contain cycles due to recursive calls.

- **flow-sensitivity**, where the analysis respects instruction ordering, preventing information from flowing "backwards" through the program. Much like context-sensitivity, flow-sensitivity introduces overheads in the analysis (e.g. maintaining control dependencies between program points), and it is unclear when such techniques hamper scalability..

Consider the source program depicted in Figure 1.1. This program contains some key features of typical Java programs, such as the use of fields and member methods. We may wish to perform some analyses which attempt to verify, statically, that the assertions in the program hold. By inspection it is clear to see that they do in fact hold, though in general we would prefer a more automatic technique, as verifying assertions in larger codebases can become difficult. In order to validate the assertions we require a sufficiently accurate analysis, which has an understanding of the program's memory and can determine, based on what the heap contains, whether the individual assertions will pass.

Observe some of the difficulties this program presents to a points-to analysis. Firstly, many assertions contain field loads/stores, implying that the analysis has to at least be able to work in the presence of fields. Even if the assertions themselves did not contain fields, then the analysis still needs to track field information, in case dataflows exist through those fields to the variables in the assertion statements. Hence, some degree of field sensitivity is important in this case if the analysis is to remain precise.

In order for points-to analysis to properly reason about this program, we also expect to require some notion of context sensitivity. The `main` method makes two distinct calls to `Obj.copyIn`. Firstly, we need to handle Java's receiver semantics, as this call is to a member method of some particular `Obj` instances. Secondly, the two calls must be treated distinctly, in order to prevent spurious information flows from entering at one method call and exiting at the

other. With a typical context-insensitive analysis, both calls to `copyIn` would be treated as calling the same code, so the `th5` object, which enters `other` on the call at Line 20, would flow to the return and be assigned to `b2` on Line 22. Hence, the points-to analysis will need to be context sensitive if it is to accurately verify the assertions.

To perform points-to analysis on the above source program, we need to understand how memory-information is created by the program, and how it moves around. Rather than trying to follow the source code, we can visualise the program as a series of variables and abstract heap locations from which we will draw a vertex set. Given such vertices we can construct edges between them according to the statements in the program. Note that, since the edge that is created depends only on the semantics of its statement and not on the order, we will lose flow information in this translation. For simplicity the example analysis will be flow-insensitive, as translation schemes that preserve flow information are more complicated than necessary for this example.

The graphical representation of the source code from Figure 1.1 is presented in Figure 1.2. Heap objects are shown in rectangles, while program variables are circles. Statements in the program are translated to one or more edges according to their semantics. Allocation instructions arise from statements like `a = new B()`, and produce an edge labelled *alloc*. Such an allocation must be assigned to a variable, but variables can be assigned from multiple allocation instructions. We create a unique abstract heap object at each allocation-site, which allows us to distinguish between some different memory objects. Statements with assignment semantics, like `a = b`, produce *assign*-labelled edges. In our example we also treat function returns and parameter passing (including the intrinsic passing of the "this" parameter) as assignments. Hence, our analysis loses the ability to distinguish between different calling contexts, and whilst this makes our example simpler to present and solve, it is nonetheless a context-insensitive formulation. The *load* and *store* edges mark edges arising from an object's field accesses, i.e. `a = b.f` and `a.f = b` respectively. The field-accessing instructions in our program use different fields, for example `other.y = ty` and `a.x = td`, and so we would like to mark this difference somehow. In Figure 1.2 a multiplicity of *load*/*store* are defined, subscripted by the specific field which is being loaded or stored. For example, Line 18 stores `ti` into the `y` field of `b`, hence an edge is created for (ti, b) labelled with $store_y$.

From the graphical depiction of the source code, we can start to infer relationships between distinct program variables by looking at certain paths in the graph. We are interested in points-to information, so the first thing we might note is that *alloc* edges immediately imply a points-to relationship, since the variable that results from allocating a specific heap object must invariably point to it. We formalise this base-case with a grammar rule:

$$\textit{points-to} \rightarrow \textit{alloc}$$

It is helpful to read this rule as "a points-to path exists wherever an allocation path already exists", but more accurately it means that the string "*alloc*" is in

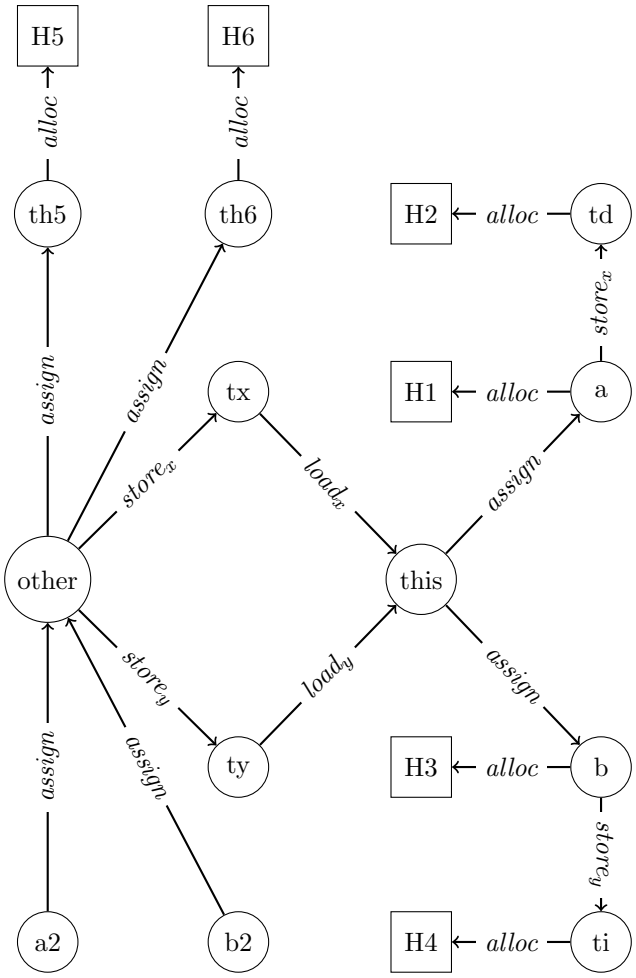


Figure 1.2: A graphical representation of the source code listing from Figure 1.1.

the language of *points-to*. Next we note that assigning one variable to another forces the target variable to point to all heap objects which the source variable already pointed to. Since paths in the language *points-to* already track the correct relationship, we simply extend them by one assignment-like statement using the rule:

$$\textit{points-to} \rightarrow \textit{assign points-to}$$

Again, the interpretation of the above is that two paths, whose associated strings are in the languages *assign* and *points-to* respectively, can be joined to form a new path whose string will be in the language *points-to*. The logic of points-to requires an additional rule in order to handle programs with field accesses. It is possible to write variables into the fields of objects, i.e. into an abstract memory location, and subsequently retrieve them into a new variable. This is effectively an indirect assignment, and it occurs not simply when the fields of the same *variable* are stored to and loaded from, but when the receiver of the load instruction merely **aliases** the store's receiver object, i.e. both receivers may point to the same memory address. For example:

```
o1.f = x; o2 = o1; y = o2.f;
```

In the above, the `o1` and `o2` variables alias one another (i.e. they both refer to the same memory location) as a result of the assignment statement. Therefore, `y` also aliases `x`, as the former is indirectly assigned to the latter *through the field* of the object which both `o1` and `o2` point-to. We re-use the CFL-R notion of paths and languages in order to reason about aliasing variables in the following way:

- Two program variables alias if there are paths from both of them to the same heap object.
- Equivalently, there must be a path, spelling a word in *points-to*, from one to the heap object, then *backwards* to the other.
- a backwards path can be constructed mechanically by looking for strings in a reverse language over edges that have been flipped.

We detail the necessary mechanical transformations which allow for backwards paths to be discovered in Section 6.2.1.1. For now we use $\overline{\text{overbar}}$ notation to denote that a backwards path is required, which yields the rule:

$$\textit{points-to} \rightarrow \textit{load}_f \ \textit{points-to} \ \overline{\textit{points-to}} \ \textit{store}_f \ \textit{points-to} \qquad \forall f \in \text{Fields}$$

Firstly, the above is not strictly a rule, but works more like a template for creating rules based on the fields that the points-to analysis cares about. For Figure 1.2, two rules are needed, one for each field x and y.

With these four rules, allocation, assignment, and two versions of indirect assignment, it is clear to see how paths in the graph relate to points-to relationships. The variable `a` points-to the heap object, called `H1`, that it allocates, which is represented in the path $\langle (a, H1) \rangle$, whose associated string is $\langle \textit{alloc} \rangle$.

Later, `a` is conceptually assigned to the `this` variable when it is used as a receiver object of a call to `copyIn`. As expected, $\langle (this, a), (a, H1) \rangle$ is a path whose labels concatenate to $\langle assign, alloc \rangle$, a word in *points-to*, and therefore denotes that `this` points-to `H1`. We now have enough paths to infer a more complicated indirect-assignment relationship. Since `this` and `a` both point-to `H1`, they are said to alias. More specifically in CFL-R, the "backwards" path $\langle (H1, a) \rangle$ spells the word $\langle \overline{alloc} \rangle$, which is in the reverse language $\overline{points\text{-}to}$. As such, the following path, together with its string, can be seen in the graph:

$$\begin{array}{ccccccc} \langle & (tx, this), & (this, a), & (a, H1), & (\overline{H1, a}), & (a, td), & (td, H2) & \rangle \\ \langle & load_x, & assign, & alloc, & \overline{alloc}, & store_x, & alloc & \rangle \end{array}$$

The above string is in the language *points-to*, hence we verify that `tx` does indeed point to `H2`, which occurs during the first call to `copyIn` with `a` as the receiver variable.

Unfortunately, due to the relatively minimal translation scheme, the above formulation of points-to analysis is overly simplified, and does not accurately capture some program semantics. As we noted, the analysis is not call sensitive, since it treats parameter passing and method returning as though they were assignment statements. The variable `th6` is passed as the parameter `other` to `copyIn` at the call on line 22, but `other` is returned from `copyIn` to the variable `a2` on line 20. In a real execution, the points-to information can actually not flow from `th6` to `a2`, as this ignores call-return semantics and also moves backwards in the control flow. Nonetheless, the path $\langle (a2, other), (other, th6), (th6, H6) \rangle$ exists in the graph, and its associated string, $\langle assign, assign, alloc \rangle$, is in the language *points-to*. Therefore, the analysis incorrectly infers that `a2` points to `H6`, and aliases `th6`.

Incorrect results, such as the spurious points-to relationship described previously, are called **over approximations**, as the analysis is reporting the existence of results which can not, in fact, exist. Since the analysis can produce over approximations, but never fails to report on results that actually do exist (at least, for the fragment of Java-like programs that only contain simple statements and avoid features like reflection, native calls, and arrays), we say the analysis is **sound**. Analyses that only report results that indeed exist but sometimes fail to discover those results would instead be called **precise**.

1.1.3 Formal Definition

We formalise the notation used to refer to CFL-R problems in this work. Whilst there are several equivalent notations that have been used to capture CFL-R, we find this notation to be both concise and clear. Most of the definitions have been adapted from Hopcroft et al. [38].

Definition 1. *Given an **alphabet** Σ, which is an arbitrary set of symbols, then a **string** is a, possibly empty, ordered sequence of symbols from the alphabet: $\omega \in \Sigma^*$.*

*A **language** is therefore a subset of the strings in its alphabet: $\mathcal{L} \subseteq \Sigma^*$.*

Of course, the above definition of languages encompasses far more than we need for this work. We are specifically interested in the context-free languages. There are several equivalent ways of distinguishing the context-free languages from the rest of the class, for example, as the languages which can be *recognised* by a pushdown automata and not a deterministic finite-state automata. For this work we will simply present them as the class of languages whose strings can be enumerated by the transitive productions of a context-free grammar.

Definition 2. *A **context-free grammar** is a 4-tuple $(\mathcal{T}, \mathcal{N}, \mathcal{P}, S)$ of:*

- ***terminals**, $\mathcal{T}$, the subset of the alphabet Σ which is relevant to the language (i.e. which appear in its strings).*
- ***non-terminals**, $\mathcal{N}$, additional symbols which are not present in the alphabet ($\mathcal{N} \cap \Sigma = \emptyset$). These symbols are used as placeholders for sub-strings in the language.*
- ***productions**, $\mathcal{P}$, a set of substitution rules of the form $A \to \alpha$, where $A \in \mathcal{N}$ and $\alpha \in (\mathcal{T} \cup \mathcal{N})^*$. When a production has an empty-string on the right-hand-side, we use the special symbol ε.*
- ***start-symbol**, $S \in \mathcal{N}$, a distinguished nonterminal which is a valid placeholder for the entire language.*

The context-free grammar defines a system by which strings of real (terminal) and variable (non-terminal) symbols can be generated. The step of applying a rule to a string in order to generate a new string is called **derivation**.

Definition 3. *Given strings $\alpha, \beta, \gamma \in (\mathcal{T} \cup \mathcal{N})^*$. A string $\alpha N \beta$ can be replaced by a new string $\alpha\gamma\beta$ in the presence of a production rule $N \to \gamma$. In this case we say it **derives** the new string: $\alpha N \beta \Rightarrow \alpha\gamma\beta$.*

Given the definition of a context-free grammar, we define the **context-free languages** as the set of all languages which can be enumerated via repeated derivations of the grammar.

Definition 4. *A **transitive derivation** ($\overset{*}{\Rightarrow}$) refers to a string transformation that requires zero-or-more single derivation steps. Given $\alpha, \beta, \gamma \in (\mathcal{T} \cup \mathcal{N})^*$, the rules for transitive derivation are as follows:*

- *$\alpha \overset{*}{\Rightarrow} \alpha$, all strings reflexively produce themselves.*
- *If $\alpha \Rightarrow \beta$ and $\beta \overset{*}{\Rightarrow} \gamma$ then $\alpha \overset{*}{\Rightarrow} \gamma$, adding a new derivation step to a transitive derivation is allowed.*

Then the grammar $(\mathcal{T}, \mathcal{N}, \mathcal{P}, S)$ defines the language $\{\omega \mid S \overset{*}{\Rightarrow} \omega, \omega \in \mathcal{T}^*\}$

For completeness, we note that the class of "all languages that can be defined by deriving a context-free grammar" actually includes simpler classes of language, such as the regular languages. The machinery of CFL-R is capable

of handling such a *regular-language reachability* problem, in the event that the grammar used defines a regular language (i.e. one that can be recognised by a deterministic finite state automata [38]). As such, we simplify our presentation in the sense that the CFL-R machinery is expected to handle any language that a context-free grammar can define.

We now turn to the "reachability" component of CFL-R. In our context, we are concerned with edge-labelled graphs. Since the problem searches for paths whose labels concatenate to form strings in the language, it makes sense to label the edges with members of the terminal set $\mathcal{T}$.

Definition 5. *Given an arbitrary set of vertices V, and a set of $\mathcal{T}$-labelled edges $E \subseteq V \times V \times \mathcal{T}$, a **graph** is a pair of vertices and labelled edges, $G = (V, E)$.*

For convenience we assume that labelled edges are actually triples, drawn from the endpoint vertices and a terminal label. An equivalent formalism would be to define a labelling function which maps edges to sets of labels (since it is possible to have multiple labels for a given edge pair). Notationally we write $B(a, c)$ as a short-hand to refer to a B-labelled edge from a to c. Sequences of connected edges are called **paths**.

Definition 6. *A **path** is a, possibly empty, ordered sequence of edges, such that the source-vertex of each edge matches the sink vertex of the preceeding edge (if there is one).*

- *The empty-sequence $\langle\rangle$ is a path.*
- *Every edge $A(u, v) \in E$ forms a path $\langle A(u, v)\rangle$.*
- *$\forall A, B \in \mathcal{T}, w, u, v \in V$, if $A(w, u)$ is an edge, and $\langle B(u, v), \ldots\rangle$ is a path, then $\langle A(w, u), B(u, v), \ldots\rangle$ is also a path.*

Let Π represent the set of all paths.

The set of paths is infinite in the presence of cycles, though this will not be a problem when calculating the solution to a CFL-R problem. Previously, we have informally discussed the notion of a "path's string" as the concatenation, in order, of its labels. We formalise this notion now.

Definition 7. *A **path-word** is the string formed by concatenating the labels of each edge in a path. Given a path $p = \langle T_1(v_0, v_1), T_2(v_1, v_2), \ldots, T_k(v_{k-1}, v_k)\rangle$ then its path-word is $\omega(p) = T_1 T_2 \ldots T_k$. Also, $\omega(\langle\rangle) = \varepsilon$.*

We now have enough to formally define the solution to an instance of the CFL-R problem. Informally, the solution contains all the pairs of vertices in the given graph that are reachable by a path whose word forms a string in the given language.

Definition 8. *Given a context-free language L defined by the grammar $(\mathcal{T}, \mathcal{N}, \mathcal{P}, S)$ and a $\mathcal{T}$-labelled graph $G = (V, E)$ the **solution** of the CFL-R problem* CFL-R(L, G) *is:*

$$\{(v_0, v_k) \mid p = \langle T_1(v_0, v_1), \ldots, T_k(v_{k-1}, v_k)\rangle \in \Pi, \omega(p) \in L\} \cup \{(v, v) \mid \varepsilon \in L, v \in V\}$$

At this stage it is not clear that the solution is computable, let alone efficient to compute. Firstly, the path set is infinite in the presence of cycles, so enumerating all the paths to see which ones have a path-word in the language is not feasible. Secondly, the words in a language are also innumerable, which can be seen from the rules for derivation. Each production rule potentially creates more non-terminals in the right-hand-side of the derivation than were present on the left. Given a cycle of recursive production rules, it is possible to generate terminal strings of arbitrary length, and hence impossible to enumerate them all. Despite this, every instance of a CFL-R problem has a finite solution, which can be computed in a reasonable amount of time [47]: in fact, there are algorithms which have better than cubic time-complexity [17].

1.2 Open Issues in CFL-R Research

The CFL-R formalism is a promising avenue when exploring potential vehicles for computing program analyses. In CFL-R, the phrasing of many analysis problems, particularly those involving constrained information transfer, is straightforward. Several other advantages exist when the CFL-R framework is being used, including composability and on-demandness. CFL-R problems are inherantly composable, due to the fact that context-free languages are closed under union [38]. In short, two languages can be composed by taking the start symbol of each grammar, S', S'', and making a new start symbol for the union-language S with two new productions $S \rightarrow S', S \rightarrow S''$. This allows us to build more complicated analyses simply by re-using prior formulations as non-terminals in the new analysis' grammar. On-demandness results from the fact that the search algorithm for CFL-R paths can be automatically rephrased into a top-down version [57]. Top-down analyses are often called on-demand, as they work by incrementally expanding on an input query *as-needed* in order to limit the scope of their search for results. CFL-R problems can be automatically re-phrased into their on-demand versions, therefore any analysis which can be phrased in CFL-R can be adapted to contexts that favour top-down solutions.

However, whilst CFL-R is a useful framework, there are practical concerns over its use, which limits the impact that CFL-R research can have. Whilst the volume of research literature that is concerned with CFL-R is large, most CFL-R systems are built on an as-needed basis. As such, they are custom implementations which are usually designed to handle a single grammar, and optimised for graph instances with particular well-known characteristics. A clear example of this is the points-to solvers. As shown by the sample problem in Figure 1.2, the points-to problem can be elegantly phrased as a CFL-R instance. However, the need for high performance in points-to analysis solvers is so great that production systems must significantly diverge from the CFL-R formalism in order to maintain acceptable runtimes [27]. This raises several questions in relation to the performance of CFL-R:

- How effectively do general-purpose CFL-R solvers compare to the hand-tailored solvers which are designed for a smaller class of inputs?

- Given that few improvements to the algorithms of CFL-R will likely be found [56, 34], is it possible to trade theoretical complexity for *practical* scalability?

- Are there data-structures or evaluation-strategies which would be more suited to improve the performance of CFL-R solvers?

In addition to the limitations of practical CFL-R evaluation, there is a paucity of tools available to solve CFL-R problems. CFL-R systems see limited adoption, mostly because any general system is poorly adapted to the specific problem context that it would be used in. When general tools are needed, the only realistic approach is to co-opt a more general solving vehicle, such as Datalog [1]. Given that it is always possible to recognise the context-free subset of a Datalog problem [86], it is reasonable to ask whether, and under what circumstances, a CFL-R specific tool would be more useful than a more general one.

Finally, it is not clear whether efficient solvers are enough to allow for effective CFL-R-based systems. Ultimately, the CFL-R framework can be seen as a means of specifying program analyses **declaratively**. Declarative programming systems are designed to alleviate the burden of specifying *how* a solution should be derived, and focus instead on accurately representing *what* the solution is, i.e. as a logical program. Unfortunately, there are several limitations in declarative programs which hamper their use in high-performance contexts. Firstly, such frameworks must be seen as a layer of abstraction, meaning they are likely to suffer inefficient code-generation when translated to an actual running system. Secondly, such frameworks lack the kind of expressive power to encode domain-knowledge into the problem evaluation, where such knowledge is necessary to yield performant systems. In the logical setting, it is difficult to distinguish "core" components, i.e. the computations which account for the most compute time, from less relevant "ancillary" computations, meaning it is difficult to even decide which parts of the CFL-R problem to optimise. It is an open question how best to deal with the inefficiencies of a declarative CFL-R system in a way that achieves high performance.

1.3 Contributions

This work charts progress made towards the completion of a general-purpose, high-performance, intuitive and freely available CFL-R solving framework. Over the course of developing this system, we have examined several angles which contribute to the lack of solvers currently available for CFL-R. Each chapter of this work motivates and examines a specific limitation in the CFL-R context, and details the potential approaches which a CFL-R system might use to cope with them. For this thesis i have worked on the following peer-reviewed publications:

1. "Towards a Scalable Framework for Context-Free Language Reachability", published at the conference on Compiler Construction, 2015 [35].

2. "Giga-Scale Exhaustive Points-To Analysis for Java in Under a Minute", published at the conference on Object Oriented Programming, Systems, Languages and Applications, 2015 [27].

3. "Cauliflower: a Solver Generator for Context-Free Language Reachability", published at the conference on Logic for Artificial intelligence, Programming and Reasoning, 2017 [36].

4. "Feedback Directed Optimisations for Context-Free Language Reachability", submitted for review to the Transactions on Architecture and Code Optimisation, 2017 [37].

We begin our presentation with an overview of the research work that has been done to-date concerning CFL-R. Chapter 2 consolidates the many important developments in CFL-R research and practice, and presents some of the many analyses which CFL-R is used to solve. Work on context-free language recognition and graph reachability is also presented, as these problems are the basis which we generalise to CFL-R.

In Chapter 3 we take an in-depth look at the practical application of CFL-R analysis, through the use case of a high performance Java points-to analysis. Points-to analysis, particularly for exceptionally large Java code-bases such as the OpenJDK library, is exceedingly difficult to scale. We look into the characteristics of this particular problem, i.e. biases in the input graphs and structural differences in the evaluation mechanisms, and determine viable *practical* solutions. As the solver that we derive is specifically catered towards Java points-to analysis, we look at why the solver performs well with a parametric complexity analysis, showing that whilst the theoretical complexity may be bad, in practice the complexity is actually likely superior to a traditional CFL-R approach. Finally, we examine what opportunities there are for generalising these techniques, with a particular focus on adopting better evaluation procedures, data structures, and logical transformations to improve a generic CFL-R engine. This chapter is largely related to work undertaken with Dietrich et al. [27] on the "Gigascale" Java points-to analysis.

Gaining from our insights into the practical realities of analyses based on CFL-R, we explore the potential that improvements to (1) the evaluation strategy and (2) data structures offer to CFL-R, in Chapter 4. Common general-purpose algorithms for CFL-R are theoretically performant [47], however in practice their evaluation strategy is ineffective. We assess the factors that contribute to this inefficiency, including redundant computations, poor cache-awareness, a lack of parallelism, and ineffective use of data structures. Many solutions to these problems have been encountered in the context of Datalog before, so as a first step we look at how effectively the technologies which power Datalog can be applied to CFL-R. In addition to the Datalog technologies, suitable data structures are repurposed from the field of graphics processing. The use of quadtrees forms a major improvement over simpler data-structures, and they are uniquely useful for the kind of binary relational-join operations that CFL-R problems depend on. An experimental analysis of this new CFL-R solver

against the more standard CFL-R algorithms is performed, which verifies that performance improvements can be made practically, despite a worse theoretical time complexity. The bulk of our work on evaluation strategies and data structures has been published as a proceedings article at the conference on Compiler Construction [35].

In a practical setting, phrasing a given analysis in CFL-R is still insufficient to achieve desired performance. After expressing a problem within the CFL-R framework, the tedious and mostly manual task of optimising the evaluation strategies must be undertaken. For this reason, the potential of automatic optimisation of CFL-R specifications is examined in Chapter 5. We first look at the kinds of optimisations which are performed in practice, determining what they are for and how they work. We map this to CFL-R by phrasing these optimisations as linguistic transformations, which can be enumerated mechanically and verified to be *language preserving*. To identify which optimisations will be most useful, we develop an accurate model of execution inefficiency based on wasted computations, or **dead-ends** in a graph-theoretic sense. The motivation for this approach is that, ultimately, the same solution set will be generated in the presence of a language-preserving transformation, and therefore only wasted computations should impact runtime. The execution model is combined with a feedback-directed compilation approach to incrementally improve a CFL-R specification. We examine the effectiveness of this feedback-directed optimisation approach in an experimental case-study, and show that automatic techniques rival hand-optimised efforts. The bulk of this chapter was submitted for a journal publication to the Transactions on Architecture and Code Optimization [37].

The advances made towards understanding the algorithms, data structures and optimisations needed for effective CFL-R are used to develop a general-purpose solver, Cauliflower, in Chapter 6. The tool focuses on combining the advances detailed in earlier chapters, and refining them in a re-usable framework. Firstly, more expressive semantics for CFL-R are proposed formally and we specify how to evaluate them practically. These semantics can be used to phrase many of the CFL-R variations that appear in the research literature, obviating the need to create custom systems to solve these problems. To assist in achieving the kind of specialised high-performance that hand-optimised tools achieve, Cauliflower uses a synthesis approach to develop specialised parallel solver code for a given CFL-R problem. The performance of Cauliflower is evaluated both against Datalog, which is frequently used for solving CFL-R problems generically, and against Gigascale, which represents an idealised high-performance implementation. A concise presentation of the Cauliflower tool was published in the conference on Logic for Programming, Artificial Intelligence and Reasoning [36].

We conclude our work in Chapter 7 with a summary of our findings and an overview of several promising avenues for future work.

Chapter 2

Literature Review

2.1 Foundational Work

The CFL-R formalism is a generalisation of two well-known computational problems, **language recognition** and **graph reachability**. We survey the important historic works done for these fields, as a means of backgrounding our current focus. Since CFL-R was first phrased in connection with Datalog, and many of the advances we make come from or contrast with Datalog, we will also provide an overview of that formalism.

2.1.1 Language Recognition

Context-free languages have been extensively researched as computational models since they were first formalised by Chomsky and Miller in [19]. Formal languages are defined in terms of the automata that is able to recognise them, in this way the context-free languages are defined as those languages recognisable by a nondeterministic automaton with a push-down stack. Context-free languages are a superset of the regular languages, those that do not require the push-down stack, and can not recognise the context-sensitive languages, those that require a fixed-size Turing machine. The simplicity of their recognition algorithms, combined with their expressivity, makes context-free languages suitable for encoding computer programs, hence the syntax of most programming languages is defined by context-free grammars.

The properties of context-free languages have been established in the literature for some time. A sufficient summary of them, and related formalisms, can be found in [38]. The most fundamental are language membership tests, for deciding if a language is context-free. The Pumping Lemma is often used for this task, and is stated in [38] as:

Lemma 1 (Context-free Pumping Lemma)**.** *There exists a constant n such that all strings $s = abcde$ in the context-free language L, with $|s| \geq n$, $|bcd| \leq n$, $|bd| > 0$, imply that $\forall i \geq 0 : s' = ab^icd^ie \in L$.*

Informally we can read this as stating that sufficiently long strings in a context-free language can be "pumped" by repeating two portions of the string an arbitrary number of times. Deciding membership in this way is especially useful, as we can use a similar regular language pumping lemma (which repeats a single substring in the middle of the sentence) to exclude the subclass of regular languages. Besides membership, the closure properties are also well known. Context-free languages are closed under union, substitution, repetition and concatenation, but not under intersection, complementation or difference. We sketch the proofs from [38] here:

Proof. $L_1 = \{a^n b^n c^m \mid n, m \geq 0\}$ and $L_2 = \{a^m b^n c^n \mid n, m \geq 0\}$ are both trivially context-free languages. Their intersection $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$, is not a context-free language due to the pumping lemma.

Consequently, complementation and difference must be disallowed:

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$
$$\overline{L_1} = \{x^n \mid x \in \{a, b, c\}, n \geq 0\} \setminus L_1$$

□

Recognition of context-free languages is a long-researched practical computational problem. The first efficient algorithms for recognition were proposed independently by Cocke [22], Younger [87] and Kasami [41], and they are collectively referred to as the CYK algorithms. We present the tabularisation based algorithm here. Let L be the context-free language defined by the grammar $\mathcal{P}$ in Chomsky normal-form [43] with start symbol S, and let $w = w_1 w_2 w_3 ... w_n$ be a sentence. Excluding the trivial case where $|w| = 0$ and $S \to \varepsilon \in \mathcal{P}$:

$$T_{x,y} = \begin{cases} \{A \mid A \to w_x \in \mathcal{P}\} & : x = y \\ \{A \mid A \to BC \in \mathcal{P}, \exists x \leq k < y \text{ s.t. } B \in T_{x,k} \wedge C \in T_{k+1,y}\} & : x < y \end{cases}$$

$S \in T_{1,n} \Rightarrow w \in L$

It is interesting that this algorithm is known to have cubic runtime. The simple explanation shows that the $|w| = n$ string imposes n^2 table elements to be filled, and each element of this table requires up to n sub-elements to be read from the table (since k ranges between x and y), hence $\mathcal{O}\left(n^3\right)$ operations are required. Note, also, that the algorithm requires Chomsky-normalisation, but this can be relaxed with a modified algorithm due to Okhotin [54].

Valiant later presented a faster algorithm in [76]. The technique relies on reducing recognition to matrix multiplication, which is known to be computable in approximately $\mathcal{O}\left(n^{2.3}\right)$ [23]. Interestingly this shows that context-free language recognition is no harder than matrix multiplication, which grants it membership in the complexity class $\mathscr{NC}$.

2.1.2 Graph Reachability

The problem of determining reachability in directed graphs is one of the most fundamental and well-researched computational problems. Pairwise reachability is also known as **transitive closure** (TC) (or more accurately, reflexive transitive closure), and it deals with the problem of finding, for an input graph $G = (V, E)$, all the pairs of vertices (u, v) such that there exists a path which begins at u and ends at v. Several factors are considered when designing efficient TC algorithms.

The most theoretically efficient TC algorithms derive from matrix multiplication [31]. We outline the procedure here. Given a directed graph we must pre-process with (1) cycle elimination and (2) topological sorting, which can be done in less than $\mathcal{O}\left(n^2\right)$ time. Now, the adjacency matrix of the graph is of the form $\begin{bmatrix} A & C \\ 0 & B \end{bmatrix}$, an upper-triangular matrix where A and B are also upper triangular matrices. Let X^* be shorthand for multiplying the $n \times n$ matrix X by itself $\log n$ times. Then $\begin{bmatrix} A & C \\ 0 & B \end{bmatrix}^* = \begin{bmatrix} A^* & A^*CB^* \\ 0 & B^* \end{bmatrix}$. Applying this argument recursively yields that the $\log n$-th power of the adjacency matrix can be calculated in $\mathcal{O}\left(BMM\right)$ operations, i.e. the number of operations needed by the fastest boolean matrix multiplication algorithms. It can be shown that the $\log n$-th power of an adjacency matrix with all diagonal bits set corresponds to the reflexive TC of that matrix. Hence, TC is no harder than boolean matrix multiplication, which, at time of writing, can be solved in $\mathcal{O}\left(n^{2.31}\right)$ [23].

Algorithm 1 Nuutila's successor-set merging algorithm for transitive closure.

Require: (V, E) is acyclic
  function NUUTILA(V, E)
    $S \leftarrow \emptyset$
    while $\exists v \in V$ s.t. $v \notin S \wedge \forall (v, u) \in E : u \in S$ **do**
      $S \leftarrow S \cup \left\{ v \mapsto \bigcup_{(v,u)\in E} \{u\} \cup S(u) \right\}$

The relationship with matrix multiplication indicates that TC can be computed in a theoretically efficient manner. Unfortunately, the constant factors associated with fast matrix multiplication algorithms, such as the one by Coppersmith and Winograd [23], are prohibitively large. Work by Nuutila [52] focuses instead on algorithms which are efficient in a practical setting. Nuutila's approach is reproduced in Algorithm 1. This algorithm constitutes a straightforward approach to computing transitive closure via successor-set merging. Firstly, its runtime is theoretically worse than the matrix-multiplication approaches, having $\mathcal{O}\left(n^3\right)$ time complexity. However, many practical optimisations can be made that improve the algorithm's runtime, such as using concise bit-sets to represent successors, ordering the merging steps to take advantage of cache locality, or partitioning the graph into parallel executions.

2.1.3 Datalog

Datalog is a language for expressing computational problems in a simple logical framework [1]. Datalog derives from a more expressive computational logic called Prolog, which was an early example of the purely-declarative logic programming languages. Importantly, Datalog differs from Prolog in that only a subset of semantics are supported, which are necessary to guarantee that every Datalog query terminates [63]. Despite guaranteeing termination, Datalog is P-complete [63], implying it is powerful enough to encode and solve every polynomial-time algorithm. Briefly, Datalog is given assertions in the form of *facts* and *rules*:

$$F(foo, bar). \qquad B(x, y, z) \leftarrow A(x, z), C(y, x).$$

Facts indicate that the given predicate is true for the arguments in its body, whilst rules derive additional truth. If there exists a variable substitution of the predicate arguments on the right-hand side, such that all the right-hand predicates can be shown to be true, then the left-hand predicate (with the same variables substituted) must also be true. The collection of true predicates given by the asserted facts form a subset of the knowledge base called the *existential data-base*, while the derived truths form the *intensional data-base*.

Importantly, CFL-R problems capture a certain sub-class of Datalog, specifically the class of binary relations over chain-rules [86]. A relation is binary if its predicates all have two arguments, and a rule is a chain rule if its right-hand side contains relations that join with each other in sequence, and its left-hand predicate's arguments are the first and last elements of the sequence. The following generalises a chain rule:

$$H(v_0, v_k) \leftarrow B_1(v_0, v_1), B_2(v_1, v_2), \ldots, B_k(v_{k-1}, v_k).$$

Yannakakis originally specified how to translate this subclass of datalog into CFL-R [86], which allowed for the more performant CFL-R algorithms to be used in solving it.

Datalog has recently become popular as a means of specifying program analysis problems. The first improvements came from Whaley et al. [82], in the use of binary decision diagrams (BDD). A BDD is a sparse representation of a strictly-ordered integer set, supporting membership queries in logarithmic time. Given a mapping from predicate values to integers, any datalog relation can be concisely stored in such a BDD. Many operations, such as relational joining and membership testing, can be handled efficiently by BDDs, though their exact performance is very sensitive to a factor called **variable ordering**, which specifies the order in which binary decisions are made in constructing the diagram. Choosing a poor ordering can lead to large differences in the size of the decision diagram, which reduces the overall performance of diagram queries. Their work culminated in the development of a Binary Decision Diagram-Based Deductive Data-Base, BDDBDDB, which performed well in practical experiments and could solve large analysis problems.

The sucess of declarative analysis approaches led to the development of the DOOP framework [66]. Though program analysis was presented more as a use case for BDDs, the DOOP framework was foremost an analysis system. The advantage of DOOP over previous approaches to program analysis is that the analysis logic could be specified in a relatively concise and flexible language, and the intricacies of evaluating this logic could be deferred to a high-performance Datalog solver. DOOP makes use of the proprietary Datalog system Logicblox [5], which is an extension of pure Datalog with function symbols, limited function programming, and some syntactic conveniences.

Some notable examples of freely available Datalog solvers also include the engines Z3 [49] and Soufflé [64]. Z3 is actually a solver for the more expressive class of Satisfiability Modulo Theories (SMT). In fact, the expressive power of SMT means that Z3 is regularly used not merely for advanced forms of program analysis, but also many synthesis and verification tasks. Due to the need for high performance datalog systems, Z3 provides a specialised implementation of its fixpoint logic solver, called μZ. Whilst Z3 is extensible and well-known, its runtime performance is not as impressive as the more specialised Datalog system Soufflé (see Chapter 6). Soufflé uses a synthesis approach to generate solvers from an input specification. The added overhead of synthesising a solver is reasonable in this use case, as the Datalog specification itself changes rarely, whilst the existential data-base which it runs with changes frequently [64]. Statically synthesised solvers have significant runtime advantages over runtime systems (like Logicblox, Z3, and BDDBDDB), as they avoid the need for dynamic runtime control mechanisms and can generate more cache-aware code. The synthesis approach of Soufflé was very influential in the design choices made for the Cauliflower solver.

2.2 Algorithms

The standard approach to solving general CFL-R problems (i.e problems where the grammar itself is an input) is commonly attributed to Melski and Reps [47]. Whilst this is one of the first complete and explicit presentations of the algorithm, it should be noted that cubic-time solvers were known for special cases much earlier [28], and general cases not long afterwards [86]. The algorithm has cubic time-complexity. An intuitive explanation is that it can create at most $|V|^2$ results, and finding each result may require searching $|V|$ vertices for an end-point. Since the grammar is itself variable, a more accurate time-complexity is parameterised by the size of the grammar k, which yields $\mathcal{O}\left(n^3k^2\right)$

The Melski-Reps algorithm was considered optimal [34] for most of its history, and was only improved upon relatively recently. A sub-cubic algorithm was developed by Chaudhuri [17], which improved the runtime to $\mathcal{O}\left(\frac{n^3}{\log n}\right)$. This improvement relies on the "four russians' trick" [6], a generic means of performing set-operations in aggregate. Informally, if A and B are two sets, they can be represented as a bitvector in $\mathcal{O}(n)$ space, then for a machine to be able to address its ram, which is surely larger than n, it must have $\log n < w$-sized words,

Algorithm 2 The summarisation-based algorithm due to Melski and Reps [47]

Require: $\mathcal{P}$ is in binary normal-form [43]

```
1:  function MelskiReps(T, N, P, S, V, E)
2:      R ← E
3:      for all A → ε ∈ P do
4:          R ← {A(v, v) | v ∈ V} ∪ R
5:      W ← R
6:      while W ≠ ∅ do
7:          X(u, v) ← dequeue an edge from W
8:          for all w s.t. H → YX ∈ P ∧ Y(w, u) ∈ R ∧ H(w, v) ∉ R do
9:              W, R ← insert H(w, v)
10:         for all w s.t. H → XY ∈ P ∧ Y(v, w) ∈ R ∧ H(u, w) ∉ R do
11:             W, R ← insert H(u, w)
        return R
```

thus $A \cup B$ and $A \cap B$ can be computed in $\mathcal{O}\left(\frac{n}{logn}\right)$ time, using word-length bitwise operations. The trick also applies to Turing machines, since it is possible to pre-compute a lookup table for the word-sized operations in $\mathcal{O}(n)$ time and space. Chaudhuri applied this approach to the matrix-multiplication-like dynamic transitive closure needed fro CFL-R (though he was specifically dealing with a related problem known as recursive state-machine/pushdown automata reachability). Unfortunately, the trick relies on a dense representation of the input problem. Whereas Melski-Reps can use a sparse representation, which is only $\mathcal{O}\left(kn^3\right)$, the Chaudhuri algorithm is actually $\Theta(kn^3)$. We perform statistical analyses of some common CFL-R problems in Chapter 4, which indicate that Chaudhuri's approach would be prohibitively space-intensive to solve.

2.3 Applications

One of the earliest applications of CFL-R research was in the verification of network security protocols. Dolev, Even and Karp identified an important class of network protocols called "ping-pong" protocols [28], which worked by sending messages in a strictly back-and-forth pattern between participants. The primary concern is that such a protocol could be subverted in a way that would allow malicious parties to trick other users into decrypting messages on their behalf. The authors represented a given message protocol as a graph, where transitions were labelled with operations (encrypt, append a username, remove padding, etc.). Clearly combinations of operations (encrypt with k_{pub} and decrypt with k_{priv}, appending a username and removing a username) yielded the same text afterwards as before. The task then was to find a path in the graph which, in combination with the operations performed first by an honest participant, could yield the plaintext of their message. At the time, the best known algorithms for this verification problem were $\mathcal{O}\left(n^8\right)$ in the number of operations, so translating

to the CFL-R context automatically gave a cubic-time algorithm.

Much of the popularity of CFL-R can be attributed to Reps et al., who presented CFL-R as a vehicle for performing program analyses, particularly dataflow analyses [57, 59]. Dataflow problems track the state of a particular *dataflow fact* as it flows through the program. To convert this to CFL-R, we begin by computing the control-flow graph of a program. Each control node v is represented by a series of vertices $v_0, \ldots v_s$, each of which represents a potential state of the dataflow fact. If there is a transition between two nodes (v, u) in the control flow graph, then we make a subgraph such that (v_i, u_j) is an edge in the expanded graph when state i can transition to j during the control-flow transition (v, u). Clearly this model of dataflow is generic, but it is also readily solvable via TC algorithms, as the edges do not have labels. CFL-R is used to ensure *context-sensitivity*: instead of using unlabeled transitions, we label method calls/returns with an open/closed parenthesis for their callsite, and intra-procedural transitions with an arbitrary symbol n. Then, given the language:

$$S \to \varepsilon \mid n \mid SS \mid ({}_i S)_i \qquad \forall i \in \text{call-sites}$$

We have that $S(v_i, u_j)$ in the CFL-R solution implies that the dataflow fact in state i at point v in the program can reach point u in state j, and that this transition will not violate call/return semantics.

Shape analysis is a technique particularly relevant to imperative and functional programs. The analysis attempts to understand the heap-shape of different data structures, i.e. to determine whether they are cyclic, or behave like trees, or lists [60]. In this way, a heap-shape analysis can be used to recognise data-structures by their structural properties, which would allow the compiler to substitute more efficient structures where it sees fit. Heap-shape analyses are also used in termination analyses, for example a loop over a cyclic list may not terminate. Reps formulated the shape analysis problem as a CFL-R instance [55, 57], which automatically presented a (nearly) cubic runtime complexity for shape-analysers and allowed a demand-version of the analysis to be phrased (i.e. a top-down solver, see Section 1.2).

Another important component of accurate program analyses is the issue of control flow. Many formulations of flow analyses are not context sensitive, in that they do not accurately deal with the flow between function calls and returns. This problem is not limited to imperative programs, but is also visible in functional programs, where call/return is the most predominant flow mechanic [79]. It is desirable, therefore, to ensure that functional control-flow analyses are actually context sensitive. Vardoulakis and Shivers present the first truly context-sensitive flow analysis, CFA2, as a CFL-R problem [79]. The analysis uses call/return parenthesis-matching, similar to the way that Sagiv et al. formulate the matching callsite problem for dataflow [62]. In a preliminary analysis of Scheme programs, the CFA2 analysis is shown to be far more accurate than tradidional depth-bounded flow analysis frameworks like k-CFA.

Returning to the security context, CFL-R has seen recent attention as a vehicle for performing **taint analysis**. Taint analysis is a style of information-

flow analysis, particularly relevant to internet and smartphone applications [7], which focuses specifically on whether so-called "secure" program data (such as phone contacts, secret keys or session information) is able to flow from a *taint source* to a publicly readable sink. These systems are heavily optimised for speed and precision on typical use cases, so they differ greatly from the underlying logic. Nonetheless, many taint analyses can be phrased as CFL-R problems. Bastani et al. develop an extension to a taint analysis problem, which relies on CFL-R to infer specifications for their input programs. Normally, an analysis framework which is unable to reason about some specification logic will over-approximate it (such as conservatively assuming that after a reflective call, every method has been invoked), however, greater accuracy is possible if we try to infer specifications around these "invisible" portions. Given an unknown code portion, the tool mechanically generates several likely CFL-R subproblems, posing these to a human auditor to select the best candidate. This is uniquely possible for CFL-R, as problem instances are simple enough to mechanically generate and reason about, yet remain appreciable to humans for understanding and verifying.

2.3.1 Points-to Analysis

The points-to problem is a very common analysis problem, which is central to many optimisation passes and often forms the base of more complicated analyses like taint analysis. Given an input computer program, points-to analysis attempts to determine a memory invariant, i.e. a sound over-approximation of the program's runtime memory configuration. Specifically, the analysis tracks pointer variables in the source code, and determines which memory locations (usually abstract locations, since even the number of memory locations is undecidable in general) are pointed-to by which pointer variables. Points-to analysis was presented as our running example in Section 1.1.2. Traditionally, a common way to phrase points-to analysis was via an inclusion-based constraint system [4]:

$$\texttt{x = \&y;} \Rightarrow \{y\} \subseteq pt(x) \qquad \texttt{a = b;} \Rightarrow pt(b) \subseteq pt(a)$$

$$\texttt{v = *l;} \Rightarrow \bigcup_{i \in pt(l)} pt(i) \subseteq pt(v) \qquad \texttt{*s = u;} \Rightarrow \forall i \in pt(s) : pt(u) \subseteq pt(i)$$

The subset-based approach is slower but more precise than so-called "equality-based" analyses [71], which equivocate points-to sets into disjoint groupings.

Points-to analysis is arguably one of the most fundamental program analyses, which forms the basis from which to answer most reasonable analysis questions, such as dataflow [62], and information-flow [61]. Further, points-to information is used to provide static over-approximations of a program's call-graph, known as call-graph construction, which allows subsequent analyses to reason about the likely landing methods of a given call, which may involve function pointers or virtual dispatch. For this reason the body of literature on points to analysis is large [67, 45, 33, 32, 15, 25, 18, 48]. We give a brief overview here of several important factors that pertain specifically to points-to analysis:

- **Context-sensitivity** refers to the ability of a program analysis to distinguish between analysis results based on different calling contexts. There are several approaches to achieving context sensitivity in a given analysis. *Summarisation* is where the analysis is performed for a given context, and its results are inline at different callsites [53]. *Cloning* is where the analysis is performed multiply, once for each callsite [83]. Note that these approaches are not absolute, scalability is often difficult to guarantee given arbitrary cloning/summarisation [67], so often the analysis only appliex context-knowledge selectively, possibly based on heuristics about important parts of the source code. For the purposes of our discussion, we will include **heap-sensitivity** and **object-sensitivity** as kinds of context-sensitivity, in the sense that "context" refers only to a restriction of analysis results based on some program factor(usually callsites, but in this case heap-states and callee-objects respectively). Heap-sensitivity attempts to reduce pollution of the analysis' abstract heap by differentiating between memory objects allocated by the same source level instruction, which we discussed earlier was a key simplification made by Andersen's analysis [4]. Object sensitivity is similar to the above callsite-based context-sensitivity, though context is derived from a call's receiver object, instead of the call instruction. Using receiver objects can be beneficial, particularly for languages like Java which make heavy use of virtual dispatch.

- **Flow-sensitivity** is relevant where analysis results encode temporal relationships between results. For simplicity, it is common to develop an analysis which does not track the order of instruction executions, in which case the program semantics are to execute any in struction at any time. Flow-insensitivity is an over-approximation which may conclude, for example, that data is shared between given variables even if one did not hold the data *at the time* it was copied to the other. A popular way to cheaply provide partial flow-sensitivity is to rely on an intermediate representation based on static single-assignment [44] (SSA). Programs in SSA treat all variables as constant valued, by creating artificial variables at program points where source-level variables would need to be modified, or where several values reach the same program point via different control-flow paths (using special *phi* instructions). SSA is usually strictly a register-level construct, and does not provide the same flow-sensitive guarantees to heap objects. For this reason, analyses which leverage SSA in LLVM, for example, only provide partial flow-sensitivity, for variables that are not stored-to/loaded-from the heap.

Early work by Reps phrased the inclusion-based analysis as a CFL-R problem [57] which allows for a simple presentation of a points-to solver (based on the summarisation algorithm). Importantly, CFL-R formulations make explicit that operations like set merging and iteration over points-to sets are not actually necessary to the analysis (though in performant implementations of Andersen's analysis, such operations were avoided by relying on lazy evaluation techniques).

Owing to the difficulties of evaluating the general case of CFL-R, researchers have turned to more restricted classes of CFL-R in pursuit of better evaluation techniques. One important sub-class is the **Dyck-reachability** problem, which is believed to cover a large portion of practical CFL-R analyses [42]. Dyck-reachability formalises the notion of parenthesis-matching in CFL-R. Given a set of brackets, this reachability subclass restricts the grammar to exactly:

$$S \to SS \mid \varepsilon \mid ({}_i S)_i \qquad \forall i \in \text{brackets}$$

In truth, despite the restriction, there are no algorithms known to solve the Dyck language problem over general graphs any faster than for arbitrary context-free languages. To make algorithmic progress, researchers also restrict the class of graph under consideration. Initial progress was made by Yuan and Eugster for bi-directional trees [88]. The bi-directional property requires that for every terminal-labelled edge $({}_i(s,t)$, there exists a matched reverse-edge $)_i(t,s)$. Similarly, closing parentheses should match opening ones in reverse. Trees with this property actually occur naturally in some object-flow problems (i.e. analyses involving flow of information between types) specifically for type heirarchies. By exploiting symmetry properties in the paths that are found by Dyck-reachability on bi-directional trees, Yuan and Eugster were able to formulate a fast online algorithm with $\mathcal{O}(n \log n \log k)$ pre-computation for queries in $\mathcal{O}(1)$ time. This work was subsequently extended to bi-directional graphs, and an algorithm with superior runtime on trees and good runtime on graphs was devised, by Zhang et al. [89]. Those authors noticed that the reachability relation was actually an equality relation in bi-directional graphs which allowed them to use disjoint-set structures to record co-reachable classes of nodes. Their presented algorithm runs in $\mathcal{O}(n)$ time on trees, and $\mathcal{O}(n + m \log m)$ on graphs.

An important consideration for alias analyses is the size of the result sets. Typical CFL-R analyses assume a bottom-up approach, owing in part to the fact that the standard algorithms (Both Belski-Reps and Chaudhuri's, see Section 2.2) compute the all-pairs solution to the CFL-R problem. For performance reasons, computing every pair of language-reachable vertices may be prohibitively expensive, as over-approximations in the points-to formalism, coupled with the sheer size of the input program, significantly increase the size of the CFL-R solution. However, CFL-R problems also permit *top-down* solvers, i.e. solvers which only derive the minimal number of pairs needed to verify a particular pair, called the *query*. Such solvers can be automatically derived simply by converting the CFL-R problem back to Datalog and using existing top-down solvers from that technology, or using a transformation logic like Magic Sets or Subsumptive Tabling [75]. However, more efficient solvers can be made by examining the specific evaluation logic for CFL-R based points-to and adapting machinery. One early approach was described by Sridharan et al. [70], and described a demand-driven analysis for Java points-to analysis. Subsequently, a version of a CFL-R-based demand analysis for C programs was developed by Zheng and Rugina [91]. Importantly, though their implementation of the demand algorithm is time-bounded (so the analysis conservatively reports "maybe" when it runs out of time), they are able to achieve very high

precision with a relatively short timeout of 0.5ms on the SPEC2006 benchmarks. Finally, the demand analyses were extended to facilitate context sensitivity by Yan et al. [85]. The search algorithm is complicated by the need to ensure context sensitivity, meaning that timeouts are needed to prevent infinite recursions. Nonetheless, their results give precision improvements on average over two state-of-the-art points-to analyses for Java.

The need for precision in points-to analysis has led some researchers to examine the linguistics of CFL-R closely. Sridharan and Bodík developed a technique for feasibly managing context-sensitivity by language refinement [68]. One interesting property of context-free languages is their intersection; it is known that the intersection of two context-free languages is not necessarily itself context-free, despite that the intersection of two *regular* languages is itself regular [38]. A direct result of this is that it is impossible to phrase an analysis which is both completely field sensitive and completely context-sensitive [58], one or both of these mechanisms must be weakened. Interestingly, though, the intersection of a regular language and a context-free language *is* context-free. Sridharan and Bodík describe how to approximate context sensitivity (i.e. accurate for a c-deep call-stack) with a regular language, and therefore gain context sensitivity with this approximation. Further, the accuracy of the analysis could be iteratively improved, simply by increasing the precision of the context-sensitive component, hence this is a refinement technique. Their analysis was incorporated into a cast-safety checker for Java programs, and was able to verify the safety of more casts than a context-insensitive analysis and a competing sensitive analysis.

Related to the notion of demand-driven analyses is the concept of incremental analyses. In some contexts, such as IDEs, the analysis's code base changes frequently in small ways. Where this happens it would be computationally wasteful to throw out previously computed results and start again from scratch for every minor change. A better solution would be to somehow *update* a partially-completed analysis with new information. Incremental approaches to transitive closure have been examined by researchers for a long time [3], as they are necessary for many data-processing applications. Lu et al. propose an adaptation of the CFL-R algorithm for incremental evaluation [46]. The transformation is fairly straight-forward, as new terminals can be simply added to the graph and Melski-Reps' worklist (see Algorithm 2), which will derive any new edges. The *removal* of terminal edges introduces some complications to this technique, where the authors suggest partial re-evaluation of the graph by looking at the subgraph connected to the edge being removed.

Chapter 3

Applications

This chapter examines the practical intricacies of CFL-R based analyses on realistic benchmarks. The majority of this work was published to OOPSLA, 2015, in the paper "Giga-Scale Exhaustive Points-To Analysis for Java in Under a Minute" [27].

3.1 Java Analysis

Java is a common programming language which features in many high-performance settings, particularly on web servers and smartphone applications. Indeed, the ubiquity of Java programs is itself enough to justify study into effective analyses, as the security and performance of the Java language is one of the more important issues facing computing practice. However, part and parcel with being ubiquitous, Java applications are the focus of significant security attacks [21] which cover a diverse range of attack vectors, meaning that effective security analyses must deal with as many features of the Java language as attackers can. Aside from being prime targets, Java applications are uniquely vulnerable by virtue of incorporating large and readily pluggable libraries. Attacks are possible from two directions, then: there are opportunities for users of an untrusted library to have their program act maliciously when supposedly safe operations are used, but there is also the chance that library code might be subverted by malicious applications. The latter of these concerns is of great concern for the OpenJDK library, sometimes called the standard/runtime library, as its code forms the basis of all Java programs, and is expected to manage very privileged data such as Java's security model [21].

The immediate question becomes, how can we analyse such a program as the OpenJDK. Any analysis must deal with (1) a very large codebase, with millions of variables/instructions and hundreds of thousands of methods, (2) a library, meaning that actual application code is missing, and must be conservatively approximated, and (3) a feature-rich problem, which makes use of all Java semantics (after all, it *defines* those semantics). Security exploits present

an intricate interplay of all three components in their attacks, where they make use of diverse parts of the library (which developers never expected to be used together), contrive perverse application code whose execution patterns are difficult to predict, and exploit multiple features of the Java language, particularly reflection and native methods. The complications imposed by this use case are therefore interesting not only for the OpenJDK, but in the development of any scalable and accurate analysis.

In this chapter we focus on the particular use-case of points-to analysis for large Java libraries. The points-to analysis problem is one of the most fundamental program analyses, and it underpins many compiler optimisations and automated security verifiers, which production systems rely on. Given a Java program, we must compute an approximation of the program's memory configuration, in which the potential heap objects that a given reference variable may "point to" at runtime are discovered. Since this problem would be undecidable in the face of the unbounded memory requirements of most non-trivial programs, we desire two properties to make the analysis feasible and useful: the heap locations should be abstracted (in our case by the instruction which allocated them [4]), and the analysis must over-approximate the true result (i.e. it must be sound). The first requirement is needed to keep the analysis decidable, and is one of the most common abstractions used in points-to analysis, though another recently-popular method of abstraction is the access-path abstraction [18], which treats concatenations of field accesses as abstracted heap objects. We desire a sound over-approximation simply because this is necessary if we want to make use of an analysis' results; if the tool returns spurious answers and does not return some valid results then far more manual effort is needed to verify its output.

One particular feature of Java which we must deal with at this point is the *virtual dispatch* mechanism. Give a variable v with statically declared type T, calls to the instance method m (i.e. `T v;...v.m();`) will not necessarily land in the declaration of m in the class T, even if there is such a method, but may in fact land in any sub-class of T, T', that *overrides* m. Immediately we have two potential abstractions of the virtual dispatch mechanism, a simpler one based on the conservative Class Hierarchy Analysis [26] (CHA), and one using approximations of runtime type information, called Rapid Type Analysis (RTA). For CHA, we take the conservative assumption that a variable can be *every* overriding subclass of its statically declared type (with the unfortunate corollary that `Object` variables are every type). The call graph can be statically wired up, but each call-site will lead to multiple overriding methods For RTA, we look at the declared types of the heap objects that are pointed-to by reference variables. Since heap objects are abstractions of allocation instructions (i.e. a call to `new X();`) we know their static type. Some additional runtime logic is needed to "wire up" the virtual calls on-the-fly, such that if we discover that v points-to a heap object of type T'', then only the version of m which T'' invokes can actually be the landing of a call `v.m()`.. It is of interest which of these techniques is actually faster in practice: since CHA is more precise, it reduces the computations needed to derive solutions, but it also requires the call-graph

to be regenerated dynamically, adding to the runtime overhead.

3.1.1 Library Support

One issue that is rarely dealt with in the normal points-to analysis literature is the so-called "open world" assumption inherent to static analysis of dynamically linked objects, like Java libraries. Typically, points-to analysis deals with applications, e.g. in the context of optimisation or security verification. In this context the class hierarchy, and therefore the virtual dispatch tables, are fixed and known statically. When examining libraries in a stand-alone manner, particularly Java libraries, we can make few guarantees about the client program's runtime behaviour, and therefore we must make conservative assumptions when analysing the library.

Consider a Java library call like `DateFormat.format(Date d)`. Users are free to override `java.util.Date` in arbitrary ways, so it is not accurate to assume that invoking something like `d.getTime()` will actually result in a call to the method `java.util.Date.getTime()`. Few safe assumptions can be made when invoking methods on objects which were given as a parameter in a publicly accessible method, though program constructs like final objects or constness may help.

Unfortunately, a true conservative over-approximation for application behaviour would be the union of all possible behaviours. As stated before, calling any method on a user-provided object, can lead to arbitrary behaviour. In fact, such a call may result in *every* publicly accessible library method being invoked, regardless of the return types or calling requirements of those methods. Such a case is unlikely, nonetheless it must be dealt with semantically. Importantly for the analysis, though, we can avoid this pitfall. Firstly, since this is a library analysis, we must do away with the notion of a single entry-point, and consider that any public method of a public class might be the entry point. This being the case, however, it does not matter if methods called on user-given objects call every public method, since the analysis already assumes that those methods might be the *original* entry-point of the application code. In this way, any change to the state of the library that might occur as a result of calling the user-defined method, is already handled by the parts of the analysis that assumed a different entry point. To summarise, it is sufficient to treat every public method as a program entry point, and ignore the fact that user defined methods might call into them.

3.1.2 Limitations

Unfortunately, given the scale of the OpenJDK library, some important analysis features will be ignored in this presentation. Most prominently, this analysis will make no attempt to address context sensitivity. In principle, any context insensitive analysis can simulate context sensitivity by techniques that inline function calls (though no actual inlining will take place, things will appear to be inlined in the call graph). This technique does not work for recursive

methods, for which a suitable approximation must be used, but it is otherwise sound and improves the precision compared to a context insensitive approach. However, two factors significantly degrade the runtime performance (which we foresaw would be insufficient for analysing the OpenJDK). Primarily, inlining methods leads to an exponential expansion in the apparent size of the code base. Each method is cloned potentially at every call-site, and this cloning occurs inside those clones, and so on. The resulting exponential expansion in the size of the CFL-R graph would lead to an unsolvably large problem. Further, though context sensitive analyses are more precise, so the intuition is that they should be faster [67], actually they typically exhibit much larger points-to sets. Though context information improves accuracy and reduces the number of points-to relations compared with insensitive analyses, this reduction is only visible when context information is elided from the points-to sets. During the analysis' execution, a given variable (which has been duplicated over every valid context) may point to fewer heap objects, but those it does point to will be duplicated over several contexts. Hence, the size of its points-to set may in fact be larger. Ultimately, the significant increase in computational load associated with context sensitivity renders this technique infeasible for large Java library analysis without further improvements to the technology.

Whilst our analysis does reason about virtual dispatch mechanisms, this does not extend to a precise treatment of types in all factors of the analysis. Notably, we treat the casting operations as semantically equivalent to an assignment. In a normal analysis, the static knowledge that `x = (Foo)y;` would allow us to selectively propagate points-to information to x. Not every heap object that y points-to can be reached from x after this operation, as any object with type T which is not a subtype of `Foo` would cause that statement to throw a class cast exception. Casting should therefore be treated as a points-to propagation filter. In the CFL-R setting, designing semantics for this kind of cast is trivial, and uses the same trick as matching field accesses. First, let every heap-object's vertex v be associated with self-loops $T_x(v, v)$ for every reflexive supertype x of the allocated type of v. This assumption simplifies the presentation, but it is also possible to compute the necessary loops given only the allocated type and an inheritance tree as part of the input CFL-R problem. Then, the rules:

$$\textit{points-to} \rightarrow \textit{cast}_x \ \textit{points-to}\ T_x \qquad \forall x \in \text{Types}$$

conditionally propagate points-to paths when the allocated type is a subclass of the cast's type. Unfortunately, computing this information on-the-fly during analysis of the OpenJDK is prohibitively expensive. As we shall see, there are significant advantages to be made by exploiting symmetries in the propagation of information by assignment [33]. Ultimately, the precision advantage that would be gained by accurate treatment of casts was deemed less important than the performance advantages of simplifying this language feature.

Finally, due to the overriding need for performance, particularly on the target library of the OpenJDK, our presentation focuses on one *specific* formulation of points-to analysis, to the exclusion of more general approaches. We need to

make significant modification to the traditional Melski-Reps algorithm in order to meet performance requirements. Primarily, the summarisation approach is limited by the need to update and, ultimately, exhaust a worklist. The number of points-to relations in the OpenJDK library is large (on the order of billions), especially when compared with the number of terminal-labelled edges (less than two million). As a result, the worklist grows quickly, and dequeuing edges for checking becomes prohibitively expensive. Further, the worklist is, essentially, non-deterministically ordered, and so no guarantees can be made that queued items will be dequeued in a useful order. Considerations, like the need for cache locality, are paramount in ensuring high performance targets are met. For this reason, we fix the evaluation logic with the intention of examining closely where the opportunities are for speeding up that logic specifically. The formalism we will use closely models the simple example formulation given in Section 1.1.2. When effective means of improving the evaluation are discovered, we can then look into generalising our results.

3.2 Gigascale Analysis

The Gigascale analysis [27] examines the points-to problem in an attempt to understand how a specific logic can be intricately optimised. The overriding concern in the development of this analysis is **scalability**, that is, the ability to run on extremely large data sets. To this end, we ignore concerns for theoretically efficient algorithms, especially where these algorithms do not yield acceptably fast implementations. It should be noted that, although scalability is an overriding concern, we do not ignore the need to produce a reasonably precise result as output. In regard to this, we fix the minimum acceptable precision as equivalent to the points-to formulation presented in our running example (Section 1.1.2).

For example, Chaudhuri's CFL-R algorithm is the fastest known solver for points-to problems with at least this level of precision (though faster algorithms exist for imprecise/restricted formulations). Unfortunately, the subcubic algorithm specifies that edge-relations be represented densely (so that set operations may be performed in aggregate). There are ≈ 1.4 million vertices in the OpenJDK graph, meaning that every input relation would be $\approx$ 230GB. Clearly, then, the space-complexity of the subcubic algorithm renders it unusable in our context.

3.2.1 Opportunities in Points-To

The most important modification to the evaluation strategy which underpins Gigascale's performance is the change that it makes to how the analysis progresses, which we will call the **driver**. Consider the influence that the worklist has on runtime in the Melski-Reps algorithm (Algorithm 2). The stated complexity of this algorithm is $\mathcal{O}\left(n^3\right)$, but assume instead that the maximum size of the output was parameterised with w, in which case the algorithm's para-

metric complexity is $\mathcal{O}(wn)$. Technically, since every worklist member must be dequeued and checked against the rest of the graph, that complexity is closer to a lower bound. The specifics of the OpenJDK problem, and the reason that the analysis is called "Giga"-scale, is that there are ≈ 1.5 billion points-to pairs in the result. Since the runtime of the worklist algorithm scales linearly with the output size, it is unusably slow when executed on this reasonably dense problem. In contrast, the Gigascale analysis is driven by a *refinement* technique. Instead of incrementally expanding on the result-set by applying points-to rules to newly discovered edges (as per the summarisation approach), Gigascale initially over-approximates a crucial component of those new relations, called the **bridge** set, then iteratively validates bridges. Given that, in practice, the size of the bridge set is relatively small (≈ 64 thousand bridges), this yields significant performance improvements.

Bridges are an informal notion that captures the indirect assignment of variables via field loads/stores, and hence the propagation of points-to sets. In the presentation in Section 1.1.2, the points-to language had an indirect assignment rule given as:

$$\textit{points-to} \rightarrow \textit{load}_f\ \textit{points-to}\ \overline{\textit{points-to}}\ \textit{store}_f\ \textit{points-to}$$

It is the need to match fields, as captured in this rule, that causes the points-to analysis to be a CFL-R problem. We can use simple linguistic transformations to separate field matching into a separate rule:

$$\textit{bridge} \rightarrow \textit{load}_f\ \textit{points-to}\ \overline{\textit{points-to}}\ \textit{store}_f \quad \textit{points-to} \rightarrow \textit{bridge}\ \textit{points-to}$$

Without the bridge rule, the rest of the points-to language is actually regular, and can be specified by the rule $\textit{points-to} \rightarrow (\textit{assign}|\textit{bridge})^*\,\textit{alloc}$. It is relatively easy to solve the bridge-free problem, and we can avoid the complex gadget proposed by Yannakakis [86] for regular-language reachability. In fact, we need only calculate:

$$\textit{points-to} = \mathrm{TC}(\textit{bridge} \cup \textit{assign}) \bowtie \textit{alloc}$$

Though the bridge-free component can be solved via normal transitive closure, we now require a means of calculating the bridges. Instead of incrementally building a valid bridge set, in the style of the Melski-Reps algorithm, instead assume that we can over-approximate the set with an oracle O_{bridge}. It is always possible to generate such an oracle, since we can simply use the trivial **all-pairs** approximation:

$$O_{bridge} = \{(u, v)_{(p,q)} \mid \exists f, g : \textit{load}_f(u, p), \textit{store}_g(q, v) \in E\}$$

Choosing a good oracle is a matter relevant to fine tuning the algorithm's runtime, and is discussed further in Section 3.2.2.1. At this stage, we simply point out that, given an oracle, it is possible to construct a different evaluation strategy. Given an initially empty set of *true* bridges, it is possible to confirm, from the points-to relations in the graph, that some of O_{bridge} is indeed valid, while some remains unconfirmed. This works iteratively, where the bridges of set i

Algorithm 3 Gigascale points-to analysis [27]

```
1: function GIGASCALE(alloc, assign, load, store)
2:     O_bridge ← GENERATE_ORACLES(load, store)
3:     t ← assign
4:     while ∞ do
5:         p ← TC(t) ∘ alloc
6:         v ← {(u, v) | (u, v)_(b_u,b_v) ∈ O_bridge, p(b_u) ∩ p(b_v) ≠ ∅}
7:         if v = ∅ then return p
8:         else
9:             t ← t ∪ v
10:            O_bridge ← O_bridge \ v
```

can be confirmed by any of the bridges in sets $j < i$. Given a bridge between u and v, which is noted to depend on p and q, then:

$$(u,v)_{(p,q)} \in bridge_i \Leftrightarrow \exists h : (p,h),(q,h) \in \mathrm{TC}(assign \cup \bigcup_{j<i} bridge_j) \bowtie alloc$$

The above strategy has some interesting properties which we shall note here. As an algorithm, it must iteratively attempt to validate O_{bridge} relations, and add them to a running transitive closure. Running this until a fixpoint is reached, we perform a transitive closure each iteration, and we may only validate one bridge per iteration, where there are at most n^2 bridges. The time complexity is therefore $\mathcal{O}\left(n^2\ TC(n)\right)$ [27], so algorithmically it seems worse than the Melski-Reps approach. Indeed, one can construct pathological cases where this is in fact the runtime, but in practice it may be far better. The actual number of iterations is determined by the *depth* of the field-access stack: fields used in indirect assignment which alias *via direct assignments only* will be validated in the first iteration, and doubly-indirected bridges will be validated in the second iteration. The intuition is that programmers can not over-burden themselves by conceptualising arbitrarily deep field accesses, so the actual indirection depth will be quite small. On a benchmark study, the depth is a small constant around 5, with the deepest codebase having 9 levels of indirection [27]. So, since the number of bridges is very small, and they will be validated in an effectively constant number of iterations, the runtime of Gigascale-style evaluation will actually outperform the worklist approach on realistic problems.

3.2.2 Implementation Optimisations

Implementing the Gigascale analysis requires much effort in understanding and tweaking the optimisations. Given that the analysis is designed for solving the OpenJDK, we are not at this stage concerned about whether these techniques apply generally.

Pseudocode for the Gigascale analysis is reproduced in Algorithm 3. We leave GENERATE_ORACLES and TC as black-boxes for the time being. As we

shall see, there are interesting tradeoffs that occur when choosing different bridge oracle generation techniques. The verification of bridges occurs on line 6, on the assumption that the bridge between u and v was created by a load-store pair with base variables b_u and b_v respectively. Clearly if $p(b_u)$ and $p(b_v)$ have a non-empty intersection, then there exists some h for which $(b_u, h), (b_v, h) \in$ *points-to*, in which case $\langle load(u, b_u), points\text{-}to(b_u, h), \overline{points\text{-}to}(h, b_v), store(b_v, v)\rangle$ is a path in the graph, implying an indirect assignment between u and v, thus the Gigascale analysis computes the same points-to relation as the running example of Section 1.1.2.

3.2.2.1 Oracle Generation

This section refers to the technique used to over-approximate the bridges, required in Algorithm 3 line 2. There are two easy optimisations which improve the practical runtime of certification without changing its theoretical complexity. In the *certifier* direction, a given pair of base variables (whose non-empty points-to intersection implies that a bridge is valid) can actually be used to validate several bridges at once. In fact, this use case occurs in practice regularly. Firstly, via a multiplicity of fields, `m.f = a; m.g = b; ... x = n.f; y = n.g;`, then if m aliases n this would certify both the f-field bridge and the g-field bridge. Secondly, via a multiplicity of uses, `m.f = a; ...x = n.f; y = n.f;`, so if m aliases n this would certify both the (x, a) and (y, a) bridges. In the *certified* direction, there may be several base pairs which could validate a given bridge, but if any one of them succeeds the others can be ignored in future iterations (since they would only confirm a bridge already known to exist). Again this occurs in practice, particularly where there are several control-flow paths between the field store/load. To capitalise on these redundant cases, the Gigascale implementation uses high-performance map and set implementations to track which base pairs validate which bridges, and also which validations obviate which base pairs.

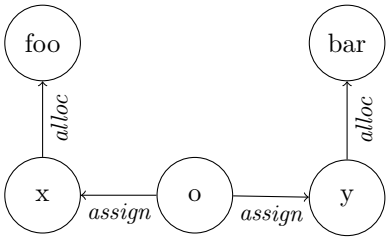
The naïve all-pairs oracle is only intended to demonstrate that it is always possible to generate an over-approximation. From Algorithm 3, clearly the validation step on line 6 becomes more complex with a coarser over-approximation. Thus it would seem desirable to compute a precise oracle set. Alternatively, the certification step can be made very fast (particularly when using the compressed sparse bit-sets discussed in Section 3.2.2.2), so it might be better to save computation in GENERATE_ORACLES. On one end of the spectrum we have the very fast but less precise **same-field** oracle, used by Sridharan and Bodík [68]:

$$O_{bridge} = \{(u, v)_{(p,q)} \mid \exists f : load_f(u, p), store_f(q, v) \in E\}$$

This oracle is simply a restriction of the all-pairs approximation to bridges over the same field. Field sensitivity can be enabled or disabled by swapping between the all-pairs and the same-field oracles in the Gigascale analysis. Note, though, since its runtime is largely determined by the size of the oracle set, field-sensitive points-to is actually *faster* to compute using the Gigascale analysis.

```
1  Foo x = new Foo();
2  Object o = x;
3  Bar y = new Bar();
4  o = y;
```

(a) Source code



(b) CFL-R graph

Figure 3.1: Case showing how the aliasing relationship is not transitive.

On the OpenJDK, the number of all-pairs oracles is $\approx 7.4 \times 10^9$, as compared to $\approx 5.3 \times 10^5$ using the same-field oracle.

Computing all pairs of variables which load and store the same field is a significant improvement in precision, but techniques with even higher precision may be more viable. Zhang et al. present a technique for computing the bi-directional Dyck reachability problem for points-to [89]. Their points-to language over-approximates the native CFL-R formulation in two ways. Firstly it treats assignment statements as bi-directional, in the sense that `a = b` forces the variable classes for a and b to equivocate. Secondly, it treats variable aliases as transitive, which is not the case. Consider the example in Figure 3.1. Using the points-to formulation from Section 1.1.2 we have that x and o point to foo, and y and o point to bar, meaning that o aliases x and y. Despite this, x does not alias y, as those two variables do not point to a common heap object. In the Zhang et al. formulation, variables are stored in a disjoint-set data structure, which causes a Steensgaard style transitive propagation of points-to information [71]. The x and o groups are unioned, then the o and y groups, meaning that y and x finish in the same class (which the analysis interprets as an alias). Hence, bi-directional Dyck-reachability is a fast means of over-approximating the alias sets, and is therefore useful as a bridge oracle. Let d store the result of the Zhang et al. analysis, such that $d(v)$ is the class of aliasing variables associated with v. Then the GENERATE_ORACLES returns the following set, called the **bi-Dyck** oracle:

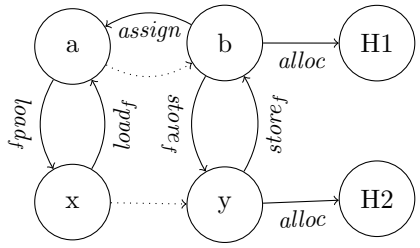
$$\{(u, v)_{(p,q)} \mid \exists f : load_f(u, p), store_f(q, v) \in E, d(p) = d(q)\}$$

Bi-Dyck is a restriction of the same-field oracle to those pairs which also have aliasing base variables in the Zhang et al. analysis. On the OpenJDK, bi-Dyck exhibits significantly more precision than the same-field approximation, having only ≈ 84 thousand bridges, less than $\frac{1}{6}$th of the same-field pairs, and very close to the true number of ≈ 64 thousand valid bridges.

Whilst the generation of oracles is an important factor in Gigascale's overall runtime, their validation is also important. The technique used in the analysis is comparable to a bottom-up approach, since it begins assuming none of the

```
1  Object a, x;
2  Object b = new H1();
3  Object y = new H2();
4  y.f = b;
5  b.f = y;
6  x = a.f;
7  a = x.f;
8  b = a;
```

(a) Source code



(b) CFL-R graph

Figure 3.2: Case showing self-supporting bridges. Dotted edges show bridges inferred by the bi-Dyck stage. The (a, b) and (x, y) bridges mutually support each other.

bridges are valid, and iteratively tries to validate more of them. An alternative approach is to iteratively shrink the bridge oracles in a top-down approach. In the top-down case, a bridge which can not be confirmed to be valid when assuming the validity of all the bridges will be evicted from the set. The top-down version of Gigascale is presented here:

1: **function** GIGASCALE_TOPDOWN($alloc$, $assign$, $load$, $store$)
2: $O_{bridge} \leftarrow$ GENERATE_ORACLES($load$, $store$)
3: **while** ∞ **do**
4: $p \leftarrow \mathrm{TC}(assign \cup O_{bridge}) \circ alloc$
5: $o \leftarrow \{(u, v)_{(b_u, b_v)} \mid (u, v)_{(b_u, b_v)} \in O_{bridge}, p(b_u) \cap p(b_v) = \emptyset\}$
6: **if** $o = O_{bridge}$ **then return** p
7: **else**
8: $O_{bridge} \leftarrow o$

The top-down approach has some advantages in comparison with the chosen bottom-up technique. Each iteration of the analysis only removes edges, allowing us to use knowledge about dominance and connectivity to maintain a faster incremental transitive-closure structure, whilst the bottom-up approach gains little benefit by not simply re-computing the entire relation each time. The p variable is *always* an over-approximation of the points-to set, so we are able to terminate the analysis early (possibly in a time-critical context) and be confident that no points-to sets will be missed. Unfortunately, the top-down approach does not yield a fully precise answer, as some bridges may be **self-supporting**, despite the fact that they would never be validated in the bottom-up approach. A self-supporting bridge is such a pair that depends (possibly indirectly) on itself in order to pass validation. Figure 3.2 shows some bridges which can not be invalidated by the top-down strategy. The same-field oracle would create the (a, b) and (x, y) bridge edges, since they are load/store bases for the same field f. Also, the bi-dyck oracle makes the same bridges since a and b are in the same aliasing class (due to the assignment `b = a`), and therefore x and y

are in the same class (as bases of the a and b load/stores). After which we can not invalidate the bridges, as the analysis assumes x and y both point to $H2$, since their base variables a and b both point to $H1$, since their base variables x and y both point to $H2$, etc. On most benchmarks, the imprecision caused by self-supporting bridges was usually small, either they did not occur, or they led to over-approximation by $\approx 0.01\%$ compared with bottom-up. Notably though, for the HSQLDB benchmark of the DaCapo suite [13], precision was significantly worse ($\approx 44\%$), as the self-supporting bridges occurred in critical and frequently used sections of the code. Ultimately, bottom-up evaluation was preferred as we could not make guarantees about the likelihood of running into these adverse cases with well-connected self-supporting bridges.

3.2.2.2 Data Structures

The size of the OpenJDK is a key factor in many of the design choices for the Gigascale analysis. The characteristics of the points-to solution are very different for this large library, when compared to typical benchmark applications. In an evaluation of the DaCapo benchmarks we determined that, for the simple points-to problem, program variables on-average pointed to between 0.5 and 2.9 abstract heap locations. Further, the average points-to set sizes did not correlate with the size of the input program, as the larger benchmarks had on average 1.6 heap objects per variable, whereas the relatively small H2 case had 2.03. In comparison, the OpenJDK has an average of ≈ 969 heap locations per variable [27], three orders of magnitude more than the smaller benchmarks. The difference is understandable, and is partly due to the compounding effects of imprecision on this large benchmark, but also, being Java's runtime library, the OpenJDK is more interconnected than a typical application, and deals with simpler classes (like Object) than those defined in application code.

Ultimately, the foremost concern, when evaluating large problems at the scale of the OpenJDK, is data movement. Shipping large volumes of data to and from main memory in an unstructured manner causes significant delays, potentially rendering the analysis unusable. The most important factor is ideal cache utilisation, which can be achieved in several ways for the points-to analysis. In Algorithm 3, variables are associated with their points-to information, which need to be frequently examined both for calculating the partial points-to data (line 5) and for certifying bridges (line 6). To maximise cache utilisation, we need to make sure that (1) the points-to set for a variable is kept in a cache-friendly way (i.e. that it does not exceed a cache line), and (2) that the order in which variables are accessed does not excessively miss the cache or evict soon-to-be-used blocks.

One idea for improving the cache friendliness of the points-to sets is to use space-efficient data structures, like **bitsets**. The bitset data structure is designed for representing lists of boolean data. In connection with points-to analysis the boolean data is a true/false flag for each heap object if the given variable points-to that object. From Figure 3.1, given an integer mapping for the heap objects foo and bar based on their order of appearance in the code,

the bitset associated with x is $\langle 10 \rangle$, whereas o has $\langle 11 \rangle$. An important practical characteristic of these bitsets is that they are sparse in practice. The average points-to bitset has between 0.5 and 2.9 set-bits in the DaCapo benchmarks, while there are 3000 to 70000 heap objects in those benchmarks. Even the OpenJDK, despite being much denser than DaCapo applications, is still sparse, having $\approx 970/347000$ set-bits per variable. It is this property of the points-to sets, which Van Schaik and De Moor observed more generally in transitive closure problems [78], that prompted the use of bitsets which were both *sparse* and *compressed*.

Compressed Sparse Bitsets (CSBs) are data structures which provide the functionality of a bitset in a more memory efficient form. Firstly, instead of a dense representation, large chains of unset bits are ignored. The set is divided into *blocks*, each of which stores a fixed number of set-bits, but only blocks with at least one set-bit are retained, noting their offset from the whole set, while zero-blocks are not tracked at all. Further, since individual blocks have low entropy (i.e. they typically only set one or two bits), we can further compress them using simple run-length encoding techniques or similar [78]. CSBs are mostly useful where memory bottlenecks are the greatest limitation to the program's runtime, as they impose additional computation over a normal bitset in terms of accessing set bits and decompressing the blocks. The result is a concise data structure, which can be used to track points-to information with significantly improved cache utilisation.

3.2.2.3 Transitive Closure

Together with validating bridge edges, a significant portion of Gigascale's runtime is spent performing TC iteratively. Whilst there are TC algorithms with a time-complexity that is equivalent to boolean matrix multiplication [31], i.e. $\mathcal{O}\left(n^{2.3}\right)$, in practice these algorithms do not perform well, due to their significant constant overheads. To compute TC efficiently, Gigascale favours the more practical approach of **successor-set merging** developed by Nuutila [52]. The practical advantages of Nuutila's approach were discussed in Section 2.1.2, and the technique shown in Algorithm 1.

In the points-to analysis context, we can make additional improvements to Nuutila's algorithm due to the problem's restrictions. Gigascale's running points-to set is given to be $\mathrm{TC}(t) \circ \mathit{alloc}$ on line 5. In other words, the transitive closure of the carrier relation t is computed *in full*, then this is composed (via relational join) to the *alloc* edges. Firstly, since *alloc* relates vertices to heap objects, we can guarantee that the heap objects are all leaves, implying that $TC(t \cup \mathit{alloc})$ will compute the same result. Secondly, only some of the variables will have their points-to sets queried in each iteration of Gigascale. Specifically, until we return the complete points-to set, only those variables that appear as the base variables of the bridges need to be queried for aliasing. As such, a demand-driven approach, incorporating lazy evaluation, is desirable. Note that this refers only to returning points-to queries on demand, whereas computation of the transitive closure it still performed in advance. Thirdly, the points-to

sets themselves should be propagated, rather than the successor sets, as we only care what heap objects a given variable points to, and not which variables were directly/indirectly assigned to it in order for that to be the case. And finally, we can capitalise on some equivalence properties of points-to graphs [33], allowing us to compress the graph in a result-preserving way before performing transitive closure. Combining these optimisations yields a fast algorithm for computing points-to sets. Gigascale's implementation features successor sets of heap objects only (i.e. we do not record intermediate variables) stored in compressed sets. Further, successor sets are only propagated in a demand driven way, hence lazily, in the event that a variables is queried. Since only a very small subset of program variables actually form base pairs of the bridge oracle, this results in significantly fewer set propagations.

An important consideration when applying our specific TC approach to Gigascale's points-to analysis came from the interplay between the successor-set merging idea, and the CSBs. The memory footprint of each variable's successor set is largely determined by the *number of blocks* it uses, rather than the number of set-bits. Without some effort, we would expect the set bits to distributed uniformly at random over the range of heap objects. Further, as each variable is associated with such a set, those sets must all be kept in an array in memory, meaning that merging or intersecting two sets would on average require looking up two random entries in the array of variables. Both of these factors adversely impact cache utilisation. Instead, suppose that we numbered variables and heap objects topologically, i.e. they are represented by an integer from a depth-first pre-order. Now the points-to set for a given variable is very likely to bunch together over a small part of the range of heap objects, since all those heap objects are likely to receive numbers after the depth-first traversal of the variable's node. Further, since the integer representation of the variables are related to their successors/predecessors, merge operations between them in the propagation will frequently use nearby or adjacent blocks in the array of points-to sets. The result of which is that as CSB representation of the successor set will be both very local (by virtue of the numbering) and small (by virtue of the co-location of adjacent blocks, which implies that the adjacency data has low entropy and compresses easily). Thus, by pre-computing the mapping between variables/heap objects to integers in a depth-first traversal, we significantly improve the cache utilisation of the Gigascale analysis.

3.3 Effectiveness

We implement the Gigascale points-to analysis as a Java program, for verifying its effectiveness on very large problem cases. The analysis is made publicly available[1] for download and testing purposes.

[1] via Bitbucket `https://bitbucket.org/jensdietrich/gigascale-pointsto-oopsla2015`

Table 3.1: Breakdown of the Benchmarks used in our evaluation, showing the sizes of the vertex and edge sets, as well as statistics on the result sets. This is a reproduction of a dataset published by Dietrich et al. [27]

Bench	$\|V\|$	$\|E\|$	$\|points\text{-}to\|$	Avg.	Max
sunflow	15,464	15,957	16,354	1.06	140
lusearch	15,774	14,994	9,242	0.59	35
luindex	18,532	17,375	9,677	0.52	35
avrora	24,690	25,196	21,532	0.87	342
eclipse	41,383	40,200	21,830	0.53	88
h2	44,717	56,683	92,038	2.06	457
pmd	54,444	59,329	60,518	1.11	221
xalan	58,476	62,758	52,382	0.90	221
batik	60,175	63,089	45,968	0.76	681
fop	86,183	83,016	76,615	0.89	285
tomcat	111,327	110,884	82,424	0.74	325
jython	191,895	260,034	561,720	2.93	1,878
tradebeans	439,693	466,969	696,316	1.58	581
tradesoap	440,680	468,263	698,567	1.59	581
openjdk	1,621,634	1,964,146	1,570,820,597	968.67	82,665

3.3.1 Experimental Performance

We verify the practical advantages of Gigascale-style analysis on real-world benchmarks drawn from the DaCapo suite [13], and the OpenJDK. The DaCapo programs are a collection of real-world Java codebases which represent the typical behaviour of Java, and are useful for cross-comparing results and insights gained in other research fields. Specifically, we present results related to the "bach" version of DaCapo (aka the 2009 version), though previous versions of Gigascale were tested on the 2006 version. The OpenJDK is a publicly available open-source implementation of Java's runtime support library. Unfortunately, due to the size of the problem, specialised (and proprietary) tools were used to convert the codebase into a CFL-R problem, amenable for solving by Gigascale.

A breakdown of the experimental benchmarks, together with some of the statistics about their solution set, is presented in Table 3.1. Entries in this table are ordered by the size of the vertex set (i.e. n in their order-theoretic notation). The datasets have some interesting properties, specifically related to differences between the sizes of input and output edges. For all benchmarks, the number of edges (i.e. Java statements) roughly correlates with the number of variables. The largest difference is found in the JYTHON benchmark, with 36% more edges. This verifies our assumption that points-to graphs are typically sparse (and hence that sparse bitsets will be useful for representing them). The size of the points-to sets correlates less with the sizes of the vertex or edge sets. The ECLIPSE and LUINDEX cases have significantly fewer points-to results than input

Table 3.2: Oracle generation, precision and certification statistics. The number of same-field (SF) and bi-Dyck (BD) oracles generated, as compared to the true size of the bridge set. The precision of the top-down refinement approach both initially (Init) and when no more bridge edges can be invalidated (Fin). The number of iterations needed in top down (TD) and bottom up (BU) refinement strategies. The OpenJDK problem is too large to calculate its precision. This is a reproduction of a dataset published by Dietrich et al. [27].

Benchmark	SF	BD	$\lvert bridge \rvert$	Init	Fin	TD	BU
sunflow	3,934	886	747	0.9857	1.0000	2	4
lusearch	5,037	793	770	0.9909	1.0000	3	4
luindex	8,722	1,228	1,212	0.9978	0.9998	2	5
avrora	4,630	756	722	0.9949	0.9953	2	6
eclipse	6,444	1,430	1,096	0.9460	0.9606	4	5
h2	12,174	3,492	3,281	0.8844	1.0000	3	5
pmd	65,849	1,908	1,752	0.9736	1.0000	3	6
xalan	31,041	2,233	2,104	0.9755	0.9998	2	6
batik	19,405	2,397	1,948	0.9857	1.0000	3	4
fop	12,551	1,962	1,744	0.9908	1.0000	4	8
tomcat	30,863	8,052	7,700	0.9830	1.0000	4	5
jython	55,776	8,646	8,082	0.8155	0.9997	3	7
tradebeans	149,642	30,863	29,155	0.9661	0.9988	4	6
tradesoap	149,724	30,894	29,173	0.9662	0.9988	4	6
openjdk	538,274	84,415	64,716	*	*	5	6

edges, while H2 has substantially more. Importantly, the OpenJDK is itself an outlier here, having several orders of magnitude more points-to edges than input edges. Table 3.1 also shows the average and maximal points-to set sizes per variable. Interestingly, the averages and maximums show little relationship with some intuitive metric like $\frac{|points\text{-}to|}{|V|}$; BATIK has an unusually low average (for its size), despite that its most connected variable has an abnormally large points-to set. Note that not every CFL-R vertex is a variable, as some represent heap objects.

Turning to the issues surrounding the generation and validation of bridge oracles, we have tabulated the important statistics in Table 3.2. The number of true bridges (i.e. indirect assignments via field loads/stores) is recorded in the column $|bridge|$, which we can contrast with the same-field and bi-Dyck oracles. Sridharan and Bodík's oracle [68] is fast to compute, but over-approximates the true number of bridges by a factor of 5× to 10× (calling PMD an outlier, with 37×). The alternative oracle, computed with the technique by Zhang et al. [89], is much more accurate, where the worst approximation is by about 1.3× in ECLIPSE. This table also compares the Gigascale bottom-up evaluation strategy with the top-down version discussed in Section 3.2.2.1. Init and Fin measure the *precision* of the results of using the top-down evaluation, which can be imprecise due to self-supporting bridges. Precision is calculated here as

the size of the points-to result (according to the formulation from Section 1.1.2) as a fraction of the result assuming *all* bi-Dyck oracles are true. Unfortunately, the OpenJDK benchmark is so large that a "correct" points-to set could not be calculated, hence we omit precision for this case. We see that the top-down strategy invalidates most spurious oracles. Half of the DaCapo benchmarks completed with equal precision to the "true" result, and of the half that did not, the worst case was only wrong 0.5% of the time. Further, the number of iterations that Gigascale needed to invalidate the incorrect oracles (in a top-down approach) is typically much smaller than the number needed to validate the correct ones (in a bottom up approach). The number of iterations significantly affects the runtime of Gigascale, i.e. it makes up 40%-70% of execution time. If not for the unpredictable imprecision, which we have only observed in the HSQLDB benchmark from DaCapo 2006, top-down refinement would be the superior approach.

We verify our claims about the runtime performance of the Gigascale (GS) approach by comparing it to alternative implementations. The baseline comparison is an implementation of the Melski-Reps worklist algorithm (WL) as presented in Algorithm 2. Our implementation of WL is tailored specifically to the points-to grammar used by this analysis, so it avoids some of the overheads associated with dynamically selecting and evaluating the rules (i.e. lines 8 and 10). The Difference Propagation algorithm (DP) [69], is used to make a comparison against effective techniques in the research literature. DP uses a similar working-set based approach to WL, but instead propagates whole sets in an intelligent way, which improves the cache locality. The high performance LogicBlox (LB) system [5], which has seen recent use as a program analysis engine, is the nearest comparison available for high performance analyses. The LB solver requires a custom logic script to emulate the behaviour of the Gigascale analysis, which we have optimised manually. Though the DOOP framework has a field-sensitive analysis [16], which is similar to Gigascale's, their logic is more complicated and incompatible, so we could not fairly use it for comparison.

All implementations were run on an Intel® i7-4790 processor, with 32GB RAM, under Lubuntu 14.10. The DaCapo benchmarks come from the "bach" version (i.e. DaCapo 2009), and the 1.7.0 b147 version of the OpenJDK was used for the very large benchmark. We use a combination of the Soot tool [77], version 2.5.0, and the DOOP framework [16], version r-160113, to convert the Java programs into `.csv` files, which Gigascale uses. The very large OpenJDK library was converted to its `.csv` dataset using proprietary tools, which are not freely available. A timeout of two hours was applied to each experimental run.

The runtime of the various implementations on each benchmark is presented in Figure 3.3. The WL and DP implementations exhibit a roughly quadratic growth rate according to problem size. Whilst both approaches are expected to run in roughly cubic time, this only occurs for pathological inputs, whereas these benchmarks are drawn from realistic problems. Neither of these approaches were designed for the extremely large OpenJDK dataset, so they unsurprisingly timed out for this case. The LB implementation performs well for the DaCapo benchmark, but is unsuited for handling very large problems. Firstly, LB has some

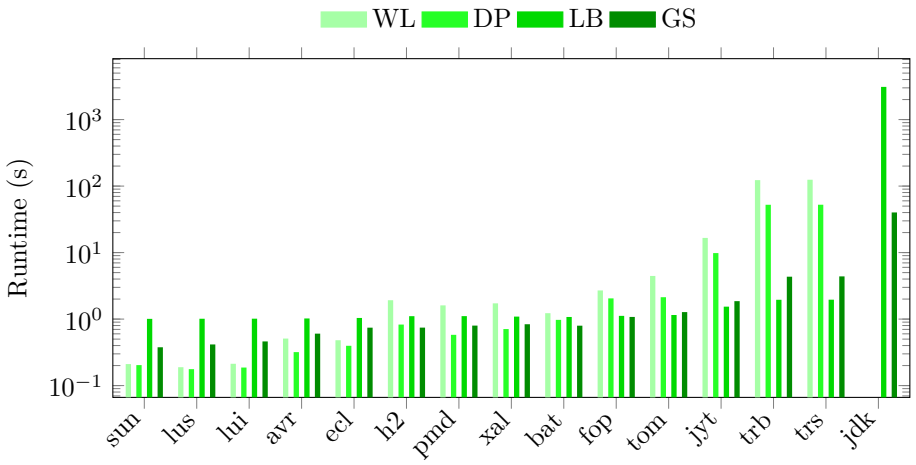


Figure 3.3: Runtimes for the competing implementations of the points-to analysis. All analyses compute the same points-to set (using different strategies), their correctness is confirmed offline by checking their outputs against each other. This is a reproduction of a dataset published by Dietrich et al. [27].

large constant overheads (it is designed more like an SQL server, using interprocess communication), which cause it to under perform on smaller benchmarks, but these overheads are amortised quickly as the problem sizes grow. LB also performs well on the larger DaCapo problems, as its internal datastructures are very efficient. The larger DaCapo problems are still much sparser than Gigascale is designed for, hence LB actually outperforms it. Importantly, only Gigascale copes well with the very large OpenJDK dataset, outperforming LB by almost two orders of magnitude. We attribute much of Gigascale's superior performance to the use of cache-aware data structures and an intelligent evaluation mechanism. LogicBlox uses a bottom-up approach common in Datalog, meaning that, like WL and DP, its runtime is influenced by the size of the *output*. Gigascale's evaluation depends mostly on the number of iterations needed to validate bridges, as well as the size of the transitive closure, so it avoids processing the extremely large output dataset.

Figure 3.4 shows the memory requirements for the competing points-to analysis implementations. We see similar growth in memory amongst the WL, DP and GS implementations on the DaCapo benchmarks. The former two implementations do not have any specific memory advantages, and simply represent points-to sets and input relations in flat memory structures, whereas Gigascale's compression features do not exhibit much advantage on these less dense benchmarks. LB has some constant overheads (similar to the overheads associated with runtimes), which mean it performs poorly on smaller benchmarks but then quite well on the larger DaCapo problems. The advantages of Gigascale are seen on the OpenJDK problem, where the compression techniques significantly re-

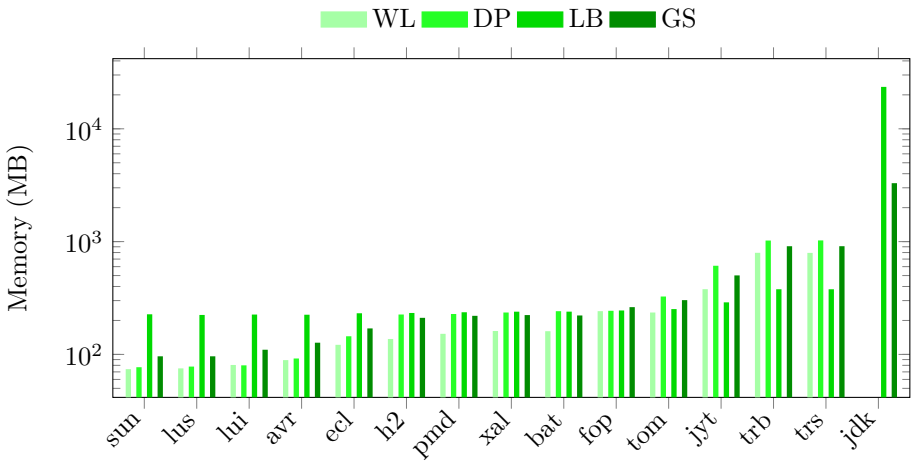


Figure 3.4: Memory usage for the competing implementations of the points-to analysis. This is a reproduction of a dataset published by Dietrich et al. [27].

duce its memory requirements as compared with LB. To our knowledge, LB does not use in-memory compression techniques, but only compresses when writing to disk.

3.3.2 Parametric Analysis

In order to account for the superior performance of Gigascale, we formulate a parametric complexity analysis. In essence, the complexity analysis presented in Section 3.2.1 is very coarse, and does not illustrate how *realistic* practical assumptions that we make about the input graphs can significantly influence performance. We compare how these parameters contribute to the runtime of the worklist algorithm as a means of pointing out the difference.

Let the input problem be characterised as having v input variables and h heap objects. There are still e edges in the graph (i.e. statements), but let the number of valid bridge edges (i.e. matching load-store pairs with aliasing base variables) be b. Assume also that the *depth* of the bridges, is no more than d, i.e. that a valid bridge will transitively depend on at most d other bridges. Finally, assume that each variable points to p heap objects.

Looking at Algorithm 2, we see that from an initial size of h, the worklist will always grow to contain $\max(vp, e)$ items, and each one must be dequeued to find edges which match the rule. Extending a pt-edge by one vertex involves looking up $\max(v, e)$ potential endpoints of an assignment, or b endpoints of a bridge. In total the Melski-Reps algorithm, as characterised by these parameters, should run in $\mathcal{O}\left(\max(vp, e)(\max(v, e) + b)\right)$ time.

Looking at Gigascale, we gain a new understanding of the major components of its runtime. The bi-Dyck reachability component is known to run

in $\mathcal{O}(n + e \log e)$ time [89], but this is needed only to determine a bridge approximation in advance. Subsequently, Gigascale iterates d times, where each iteration requires a modified TC procedure and a validation step. The modified TC first requires topological ordering, which is done in at most $e + b$ steps, and then the points-to set propagation. In a naïve implementation, propagation would take vp steps to propagate a p-sized set to v vertices, but we *lazily* propagate to the oracle's base variables as-needed, hence merely bp steps are used. The oracle-validation step uses bp operations, checking the points to set of each bridge. Thus the total time complexity is actually:

$$\mathcal{O}(e \log e + d(bp + bp + b + e)) = \mathcal{O}(e \log e + d(bp + e))$$

Note that the parametric complexities are even coarser than the stated time complexities, given pessimistic assumptions about how they relate to n. Firstly, v and p are both $\mathcal{O}(n)$, the former by definition and the latter by the argument that the number of distinct heap objects that a variable points-to can not exceed the number of vertices in the graph problem. On the other hand e, d and b are all $\mathcal{O}(n^2)$, e and b may contain every pair of vertices, and the bridge pairs can be adversely constructed such that each one depends on exactly one other bridge, hence the height d would be equal to b. Then the parametric complexity of the Melski-Reps algorithm is equivalent to $\mathcal{O}(n^4)$, which we know to be worse than the actual worst-case bounds. Gigascale's is still $\mathcal{O}(n^5)$, though this may be a coincidence. The intuitive explanation for this is that truly pessimistic assumptions about all the parameters can not coincide, for example $p \leq h$, but $n = v + h$, so $vp < n^2$.

Importantly, though, the parametric analysis gives us insight into why Gigascale's evaluation style is effective on extremely large problems. One major component of the Melski-Reps runtime is vp, the size of the points-to set. Every points-to edge must be added to the worklist, and thus be processed at some point by the algorithm. On the other hand, Gigascale has no vp term, and instead verifies bridges themselves in bp steps. In practice, the number of bridges b is significantly smaller than the number of vertices: in the OpenJDK, 85 thousand as compared to 1.4 million. The p factor here comes from the size of the points-to sets, either when verifying bridges (i.e. intersecting two points-to sets) or propagating the sets in the transitive closure (i.e. unioning them). But Gigascale uses an intelligent compression technique, and numbers heap objects depth-first, which makes them more likely to be co-located, so the literal blocks in the CSB are dense, rendering p much smaller in practice. Combining these factors, the bp term, whilst it could be on $\mathcal{O}(n^3)$, in practice is closer to $\mathcal{O}(n)$, and it exhibits linear behaviour. Further, Gigascale performs exactly d iterations (i.e. the number needed to verify all the bridges), which we have found to be a small constant < 10 in all benchmarks. Interestingly, the depth is not strongly correlated with the problem size, the greatest depth is for `fop`'s 8, a middle-sized problem, whilst the very large OpenJDK is equal with the very small `avrora` problem, having 6. These observations about the relatively small sizes of b p and d marry with the fact that e is actually close to $|V|$ (i.e.

n, see Table 3.1), meaning that the bi-Dyck oracle is only slightly worse than linear time. We therefore expect that, for extremely large *practical* problems, Gigascale scales better than a quadratic algorithm.

3.3.3 Generalising

The Gigascale case study is intended to demonstrate and explore the kinds of techniques/approaches which are used in practice to develop performant analyses. Whilst the Gigascale analysis is very efficient, compared with state of the art approaches, significant manual effort was needed simply to develop a *single* analysis, predominantly for a *single* use case. Ideally, we desire general techniques, which we can apply automatically to future problems. We examine the important performance considerations here and develop techniques for generalising them.

The most fundamental change to the Gigascale analysis was the observation that bridge edges could be over-approximated in an initial phase, and subsequently verified. This observation came from examining the language rule for indirect assignment:

$$\textit{points-to} \rightarrow \textit{load}_f\ \textit{points-to}\ \overline{\textit{points-to}}\ \textit{store}_f\ \textit{points-to}$$

This rule is amenable to being over-approximated by the bi-Dyck oracle as it is mostly a Dyck rule, where $\textit{load}_f$ matches with $\textit{store}_f$ and *points-to* matches $\overline{\textit{points-to}}$. It is difficult to contrive a general mechanical procedure for identifying Dyck-subrules. Such a procedure would have to reason about (1) what the matching pairs of relations are, either by verifying that they always appear in mirrored pairs in the rules, or by using heuristics, and (2) how to formulate the bi-Dyck problem based on the matching pairs and the *other* rules in the problem.

On the other hand, instead of looking for bi-Dyck oracles, it may be more feasible to find same-field oracles automatically. A same-field oracle exists whenever a pair of fields appears in a mirror around some (possibly empty) middle string. In the case of the indirect assignment rule, the mirrored pairs are $\textit{load}_f$ and $\textit{store}_f$, with *points-to* $\overline{\textit{points-to}}$ as the middle string. Depending on the size of the input grammar, it may be feasible to simply enumerate all the potential pairings and middle-strings in a brute-force manner, though deciding which combination to develop an oracle for may be impractical, especially if it is difficult to judge how effective the oracle will be.

Another important factor in the points-to analysis is that, on the assumption that bridges can be approximated, the rest of the language becomes regular (as mentioned in Section 3.2.1). It is of general research interest to automatically discover the *regular subset* of a context-free language. Coarse approximations of the context-free subset of the language can be made by applying the pumping lemma [38], observing that recursive derivations may lead to strings with mutually dependant substrings (and hence, necessitate memory for their recognition). Of the three rules needed for the simple points-to analysis from Section 1.1.2,

only the indirect assignment rule is recursive, hence only one rule actually demands the full power of CFL-R, while the other rules can be solved with a simpler formalism like regular-language reachability (or even TC, as was the case for Gigascale). In general, though, we can not guarantee that a language will be presented in a way that is amenable for recognising context-free subproblems, though this may be an interesting avenue of future work. For these reasons we do not expect that large-scale changes to the evaluation strategy can be feasibly generalised, though it may be possible to develop heuristics for certain cases.

Gigascale makes use of CSBs to significantly improve both its memory efficiency and its cache utilisation. A CSB is a naturally portable data-structure, and we might devise means of applying it to arbitrary CFL-R problems. One option is to represent the CFL-R graph using a sparse representation based on CSBs. Whilst this is an appealing *memory saving* feature, computationally it may be ineffective. Gigascale could improve the performance of points-to using CSBs because it performed operations on them in aggregate (like merging and intersecting). Applying the CSB structures to the Melski-Reps worklist algorithm (Algorithm 2), for example, allows for some aggregate operations, such as extending the paths from lines 8 and 10, but also requires some "membership query" style operations (line 7). To capitalise on the advantages of CSB structures, an evaluation approach which favours aggregate operations (such as Chaudhuri's [17]) would need to be adopted. In general though, using CSBs is a secondary consideration; the actual advantage that Gigascale has over the worklist approach is that it uses data structures which are *cache-aware*. Any relatively concise data structure, such as representations that use contiguous memory blocks, or structures that are compressed, would be sufficient to maintain good cache performance.

Gigascale makes particularly efficient use of its CSBs by deriving its variable numbering scheme from a depth-first search. The worklist algorithm does not specify the need for a numbering, though for performance reasons numbering techniques may be used [14]. Numbering vertices according to the depth-first search significantly increased the likelihood that most descendant vertices (i.e. the heap objects that the variables might point to, which are always leaves) would be grouped into the same CSB blocks. For Gigascale, this was a good idea based on the specific language chosen, and the fact that only the points-to relation was needed (i.e. all leaves reachable from a vertex), rather than a more complex successor relation. In General, the CFL-R problem may not present such an ideal case. For example, even in the simple points-to language, the *bridge* relation ($\mathit{bridge} \rightarrow \mathit{load}_f\ \mathit{points\text{-}to}\ \overline{\mathit{points\text{-}to}}\ \mathit{store}_f$) would be unlikely to receive any benefit, as pairs in this relation occur between paths which involve backtracking, and hence their endpoints will have potentially very distant numberings.

On the other hand, intelligent vertex numbering schemes can be considered in many cases. Looking at the logic of the simple points-to analysis, we can devise at least a partition between vertices and heap objects mechanically. Using disjoint sets, and starting with a unique class representing the sources and sinks

of each relation (e.g. $load_{src}$), we traverse the grammar and unify two classes if they are adjacent in a rule (e.g. *points-to* $\rightarrow$ *assign points-to* causes $points\text{-}to_{src}$ to unify with $assign_{snk}$). Also, we unify a relation's source in the rule's head with the first label's source and the head's sink with the last label's sink (e.g. *points-to* $\rightarrow$ *assign* causes $points\text{-}to_{snk}$ to unify with $assign_{snk}$). In the case of the reverse relation $\overline{points\text{-}to}$, simply flip the source and sink classes. Applying this procedure to the whole grammar we can mechanically derive two distinct classes, one for the sinks of *assign* and *points-to* (i.e. the heap objects) and a second for the rest of the endpoints (i.e. variables). Endpoints from different classes are never joined to form a path, meaning that if the variables were grouped by endpoint classes, then only a *contiguous* subset of those vertices would need to be searched when extending paths. Thus, this simple vertex numbering technique may improve the cache utilisation of adjacency lookups and path construction for CFL-R problems.

Chapter 4

Algorithmic Improvements

This chapter demonstrates a practical and scalable approach to solving CFL-R problems. Much of this work was published to CC, 2015, in the paper "Towards a Scalable Framework for Context-Free Language Reachability" [35].

4.1 Opportunities

The CFL-R formalism is fundamentally important in the study and practice of program analysis. Some of the most important problems facing computer scientists today are captured by this class of problem, including formal security verification [28, 10], constraint solving [47], shape analysis [57], data- [59] and control-flow [79], set-constraints [42, 47], and alias analysis [89, 46, 91, 70, 85]. Although there is demand for effective CFL-R algorithms/solvers, which are able to solve the above problems, unfortunately the state of the art techniques fall short of this goal. The CFL-R problem is known to be difficult [56, 34], implying that it is difficult to devise scalable approaches for CFL-R.

Since general techniques can not handle large real-world problem instances, a one-off approach is often adopted on a *per problem* basis. In Chapter 3 we examined a simple field-insensitive points-to analysis. This kind of close examination demonstrates a typical problem facing CFL-R practitioners, namely that a lack of sophisticated and effective tools for solving CFL-R mean that significant manual effort is needed to develop scalable solvers. Further, the techniques that are effective for one problem may not be generalisable to all CFL-R instances; Section 3.3.3 discusses the specific techniques needed for the Gigascale analysis, and notes how these could be generalised. Indeed many CFL-R analyses can only be made scalable by adopting approaches which do not generalise [85, 90], meaning that much of the progress made by CFL-R researchers is not applicable to the formalism itself. To facilitate the study and use of CFL-R as a solving vehicle, we develop approaches which are both scalable and *general*.

The first area of weakness regarding the scalability of CFL-R is its algorithms. There are two well-known general approaches to solving arbitrary CFL-

R problems. Melski and Reps developed an algorithm based on dynamic programming [47], which is widely used in practice, but has cubic time-complexity. Further, the algorithm performs many wasteful computations, specifically related to rediscovering reachable paths that it already knows, and frequently searching through its grammar to find applicable rules for a given worklist edge. An improved algorithm, due to Chaudhuri [17], performs only $\mathcal{O}\left(\frac{n^3}{\log n}\right)$ work. Chaudhuri's faster approach adapts the Four Russians' Trick, increasing memory requirements significantly for a less noticeable speedup. Both algorithms are summarised in Section 2.2. Learning from Gigascale, we eschew these algorithms, which have superior theoretical time or space complexities, in favour of *practical* approaches, which capitalise on domain knowledge which the problem may exhibit.

A more effective approach exists for Datalog problems, known as the **semi-naïve** strategy [1]. The similarities between Datalog and CFL-R are well-known [86], meaning the application of Datalog evaluation to CFL-R is straightforward. Naïvely, it is possible to evaluate a Datalog problem simply by examining every combination of rules and facts until no new facts can be derived. This evaluation strategy is simple, but there are two observations which critically reduce the number of combinations that need to be examined:

- At any given time in the evaluation, potentially only a *subset* of the rules should be examined. For a rule like `A(x,y) :- B(y,z), C(x,z).`, it is clear that the `A` relation depends on `B` and `C`, thus it is only necessary to evaluate this rule when `B` and `C` are finished being evaluated. Note that rules in cycles need to be iteratively re-evaluated. We call the sequence for evaluating rules the **evaluation ordering**.

- When a rule is evaluated, only *recently discovered* information can possibly yield new results. Given `A(x,y) :- B(y,z), C(x,z).`, we only need to search for new `A` pairs if either a `B` or `C` pair was added since the last time we expanded this rule. We call the propagation of recent information to new data **delta expansion**, which is similar to the difference propagation strategy [69].

The second area of weakness in CFL-R's scalability lies in how data is used in the algorithms. The worklist algorithm (Algorithm 2) processes individual items from a worklist, and attempts to extend a path directly adjacent to this item. Using the data in this way does not capitalise on either spatial or temporal locality; we have no guarantees that the edges we need to search through are contiguous or local in memory, and we have no idea when a given worklist item or edge will be needed in the future, since the worklist is populated essentially at random during the search. For very high-performance applications, such as the Gigascale analysis [27], improving the cache utilisation yields significantly better performance than devising ways of minimising algorithmic complexity. Part of the data locality issues can be solved by adopting an evaluation which performs operations *in aggregate* (as opposed to individually), but better structures

are needed if general CFL-R evaluation is to scale as well as problem-specific alternatives.

To represent and operate on data in a cache-efficient way, we adapt the **quadtree** data structure [30] to CFL-R. Quadtrees are common in computer geometry and graphical applications as a means of representing data spatially. Quadtrees represent geometric data in a multi-dimensional fashion (quad- referring to the four quadrants of the 2D plane), where each "level" of the tree halves the range of values in the dimension, until a leaf node is reached which represents that an item exists spatially in that region. A quadtree is similar to the *kd-tree* data structure, though the former is flatter, by virtue of splitting on all dimensions at once, whilst the latter splits on a different dimension at every level. In the CFL-R context, we represent the input graph's adjacency matrix as a Cartesian plane, noting that quadtrees present an elegant means of evaluating matrix multiplication, which is necessary for finding paths in the graph.

We implement semi-naïve evaluation for CFL-R as an algorithmic template, which is agnostic to an abstract datatype (ADT) encoding for binary relations. Abstracting here facilitates several high-performance data-structures, including efficient and cache-aware quadtrees, as well as **B-trees**, which are used in commercial and research database implementations. Clients of the CFL-R analysis can even tailor their choice of data-structure to capitalise on problem-specific knowledge. Compared with Algorithm 2, the new evaluation template and data-structures exhibit better memory utilisation, improve the practical runtime performance, especially for sparse problems, and obviate the need for an expensive grammar-normalisation operation, which is required as a pre-step to the worklist approach.

We outline our contributions as follows:

- We develop a new algorithm template for CFL-R by specialising the semi-naïve Datalog evaluation strategy. The algorithm performs fewer redundant computations, avoids normalising the input grammar, and allows different relational ADTs to be implemented.

- We survey several data-structures to facilitate the semi-naïve template, specifically quadtrees. Quadtrees have efficient memory utilisation, especially for sparse problems, further improving its applicability.

- We analyse the algorithm's performance experimentally on Java points-to analysis and taint-analysis benchmarks, showing up to 11x speedup and 91% memory reduction. The use of differing benchmarks verifies the generality of our CFL-R template.

4.2 Algorithms

4.2.1 Preamble

Since we are developing a new evaluation strategy for CFL-R, we endeavour to show that this is indeed sound and precise with respect to the definition of the CFL-R problem (Definition 8). The important formal concepts concerning CFL-R were presented in Section 1.1.3, though a few additional ones are needed here. We first show that the CFL-R solution is equivalent to the fixpoint of a certain function, which incrementally summarises CFL-R paths from an initial graph. In traditional presentations of the CFL-R algorithm (and indeed of context-free languages), an assumption is made that the grammar is in Chomsky normal-form [20]. Whilst this form is useful for reasoning (since the number of derivation steps for recognising strings of length n is always $\mathcal{O}(n)$, in practice the algorithms are designed to handle arbitrary grammars, so we prove the stronger notion of their correctness for *any context-free grammar*. In Chomsky normal-form the number of steps needed to derive a string from the start symbol is based on the length of the string, but for our proofs we need a notion of the number of steps in that derivation.

Definition 9. *The **derivation length** between two strings is the number of derivations used to transform the initial string into the final string:*

$$\gamma \in \Sigma^* \text{ implies: } \gamma \overset{0}{\Rightarrow} \gamma$$

$$B \to \beta \in \mathcal{P} \wedge \alpha\beta\gamma \overset{i}{\Rightarrow} \delta \text{ implies: } \alpha B \gamma \overset{i+1}{\Longrightarrow} \delta$$

It is important to note that if a string can be derived, it can be derived in a finite number of steps.

Lemma 2. *If $\alpha \overset{*}{\Rightarrow} \beta$ then there must be an i such that $\alpha \overset{i}{\Rightarrow} \beta$.*

Proof. There are two cases from Definition 4: If $\alpha = \beta$, then we have $i = 0$ immediately. Otherwise, $\exists\gamma : \alpha \Rightarrow \gamma \wedge \gamma \overset{*}{\Rightarrow} \beta$. From Definition 3, $\alpha \Rightarrow \gamma$ implies that: $\alpha = \alpha_h A \alpha_t$, $\gamma = \alpha_h \gamma_m \alpha_t$, and $A \to \gamma_m \in \mathcal{P}$. So from the inductive hypothesis, $\alpha_h\gamma_m\alpha_t \overset{j}{\Rightarrow} \beta$ gives us $i = j + 1$. □

Corollary 1. *There is always a shortest derivation. If $\alpha \overset{*}{\Rightarrow} \beta$ then $\exists i$ such that $\alpha \overset{i}{\Rightarrow} \beta \wedge \forall\alpha \overset{j}{\Rightarrow} \beta : i \leq j$.*

We intend to induct on the derivation steps, but this is impossible when a derivation is *cyclic*. To deal with cycles, we point out an important property of the shortest derivation: that the shortest chain of derivations must be acyclic.

Lemma 3. *If $\alpha \overset{i}{\Rightarrow} \beta$ is the shortest derivation, then $\nexists j > 0$ having the property that $\alpha \overset{k}{\Rightarrow} \gamma \overset{j}{\Rightarrow} \gamma \overset{i-j-k}{\Longrightarrow} \beta, k \geq 0$.*

Proof. i is the smallest, but $\alpha \overset{k}{\Rightarrow} \gamma \overset{i-j-k}{\Longrightarrow} \beta$ has derivation length $i - j < i$, a contradiction. □

The solution to the CFL-R problem is itself defined as all pairs that can be reached by a (possibly empty) path spelling a word in L. In a more general sense, we may need all paths that can be reached by any non-terminal in L's grammar. Fortunately, a Language can be broken up into arbitrary sub-languages.

Definition 10. *Given a language L with grammar $(\mathcal{T}, \mathcal{N}, \mathcal{P}, S)$, i.e. $\omega \in L$ implies that $S \overset{*}{\Rightarrow} \omega$, there is a **sub-language** L_N $\forall N \in \mathcal{N}$, defined by the grammar $(\mathcal{T}, \mathcal{N}, \mathcal{P}, N)$.*

4.2.2 Fixpoint Solution

The Definition of the CFL-R solution (Definition 8) gives us exactly the "all-pairs" solution of the CFL-R problem, which we treat as the most general solution. However, since the definition of the CFL-R solution depends on infinite sets (like Π and L), it is unclear if (or how) it can be computed; it is simply stated that the CFL-R problem has a finite, computable solution [47].

We now desire to show formally that the solution is both finite and decidable. We require these properties in order to show that a given algorithm (such as Melski and Reps', or our own) indeed computes the *correct* solution. Decidability follows from demonstrating that CFL-R solutions have a fixpoint calculation, where the solution is the least fixpoint. To this end we show a bijection between the solution space and a *configuration* lattice. The intuition is that non-terminal edges in a configuration can summarise the existence of paths in the language of that non-terminal. Let $\mathfrak{L} = \mathcal{T} \cup \mathcal{N}$ refer agnostically to terminals or nonterminals.

Definition 11. *A **configuration** $c \in 2^{\mathfrak{L} \times V \times V}$, is a member of the complete lattice of configurations, ordered by $\subseteq$, with $\bot = \emptyset$ and $\top = \mathfrak{L} \times V \times V$. We reuse the shorthand notation $A(u, v) \in c \Rightarrow u, v \in V, A \in \mathfrak{L}$.*

Additionally, new non-terminal edges can be derived from an input configuration, when a path labelled with the right-hand-side of its production exists. The *configuration-expansion* function is used to extend a partial solution in this way:

Definition 12. *Given a CFL-R problem for the language L with grammar $(\mathcal{T}, \mathcal{N}, \mathcal{P}, S)$ and the graph $G = (V, E)$, there exists a **configuration expansion** function $\mathcal{F} : 2^{\mathfrak{L} \times V \times V} \to 2^{\mathfrak{L} \times V \times V}$ defined as:*

$$\begin{aligned}\mathcal{F}(c) =& c \cup \{A(u,u) \mid u \in V, A \to \varepsilon \in \mathcal{P}\} \cup \\ & \{A(v_0, v_k) \mid A \to B_1 \dots B_k \in \mathcal{P}, B_1(v_0, v_1) \in c, \dots, B_k(v_{k-1}, v_k) \in c\}\end{aligned}$$

According to the Tarski-Knaster theorem [74], repeated applications of $\mathcal{F}$ will lead to a fixpoint solution if and only if the fixpoint function is monotonically increasing over a complete, finite domain. The monotonicity of $\mathcal{F}$ is shown here:

Lemma 4. $c \subseteq c' \Rightarrow \mathcal{F}(c) \subseteq \mathcal{F}(c')$.

Proof. Assume $e = A(u, v) \in \mathcal{F}(c)$, structurally either $e \in \{A(u, u) \mid u \in V, A \to \varepsilon \in \mathcal{P}\} \subseteq \mathcal{F}(c')$, or $A \to B_1 \dots B_k \in \mathcal{P}$, and $\forall 0 < i \leq k : B_i(v_{i-1}, v_i) \in c \subseteq c'$, then $A(v_0, v_k) = A(u, v) \in \mathcal{F}(c')$. □

Thus the least-fixpoint solution can be computed by Kleene's iteration on the monotonic function. We term the fixpoint $\mathit{lfp}_{\mathcal{F}} = \mathcal{F} \circ \ldots \circ \mathcal{F}(E)$. Then equivalence between the least-fixpoint $\mathit{lfp}_{\mathcal{F}}$ and the CFL-R solution is sufficient to prove that CFL-R is decidable.

Lemma 5. $(u, v) \in \text{CFL-R}(L, G)$ *if and only if* $S(u, v) \in \mathit{lfp}_{\mathcal{F}}$

Proof. In the "if" direction we induct on the applications of $\mathcal{F}$.

- The base case has $S(u, v) \in \mathcal{F}(E)$. Possibly $S \to \varepsilon \in \mathcal{P}$ and $u = v$, hence $(u, v) \in \{(v, v) \mid \varepsilon \in L, v \in V\}$. Otherwise $S \to T_1, \ldots, T_k \in \mathcal{P}$, $\forall i \in [1, k] : T_i(w_{i-1}, w_i) \in E$ and $u = w_0, v = w_k$, hence $\langle T_1(w_0, w_1), \ldots, T_k(w_{k-1}, w_k)\rangle \in \Pi$ and $S \overset{*}{\Rightarrow} T_1 \ldots T_k$.
- Inductively, $S(u, v) \in \mathcal{F}^i(E) \setminus \mathcal{F}^{i-1}(E), i > 1$. Thus, $A \to B_1 \ldots B_k \in \mathcal{P}$ with $\forall j \in [1, k] : B_j(w_{j-1}, w_j) \in \mathcal{F}^{i-1}(E)$ (the ε case would have occurred in the base case it can not occur now). Using Definition 10 and the inductive hypothesis, we know $(w_{j-1}, w_j) \in \text{CFL-R}(L_{B_j}, G)$, so let p_j be the path having $\omega(p_j) \in L_{B_j}$. Then with $p = p_1 \bullet \ldots \bullet p_k$, we have $p \in \Pi$ and $S \Rightarrow B_1 \ldots B_k \overset{*}{\Rightarrow} \omega(p)$, hence $(w_0, w_k) = (u, v) \in \text{CFL-R}(L, G)$.

In the "only if" direction, $(u, v) \in \text{CFL-R}(L, G)$ implies that there is some $p \in \Pi$ having $S \overset{d}{\Rightarrow} \omega(p)$, with d the smallest number of derivations. Induct on d.

- $d = 0$ is impossible, since it implies $\omega(p) = S$ even though $S(u, v) \notin E$.
- When $d = 1$ there are two options, possibly $S \Rightarrow \varepsilon$, but then $u = v$, $p = \langle\rangle$ and $S \to \varepsilon \in \mathcal{P}$, hence $S(v, v) \in \mathcal{F}(E)$. Otherwise $S \Rightarrow T_1 \ldots T_k$ and $p = \langle T_1(w_0, w_1), \ldots, T_k(w_{k-1}, w_k)\rangle$, hence $\forall j \in [1, k] : T_j(w_{j-1}, w_j) \in E$ which causes $S(w_0, w_k) = S(u, v) \in \mathcal{F}(E)$.
- Inductively, given $d > 1$, there must be a $S \to B_1 \ldots B_k \in \mathcal{P}$, which grants that $S \Rightarrow B_1 \ldots B_k \overset{d-1}{\Longrightarrow} \omega(p)$. Partition p into $p_1, \ldots, p_k$ such that $\forall j \in [1, k] : B_j \overset{*}{\Rightarrow} p_j$. Using Definition 10 we have that $B_j(w_{j-1}, w_j) \in \mathit{lfp}_{\mathcal{F}}$, which completes that $S(w_0, w_k) = S(u, v) \in \mathit{lfp}_{\mathcal{F}}$

□

This establishes the correctness of solutions computed via some configuration expansion fixpoint. The equivalence between $\mathit{lfp}_{\mathcal{F}}$ and the Melski-Reps algorithm [47] was discussed informally in their work, and since our contribution is a different algorithm, we will present only the equivalence of the fixpoint calculation to our novel formulation.

4.2.3 CFL-R Semi-naïve

We design our algorithm by using Datalog **semi-naïve** evaluation [1] as a scaffold. The principal idea is to avoid redundant computations by tracking recently

Algorithm 4 Naïve Relational CFL-R.

```
1: procedure NAÏVE(𝓛, G)
2:     for all A ∈ 𝔏 do
3:         𝒢(A) ← {(u, v) | A(u, v) ∈ E}
4:         if A → ε ∈ 𝒫 then
5:             𝒢(A) ← {(u, u) | u ∈ V}
6:     while 𝒢 is growing do
7:         for all A → B_1 ... B_k ∈ 𝒫 do
8:             𝒢(A) ← 𝒢(A) ∪ 𝒢(B_1) ∘ ... ∘ 𝒢(B_k)
```

discovered knowledge, called delta-sets Δ, and propagating these deltas to discover new Datalog facts. The algorithm we present is a template which abstracts the implementation of some relational abstract datatype (ADT). The contract of the ADT is a binary relation, encoding the operations of a Boolean lattice (union, intersection, complement) as well as difference and **composition**, defined as:

$$(a, b) \in \mathbf{R}, (b, c) \in \mathbf{S} \Rightarrow (a, c) \in \mathbf{R} \circ \mathbf{S}$$

The correctness of semi-naïve is shown by first proving the correctness of an intermediate approach, a naïve fixpoint computation shown in Algorithm 4, then extending that proof to the completed semi-naïve formulation shown in Algorithm 5. Note that naïve evaluation is only a proof vehicle, we do not experiment with such an algorithm, on the intuition that its unfavourable performance will be as true of CFL-R as it was of Datalog [1].

Algorithm 4 closely emulates the configuration expansion function from Definition 12. The algorithm initially discovers the input and epsilon relations, before the while loop on Line 6. Subsequently, new relations are discovered by iteratively composing known ones, according to the production chosen by Line 7.

We demonstrate its execution on the example in Figure 1.2. Current knowledge $\mathcal{G}$ is seeded with input relations, so that e.g.:

$$\mathcal{G}(\mathit{assign}) = \{(\mathit{this}, a), (\mathit{this}, b), (a2, \mathit{other}), (b2, \mathit{other}), (\mathit{other}, \mathit{th}5), (\mathit{other}, \mathit{th}6)\}$$

Note that non-terminals can only have ε-relations at this stage, which the points-to analysis' grammar does not produce, hence $\mathcal{G}(\mathit{points\text{-}to}) = \emptyset$. After Line 6 is reached, we iterate over productions. Let $\mathit{points\text{-}to} \rightarrow \mathit{alloc}$ be chosen first, then Line 8 becomes:

$$\mathcal{G}(\mathit{points\text{-}to}) \leftarrow \mathcal{G}(\mathit{points\text{-}to}) \cup \mathcal{G}(\mathit{alloc})$$

i.e. variables that allocate their heap objects must point-to them Next, let $\mathit{points\text{-}to} \rightarrow \mathit{assign}\ \mathit{points\text{-}to}$ be chosen, such that $(\mathit{this}, H1) \in \mathcal{G}(\mathit{points\text{-}to})$ since $(\mathit{this}, a) \in \mathcal{G}(\mathit{assign})$ and $(a, H1) \in \mathcal{G}(\mathit{points\text{-}to})$. The iterations continue until no new points-to paths can be discovered.

Algorithm 5 Semi-naïve Relational CFL-R.

```
1: procedure SEMI_NAÏVE(ℒ, G)
2:     for all A ∈ 𝔏 do
3:         𝒢(A) ← {(u, v) | A(u, v) ∈ E}
4:         if A → ε ∈ 𝒫 then
5:             𝒢(A) ← {(u, u) | u ∈ V}
6:         Δ_A ← 𝒢(A)
7:     for all [C] ∈ RTDG(𝒫) do
8:         while ∃D ∈ [C], Δ_D ≠ ∅ do
9:             Δ_cur ← Δ_D
10:            Δ_D ← ∅
11:            for all A → B_1 ... D ... B_k ∈ 𝒫 do
12:                Δ_A ← Δ_A ∪ (𝒢(B_1) ∘ ... ∘ Δ_cur ∘ ... ∘ 𝒢(B_k) \ 𝒢(A))
13:                𝒢(A) ← 𝒢(A) ∪ Δ_A
```

Lemma 6 (Naïve correctness)**.** $A(u, v) \in lfp_{\mathcal{F}} \Leftrightarrow (u, v) \in \mathcal{G}(A)$.

Proof. Inducting on the number of iterations of the While-loop (Line 6), noted $\mathcal{G}^i$, against the number of applications of $\mathcal{F}$.

In the base case $(u, v) \in \mathcal{G}^0(A) \Leftrightarrow A(u, v) \in E \cup \{A(u, u) \mid u \in V, A \to \varepsilon \in \mathcal{P}\} \Leftrightarrow \mathcal{F}^0(E) \cup \{A(u, u) \mid u \in V, A \to \varepsilon \in \mathcal{P}\} \subseteq \mathcal{F}^1(E)$, from Lines 3 and 5.

Then inductively $(u, v) \in \mathcal{G}^{i+1}(A) \setminus \mathcal{G}^i(A) \Leftrightarrow A \to B_1 \dots B_k \in \mathcal{P} \wedge (u, w_1) \in \mathcal{G}^i(B_1) \wedge \dots \wedge (w_{k-1}, v) \in \mathcal{G}^i(B_k) \Leftrightarrow A(u, v) \in \{A(v_0, v_k) \mid A \to B_1 \dots B_k \in \mathcal{P}, B_1(v_0, v_1) \in \mathcal{F}^{i+1}(\emptyset), \dots, B_k(v_{k-1}, v_k) \in \mathcal{F}^{i+1}(\emptyset)\} \subseteq \mathcal{F}^{i+2}(\emptyset)$, from Line 8[1]. □

Our Semi-naïve evaluation strategy is presented in Algorithm 5. Left informally here is the reverse-topological-dependency-grouping $\text{RTDG}(\mathcal{P})$ function, similar to the *precedence graph* from semi-naïve evaluation [1]. $\text{RTDG}(\mathcal{P})$ can be pre-computed in time linear to the size of the grammar.

Definition 13 (Reverse Topological Dependency Grouping)**.**

$$A \geq B \Leftrightarrow A \to \dots B \dots \in \mathcal{P}$$
$$\text{RTDG}(\mathcal{P}) = [C_1], [C_2], \dots \text{ such that } i \leq j \Leftrightarrow \forall A \in [C_i], B \in [C_j] : A \not\geq B$$

Algorithm 5 extends Algorithm 4 with the notion of difference sets, labelled Δ. Instead of finding new relations by examining all prior knowledge, we require one of the Δ-relations be used. The Δ sets are initially identical to the input edges, Line 6.

[1] There is a slight misalignment here, potentially $B_j(w_{j-1}, w_j)$ was discovered this iteration, not last iteration. We have elided this case for brevity, its effect on the proof is to reduce i by 1.

For the running example from Figure 1.2 we observe the evaluation of points-to by the semi-naïve algorithm. At line 7, assume the reverse topological dependency ordering has:

$$\langle \{\textit{assign}, \textit{load}, \textit{alloc}, \textit{store}\}, \{\textit{points-to}\} \rangle$$

as the ordered groups, and Δ_{assign} is not empty, so we have $\Delta_{cur} = \Delta_{assign} = \textit{assign}$. For this iteration, only $\textit{points-to} \rightarrow \textit{assign}\ \textit{points-to}$ is used, i.e. $\Delta_{points\text{-}to} \leftarrow \Delta_{assign} \circ \emptyset$, so no new relations are added. Only Δ_{alloc} will add relations to *points-to* for $[C_1]$, so when $[C_2] = \{\textit{points-to}\}$ is chosen, $\Delta_{points\text{-}to}$ is not empty. Assume the load-store rule was chosen, then since *points-to* appears three times in this rule, there will be three evaluations:

$$\Delta_{points\text{-}to} \leftarrow \Delta_{points\text{-}to} \cup \mathcal{G}(\textit{load}) \circ \Delta_{cur} \circ \overline{\mathcal{G}(\textit{points-to})} \circ \mathcal{G}(\textit{store}) \circ \mathcal{G}(\textit{points-to})$$
$$\Delta_{points\text{-}to} \leftarrow \Delta_{points\text{-}to} \cup \mathcal{G}(\textit{load}) \circ \mathcal{G}(\textit{points-to}) \circ \overline{\Delta_{cur}} \circ \mathcal{G}(\textit{store}) \circ \mathcal{G}(\textit{points-to})$$
$$\Delta_{points\text{-}to} \leftarrow \Delta_{points\text{-}to} \cup \mathcal{G}(\textit{load}) \circ \mathcal{G}(\textit{points-to}) \circ \overline{\mathcal{G}(\textit{points-to})} \circ \mathcal{G}(\textit{store}) \circ \Delta_{cur}$$

as well as a single evaluation for the $\textit{points-to} \rightarrow \textit{assign}\ \textit{points-to}$. Some new $\Delta_{points\text{-}to}$ edges have been discovered by these evaluations, so another round with $\Delta_{cur} = \textit{points-to}$ will occur. As we shall see, the semi-naïve evaluation strategy is equivalent to the naive formulation from Algorithm 4.

Lemma 7. *With $\mathcal{G}$ referring to Algorithm 5, and $\hat{\mathcal{G}}$ referring to Algorithm 4:* $(u, v) \in \mathcal{G}(A) \Leftrightarrow (u, v) \in \hat{\mathcal{G}}(A)$

Proof. We induct $\Rightarrow$ on the ith iteration of Algorithm 5 Line 12. We induct $\Leftarrow$ on the jth iteration of Algorithm 4 Line 6. The base cases for both inductions arises from: $\mathcal{G}^0(A) = \hat{\mathcal{G}}^0(A) = \{(u, v) \mid A(u, v) \in E\} \cup \{(u, u) \mid A \rightarrow \varepsilon, u \in V\}$.

For soundness ($\Rightarrow$), $(u, v) \in \Delta_A^i \Leftrightarrow A \rightarrow B_1 \dots B_d \dots B_k \wedge (u, w_1) \in \mathcal{G}^{i-1}(B_1) \wedge \dots \wedge (w_{d-1}, w_d) \in \Delta^{i-1}(B_d) \wedge \dots \wedge (w_{k-1}, v) \in \mathcal{G}^{i-1}(B_k)$. Then the inductive hypothesis gives us $(u, w_1) \in \hat{\mathcal{G}}(B_1) \wedge \dots \wedge (w_{k-1}, v) \in \hat{\mathcal{G}}(B_k) \Rightarrow (u, v) \in \hat{\mathcal{G}}(A)$.

For completeness ($\Leftarrow$), $(u, v) \in \hat{\mathcal{G}}^j(A) \Leftrightarrow (u, w_1) \in \hat{\mathcal{G}}^{j-1}(B_1), \dots, (w_{k-1}, v) \in \hat{\mathcal{G}}^{j-1}(B_k)$, so inductively, $(u, w_1) \in \mathcal{G}(B_1), \dots, (w_{k-1}, v) \in \mathcal{G}(B_k)$. But, $A \rightarrow B_1 \dots B_k \in \mathcal{P} \Rightarrow A \geq B_1, \dots, B_k$, and $\exists d \in [1, k] : \forall h \in [1, k] : B_d \not< B_h$, i.e. all the B labels are less than or equal to A, and at least one of them is no less than all the others. So $\exists i$ such that $(u, w_1) \in \mathcal{G}^{i-1}(B_1), \dots, (w_{d-1}, w_d) \in \Delta_{B_d}^i, \dots, (u, w_1) \in \mathcal{G}^{i-1}(B_k) \Leftrightarrow (u, v) \in \mathcal{G}^{i+1}(A)$. □

We now discuss the complexity of our template by assuming known time-complexities for the ADT's operations. This produces a parametric complexity function, which gives the true complexity after specifying a concrete datatype. A sketch of the proof is shown below.

Lemma 8 (Semi-naïve complexity)**.** *For problems with k labels and n vertices, and where $\cup, \setminus$ and $\circ$ have unknown time-complexity,* SEMI_NAÏVE *terminates in* $\mathcal{O}\left(k^2 n^2 (\cup + \setminus + k\circ)\right)$ *time.*

Proof sketch. The entire work of the For-loop (Line 2) is completed in $\mathcal{O}\left(kn + kn^2\right)$ time, since it writes at most each edge once, and every vertex's epsilon-edge for at most all labels.

The loops on Lines 7 and 8 loop while there is a non-empty Δ. Deltas are updated on one of k iterations of the for-loop on Line 11, by the composition on Line 12. In the worst case, only one new pair will be added to the Δ, so kn^2 iterations in total. Each iteration in-turn causes k loops of Line 11, and in each iteration, one $\setminus$, two $\cup$, and k $\circ$ operations take place, totalling: $\mathcal{O}\left(kn + kn^2 + kn^2(k(\cup + \setminus + k\circ))\right) = \mathcal{O}\left(k^2n^2(\cup + \setminus + k\circ)\right)$ □

4.3 Data Structures

The semi-naïve CFL-R algorithm from Section 4.2.3 is dependant on a user-chosen relational abstract data-type (ADT). In practice, the chosen ADT depends on properties of the input. Our templated semi-naïve solver is designed around the idea of facilitating arbitrary relational ADTs.

Our main relational ADT contribution in this paper is the **quadtree** formulation of Boolean matrices. With suitable operations, quadtrees are efficient both in space and computational cost, especially for the sparse problems encountered in our taint-analysis and points-to experiments (Section 4.4). We show the theoretical advantages of quadtrees, and detail the implementation and optimisations necessary for high performance.

The quadtrees are compared against standard implementations of relations via **neighbourhood maps**. A neighbourhood map is a kind of lookup table that lists, for a given element x of the relational domain, all the elements that x relates to (the successor neighbourhood) or that relate to x (the predecessor neighbourhood). The neighbourhood maps in our evaluation are implemented via red-black trees [12], hash-tables, and B-trees [11]. A summarisation of the theoretical space and time bounds for the different relational ADT implementations is presented in the following table:

ADT	Space Complexity	Time Complexity (composition)
Quadtrees	$\mathcal{O}\left(\min(n^2, m\log n)\right)$	$\mathcal{O}\left(\min(n^3, m^2 n\log n)\right)$
Concise quadtrees	$\mathcal{O}\left(\min(n^2, m\log n)\right)$	$\mathcal{O}\left(\min(n^3, m^2 n\log n)\right)$
Red-black trees	$\mathcal{O}\left(m\right)$	$\mathcal{O}\left(m^2\log n + n\log n\right)$
Hash maps	$\mathcal{O}\left(m\right)$	expected $\mathcal{O}\left(m^2 + n\right)$
B-trees	$\mathcal{O}\left(m\right)$	$\mathcal{O}\left(m^2\log n + n\log n\right)$

4.3.1 Quadtrees

The primary data-structure of interest in this discussion is an adaptation of **quadtrees** to the semi-naïve evaluation. The adaptation forms one of the contributions of this thesis, as we studied the intricacies of quadtrees in relation to CFL-R and made tradeoffs in their design choices relevant to this field.

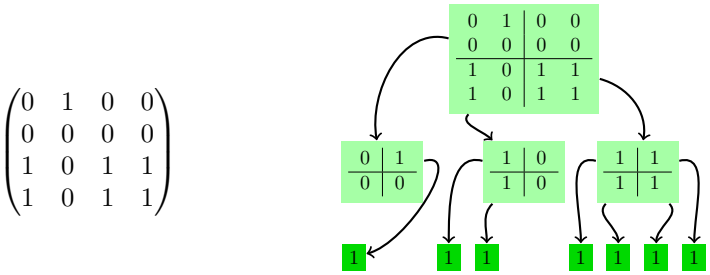


Figure 4.1: Depiction of how a Boolean matrix (left) is encoded in a quadtree(right).

Quadtrees are well-known for computational geometry applications, where spatial information is useful [30]. We first present Boolean matrices as a relational structure, and later encode these matrices as quadtrees.

Given a mapping between the relational domain and the integers, we can encode a relation $\mathbf{A}$ as a Boolean matrix $\hat{\mathbf{A}}$ via:

$$\hat{\mathbf{A}}_{ij} = \begin{cases} 1, & (i,j) \in \mathbf{A} \\ 0, & \text{otherwise} \end{cases}$$

which leads to the definitions $\mathbf{A} \cup \mathbf{B} = \hat{\mathbf{A}} \vee \hat{\mathbf{B}}$, $\mathbf{A} \setminus \mathbf{B} = \hat{\mathbf{A}} \wedge \neg\hat{\mathbf{B}}$ and $\mathbf{A} \circ \mathbf{B} = \hat{\mathbf{A}} \cdot \hat{\mathbf{B}}$. If we use Boolean matrices for the relational structure, composition would take $\mathcal{O}\left(n^{2.3}\right)$ time [23], and union and difference would take $\mathcal{O}\left(n^2\right)$ time. Furthermore we desire better storage than the $\Theta(n^2)$ space requirements of a dense Boolean matrix.

Instead, we store the Boolean matrix in a quadtree structure, as shown in Figure 4.1. The motivating factor behind quadtrees is their superior performance for sparse datasets. For the relation *alloc* from Figure 1.2, which has few elements element, a binary matrix representation requires $\mathcal{O}\left(n^2\right)$ memory, whereas a quadtree representation has only $\mathcal{O}\left(\log n\right)$ tree-nodes. Indeed, for the points-to analysis, all load and store relations have very few elements, so the advantages of quadtrees become more obvious. The sparsity bound is shown below:

Lemma 9. *The quadtree requires* $\mathcal{O}\left(\min(n^2, m \log n)\right)$ *space to store.*

Proof. Consider the complete matrix with side-length n, its quadtree has n^2 leaves, each representing a single 1 element, with the parent layer having $\frac{1}{4}$ as many nodes as its child layer. The total number of nodes is at most:

$$\sum_{f=0}^{\infty} n^2 \frac{1}{4^f} = \frac{n^2}{1 - \frac{1}{4}} = \frac{4n^2}{3}$$

Sparser matrices strictly remove nodes, hence $\frac{4n^2}{3}$ is an upper bound. For a more practical bound, we say that $m < n^2$ bits are set. In this case, it takes at most

$\log n$ nodes to join each of its m set bits to the root of the tree. This count is bounded above by the known n^2 limit, requiring $\min(n^2, m \log n)$ constant-sized nodes. □

Corollary 2. *The time complexity of union, intersection, set-difference, and copy operations is also* $\mathcal{O}\left(\min(n^2, m \log n)\right)$.

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right)\left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right) = \left(\begin{array}{c|c} A_{11}B_{11} \cup A_{12}B_{21} & A_{11}B_{12} \cup A_{12}B_{22} \\ \hline A_{21}B_{11} \cup A_{22}B_{21} & A_{21}B_{12} \cup A_{22}B_{22} \end{array}\right) \tag{4.1}$$

Equation 4.1 shows the recursive procedure for quadtree multiplication. This formulation allows us to determine both an upper-bound and average-case complexity for the composition operation.

Lemma 10. *Multiplication of two quadtrees requires* $\mathcal{O}\left(\min(n^3, m^2 n \log n)\right)$ *time.*

Proof. We enumerate the work at each level of recursion (i.e. the multiplications for all trees at height $1 \leq i < \log n$) similar to the master theorem. The work of multiplication at level i is $M_i = 8M_{i-1} + 4U_{i-1} + 1$, 8 recursive calls to multiplication, 4 unions and a constant amount. The key observation is that the work of four unions at level i can not exceed the work of one union at $i+1$, since at worst it must create the quadtree that is used in the upper-layer's union. From Corollary 2 we have that the uppermost unions for the complete and average cases are n^2 and $m^2 \log n$ respectively, since the multiplication of two m-dense matrices can not create one denser than m^2. The complete case and the average case appear side-by side.

$$\begin{array}{rcl|r} n^2 + 1 & & & m^2 \log n + 1 \\ 8(n/2)^2 + 8 & = & 2n^2 + 8 & 2m^2 \log n + 8 \\ 64(n/4)^2 + 64 & = & 4n^2 + 64 & 4m^2 \log n + 64 \\ & \cdots & & \cdots \\ 8^{\log n}(n/2^{\log n})^2 + 8^{\log n} & = & n(n^2) + n^3 & m^2 + m^2 \end{array}$$

Summing over all terms, the complete case yields n^3 as expected. For the average case, note that the lowest level requires $m^2 + m^2$ work. Since we know each input matrix has at most m leaves, we can not expect to reach the bottom level of recursion more than m^2 times (every pair of leaves), then no more than m^2 1-high unions and constant operations can occur. Summing over all terms, the average case yields $m^2 n \log n + m^2 \log n$. Hence the time required is $\mathcal{O}\left(min(n^3, m^2 n \log n)\right)$. □

The normal implementation of quadtrees makes use of heap-allocated tree-nodes with pointer references to sub-quadtrees. On a RAM, this retains the theoretical performance characteristics shown above. Using a recursive data structure presents several advantageous properties.

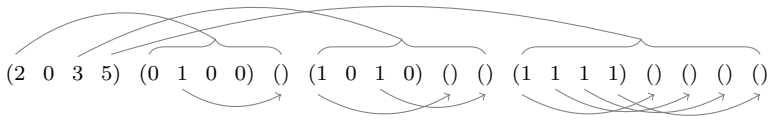


Figure 4.2: A concise representation of the quadtree in Figure 4.1. Each group of 4 integers is a node in the concise tree, the value records how many nodes are in the sub-quadrants of this node, in the order top-left, top-right, bottom-left, bottom-right. () represents a leaf node, whose value is irrelevant.

In certain cases, the union of two quadtrees subsequently disposes of one. Such an **absorbing-union** occurs in Algorithm 5, Line 12, between the temporary composed term and the Δ-term. Empty subtrees in the absorbing quadtree adopt the node from the absorbed quadtree (and all of its children), which avoids excessive memory allocation/deallocation. Similar tricks can be applied to the set-minus operation.

Beyond the basic implementation strategy, there are some optimisations which can be employed. These optimisations present a tradeoff in time/space complexity, but due to the practical advantages of cache locality, look-ahead, and memory-bandwidth, they usually provide significant advantages.

4.3.1.1 Concise Representation

The most glaring deficiency of the pointer-based implementation of quadtrees is its unpredictable memory behaviour. Look-ahead caching can not occur for the pointers, resulting in frequent cache-misses and a saturated memory bus. The efficiency limitations of pointer-structures, such as linked lists and trees, is well known [2].

To facilitate the simple cache heuristics which modern processors employ, we store quadtrees in contiguous memory regions with a **concise** representation. Concise representations are derived from a long-known bijection between rooted trees and properly-balanced parenthesis structures [29], they are also called *succinct* data structures [51]. Each quadtree node is represented by a tuple of four integers, where tuple elements 1,2,3 and 4 refer to the number of nodes in the quadtree rooted at this node's top-left, top-right, bottom-left and bottom-right quadrants respectively. Leaf nodes (i.e set bits in the matrix) have an arbitrary value, as only their presence or absence is significant. The representation uses more memory than necessary (only the presence of a subtree needs to be recorded), in order to facilitate skipping over subtrees that are irrelevant to the current computation (i.e. for faster membership checks). A concise encoding of the quadtree in Figure 4.1 is shown in Figure 4.2. Since the top-left subtree has two nodes (the node itself and its top-right leaf), the first node records this with a 2 in tuple position 1.

Concise representations have significant practical advantages. The tree is represented in a contiguous block, so copy operations such as Algorithm 5 line 6

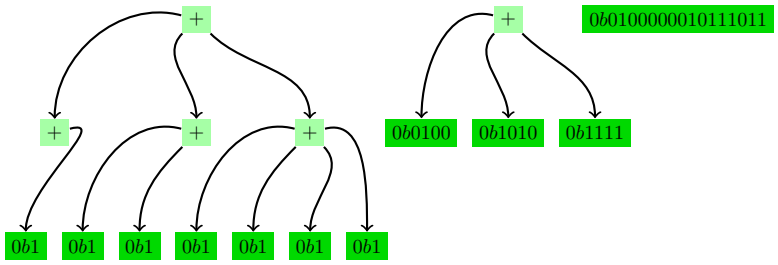


Figure 4.3: The Figure 4.1 matrix encoded with leaf bitmaps of 1-bit (i.e. as normal), 4-bits and 16-bits.

can be performed by fast low-level `memcpy` operations. Union and set-minus operations access the tree in a left-to-right manner, which makes cache prefetching possible. Conversely, concise trees are not easily mutable. Leaves must be inserted in order (leaf a is inserted before leaf b if it would appear to the left of b in the concise representation), to prevent shuffling the array. Furthermore, union, set-minus and composition can not occur "in-place", demanding additional data copying. In practice, cache locality justifies the tradeoff, as we show in Section 4.4.

Note that the additional cost of maintaining the concise representation does not change the order-theoretic properties of quadtree operations. Consider the union of two concise-trees. An inefficient implementation involves utilising a stack to walk the two inputs first-to-last. When a "leaf" node is encountered in either, inserting it is at most an $\mathcal{O}(\log n)$ operation, since it will always be inserted in the last position of the output, with no more than $\log n$ new nodes being required and $\log n$ updates to current nodes to account for the new subtree. The total work is therefore bounded by $\mathcal{O}(m \log n)$, the same as the previous result, though the initial allocation of the output list must conservatively be as large as **both** inputs. Since the complexity of multiplication is derived from union, the same argument holds in this case.

4.3.1.2 Leaf Bitmaps

As depicted in Figure 4.1, leaf nodes are represented with a full tree-node (whose child pointers become irrelevant). In a practical implementation, this implies using 4×64-bit pointers to encode 1 bit of information (the presence or absence of a leaf). For example, assuming leaf nodes have the same memory size as non-leaves (i.e. 4×64-bit pointers) one node can encode the presence-or-absence of all the leaves in a 16×16 subtree.

Figure 4.3 compares leaf-bitmap encodings for 1×1, 2×2 and 4×4 leaf-matrices. This optimisation is strictly advantageous for a pointer-based implementation. We no longer require nodes for 4 levels above a leaf, all of which are encoded in the bitmap of the 5th-lowest node. In the most advantageous

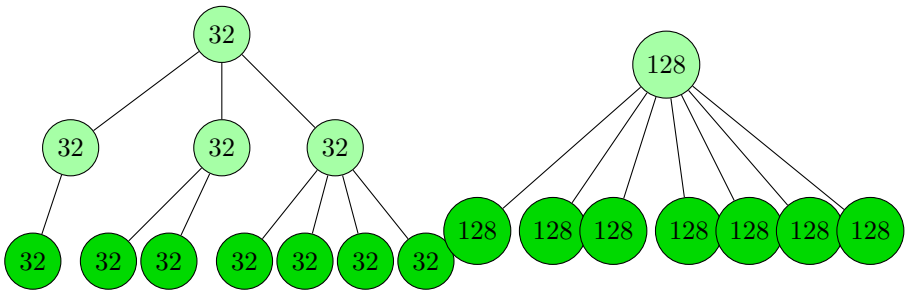


Figure 4.4: Side-by-side comparison of the nodes in a quadtree and a 16-tree, both representing the matrix in Figure 4.1. Nodes are labelled with their size (in bytes).

case, a complete subtree with 340 nodes (10 KB), is represented by a single node (32 B). In practice, problem instances are highly sparse, which reduces the saved-memory substantially. Using a scheme with 256-bit leaf matrices, the Figure 4.1 matrix is encoded in a single node.

When applied to a concise representation (Section 4.3.1.1), this optimisation is not so straightforward. For the subtree-sizes concise-representation, a concise-node is only 128-bits, which is too small for a 16×16 tree, and larger than the next possible 8×8 subtree. If we used 64-bit integers, the larger tree could be represented in a single concise-node (similar to the pointer-based implementation), but this increases the size of parent nodes. For the subtree-presence concise-representation, leaf bitmaps are worse for sparse graphs. Using a similar argument to Lemma 9, it can be shown that the subtree-presence encoding is smaller than a leaf bitmap when the number of set bits is less than $4 \log(\textit{bitmap-side-length})$. Furthermore, in both cases the size of the bitmap can be larger or smaller than the "node size". It is easy to track the depth of the tree walk, and switch to bitmap evaluation when necessary. A 128×128 bitmap causes the switch to occur $\log 128 = 7$ levels above the normal "leaf-level".

Efficient multiplication of the leaf bitmaps is outside the scope of this work. A comparison of existing techniques for multiplying dense-bitmap representations is discussed in [9].

4.3.1.3 Node Squashing

The cache-locality issues of the pointer representation can be alleviated by using larger nodes, in the style of a B-tree. Nodes in this style that squash n layers contain 4^n pointers to subtree nodes. Traversing the pointer-tree causes a lookup for every node, hence using fewer large-nodes has a clear advantage. Just as for B-trees, the size of the node is chosen to optimise the cache-locality properties. On a normal desktop machine, the 512 B 64-tree nodes pose a good tradeoff, whereas for servers (with larger caches) the 2 KB 256-tree nodes are superior.

The disadvantage of this mechanism lies in the over-provision of pointers in

$$
\begin{aligned}
I &= (A_{11} + A_{22})(B_{11} + B_{22}) & V &= (A_{11} + A_{12})B_{22} \\
II &= (A_{21} + A_{22})B_{11} & VI &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
III &= A_{11}(B_{12} - B_{22}) & VII &= (A_{12} - A_{22})(B_{21} + B_{22}) \\
IV &= A_{22}(B_{21} - B_{11})
\end{aligned}
$$

$$
\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right)\left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right) = \left(\begin{array}{c|c} I + IV - V + VII & III + V \\ \hline II + IV & I - II + III + VI \end{array}\right)
$$

Figure 4.5: Strassen's matrix multiplication strategy [72].

the sparse case, i.e. when a tree node has mostly `null` pointers. In practice this arises often, due to the sparse nature of most input instances. A 4-level chain requires 128 B in the quadtree case (4 256-bit nodes), and 2 KB in the 256-tree case. The tradeoff between node-count and node-size is visualised in Figure 4.4.

We can combine this squashing technique with the leaf bitmaps presented previously, to negate the costs of long leaf-chains. The size of the bitmap is significantly increased, such that a 64-tree and 256-tree leaf bitmaps encode 64×64 and 128×128 matrices respectively.

4.3.1.4 Strassen's Multiplication

Section 4.3.1 presented binary matrix multiplication as a vehicle for relational composition, and used the standard recursive-submatrix multiplication. Instead of this cubic technique, an improved multiplication algorithm, such as Strassen's algorithm [72], could be used. Figure 4.5 shows Strassen's technique for matrix multiplication, which performs fewer recursive multiplications than the recursive procedure of Equation 4.1, leading to a better time-complexity of $\mathcal{O}\left(n^{\log_2 7} \approx n^{2.81}\right)$, and is particularly suited to quadtrees, since it uses the four sub-matrices to compute the coefficient terms. More complex techniques [23] are impractical both for quadtrees and multiplications in general.

In connection with combinatorial multiplication techniques, we have performed a numerical analysis of quadtrees and the effects of Strassen's multiplication. Firstly, quadtrees encode zero-quadrants, which immediately allows some recursive submultiplications to be avoided, i.e. if A_{11} is the zero matrix, the recursive procedure can avoid two multiplications and two unions, whereas Strassen's algorithm can only avoid two additions (I and V), a subtraction (IV) and a multiplication(III). Enumerating all combinations, only when neither A nor B has a zero-quadrant does Strassen's algorithm perform fewer multiplications.

We can characterise the saturation of quadtree quadrants by enumerating all binary matrices. Each internal (i.e. non-leaf) quadtree node has a given number of children i between 0 and 4. Counting the number of such nodes in a 5-tuple, the Quadtree in Figure 4.1 has counts $\langle 0, 1, 1, 1, 1 \rangle$, since it has four internal nodes with one, two, three and four children in them. Let the pairwise sum

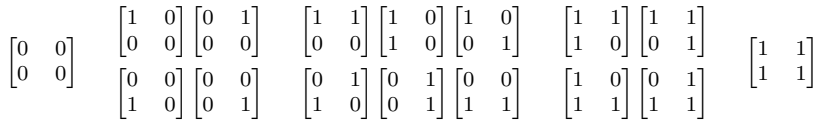


Figure 4.6: Representations of all 2×2 matrices, grouped according to their configuration count.

for *all possible* matrices of height i be its **configuration** c_i. For example, all matrices with a 1-high quadtree are depicted in Figure 4.6, so $c_1 = \langle 1, 4, 6, 4, 1 \rangle$. Intuitively, configurations relate to the binomial coefficients, since an internal node chosen at random will have $\binom{4}{c}$ ways of configuring c child nodes. Let $h_i = 2^{2^{2i-2}} - 1$ record both the number of non-zero matrices with side-length 2^{i-1} and the number of non-zero quadtrees of height i. It can be shown via recurrence that the configurations for height i are:

$$c_0 = \langle 1, 0, 0, 0, 0 \rangle$$
$$c_i = \langle 1, 4h_i, 6h_i^2, 4h_i^3, h_i^4 \rangle + (4 + 12h_i + 12h_i^2 + 4h_i^3)(c_i - c_0)$$

This number allows us to reason about matrix multiplication. In large random matrices, the h_i^4 term grows significantly larger than any other term. Since the number of 4-child nodes dominates any other kind of node, we expect Strassen's multiplication to perform fewer multiplications. For our purposes, unfortunately, the input graphs are typically *sparse*, which implies Strassen's approach may slow down multiplication.

The input problems we use in our evaluation, far from being unknown, are mostly sparse. Unfortunately, Strassen's multiplication is pathologically bad in the case of sparse matrices. Consider the VI submatrix in Strassen's formulation, $(-A_{11} + A_{21})(B_{11} + B_{12})$, which potentially doubles the density of its operands. Though there are fewer multiplications at this stage of the recursion, lower stages will have more saturated nodes (due to their increased density), and recursively more multiplications will be required. Experimental evaluation of the breadth of the recursion reveals that Strassen's algorithm calls its multiplication subroutine almost 1000x more often.

Nonetheless, Strassen's multiplication motivates the use of quadtrees and algebraic composition, as they alone provide theoretical advantages in the dense case. Our general approach to CFL-R can capitalise on dense cases (when they do occur), since Strassen's multiplication presents a viable strongly sub-cubic algorithm. Our performance evaluation in Section 4.4 involves sparse problems (since these occur in practice), hence Strassen's multiplication is not used.

4.3.2 Neighbourhood Maps

An alternative to quadtrees is the neighbourhood map. A relation $\mathbf{R}$ can be interpreted as a directed graph, having the domain of $\mathbf{R}$ as its vertex set and the relation itself as its edge set (i.e. a subgraph of the original input graph of edges labelled with R). Then the relational ADT can be implemented by introducing two neighbourhood mapping functions $N^+_{\mathbf{R}} : V \mapsto 2^V$ and $N^-_{\mathbf{R}} : V \mapsto 2^V$, defined as $N^+_{\mathbf{R}}(u) = \{v \mid (u,v) \in \mathbf{R}\}$ and $N^-_{\mathbf{R}}(v) = \{u \mid (u,v) \in \mathbf{R}\}$, respectively. For the problem in Figure 1.2, relations and labels are interchangeable concepts, so for the relation *assign* we have: $N^+_{assign} = \{this \mapsto \{a, b\}, other \mapsto \{th5, th6\}, a1 \mapsto \{other\}, b2 \mapsto \{other\}\}$.

Mapping function $N^+_{\mathbf{R}}(v)$ reproduces the set of R-labelled successors of v, and $N^-_{\mathbf{R}}(v)$ reproduces the R-labelled predecessors of v. Thus, either neighbourhood can substitute the relation itself from the identity: $(u,v) \in \mathbf{R} \Leftrightarrow v \in N^+_{\mathbf{R}}(u) \Leftrightarrow u \in N^-_{\mathbf{R}}(v)$. The relational operations (union, set-difference, and composition) are defined for neighbourhood maps by expanding the previous identity. By way of demonstration, composition is formulated as:

$$\begin{aligned}\mathbf{R} \circ \mathbf{S} &= \{(u,v) \mid \exists w \ s.t. \ (u,w) \in \mathbf{R} \wedge (w,v) \in \mathbf{S}\} \\ &= \{(u,v) \mid w \in V, u \in N^-_{\mathbf{R}}(w), v \in N^+_{\mathbf{S}}(w)\} \\ &= \bigcup_{w \in V} N^-_{\mathbf{R}}(w) \times N^+_{\mathbf{S}}(w)\end{aligned}$$

Then we see the composed-neighbourhood follows from:

$$v \in N^+_{\mathbf{R}\circ\mathbf{S}}(u) \Leftrightarrow (u,v) \in \mathbf{R} \circ \mathbf{S} \Leftrightarrow (u,v) \in \bigcup_{w \in V} N^-_{\mathbf{R}}(w) \times N^+_{\mathbf{S}}(w) \Leftrightarrow v \in \bigcup_{\{w \mid u \in N^-_{\mathbf{R}}(w)\}} N^+_{\mathbf{S}}(w)$$

Thus composition of the neighbourhood functions is defined as:

$$N^+_{\mathbf{R}\circ\mathbf{S}} = \{u \mapsto \bigcup_{\{w \mid u \in N^-_{\mathbf{R}}(w)\}} N^+_{\mathbf{S}}(w) \mid u \in V\}$$

In Figure 1.2, since $H1 \in N^+_{alloc}(a)$ and $this \in N^-_{assign}(a)$, we have $H1 \in N^+_{assign \circ alloc}(this)$, as expected. The predecessors are defined symmetrically.

From the above explanation, we consider the implementation of a neighbourhood map. The map supports queries for the neighbourhood set of a given vertex, and can therefore be implemented as a lookup table. Due to the sparse nature of the input problems, the lookup table only contains key entries for non-empty neighbourhoods, in which case the map is an index of the non-empty neighbourhoods. The neighbourhood itself is a set of vertices, and must support the operations of union, intersection, set-difference and cross-product. Once again we desire a sparse representation, and it is convenient to re-use the kind of index that mapped the non-empty neighbourhoods to the set of neighbours. In this way, the lookup table may be implemented by, for example, a hash-table which maps vertices to hash-sets of vertices.

The complexity of neighbourhood map operations is described in terms of the lookup tables which implement them. Lookup tables support the operations of element-lookup, addition and deletion, which can be treated as parameters. The above formulation of composition calls for a lookup of the successor and predecessor neighbourhood n times, and adds at most n^2 new edges to the result each time, thus the parametric complexity is $\mathcal{O}\left(nLU + n^3 ADD\right)$. Allowing for only m pairs in the relations, we can also derive an **average-case** complexity, namely $\mathcal{O}\left(nLU + m^2 ADD\right)$. Set-difference and union operations require $\mathcal{O}\left(nLU + mDEL\right)$ and $\mathcal{O}\left(nLU + mADD\right)$ time respectively, via a similar argument.

To de-parameterise the neighbourhood map's complexity, we fix an implementation of the lookup table. Tables implemented in balanced trees (the red-black tree and B-tree in our evaluation) have height at most $\log n$, and thus perform additions, deletions, and lookups in $\mathcal{O}\left(\log n\right)$ time. Therefore, the average case analysis yields time complexities of $\mathcal{O}\left(m^2 \log n + n \log n\right)$ for composition, and $\mathcal{O}\left(m \log n + n \log n\right)$ for union and difference.

4.4 Experimental Results

4.4.1 Methodology

We implement multiple data-structures for use in our semi-naïve algorithm. These are used as a comparison to the standard CFL-R algorithm, due to Melski and Reps [47], called the worklist (`WL`) method.

- Red-Black-Tree (`RB`) mappings of binary-relations, due to the C++ STL's `std::map` implementation.
- Hash-Maps (`UM`) of binary relations, due to C++11's `std::unordered_map`.
- B-Trees (`BT`) of binary relations, using Google's [2] implementation.
- Quadtrees (`QT`), as described in Section 4.3.1 and implemented in-heap with pointers.
- Concise Quadtrees (`CQ`), implements the concise representation for quadtrees (Section 4.3.1.1), with improved cache locality at the cost of additional copying.

Our evaluation focuses on two CFL-R use-cases. Primarily, we examine field-sensitive context-insensitive points-to analysis for Java programs, due to Sridharan et al [70], with standard modifications to optimise evaluation. The modified grammar is presented in the appendices. This grammar is parameterised by object fields and class types. Also included is the taint-analysis for Android programs due to Bastani et al. [10].

[2] `https://code.google.com/p/cpp-btree/`

Table 4.1: Statistical information on the DaCapo and Taint benchmarks. E_S is the number of solution-labelled edges, BNF shows the number of labels after converting the grammar to binary normal-form, and $|Quad|$ shows the total number of nodes in the quadtrees for the `QT` and `CQ` implementations.

Benchmark	$\|V\|$	$\|E\|$	$\|E_S\|$	$\|\mathcal{T} \cup \mathcal{N}\|$	BNF	$\|Quad\|$
24	4,063	39,303	17,201	407	407	115,141
54	8,297	167,080	38,965	1,187	1,187	482,070
lui	22,699	44,377	12,288	1,653	2,301	213,084
lus	22,699	44,377	12,288	1,653	2,301	213,084
pmd	32,295	166,832	34,402	1,755	2,399	541,398
ant	32,927	76,919	20,898	1,095	1,520	332,454
ecl	33,912	61,725	17,119	2,257	3,172	310,201
blo	40,989	117,894	57,829	1,900	2,650	464,537
xal	46,780	111,885	42,461	2,449	3,436	500,399
cha	49,893	113,332	39,753	2,914	4,166	520,986
fop	53,851	111,690	38,802	3,495	4,742	537,345
hsq	63,281	317,361	200,762	2,817	3,974	1,122,353
jyt	78,639	524,547	383,917	3,351	4,588	1,755,173

Points-to benchmarks are drawn from the DaCapo [13] suite, with program facts generated by the DOOP [15] static-analysis framework. The taint-analysis problems are provided directly by the authors, and include the necessary grammar file. Input data for the taint-analysis benchmarks are anonymised: we refer to them as 24 and 54.

Experiments are run on a 32-core 2.1GHz Intel®Xeon E5-2450 server, with 132GB of RAM, running Fedora 18. We compile with GCC 4.7.2 using the `-O3` optimisation flag.

4.4.2 Benchmarks

Table 4.1 records problem-specific characteristics of the benchmarks we used. We order the benchmarks by the size of their vertex set, which is n for our complexity arguments. Note that the taint-analysis benchmarks are much smaller than the DaCapo problems in terms of vertex-set, but have a dense edge-set. Despite this, all benchmarks are reasonably sparse, with the edge set being roughly 2x-20x larger than the vertex set. Converting the input problem to binary normal-form increases the number of labels by up to 50%; the taint-analysis problems are distributed already-normalised.

4.4.3 Normalisation

Figure 4.7 shows the effect of grammar normalisation on runtime and memory-size. We compare the `WL` version, which is typically presented for normalised grammars, with a subset of the semi-naïve implementations which do not re-

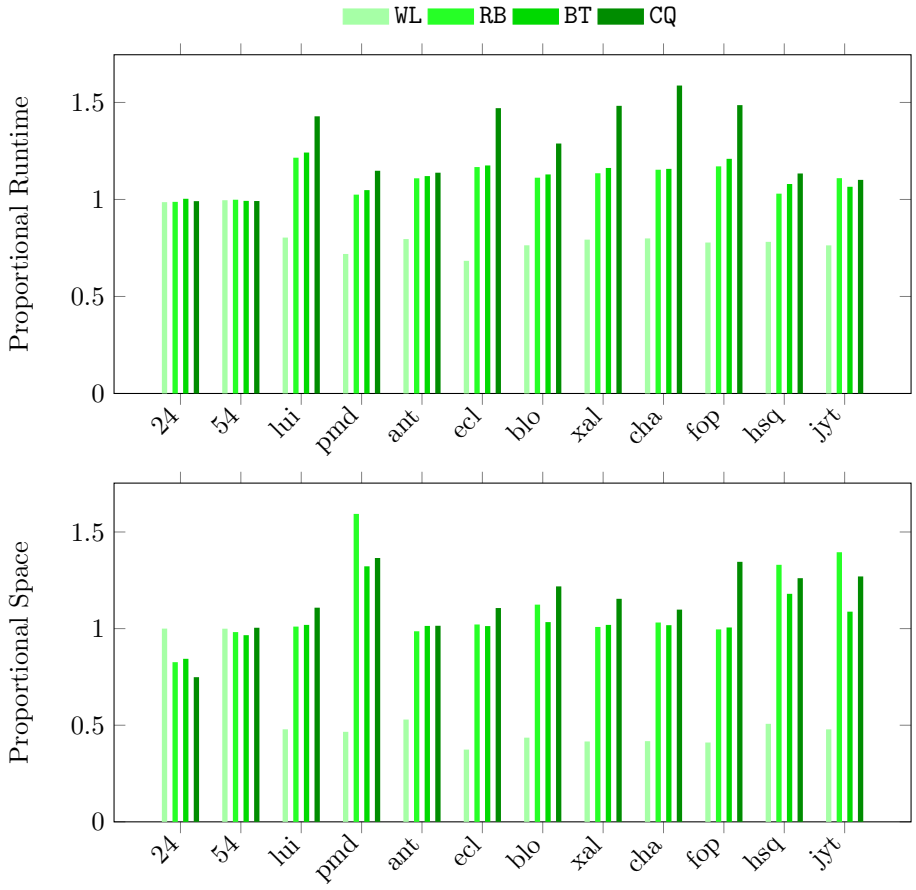


Figure 4.7: The effect of grammar normalisation on execution-time and memory-requirement, $y = \frac{normalised}{unmodified}$, for each benchmark.

quire normalisation. As expected, the pre-normalised taint-analysis problems show little difference. The findings validate our claim: the overhead of tracking and storing additional information for the labels created during normalisation increases runtime and memory footprint for the semi-naïve strategy. We also verify the known `WL` limitation, namely that normalisation is necessary for improved memory and time performance.

Henceforth, comparisons of implementations will be of no-normal-form versions of semi-naïve problems normalised against the binary normal-form version of the worklist algorithm.

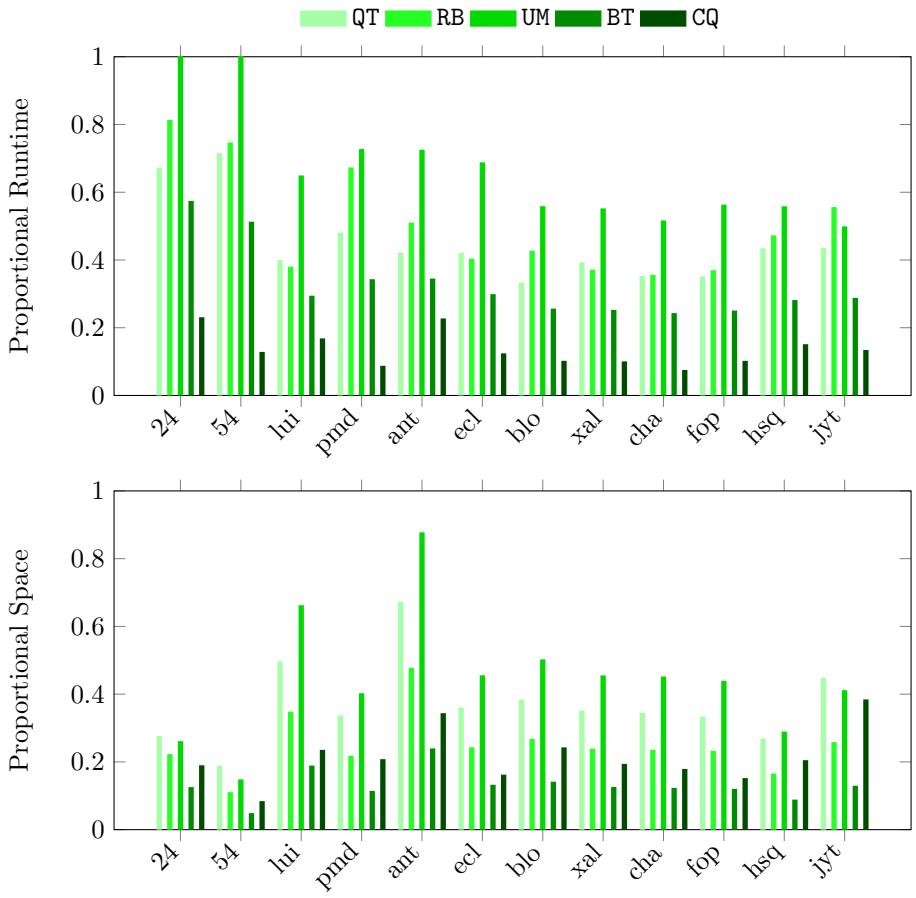


Figure 4.8: Proportional runtime and space requirements as a fraction of the normalised-worklist runtimes.

4.4.4 Performance

We show the comparative runtime performance of the implementations, as compared with the worklist method, in Figure 4.8. Firstly, we see how hash-maps are unsuitable for this kind of problem, due to uncacheable behaviour; in our taint benchmarks we even see runtime slowdown using this method. The `QT` implementation halves the runtime on average, however this can be improved upon by using a more cache aware B-tree data-structure. The clear winner, in terms of speed, is the concise quadtree implementation `CQ`, which outperforms B-Trees, and achieves up-to $\frac{1}{10}$th the runtime of `WL`.

Further, we see the superior scaling behaviour of the semi-naïve implementation. Note that the least speedups are for smaller benchmarks, and the comparative advantage grows with problem-size for most implementations.

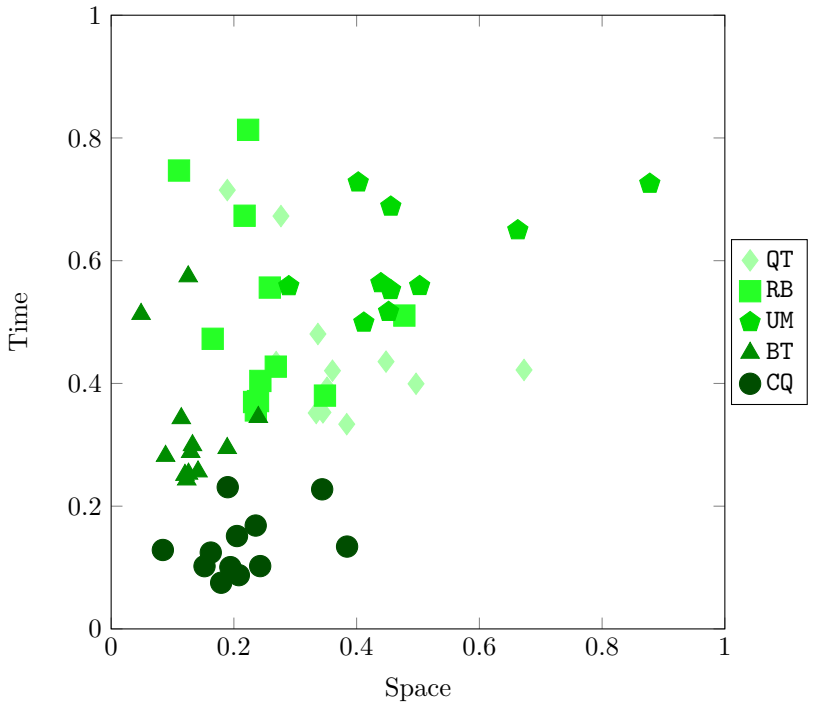


Figure 4.9: Memory/Runtime tradeoff exhibited by the different data-structures for the semi-naïve implementation.

Figure 4.8 also plots the relative size-demands of the implementations against the binary normal-form `WL` implementation. Primarily we do not see the same kind of scaling behaviour as we did with time requirements, so whilst memory utilisation is better with the semi-naïve method, only a constant improvement can be expected. Again, hash maps are usually a bad choice for the relational data-structure, with as little as 12% shrinkage. We observe that the `BT` implementation is the universally superior option, on average 14% of `WL`'s size, compared with the `CQ`'s 22%.

Given that the `CQ` tree implementation has superior runtime, whilst the `BT` binary relations have smaller memory demands, we wish to understand how the choice of data-structure imposes a space-time tradeoff. Figure 4.9 plots the relative improvement for runtime and size. Here we assume equal importance for space and time considerations; different priorities may influence implementation choice. We see the `CQ` cluster has superior overall performance, since it is closer to the origin. Whilst the `BT` implementation uses less memory, the cost to runtime is significant in comparison.

Chapter 5

Declarative Optimisation

This chapter examines the efficiency and performance aspects of CFL-R logic *itself*, as opposed to the evaluation of that logic. We develop a novel feedback-directed optimisation approach for improving the performance of CFL-R solvers by transforming their underlying logic. A version of this work is under review for TACO, and is titled "Feedback-Directed Optimisations for Context-Free Language Reachability" [37].

5.1 Declarative Languages

CFL-R has a reduced Datalog semantics [1], where the rules are limited to a chain format over binary relations, i.e., CFL-R solves logic problems where all clauses are of the form:

$$H(v_0, v_k) \text{ :- } B_1(v_0, v_1), B_2(v_1, v_2), \ldots, B_k(v_{k-1}, v_k).$$

Whilst we can improve the efficiency of CFL-R solvers by adapting more effective evaluation strategies to the problem, these techniques treat the *logic* of the problem as a black box. In fact, as we encountered in the high-efficiency points-to analysis from Chapter 3, performance can be greatly improved by examining and modifying the logic that underpins our analysis. In that work, the points-to grammar was converted into an equivalent form:

$$\textit{points-to} \rightarrow (\textit{assign} \mid \textit{bridge})^* \textit{alloc}$$

This implies a transformation of the underlying logic, which in this case was needed to expose regularity in the context-free grammar. We now explore the performance implications of **logic transformations** for the CFL-R problem.

5.1.1 Hints

The strength of declarative approaches is that they allow programmers to focus on the logic of solutions, and ignore the implementation details of evaluating

it. To this end, Chapter 4 discusses the tradeoffs associated with evaluating a given CFL-R logic. But the issue is made more complicated by the fact that there can be alternative *equivalent* formulations of the same logic. Consider the two CFL-R grammars below:

$$S \to \varepsilon \mid a\ S\ b \tag{5.1}$$

$$S \to \varepsilon \mid a\ R \qquad R \to S\ b \tag{5.2}$$

Both grammars encode the canonical context-free language $\{a^n b^n \mid n \geq 0\}$, i.e. strings of some number of "a"s followed by the same number of "b"s. If a system were truly declarative, then two logically equivalent (but non-identical) formulations should imply the same evaluation strategy. In practice, though, solvers for declarative problems like this tend to interpret the logic more literally. It is (currently) unreasonable to expect that a system could recognise the underlying logic, and always adopt the most efficient solver for that logic, as this would require human levels of intelligence. Instead, we can conceive of *normal-forms*, and then say that both grammars yield the same normal form (and therefore can be recognised as the same), though for arbitrary grammars even this may not be the case. For the above case, Grammar 5.1 is in *reduced normal-form*, which 5.2 can be converted to. Alternatively if we choose *binary normal-form*, which 5.2 is in, then 5.1 may or may not be converted to the same grammar (allowing for renaming nonterminals) depending on how we split the $a\ S\ b$ term. More importantly, though, it might be (operationally) useful not to treat these two grammars as the same.

Consider the problem of searching for CFL-R paths according to the above grammars. If we know nothing about the structure of the graph, then we can say nothing about the best order of operations, and should simply choose a good algorithm (such as the semi-naïve approach). On the other hand, if we knew the graph was **biased** in some way, such as by having significantly more a-labelled edges than b, we can make stronger statements about the evaluation strategy. Assume that a new S edge is discovered and we are trying to see if another S path can be built using the new edge as the middle. It is smarter to look at b first, as doing so is more likely to avoid needless searching; there are fewer b-edges, so if there is no new path that is more likely because it does not join to a b than that it does not join to an a. The semi-naïve evaluation strategy does not consider optimisations such as this. Indeed, the notion that "it is better to prioritise certain labels for evaluation" falls outside the declarative scope of CFL-R. Nonetheless, biases in the inputs are common in real-world problems, and it is very useful for declarative solvers to be able to reason about such *domain knowledge*.

For this reason, many declarative systems allow users to encode domain knowledge via implicit or explicit constructs. Unlike typical declarative directives, such constructs specify implementation details and do not affect the logic of the problem. We call these constructs **hints**. Hints allow specification writers to manually intervene, possibly because the declarative solver in question is unable to reason strongly enough about a good evaluation strategy. In SQL,

join-hints are frequently used to coerce the RDBMS to use the most efficient join strategy [73]. In this case, the declarative system was usually too slow (or outright incorrect) when deciding how best to join relations, so the programmer explicitly told it how. Sometimes hints are implicit, and are more like artefacts of the declarative system's translation/code generation mechanism. In LogicBlox (3.9.0), a declared relation `A(x,y,z)` will be indexed from right to left [5], so a join like `P(i,j),Q(k,j)` makes efficient use of the index, while `N(r,s),M(r,t)` is inefficient. In Soufflé, a declared rule will be examined in order unless a specific order is given [40], so a rule `A(x) :- B(x), C(x)` will look at the `B` relation before the `C` relation. The above hints can be used strategically by specification writers to encode domain knowledge, and improve the runtime of declarative systems.

Whilst hints are necessary to achieve high performance systems, unfortunately they require significant manual intervention. In the case of explicit directives, such as SQL's join-hints, manual intervention is reasonable, since the hints are designed to overcome limitations in the automatic procedures. On the other hand, much manual effort can be alleviated by adopting more effective automated techniques. In the case of Soufflé or LogicBlox's hints, the problem is not that an automatic approach falls short, but that it lacks the kind of information needed to make good decisions. Returning to the $\{a^n b^n \mid n \geq 0\}$ example, we can conceive of an implicit CFL-R hint; Grammar 5.2 implies that it is better to find $S\ b$ sub-paths first (calling these R) before looking for the full $a\ S\ b$ paths (now in the form $a\ R$), whereas Grammar 5.1 does not imply that looking for $S\ b$ first is superior to looking for $a\ S$ first. Tying CFL-R's operational semantics to the literal expression of the grammar in this way allows us to give hints to CFL-R solvers. Thus, if the search for $S\ b$ is operationally useful, possibly because b-edges are rare and looking for those sooner cuts down on many wasted searches, then Grammar 5.2 may yield more effective solvers.

The problem of optimising CFL-R can now be more properly addressed; there are two important components:

- The hints which are given to CFL-R solvers. We specifically want to know how programmers can influence the execution semantics of their CFL-R solvers, either by explicit or implicit modifications to the logic of the input specification. Knowing this, we can **mechanically apply** hints to a specification, provided we can verify that such a hint will improve performance.

- The circumstances which make a hint useful. As we saw, coercing the execution in a certain way is motivated by domain knowledge for the given inputs (or classes of inputs), which means that one evaluation strategy may be more or less effective than another. We examine **execution feedback** to discover biases in the inputs, and choose from a collection of known hints in order to derive a suitable optimisation.

For the purpose of our discussion, an *optimisation* is a hint which is automatically discovered and mechanically applied.

5.2 Executing CFL-R

In order to reason about inefficiencies in the *logic* of CFL-R, we look at the evaluation of CFL-R logic, based on the algorithms for it. The underlying nature of the CFL-R problem, which unsurprisingly can be phrased as a matrix-based reachability problem (Section 5.2.2), is exposed, which allows us to precisely quantify the problem. A direct byproduct of this quantification is a method of counting waste, which occurs when **dead-ends** are discovered during the search for CFL-R paths. Our modelling of waste gives us *a priori* knowledge of inefficient formulations, and therefore forms the basis from which we optimise the CFL-R logic.

5.2.1 CFL-R Evaluation

The Melski-Reps approach is detailed in Algorithm 2. There are several important factors in this evaluation:

- Grammar rules must be in binary normal-form (BNF). A grammar in BNF has at most two terms on the right-hand-side of its rule bodies. It should be noted that any CFL-R grammar can be converted to BNF with only a linear increase in the size of the grammar [43]. Clearly the way that rules are broken up into BNF will influence the evaluation strategy, i.e the Melski-Reps algorithm looses the domain knowledge about which BNF rules came from which original grammar rules.
- Evaluation is driven by the worklist. The worklist functions much like the delta-sets of the semi naïve strategy, in that new paths are only discovered as a result of walking over recently found summary paths.

These two factors influence how the logic of a given CFL-R problem can influence performance. Firstly, given that the presence of paths in the input graph will not change depending on how long rules are broken up into their BNF form, we can influence *how* the paths are found by breaking the rules up in a strategic way. Assuming there are better or worse ways to find a given path, then a logical optimisation would take the form of determining the best means of searching for a path. Further, since path discovery is driven by the worklist, we can also assume that different logic will prompt different evaluation orders, which in turn causes worklist items to be discovered (i.e. queued) in a more favourable order. Chaudhuri's approach differs only slightly from the Melski-Reps algorithm. For this reason, the same kinds of modifications to the logic that would optimise a worklist evaluation should also be useful for a Chaudhuri-style solver.

5.2.2 Matrix-based Solvers

To begin our presentation of a solver/algorithm agnostic understanding of the costs associated with CFL-R logic, we phrase CFL-R via its underlying connection with matrix multiplication. The connection between graph reachability

Algorithm 6 A matrix-multiplication-based algorithm for solving CFL-R.

```
1: procedure MATRIX-CFLR(𝒯, 𝒩, 𝒫, V, E)
2:     for all L ∈ 𝒯 ∪ 𝒩 do
3:         L ← ∅
4:         for all L(u, v) ∈ E do
5:             L^{u,v} ← 1
6:     for all A → ε ∈ 𝒫 do
7:         A ← A ∪ I
8:     while any matrix is changing do
9:         for all H → B_1 ... B_k ∈ 𝒫 do
10:            H ← H ∪ ⨉_{i∈[1,k]} B_i
```

and matrix multiplication is well-established in the research literature [50], for completeness we reproduce the relevant results here.

Definition 14. *Given a graph $G = (V, E)$, and some mapping $V \mapsto \mathbb{Z}$, then the **adjacency matrix** $\mathbf{G} \in [0,1]^{|V|\times|V|}$ is the boolean matrix having:*

$$\mathbf{G}^{i,j} = \begin{cases} 1 & (i,j) \in E \\ 0 & \textit{otherwise} \end{cases}$$

Theorem 1. *There is a k length path in G between i and j if and only if cell i, j is 1 in $\mathbf{G}^k$*

Proof. See, for example Warshall's work [81]. □

There are several useful corollaries that follow immediately from this result:

- $(\mathbf{G} \cup \mathbf{I})^k = \bigcup_{i\in[0,k]} \mathbf{G^i}$ are the k-or-shorter paths.
- $(\mathbf{G} \cup \mathbf{I})^\infty$ is the reflexive transitive closure.

Importantly, though, we can apply these results to compute the solution to CFL-R problems. Using boolean matrix multiplication and addition, we can phrase the CFL-R problem with Algorithm 6. Much like the other general-purpose CFL-R algorithms (such as Algorithm 2), we begin by accounting for terminal labels and epsilon edges, and subsequently discover paths by concatenating known edges in sequence. Here matrix multiplication forms the vehicle for "concatenation", as opposed to the notion of a path-walk. Whilst this approach is too inefficient to be used practically, it gives us insights into where CFL-R formulations are themselves inefficient (as opposed to there being inefficiencies in their solvers), by looking at the underlying reachability problem.

Firstly, the algorithm is indeed sound and precise with respect to the CFL-R definition. Every epsilon-path that would form a solution to CFL-R(L, G) will be found by the algorithm:

Lemma 11. $\forall N \in \mathcal{N} : N \overset{*}{\Rightarrow} \varepsilon$ *implies that* $\mathbf{I} \subseteq \mathbf{N}$.

Proof. If ε is in the language with start symbol N, then there are two cases:

- $N \to \varepsilon \in \mathcal{P}$, then $\mathbf{I} \subseteq \mathbf{N}$ follows trivially from line 7.
- $N \Rightarrow B_1 \dots B_k \overset{*}{\Rightarrow} \varepsilon$, but then $\forall i \in [1,k] : B_i \overset{*}{\Rightarrow} \varepsilon$ is also true and has strictly fewer derivations, so induct using the above as a base case.

□

In a similar manner we also discover all path-words that appear in the solution:

Lemma 12. $\forall N \in \mathcal{N} : \langle T_1(v_0,v_1), \dots, T_k(v_{k-1},v_k)\rangle \in \Pi \wedge N \overset{*}{\Rightarrow} T_1 \dots T_k$ *implies that* $\mathbf{N}^{v_0,v_k} = 1$.

Proof. Firstly $\forall T \in \mathcal{T} : T(u,v) \in E \Rightarrow \mathbf{T}^{u,v} = 1$. Then we have two cases:

- $N \to T_1 \dots T_k \in \mathcal{P}$, then apply Theorem 1 to line 10.
- $N \Rightarrow B_1 \dots B_j \overset{*}{\Rightarrow} T_1 \dots T_k$, so partition $T_1 \dots T_k$ into (possibly empty) substrings having $\forall i \in [1,j] : B_i \overset{*}{\Rightarrow} T_{i_l} \dots T_{i_h} \vee B_i \overset{*}{\Rightarrow} \varepsilon$. In the epsilon case, lemma 11 holds. Otherwise $B_i \overset{*}{\Rightarrow} T_{i_l} \dots T_{i_h}$ has strictly fewer derivations. We can induct, using the above as the base case, to show $\mathbf{B_i}^{v_{i_l-1},v_{i_h}} = 1$, and apply Theorem 1 to line 10.

□

Importantly, the matrix-based formulation necessarily finds only those pairs which do imply a reachability path spelling the word in that language:

Lemma 13. $\forall N \in \mathcal{N} : \mathbf{N}^{p,q} = 1$ *implies that* $(p,q) \in \text{CFL-R}((\mathcal{T},\mathcal{N},\mathcal{P},N),G)$.

Proof. There are three places where a cell can be set to 1:

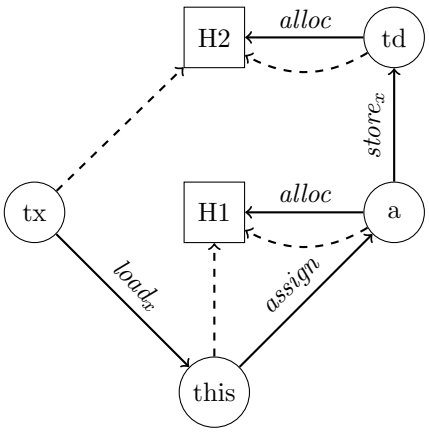
- Line 5, then $N(p,q) \in E$, so $N \in \mathcal{T} \Rightarrow N \notin \mathcal{N}$, this case is irrelevant.
- Line 7, then $N \to \varepsilon \in \mathcal{P}$, implying $(p,q) \in \{(v,v) \mid v \in V, \varepsilon \in (\mathcal{T},\mathcal{N},\mathcal{P},N)\}$
- Line 10, then $N \to B_1 \dots B_k \in \mathcal{P}$ and $\exists v_1, v_2 \dots v_{k-1}$ such that $\mathbf{B_1}^{p,v_1} = \mathbf{B_2}^{v_1,v_2} = \dots = \mathbf{B_k}^{v_{k-1},q} = 1$. Using the previous two cases as the base-case, inductively assume that $\forall i \in [1,k] : \mathbf{B_i}^{v_{i-1},v_i} = 1 \Rightarrow (v_{i-1},v_i) \in \text{CFL-R}((\mathcal{T},\mathcal{N},\mathcal{P},B_i),G)$. Let $\pi = \langle T_1(p,v_{1_1}), \dots, T_l(v_{k-1_m},q)\rangle$ be the path formed by concatenating all the sub-paths, therefore $N \Rightarrow B_1 \dots B_k \overset{*}{\Rightarrow} T_1 \dots T_l$ and $\pi \in \Pi$, hence $(p,q) \in \text{CFL-R}((\mathcal{T},\mathcal{N},\mathcal{P},N),G)$.

□

Hence, the algorithm is sound and precise:

Lemma 14. *Algorithm 6 computes* $\text{CFL-R}(\mathcal{L},G)$.

Proof. The algorithm is sound from Lemmas 11 and 12. The algorithm is precise from Lemma 13. □



(a) The graph, dashed lines indicate *points-to*-reachability.

$$\textit{points-to} \rightarrow \textit{alloc} \tag{5.3}$$
$$\textit{points-to} \rightarrow \textit{assign points-to} \tag{5.4}$$
$$\textit{points-to} \rightarrow \textit{load}_x \ \textit{points-to} \ \overline{\textit{points-to}} \ \textit{store}_x \ \textit{points-to} \tag{5.5}$$

(b) The grammar, only the x field is needed for this subset.

Figure 5.1: A subset of the points-to problem from Figure 1.2

.

The most important takeaway from the matrix-based CFL-R algorithm is an understanding that matrix multiplication underpins the evaluation of CFL-R logic. In the actual algorithms for CFL-R the multiplication is substituted for a similar vehicle, such as relational joins in the semi-naïve formulation, or an edge search in the Melski-Reps algorithm (which is essentially the witness problem for row vectors). The assumption, therefore, is that if we can show an inefficiency in the logical structure of a given CFL-R formulation, then this inefficiency will be present no matter how the problem is solved. Hence, we propose *logical* optimisations, which are agnostic with respect to the solving algorithm. Of course, such optimisation efforts are meaningless unless the solver treats the input specification at least partially imperatively, since a truly declarative system can not be perturbed by modifying the input specification in a logically equivalent manner. As we discussed in Section 5.1.1, this is a reasonable assumption to make for most systems.

5.2.3 Wasted Searches

The most important logical inefficiency that this work explores is based on the execution of a *wasted search*. In the matrix-formulation from Algorithm 6, the search occurs on line 10, i.e. in the expansion of a rule $H \rightarrow B_1 \dots B_k$. This is, in essence, a chain matrix multiplication (CMM), which implies that performing the multiplication left-to-right is not the most efficient approach [39]. Consider the CFL-R problem in Figure 5.1, a subset of the running example from Figure 1.2. When evaluating Rule 5.5, we must choose an efficient bracketing

with which to evaluate the CMM. Unfortunately, the dynamic programming approach used to determine good CMM strategies assumes that the matrices have varying sizes. On the other hand, the adjacency matrices of Figure 5.1a are all 6×6, so no useful strategy can be determined in this way. Nonetheless, there are better and worse ways to evaluate the CMM, assume that a bracketing was used such as the following:

$$\left(\mathit{load}_x(\mathit{points\text{-}to}\ \overline{\mathit{points\text{-}to}})\right)\ (\mathit{store}_x\ \mathit{points\text{-}to})$$

The computation of $\mathit{points\text{-}to}\ \overline{\mathit{points\text{-}to}}$ generates a relatively dense intermediate result, for Figure 5.1a there are 8 nonzero cells:

$$(td, td), (a, a), (this, this), (tx, tx), (td, tx), (tx, td), (a, this), (this, a).$$

By comparison, the $\mathit{store}_x\ \mathit{points\text{-}to}$ term yields an intermediate result with only a single nonzero cell, $(a, H1)$. Since the CFL-R solver must search for paths where the concatenation of labels forms a word in the language, these nonzero intermediate cells correspond to valid sub-paths in the search. The implication, then, is that if the path search is evaluated in any way like the above CMM, then it will (needlessly) discover many intermediate paths. An alternative bracketing for the CMM is:

$$\left((\mathit{load}_x\ \mathit{points\text{-}to})(\overline{\mathit{points\text{-}to}}\ \mathit{store}_x)\right)\ \mathit{points\text{-}to}$$

With this association, every intermediate multiplication yields a sub-matrix with exactly one nonzero element. Thus, assuming some correspondence between the CMM and the actual evaluation, the latter bracketing strategy should produce a more efficient solver, since it does not discover as many intermediate sub-paths.

We can also look closely at the actual multiplications to discover inefficiencies. Consider the $\mathit{load}_x\ \mathit{points\text{-}to}$ term. At the point in the evaluation where this term becomes relevant, there are three nonzero cells in *points-to* and one in load_x. If the solver worked with dense multiplications (aside from being very inefficient anyway) it would not matter whether we looked at *points-to* or load_x first. On the other hand, if the solver in any way accounted for sparsity in the graph, then it would matter how many nonzero cells there were. The solver could start with load_x and try to connect it to *points-to*, which means looking once for a witness from amongst three edges. Alternatively, the solver could start with *points-to* and try to connect it to load_x, which means looking three times for a witness from amongst one edge. These two approaches, which we shall say differ in their **directionality**, would naturally have different execution costs. When discussing wasted effort, since only one intermediate $\mathit{load}_x\ \mathit{points\text{-}to}$ path exists, we assume that starting with *points-to* incurs one successful search and two wasted searches, whilst starting with load_x has only the successful search.

Therefore, it is not enough to simply consider a CMM bracketing when understanding the logical cost of a path search. Instead, we adopt the more general notion of a **search order**, which can be any permutation of the terms in the CMM.

Definition 15. *Given a rule with body terms $B_1 \dots B_k$, the* ***search order*** *σ is an ordered permutation of the body terms:*

$$\sigma = \langle B_{\sigma_1}, \dots, B_{\sigma_k} \rangle \in \mathfrak{S}(\{B_1, \dots, B_k\})$$

To understand how a search order relates to the CMM bracketing, consider how we would convert the former into the latter. Consider a search order $\langle \mathit{load}_x, \mathit{store}_x, \overline{\mathit{points\text{-}to}}, \mathit{points\text{-}to}_1, \mathit{points\text{-}to}_2 \rangle$ for Rule 5.5, where $\mathit{points\text{-}to}_i$ is the ith occurrence of *points-to* in the rule. Dequeue relations from the search in order, and greedily multiply them where they *should join* according to the rule:

- load_x is dequeued, there is nothing to join it to.
- store_x is dequeued, it does not join with the current pool containing only load_x.
- $\overline{\mathit{points\text{-}to}}$ is dequeued, and makes the first subterm $(\overline{\mathit{points\text{-}to}}\ \mathit{store}_x)$, load_x remains in the pool.
- $\mathit{points\text{-}to}_1$ is dequeued, joins with load_x, then the resulting subterm is joined with the previous $(\overline{\mathit{points\text{-}to}}\ \mathit{store}_x)$ subterm.
- $\mathit{points\text{-}to}_2$ is dequeued and joins the subterm, completing the CMM.

Hence $\big((\mathit{load}_x\ \mathit{points\text{-}to})(\overline{\mathit{points\text{-}to}}\ \mathit{store}_x)\big)\ \mathit{points\text{-}to}$ is the CMM bracketing associated with the above search order. Clearly multiple search orders can encode the same CMM bracketing, but where they differ is the directionality of the multiplication (i.e. which term was already known to the search and which term was just added). As we noted earlier, directionality can cause waste if overly large relations are encountered first in the multiplication. The directionalities with the above search order are:

- $\overline{\mathit{points\text{-}to}} \times \mathit{store}_x$ is **left**, i.e. store_x is already in the search, then $\overline{\mathit{points\text{-}to}}$ is subsequently added to the right of it.
- $\mathit{load}_x \times \mathit{points\text{-}to}$ is **right**.
- $(\mathit{load}_x \mathit{points\text{-}to}) \times (\overline{\mathit{points\text{-}to}}\ \mathit{store}_x)$ is **left**, the right subterm was computed before the left subterm, hence the left joins it *on its left.*
  - For simplicity, we might say that $\mathit{points\text{-}to}_1$ is in the **middle** of load_x and $(\overline{\mathit{points\text{-}to}}\ \mathit{store}_x)$.
- $(\mathit{load}_x \mathit{points\text{-}to}\ \overline{\mathit{points\text{-}to}}\ \mathit{store}_x) \times \mathit{points\text{-}to}$ is **right**.

The notion of directionality allows us to begin to reason about the cost of a given rule evaluation. Consider the rule $\mathit{points\text{-}to} \to \mathit{assign}\ \mathit{points\text{-}to}$, assuming that it is evaluated exactly after the allocations were added to *points-to*. Let **A**

and $\mathbf{P}$ be the adjacency matrices for *assign* and *points-to* respectively. In that case, the matrix algorithm would encounter the following problem:

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad \mathbf{P} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad \mathbf{P}' = \mathbf{P} \cup (\mathbf{A} \times \mathbf{P})$$

which results in:

$$\mathbf{A} \times \mathbf{P} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad \mathbf{P}' = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

In other words, the *points-to*-labelled summary edge $(this, H1)$ was discovered along the path $\langle assign(this, a), alloc(a, H1)\rangle$. But now consider the problem sparsely and with directionality. One option is to add *points-to* on the right of *assign*. We know *assign* ends once at the sink-vertex a, since it is already in the search. We then look at *points-to* once, in its a-row (i.e. the second row), and see that it has a successor $H1$ (the 5th column). No effort is wasted here, since a search was required in the a row and it yielded a successful witness in the $H1$ column. The other option is to add *assign* left of *points-to*. There are two edges in *points-to*, with different sources, looking into *assign* for a vertex with a sink at a succeeds (the edge is $(this, a)$), whereas looking for a vertex with a sink at *td* fails. Thus some search effort has been wasted in the latter case, and this occurred when a **dead-end** was discovered.

Dead ends are fundamental in our discussion of the CFL-R logic's efficiency. Unlike the somewhat nebulous notion of wasted search, dead-ends are observable artefacts which the evaluation can discover, and therefore which can be counted exactly.

Definition 16. *Given that a path is to be extended by appending or prepending a new group of edges,* ***dead-ends*** *occur when paths from the initial group do not meet any edge in the new group.*

Dead-ends relate to the search order in that the order defines which sub-paths are the "initial group" and which relation is the "new group". For example, given the search for Rule 5.5 via the search order:

$$\langle load_x, store_x, \overline{points\text{-}to}, points\text{-}to_1, points\text{-}to_2\rangle$$

- $load_x$ is the only relation in the search, so no dead-ends will be found yet
- $store_x$ does not connect to $load_x$, again no dead-ends will be found
- $\overline{points\text{-}to}$ connects $store_x$ on the latter's left, dead-ends are found where an edge in $store_x$ can not connect to any edge in $\overline{points\text{-}to}$.
- $points\text{-}to_1$ joins $load_x$ on the latter's right, so dead-ends occur where $load_x$ sinks do not meet *points-to* sources. Subsequently ($load_x$ *points-to*) joins ($\overline{points\text{-}to}$ $store_x$) on the latter's left, so dead-ends are discovered if $\overline{points\text{-}to}$ $store_x$ subpaths begin where no $load_x$ *points-to* subpath ends.

  - Again, we could say *points-to*$_1$ is in the middle of ($\overline{\textit{points-to}}$ *store*$_x$) and *load*$_x$, thus avoid deciding if it is left or right first.

- *points-to*$_2$ joins (*load*$_x$ *points-to* $\overline{\textit{points-to}}$ *store*$_x$) on the latter's right, dead-ends are discovered where the subpath ends if no suitable *points-to* edge begins there.

We say that dead-ends exist where *known* subpaths do not meet *new* edges because we believe this to be a reasonable property of many search strategies. Consider the Melski-Reps approach (Algorithm 2), where the "known" sub-path is the dequeued worklist edge and the "new" edges come from the relation used to extend that path on Lines 8 and 10. It is reasonable to count dead ends where the worklist edge does not join anything, since the algorithm spent search effort discovering that edge, and failing to extend the path constitutes some waste in discovering it. Orthogonal notions for the Chaudhuri and semi-naïve evaluation strategies can be observed, which motivate why it is better to count dead-ends where known subpaths can not be extended with a candidate set of edges.

The number of dead ends, importantly, can be computed exactly. We begin by noting that, whilst the *presence* of paths can be discovered using boolean matrix multiplication (as per Theorem 1), the *number* of paths is found if we use numeric multiplication instead.

Lemma 15. *Given* $\mathbf{G}$*, a numeric adjacency matrix of* G*,* $(\mathbf{G} \times \mathbf{G})^{u,v}$ *counts the number of two-step paths from* u *to* v *in* G*.*

Proof. Numeric multiplication is defined such that $(\mathbf{G}\times\mathbf{G})^{u,v} = \sum_{i\in[1,|V|]} \mathbf{G}^{u,i}\mathbf{G}^{i,v}$. Since $G^{x,y} = 1$ when there is an (x, y) edge and zero otherwise, the previous can be rewritten to $|\{i \mid i \in V, (u, i) \in E, (i, v) \in E\}|$, i.e. the number of intermediate vertices on a two-step path between u and v. □

Henceforth, all matrices and matrix-arithmetic will be assumed to be numeric, and the semantics of binary matrices will be simulated using the usual **signum** function:

$$sgn\,[\mathbf{A}]^{i,j} = \begin{cases} 0 & \mathbf{A}^{i,j} = 0 \\ 1 & \text{otherwise} \end{cases}$$

Lemma 15 forms the basis from which we derive the dead-end count. We omit the corollaries which give us *numeric* variants of Theorem 1, save for the following important result:

Corollary 3. *The number of* u, v *paths with path-word* $B_1 \dots B_k$ *is* $\left(\bigtimes_{i\in[1,k]} B_i\right)^{u,v}$

Thus we can relate *partial* path-searches to numeric matrices, and this allows us to reason about the discovery of dead-ends. As indicated above, there are three cases for how dead-ends can arise during the search, depending on whether the newest label-set in the search joins previous ones on the left, middle, or right.

Lemma 16. *Given* $\mathbf{P}$ *and* $\mathbf{Q}$*, two numeric matrices representing partially completed path-searches, and an adjacency matrix* $\mathbf{N}$*, the number of dead-ends discovered by adding* $\mathbf{N}$ *to the search is:*

$$\begin{aligned}\mathbf{P}\times\mathbf{N} &\Rightarrow \vec{1}\cdot\mathbf{P}\cdot\left(\vec{1}^T - sgn\left[\mathbf{N}\cdot\vec{1}^T\right]\right)\\ \mathbf{N}\times\mathbf{Q} &\Rightarrow \left(\vec{1} - sgn\left[\vec{1}\cdot\mathbf{N}\right]\right)\cdot\mathbf{Q}\cdot\vec{1}^T\\ \mathbf{P}\times\mathbf{N}\times\mathbf{Q} &\Rightarrow \vec{1}\cdot\mathbf{P}\cdot(\mathbf{1}-\mathbf{N})\cdot\mathbf{Q}\cdot\vec{1}^T\end{aligned}$$

Proof. For $\mathbf{P}\times\mathbf{N}$, we have $sgn\left[\mathbf{N}\cdot\vec{1}^T\right]$ is a vector which is 1 at element v if v is a source of an edge in $\mathbf{N}$ and 0 otherwise. Negate this vector to form the vector indicating *non-sources* of $\mathbf{N}$. $\vec{1}\cdot\mathbf{P}$ is a vector where element u is i if there are i paths in p ending at u. The dot product of these two is therefore the sum of all paths in $\mathbf{P}$ ending at a vertex that is *not* a source in $\mathbf{N}$.

$\mathbf{N}\times\mathbf{Q}$ follows symmetrically.

For $\mathbf{P}\times\mathbf{N}\times\mathbf{Q}$, the term $\mathbf{P}\cdot(\mathbf{1}-\mathbf{N})\cdot\mathbf{Q}$ is a numeric matrix recording all paths beginning with $\mathbf{P}$, passing through a non-existent edge in $\mathbf{N}$, and continuing with $\mathbf{Q}$. In other words, it records a path for every combination of $\mathbf{P}$ and $\mathbf{Q}$ that *can not* be joined by an $\mathbf{N}$-edge in the middle. □

We use the above lemma to calculate the number of dead ends encountered in a straightforward search. For the problem in Figure 5.1a, assume we are searching for Rule 5.5, with *points-to* $= \{(td, H2), (a, H1), (this, H1)\}$ (i.e. before the discovery of the $(tx, H2)$ path). Let the search order be left-to-right: $\langle load_x, \textit{points-to}_1, \overline{\textit{points-to}}, store_x, \textit{points-to}_2\rangle$. The relevant adjacency matrices for these relations are $\mathbf{L}$, $\mathbf{P}$ and $\mathbf{S}$ for $load_x$, *points-to* and $store_x$ respectively, implying that $\mathbf{P}^T$ records $\overline{\textit{points-to}}$. The search proceeds as follows

1. $load_x$ is chosen from the search, no other relations are currently in the search, the search space is represented by $\mathbf{L}$.

2. $\textit{points-to}_1$ is chosen, represented by $\mathbf{P}$, and joins the current search on the right. The number of dead ends is:

$$\vec{1}\cdot\mathbf{L}\cdot\left(\vec{1}^T - sgn\left[\mathbf{P}\cdot\vec{1}^T\right]\right) = \vec{1}\cdot\begin{pmatrix}0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&1&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\end{pmatrix}\cdot\begin{pmatrix}0\\0\\0\\1\\1\\1\end{pmatrix} = 0$$

3. $\overline{\textit{points-to}}$ is chosen, though now the current path-space is represented by $\mathbf{L}\times\mathbf{P}$. According to Lemma 16, we have:

$$\vec{1}\cdot(\mathbf{L}\cdot\mathbf{P})\cdot\left(\vec{1}^T - sgn\left[\mathbf{P}^T\cdot\vec{1}^T\right]\right) = \vec{1}\cdot\begin{pmatrix}0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&1&0\\0&0&0&0&0&0\\0&0&0&0&0&0\end{pmatrix}\cdot\begin{pmatrix}1\\1\\1\\1\\0\\0\end{pmatrix} = 0$$

4. $store_x$ is chosen, let $\mathbf{C} = \mathbf{L} \cdot \mathbf{P} \cdot \mathbf{P}^T$ represent the current progress of the search.

$$\vec{1} \cdot \mathbf{C} \cdot \left(\vec{1}^T - sgn\left[\mathbf{S} \cdot \vec{1}^T\right]\right) = \vec{1} \cdot \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = 1$$

   A dead end is found when the new relation is added. We see that this is indeed the case, as the current path-space held two subpaths:

   - $\langle load_x(tx, this), points\text{-}to(this, H1), \overline{points\text{-}to}(H1, a)\rangle$
   - $\langle load_x(tx, this), points\text{-}to(this, H1), \overline{points\text{-}to}(H1, this)\rangle$

   Whilst the first joins (a, td) in $store_x$, the second sub-path fails to join any edge in $store_x$ and dead-ends here.

5. Finally $points\text{-}to_2$ is chosen, with $\mathbf{C} = \mathbf{L} \cdot \mathbf{P} \cdot \mathbf{P}^T \cdot \mathbf{S}$. We have:

$$\vec{1} \cdot \mathbf{C} \cdot \left(\vec{1}^T - sgn\left[\mathbf{P}^T \cdot \vec{1}^T\right]\right) = \vec{1} \cdot \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} = 0$$

Thus we conclude that a single dead end is discovered during the above search. Though this gives us confidence that a closed form for the dead-end count can be derived, there is one additional complication, which arises in the case of **disjoint** searches. Such searches have terms that do not connect to previously explored terms, e.g. $\langle points\text{-}to_2, load_x, points\text{-}to_1, \overline{points\text{-}to}, store_x\rangle$. Observe that $\langle load_x, points\text{-}to_1, \overline{points\text{-}to}, points\text{-}to_2, store_x\rangle$ demands exactly the same joins (i.e. the same directionality) at the same times in the search, but the former begins with $points\text{-}to_2$ whilst the latter does not examine this relation until much later.

In order to quantify the costs of arbitrary searches, we must understand how disjointedness influences the search cost. Assuming that the solver, at least in some way, interprets the request to search for $points\text{-}to_2$ first, instead of fourth, then we can not assume these two orders will have the same cost. This is a reasonable assumption, since if the solver does not treat the two orders differently, then our cost analysis will merely determine that one of the two has lesser cost, despite this not being a real property of the solver. The orders differ in when $points\text{-}to_2$ is examined, so intuitively we assume that the search somehow "enqueues" these paths. This being the case, the first search is wasteful of information, since it has searched for $points\text{-}to_2$ well before it was needed, and that search wastes computational resources for every other term until it is actually needed. To account for computations that are wasted maintaining partial searches, we propose a **bias** function. This uses a simple heuristic based on the size of the unrelated paths (i.e. the cost associated with maintaining unrelated paths is proportional to the expected size of those paths):

Definition 17. *The **bias** when maintaining* $\mathbf{U}$*, an unnecessary group of paths in the path search, is* $|\mathbf{U}| = \vec{1} \cdot \mathbf{U} \cdot \vec{1}^T$.

The only outstanding issue is the task of determining, given a partially completed search, which terms are connected and which are disconnected. We encode these connections via the notion of **chains**. The chains are the group of *contiguous* subsets of body terms that collectively encompass the current partially completed search:

Definition 18. *The **chains** of $S \subseteq \{B_1, \ldots, B_k\}$ are given by the function:*

$$\begin{aligned} \mathit{chains} &:: 2^{\mathcal{T} \cup \mathcal{N}} \to 2^{2^{\mathcal{T} \cup \mathcal{N}}} \\ \mathit{chains}(S) &= \{\{B_l, \ldots, B_u\} \mid B_{l-1} \notin S, B_{u+1} \notin S, \forall i \in [l,u] : B_i \in S\} \\ S &= \bigcup_{\{B_l, \ldots, B_u\} \in \mathit{chains}(S)} \{B_l, \ldots, B_u\} \end{aligned}$$

The chains are used to understand how subsets of the rule's body connect to one another at each step in the search. For example, consider the search for Rule 5.5 using the search-order $\langle \mathit{load}_x, \mathit{store}_x, \mathit{points\text{-}to}_1, \mathit{points\text{-}to}_2, \overline{\mathit{points\text{-}to}} \rangle$:

1. load_x is dequeued, which is also the first term in its rule-body (i.e. it is B_1), and connects to nothing as there is nothing else in the search. The chains are $\{\{\mathit{load}_x\}\}$.

2. The subset of body terms becomes $\{\mathit{load}_x, \mathit{store}_x\}$, which are $\{B_1, B_4\}$ according to the order they appear in their rule. The chains therefore become $\{\{\mathit{load}_x\}, \{\mathit{store}_x\}\}$.

3. With $\mathit{points\text{-}to}_1$, the terms are now $\{B_1, B_2, B_4\}$, i.e. a new search term ($B_2 = \mathit{points\text{-}to}_1$) joins with a previous chain ($B_1 = \mathit{load}_x$). The chains are now $\{\{\mathit{load}_x, \mathit{points\text{-}to}_1\}, \{\mathit{store}_x\}\}$.

4. The $\mathit{points\text{-}to}_2$ term again adjoins an existing chain. According to the order of occurance in their rule body, the terms $\{1, 2, 4, 5\}$ are now in the search, so $\{\{\mathit{load}_x, \mathit{points\text{-}to}_1\}, \{\mathit{store}_x, \mathit{points\text{-}to}_2\}\}$ are the new chains.

5. $\overline{\mathit{points\text{-}to}}$ completes the rule, the final term always results in a singleton chain of all body terms: $\{\{\mathit{load}_x, \mathit{points\text{-}to}_1, \overline{\mathit{points\text{-}to}}, \mathit{store}_x, \mathit{points\text{-}to}_2\}\}$

Using the idea of chains and our counting strategies for dead-ends and bias, we can now construct a closed-form which quantifies the waste associated to a rule's search order. At this time, search order is the most relevant contributing factor towards waste, extending our presentation to include other artefacts of the CFL-R problem's logic is straightforward, hence we leave it to Sections 5.3.3 and 5.3.4. We first define some convenience notation.

Definition 19. *Given an (indexed) list of body terms $B_1, \ldots, B_k$, and their associated adjacency matrices $\mathbf{B_1}, \ldots, \mathbf{B_k}$, we define a **path-space** to be the numeric matrix:*

$$\mathbf{B}_{[l,u]} = \bigtimes_{i \in [l,u]} \mathbf{B_i}$$

Our chains function can now be used to define variants of the dead-ends and bias functions which are based on the addition of a new term to some given terms.

Definition 20. *Given $S \subset \{B_1, \ldots, B_k\}$, a subset of some rule's body terms, and $B_i \in \{B_1, \ldots, B_k\} \setminus S$, a new term not in S, we have the following:*

$$\begin{aligned}
\textit{dead-ends} &:: 2^{(\mathcal{T} \cup \mathcal{N})} \times (\mathcal{T} \cup \mathcal{N}) \mapsto \mathbb{Z} \\
\textit{bias} &:: 2^{(\mathcal{T} \cup \mathcal{N})} \times (\mathcal{T} \cup \mathcal{N}) \mapsto \mathbb{Z} \\
\textit{waste} &:: 2^{(\mathcal{T} \cup \mathcal{N})} \times (\mathcal{T} \cup \mathcal{N}) \mapsto \mathbb{Z} \\
\textit{dead-ends}(S, B_i) &= \begin{cases}
\vec{1} \cdot \mathbf{B}_{[l,i-1]} \cdot (\mathbf{1} - \mathbf{B_i}) \cdot \mathbf{B}_{[i+1,u]} \cdot \vec{1}^T & \{B_l, \ldots, B_{i-1}\} \in \textit{chains}(S) \\
& \wedge \{B_{i+1}, \ldots, B_u\} \in \textit{chains}(S) \\
\vec{1} \cdot \mathbf{B}_{[l,i-1]} \cdot \left(\vec{1}^T - \textit{sgn}\left[\mathbf{B_i} \cdot \vec{1}^T\right]\right) & \{B_l, \ldots, B_{i-1}\} \in \textit{chains}(S) \\
\left(\vec{1} - \textit{sgn}\left[\vec{1} \cdot \mathbf{B_i}\right]\right) \cdot \mathbf{B}_{[i+1,u]} \cdot \vec{1}^T & \{B_{i+1}, \ldots, B_u\} \in \textit{chains}(S) \\
0 & \textit{otherwise}
\end{cases} \\
\textit{bias}(S, B_i) &= \prod_{\{B_l, \ldots, B_u\} \in \textit{chains}(S \cup B_i), i \notin [l,u]} \vec{1} \cdot \mathbf{B}_{[l,u]} \cdot \vec{1}^T \\
\textit{waste}(S, B_i) &= \textit{bias}(S, B_i)\, \textit{dead-ends}(S, B_i)
\end{aligned}$$

We use these functions to derive the cost associated with each rule, and thus the cost of the entire search:

Definition 21. *Given a rule $H \to B_1 \ldots B_k$ is being expanded via the search-order $\sigma = \langle B_{\sigma_1}, \ldots, B_{\sigma_k} \rangle$, we define the **cost** to be:*

$$\begin{aligned}
\textit{cost} &:: (\mathcal{T} \cup \mathcal{N})^* \mapsto \mathbb{Z} \\
\textit{cost}(\sigma) &= \sum_{i \in [1,k]} \textit{waste}(\{B_{\sigma_j} \mid j \in [1,i)\}, B_{\sigma_i})
\end{aligned}$$

We can now calculate the cost of searching for a given rule using a fixed search-order. Let $\mathbf{L}, \mathbf{S}$ and $\mathbf{P}$ be the adjacency matrices for $\textit{load}_x$ $\textit{store}_x$ and *points-to* respectively, and assume the first three *points-to*-labelled paths from Figure 5.1a have been discovered. Rule 5.5 would then have the cost, based on the order $\langle \textit{points-to}_2, \textit{points-to}_1, \textit{load}_x, \overline{\textit{points-to}}, \textit{store}_x \rangle$, of:

1. $\textit{dead-ends}(\emptyset, \textit{points-to}_2) = 0$ so we do not observe any cost yet.
2. $\textit{dead-ends}(\{\textit{points-to}_2\}, \textit{points-to}_1) = 0$ since the new relation does not adjoin any of the known chains.
3. $\textit{load}_x$ joins the $\textit{points-to}_1$ term which is already in the search. This yields $\textit{dead-ends}(\{\textit{points-to}_1, \textit{points-to}_2\}, \textit{load}_x) = \left(\vec{1} - \textit{sgn}\left[\vec{1} \cdot \mathbf{L}\right]\right) \cdot \mathbf{P} \cdot \vec{1}^T = 2$. Also since $\textit{bias}(\{\textit{points-to}_1, \textit{points-to}_2\}, \textit{load}_x) = 3$ we find that the waste for this search term is 6.

4. Adding $\overline{\textit{points-to}}$, we see that it joins the chain $\{\textit{load}_x, \textit{points-to}_1\}$. The bias is $|\mathbf{P}| = 3$ since $\{\textit{points-to}_2\}$ is a disconnected chain, though this does not matter since we have $\vec{1} \cdot (\mathbf{L} \cdot \mathbf{P}) \cdot \left(\vec{1}^T - \textit{sgn}\left[\mathbf{P}^T \cdot \vec{1}^T\right]\right) = 0$ dead-ends.

5. $\textit{store}_x$ is in the middle of the two chains $\{\textit{load}_x, \textit{points-to}_1, \overline{\textit{points-to}}\}$ and $\{\textit{points-to}_2\}$. We have the dead-ends as $\vec{1} \cdot (\mathbf{L} \cdot \mathbf{P} \cdot \mathbf{P}^T) \cdot (\mathbf{1} - \mathbf{S}) \cdot \mathbf{P} \cdot \vec{1}^T$, i.e.:

$$\vec{1} \cdot \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot \vec{1}^T = 5$$

Thus the cost of that search order is 11, i.e. that many dead-ends will be discovered, or partial searches will be wasted, during a search based on that order.

5.2.3.1 An Execution Model

Our discussion of the costs associated with the *logic* of a CFL-R problem has thus-far been agnostic with respect to the actual solver. Whilst we may be able to formulate more accurate costs tailored towards a specific solver, we feel that CFL-R is uniquely positioned by being simple enough to allow for a more general approach towards understanding costs. Hence we focus on the costs of the logic, rather than the cost of the actual execution. Nevertheless, it remains to be shown that this "logical-cost" correlates to actual costs associated with execution.

The mapping between logic-cost and execution-cost is, of course, solver specific. Since we are unable to enumerate the solver algorithms, we can not even make a guarantee that the mapping is monotonic, let alone accurate. On the other hand, the factors that lead to dead-ends arising *must* have some presentation (possibly as an actual artefact of execution), so long as it is a reasonable assumption that the solver interprets the CFL-R logic at least partially literally. We can make the strong statement that *most* declarative systems must, at least in some corner cases, take literal hints from the specification. Such a system that took no hints is either trivially simplistic or has the ability to reason about the problem's logic on-par with human beings. On the other hand, the degree to which hints are taken may vary greatly, where some solvers interpret the specification mostly imperatively (such as our own solver, detailed in Section 6), whereas some may limit the expressive power (such as the Melski-Reps algorithm, which only accepts specifications in binary normal-form).

To demonstrate that the mapping between waste and execution inefficiency is not arbitrary, we devise a simple CFL-R solver for arbitrary grammars. Our solver will evaluate rules by using nested loops which iterate over each relations. Assume that we have an indexed data-structure, which is associated with each relation, allowing iteration, insertion, and fast lookup for successors and predecessors of a given vertex. We construct loop nests in-order for each of

```
for all (v3, v4) ∈ store_x do
  for all (v3, v2) ∈ points-to do
    for all (v4, v5) ∈ points-to do
      for all (v1, v2) ∈ points-to do
        for all (v0, v1) ∈ load_x do
          points-to ∪= {(v0, v5)}
```

(a) $\sigma = \langle store_x,\ \overline{points\text{-}to},\ points\text{-}to_2,\ points\text{-}to_1,\ load_x \rangle$

```
for all (v0, v1) ∈ load_x do
  for all (v3, v2) ∈ points-to do
    for all (v4, v5) ∈ points-to do
      if (v1, v2) ∈ points-to then
        if (v3, v4) ∈ store_x then
          points-to ∪= {(v0, v5)}
```

(b) $\sigma = \langle load_x,\ \overline{points\text{-}to},\ points\text{-}to_2,\ points\text{-}to_1,\ store_x \rangle$

Figure 5.2: Example evaluation code for Rule 5.5 based on two different search orders.

the relations in the search order, e.g. the first item makes the outermost loop and the second item will be nested immediately inside it. Each loop iterates over the pairs in its associated relation, and, where possible, limits a search if an outer loop has already bound that vertex. When both source and sink vertices are bound by outer loops, we can replace the loop with a simple existence check. The innermost scope adds a new pair to the relation of the rule's head, according to what the first and last vertices in the search are. Figure 5.2 shows the execution code for a search for Rule 5.5 based on two different search orders. We have chosen the variable names (i.e. the vertices) to indicate the index in the path at which they occur. Since we assumed that each relation has a fast bi-directional index (i.e. can be indexed from sources and sinks), the rule evaluation code simply uses *points-to* when searching for the $\overline{points\text{-}to}$ term (i.e. its reversal), as it is reasonable to assume that an efficient solver can avoid computing reverse relations in such cases.

We now extend the loop-nest based rule evaluation to a complete CFL-R solver in a naïve manner. Construct a loop nest for each rule, according to a fixed search order per-rule. During an iteration, every rule will be evaluated in the order they are defined in the language, so Rule 5.3 would be first, followed by 5.4. So long as the evaluation discovers a single path (i.e. the innermost nest is reached and a new pair is added to the relation), we continuously iterate the rule. Whilst this evaluation strategy is not efficient, it serves to demonstrate the correlation between logic-cost and execution.

Previously we saw that $\langle points\text{-}to_2, points\text{-}to_1, load_x, \overline{points\text{-}to}, store_x \rangle$ was predicted to have a cost of 11. Under the simplistic execution model presented, a dead-end implies a loop iteration that can not succeed. This was based on the assumption that three of the *points-to* paths had been discovered, and this evaluation of Rule 5.5 would yield the fourth. We track the iteration space of the search, to show that this is indeed the case. The constructed loop-nest, and the state of the iteration space when dead-ends are discovered, are depicted in Figure 5.3. According to the table, a loop was encountered where no suitable pair existed in the associated relation. Looking more closely at the prediction, we confirm that 6 dead ends occurred when adding $load_x$, i.e. 2 of the sources of the join relation $points\text{-}to_1$ are not sinks in $load_x$, and those were encountered 3 times each, once for every choice of $points\text{-}to_2$ which was already in the search.

for all $(v_4, v_5) \in$ *points-to* **do**
  for all $(v_1, v_2) \in$ *points-to* **do**
    for all $(v_0, v_1) \in load_x$ **do**
      for all $(v_3, v_2) \in$ *points-to* **do**
        if $(v_3, v_4) \in store_x$ **then**
          points-to $\cup = \{(v_0, v_5)\}$

#	$load_x$	*points-to*	$\overline{\textit{points-to}}$	$store_x$	*points-to*
1	$(?, td)$	$(td, H2)$			$(td, H2)$
2	$(?, a)$	$(a, H1)$			$(td, H2)$
3	$(tx, this)$	$(this, H1)$	$(H1, a)$	(a, td)	$(td, H2)$
4	$(tx, this)$	$(this, H1)$	$(H1, this)$	$(this, td)$	$(td, H2)$
5	$(?, td)$	$(td, H2)$			$(a, H1)$
6	$(?, a)$	$(a, H1)$			$(a, H1)$
7	$(tx, this)$	$(this, H1)$	$(H1, a)$	(a, a)	$(a, H1)$
8	$(tx, this)$	$(this, H1)$	$(H1, this)$	$(this, a)$	$(a, H1)$
9	$(?, td)$	$(td, H2)$			$(this, H1)$
10	$(?, a)$	$(a, H1)$			$(this, H1)$
11	$(tx, this)$	$(this, H1)$	$(H1, a)$	$(a, this)$	$(this, H1)$
12	$(tx, this)$	$(this, H1)$	$(H1, this)$	$(this, this)$	$(this, H1)$

Figure 5.3: Loop-nest and progress of Rule 5.5 when dead-ends are found. Both according to the search order $\langle \textit{points-to}_2, \textit{points-to}_1, load_x, \overline{\textit{points-to}}, store_x \rangle$. When a pair is needed to construct a path that does not exist in its relation, this is noted with red text.

The other 5 dead-ends were found when $store_x$ was included in the search.

5.2.3.2 Approximating Waste

The techniques described accurately calculate the costs associated with the logic of CFL-R problems. Whilst this model is useful for its accuracy, it is not actually reasonable to be used in practice, as the algorithmic complexity of calculating the cost far exceeds that of solving the CFL-R problem. The best algorithms we have for computing matrix multiplication [23] are the major bottleneck in the cost calculation, and they are assumed to have time-complexity $\mathcal{O}\left(n^{2.4}\right)$. Note that boolean matrix multiplication has no faster algorithm. This is almost as bad as the Melski-Reps approach, Algorithm 2, which is known to run in $\mathcal{O}\left(n^3\right)$ [47]. The cost calculation itself also calls for multiple multiplications: each call to $\textit{dead-ends}(S, B_i)$ requires $\mathcal{O}\left(|S|\right)$ such multiplications, as does $\textit{bias}(S, B_i)$. The running time of $\textit{cost}(\sigma)$ is therefore $\mathcal{O}\left(|\sigma|^2 n^{2.4}\right)$, as the cost function re-calculates *waste* once for every search term. For this discussion, we will limit our focus to simple approximations of the cost model, to better understand the problem, and leave more complicated strategies for future work.

As it is not reasonable to use real matrices in the cost calculation, we must devise an abstraction. Ideally, our abstraction will still give useful results, though it is possible that the inaccuracy of those results will adversely affect any optimisations that could be made with them. Firstly, we must avoid using matrices to represent relations in the search. Assume instead that the adjacency matrix $\mathbf{M}$ can be abstracted via some suitable approximation $\widetilde{\mathbf{M}}$. To apply the abstraction to the cost calculation, we require several operations to be defined over the abstract domain:

- We build abstract path-spaces via "abstraction **multiplication**": $\widetilde{\mathbf{A}} \cdot \widetilde{\mathbf{B}} = \widetilde{\mathbf{C}}$.
- We **project** the abstraction to the counts of its source and sink domains with $\vec{1} \cdot \widetilde{\mathbf{M}}$ and $\widetilde{\mathbf{M}} \cdot \vec{1}^T$ respectively.
- we take the **complement** of an abstraction or an abstract source/sink domain with $\mathbf{1} - \widetilde{\mathbf{M}}$ and $\vec{1} - \widetilde{\mathbf{M}}$ respectively.

In this way, any abstraction that allows for the operations of multiplication, projection, and complement, can be used as a substitute for the adjacency matrices in the cost calculation. Let us assume that the operations have time-complexities $\mathcal{O}(M)$, $\mathcal{O}(P)$, and $\mathcal{O}(C)$ for multiplication, projection, and complement respectively. We can derive the parametric time-complexity for the abstracted cost calculation to be $\mathcal{O}\left(|\sigma|^2 M + |\sigma|P + |\sigma|C\right)$, given that each call to *dead-ends* has at most two projections and one complement.

As long as an abstraction is being used to represent the matrices, we can also devise an abstraction for the execution model. Since the accuracy of the technique is already largely tied to the degree of abstraction for the adjacency matrices, it does not compound the inaccuracy by much when we also assume a simpler execution occurs. To this end, we propose that each $\widetilde{\mathbf{M}}$ should only represent the relation M as it is upon conclusion of the CFL-R algorithm, so since *points-to* terminates with $\{(td, H2), (a, H1), (this, H1), (tx, H2)\}$ when run on the problem in Figure 5.1, the approximation $\widetilde{\mathbf{P}}$ should abstract those four edges. Further, we assume that the CFL-R solver will terminate after evaluating each rule *once*. In reality, rules can be re-evaluated up to $\mathcal{O}\left(n^2\right)$ times, in the case that the rule is cyclic and each evaluation returns a single new result. To model the cyclic behaviour of some rules, it is sufficient to weight their cost according to the proportion of the execution time that they account for, though in our implementation we do not use the weights as it is sufficient to optimise the CFL-R logic according to the "single execution" assumption (i.e. weighting the cost does not noticeably improve the performance of optimised CFL-R problems).

Computing cost in an abstract domain is therefore possible for any suitable abstraction. Here we describe the abstraction used in our implementation (see Section 5.4), which has constant-time operations for multiplication, projection and complement. Our abstraction characterises each relation (i.e. each adjacency matrix) as a 3-tuple having:

$$\widetilde{\mathbf{R}} = (|\{s \mid \exists t : (s,t) \in R\}|, |R|, |\{t \mid \exists t : (s,t) \in R\}|)$$

I.e., each relation is represented by its size, and the number of source and sink vertices that adjoin the relation. Using the above abstraction we can devise suitable operations for multiplication, projection and complement, based on a probabilistic model. Given vertices V, an edge in a relation having t sinks will join an edge in a relation having s sources with probability $\frac{st}{|V|^2}$. We apply this logic to derive the following:

$$
\begin{aligned}
(s_A, A, t_A) \cdot (s_B, B, t_B) &= \left(\frac{s_A t_A s_B}{|V|^2}, \frac{AB}{|V|}, \frac{t_B t_A s_B}{|V|^2} \right) \\
\vec{1} \cdot (s, M, t) &= (s, M, |V|) \\
(s, M, t) \cdot \vec{1} &= (|V|, M, t) \\
\mathbf{1} - (s, M, t) &= \left(|V| - \frac{s}{|V|}, |V|^2 - M, |V| - \frac{t}{|V|} \right) \\
sgn\,[(s, M, t)] &= \left(s, \frac{stM}{|V|^2}, t \right)
\end{aligned}
$$

The abstracted *sgn* function is included as a matter of convenience. Clearly each abstract operation requires only a fixed number of multiplication or subtractions, thus they have constant time. This abstraction is sufficient for the purpose of optimising logical specifications, which we verify in Section 5.4. It may be that more accurate abstractions would yield better optimisations, though we leave such an exploration for future work.

5.3 Optimising Execution

The cost model we have designed for CFL-R logic has the useful property that it predicts costs *a-priori*. Specifically, given sufficient information about the input problem, the graph and the specification logic, we can make accurate guesses about the discovery of dead ends under varying conditions, particularly variations in the logic. We therefore use this model to understand and predict the performance of various logically-equivalent formulations of the input problem, and choose the best variant as an optimisation. We implement our optimisation framework for the prototype CFL-R solver described in Chapter 6.

CFL-R is uniquely viable as an optimisable logic fragment, as we can reason about logically equivalent formulations more strongly. In the case of CFL-R, logical equivalence refers specifically to *linguistic* equivalence. In general, it is undecidable to determine if two context-free grammars compute the same language [38], however we have the simpler problem of determining if a transformed language is equivalent. Thus, any transformation which is *language preserving* is guaranteed to be equivalent. We consider several transformations which are known to be language preserving, but can yield favourable execution performance under the right circumstances.

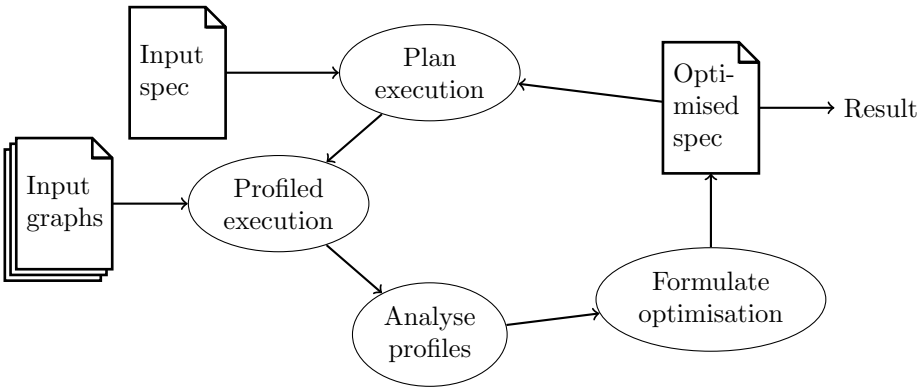


Figure 5.4: Outline of the feedback-directed optimisation approach

5.3.1 Feedback

Our optimisation approach is powered by knowledge of the input problem. We are mostly interested in characterising the input relations such that we can approximate the cost model (as per Section 5.2.3.2). Hence, we propose a profile-driven feedback-directed approach, such as the one shown in Figure 5.4. Feedback direction is useful in cases where the input specification alone is not enough to reason about optimisations. As we detail later in this section, the optimisations are necessarily tied to runtime data such as the relative sizes of relations, and the proportion of execution time that the rule evaluations demand. It is impossible to know, simply by looking at a spec, whether any transformation to the logic will yield superior runtimes, even if we have detailed knowledge about the solver. This is because the discovery of dead ends is necessarily tied to the size and density of the input graph, as well as the distribution of terminal labels amongst its edges.

The optimisation framework itself is a straightforward cyclic feedback-transform approach. Starting from an initial version of the CFL-R logic, called the **specification**, we derive a solver using a prototype CFL-R engine (see Chapter 6). The solver executes in a profiling manner on the training data, a representative sample of the problem which the user specifies to the optimiser. Profiling data is processed by our tool, from which several candidate transformations can be proposed. Whichever transformation is predicted to be most effective, using the cost model from Section 5.2.3, will be applied to the input specification to form an optimised specification. We repeat the optimisation cycle using the optimised specification as input, and running on the same training data. The optimisation loop ends either because a pre-set limit on the number of cycles is reached, or the framework determines that no optimisation is likely to improve performance (again using the cost model). As a practical means of handling inaccuracies in the cost model, the optimiser can roll-back and blacklist transformations if sub-

sequent runs of the optimiser do not improve performance sufficiently.

Unlike, for example, SQL queries, which change frequently along with the data on which they are run, CFL-R problems feature a mostly static grammar and variable input graphs. As a result, though SQL favours an online/dynamic optimisation approach, we propose optimising the specifications **offline**, via the use of training data. An offline approach is useful, as it allows us to derive the most efficient *solver* for the optimised logic, which avoids the overheads associated with introspection and dynamic optimisation. In the CFL-R context, we see that for a given problem (like points-to analysis) the grammar has a fixed form (such as the grammar in Figure 5.1b), whereas the input graphs are tied to the source code being analysed. Hence, optimising the problem's grammar usually gives a high-performance solver for every instance which shares that grammar. Of course, optimising based on training data is only useful if the training set is actually representative. In practice, we find that this assumption is reasonable to make.

Our optimisation framework is designed to work in an iterative approach. Iteration is chosen with the intention of making a tradeoff of time for improved accuracy (i.e. confidence in the optimisations). The model allows us to reason about the relations that result from transformations, i.e. if the transformation requires a new rule be added, then we can predict the runtime and outputs of that rule. Nevertheless, we do not wish to compound the inaccuracy of these predictions when making *further* optimisations, i.e. by using predicted relationships when predicting further output sizes. Each iteration requires reformulating an execution plan for the specification (according to the semi-naïve evaluation approach from Chapter 4) and re-executing the input graphs with profiling instrumentation. Ultimately this means the optimisation process *itself* is very expensive, as it takes significantly longer to optimise a specification than it does to run it over its training data. The assumption we made regarding offline optimisation, namely that for a certain class of CFL-R problem the input graphs will have similar characteristics, also allows us to use the more expensive approach. So long as problems that share their grammar do indeed have similar characteristics, the optimiser can train using *smaller* datasets, and the optimisations discovered will likely prove useful for large problems. In our experimentation, we found that small benchmarks were able to yield orders of magnitude speedup for the largest problems in our study, and brought instances, which had previously timed out, within solvability (see Section 5.4).

5.3.2 Ordering

Unsurprisingly, based on the mathematics of dead ends, the first optimisation we detail is ordering. Ordering strongly influences the discovery of dead ends, and the other optimisations are secondarily important to further reduce the dead-ends that reordering the rules can not eliminate. Ordering is indicated by affixing the following notation to the relevant rule:

$$A \to B\ C^{\{2\}}\ D^{\{1\}} \Leftrightarrow \sigma = \langle B, D, C \rangle$$

The actual operational meaning of the evaluation order will, of course, differ depending on the solving algorithm being used. Consider that the semi-naïve approach is chosen for solving the actual CFL-R problem. In this strategy, the order of rule evaluations is chosen based on the topological order formed by dependencies between the rules. Each rule is evaluated in several ways, depending on which of the body terms is chosen as the **delta relation**. For example, the following might be chosen to evaluate Rule 5.5 from Grammar 5.1b:

$$\textit{points-to} \rightarrow \textit{load}_x \ \Delta_{\textit{points-to}} \ \overline{\textit{points-to}} \ \textit{store}_x \ \textit{points-to}$$

i.e. $\textit{points-to}_1$ is the delta relation. Next, the semi-naïve strategy joins the body terms to form paths via **relational composition**, which itself is defined by the data structure is used to store the relation (as detailed in Section 4.3). Therefore, the search order should be interpreted as the order in which composition occurs. The semi-naïve strategy is interesting because it proposes several evaluation strategies for each rule. In the event that there are different optimal orderings for the different versions, then the search-order can only optimise one of these. For simplicity, we will assume that there is an order which is reasonably effective for all versions of the rule.

To automatically determine an effective ordering, the optimiser uses the approach described by Selinger [65]. Selinger's algorithm is the standard technique for determining effective join orders, and is used in industry and research for many RDBMSs. The algorithm determines a near-optimal search order on a per-rule bases. We tailor the technique to our specific cost function, which determines the best ordering for a given rule $H \rightarrow B_1 \dots B_k$, in the following way:

1. Construct the subset lattice of body terms in the usual way:
   - The vertices of the lattice are drawn from $\{S \mid S \subseteq \{B_1, \dots, B_k\}\}$
   - For each node S, draw a directed edge (S, S') between it and another subset if and only if $S' = S \cup \{B_i\} : B_i \in \{B_1, \dots, B_k\} \setminus S$.
   - $\bot = \emptyset$ and $\top = \{B_1, \dots, B_k\}$
2. Weight the edges of the subset lattice. The edge $(S, S \cup \{B_i\})$ has weight $\textit{waste}(S, B_i)$. In other words, the weight of the edge is the amount of wasted computation that occurs when the body term not in S is added to it.
3. The minimal weight path from $\bot$ to $\top$ is therefore the best search order. This follows directly from the observation that each edge adds a single body term to the search order, and every successor vertex S' of S is strictly a superset: $S' \supset S$. Hence the path with minimum weight corresponds to the order which adds all body terms whilst minimising the wasted search effort.

In our actual implementation, we simply enumerate all orderings in the event that the rule body is small enough, and fall back to Selinger's approach when the

rule is too large. Minor optimisations of this strategy include determining edge weights and subset memberships in a lazy fashion, i.e., only when the Dijkstra's search requires determining the path cost between those vertices. The usual relational joining problem for which Selinger's is designed has more degrees of freedom than the search order we use in the CFL-R context, such as the join style, association, and directionality. Hence, it delivers a potentially sub-optimal solution which does not explore some options. However, the version of Selinger's algorithm used by our optimiser does explore all potential orderings (that is, all orderings could be candidates in the search), so we are guaranteed to derive the best ordering for the CFL-R problem.

5.3.3 Promoting

The notion of pre-computing sub-paths for a search is captured by the **promotion** optimisations. In general, promotion is the process of factoring out subsequences of body terms in one rule into their own rule. Rule promotions can be seen as a space/time tradeoff; calculating a subpath every time requires no additional storage space, but uses computational resources, whereas computing that sub-path in advance means that it must be stored somewhere. As a general guideline, promotion is an effective optimisation where the promoted rule changes infrequently (or not at all), and where it is not too large (so it uses less memory), or where the calculation is quite expensive. There are several cases which our solver recognises where one or more rules could benefit from promotion:

- **Common sub-expressions** occur when multiple rules contain the same sequences of body terms. Consider the following rules:

$$\begin{array}{ll} A \to B\ C\ D & A \to B\ X \\ E \to F\ C\ D & E \to F\ X \\ & X \to C\ D \end{array}$$

  Both grammars compute the same languages for nonterminals A and E, though the rightmost grammar factors the $C\ D$ sub-paths into their own nonterminal. The optimiser would choose such an optimisation if it predicted, using the cost model, that X was actually small, or $C\ D$ was expensive to compute.

- **Cyclic redundancies** are similar to common sub-expressions, except that they occur in one cyclic rule rather than multiple rules. The following rules show a normal and promoted cyclic rule:

$$\begin{array}{ll} C \to C\ P\ Q & C \to C\ X \\ & X \to P\ Q \end{array}$$

  Whilst it appears that no saving occurs here, since we have to compute $P\ Q$ anyway and we are only wasting memory to store it, the rule is in

fact cyclic. Cyclic rules typically must be executed multiple times until a fixpoint is reached, meaning that if P and Q are not also cyclic, the above significantly cuts down on the wasted computation associated with computing $P\ Q$. On the other hand, if P or Q depended on C in their grammars, then that would require cyclicly recomputing X anyway.

- **Summary chains** allow the optimiser to minimise the bias that contributes to waste in the search. As such, they assist more towards making other optimisations (like reordering) more effective. Given that a particular join, or series of joins, in a rule's body is guaranteed to encounter dead ends, the optimiser can only realistically explore that subpath when the bias is as small as possible. On the other hand, if the dead ends are encountered in a completely separate rule, then we are able to explore the subpath (via its intermediate stored form) in better time. In Figure 5.1, the very small load_X and store_x relations mean that a search order like $\langle \mathit{load}_x, \mathit{points\text{-}to}_1, \mathit{store}_x, \overline{\mathit{points\text{-}to}}, \mathit{points\text{-}to}_2 \rangle$ is optimal (indeed, it has zero dead-ends on Graph 5.1a). Nevertheless, the disjoint search has bias, which comes from the load_x *points-to* subpaths that are searched first. Consider instead:

  $$\mathit{points\text{-}to} \to L\ S\ \mathit{points\text{-}to} \quad L \to \mathit{load}_x\ \mathit{points\text{-}to} \quad S \to \overline{\mathit{points\text{-}to}}\ \mathit{store}_x$$

  We retain the advantages of the search order above, whilst also ensuring that no bias is encountered during any join (assuming a search order now of $\langle L, S, \mathit{points\text{-}to} \rangle$, $\langle \mathit{load}_x, \mathit{points\text{-}to} \rangle$ and $\langle \mathit{store}_x, \mathit{points\text{-}to} \rangle$ respectively). In this way the promotion allows us to encounter dead-ends in a more advantageous way, which may reduce wasted searches.

When the optimiser is considering making a promotion, it calculates an approximation of the new relation (i.e. the one that stores the promoted subchain), and uses that to calculate dead ends in all the resulting rules. We use heuristic weights to account for the affects of cyclic rules. For example, a cyclic redundancy is essentially free to compute (itself), but will be used as many times as the rule it was promoted from cycles, therefore we weight the waste of the promoted rule significantly less than the resulting cyclic rule.

5.3.4 Filtering

When the CFL-R evaluation requires examining very large relations and joining them multiple times, it may be advantageous to focus on a small subset of those relations. We propose the use of **filters**, as a means of restricting a given edge-set to only those pairs which are known to join as they are needed. Consider Rule 5.5, as applied to Graph 5.1a. The *points-to* relation is very large when compared with the other relation's sizes, yet it is joined three times. Assuming Graph 5.1a is representative, we could be wasting significant computational resources simply examining *points-to* three times for each evaluation of Rule 5.5. Instead, observe that only a subset of *points-to* is actually needed, namely the

subset with source vertex in the sink of $load_x$ or either endpoint of $store_x$. Since $load_x$ and $store_x$ are terminal symbols in the language, the set of vertices which *points-to* needs to begin at can be computed in advance. Given such a filtering set, we can calculate a subset of *points-to* that is needed for Rule 5.5 and use that instead.

Let ${}^{<}R = \{(u,u) \mid \exists v : (u,v) \in R\}$ be the source-filter of an edge-set R. Similarly $R^{>}$ is the sink-filter. Graphically, the filters are self-loops, which join with the target relation in order to project a subset of it. We can then reconstruct Rule 5.5 as follows:

$$F \rightarrow load_x^{>} \mid {}^{<}store_x \mid store_x^{>} \quad P \rightarrow F\ points\text{-}to \quad points\text{-}to \rightarrow load_x\ P\ \overline{P}\ store_x\ P$$

Firstly, F is only evaluated once (since it depends only on terminals). Observe, also, that *points-to* only appears once in the body of these rules. The implication is that significantly fewer computational resources will be wasted reading *points-to* multiple times. Of course, the mathematics of dead-ends also justifies the creation of filters, as the filtered relation will always encounter as few or fewer dead ends during its search.

Unfortunately, filtering is not always applicable to the problem context. Filtering transformations require semantics that go beyond the normal scope of CFL-R, so it is questionable whether an arbitrary solver would be able to recognise and adopt a different evaluation strategy in the presence of a filtering transformation. Further, filtering is highly sensitive to the actual computer architecture, so our logical optimiser often mistakes effective cases for filtering, at which point we rely on the optimiser rolling back and blacklisting that ineffective transformation.

5.4 Experiments

By design, the costs associated with CFL-R are agnostic with respect to the solver. Of course, some solvers do not permit instruction that would allow us to avoid known inefficiencies, though this is an orthogonal issue. For example, a Melski-Reps based solver expects relations in binary normal-form, and always extends a path from its dequeued worklist edge. It would seem, then, that ordering is not a useful optimisation to a Melski-Reps solver, but this is not actually true; the order details the best way to decompose long rule chains into binary chains. Thus, we expect the inefficiency calculations and logic transformations discussed previously will be applicable to most reasonable solver implementations.

5.4.1 Setup

In the case of our chosen solver, we implement the feedback-directed optimisation framework as part of the prototype CFL-R engine, which is discussed in Chapter 6. Briefly, the engine synthesises solvers from the input CFL-R language, which in turn use the semi-naïve evaluation strategy. All the logical

optimisations discussed in Section 5.3 are understood by the prototype CFL-R solver, and it can be instructed to run in a profiling mode, which we then use to perform the feedback optimisation strategy.

For our experiments, we examine the case-study of **points-to analysis**, which is a fundamental program analysis. The points-to analysis performed for our experiment is more complicated than the simple example from Figure 5.1b, by virtue of the following:

- The analysis is **field sensitive**, in the sense that it tracks points-to relationships in a way that respects the semantics of field loading and storing. Simple analyses simply treat all loads or stores to an object as a kind of indirection, similar to `a = *b` and `*b = a` from C/C++. Such analyses are imprecise in the presence of multiple fields, and treat all fields the same.

- The analysis is sensitive towards **dispatch** semantics. The semantics of virtual dispatch, where the runtime target of a method invocation is determined by the runtime type of a call's receiver object, are common amongst object-oriented languages. The points-to analyses of Figures 1.2 and 5.1 treat method calls as assignments, in the sense that the i-th parameter is assigned from the call-site's i-th argument, and `this` from the receiver object. To handle virtual dispatch, we use the *points-to* relation to adapt the call-graph on-the-fly according to the potential types of the receiver objects.

We propose two analyses, called Virtual and Conservative, which vary in how they treat the call-graph. For Virtual, the points-to relation is used to construct paths between call-site *method descriptors* and declared-method *types*. In other words, the call-graph is discovered on-the fly so long as points-to relations are discovered. For example, if o is the receiver object of a call-site, and $(o, h) \in$ *points-to*, then let t be the static type of the object(s) allocated by h, thus the call-site will invoke whichever version of the method that t objects call into. For Conservative, we statically over-approximate the possible targets of the call-site invocations, i.e. the method called by a call-site with receiver object of static-type t is any method declared by a subtype of t, or the closest supertype if t does not override the method. These two analyses were chosen as they provide a similar outcome (i.e. they are refinements of points-to analysis with some type-awareness) but they have different characteristics. The Virtual analysis builds up its dispatch table during computation, making the computation of the relation relevant to the optimiser, whilst Conservative does not, meaning the optimiser will be unable to leverage those computations to improve performance. The grammars for Virtual and Conservative, along with a summary of the semantic interpretation for the different edge labels, are shown in Figure 5.5.

We run our experimental analysis on standard Java benchmarks, drawn from the DaCapo suite [13]. The DaCapo benchmarks have two major releases: the initial release from 2006 and a subsequent 2009 version (called "Bach"). We

Label	Code	Semantics (for an (s, t) edge)
ACTUAL_i	`a = o.compare(s);`	s is the i-th argument to the call at site t
ALLOC	`s = new Object();`	a `new` statement at site t is assigned to s
ASSIGN	`s = t;`	t is assigned to s
DECLARED_TYPE	`String s;`	t is the declared type of variable s
DEFINER_d	`String toString()`	s is the type that defines (overrides) methods having descriptor d with an implementation whose signature is t
FORMAL_i	`void foo(s)`	s is the i-th parameter in a method with signature t
LOAD_f	`s = t.f;`	t's f-field is loaded into variable s
RECEIVER_d	`bar = s.d();`	s is the receiver variable of the call at site t to a method with descriptor d
REFLEX_SUBTYPE	`class S extends T`	t is a reflexive transitive subtype of s
RETURNED	`return s;`	the method with signature t returns s
RETURNING	`s = foo.bar();`	s is assigned the return value at site t
STORE_f	`s.f = t;`	t is stored into s's f-field
TYPE	`foo = new T();`	t is the type of the object allocated by a `new` statement at site s

```
pt: ALLOC
  | ASSIGN pt
  | LOAD_f pt -pt STORE_f pt
  | FORMAL_i -DEFINER_d -DECLARED_TYPE
    -REFLEX_SUBTYPE RECEIVER_d
    -ACTUAL_i pt
  | RETURNING -RECEIVER_d
    REFLEX_SUBTYPE DECLARED_TYPE
    DEFINER_d -RETURNED pt ;
```

(a) Conservative

```
pt: ALLOC
  | ASSIGN pt
  | LOAD_f pt -pt STORE_f pt
  | FORMAL_i -DEFINER_d -TYPE
    -pt RECEIVER_d -ACTUAL_i
    pt
  | RETURNING -RECEIVER_d pt
    TYPE DEFINER_d -RETURNED
    pt ;
```

(b) Virtual

Figure 5.5: The points-to grammars used for our experimentation, and the semantics of the terminal-labelled edges. Note that "ternary" relations are placeholders for a multiplicity of relations, one for each element in the third domain (e.g. ACTUAL_i → ACTUAL_1, ACTUAL_2, ...).

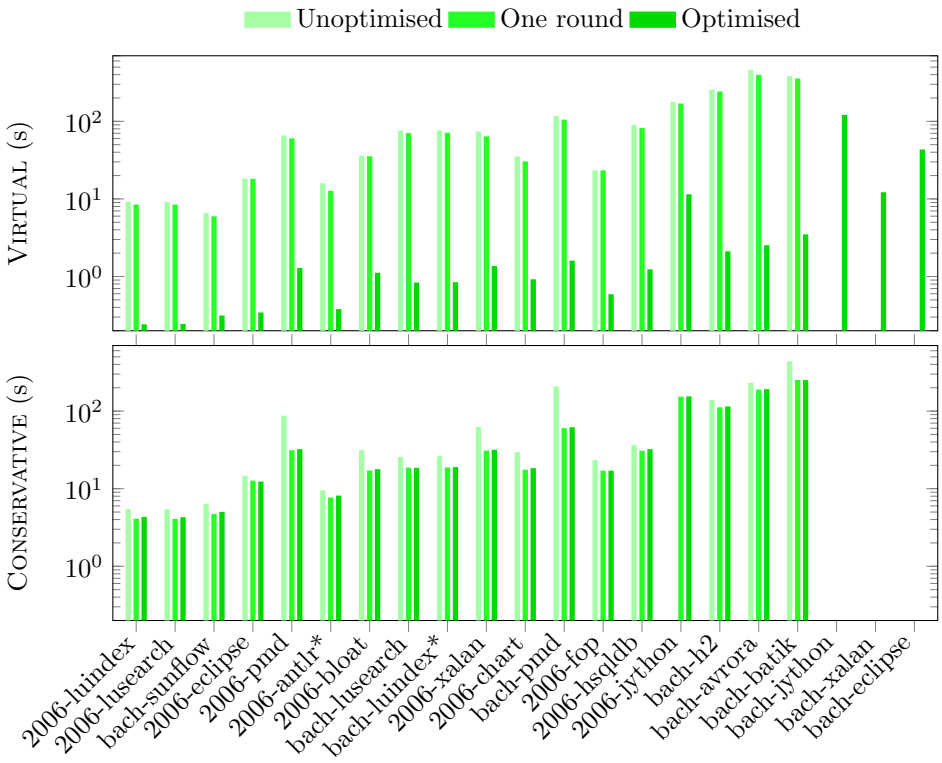


Figure 5.6: Runtimes for solving the Virtual and Conservative points-to analyses with and without optimisations.

run experiments over both suites. We use standard tools to generate input graphs from the DaCapo programs, specifically we use DOOP [66] as a means of generating semantic graphs from the input programs, which we subsequently process for input to our prototype solver using simple text manipulation. The experimental machine is a 3.2GHz Intel® i5-4570 CPU and 8GB RAM, running Ubuntu 16.04 in desktop mode. All execution time statistics, including the time taken for solving problems, optimising grammars, and comparing with alternative tools, is taken as an average over three runs.

5.4.2 Results

We first examine the effectiveness of optimisation for the different grammars. Figure 5.6 shows the runtimes of solving the Virtual and Conservative problems over the DaCapo benchmarks. The plot compares three versions of each solver, the first is not optimised (i.e. it was written in the most straightforward manner), the second has been optimised with a single pass, and the third has been optimised with unlimited passes (though in practice, 6 passes were needed

Table 5.1: Comparison of the virtual dispatch analysis used to demonstrate our optimiser against the well known DOOP framework's context-insensitive analysis. This is a re-production of a dataset published in [37].

Bench	$\|V\|$	$\|E\|$	$\|pt\|_V$	$\|pt\|_D$	T_V	T_D
2006-luindex	54,783	100,359	9,628	4,863	0.24	11.33
2006-lusearch	54,990	100,731	9,678	2,365	0.24	10
bach-sunflow	66,390	137,747	13,943	14,991	0.31	13.33
2006-eclipse	76,112	137,306	11,234	13,435	0.34	15
2006-pmd	79,274	167,757	17,186	4,173	1.28	12.67
2006-antlr*	84,643	158,367	13,437	18,504	0.38	15.33
2006-bloat	95,249	179,023	39,058	25,733	1.12	14.33
bach-lusearch	97,666	177,561	18,032	3,483	0.83	13.67
bach-luindex*	99,206	180,611	18,448	10,548	0.84	15.33
2006-xalan	107,222	217,835	22,019	2,271	1.37	13.33
2006-chart	109,756	220,018	24,234	7,557	0.92	15
bach-pmd	110,221	234,508	23,367	6,677	1.6	14.33
2006-fop	116,029	233,719	23,938	12,606	0.59	16
2006-hsqldb	148,873	265,518	32,541	2,413	1.24	15.33
2006-jython	161,015	423,961	42,299	202,481	11.46	27.67
bach-h2	174,966	331,573	76,029	5,217	2.11	18.67
bach-avrora	191,641	369,613	49,461	4,837	2.53	17.67
bach-batik	293,646	605,777	57,516	23,381	3.49	39
bach-jython	463,573	1,560,308	714,448	228,624	121.24	113.67
bach-xalan	552,233	1,068,759	143,796	3,519	12.19	59.33
bach-eclipse	1,000,511	1,967,814	292,016	30,827	43.38	259.67

by both benchmarks). Benchmarks with an asterisk (*) were used as training data for the optimiser. The VIRTUAL problem showed significant improvement when optimised, showing two orders of magnitude speedup and even allowing problems which previously timed-out to be solved in a reasonable amount of time. The CONSERVATIVE benchmark was less amenable to optimisation, and whilst some improvement was made by performing a single transformation, further changes did not gain from there. This shows us firstly that logical optimisation is sensitive to the input problem. As the logic of CONSERVATIVE is mostly controlled by a relatively dense input edge set (i.e the static call-graph, a coarse over-approximation), we see that few improvements can be made to allow the logic to work faster. By comparison, VIRTUAL discovers most of its edges on-the-fly, so improving its logic yielded significantly better solvers.

Since our experimental setup uses the DOOP framework as a means of generating test cases, we are interested in the relevant statistics regarding the similarities and differences with that tool. Table 5.1 records metrics about the datasets, including the number of vertices and input (terminal) edges for each problem. Importantly, $|E|$ is mostly $\mathcal{O}(|V|)$, differing by between $2\times$ and $3\times$. The columns $|pt|_V$ and $|pt|_D$ show the number of points-to edges as discovered

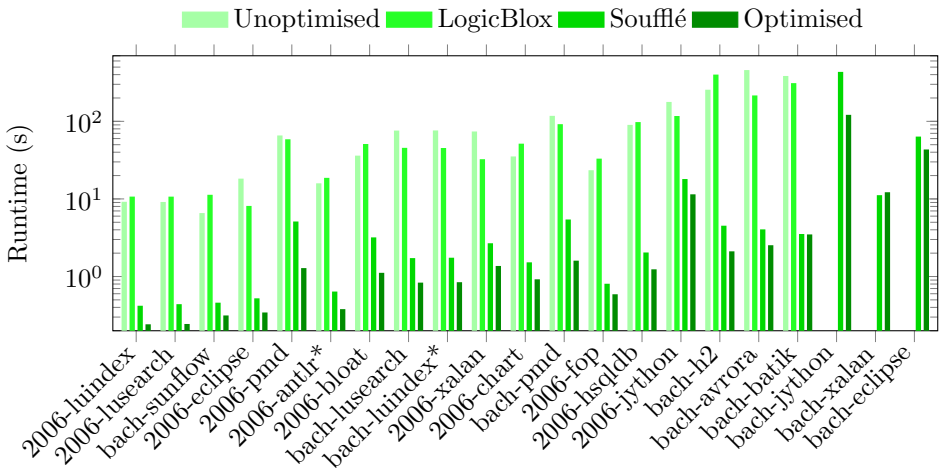


Figure 5.7: Runtimes of LogicBlox and Soufflé on the VIRTUAL problem, as compared with our optimised and unoptimised versions.

by VIRTUAL and DOOP respectively. As we see, the logic of the two techniques differs, so we would not expect the same results. Nonetheless, VIRTUAL is a good characterisation of the input problems, as its output size roughly correlates with the more accurate DOOP, hence VIRTUAL and CONSERVATIVE are representative of points-to analyses. The T_V and T_D record the runtimes for VIRTUAL (fully optimised) and DOOP respectively. We see that DOOP has some overheads which we would expect of industry-grade analyses, though the runtime can vary wildly, having between 0.9× and 10× slowdown/speedup when compared with VIRTUAL.

Next, we compare the utility of optimisation against commodity logic systems based on Datalog. Figure 5.7 examines how optimisation compares with using the tuned solvers LogicBlox [5] and Soufflé [40]. LogicBlox does not make any optimisations to its input specification, to the best of our knowledge, hence we see that its performance is only slightly better than the runtime of our unoptimised specification on the prototype solver. Soufflé uses a heuristic based optimiser for scheduling rules, and we see that it performs well, though still this approach is not as effective as our prototype solver when using the most efficient logic. We conclude that fast solvers alone are not enough to ensure high-performing applications, but that the logic of those solvers also needs to be optimised, which can be done in an automatic way.

Finally we examine the individual transformations, and see how they influence runtimes individually. We instruct the optimiser to consider only a single class of transformations (i.e. ordering, promotion, or filtering) and optimise the VIRTUAL specification with a single round. The results of these transformation, and the runtimes of those optimised specifications on the input benchmarks are shown in Figure 5.8. The ordering transformation observes a very small

```
pt: FORMAL_i{6}
    -DEFINER_d{5}
    -TYPE{3} -pt{2}
    RECEIVER_d{1}
    -ACTUAL_i{4}
    pt{7} ;
```

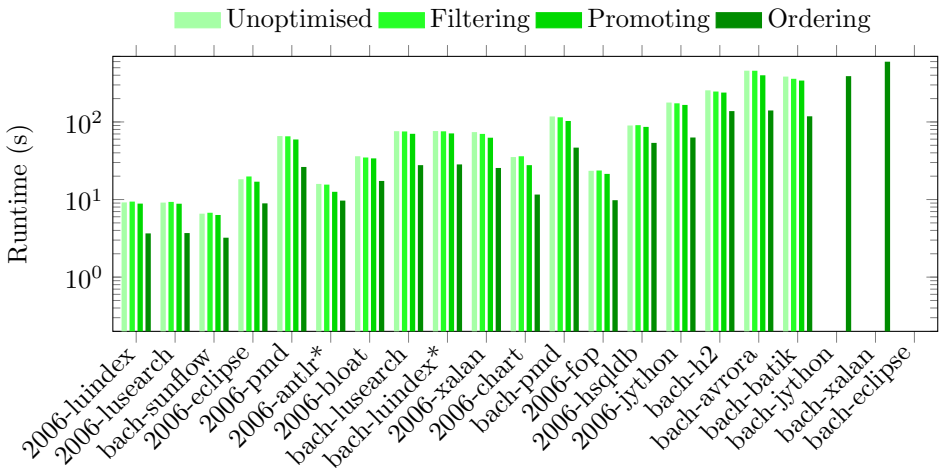
(a) Ordering

```
X_d: TYPE DEFINER_d ;
pt: FORMAL_i -X_d
        -pt RECEIVER_d
        -ACTUAL_i pt
  | RETURNING
        -RECEIVER_d pt
        X_d -RETURNED
        pt ;
```

(b) Promotion

```
F: STORE_f>
  | <STORE_f ;
X: F pt ;
pt: LOAD_f pt -X
        STORE_f X ;
```

(c) Filtering



(d) Comparing the effectiveness of the different transformations. The virtual dispatch grammar is optimised with a single pass.

Figure 5.8: The VIRTUAL problem when allowed a single optimising transformation, and the runtimes for those optimised specifications on our benchmark suite.

`RECEIVER_d` relation (i.e. the relation between callsites and their receiver variable), and uses this as a bottleneck to keep the number of intermediate paths small. The optimiser promotes `TYPE DEFINER_d` (i.e. sub-paths joining heap objects to the method objects of that type call via signature d), which is redundant in two rules: one linking parameters to arguments, and one linking return statements to returned variables. Filtering is performed for two of the three uses of *points-to* in the equivalent version of Rule 5.5, thus maintaining a smaller filter was deemed superior to performing fewer reads of the larger relation. We see that ordering is by far the most effective standalone optimisation, whilst filtering is quite volatile, and produces a slight slowdown on some of the smaller datasets.

Chapter 6

CFL-R Tools

This chapter describes a tool for synthesising CFL-R solvers from an input DSL, called **Cauliflower**. Cauliflower is publicly available as an open source [1] project. Much of this chapter was published to LPAR, 2017, in the paper "Cauliflower: A Solver Generator for Context-Free Language Reachability" [36].

6.1 Design Goals

The CFL-R problem is an important framework for formalising and evaluating many program analyses. The conceptually simple framework lends itself to diverse analyses by virtue of closely modelling relationships amongst program constructs, making CFL-R amenable to phrasing data-flow [59], object-flow [88], control-flow [79], specification inferencing [10], set-constraint solving [47], shape-analysis [57] and alias-analysis [68, 27]. Despite its generality, the actual solvers for CFL-R problems (particularly all of the above cases) are custom-made on a per-problem basis. We would expect that such a general and useful framework should have specialised tools for solving it, yet in our survey of the literature we have found no such system. Hence, there is a need to develop capable and robust tools to assist the research and development in the CFL-R space.

When designing a CFL-R tool, we must first understand the reasons why such a tool does not *already* exist. Our first clue comes from the diversity of CFL-R problems themselves. A CFL-R application developer needs to focus on the dual problems of understanding their input graphs (the problem instances) and grammars (the problem logic). It is tempting to see the logic as inherently tied to the solver, in which case the diverse array of problems demands a diverse array of solvers. In this work we aim to show that it is better to optimise solvers for the underlying CFL-R problem, using superior evaluation strategies and datastructures, and to optimise the problem's logic separately. Further, the diversity of CFL-R applications limits the ability of new CFL-R research to build on the results of older work. CFL-R practitioners who adopt solvers

[1] `https://cauliflower-cflr.github.io/`

from the research literature will find they perform very poorly on even slightly different logic, as those solvers were tailored to the specific logic of their problem class. As Chapter 5 suggests, inefficiencies in the *logic* of the CFL-R problem, at least in the face of biases in the inputs, is a major cause of under-performing solvers. Knowing this, we can begin to formalise the key concerns that our high-performance CFL-R tool must address.

6.1.1 Expressibility

Many CFL-R problems that we see in the research literature do not follow a strict adherence to the CFL-R formalism. Firstly, many applications diverge from the usual evaluation semantics. Many problem contexts require demand-driven analyses [70] whilst others use incremental techniques [46]. The standard algorithms for CFL-R evaluate in a bottom-up fashion [17, 56], and so are not directly applicable to those contexts.

More interesting are the extensions to the CFL-R framework itself, which many applications use. As we have seen in the running example (Figure 1.2), the use of *reversal* is common for many CFL-R problems. Sridharan et al. discuss the mechanical approach to formulating reversal [70] in their work, though many later advances simply assume the ability to reverse a CFL-R language without discussing its implications [27]. Another common semantic extension to CFL-R is the use of *negation*, or the absence of edges. Xu et al. devise a means of improving an analysis' performance by computing a fast pre-analysis and negating its results [84]. Negation is a common element of Datalog problems, and several different semantic interpretations of negation have been proposed for that context.

When designing a general purpose CFL-R tool, we must be able to account for these diverse requirements. We group these needs together as **expressibility** concerns. Firstly, we must cover the known expressibility limitations of CFL-R solvers, which we achieve by providing means of handling the semantic requirements demanded by the individual tools found in the research literature. Secondly, we provide additional semantic extensions to supplement these expressibility concerns, based on foreseeable needs, and which can be implemented efficiently in the solver. The resulting solver is able to formulate and handle classes of input which previously required more powerful/general machinery than CFL-R could provide, yet still maintains the performance advantages associated with the simple CFL-R formalism.

6.1.2 Performance

Primarily, any general purpose tool which solves CFL-R problems must be designed for high performance. In Chapter 4 we saw that current tools for CFL-R have significant scalability limitations. Indeed, CFL-R is known to be in the 2NPDA class [34], for which $\mathcal{O}\left(\frac{n^3}{\log n}\right)$ is the best known bounds [17]. Indeed, CFL-R is often the bottleneck in the applications that require it, and it is not

expected that parallel processing can be used to significantly improve runtime performance [56]. So long as tools for CFL-R are underperforming, at least when compared to custom implementations, we can not expect practitioners to prefer them to hand-tailored implementations.

Part of the significant improvement that custom solutions see over generic solvers is in the availability of domain knowledge. Developers who go the route of catering a solver to their specific problem can implement features and optimisations which are necessarily effective in their domain. The optimisation approach described in Chapter 5 is a useful automatic means of allowing a generic solver to accommodate for the domain knowledge, though we also note that, even without such an optimiser, the transformations used to perform optimisations can still be implemented manually, a process which should still be faster than developing a solver from scratch.

When considering the performance aspects of a CFL-R solver, we must compare it to alternative customised and generic tools. We do not expect that a general purpose solver can outperform manually developed solvers, nonetheless we hope to keep the performance differences minimal. Importantly, though, a CFL-R solver must prove its utility as compared to more *general* tools. There already exists high-performance solvers for more general classes of logic including Datalog [5, 40] and SMT [49]. Since CFL-R is a simpler formalism, we expect to be able to make assumptions about its execution that the more general logics can not, and in this way outperform them. Thus, we aim for the performance of the CFL-R engine to be somewhere between that of high performance hand-tailored implementations and that of more general/powerful solvers.

6.1.3 Convenience

Manually developed CFL-R solutions have the advantage over general ones in terms of expressibility and performance. Naturally, the developer can implement the semantic features necessary for their problem in any custom solver. Further, the hand-optimised approach either outperforms the generic solver, or it performs on par with it by virtue of being a manual implementation of the generic technique, though intuitively the former is far more likely. On the other hand, generic approaches usually have the advantage of **convenience**.

In understanding how a CFL-R tool can be convenient, we must first examine how it fits into the larger picture of a target application. We can think of a complete CFL-R system as a combination of several stages. Firstly, the input problem must be converted into something that the CFL-R solver can understand and process. In the case of a program analysis like points-to analysis, the input is a program, possibly as source code or an intermediate representation (though, rarely machine code), and this must be converted to a semantic graph which represents the semantics that the analysis cares about, such as a pointer-expression graph [90]. Next, the CFL-R solver runs on its input problem, which produces an output as a set of reachable pairs. The points-to analysis generates a relation between pointer/object variables and the abstract heap objects which they may reference at runtime. Finally, the target application interprets the

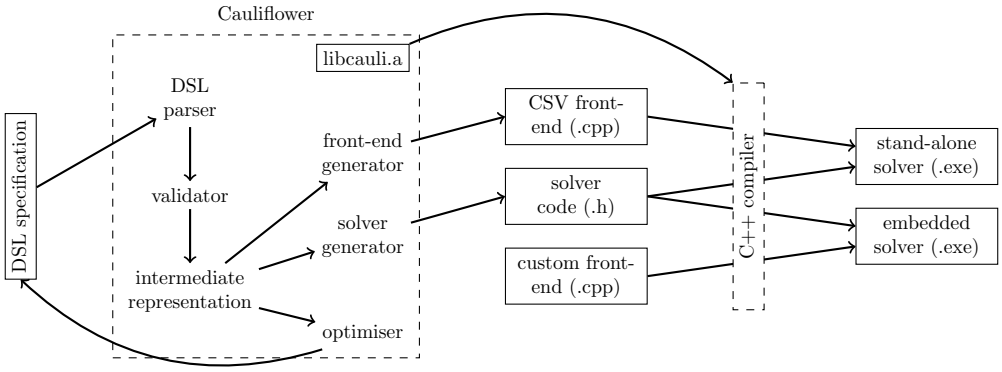


Figure 6.1: Schematic overview of the Cauliflower system, a solver generator for CFL-R problems. This is an updated version of a figure published by Hollingum and Scholz [36].

results, using them in some way, for example a system might use the points-to information as part of a bug-checking tool. Since the first and last stages of such a system design are unavoidably linked to the target application, we can not expect a CFL-R engine to assist with these. The automatic decomposition of input problems (like source-code) into semantic information (like Datalog facts) is an interesting topic that we leave for future work.

Where the CFL-R tool must prove useful is in the middle stage of the pipeline. We can immediately see two important use cases for such a tool, *standalone* and *embedded*. For a standalone application, the generation of input graphs/problems occurs in advance, and the CFL-R solver is run on these problems in isolation. This case is most similar to the kinds of experiments we have performed (See Sections 4.4 and 5.4), though we also foresee use for such an external tool in a development/rapid-prototyping setting. For embedded applications, the input problem is immediately given to the CFL-R solver, which may run sequentially or concurrently with the generation of facts. Ultimately embedded solvers are the most performant approach to using CFL-R, as they avoid the overheads associated with file I/O and forking execution to a standalone solver.

6.2 Cauliflower

We develop a powerful and general tool, catered towards the CFL-R domain, called Cauliflower. The name was chosen as it is the most common english root words with C-F-L-R appearing in order. Cauliflower is a solver-generator for CFL-R problems, which works by synthesising actual solver code from an input specification, written in our custom DSL. Solving code is generated as a header-only library, which makes use of the advanced data structures and

parallel utilities provided by the Cauliflower library (also header-only). Further, a rapid-prototyping focused stand-alone front-end is automatically generated, though the user is free to replace this with a custom front-end in the case of embedded solvers. We present a schematic overview of the Cauliflower system in Figure 6.1.

6.2.1 Semantics

We begin by formalising several semantic extensions to CFL-R. These semantics are intended to be used as primitive operations which the CFL-R grammar may use. Note that CFL-R is a subset of Datalog [1], therefore it is always possible to extend the "semantics" of CFL-R to that of Datalog, however doing this implies loosing the algorithmic superiority of CFL-R, which is known to be $\mathcal{O}\left(n^3\right)$ [56], where Datalog is only guaranteed to be polynomial. For this reason, we show that our semantic extensions are all phraseable via CFL-R gadgets (though somewhat inefficiently) and also can be solved by trivially extending the Melski-Reps algorithm, i.e. they retain the $\mathcal{O}\left(n^3\right)$ runtime for CFL-R problems. We explore the following semantics:

- **Reversal**, for computing paths that traverse the graph in a backwards direction over a backwards language. The notations $A^{\leftarrow}$ or $\overline{A}$ refer to the relevant reversed semantics of A.
- **Templating**, for computing paths over a runtime-dependant index for each relation, i.e. for n-ary relations. The notation $A^{[]}$ refers to the relevant templated semantics of A.
- **Branching**, for computing paths that diverge and reconnect in a structured manner. The notation $A^{\cap}$ refers to the relevant branched semantics for A.
- **Disconnection**, for computing paths over edges/paths that do not exist in the graph. The notation $A^{\neg}$ refers to the relevant disconnected semantics of A.

These semantic extensions capture most of the uses of CFL-R in the research literature, and where they do not, at least make formulations easier (such as demand-drivenness, which benefits from disconnection semantics). It is possible that other semantic extensions could be considered which similarly retain the current complexity of CFL-R algorithms, so we leave this exploration for future work.

6.2.1.1 Reversal

The need for reversal is well understood in CFL-R research [70, 68, 91], though frequently it is defined informally or ad hoc. Reversal is needed for many CFL-R problems, even the running example from Section 1.1.2 requires reversed *pt* paths to compute the aliasing relationship.

Reversal semantics allows for specifications to be simpler and execute efficiently, while avoiding manual efforts, as was needed by Sridharan et al. [70]. Reversal implies simultaneously reversing the language and the path in the CFL-R problem. Language reversal refers to the mechanical translation of the input grammar to a reversed form, which matches input words in the language spelled backwards [38]. Path reversal refers to concatenating the labels of a path in reverse order (alternatively, moving backwards along the arcs). To compute CFL-R reversal, we present a gadget for finding words in the reverse language over edges in the reversed graph:

Definition 22 (CFL-R Reversal)**.** *Given* $\mathcal{L} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, S)$ *and* $\mathcal{T}$*-labelled* $G = (V, E)$*:*

$$\begin{aligned}
\mathcal{P}^{\leftarrow} &= \{A^{\leftarrow} \to {B_k}^{\leftarrow} \dots {B_1}^{\leftarrow} \mid A \to B_1 \dots B_k \in \mathcal{P}\} \\
\mathcal{L}^{\leftarrow} &= (\mathcal{T} \cup \{T^{\leftarrow} \mid T \in \mathcal{T}\}, \mathcal{N} \cup \{N^{\leftarrow} \mid N \in \mathcal{N}\}, \mathcal{P} \cup \mathcal{P}^{\leftarrow}, S) \\
G^{\leftarrow} &= (V, E \cup \{T^{\leftarrow}(v, u) \mid T(u, v) \in E\}),
\end{aligned}$$

the ***reverse-enabled CFL-R solution*** *is,*

$$\textsc{cfl-r}^{\leftarrow}(\mathcal{L}, G) = \textsc{cfl-r}(\mathcal{L}^{\leftarrow}, G^{\leftarrow})$$

Cauliflower avoids expensively computing and recording reverse information by noting the following:

Remark 1.

$$A(u, v) \in \textsc{cfl-r}^{\leftarrow}(\mathcal{L}, G) \Leftrightarrow A^{\leftarrow}(v, u) \in \textsc{cfl-r}^{\leftarrow}(\mathcal{L}, G)$$

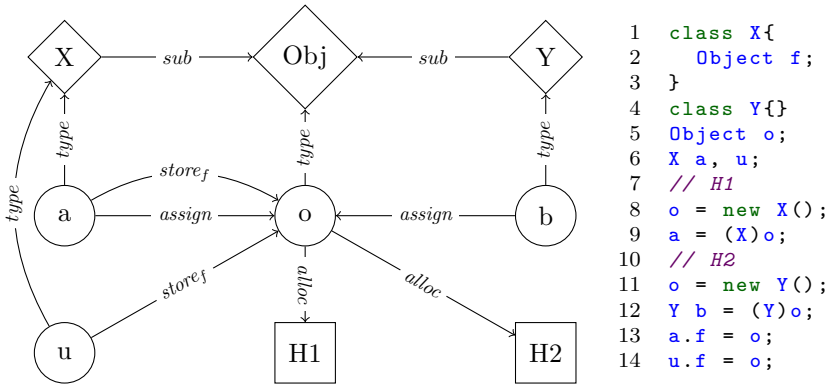
6.2.1.2 Templating

In traditional CFL-R, we specify each rule in advance. To enable field sensitivity for the points-to analysis in Section 1.1.2, we specify a different rule for each potentially matching field:

$$\begin{aligned}
&\textit{points-to} \to \textit{load_f}\ \textit{points-to}\ \overline{\textit{points-to}}\ \textit{store_f}\ \textit{points-to} \\
&\textit{points-to} \to \textit{load_g}\ \textit{points-to}\ \overline{\textit{points-to}}\ \textit{store_g}\ \textit{points-to} \\
&\qquad \dots
\end{aligned}$$

In practice we can limit the rules specified to only those fields we actually encounter, though this information is only known on a per-problem basis, and hence the grammar must vary with the graph in these cases. In a templated problem we allow for meta-rules and meta-edges to be specified, which expand to a series of rules/edges as needed by the specific problem instance, i.e. we write a single meta-grammar and this is used for multiple input graphs.

Templating is a convenience notation which enables the points-to analysis from Figure 1.2 to be field-sensitive, as well as the virtual-dispatch analysis from Section 5.4 to use ternary relations for its parameter index and method




```
1  class X{
2    Object f;
3  }
4  class Y{}
5  Object o;
6  X a, u;
7  // H1
8  o = new X();
9  a = (X)o;
10 // H2
11 o = new Y();
12 Y b = (Y)o;
13 a.f = o;
14 u.f = o;
```

Figure 6.2: Code and assignment-graph for a type-aware alias analysis. Classes are given a diamond-shaped representative vertex. *type* edges encode the "was declared with type" relationship, and *sub* edges indicate an "is a subtype of" relationship.

descriptor edges. The specification writer declares the rules once, encoding fields or call-sites in different templates. Since the templates are only expanded into rules at runtime, the same specification can be re-used on different datasets with different template values.

Like reversal, templating is common in CFL-R research [27], though its definition has implications for the complexity of CFL-R solvers [47]. We abridge a complicated formal definition with this intuitive explanation. A label template L is defined over a template-set S, such that the label L_S is a placeholder for any $s \in S$. When used in a rule, the form L_x implies that whenever an L label appears, its template-set-member shall be named x, which allows different labels to match on their template. In Figure 1.2, the fields form a template-set with members $\{x, y\}$. The label $load_i$ can stand for a load of any one field, and we ensure field-sensitivity by matching the loaded field with the stored field in the rule *points-to* $\rightarrow$ $load_i$ *points-to* $\overline{\textit{points-to}}$ $store_i$ *points-to*.

6.2.1.3 Branching

Branching is a novel and powerful means of combining rules and analyses. A branching path can be understood intuitively as an additional constraint on paths, which allows the user to encode that multiple conditions must be upheld to derive a relationship between two components of the graph. Consider a simple extension to the points-to problem from Section 1.1.2, based on the problem in Figure 6.2. The new graph includes semantic information for types (as vertices drawn with a diamond), a variable's allocated type, and a transitive subtyping relationship between types. We formulate the alias analysis by extending the

grammar of Section 1.1.2 with the rules:

$$\begin{aligned} rsub &\rightarrow sub \mid \varepsilon \\ alias &\rightarrow (\textit{points-to}\ \overline{\textit{points-to}}) \cap (\textit{type}\ rsub\ \overline{\textit{type}}) \\ alias &\rightarrow (\textit{points-to}\ \overline{\textit{points-to}}) \cap (\textit{type}\ \overline{rsub}\ \overline{\textit{type}}) \end{aligned}$$

The new grammar first identifies reflexive subtypes as either direct subtypes or the self-relation. Subsequently, it reports alias only as those variables that refer to the same heap object **and** the type of one is a reflexive subtype of the other. We use the intersection notation here to denote a branch, which we can see in Figure 6.2 as two distinct paths which connect at their source and sink vertices: $\langle \textit{assign}(a,o), \textit{alloc}(o,H1), \overline{\textit{alloc}}(H1,o)\rangle$ is a sequence with its path-word in $\textit{points-to}\ \overline{\textit{points-to}}$, and $\langle \textit{type}(a,X), \overline{rsub}(X,Obj), \overline{\textit{type}}(Obj,o)\rangle$ is in $\textit{type}\ \overline{rsub}\ \overline{\textit{type}}$, hence a and o alias in the type sensitive formulation, whilst a and b do not, despite our simple flow-insensitive formulation assuming both variables point to $H1$ and $H2$.

It is known that context-free languages are not closed under intersection [38], i.e. the language generated by intersecting two context-free languages is, in general, not context-free. In our extended semantics, branching refers to two paths having *the same endpoints*. This is a weaker form of language intersection on graphs. Informally, a branching CFL-R path exists where the, possibly distinct, paths for *every* term in the intersection exist. Formally, let the branching-enabled CFL-R solution be $\textsc{cfl-r}^{\cap}(\mathcal{L}, G)$, then:

$$A(u,v), B(u,v) \in \textsc{cfl-r}(\mathcal{L}, G) \Leftrightarrow (A \cap B)(u,v) \in \textsc{cfl-r}^{\cap}(\mathcal{L}, G).$$

And a branching production is of the form:

$$X \rightarrow Y_{1,1}Y_{1,2} \ldots \cap Y_{2,1} \ldots \cap \ldots$$

To simplify the formalisms of CFL-R intersection, we desire rules of the form $X \rightarrow Y \cap Z$. We define an Intersecting Normal-Form (INF) which extends binary normal-form [43] for branching productions with two components. The following scheme converts a grammar to INF without loss of generality:

- while any rule is of the form $X \rightarrow \ldots \cap Y_{i,1} \ldots Y_{i,k} \cap \ldots$, create the rule $Y_i' \rightarrow Y_{i,1} \ldots Y_{i,k}$ and replace the original rule with $X \rightarrow \ldots \cap Y_i' \cap \ldots$.
- while any rule is of the form $X \rightarrow Y_1 \cap Y_2 \cap \ldots$, create the rule $X' \rightarrow Y_1 \cap Y_2$ and replace the original rule with $X \rightarrow X' \cap \ldots$.
- convert the remaining productions into binary normal-form.

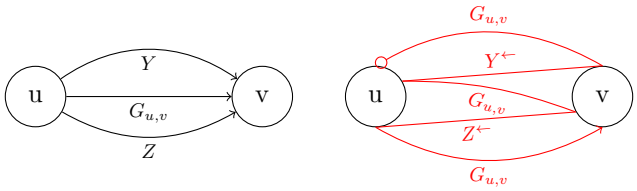
We present a gadget for branching, which makes use of the reverse-enabled CFL-R semantics (Definition 22), and the templating notion, assuming $\mathcal{L}$ is in INF.

Definition 23 (CFL-R Branching)**.** *Given* $\mathcal{L} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, S)$ *and* $\mathcal{T}$*-labelled* $G = (V, E)$*:*

$$\begin{aligned}
\mathcal{T}^{\cap} &= \mathcal{T} \cup \{G_{u,v} \mid u, v \in V\} \\
E^{\cap} &= E \cup \{G_{u,v}(u, v) \mid u, v \in V\} \\
\mathcal{P}^{\cap} &= (\mathcal{P} \setminus \{X \to Y \cap Z \mid X \to Y \cap Z \in \mathcal{P}\}) \cup \\
&\quad \{X \to G_{u,v} Y^{\leftarrow} G_{u,v} Z^{\leftarrow} G_{u,v} \mid X \to Y \cap Z \in \mathcal{P}, u, v \in V\} \\
\mathcal{L}^{\cap} &= (\mathcal{T}^{\cap}, \mathcal{N}, \mathcal{P}^{\cap}, S),
\end{aligned}$$

the ***branching-enabled CFL-R solution*** *is,*

$$\text{CFL-R}^{\cap}(\mathcal{L}, G) = \text{CFL-R}^{\leftarrow}(\mathcal{L}^{\cap}, (V, E^{\cap})) \setminus \{G_{u,v}(u, v) \mid u, v \in V\}$$



Our gadget relies on the traversal of the templated "gate edges" $G_{u,v}$, to ensure a matched path actually begins and ends at the correct vertices. The gate edges can be viewed as a ternary templated label with a template set $T \subseteq V \times V$, i.e. a template over pairs of vertices. Consider the INF rule $X \to Y \cap Z$, as shown above-left. According to the gadget, this rule will be removed from $\mathcal{P}$ and replaced with:

$$X \to G_{u,v}\ Y^{\leftarrow}\ G_{u,v}\ Z^{\leftarrow}\ G_{u,v}$$

Since the gadget adds a gate $G_{u,v}$ for every pair (u, v), we guarantee there exists a path for the branching rule, the actual walk of which is shown above-right. Note that this increases the number of terminals by $\mathcal{O}\left(|V|^2\right)$ (i.e. the instantiation of the gate template) and the number of productions by $\mathcal{O}\left(|\mathcal{P}||V|^2\right)$. The gadget is inefficient to use in practice, i.e., it is designed to demonstrate branching, not compute it. Internally, Cauliflower uses set-intersection operations to explore branched paths.

6.2.1.4 Disconnection

Disconnection, another novel mechanism in CFL-R, allows specification-writers to simplify analysis formalisms. Instead of creating an entirely new analysis for disconnection-problems, we can simply re-use a problem that discovers the connected edges, and write rules that match the *absence* of those connections. To demonstrate, we build a very simple null-dereference analysis by disconnecting the type-aware alias analysis from Figure 6.2. It discovers variables which are loaded-from or stored-to that also do not point to anything. This analysis will make use of branching and reversal semantics. Firstly, observe that

$(store_f\ \ store_f{}^{\leftarrow}) \cap \varepsilon$ matches paths which "begin with a store edge, then walk back along a store edge, *and* traverse the empty path", in other words, it discovers a self-loop at the source of any $store_f$ edge. Further, $alias \cap epsilon$ matches paths between two variables "which point to the same object *and* traverse the empty path", i.e. a self loop from an object to itself if that object points to something. We combine these two observations to write a rule which discovers store sources and load sinks that do not point to anything:

$$null\_deref \rightarrow (store_f\ \ store_f{}^{\leftarrow}) \cap alias^{\neg} \cap \varepsilon$$
$$null\_deref \rightarrow (load_f{}^{\leftarrow}\ \ load_f) \cap alias^{\neg} \cap \varepsilon$$

We avoid having to contrive a "not-alias" analysis, and simply re-use the existing points-to analysis to discover new relations. For the example in Figure 6.2, a is the source of a *store* edge, as is u. since a points to $H1$, it clearly aliases itself, hence it will not cause a $null\_deref$ loop to be created at a. On the other hand u is a store's source, and u does not alias itself (since it points to nothing), so the self-loop will be created there. Hence, our simple analysis discovers the null dereference in the source code.

Cauliflower's disconnection is defined according to the semantics of non-recursive negation [1]. Non-recursive negation implies that the absence of a path in the *finished* solution means the presence of a "negative path" in the disconnection-enabled solution. From a practical perspective, this forbids the use of disconnection in rules where the disconnected term depends on the rule's head (i.e. in cycles).

Definition 24 (CFL-R Disconnection)**.** *Given* $\mathcal{L} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, S)$ *and* $\mathcal{T}$*-labelled* $G = (V, E)$*:*

$$\mathcal{T}^{\neg} = \mathcal{T} \cup \{T^{\neg} \mid T \in \mathcal{T}\}$$
$$\mathcal{N}^{\neg} = \mathcal{N} \cup \{N^{\neg} \mid N \in \mathcal{N}\},$$

the ***disconnection-enabled CFL-R solution*** *is,*

$$\textsc{cfl-r}^{\neg}((\mathcal{T}^{\neg}, \mathcal{N}^{\neg}, \mathcal{P}, S), G) =$$
$$\textsc{cfl-r}(\mathcal{L}, G) \cup \{X^{\neg}(u, v) \mid X(u, v) \notin \textsc{cfl-r}(\mathcal{L}, G)\}$$

6.2.2 CauliDSL

The user interfaces with cauliflower by providing their grammar specifications to it, written in our domain-specific language (DSL), called CauliDSL. Figure 6.4 exemplifies the syntax of CauliDSL on the null-dereference analysis discussed in Section 6.2.1.4. Our DSL syntax borrows from BNF grammars [8]. CauliDSL directives are divided into two groups, **types** are identified by a left arrow `<-`, and **rules** by the right arrow `->`.

Rule declarations define the productions which make up the CFL-R language. Line 21 specifies that the *points-to* relationship exists between nodes joined by a path labelled $\langle assign, points\text{-}to \rangle$. Rules are written in a combination

$$
\begin{aligned}
\mathit{rsub} &\rightarrow \mathit{sub} \\
\mathit{rsub} &\rightarrow \varepsilon \\
\mathit{deref} &\rightarrow (\mathit{store}_f\ \overline{\mathit{store}_f}) \cap \varepsilon && \forall f \in \mathit{fields} \\
\mathit{deref} &\rightarrow (\overline{\mathit{load}_f}\ \mathit{load}_f) \cap \varepsilon && \forall f \in \mathit{fields} \\
\mathit{points\text{-}to} &\rightarrow \mathit{alloc} \\
\mathit{points\text{-}to} &\rightarrow \mathit{assign}\ \mathit{points\text{-}to} \\
\mathit{points\text{-}to} &\rightarrow \mathit{load}_f\ \mathit{alias}\ \mathit{store}_f\ \mathit{points\text{-}to} && \forall f \in \mathit{fields} \\
\mathit{alias} &\rightarrow (\mathit{points\text{-}to}\ \overline{\mathit{points\text{-}to}}) \cap \mathit{type\text{-}compat} \\
\mathit{type\text{-}compat} &\rightarrow \mathit{type}\ \mathit{rsub}\ \overline{\mathit{type}} \\
\mathit{type\text{-}compat} &\rightarrow \overline{\mathit{type\text{-}compat}} \\
\mathit{null\text{-}deref} &\rightarrow \mathit{deref} \cap \mathit{alias}^{\neg}
\end{aligned}
$$

Figure 6.3: A complete listing of the null-dereference analysis' grammar, featuring limited type-sensitivity.

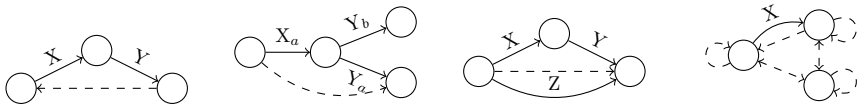
of standard BNF grammar syntax, and Cauliflower-specific notation for reversal (`-`), templating (`[i]`), branching (`&`), and disconnection (`!`). As a convenience, CauliDSL disambiguates associativity of its rules with `(` and `)`, and provides epsilon notation with `~`. The semantic operations align with those discussed in Section 6.2.1, and they are demonstrated in-isolation in Figure 6.5.

A type declaration constrains the source and sink of the edge labels to certain domains, which are partitions of the input vertices. Types are useful both to assist the programmer in writing Cauliflower specifications, via static type-checking, and to optimise the execution plan. Line 2 declares that `alloc` edges connect `V` vertices to `H` vertices (i.e. variables to heap-objects). Therefore, the rule `foo->alloc,alloc` is invalid, since it tries to connect an endpoint from the domain `H` (the sink of `alloc`) to one from `V` (the source of `alloc`). Because of the performance and correctness advantages, type declaration is mandatory in the CauliDSL.

Both the type and the relevant rules are annotated to express template semantics. Line 4 declares that *load* joins `V` vertices to `V` vertices, and have a template over `F`, i.e. it joins variables to variables, and has a different version of itself for each field. The rule on Line 22 indicates a *points-to* relationship exists between the endpoints of a $\langle \mathit{load}_f, \mathit{alias}, \mathit{store}_f, \mathit{points\text{-}to} \rangle$ path only when the instantiated versions of *load* and *store* share the same field. The syntax of templating is deliberately similar to array syntax from the C-family languages, due to the fact that Cauliflower implements field templates via an array of relations.

```
1  // Terminals
2  alloc <- V.H;
3  assign <- V.V;
4  load[F] <- V.V;
5  store[F] <- V.V;
6  type <- V.T;
7  sub <- T.T;
8  // Nonterms
9  rsub <- T.T;
10 deref <- V.V;
11 pt <- V.H;
12 alias <- V.V;
13 type_c <- V.V;
14 null_d <- V.V;
15 // Rules
16 rsub -> sub;
17 rsub -> ~;
18 deref -> (store[f], -store[f]) & ~;
19 deref -> (-load[f], load[f]) & ~;
20 pt -> alloc;
21 pt -> assign, pt;
22 pt -> load[f], alias, store[f], pt;
23 alias -> (pt, -pt) & tc;
24 type_c -> type, rsub, -type;
25 type_c -> -type_c;
26 null_d -> deref & !alias;
```

Figure 6.4: CauliDSL specification for the null-dereference analysis, as shown in Figure 6.3.



(a) `-(X,Y)` (b) `X[f], Y[f]` (c) `(X,Y) & Z` (d) `!X`

Figure 6.5: Cauliflower's semantic extensions. Dashed lines show the paths recognised by the CauliDSL rule-body underneath each figure. This is a reproduction of a figure published in LPAR-21 [36].

6.2.3 Solver Generator

Cauliflower translates the specified CFL-R application into an efficient C++ solver. The execution plan follows the course of the efficient semi-naïve evaluation from [35], and makes alterations where necessary to generate the extended semantics operations for reversal, intersection, negation and templating. Specifically, Cauliflower solvers continually evaluate rules (i.e., search for paths), until no new results can be found. The exact procedure for "evaluating a rule" varies depending on the rule's definition. We demonstrate the code-generation and performance considerations by looking at how DSL instructions cause different codes to be generated.

In the simple case, the rule has the form `A->B,C,D`. The structure of a rule allows us to **bind** the endpoints of most labels, which constrains the search-space when iterating over edges in the data structure, i.e., when a source is fixed (by being connected to another label) then we only iterate over the edges that actually begin at that source. The previous rule produces the pseudo-code listed below (note how the source of C is bound by the sink of B):

for all $(b1, b2) \in B$ **do**
  for all $(b2, c1) \in C$ **do**
    for all $(c1, d1) \in D$ **do**
      $A \leftarrow A \cup \{(b1, d1)\}$

According to the semi-naïve evaluation strategy, each rule has a different expansion procedure according to the choice of "delta relation" (Δ). Further, new information is added to the delta relation, for use in later rule expansions. For this reason, the above code actually generates three rules, such as the following for Δ_B:

for all $(b1, b2) \in \Delta_B$ **do**
  for all $(b2, c1) \in C$ **do**
    for all $(c1, d1) \in D$ **do**
      $\Delta_A \leftarrow (\Delta_A \cup \{(b1, d1)\}) \setminus A$
$A \leftarrow \Delta_A$

Maintaining a high standard of solver performance is key to Cauliflower's design goals. We ensure generated code meets this standard by using efficient, cache-aware data-structures, and instrumenting parallel regions. Cauliflower's data structures are specially designed for relational algebra tasks, such as CFL-R. We use a customised trie-like data-structure [64] to record the edges for a single label (including different structures per template-set-element of a templated label). This structure presents an efficient index, so that projecting the successors/predecessors of a given vertex is very fast. When x is fixed, code such as `for (x,y) in Z` requires only one fast lookup to return the set of all the successors labelled with Z.

A reverse term, such as `A->B,-C,A`, is similarly converted to output code. Thanks to Remark 1, we can simply swap the binding that we use for the reverse term. The above rule binds the C relation in reverse, to produce:

for all $(b1, b2) \in B$ **do**
  for all $(c1, b2) \in C$ **do**
    for all $(c1, a1) \in A$ **do**
      $A \leftarrow A \cup \{(b1, a1)\}$

In `A->B&((C&!D),E)`, binding is similarly used to constrains both endpoints in intersection and the absence of endpoints in negation. Cauliflower does not require that rules are in INF, so the above rule is valid, and would generate:

for all $(b1, b2) \in B$ **do**
  for all $(b1, c1) \in C$ **do**
    if $(b1, c1) \notin D$ **then**
      if $(c1, b2) \in E$ **then**
        $A \leftarrow A \cup \{(b1, a1)\}$

Handling negation efficiently is a non-trivial exercise. The semi-naïve evaluation strategy makes assumptions about the sparsity of the input problem. Since the negation of a sparse relation is always dense (though the converse is not

true), allowing the user to negate clauses indiscriminately can have performance implications. Specifically, a rule such as `A -> !B` will cause the cross product of `B`'s endpoint domains to be added to `A`. As the intention of Cauliflower is to express analyses which are inherently feasible, we treat such unbound negations as an error, i.e., negated relations are only valid when the source and sink are bound by non-negated endpoints. Operationally, the solver must conservatively assume any path could exist unless and until there is a guarantee that no such edge can arise. Consider the rules:

```
A -> !A
Y -> Y, !Z
```

The first rule is invalid, since `A` depends on its own negation, adding an edge to it because it is currently absent in `A` leads to a contradiction of Definition 24. The second rule may be valid, if `Z` does not transitively depend on `Y`, since currently absent `Z` paths will remain absent for the entire computation. Otherwise Cauliflower can make no guarantees about absent `Z` paths, and the rule is invalid.

Templating demands that meta-rules be allowed as a place-holder for multiple similarly-structured rules, which depend on one-or-more templates. The grammar-writer need not predict what values the rule/label might have in the graph, and can avoid manually enumerating all possibilities by annotating the relevant types and rules. As such, Cauliflower treats the template-set as runtime data (like vertices), and repeatedly expands the rule over each element when expanding one templated rule. Note that the shape of a templated rule does not change with the indices, which allows code to be generated for the rule-template in advance, despite the unknown indices. For example, the rule `A[x] -> B[y],C[x]`, will generate code:

for all $y \in$ TEMPL(B) **do** ▷ returns B's template-set
  for all $x \in$ TEMPL(A, C) **do**
    for all $(b1, b2) \in B_y$ **do**
      for all $(b2, c1) \in C_x$ **do**
        $A_x \leftarrow A_x \cup \{(b1, c1)\}$

Note that the outermost loop here is not over a relation, but instead is over a template-set. Cauliflower switches to a coarse-grained parallelism here, where each set-element is examined in parallel.

Outside of each rule, solver code is orchestrated to maximise data locality. This improves the overall performance of the code, by keeping as much relevant data as possible in the cache. Consider the group of rules:

$$A \to B \qquad X \to Y\ Z \qquad C \to B$$

To maximise cache utilisation, Cauliflower identifies where the same relation is being used, and co-locates their rule expansion. The relations will be updated in the order A, C, X, which is superior to their written order A, X, C, as it allows

B to be kept in cache for both reads, instead of risking it being evicted when X is updated. In the case where there are conflicting opportunities for grouping rules, the semi-naïve evaluation strategy breaks ties and forces topologically inferior relations to be evaluated first.

As Chapter 5 discussed, CFL-R specifications are amenable to automatic optimisation. The approach detailed in that chapter was to use a feedback directed approach in order to inspect the execution of a given CFL-R solver on test data, and rewrite the grammar using that information. Cauliflower facilitates this approach by configurations which enable profiling. The generated cauliflower code can be emitted with source instructions that reflect various computations within the solver, such as execution times for evaluating certain rules, and overall runtimes for the solver itself. The cauliflower tool itself is subsequently able to read the profiled logs from its generated solvers, and rewrite input grammars based on that information. In this way, cauliflower can perform the feedback directed optimisation described in Chapter 5 on arbitrary grammars.

6.2.3.1 Parallelism

Cauliflower identifies both coarse-grained and fine-grained opportunities for parallelism. For the above code, fine-grained concurrency is possible by iterating over the B relation in parallel. We make use of the OpenMP [24] directives for defining parallel regions, which further allows Cauliflower solvers to adapt to individual parallel hardware. Further, the Cauliflower data-structures are designed to allow for a special bi-modal parallelism, i.e., they are lock-free, but the structure can only be written to or read from (but not both) in parallel. In fact, due to parallel generation, the above reversal rule generates:

$B^* \leftarrow$ disjoint partitions of B
$A' \leftarrow \emptyset$ ▷ this temporary is thread-local
for all $B' \in B^*$ **in parallel do**
  for all $(b1, b2) \in B$ **do**
    for all $(c1, b2) \in C$ **do**
      for all $(c1, a1) \in A$ **do**
        $A' \leftarrow A' \cup \{(b1, a1)\}$
for each A' **in parallel do**
  $A \leftarrow A \cup A'$

Semi-naïve evaluation allows for bi-modality to be used here, since we can always guarantee that there are no read-write conflicts. In the above code, we wrote to a temporary relation A' in each core, and then collectively updated the real A edges in parallel. Generating code like this avoids reading and writing the same relation in parallel, thus preserving the bi-modality.

6.3 Viability

We now compare the performance of Cauliflower's generated-code to a modern high-performance hand-optimised points-to analysis for Java. Our results show

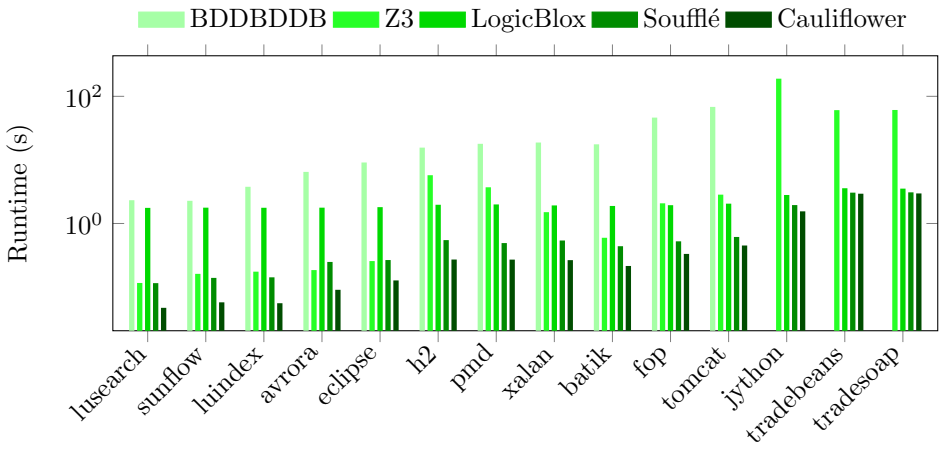


Figure 6.6: Comparison of runtimes for Cauliflower against several commercial and research-grade Datalog engines.

that Cauliflower is viable not merely as a prototyping tool, but competes well with an optimised implementation. Dietrich et al. [27] present an algorithm to solve the context- and flow- insensitive field-sensitive points-to analysis for Java, particularly on large benchmarks. We have examined this implementation closely in Chapter 3; for these experiments we refer to that implementation as GigaScale. A semantically equivalent field-sensitive points-to analysis (i.e. a field sensitive version of Figure 1.2) is encoded in Cauliflower's DSL to produce the Java points-to solver Cauliflower. Note that the Cauliflower specification was carefully designed to maximise performance, in a way that the optimisation techniques described in Chapter 5 can achieve. We confirm offline that Cauliflower is correct, since it produces an identical VarPointsTo relation to the one computed by GigaScale. These experiments are run on a 2.1 GHz Intel® Xeon® CPU E5-2450 with 16 cores and 128 GB RAM under Linux.

6.3.1 Comparison with General Tools

Cauliflower is designed to alleviate programmer efforts whilst maintaining a high standard of execution performance. The first requirement, therefore, is that Cauliflower's solvers are superior to the *other* general purpose solvers. Figure 6.6 presents a time comparison between Cauliflower and four modern Datalog tools, BDDBDDB [82], Z3 [49], LogicBlox [5], and Soufflé [64]. A timeout of 10 minutes is applied, which disqualifies the OpenJDK benchmark. As for the performance, the next best solver, Souffle, is slower than Cauliflower on every benchmark, having an average speedup of 1.89x and a speedup weighted by problem size of 1.34x.

This difference is expected, since the Datalog tools will have two disad-

Table 6.1: Comparison of memory usage for Cauliflower's generated solver against the hand-optimised Gigascale solver described in Chapter 3, together with the size of the input graph and *points-to* relation. Memory usage does not change significantly for parallel/sequential executions of Cauliflower. This is a re-production of a dataset published in [36].

Benchmark	$\|V\|$	$\|$*points-to*$\|$	Gigascale (MB)	Cauliflower (MB)
lusearch	14,994	9,242	98.8	4.5
sunflow	15,957	16,354	96.4	15.8
luindex	17,375	9,677	120.4	32.6
avrora	25,196	21,532	130.8	35.9
eclipse	40,200	21,830	180.8	66.9
h2	56,683	92,038	206.6	82.4
pmd	59,329	60,518	206.0	91.5
xalan	62,758	52,382	223.8	102.9
batik	63,089	45,968	213.2	101.6
fop	83,016	76,615	391.2	138.3
tomcat	110,884	82,424	445.0	181.2
jython	260,034	561,720	708.2	331.5
tradebeans	466,969	696,316	1,533.4	762.6
tradesoap	468,263	698,567	1,529.4	757.8
openjdk	1,963,997	1,570,820,597	3,531.6	60,720.2

vantages against a specialised CFL-R solver. Firstly, Datalog has a constant overhead, based on the creation and management of more general purpose data structures, leading to a high average improvement for Cauliflower. Secondly, a smaller, size-dependant, overhead, due to the slightly less efficient evaluation approaches (Datalog can not make as strong assumptions about execution as Cauliflower). In effect, since the second overhead is smaller, the comparative advantage of Cauliflower shrinks on the larger benchmarks.

6.3.2 Comparison with Specialised Tools

Table 6.1 presents the memory usage of Cauliflower and GigaScale on the DaCapo [13] benchmarks (2009 version). Input datasets are distributed with the GigaScale system, and were originally created using DOOP [16], we use them as-is for these experiments, with minor text-formatting to make them readable by Cauliflower. It also shows the size of the input problem and output `VarPointsTo` relation. GigaScale exhibits more memory overheads as compared with Cauliflower, though these amortise on the largest benchmark, OpenJDK, which we attribute to the use of compression. Importantly, we see that Cauliflower's memory usage correlates roughly with the size of the input problem, though it is slightly better for denser problems (like jython).

Figure 6.7 presents a comparison of runtimes between GigaScale and three versions of Cauliflower, namely (1) a sequential version of the solver, (2) a parallel solver running on 1 core (mostly to demonstrate the overheads of

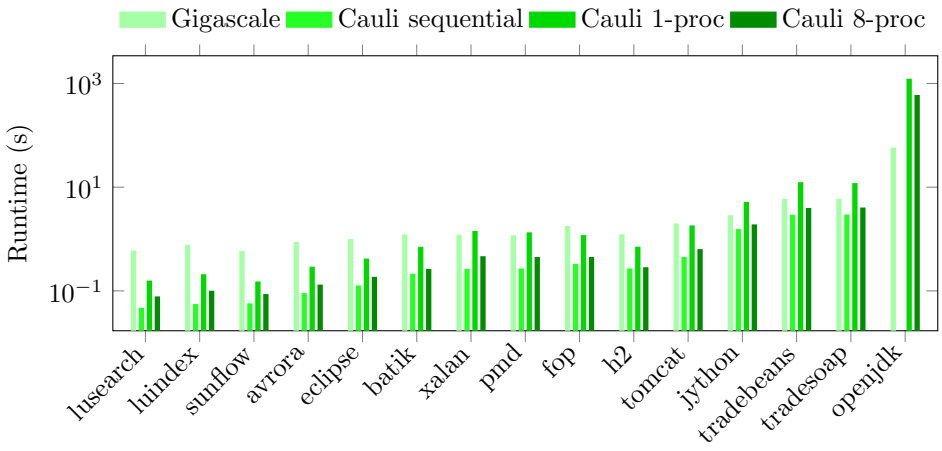


Figure 6.7: Comparison of runtimes for Cauliflower against the high performance Gigascale analysis.

parallelism), and (3) the same parallel solver running on 8 cores. On smaller benchmarks, CAULIFLOWER outperforms GIGASCALE in execution time. We attribute this to overheads which GIGASCALE has, that improve its performance on larger benchmarks. Small parallel executions of CAULIFLOWER are also faster than GIGASCALE, when allocating sufficient processors to CAULIFLOWER (columns Cauli-T 1-proc and Cauli-T 8-proc), though the overheads of parallelism are not justified here, as sequential execution outperforms even the 8-core version. Memory usage also favours CAULIFLOWER for the smaller benchmarks (columns Giga-M and Cauli-M). Dietrich et al. [27] mention compression techniques which account for this difference, i.e. CAULIFLOWER memory scales linearly with problem size, whilst GIGASCALE compression has a high overhead but sub-linear scaling.

6.3.3 Execution Details

We now explore the breakdown of time in finer detail. For each iteration of the algorithm, we chart the amount of time taken and the size of the working set (the new information needed for this iteration) in Figure 6.8. The working-set size is the same in the presence of parallelism, hence only one series is visible in the working-set size plot. Spikes in the execution indicate large infrequent updates to the *VarPointsTo* relation (from the *bridge* relation). Looking at the time disparity between 1 and 8 cores, we see speedup is better for larger working-set sizes (though this difference is hidden visually in a log-scale). We also see that earlier iterations have a greater effect on time than latter iterations; spikes in the time graph are more pronounced on the left side.

Parallelism is used to improve the relative performance of CAULIFLOWER as compared with GIGASCALE On the largest benchmark, GIGASCALE outper-

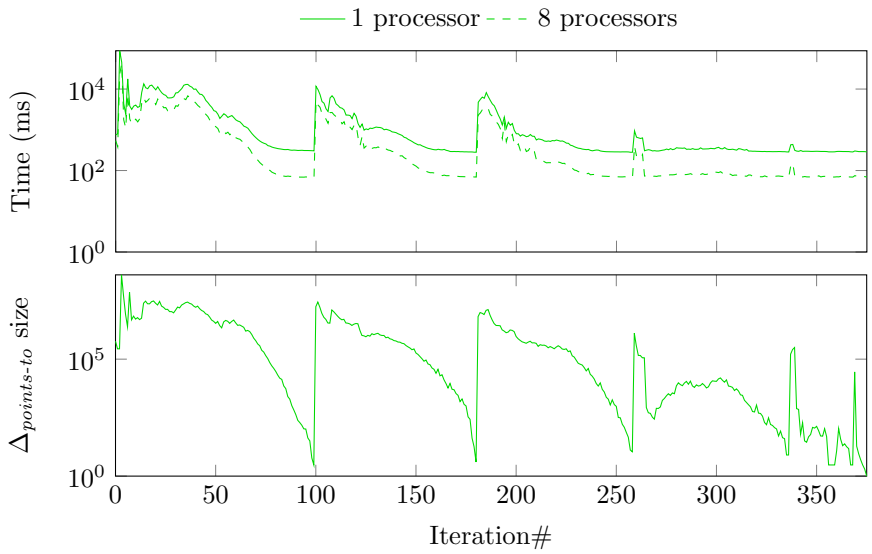


Figure 6.8: Running log over the OpenJDK benchmark, showing the size of the $\Delta_{points\text{-}to}$ set, and, the time needed to expand rules containing that delta relation (on 1 and 8 cores).

forms Cauliflower by a reasonable margin. The GigaScale tool is hand optimised for large benchmarks, so the disparity is understandable, and may be reasonably traded-off with Cauliflower's faster development cycle. In this case, Cauliflower's parallelism narrows the gap. We show the parallel gains with increasing core count in Figure 6.9, on the smallest and largest DaCapo benchmark, as well as the OpenJDK. Parallelism allows the runtime to be cut by over 50% with 8 cores. For OpenJDK, the ≈10 minute runtime is a reasonable result; [27] compares GigaScale to the commodity LogicBlox engine, and report its runtime as over 50 minutes on faster hardware (their own time for GigaScale is 40 seconds to our ≈60 seconds). Unfortunately, according to Amdahl's law, perfect speedup is impossible, since CFL-R problems necessitate much serial computation [56].

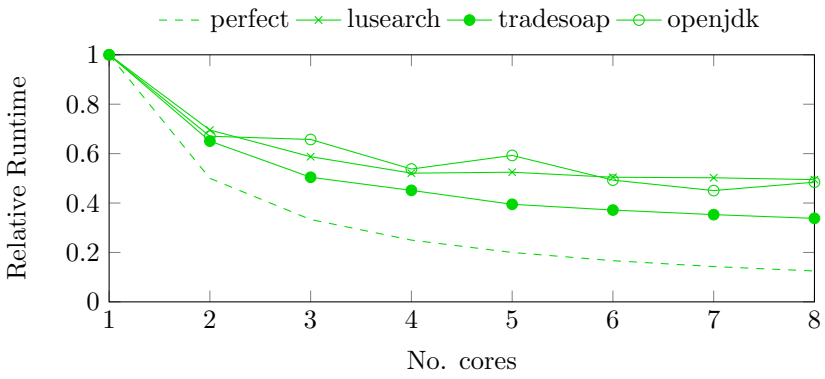


Figure 6.9: Relative runtime due to parallelism (smaller is faster).

Chapter 7

Conclusion

The CFL-R problem is a fundamental formalism that underpins some of the most important analyses in use today. The historic advances of CFL-R are mostly theoretical [47, 58], and have allowed researchers to phrase and reason about the intricacies of program analyses in a coherent manner. Unfortunately, whilst the theory of CFL-R is well established, the practical implementations of the formalism are lacking. In this work we have looked at the key limiting factors that prevent widescale adoption of CFL-R, in the hopes of designing and developing useful tools which researchers and practitioners can apply to their specific problems. In this way, we hope to further the reach of practical CFL-R efforts, so that the potential of CFL-R as a real-world analysis vehicle can be met.

Our efforts to improve the practical performance of CFL-R solvers began by looking in-depth at a real program analysis based on CFL-R. The **Gigascale** analysis, as discussed in Chapter 3, forms the groundwork for our understanding of the lengths to which real solvers go in order to guarantee performance. We saw how the regular subset of an analysis' grammar could be sped up using a more efficient solving vehicle, transitive closure (TC) in this case. In addition to the TC optimisation, the remainder of the CFL-R logic could be phrased as a verification problem, where an over-approximation of the analysis goal (the bridge set) was incrementally verified in a process that was significantly faster than building the set from scratch. We examined how several additional optimisations, including the use of compressed data structures and ordering of the input vertices, were needed to ensure good performance. The result was an analysis that was able to solve the field-sensitive points-to problem on a very large graph in under a minute [27]. Our work on Gigascale indicated to us that the evaluation strategy for CFL-R would be key in generalising the performance improvements.

We next explored the adoption of different evaluation procedures for CFL-R problems. Chapter 4 details our work on improving the algorithms and data-structures used for CFL-R. CFL-R was originally phrased as a means of speeding up Datalog computations [86], so it is interesting that we began by adopting

superior Datalog machinery and applying it to CFL-R. The **semi-naïve** strategy was instrumental here, as it allows us to cut down on redundant computations which traditional CFL-R algorithms, such as the Melski-Reps approach shown in Algorithm 2. Semi-naïve requires the use of newly discovered relations, called the delta set when propagating new information, i.e. one of the new path's edges must be newly discovered. Further, rules are ordered in a strategic way based on dependency information between them, such that we only search for paths according to a rule when we are likely to discover new edges there. The superior evaluation strategy was enhanced by the use of efficient datastructures for CFL-R. **Quadtrees** were chosen as an ideal structure, as they handle the kinds of sparse problems associated with CFL-R.

Following the adoption of improved algorithms, we began to look at means of improving the *logic* of CFL-R problems. We described an **optimisation** strategy for CFL-R specifications in Chapter 5. We examined a formal model describing the costs of evaluating CFL-R logic, based on predicting the discovery of dead-ends (i.e. wasted search effort). The formal model was shown to be accurate for a simple evaluation mechanism, though intuitively it must be effective for any *reasonable* solver. We described a reasonable heuristic technique which made predicting dead-ends discoveries computationally feasible. We applied this modelling in a feedback-directed optimiser for CFL-R logic. The optimiser profiled CFL-R execution, to build the model, then trialled several transformations a-priori, choosing the best one to optimise the input specification with. We looked at reordering, promoting and filtering transformations in our study, though any language preserving transformation may prove useful. The result of this work is a fully automatic logic optimiser, which we implemented as part of a prototype CFL-R solver.

The development of a tool for evaluating arbitrary CFL-R problems is the culmination of this work. We applied the ideas discussed earlier, and developed the **Cauliflower** system, as detailed in Chapter 6. Cauliflower is a solver-generator for CFL-R problems, which generates C++ code for solving any problem which can be described in our system. We design a DSL for CFL-R problems, based on BNF grammars, which includes a simple type system. Cauliflower makes programming CFL-R easier by enabling enhanced semantics, which allow users to encode reversal, templating, branching and disconnection in their problems. In an experimental study, we showed that Cauliflower was superior to more general alternatives (like Datalog), though not as effective as manually developed solutions (like Gigascale). Ultimately, the CFL-R tool Cauliflower can be used to further research and development in the CFL-R context.

In short we have:

- Examined the case of a high performance CFL-R based analysis, whilst developing a fast points-to analysis.
- Detailed the evaluation and data optimisations needed to improve CFL-R algorithms.
- Described techniques to recognise and apply optimisations to the logic of

CFL-R problems.

- Developed a high-performance general-purpose CFL-R solver, capable of solving most problems examined by the research literature.

7.1 Future Work

This work explored the area of CFL-R, with the intention of building viable tools and solutions which will assist in CFL-R research and practice. Whilst our stated goal of achieving a general purpose and highly extensible CFL-R solver was successful, many promising avenues remain which would allow CFL-R tools, specifically Cauliflower, to be even more useful.

We here detail several key advances which we believe would be most beneficial for future CFL-R research to focus on.

The use of Quadtrees as a data structure tailored for CFL-R is promising, though their adoption requires several additional improvements. Firstly, quadtrees were only examined in the context of two-dimensional spaces (i.e. binary relations). Ideally, a full exploration of Quadtrees could be undertaken in the context of *n-trees*, i.e. higher dimensional trees. This would allow the use of quadtrees to phrase higher arity problems, such as the kind found in Datalog, and could be used for the enhanced semantics proposed in Chapter 6. This line of research relates to the idea of experimenting with varying quadtree representation on different platforms. For simplicity, our quadtrees were optimised for a fixed platform (with a 64-bit architecture), it would be interesting to see if the tradeoffs made for that word-size or cache-width could be generalised based on architecture. Further, we hope to explore parallel algorithms for multiplying quadtrees (i.e. parallel relational joins). Whilst quadtrees are superior for sequential evaluations, highly specialised parallel data structures ultimately outperform when given enough computational resources. Given a parallel algorithm for multiplying quadtrees, we expect that CFL-R solvers will become significantly faster in practice.

The evaluation strategy used for the CFL-R research was more effective experimentally than the prior approaches, at least on our benchmarks. Nevertheless, we are interested in applying other algorithms which were developed more recently. Most computation time goes into joining relations (i.e. discovering paths), so this area deserves some focus. Particularly, there is recent work in the Datalog field which proposes joining all relations at once (instead of one pair at a time), which may have practical gains in performance [80]. Ultimately though, there is still work to be done improving the current semi-naïve evaluation approach. Cauliflower conservatively assumes that each relation needs to be indexed from both directions, though an alternate evaluation strategy may obviate the need for maintaining bi-directional indices. In such cases, execution times could be potentially halved.

The advances made by Gigascale were mostly to do with recognising subsets of the logic which could be evaluated with superior algorithms. Ideally, we

would be able to recognise these subsets and apply optimisations to capitalise on them in a mechanical fashion. One such language feature is the presence of Dyck rules in the language. There are superior algorithms for evaluating Dyck grammars [89], though it may be possible to apply the Gigascale idea directly and use an over-approximate/validation approach in a general evaluation. It may be viable to mechanically recognise the *regular* portions of a grammar for which TC algorithms can be used directly, though ideally we could find the kinds of equivalence classes which language rules encode [33]. The above features require significant effort to understand how and when they occur in a CFL-R grammar, and how to capitalise on them when they do occur.

Finally, specifically related to the optimisations of Chapter 5, we wish to examine better optimisation approaches. The optimisation procedure we developed used a very coarse 1-dimensional summary to represent the binary relations used in CFL-R. Since using the actual relation is equivalent to solving CFL-R, we would like to explore how varying the accuracy of the approximation would improve the optimiser. Our approach relies on feedback direction and a cost model to predict good optimisations a-priori, though it is also possible to adapt undirected techniques, such as genetic algorithms, to discover better formulations without a cost model. Further, since any language preserving transformation is a candidate for optimisation, we wish to examine other transformations, such as rule substitution (i.e. the reverse of promotion), and develop techniques for reasoning about their performance. Importantly, the optimisation work is very general, so it is also possible to extend our ideas to the Datalog context. Using feedback-directed optimisation for Datalog specifications could alleviate much of the burden that faces Datalog practitioners, though their formalism is more complicated than CFL-R, so we do not expect the optimisers to be as effective there.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[2] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[3] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 253–262, New York, NY, USA, 1989. ACM.

[4] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[5] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1371–1382, New York, NY, USA, 2015. ACM.

[6] V. L. Arlazarov, E. A. Dinits, M. A. Kronrod, and I. A. Faradzhev. On economical construction of the transitive closure of a directed graph. *Doklady Akademii Nauk SSSR*, 194(3):487, 1970.

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[8] John W Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. *Proceedings of the International Comference on Information Processing, 1959*, 1959.

[9] Gregory V Bard. Accelerating cryptanalysis with the method of four russians. *IACR Cryptology ePrint Archive*, 2006:251, 2006.

[10] Osbert Bastani, Saswat Anand, and Alex Aiken. Specification inference using context-free language reachability. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 553–566. ACM, 2015.

[11] R. Bayer and E.M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[12] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta informatica*, 1(4):290–306, 1972.

[13] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.*, 41(10):169–190, October 2006.

[14] François Bourdoncle. *Efficient chaotic iteration strategies with widenings*, pages 128–141. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[15] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM.

[16] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not.*, 44(10):243–262, October 2009.

[17] Swarat Chaudhuri. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 159–169, New York, NY, USA, 2008. ACM.

[18] Ben-Chung Cheng and Wen-Mei W. Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 57–69, New York, NY, USA, 2000. ACM.

[19] Noam Chomsky and George A Miller. *Introduction to the Formal Analysis of Natural Languages*. Wiley, 1963.

[20] Noam Chomsky and Marcel P Schützenberger. The algebraic theory of context-free languages. *Studies in Logic and the Foundations of Mathematics*, 35:118–161, 1963.

[21] Cristina Cifuentes, Andrew Gross, and Nathan Keynes. Understanding caller-sensitive method vulnerabilities: A class of access control vulnerabilities in the java platform. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2015, pages 7–12, New York, NY, USA, 2015. ACM.

[22] John Cocke. *Programming languages and their compilers*. Courant Institute Math. Sci., New York, NY, USA, 1970.

[23] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251 – 280, 1990. Computational algebraic complexity editorial.

[24] Leonardo Dagum and Rameshm Enon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[25] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. *Estimating the Impact of Scalable Pointer Analysis on Optimization*, pages 260–278. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[26] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Mario Tokoro and Remo Pareschi, editors, *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 77–101, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[27] Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. Giga-scale exhaustive points-to analysis for java in under a minute. In *Proceedings of the 30th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '15, pages 535–551. ACM, 2015.

[28] D. Dolev, S. Even, and R.M. Karp. On the security of ping-pong protocols. *Information and Control*, 55(13):57 – 68, 1982.

[29] A. Cayley Esq. Xxviii. on the theory of the analytical forms called trees. *Philosophical Magazine Series 4*, 13(85):172–176, 1857.

[30] R.A. Finkel and J.L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[31] M. J. Fischer and A. R. Meyer. Boolean matrix multiplication and transitive closure. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 129–131, October 1971.

[32] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. *SIGPLAN Not.*, 42(6):290–299, June 2007.

[33] Ben Hardekopf and Calvin Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis: 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007. Proceedings*, pages 265–280, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[34] Nevin Heintze and David McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Logic in Computer Science, 1997. LICS'97. Proceedings., 12th Annual IEEE Symposium on*, pages 342–351. IEEE, 1997.

[35] Nicholas Hollingum and Bernhard Scholz. Towards a scalable framework for context-free language reachability. In Björn Franke, editor, *Compiler Construction*, volume 9031 of *Lecture Notes in Computer Science*, pages 193–211. Springer Berlin Heidelberg, 2015.

[36] Nicholas Hollingum and Bernhard Scholz. Cauliflower: a solver generator for context-free language reachability. In Thomas Eiter and David Sands, editors, *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pages 171–180. EasyChair, 2017.

[37] Nicholas Hollingum and Bernhard Scholz. Feedback directed optimisations for context-free language reachability. Technical report, Department of Computer Science, The University of Sydney, March 2017.

[38] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Pearson Education International Inc., Upper Saddle River, NJ, USA, international edition, 2003.

[39] T. C. Hu and M. T. Shing. Computation of matrix chain products. part i. *SIAM Journal on Computing*, 11(2):362–373, 1982.

[40] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. *Soufflé: On Synthesis of Program Analyzers*, pages 422–430. Springer International Publishing, Cham, 2016.

[41] Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical report, DTIC Document, 1965.

[42] John Kodumal and Alex Aiken. The set constraint/cfl reachability connection in practice. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 207–218, New York, NY, USA, 2004. ACM.

[43] Martin Lange and Hans Leiß. To cnf or not to cnf? an efficient yet presentable version of the cyk algorithm. *Informatica Didactica*, 8:2008–2010, 2009.

[44] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, 2004.

[45] Ondřej Lhoták and Laurie Hendren. *Scaling Java Points-to Analysis Using Spark*, pages 153–169. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[46] Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. An incremental points-to analysis with cfl-reachability. In Ranjit Jhala and Koen De Bosschere, editors, *Compiler Construction*, volume 7791 of *Lecture Notes in Computer Science*, pages 61–81. Springer Berlin Heidelberg, 2013.

[47] David Melski and Thomas Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248(12):29 – 98, 2000. PEPM'97.

[48] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A gpu implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 107–116, New York, NY, USA, 2012. ACM.

[49] Leonardo Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[50] Ian Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56 – 58, 1971.

[51] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[52] Esko Nuutila. *Efficient transitive closure computation in large digraphs.* PhD thesis, PhD thesis, Helsinki University of Technology, 1995. Acta Polytechnica Scandinavica, Mathematics and Computing in Engineering Series, 1995.

[53] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. *Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis*, pages 165–180. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[54] Alexander Okhotin. Fast parsing for boolean grammars: A generalization of valiants algorithm. In Yuan Gao, Hanlin Lu, Shinnosuke Seki, and Sheng Yu, editors, *Developments in Language Theory*, volume 6224 of *Lecture Notes in Computer Science*, pages 340–351. Springer Berlin Heidelberg, 2010.

[55] Thomas Reps. Shape analysis as a generalized path problem. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '95, pages 1–11, New York, NY, USA, 1995. ACM.

[56] Thomas Reps. On the sequential nature of interprocedural program-analysis problems. *Acta Informatica*, 33(5):739–757, 1996.

[57] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, 1998.

[58] Thomas Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, January 2000.

[59] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[60] John C Reynolds. Automatic computation of data set definitions. IFIP Congress, 1967.

[61] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, January 2003.

[62] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In PeterD. Mosses, Mogens Nielsen, and MichaelI. Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 651–665. Springer Berlin Heidelberg, 1995.

[63] John S. Schlipf. Complexity and undecidability results for logic programming. *Annals of Mathematics and Artificial Intelligence*, 15(3):257–288, 1995.

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

[65] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

[66] Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, volume 6702 of *Lecture Notes in Computer Science*, pages 245–251. Springer Berlin Heidelberg, 2011.

[67] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. *SIGPLAN Not.*, 46(1):17–30, January 2011.

[68] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. *ACM SIGPLAN Notices*, 41(6):387–400, 2006.

[69] Manu Sridharan and Stephen J Fink. The complexity of Andersens analysis in practice. In *Static Analysis*, pages 205–221. Springer, 2009.

[70] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 59–76, New York, NY, USA, 2005. ACM.

[71] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM, 1996.

[72] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[73] David Taniar, Hui Yee Khaw, Haorianto Cokrowijoyo Tjioe, and Eric Pardede. The use of hints in sql-nested query optimization. *Information Sciences*, 177(12):2493 – 2521, 2007.

[74] Alfred Tarski et al. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[75] K. Tuncay Tekle and Yanhong A. Liu. More efficient datalog queries: Subsumptive tabling beats magic sets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 661–672, New York, NY, USA, 2011. ACM.

[76] Leslie G. Valiant. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10(2):308 – 315, 1975.

[77] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings CASCON '99*. IBM Press, 1999.

[78] Sebastiaan J. van Schaik and Oege de Moor. A memory efficient reachability data structure through bit vector compression. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 913–924, New York, NY, USA, 2011. ACM.

[79] Dimitrios Vardoulakis and Olin Shivers. Cfa2: A context-free approach to control-flow analysis. In AndrewD. Gordon, editor, *Programming Languages and Systems*, volume 6012 of *Lecture Notes in Computer Science*, pages 570–589. Springer Berlin Heidelberg, 2010.

[80] Todd L Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481*, 2012.

[81] Stephen Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, January 1962.

[82] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *Programming Languages and Systems: Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005. Proceedings*, pages 97–118, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[83] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *SIGPLAN Not.*, 39(6):131–144, June 2004.

[84] Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings ECOOP'09*. Springer, 2009.

[85] Dacong Yan, Guoqing Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 155–165, New York, NY, USA, 2011. ACM.

[86] Mihalis Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, pages 230–242, New York, NY, USA, 1990. ACM.

[87] Daniel H Younger. Recognition and parsing of context-free languages in time n^3. *Information and control*, 10(2):189–208, 1967.

[88] Hao Yuan and Patrick Eugster. An efficient algorithm for solving the dyck-cfl reachability problem on trees. In Giuseppe Castagna, editor, *Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 175–189. Springer Berlin Heidelberg, 2009.

[89] Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. Fast algorithms for dyck-cfl-reachability with applications to alias analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 435–446, New York, NY, USA, 2013. ACM.

[90] Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. Efficient subcubic alias analysis for c. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 829–845, New York, NY, USA, 2014. ACM.

[91] Xin Zheng and Radu Rugina. Demand-driven alias analysis for c. *SIGPLAN Not.*, 43(1):197–208, January 2008.