THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Compilation Of Bottom-Up Evaluation For A Pure Logic Programming Language

Roger Clayton

This thesis is submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science at the University Of Waikato.

1999 - 2005

Abstract

Abstraction in programming languages is usually achieved at the price of run time efficiency. This thesis presents a compilation scheme for the Starlog logic programming language. In spite of being very abstract, Starlog can be compiled to an efficient executable form. Starlog implements stratified negation and includes logically pure facilities for input and output, aggregation and destructive assignment. The main new work described in this thesis is (1) a bottom-up evaluation technique which is optimised for Starlog programs, (2) a static indexing structure that allows significant compile time optimisation, (3) an intermediate language to represent bottom-up logic programs and (4) an evaluation of automatic data structure selection techniques. It is shown empirically that the performance of compiled Starlog programs can be competitive with that of equivalent hand-coded programs.

Acknowledgements

This thesis was only possible with the wisdom, patience, and optimism of my supervisor, John Cleary.

I would also like to thank the other members of the Starlog project (some of which have come and gone over the years). Thanks to Mark Utting and Bernhard Pfahringer for their support, insightful criticisms and for offering assistance whenever I needed it. Thank you to Don Smith and Lunjin Lu for their invaluable contributions to my early work.

I would like to thank my family, the Hyde family, and my friends for their unwavering support, patience, and optimism throughout this long process.

Most of all, I thank Tracy who has been the light of my life during good times and bad. You will never know how much you mean to me. We did it!

Contents

1 Introduction **1**
1.1 Where in the World is Starlog? 1
1.2 Issues with Logic Programming Languages 3
1.3 The Starlog Approach 6
1.3.1 Bottom-Up Evaluation 6
1.3.2 Negation and Strong Stratification 7
1.4 Programming in Starlog 9
1.4.1 Syntax 9
1.4.2 Input and Output 13
1.4.3 Aggregation 14
1.4.4 Destructive Assignment 15
1.5 Data-Structure-Free Programming 18
1.5.1 Types 20
1.6 Applications of Starlog 20
1.7 Organisation of this Thesis 21

2 Abstract Bottom-up Evaluation **23**
2.1 Semi-Naive Evaluation 24
2.1.1 Example Evaluation 26
2.2 Stratified Semi-Naive Evaluation 26
2.2.1 Example Evaluation 30
2.2.2 Early Failure of Negated Goals 30
2.3 Triggering Evaluation 32
2.3.1 Triggering Transformation 32
2.3.2 Optimising Triggered Programs 34
2.3.3 Example Evaluation 36
2.3.4 Early and Late Evaluation of Negated Goals 37
2.3.5 Optimisation for Non-Trigger Rule Heads 39
2.3.6 Optimisation for Exclusive Trigger Rule Heads 41
2.4 Conclusions 42

3 Indexing **45**
3.1 Starlog's Index Structures and Discrimination Trees 46
3.1.1 Static Labelled Branches 48
3.1.2 Argument Reordering 49
3.1.3 Labelled Boolean Values 51
3.1.4 Labelled Values for Functional Relationships 52
3.1.5 Multiple Argument Orders for Predicates 53

3.2 Example of an Index Structure for Starlog 54
3.3 Operations on Starlog Index Structures 56
3.3.1 Searching for Tuples in Starlog's Indexes 56
3.3.2 Inserting Tuples into Starlog's Indexes 57
3.3.3 Deleting Tuples from Starlog's Indexes 58
3.3.4 Comparisons to Relational Databases 58
3.4 Index Structure Path Definitions 59
3.5 Automatic Construction of Index Structures 61
3.5.1 Efficient Argument Index Ordering for Δ 62
3.5.2 Efficient Argument Index Ordering for Γ 63
3.5.3 Argument Order Definitions 64
3.5.4 Combining Index Structures 68
3.6 Conclusions 71

4 Starlog Data Structure Language (SDSL) 73
4.1 SDSL concepts 74
4.1.1 Comments 74
4.1.2 Index Structure Definition 74
4.1.3 Index Variables 74
4.1.4 Program Variables 75
4.1.5 Code Blocks 76
4.1.6 Negated Code Blocks 77
4.2 SDSL Instructions 78
4.2.1 Dynamic Argument Index Instructions 80
4.2.2 Labelled Branch Instructions 87
4.2.3 Labelled Boolean Value Instructions 91
4.2.4 Built-ins 93
4.3 Summary 95

5 SDSL Triggered Programs 96
5.1 Triggering Evaluation Overview 97
5.2 Adding Facts to Δ 97
5.3 Finding and Deleting Minimum Tuples 98
5.4 Delayed Querying of Negated Goals 102
5.5 Adding True Tuples to Γ 102
5.6 Activation of Rules 102
5.7 Example Triggered SDSL Program 106
5.8 SDSL Triggered Programs with Combined Δ and Γ Sets 109
5.9 Optimisations 110
5.9.1 Removal of Repeated Instructions Within Code Blocks . . 110
5.9.2 Factorisation of Repeated Prefixes Across Code Blocks . . 116
5.9.3 Deletion Completion 119
5.9.4 Removal of Redundant Code Block Nesting 122
5.9.5 Extracting Deterministic Instructions from Code Blocks . 125
5.9.6 Order of Optimisation 125
5.10 Conclusions 129

6 Compilation of SDSL to Java **131**
6.1 Implementation of Index Structures 132
6.2 Index Variables 135
6.3 Program Variables 135
6.4 Code Blocks 135
6.5 Negated Code Blocks 136
6.6 SDSL Instructions 137
6.6.1 Dynamic Argument Index Instructions 137
6.6.2 Labelled Branch Instructions 142
6.6.3 Labelled Boolean Value Instructions 144
6.6.4 Built-ins 144
6.7 Example of Compiled SDSL Code 148
6.8 Conclusions 151

7 Data Structure Selection **152**
7.1 Data Structure Implementations 153
7.2 Problems with Generic Data Structures and Manual Data Structure Selection 155
7.3 Automatic Selection 156
7.3.1 Static Selection Techniques 157
7.3.2 Dynamic Selection Techniques 161
7.3.3 Data Structure Selection at Run Time 168
7.4 Evaluation of Automatic Data Structure Selection 170
7.5 Conclusions 172

8 Conclusions and Future Work **174**
8.1 Comparison of Efficiency 175
8.2 Future Work 178
8.2.1 Detecting Functional Relationships 178
8.2.2 Memory Usage and Garbage Collection 178
8.2.3 Sharing Index Nodes between Δ and Γ 180
8.2.4 Skipping Duplicate Detection 180
8.2.5 Automatic Use of Multiple Indexing Orders 181
8.2.6 Parallel Execution 181
8.3 Summary 182

Appendices **182**

A Starlog Syntax Reference **183**

B Data Structure Implementations **187**
B.1 Empty Data Structure 187
B.2 Unsorted List 187
B.3 Sorted List 188
B.4 Balanced Binary Tree 189
B.5 Hash Table 190
B.6 Flexible Array 191

C Case Studies in Automatic Data Structure Selection **193**
C.1 Hamming Number Program 193
C.2 Prime Number Program 197
C.3 Shortest Path Program 200
C.4 Pascal's Triangle Generation 204
C.5 Transitive Closure 208
C.6 Game of Life 212
C.7 N-Queens 217

Bibliography **218**

List of Figures

1.1 Example stratification priorities and stratification order. 10
1.2 Starlog prime number program. 12
1.3 Destructive assignment program. 16
1.4 Comparison between Starlog data structures and other representations. 19
1.5 Overview of the Starlog compilation pipeline. 22

2.1 Semi-Naive Evaluation of positive programs. 25
2.2 Semi-Naive Evaluation of an example path finding program. . . . 27
2.3 Stratified Semi-Naive Evaluation. 29
2.4 Modification to Stratified Semi-Naive Evaluation to optimise strongly stratified programs. 30
2.5 Stratified Semi-Naive Evaluation of an example path finding program. 31
2.6 Modification to Semi-Naive Evaluation for early failure of negated goals. 32
2.7 Early failure for an example program. 33
2.8 Adding triggers to rules for (a) the general case and (b) an example rule (where triggers are bold positive goals). 34
2.9 Modification to Semi-Naive Evaluation to evaluate triggered rules. 34
2.10 Removing redundant rules based on the satisfaction order of positive goals. 36
2.11 Triggered version of the path finding program from Figure 2.5 with trigger goals in bold. 37
2.12 Modifications to Triggering Evaluation for early and late evaluation of negated goals. (Includes early failure for negated goals that are not definitely stratified before or after the trigger.) . . . 38
2.13 Example uses of early/late evaluation and early failure of negated goals based on the stratification order of goals. 39
2.14 Example of Non-trigger and Exclusive Trigger Heads optimisation. 43
2.15 Triggering Evaluation with all optimisations. 44

3.1 Specification of index structures in Starlog's compilation pipeline. 46
3.2 Example of indexing tuples in a discrimination tree. 48
3.3 Two index structure instances with different argument orderings. 50
3.4 Labelled branch optimisation where p/3 is the default predicate. 51
3.5 Using labelled boolean values to distinguish tuples subsumed in an index. 52
3.6 Example program and a possible index instance. 54

3.7 Index schema. 55
3.8 BNF grammar of index path definitions. 59
3.9 Path definitions corresponding to index structure in Figure 3.6 . 61
3.10 Finding the minimum stratified element using two different argument orders. 63
3.11 An algorithm to find efficient argument order definitions. 66
3.12 Example of generating argument order definitions. (Triggers are shown in bold). 67
3.13 Merging Indexes Algorithm. 69
3.14 Combining four example index structures. 70

4.1 Use of SDSL in Starlog's compilation pipeline. 73
4.2 Example use of index variables \$0 - \$3 in a Starlog index. 75
4.3 Semantics of SDSL code blocks in Prolog. 76
4.4 Example code blocks and their processing sequence. 77
4.5 Semantics of SDSL negated code blocks in Prolog. 77
4.6 Example index structure instance for demonstrating SDSL instructions. 80

5.1 SDSL code generation and optimisation in Starlog's compilation pipeline. 96
5.2 Finding and deleting the minimum element in Δ using SDSL. . . 100
5.3 Example Starlog rule and equivalent SDSL code. 105
5.4 Example Starlog and equivalent SDSL program.. 108
5.5 Index structure schema used by example program in Figure 5.4. . 109
5.6 Example Starlog and equivalent unoptimised SDSL program with all index nodes shared between Δ and Γ. 111
5.7 Index structure used by example program in Figure 5.6. 112
5.8 Logical definition of the non-interference operator. 114
5.9 Optimisations to remove repeated SDSL instruction within code blocks where $\mathbf{S_i}$ represents zero or more instructions or nested code blocks, and θ represents a set of variable bindings. 115
5.10 Example SDSL program after the removal of repeated instructions.117
5.11 Factorisation of identical prefixes of code blocks in an SDSL programs where **X** represents zero or more code blocks of any type. 118
5.12 Factorisation of common prefixes for non-identical instructions where **X** represents zero or more code blocks of any type. 119
5.13 Example SDSL program after common prefix factorisation. . . . 120
5.14 Deletion Completion optimsation. 121
5.15 Example SDSL program after Deletion Completion. 123
5.16 Rules to remove redundant code block nesting where $\mathbf{T_0}$ and $\mathbf{T_1}$ are sequences of at least one instruction and/or code blocks. . . . 123
5.17 Example SDSL program after redundant code nesting has been removed. 124
5.18 Extraction of deterministic instructions from the beginning of code blocks. 125
5.19 Example program after all optimisations have been applied. . . . 126
5.20 Flow diagram showing one order that optimisations may be applied.128

6.1 Java code generation in Starlog's compilation pipeline. 131

6.2 The Node interface. 133
6.3 Example index structure and Java implementation. 134
6.4 The NodeIterator interface. 138
6.5 The BuiltinIterator interface. 147

7.1 Data structure selection in Starlog's compilation pipeline. 152
7.2 Formula for selecting a data structure using static cost analysis. . 160
7.3 Formula for selecting a data structure using data gathered from a single run. 165
7.4 Example records used in regression analysis for two indexes. . . . 167

A.1 BNF grammar of Starlog. 186

C.1 Hamming number program. 194
C.2 Hamming number program performance. 195
C.3 Prime number program. 197
C.4 Index structure schema used for the prime number program. . . . 198
C.5 Prime number program performance. 199
C.6 Index structure schema used for the shortest path program. . . . 201
C.7 Shortest path program. 201
C.8 Shortest path program performance (random graph). 202
C.9 Shortest path program performance (chain graph). 203
C.10 Index structure schema used for the Pascal's triangle generation program. 205
C.11 Pascal's triangle program. 206
C.12 Pascal's triangle program performance. 207
C.13 Index structure schema used for the transitive closure program. . 209
C.14 Transitive closure program. 209
C.15 Transitive closure program performance (random graph). 210
C.16 Transitive closure program performance (chain graph). 211
C.17 Index structure schema used for the game of life. 213
C.18 Game of life program. 214
C.19 Game of life program performance (rabbit pattern). 215
C.20 Game of life program performance (traffic light pattern). 216
C.21 Index structure schema used for the N-queens program. 218
C.22 N-Queens program source for 6x6 board. 219
C.23 N-Queens program performance. 220

List of Tables

4.1 SDSL instruction summary table. 79
4.2 SDSL built-in operations. 94

5.1 Performance comparisons for optimised and unoptimised SDSL programs. 129

7.1 Available data structure implementations and a summary of their time complexities. 154
7.2 Sets of instructions and data structures. 158
7.3 Instruction costs used for static cost analysis. 160
7.4 Instruction cost formula using random data for cost analysis after a single run. 163
7.5 Amendments to instruction cost formula in Table 7.4 for ascending data sets. 163
7.6 Amendments to instruction cost formula in Table 7.4 for descending data sets. 163

8.1 Comparison between Starlog programs and similar hand-coded Java programs. 177

Chapter 1

Introduction

This thesis describes the process of compiling Starlog programs. Starlog [22, 23, 24, 71] is a general-purpose logic programming language designed to overcome some of the problems inherent in other logic programming languages.

Compilation is necessary for Starlog to evolve beyond the status of a "toy" language. Interpreters which are capable of running Starlog programs have existed since 1993 and have shown Starlog to be an effective language for rapid prototyping and implementation of reactive systems. Many students have successfully used Starlog to implement a variety of course related programs. However, because the implementations of the interpreters have never been focused on efficiency, all but the simplest Starlog programs are excruciatingly slow to execute. Therefore the aim of this thesis is to show that compilation of Starlog programs is not only possible, but can be made efficient so that the potential of Starlog may be realised.

This chapter initially describes research relevant to the compilation of Starlog and discusses problems which plague traditional approaches to logic programming. With reference to these problems, the Starlog language is presented as an alternative approach. The data-structure-free style of programming is introduced with comments on its expressiveness and its advantages for logic programs. The final section gives an overview of the remainder of the thesis and of the compilation process.

1.1 Where in the World is Starlog?

This section discusses research fields relevant to the compilation of Starlog. The concepts that are briefly discussed here are revisited in greater detail throughout this thesis. The term "world" in the title of this section refers to the "world" of databases, programming languages and their compilers.

Logic programming has been a recognised field of computer science for more than 20 years. The ideas behind logic programming were laid out in Kowalski's seminal papers [66] and [40], and then reviewed later by the work of Lloyd in [68]. (For an in-depth account of the history of Prolog the reader is referred to [28].) In the earlier works the concept of *bottom-up evaluation* (also known as forward-chaining [39]) is closely connected to the theories of logic programming. However the majority of logic programming implementations use the alternative *top-down evaluation* technique (known also as backward-chaining [39]). This is true of Prolog [82, 26], Mercury [56], Godel [15], XSB [96] and Lygon [116].

In contrast, Starlog uses the bottom-up evaluation technique. Although bottom-up evaluation of logic programs has a reasonable depth of research (e.g. [85, 88, 10, 86, 20, 90, 84, 7]), complete implementations of bottom-up languages are rare. Logic programming languages which evaluate programs bottom-up are obscure, but include Tokio [44], Chronolog (whose evaluation of finite-proportions of relations approximates bottom-up evaluation) [112], Templog [1], and OIL [117]. With the exception of the latter, these languages use temporal logic to control the execution of programs by way of "next", "always" and "sometimes" operators. In this respect Starlog differs from most bottom-up logic programming languages since it does not support these operators.

Instead of temporal operators, the execution of Starlog programs is controlled through program stratification. Stratification has been researched and used extensively in the field of deductive databases [68, 39, 30]. The Aditi, CORAL, LogicBase, DECLARE and Sunburst deductive databases all use various forms of stratification to control bottom-up evaluation [91]. However implementors of deductive databases have different goals than implementors of logic programming languages. In general, deductive databases are focused on efficient query processing [39] rather than exhaustive program evaluation. As a result, research into bottom-up evaluation of stratified programs has been somewhat neglected by the deductive database community in favour of query transformations and optimisation. Also, deductive databases favour persistent (or disk-based) data stores (sometimes with a client/server architecture [89]) instead of in-memory storage, and optimise their evaluation techniques accordingly [91].

Since many deductive databases use the Datalog query language [39, 30] to formalise programs, it is worth pointing out the similarities and differences between Datalog and Starlog. Both Datalog and Starlog programs consist of rules based on Horn clauses [39]. However Starlog programs allow a greater variety of functions to be used within rules – including non-deterministic functions and arithmetic operations. Like Datalog, the arguments of predicates in Starlog programs are either variables or primitive types, rather than the arbitrary terms that are possible in predicate logic. Datalog has been extended to allow both well-founded negation and stratified negation [48, 51] whereas Starlog only permits stratified negation.

The process of building compilers for imperative and procedural languages is well understood (see [2]). Techniques for lexical and syntax analysis already exist which can be used for Starlog programs. However compilation of more abstract languages (such as functional or logic programming languages) require some analysis tools, transformations and optimisations different to those of imperative and procedural languages. For example, the Mercury compiler requires determinism analysis [57] and performs complex transformations involving continuations to be compiled to C [59]. Also, like many other logic and functional programming languages, Mercury implements tail-recursion optimisations to transform recursion into iteration [59]. Compilers for abstract programming languages sometimes target *intermediate languages* [2] rather than an executable language to simplify transformations. For example, the C programming language is considered as an intermediate language for Mercury. WAM code [3] is an intermediate language used by some implementations of Prolog. However it appears there are no intermediate languages specifically designed to represent bottom-up evaluated programs (such as Starlog programs).

The possibility of automatic data structure selection by a compiler has also been proposed. In Tarjan's Turing Award interview he stated that,

> "It would be wonderful in the long run to have some kind of supercompiler that would select, off-the-shelf, the appropriate data structure to plug in to implement very high-level quasi-algorithmic specifications. Ultimately, things have to go in this direction." [43]

Although a few compilers attempt some type of automatic data structure selection (see [70, 97, 9, 13] for a selection) it is still rare, and its effectiveness is virtually unknown.

Due to the diverse issues explored in this thesis, no further background is given here. Instead, additional research relevant to this thesis is described in each chapter. In the next section we discuss problems faced by the designers of logic programming languages prior to the introduction of Starlog.

1.2 Issues with Logic Programming Languages

The design of a logic programming language is constrained by two factors. On the one hand there is the desire to give logic programs a declarative and abstract semantics which allows programmers to be oblivious to the underlying mechanics of their programs. To this end, declarative languages define program specifications rather than a set of instructions [31, 26, 49]. Languages which are abstract and declarative offer several advantages for both programmers and implementors of languages. Declarative languages are usually easier to learn since programs can be interpreted in a simplified, abstract form. Abstract interpretation of declarative programs can be used by programmers to prove programs correct and by implementors of declarative languages to prove program transformations and optimisations correct. The second constraining factor is that logic programming languages seek run time efficiency in order to be practical and competitive with other language paradigms. Historically these two constraints have conflicted with each other such that logic programming languages which focus on one of these features have difficulty achieving the other. To demonstrate this and other points in this chapter we consider two successful logic programming languages: Prolog and Mercury.

Prolog is the most famous logic programming language in use today and it is assumed that it is familiar to the reader. (In this section Prolog is discussed in general terms without reference to any particular implementation.) Pure Prolog [11] was designed as an executable language based on Horn clauses [31]. Because of pure Prolog's declarative semantics, programmers do not need to know any details of SLD (linear derivation with selection function) resolution to write correct programs[1] if they are familiar with Horn clauses and predicate logic. However to improve the efficiency of Prolog programs requires understanding the order that clauses and conditions within clauses are queried. To further improve efficiency the *cut* operator (`!`) can be used to limit searching [33].

[1]Note that Prolog programs which enter infinite loops may be correct but have prohibitive efficiency problems.

Few logic programming constructs have been so widely disparaged and yet so universally used as the cut operator [96]. One negative aspect of cuts is that programs which use them lose their declarative semantics since cuts are non-logical operators (sometimes referred to as *extra-logical* operators) that can not be expressed in Horn clauses [33]. Instead, the effect of a cut can only be interpreted when clauses are considered as sequences of procedures [65, 26]. The effect of adding cuts to programs demonstrates the conflict between having a declarative semantics and striving for efficiency.

In response to the inefficient implementations of logic programming languages produced in the mid 1980's, the Mercury logic programming language was developed with the key goal of high performance [58]. Mercury programs are written in a declarative semantics and, unlike Prolog, there are no non-logical operators [102]. However to achieve high performance programmers must specify some low-level details of their programs – namely the types, modes and determinism of all predicates [56]. Such declarations have a positive effect on the software development process for large applications since many bugs that would otherwise slip through are caught by the compiler [29, 102] and this is respected as a design decision. However these declarations expose programmers to some of the more complex implementation details of their programs. For example, asking programmers who are only familiar with predicate logic to declare their predicates deterministic, semi-deterministic, multi-deterministic or non-deterministic (see [56] for definitions of each) requires them to gain an additional level of understanding. As such, although more efficient than pure Prolog, Mercury is not as abstract or expressive [29] and still requires programmers to understand more than first order logic and predicate calculus to write correct programs.

Another problem faced by implementors of logic programming languages concerns negation. Negation-as-failure [77, 68, 82, 65], which is a commonly used operator in logic programming languages, differs from negation in classical logic in two ways.

The use of negation-as-failure in logic programs breaks the declarative semantics since the order that variables are bound in a clause can be the difference between negated queries succeeding or failing. The following Prolog program demonstrates this problem.

```
soln(0).
soln(3).

p :- X=2, \+ soln(X).
q :- \+ soln(X), X=2.
```

In this program querying p will result in 'yes' whereas q answers 'no'. q fails because X is unbound when expressions are evaluated left to right. This is probably not what was intended by the programmer. To avoid such problems, Prolog manuals recommend that negation-as-failure be used only on ground terms given that programs are interpreted left to right [65], although this is a convention and is not enforced by implementations.

However there are also problems with negation-as-failure when negated queries are ground. In classical logic, queries are either provable, refutable, or undecidable [47]. However typical logic programming languages evaluate queries to either 'yes' or 'no' by applying the *closed world assumption* [114, 68] where all ground atoms that do not follow from the facts in the program are assumed false. Negated queries in a closed world system answer 'no' to any query which does not succeed. This includes negated goals which are undecidable. Both Prolog and Mercury assume a closed world when evaluating negated goals and are insufficient when failure is not finite or when programs are non-monotonic [68].

The use of input and output in logic programs is another issue of contention. These facilities have never really fitted into the structure of logic programs in spite of the prevalence of input and output operators in other programming paradigms. Like the cut operator, adding non-logical operations to perform input and output in Prolog programs breaks the declarative semantics since the details of SLD resolution must be known to predict when each operation takes effect. The Mercury approach has a much better declarative base where input and output is achieved by passing the previous and next states of the external world as parameters to clauses [29]. Although the extra parameters can be hidden in the Mercury source code using DCG notation [57] the extra parameters may still be passed through intermediate clauses that do not directly affect the state of the world. For example, the following Mercury program fragment passes variables representing the state of the world through the `sum_of_squares/4` predicate even though this predicate does not use input or output directly. (The syntax and predicate names used in this example are based on an example in [57]).

```
main(State0, State) :- sum_of_squares(10, X, State0, State1),
                       write_string('' Sum: '', State1, State2),
                       write_int(X, State2, State).

sum_of_squares(0, 0, State, State).
sum_of_squares(X, Y, State0, State) :- X > 0, square(X, Z, State0, State1),
                                       X1 is X-1,
                                       sum_of_squares(X1, Z1, State1, State),
                                       Y is Z+Z1.

square(X, Z, State0, State) :- Z is X*X,
                               write_string('' Square: '', State0, State1),
                               write_int(Z, State1, State).
```

Although logically sound, passing arguments transparently through predicates makes programs more difficult to interpret by programmers and, if approached naively by the compiler, the extra storage and unification operations required can add run time overhead.

Logic programming languages have trouble representing destructive assignment while remaining logically pure. This is because classical logic has *logical omniscience* [4] and can not model an environment where previously true facts are modified or removed. The addition of non-monotonic operators `assert` and `retract` to Prolog which allows for destructive assignment has been met with dissatisfaction amongst logic programmers since programs that use such operators are no longer declarative (since they do not respect the commutativity of logical conjunctions [4]) and are not semantically well behaved [37] making proving programs correct more difficult. However it can not be denied that destructive assignment is an attractive feature for programming languages. Among the several reasons given in [55] for the inclusion of destructive assignment in logic programming languages, improved efficiency and easy translation of conventional algorithms to logic programs are probably the most significant. Mercury is capable of destructive assignment of relations through its extended mode system [102]. As with most features that improve efficiency in Mercury, the use of destructive assignment must be explicitly specified by programmers.

Given these issues, designing logic programming languages which are logically pure and efficient is difficult. In this thesis we explain how the logic programming language *Starlog* solves many of these problems and give the techniques necessary to compile Starlog into an efficient executable form. We do not claim that Starlog is better or worse than existing languages (indeed its potential is still unknown) however its approach to program evaluation is undeniably different from other logic programming languages.

1.3 The Starlog Approach

The increased use of non-logical operators in logic programs prompted the design of the Starlog language in 1988. Starlog was designed from the ground up as a general-purpose programming language with a logically pure framework that would also support facilities common to most other programming languages. This section describes the core concepts behind Starlog which allow these facilities. The following section gives the specifics of the Starlog language and demonstrates how Starlog can implement features which are problematic in other logic programming languages.

Since its inception the Starlog language has evolved in response to advancement in theory or for pragmatic reasons. Consequently the specification of the language given in this thesis may differ from that in previous works such as [71, 24, 23, 22]. Although some facilities have been generalised in the latest version (i.e. current versions support multiple stratifiable arguments – see below), other facilities have been removed for simplicity (i.e. the use of top-down clauses in conjunction with bottom-up rules is considered beyond the scope of this thesis). Further infomation about the current state of the Starlog project is available from the Starlog web page (www.cs.waikato.ac.nz/research/starlog).

1.3.1 Bottom-Up Evaluation

The mechanism which gives Starlog such a workable framework is the use of bottom-up evaluation with stratified programs. Bottom-up evaluation [66] is a more general term for the fixed-point evaluation described in [68] where the fixed-point can be transfinite. Bottom-up evaluation is a data-driven technique rather than goal-oriented approach [91]. That is, evaluation proceeds by automatically deriving new facts (referred to as *tuples*) in a database from rules without the need for top-level queries. Bottom-up evaluation is an efficient technique when programs have considerably more facts than rules [62] and it has been argued that bottom-up logic programs are both clearer and easier to analyse, both for correctness and complexity, than classical pseudo-code presentations [75]. In spite of such benefits, the use of bottom-up evaluation is still rare in the field of logic programming.

An abstract definition of bottom-up evaluation can be given through the use of the *immediate consequence operator* T_P. The immediate consequence operator T_P associated to a logic program P is a function that points out which consequences are deduced from a given set of tuples by the program P in a single inference step [6]. By repeatedly applying the T_P operator to a set of tuples, a model of all the true tuples is constructed.

During bottom-up evaluation there is the potential for many redundant tuples to be derived which can cripple the efficiency of these systems [88, 39]. In response to this problem, many bottom-up applications apply the *magic sets* and *magic templates* transformations [91] to their rule systems that allow their evaluation to be goal-directed when it derives tuples. This permits queries to be evaluated on demand and automatically tables previous results. These transformations add a *magic predicate* to each rule to restrict when rules are eligible for evaluation [62]. Examples of applications where the magic sets or magic templates transformations are fundamental to the evaluation of programs include the CORAL [87] and Aditi [107] deductive databases and the OIL bottom-up logic programming language [117].

The magic sets and magic templates transformations are not applied to Starlog programs for two reasons. The first is that the purpose of Starlog (as a general-purpose programming language) is to execute programs rather than answer queries. As such, Starlog programmers are free to define programs which can produce redundant tuples in the same way that programs in other languages can define infinite loops or add superfluous code which makes programs inefficient. Conversely, fastidious Starlog programmers have the power to write programs that do not produce redundant tuples. (This may involve writing rules which include the equivalent of magic predicates.) The second reason to avoid these transformations is that the introduction of goal-oriented behaviour reorders the production of tuples. For negated goals to be evaluated correctly Starlog programs are stratified (as described in the next section). The application of the magic sets transformation to a stratified programs does not preserve the stratification order in general (although the problem of evaluating unstratified magic programs has received considerable attention) [61, 88]. In Starlog we prioritise program control using a stratification order over goal-directed evaluation. Thus we consider the automatic application of the magic sets transformation unsuitable for Starlog.

1.3.2 Negation and Strong Stratification

To perform negation according to the well-founded semantics, Starlog programs are *stratified* [91]. That is, new tuples are produced by a program in a predetermined order. If a tuple is not produced at its appropriate position in the stratification order then it will never be produced. Using this property the evaluation of any negated goal becomes decidable in the negated goal's stratum.

Programs which are stratified ensure that there is no recursion through negation [88]. This is achieved when the head of any rule is later in the stratification order than (or "stratified after") the rule's negated goals. During evaluation of a rule, a rule's negated goals are decidable when the head is produced. This is because the heads of all rules are produced in stratification order and the negated goals in a rule are stratified before the head.

Note that not all programs that contain negation are stratified. Conversely, for those programs that can be stratified there is often more than one valid stratification order. However the choice of the stratification order will not affect which tuples are produced, but may changed the order that they are generated.

In the deductive database and logic programming literature different types of stratification have been explored (see [91] and [83] for a selection). Starlog programs are modularly stratified [88]. In other words, the stratification order between rule heads can depend on the bindings of variables, but only when the body of each rule is true. However we also specify that Starlog programs are *strongly stratified* [101]. That is, Starlog programs do not allow *any* uncontrolled recursion in rules by ensuring all rule heads are stratified after all positive and negative goals in the rule's body. (This is a similar to requiring all rules to be Y-stratified given the definitions of XY-stratification in [120].) To illustrate, although the following rule may be modularly stratified (because there is no recursion through negated goals) it is not strongly stratified because the head is identical to the positive body goal and therefore belongs in the same stratum (we use Prolog syntax here for simplicity).

```
p(N) :- p(N).                                                  (1)
```

However this next rule *is* strongly stratified if q/1 tuples are stratified using the natural ordering of their numeric arguments.

```
q(M) :- q(N), M is N+1.                                        (2)
```

To determine if a program is strongly stratified we first generate axioms describing the stratification order between the heads of rules and their bodies. In this thesis the notation for the binary stratification order operators are $A \ll B$ and $B \gg A$ where term A is stratified before B, and $C \underline{\ll} D$ and $D \underline{\gg} C$ denotes that C belongs in either a stratum before D or the same stratum as D. The stratification axoims generated for a program that contains rules of the form H:-$B_0, ...B_n$ are $B \ll H$, where B represents any goal in the rule. A program is strongly stratified if the set of all stratification axioms does not contain any contradications. That is, when all axioms are considered as transitive relations, there are no cases where $S \ll T$ and $T \ll S$, or where $U \ll U$. It can be seen that rule (1) will fail this test since $\mathtt{q(N)} \ll \mathtt{q(N)}$. To avoid such contradiction it is often possible to infer conditions from the rules when generating stratification axioms. For example, given rule (2), the stratification order between q/1 tuples can be defined as $\mathtt{q(N)} \ll \mathtt{q(M)} \Leftarrow \mathtt{N} < \mathtt{M}$ because the relationship $\mathtt{M}$ **is** $\mathtt{N} + 1$ implies that $\mathtt{N} < \mathtt{M}$. Although some investigation into automatic checking and inferring valid stratification orders from programs has begun, it is a topic that requires more attention. However it is considered beyond the scope of this thesis since it is not a requirement for compilation.

Strongly stratified programs reduce the opportunities for duplicate tuples to be produced. This is because it is impossible to define a recursive rule where all arguments are invariant since the head must always be stratified later than all body goals. When describing the evaluation of strongly stratified programs, it will be seen that this property leads to significant optimisation.

Since negated goals must be stratified before the head of the rule, any arguments which decide the stratification order of a negated goal must be ground before it can be satisfied. This means that a negated goal's arguments which are involved in the stratification order are either constants or appear elsewhere in the rule body in a positive goal or as output to a built-in operation. However any arguments which do not affect the stratification order of a negated goal do not have to be ground. Moreover, free variables appearing in negated goals can be constrained within the negated goal using built-in operations (see the next section for examples).

1.4 Programming in Starlog

The specifics of the Starlog programming language are now discussed. Starlog is capable of representing common programming features in a declarative and logically pure syntax. In this section we first describe Starlog's syntax and then give examples of programs which demonstrate the various features. To clarify many of the issues discussed in this section it may be useful to examine examples of Starlog programs. Two examples of Starlog programs can be found in this chapter in Figures 1.2 and 1.3 and seven further examples are included in Appendix C.

1.4.1 Syntax

Many syntactic features of Starlog are borrowed from the Edinburgh style syntax [26] [23] which has been incorporated into the ISO-Prolog standard. Consequently Starlog programs should look familiar to logic programmers. We assume a familiarity with syntactic elements of other logic programming languages including *terms*, *constants*, *variables*, *predicates* and *clauses* (see [26] or [65] for definitions of each).

This section gives an overview of the Starlog syntax. For a more comprehensive definition of the Starlog language see Appendix A.

Starlog Rules

Starlog programs include rules based on Horn clauses. The head of a rule is a tuple that corresponds to the rule's output. The body of a rule can contain positive goals, negated goals and built-in operations. Positive and negated goals are satisfied by the presence or absence of tuples in the tuple database, respectively. Built-in operations perform logical tests and operations on variables. All variables used by a rule must appear in a positive goal or as the output to a built-in operation in the body. This condition ensures that the heads of rules are completely ground when each rule body is evaluated.

Starlog rules rely on bottom-up evaluation and require a different style of programming than Prolog clauses. Therefore Starlog rules are syntactically distinguished from Prolog clauses. A Starlog rule takes the form "`H <- B.`" where `H` is the head of the rule and `B` represents the body. If the body of a rule is empty, the rule is called a *fact* and is represented as "`H.`".

Positive goals are predicates whose arguments are either constant symbols or variables (more on types later). Negated goals are predicates encapsulated in a "`not(...)`" structure.

To reduce the size of programs we permit a code optimisation for negated goals. Negated goals can include existential variables and may apply built-in operations within the "`not(...)`" structure. The ability to use existential variables in negations reduces the number of auxiliary predicates that would otherwise be required. For example, the two versions of the following program are equivalent.

Stratification Priorities (for inclusion in a Starlog program):

```
stratify r(X,Y,Z) [Y,Z,r].
stratify s(X,Y,Z) [X,Z,s].
stratify s << r.
```

Equivalent Stratification Order:

$\texttt{r(\_,Y1,\_)} \ll \texttt{r(\_,Y2,\_)} \Leftarrow \texttt{Y1} < \texttt{Y2}$
$\texttt{r(\_,Y,\_)} \ll \texttt{s(X,\_,\_)} \Leftarrow \texttt{Y} < \texttt{X}$
$\texttt{s(X,\_,\_)} \ll \texttt{r(\_,Y,\_)} \Leftarrow \texttt{X} < \texttt{Y}$
$\texttt{s(X1,\_,\_)} \ll \texttt{s(X2,\_,\_)} \Leftarrow \texttt{X1} < \texttt{X2}$
$\texttt{r(\_,Y,Z1)} \ll \texttt{r(\_,Y,Z2)} \Leftarrow \texttt{Z1} < \texttt{Z2}$
$\texttt{r(\_,Y,Z1)} \ll \texttt{s(Y,\_,Z2)} \Leftarrow \texttt{Z1} < \texttt{Z2}$
$\texttt{s(X,\_,Z1)} \ll \texttt{r(\_,X,Z2)} \Leftarrow \texttt{Z1} < \texttt{Z2}$
$\texttt{s(X,\_,Z1)} \ll \texttt{s(X,\_,Z2)} \Leftarrow \texttt{Z1} < \texttt{Z2}$
$\texttt{s(X,\_,Z)} \ll \texttt{r(\_,X,Z)}$

Stratification Order of Selected Tuples

$\texttt{s(0,7,4)} \ll \texttt{r(2,0,4)} \ll \texttt{r(2,0,5)} \ll \texttt{s(1,0,3)} \ll \texttt{s(3,0,2)} \ll \texttt{r(0,3,2)} \ll \texttt{r(5,4,1)}$

Figure 1.1: Example stratification priorities and stratification order.

```
(1)      p(X) <- q(X), not(s(X)).
         s(X) <- r(X,Y), Y > 5.

(2)      p(X) <- q(X), not(r(X,Y), Y > 5).
```

The second program is the result of unfolding the definition of `s(X)`. (A similar transformation is discussed in [33] with respect to Prolog programs.) Note that such a transformation is only possible when the negated goal is satisfied by a single rule. This optimisation can significantly reduce the size of programs and can improve program evaluation when the number of predicates is reduced. Although this code optimisation is reasonably transparent and is applied to many of the example programs in this thesis, it is not considered during the definitions of bottom-up evaluation in Chapter 2. The omission of this optimisation greatly simplifies the definitions of the evaluation strategies.

Stratification Priorities

Starlog programs include *stratification priorities* which are the programmer defined specification of the stratification order. By convention stratification priorities are grouped together at the start of Starlog programs to ease understanding. Each predicate appearing in a program is used in at most one stratification priority. To assist explanation, an example set of stratification priorities is included in Figure 1.1.

Stratification priorities are identified by the keyword **`stratify`** at the beginning of a line in a Starlog program. An *abstract tuple definition* follows this keyword to identify the predicate whose stratification order is being defined. An abstract tuple definition is an instance of a predicate whose arguments are all represented by locally unique variable names. The order that the constants and arguments of a predicate are prioritised in the stratification order is given as a list of terms where those elements earlier in the list are more significant. A variable occurring in the list indicates that the stratification order depends on the value of the corresponding argument. The values of arguments are stratified using "standard" orderings where smaller numeric values are stratified before larger ones.

Constant values in the stratification priority list indicate that the stratification order depends on static elements from the predicate. The stratification order between constants can be made explicit in other stratification definitions using the `<<` operator. For example, when the stratification order between two predicates (e.g. `t/0` and `u/1`) depends on their predicate name, constant terms representing the predicate are included at an appropriate position in the priority list (e.g. `t0` and `u1` might occur in the respective stratification priority lists of the `t/0` and `u/1` predicates). While attempting to find an ordering between the two tuples, when constant values are encountered in both priority lists the ordering between these is resolved by consulting the stratification priorities for constant terms (e.g. if there exists a stratification priority **`stratify t0 << u1`** then tuples from the `t/0` are stratified before tuples of `u/1`). If no stratification priority exists for the two constant terms (or if only one term is constant and the other is a variable) then the tuples are unordered with respect to each other, and the remainder of the stratification priority lists need to be consulted to resolve the ordering.

To reduce the size of programs, stratification priorities do not have to be included for predicates which are stratified before all the other predicates in the program. All predicates which do not have stratification priorities are stratified before those which do. This allows any predicate which holds a static set of facts that are true for the life of the program to omit its stratification priority and so reduce the size of the program.

Figure 1.1 demonstrates how stratification priorities can succinctly specify a complex stratification order. The stratification priority for the `r/3` predicate initially stratifies tuples on their second argument (as indicated by the use of the `Y` argument as the first element in the priority list) and then ordered on their third argument (using the `Z` argument). The stratification priority for the `s/3` predicate stratifies tuples on their first argument (that denoted by `X`) and then on their last argument (using `Z`). In the event that the values of arguments for `r/3` and `s/3` tuples would place these tuples in the same stratum, a final stratification priority ensures that `s/3` tuples are stratified before `r/3` tuples. Although stratification priorities are precise they can be opaque when trying to assess the order of tuples from different predicates. For this reason the stratification orders given in this thesis use the expanded $\ll$, $\gg$, $\underline{\ll}$, $\underline{\gg}$ operator notation in places where a clear understanding of the stratification order is required.

Example Program

To further demonstrate the syntax of Starlog programs, Figure 1.2 introduces the Starlog Prime Number program. This program is a variant of the well known "Sieve of Eratosthenes" which finds prime numbers (in this case) between 2 and 10,000. The program contains three predicates: each `num/1` tuple is generated with its argument bound to an integer from 2 to 10,000, each `mult/1` tuple holds a multiple of a prime number as its argument (with an upper limit of 10,000), and each **`prime/1`** tuple holds a prime number between 1 and 10,000.

To understand this program it is best to first look at its rules. There are two

```
% Prime Number Program
%-------------------------------------------------------------------
% Generates prime numbers between 2 and 10,000 by finding all multiple
% values and then using negation to find values that are not multiple
% values.

stratify num(N)    [N,num].            % Order all tuples on their arguments
stratify mult(N)   [N,mult].
stratify prime(N)  [N,prime].
stratify num << prime.                 % Ensure primes are stratified late
stratify mult << prime.

num(2).
num(M) <- num(N), M is N+1,            % Generate all numbers in range
          M < 10000.

mult(M) <- num(N), prime(P), N >= P,   % Generate multiple values
           M is N*P, M < 10000.

prime(N) <- num(N), not(mult(N)).      % Deduce prime numbers
```

Figure 1.2: Starlog prime number program.

ways in which `num/1` tuples can be produced. The first tuple is produced as a fact whose argument is bound to the value 2. The second way to produce `num/1` tuples is from the rule with the `num/1` predicate for its head. This rule should be interpreted as, given a previously true `num/1` tuple, increment the value of its argument (`N`) to find `M` and, so long as `M` is smaller than 10,000, generate a new `num/1` tuple with `M` as its argument. The rule to generate `mult/1` tuples can be interpreted in a similar fashion: Given a true `num/1` tuple with `N` for its argument and a true `prime/1` tuple with `P` as its argument, if `N >= P` find the product of `N` and `P` as `M` and, so long as `M` is less than 10,000, generate a new `mult/1` tuple with `M` as its argument. The final rule in the program determines if an integer is a prime using deduction. This rule finds a true `num/1` tuple with `N` as its argument and tests if there already exists a `mult/1` tuple with the same argument value. If no such `mult/1` tuple exists then a new `prime/1` tuple is produced with `N` as its argument.

Starlog requires all rules to be strongly stratified. The stratification order used for the Prime Number program initially orders all tuples by their argument value. This has the effect of interleaving the production of tuples from each predicate without, for example, all `num/1` tuples being produced before any prime numbers are deduced. Stratification based on argument values is sufficient for all but the last rule to be strongly stratified. The last rule (which generates `prime/1` tuples) uses the same argument in the head as it does in each of its body goals. To make this rule strongly stratified constants are added to each of the lists in the stratification priorities, and two stratification rules specify a partial order over these constants. The stratification rules ensure that `num/1` tuples and mult/1 tuples are stratified before prime/1 tuples when their argument value is shared.

1.4.2 Input and Output

We have seen that adding input and output to other logic programming languages can affect a program's declarative semantics or complicate program definitions. This is because logic programs which are evaluated top-down generally follow a single path of execution through the clauses of a program in an attempt to prove queries. However programs that are evaluated bottom-up are not constrained to such behaviour. Programs evaluated bottom-up are free to explore alternative paths of execution (simultaneously if desired) without confinement to search spaces that satisfy queries. Consequently, input and output in programs evaluated bottom-up can be performed in different paths of the execution which, depending on the program's design, may or may not affect the rest of the program.

In Starlog, output is achieved when rules produce *output tuples* that perform a side-effect. In this thesis the only output tuples are instances of either the print/1 or print_string/2 predicates. The arguments of an output tuple specifies the format of the output. Output tuples are similar to the other tuples used by a program and are produced in stratification order. The two different output predicates are explained below with examples.

The ability to add new rules to produce output rather than changing the original program is an attractive feature of Starlog output. Separation of rules that produce output from the core rules of a program assists understanding and maintenance of Starlog programs. In [23] this approach to output has been shown effective for debugging purposes. A universal debugger which provides a trace of all tuples derived by a Starlog program can be achieved by adding a single rule[2].

To demonstrate output in a Starlog program the Prime Number program from Figure 1.2 displays each prime number in the standard output by adding the following lines.

```
stratify print(P) [P,print].
stratify prime << print.
print(P) <- prime(P).          % Output prime numbers
```

The rule declares that the argument value of each new prime/1 tuple will be used as the argument in a new print/1 tuple. The print/1 tuple automatically outputs its argument to the standard output. The stratification priorities specify that these print statements are performed after each prime/1 tuple is produced. If more formatting or output is required when printing then we could use print_string/2 output tuples instead, as shown below.

```
stratify print_string(_,P) [P,print_string].
stratify prime << print_string.
print_string(Line, P) <- prime(P), Line is "\n Prime Number:"+P.
print_string(Line, M) <- mult(M), Line is "\n Multiple:"+M.
```

[2]Note that this universal debugger requires meta-programming facilities that we do not consider in this thesis. To achieve the same result without such facilities requires a rule for each unique predicate in the program.

The last argument of `print_string/2` output tuples is used only to stratify these output tuples and does not appear in the output. Note the use of string concatenation by the "+" operator which assists formatting output.

Input in Starlog is more complicated and has two parts. To initiate input an *input request tuple* is produced as the head of a rule. Input request tuples are similar to output tuples and, in fact, may perform output such as defining a prompt or, in other versions of Starlog, give the specification of a dialog box. After an input request tuple has been produced in stratification order, input is generated from some source (typically the user). The input is packaged as an argument of an *input tuple*. Input tuples are automatically generated by the program as soon as the input is received (although their effect may be delayed due to stratification). Any rule can refer to an input predicate in its body. To associate input tuples with their input request tuples, both tuples can include additional identifier arguments.

An extension to the Prime Number program from Figure 1.2 is given here to introduce both input and output. The first rule requests prompted input from the user by producing an `input_request/1` tuple whenever a new prime number is produced. When the user provides input it automatically appears as an `input/2` tuple where the first argument is the user's input value and the second argument is a key used to associate each input with its request, and to strongly stratify rules. The second rule takes any newly generated `input/2` tuple and compares the input value with the set of previously determined prime numbers. If the user's input value is a prime number then the second rule reports this fact using an output tuple.

```
stratify input_request(_,P) [P,input_request].
stratify print_string(_,P)  [P,print_string].
input_request(Prompt, Key) <- prime(P), Key is P+1,
                        Prompt is "Enter a number between 1 and"+Key.
print_string("Is Prime", N) <- input(Input, Key), Input < Key,
                                    N is Key+1, prime(Input).
```

A criticism of this approach is that because both input and output predicates can be generated independently of each other, it may be impossible to predict the order that input will be requested or output will be produced. For example, consider adding all the extensions from this section to the Prime Number program. However this can be seen as an advantage. Output can be displayed as soon as it is generated rather than waiting for other components of a program to finish. Multiple input requests could all generate dialog boxes at the same time allowing the user to decide which ones are most relevant to respond to. Alternatively, if the order that input and output is performed is critical then the stratification order within the program can be strengthened.

1.4.3 Aggregation

Aggregation (or set-grouping) [91] is the ability to combine or summarise collections of information using a function. Typical aggregate operators include *sum*, *average*, *count* [67] as well as *max* and *min*. The ability to aggregate sets of tuples in Starlog programs simplifies many programs and reduces the complexity of many operations. For example, to find the maximum value for `X` in a set of `q(X)` tuples using the following logical definition requires comparing every q/1 tuple with every other q/1 tuple (stratification priorities are omitted due to their irrelevance).

```
max(X) <- q(X), not(q(Y), Y > X).
```

If implemented naively, such an operation can require quadratic time. However other algorithms can find the maximum value in a set of elements in linear time.

Although Starlog's aggregate operators have not yet been optimised, a protocol for their use has been established. This protocol is similar to that used for input. The first stage in performing an aggregate operation is to specify the operation type and the set of values to be considered. This is achieved by generating an *aggregate request tuple.* Aggregate request tuples define the aggregate operator as the functor and give a member of the operand set as the argument. A separate aggregate request tuple is generated for each member of the operand set. The output of the aggregate operation is automatically returned as an argument in an *aggregate result tuple.* The result can be queried in the bodies of rules. In general, aggregate operations can not be performed until the entire input set is available. For this reason aggregate result tuples are stratified after their aggregate request tuples.

The *max* aggregate operator is demonstrated by the following Starlog program fragment. These rules generate a max_request/1 tuple for every q/1 tuple produced. When all possible request tuples have been generated (i.e. when the stratum that produces q/2 tuples has passed) the result is generated as a single max_result/1 tuple which is used in the second rule.

```
max_request(X) <- q(X).
print(X) <- max_result(X).
```

There are two cases which require aggregate request tuples to have extra arguments. When multiple aggregate operations are performed an extra identifier (or key) argument is necessary in both the request and result tuples to associate one with the other. That is, an extra argument is used to group sets of max_request tuples together and associate the result tuple with each set. The second requirement for an extra parameter in aggregate request tuples avoids identical tuples being removed via automatic duplicate removal. For example, sum_request(5) and sum_request(5) are the same tuple so the summed output would be sum(5). When identical operands of an aggregate operator are packaged into aggregate request tuples, if the distinction of each element is important then an extra index parameter is used which is distinct for each operand. Because each of these arguments is distinct, aggregate request tuples are distinct and are not considered duplicates. Using the previous example but with additional unique arguments with each request, sum_request(5,id1) and sum_request(5,id2) are different resulting in sum(10). Distinguishing each operand is important when using a *sum*, *average*, or *count* operator but unnecessary for *min* and *max* operations.

1.4.4 Destructive Assignment

It was previously stated that destructive assignment is an attractive facility for programming languages, however, it is problematic for logic programming languages that wish to remain logically pure. Yet destructive assignment can be implemented in Starlog without the addition of ad-hoc mechanisms.

```
% Destructuve Assignment Program
%-----------------------------------------------------------------------
% Models a destructive assignment system where labelled variables can be
% assigned, reassigned, inspected and deleted.

stratify value_request(_,T)  [T,value_request].
stratify assign(_,_,T)       [T,assign].
stratify delete(_,T)         [T,delete].
stratify value(_,_,T)        [T,value].
stratify assign << delete.
stratify value_request << value.

% The current value has been assigned, but not deleted.
value(K,V,T) <- value_request(K,T), assign(K,V,T0), T>T0,
                not(delete(K,T1), T0<T1, T1<T).

delete(K,T0) <- assign(K,_,T0).   % New assignments automatically
                                  % delete previous values.

% Test commands
assign(x, 100, 0).
value_request(x,2).
assign(x, 300, 3).
assign(y, 200, 3).
value_request(x,4).
value_request(y,4).
assign(y, 400, 4).
value_request(x,5).
value_request(y,5).
delete(y, 5).
delete(x, 6).
value_request(x,7).
```

Figure 1.3: Destructive assignment program.

Consider the Starlog program from Figure 1.3. This program describes a typical destructive assignment system where assignments are made to labelled variables and hold their values until the variable is overwritten or deleted. Assignments are made using **`assign(K,V,T)`** tuples. The first argument (`K`) of this predicate specifies the label of the variable, the second (`V`) is the new value and the final argument (`T`) indicates when in the stratification order the assignment should take effect. Variables are deleted using the **`delete(K,T)`** predicate where the `K` argument is the variable name and `T` is the stratification value. To access the current binding of a destructive variable the value is requested by producing a `value_request(K,T)` tuple where `K` is the label of the variable and `T` specifies when in the stratification order the request is made. A corresponding `value(K,V,T)` is produced by the first rule of this program where `V` is the most recent value of the variable. Given the test commands at the end of Figure 1.3, this program generates the following sequence of `value/3` tuples.

```
value(x, 100, 2).
value(x, 300, 4).
value(y, 200, 4).
value(x, 300, 5).
value(y, 400, 5).
```

In spite of Starlog's bottom-up evaluation being monotonic, destructive assignment is simulated by this program without retracting or modifying previously derived tuples. Instead, rules are constructed that refer only to the most recent `value/3` tuples in their bodies because the view of the data is restricted to the current stratum. Since the current stratum changes as the program executes, the value of a variable can change in each stratum. For example, the following (somewhat artificial) rule accesses only the most recent `value/3` tuples since using any previous `value/3` tuple would reproduce a previously derived tuple.

```
sum_of_two_variables(Sum,T) <- value(K1,V1,T), value(K2,V2,T),
                                K1 =\= K2, Sum is V1+V2.
```

Using the set of test commands from Figure 1.3 this rule produces two new tuples: `sum_of_two_variables(500,4)` when the current value of the primary stratification argument is "4" and `sum_of_two_variables(700,5)` when the stratification argument is "5".

Previous values of the assignment variables can also be accessed by a rule. This is possible when `value/3` goals are not restricted to accessing the most recently asserted `value/3` tuples. The following rule demonstrate how any previous values can be accessed.

```
sum_of_two_variables(Sum,T) <- value(K1,V1,T1), value(K2,V2,T2),
                               K1 =\= K2, T is T1+T2, Sum is V1+V2.
```

When the example data in Figure 1.3 is used with this rule the following tuples are derived:

```
sum_of_two_variables(300,6). sum_of_two_variables(500,7).
sum_of_two_variables(500,8). sum_of_two_variables(700,9).
sum_of_two_variables(500,9). sum_of_two_variables(700,10).
```

The destructive assignment program given in Figure 1.3 creates many tuples and performs many comparisons during the life of a program and consequently may be inefficient. Research into making destructive assignment efficient by compiling the code into updates and accesses of a single memory location is on-going. In Chapter 8 gives some insights into this problem, however it is not pursued to its conclusion in this thesis.

1.5 Data-Structure-Free Programming

Notice that the example programs explored so far have not used any recursive types or explicitly defined data structures within their rules. This is not by accident. Using Starlog, we advocate a data-structure-free programming style.

In [23] it was observed that students who are familiar with predicate logic are often shocked to find that logic programming requires so much effort to design data structures and the code to manipulate them[3]. However since data structures are pervasive in other languages, it is often conceded that they are necessary to write all but trivial programs.

Yet there is another approach to data storage which has had considerable success. Since relational databases were first proposed by Codd in [27] they have seen widespread use in commercial and administrative domains [105]. Over the last three decades many users have accepted the relational data model for data representation in many different applications.

The Starlog approach to data representation follows the relational model. That is, tuples represent individual relations between argument values. In the relational model arguments contain only simple values rather than complex structured objects [105]. As such, Starlog tuples do not contain any complex structures (i.e. terms with functors and arguments) as their arguments. This restriction makes Starlog easier to learn for programmers familiar with the relational model.

The restriction to the relational model does not limit the expressiveness of Starlog. Indeed, data structures commonly used by other programming languages could be represented as relations – although the use of such data structures in Starlog programs is discouraged (for reasons explained later in this section). Figure 1.4 shows three different data structures which could be used in Java or Prolog programs and their relational equivalents in Starlog. An array in Java is represented by a set of `value(K,V)` tuples where `K` is the key (or index) of each `V` value. The indexes of each value are labelled 0 to 3 for convenience. Starlog represents a Prolog list using two additional predicates. The `list_root/1` predicate holds the index of the first element in the list. `next(I,J)` tuples identify parent and child elements in the list with `I` and `J` respectively. The relational representation of the binary tree given in Figure 1.4 is similar to the list. However the parent-child relationship tuple has now been split into left and right branches. Note that if data structure need to be dynamic then the relations describing their links and values can be assigned, updated or deleted using the programming techniques for destructive assignment that were discussed in the previous chapter.

The relational approach to data representation could offer several advantages. For example, the `value/2` relations in Figure 1.4 are identical for all

[3]In this discussion we do not consider flat relations as data structures.

Java:
```
String value[] ={"a","b","c","d"};
```
Starlog:
```
value(0,a).
value(1,b).
value(2,c).
value(3,d).
```

Prolog:
`[a,b,c,d]` (equivalent to `.(a, .(b, .(c, .(d, [])))))`)

Starlog:
```
list_root(0).
value(0,a).
next(0,1).
value(1,b).
next(1,2).
value(2,c).
next(2,3).
value(3,d).
```

Prolog:
```
tree(c, tree(a, null, tree(b, null, null)), tree(d, null, null))
```
Starlog:
```
tree_root(2).
value(2,c).
left_branch(2,0).
right_branch(2,3).
value(0,a).
right_branch(0,1).
value(1,b).
value(3,d).
```

Figure 1.4: Comparison between Starlog data structures and other representations.

these data structures. Therefore this data can be represented simultaneously in these data structures without duplicating the values themselves. Another advantage is that if the order of elements is irrelevant then the data structures do not need to be traversed to find each `value/2` relation. Also, since the links between list elements and the branches of the tree are separated from their values, operations such as sorting the list or balancing the tree are simplified since only the links and branches would require updating, thus the values remain unaffected. Yet inspite of such advantages the creation of explict data structures as sets of relations is not encouraged in Starlog.

Although explicit representations of data structures are necessary for some algorithms (e.g. building a Huffman tree), the most common motivation for advanced data structures is to improve efficiency. However the relational representations of data structures in Starlog incurs additional run time overhead due to the extra relations. More importantly, the logic of the program may be obscured. Instead, the approach taken in many Starlog programs is to avoid using any explicit data structures in favour of storing and accessing relations in the tuple database without any concerns for efficiency. For example, rather than walking through each link or branch which makes up the data structures in Figure 1.4, the following rule accesses all values with no regard for the data structure.

```
p(V) <- value(_,V).
```

Note that programs which do not use explicitly defined data structures in their source code are not restricted from using advanced data structures in their tuple database. In fact, the tuple database which holds `value/2` tuples could use an array, a list, or a binary tree to construct the same structures as those given in Figure 1.4. It will be seen in later chapters that much of the research into compiling Starlog is centred around finding efficient representations for run time data. It is even argued that automated techniques to specify data structures can be more appropriate than manual selection in the context of Starlog. The benefit of ignoring the factors which affect run time efficiency is that Starlog programs are more understandable to programmers.

1.5.1 Types

Given that Starlog programs assume a data-structure-free programming style, Starlog's type system is very simple. Each predicate's arguments are either integers, strings or floating point numbers. Each argument in a predicate has only one type.

To remain abstract, types are inferred from Starlog programs. In this thesis we ignore the issue of type inference. In the few places where types are necessary for compilation to proceed it is assumed that types are provided as some form of declared relations, such as:

```
type p(int,string,float).
```

1.6 Applications of Starlog

The Starlog programming language has been used for a number of years (using interpreted environments) and has been shown effective for representing a variety of programs. Some of the more interesting programs involve:

- Reactive graphics
- MIDI-music generation
- Robotics using Lego MindStorm© Robots
- Biology simulations
- Optimisation and search problems

With the availability of a Starlog compiler, it is expected that the range and complexity of Starlog programs will increase.

1.7 Organisation of this Thesis

This thesis describes a process for compiling Starlog programs as they have been described in this chapter. The compilation techniques are focused on efficient run time performance at all stages. This section forms a guide to the remaining chapters, while pointing out the unique aspects of the compilation process along the way.

The second chapter reviews the Semi-Naive algorithm for bottom-up evaluation which is currently used by many deductive database applications. Semi-Naive evaluation is then refined for strongly stratified programs with negation. Using Semi-Naive Evaluation as a base, we describe a new optimisation called "Triggering" which can significantly reduce the number of unifications that a stratified program performs at run time. Using Triggering Evaluation allows additional optimisations to improve the evaluation of negated goals and to specialise predicates for how they are used. Chapter 2 provides the foundation for the chapters which follow.

The remaining chapters describe noteworthy stages in the compilation process. To see how each component contributes to the whole process Figure 1.5 shows the compilation pipeline. Significant stages are numbered and are referred to throughout this section.

Chapter 3 describes the database used to hold tuples at run time. Starlog programs use a unique variant of a discrimination tree (see Chapter 3) whose structure is statically defined at compile time. The tuple database's schema is specialised for improved efficiency either by the programmer (see 1 in Figure 1.5), by a heuristic which observes how each predicate is used by the program (2), or a combination of the two. Observing how data is used by a program to automatically customise a database schema is believed to be a first in the field of compiler design.

Chapter 4 gives a definition of the Starlog Data Structure Language (SDSL) which provides insert, delete and query instructions for use on a tuple database

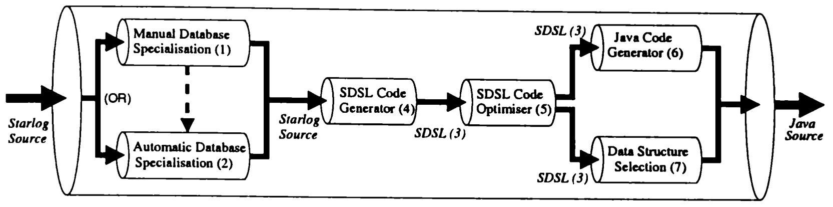


Figure 1.5: Overview of the Starlog compilation pipeline.

(3). SDSL is an intermediate language used only during compilation of Starlog programs.

In Chapter 5 it is shown how SDSL is used to implement Triggering Evaluation (4). The use of a static tuple database schema allows many instances of term matching to be performed statically at compile time. As such, programs are optimised by avoiding comparisons which never succeed. Chapter 5 concludes with six optimisations that can be applied to SDSL programs (5). Although most of these are variations on recognised optimisations for imperative programs, the scope of these optimisations is often different. The degree of optimisation achieved is evaluated using a suite of test programs.

Chapter 6 describes the process of translating SDSL programs into Java source code (6). This includes the definition of the tuple database in Java as well as the translation scheme for all SDSL instructions and facilities.

Chapter 7 argues that, given the data-structure-free style of programming, efficiency can be reintroduced to Starlog programs by customising the low-level data structures which make up the tuple database (7). Reasons are given why manual selection of data structures is unsuitable for Starlog. We present five techniques for automatically selecting data structures. Two selection techniques are given that observe only the properties of a program's source. Two more selection techniques are given that execute the program before making a selection. The final data structure selection technique delays making a selection until run time. The various automatic data structure selection techniques are evaluated using a suite of test programs. Although automatic data structure selection has been used in other contexts and compilers, this is the only known investigation which compares different approaches. Moreover, Starlog is the only instance of a logic programming language known to use automatic data structure selection.

The final chapter comments on the applicability of techniques described in this thesis to other domains. To assess the efficiency of Starlog, examples of compiled Starlog programs are compared with equivalent hand-coded programs. The chapter concludes with a discussion on future directions for research which would improve the efficiency and practicality of compiled Starlog programs.

Chapter 2

Abstract Bottom-up Evaluation

The first section of this chapter reviews Semi-Naive Evaluation – a bottom-up evaluation technique currently used by many deductive databases. Semi-Naive Evaluation is then extended for evaluation of stratified programs. Using Stratified Semi-Naive Evaluation as a foundation, the third section introduces the Triggering Evaluation technique used for Starlog programs.

To aid the understanding of bottom-up evaluation techniques we introduce a notation for bottom-up evaluated rules. Rules are generalised to the following format:

$$h \leftarrow a_1, \ldots a_m,\ not(c_1), \ldots not(c_n),\ b_1, \ldots b_k.$$

That is, for any rule there are m positive body goals, n negative body goals and k built-in calls all of which must be satisfied before the head tuple h is produced.

Alternatively, all positive goals, negative goals and built-ins of a given rule can be combined into a set called β (such that $h \leftarrow \beta$), and β is partitioned as follows:

β	$=$	$\{a_1, \ldots a_m,\ not(c_1), \ldots not(c_n),\ b_1, \ldots b_k\}$	(All elements in the body)
β^+	$=$	$\{a_1, \ldots a_m\}$	(Positive goals)
β^-	$=$	$\{not(c_1), \ldots not(c_n)\}$	(Negated goals)
β^λ	$=$	$\{b_1, \ldots b_k\}$	(Built-ins)
$\beta^\sim$	$=$	$\{c_1, \ldots c_n\}$	(Negated goals without the *not*(...) structures)

Positive goals (those in β^+) are satisfied by tuples produced by the program. These goals can be satisfied either as soon as the matching tuple is proven true or, if tuples are tabled, at any later time.

Negated goals (those in β^-) can only be satisfied when a matching tuple has not been previously generated by the program, and when there is no chance of a matching tuple being produced at any time in the future. For unrestricted logic programs this is undecidable. However, as described in the previous chapter, this can be overcome by restricting programs to those that obey a stratification order. Stratified programs will produce tuples according to a predefined partial order. Any tuple which is not generated at its time in the stratification order will never be produced. In this way negated goals can be satisfied if no matching tuples have been produced previously, and the time where any satisfying tuple could have been produced (according to the stratification order) has elapsed.

As described in Chapter 1, negated goals in Starlog may contain existential variables and built-in operations. However, for the purposes of describing bottom-up evaluation techniques, this code optimisation is ignored. As a result the definitions of bottom-up evaluation techniques are significantly simplified.

In this chapter, symbols θ and ϑ represent variable substitutions. Given a variable substitution θ, an instance of the term t is generated by $t\theta$. Note that variable substitutions are automatically constructed when terms are unified. (Although full unification is used throughout this chapter to produce variable substitutions, the following chapters which are specific to Starlog require only term matching. This property is because all Starlog tuples are ground and only goals in rules can contain variables. The use of unification in this chapter generalises the algorithms.)

2.1 Semi-Naive Evaluation

Semi-Naive Evaluation [39, 30] (sometimes called Δ-iteration [100]) has been presented previously in many different contexts and is widely used in the deductive database community [91]. The Aditi [108], CORAL [89], DECLARE [63] and Glue-Nail [35] deductive databases all use Semi-Naive Evaluation strategies. In most systems, refinements and new variants of Semi-Naive Evaluation have emerged to suit the intended application. The advantage of Semi-Naive evaluation over Naive Evaluation (a direct implementation of the fixed-point semantics from [68]) is that Semi-Naive Evaluation maintains the non-repetition property to avoid producing identical tuples in subsequent iterations [85]. For a proof that Semi-Naive and Naive Evaluation are equivalent see [10].

To begin, Semi-Naive Evaluation will be introduced for definite programs only (i.e. where rules do not have negated goals). Stratified Semi-Naive Evaluation that evaluates rules with negated goals is discussed later. This thesis presents Semi-Naive Evaluation as a set-processing procedure (similar to that in [85] and [106]) rather than a logical definition, since our goal is an implementation rather than a proof of correctness. For alternative definitions of the evaluation techniques presented in this thesis see [25].

Rules are said to be *activated* when they are selected to produce output. Semi-Naive Evaluation activates program rules only when a new tuple is generated that satisfies a positive goal in a rule's body. This ensures any new output of a rule will be based on new input, reducing repeated tuple derivations. In this chapter, new tuples derived from Semi-Naive Evaluation are distinguished from previously generated tuples using separate tables. Previously generated tuples exist in the Γ set whereas new tuples are stored in the Δ set. In this context Γ is sometimes called the *true* set whereas Δ may be referred to as the *new* set [111]. In both Γ and Δ duplicate tuples are eliminated to ensure correct and efficient evaluation.

Evaluation proceeds by taking a newly generated tuple from Δ and finding all the rules in the program where this tuple satisfies a positive goal. The remaining positive body goals are searched for in Γ. Any tuples generated from these rules are added to Δ to await processing and the tuple used to activate these rules is moved from Δ to Γ. In this way Δ stores tuples that are considered true but have not yet been used to activate or contribute to solving any program rules. Γ holds the set of true tuples after they have activated rules. This system will produce all possible tuples because the tuples in Δ that unify with the body goals in a rule will eventually activate the rule, and all previously generated tuples remain accessible through tabling in Γ.

```
Given Program $P$
$let\ Rules = \{(h \leftarrow \beta) \mid (h \leftarrow \beta) \in P \wedge \beta^+ \neq \emptyset\}$
$let\ Facts = \{(h \leftarrow \beta) \mid (h \leftarrow \beta) \in P \wedge \beta^+ = \emptyset\}$
$let\ \Xi =$ the set of all true, ground built-ins
$letfun\ conseq(h, \Gamma) = \{h_r\theta \mid (h_r \leftarrow \beta_r) \in Rules \wedge$
                              $h \in \beta_r^+\theta \wedge (\beta_r^+\theta - \{h\}) \subseteq \Gamma \wedge \beta_r^\lambda\theta \subseteq \Xi \wedge$
                              $h_r\theta \notin \Gamma\}$
$\Delta_0 = \{h\theta \mid (h \leftarrow \beta) \in Facts \wedge \beta^\lambda\theta \subseteq \Xi\}$
$\Gamma_0 = \emptyset$
$< \Delta_{i+1}, \Gamma_{i+1} > =$ if $\Delta_i \neq \emptyset$ then
                      choose $h \in \Delta_i$
                      $< (\Delta_i \cup conseq(h, \Gamma_i)) - \{h\}, \Gamma_i \cup \{h\} >$
                  else
                      $< \emptyset, \Gamma_i >$   (Termination)
                  fi
```

Figure 2.1: Semi-Naive Evaluation of positive programs.

A definition of Semi-Naive Evaluation is shown in Figure 2.1. Semi-Naive Evaluation makes a distinction between rules and facts. A fact is any program rule that has no positive goals in the body. Without positive goals, built-ins can be immediately evaluated because variable bindings do not depend on dynamic information generated at run time. A set of built-ins β^λ is true when $\beta^\lambda\theta \subseteq \Xi$ where θ is a variable substitution and Ξ is the set of all true ground built-ins. The ground heads of true facts are added to Δ. When Δ becomes empty no more rules will be activated and no more tuples are produced, thus the program can safely terminate. When Δ is not empty, evaluation continues by repeatedly selecting an arbitrary tuple h from Δ and moving it to Γ. h is also used to activate rules. The $conseq(h, \Gamma)$ function finds all rules for which one goal matches h and corresponding instances of all other goals are in Γ. The output (or heads) of these rules are collected in the output set. In the $conseq(h, \Gamma)$ function, successful unification between a body goal and new tuples creates a variable substitution θ and ensures that only relevant rules are activated. The remaining positive goals are searched for in Γ using $(\beta_r^+\theta - \{h\}) \subseteq \Gamma$. The exclusion of h from the goals satisfied by Γ is an optimisation that avoids repeatedly evaluating the goal matching h. Rules where all positive goals are satisfied, built-in operations are true and whose output is not already in Γ, produce true head tuples that are added to Δ ready for the next iteration.

The "Basic Semi-Naive Evaluation Algorithm" (BSN) given in [85, 10] as well as other definitions of Semi-Naive Evaluation (such as that in [106]) differ from the definition presented in Figure 2.1. These other definitions use all new tuples in Δ to activate rules (i.e. are set-oriented) rather than non-deterministically choosing one (tuple-oriented [118]). Evaluation using these different techniques gives the same output, however BSN will have fewer, but more computationally expensive, top level iterations. However the tuple-oriented representation of Semi-Naive Evaluation allows Starlog programs to be optimised using the triggered program transformation presented later. Although the tuple-oriented approach has been criticised for introducing a potential bottleneck [91], in general, it allows simpler implementation. Furthermore, the benefits of a set-oriented approach are greatly diminished for Starlog because rules can assert tuples arbitrarily later in the stratification order. This means that in any Semi-Naive iteration only a subset of the new tuples in Δ may activate rules. The additional partitioning required over the Δ set reduces efficiency in set-oriented evaluation. This problem is alluded to (but not thoroughly discussed) in [86].

2.1.1 Example Evaluation

To demonstrate Semi-Naive Evaluation an example program is given in Figure 2.2. This contrived example program finds some paths in a directed graph. Before evaluation begins, each edge in the graph is represented by a `path/3` fact where the first and second arguments are the source and destination nodes in the graph and the third argument is the cost (or weight) associated with the edge. The rule in the example program creates new `path/3` tuples when one path ends on the node where another path starts. However, built-in operations in the rule ensure that the cost of the first path must be greater than the cost of the second. The derivations made during evaluation are presented using a format similar to that of [85]

The Semi-Naive Evaluation of the program shown in Figure 2.2 begins by assigning Δ to the set of facts in the program. To ground the arguments of fact $\mathtt{path(a,d,C)} \leftarrow \mathtt{C\ is\ 5*2}$, the built-in operation $\mathtt{C\ is\ 5*2}$ is solved before the head is added to Δ. Γ is initially empty. From Δ the tuple $\mathtt{path(a,b,4)}$ is chosen as h. (In this example the first element in Δ will be consistently chosen for h.) To find all consequences of $\mathtt{path(a,b,4)}$ the *conseq*(h,Γ) applies all rules in the program where this tuple unifies with one of the positive goals. The remaining goals in the rule are searched for in Γ. In this case, because Γ is empty, all applications of rules fail to produce new tuples. $\mathtt{path(a,b,4)}$ is removed from the Δ set and added to Γ before the next iteration. In each iteration, any new tuples generated by the *conseq*(h,Γ) function (such as $\mathtt{path(a,c,7)}$ in iteration 2) are added to Δ. Eventually the Δ set becomes empty at which point termination occurs.

Semi-Naive Evaluation can now be modified to allow negated goals to be correctly evaluated.

2.2 Stratified Semi-Naive Evaluation

A stratification order is often used when evaluating negated goals to allow termination of programs using the fixed-point semantics [93]. By restricting programs so there is no recursion through negation, programs have an intuitive semantics, corresponding to the well-founded semantics [91]. However the evaluation techniques must respect the stratification order for an equivalent two-valued model

Program P:

```
path(a,b,4).
path(a,d,C) ← C is 5 * 2.
path(b,c,3).
path(c,d,5).
path(From,To,Cost) ← path(From,X,C1), path(X,To,C2), C1 > C2,
                         Cost is C1 + C2.
```

$\Delta_0 = \{$path(a,b,4), path(a,d,10), path(b,c,3), path(c,d,5)$\}$; $\Gamma_0 = \emptyset$	choose $h =$ path(a,b,4) ; $conseq(h, \Gamma_0) = \emptyset$
$\Delta_1 = \{$path(a,d,10), path(b,c,3), path(c,d,5)$\}$; $\Gamma_1 = \{$path(a,b,4)$\}$	choose $h =$ path(a,d,10) ; $conseq(h, \Gamma_1) = \emptyset$
$\Delta_2 = \{$path(b,c,3), path(c,d,5)$\}$; $\Gamma_2 = \{$path(a,b,4), path(a,d,10)$\}$	choose $h =$ path(b,c,3) ; $conseq(h, \Gamma_2) = \{$path(a,c,7)$\}$
$\Delta_3 = \{$path(c,d,5), path(a,c,7)$\}$; $\Gamma_3 = \{$path(a,b,4), path(a,d,10), path(b,c,3)$\}$	choose $h =$ path(c,d,5) ; $conseq(h, \Gamma_3) = \emptyset$
$\Delta_4 = \{$path(a,c,7)$\}$; $\Gamma_4 = \{$path(a,b,4), path(a,d,10), path(b,c,3), path(c,d,5)$\}$	choose $h =$ path(a,c,7) ; $conseq(h, \Gamma_4) = \{$path(a,d,12)$\}$
$\Delta_5 = \{$path(a,d,12)$\}$; $\Gamma_5 = \{$path(a,b,4), path(a,d,10), path(b,c,3), path(c,d,5) path(a,c,7)$\}$	choose $h =$ path(a,d,12) ; $conseq(h, \Gamma_5) = \emptyset$
$\Delta_6 = \emptyset$; $\Gamma_6 = \{$path(a,b,4), path(a,d,10), path(b,c,3), path(c,d,5), path(a,c,7), path(a,d,12)$\}$	

Figure 2.2: Semi-Naive Evaluation of an example path finding program.

to be generated. It appears this area has been neglected in the bottom-up evaluation literature.

Semi-Naive Evaluation, as presented in Figure 2.1, correctly evaluates stratified programs that include negated goals when the h tuples chosen from Δ are chosen according to the stratification order (rather than arbitrarily). (That is, negation within Semi-Naive Evaluation can be made equivalent to the well-founded semantics.) For proof, assume that stratified program P (with ground completion P^* and Ξ is the set of all true, ground built-ins) has the usual stratification property:

$$\forall(h_r \leftarrow \beta_r) \in P^*,\ (\Xi \models \beta_r^{\lambda}) \ \Rightarrow\ (\forall b \in \beta_r^{+},\ h_r \underline{\gg} b) \ \wedge\ (\forall n \in \beta_r^{\sim},\ h_r \gg n) \quad (1)$$

When a tuple h (chosen from Δ) successfully unifies with a goal in β_r^{+} and all built-ins are true, given $h \in \beta_r^{+}$, then $(\Xi \models \beta_r^{\lambda}) \Rightarrow h_r \underline{\gg} h$. If the only elements added to Γ are h where $h \in \Delta$ such that $\nexists h' \in \Delta,\ h' \ll h$ (i.e. h is a minimal stratified tuple in Δ), and the only elements added to Δ are h_r where $h_r \underline{\gg} h$, then $\forall d \in \Delta,\ \forall g \in \Gamma,\ g \underline{\ll} d$. In other words, the set of tuples derived by the program is partitioned according to the stratification order where all tuples in Δ are later in the stratification order than (or unordered with respect to) h and all tuples in Γ are earlier than (or unordered with respect to) h. When querying a negated goal $n \in \beta_r^{\sim}$, if $n \ll h\ \wedge\ n \notin \Gamma$ then n will never be derived by the program and $not(n)$ is conclusively true. Otherwise, if $n \ll h\ \wedge\ n \in \Gamma$ then $not(n)$ is false.

To ensure $n \ll h$, querying negated goals is delayed [20]. A first approximation of a stratum that is known to be later than n in the stratification order is the stratum of h_r. Because $h_r \gg n$ (from (1)), delaying evaluation of n until $h = h_r$ ensures that $n \ll h$. That is, when the head of the rule $h_r \leftarrow \beta_r$ becomes a minimal element in Δ, it is safe to query all negated goals in the rule ($\beta_r^{\sim}$) because the head of a stratified rule is later in the stratification order than its negated goals. Notice that this property holds for strongly stratified programs also. That is, programs that satisfy:

$$\forall(h_r \leftarrow \beta_r) \in P^*,\ (\Xi \models \beta_r^{\lambda}) \ \Rightarrow\ (\forall b \in \beta_r^{+},\ h_r \gg b) \ \wedge\ (\forall n \in \beta_r^{\sim},\ h_r \gg n) \quad (2)$$

Although the evaluation techniques presented in the remainder of this chapter assume modular stratification, they do not require strongly stratified programs to function. However opportunities for optimisation due to strongly stratified programs are pointed out along the way.

When choosing a minimal stratified tuple from Δ to activate rules there may be more than one candidate. This is because the stratification order may be a partial order. For now we require the evaluation technique to be fair when choosing tuples from Δ. The issue of fairness in the resulting implementation is solved in later sections.

To facilitate the delayed querying of negated goals, Δ holds head tuples together with their unresolved negated goals. Previously for Semi-Naive Evaluation, tuples were considered unconditionally true in both Δ and Γ. In Stratified Semi-Naive Evaluation the definition of Δ differs. Here the Δ set holds tuples which have not yet been used to activate program rules and that may be constrained by unresolved negated goals. For this reason tuples in Δ are no longer considered unconditionally true since they may yet be contradicted when these negated goals are queried. In this context Δ is referred to as the *pending* set.

```
Given Program $P$
$let\ Rules = \{(h \leftarrow \beta) \mid (h \leftarrow \beta) \in P \wedge \beta^+ \neq \emptyset\}$
$let\ Facts = \{(h \leftarrow \beta) \mid (h \leftarrow \beta) \in P \wedge \beta^+ = \emptyset\}$
$let\ \Xi =$ the set of all true, ground built-ins
$letfun\ conseq(h, \Gamma) = \{(h_r\theta \leftarrow \beta_r^-\theta) \mid (h_r \leftarrow \beta_r) \in Rules \wedge$
                    $h \in \beta_r^+\theta \wedge (\beta_r^+\theta - \{h\}) \subseteq \Gamma \wedge \beta_r^\lambda\theta \subseteq \Xi \wedge$
                    $h_r\theta \notin \Gamma\}$
$letfun\ min(\Delta) = \{(h \leftarrow \beta) \mid (h \leftarrow \beta) \in \Delta \wedge$
                 $\nexists(h_i \leftarrow \beta_i) \in \Delta, h_i \ll h\}$
$\Delta_0 = \{(h\theta \leftarrow \beta^-\theta) \mid (h \leftarrow \beta) \in Facts \wedge \beta^\lambda\theta \subseteq \Xi\}$
$\Gamma_0 = \emptyset$
$< \Delta_{i+1}, \Gamma_{i+1} > =$ if $\Delta_i \neq \emptyset$ then
                    choose $(h \leftarrow \beta) \in min(\Delta_i)$
                    if $\beta^\sim \cap \Gamma_i \neq \emptyset$ then
                      $< \Delta_i - \{(h \leftarrow \beta)\}, \Gamma_i >$
                    else
                      $< (\Delta_i \cup conseq(h, \Gamma_i)) - \{(h \leftarrow \beta)\}, \Gamma_i \cup \{h\} >$
                    fi
                  else
                    $< \emptyset, \Gamma_i >$    (Termination)
                  fi
```

Figure 2.3: Stratified Semi-Naive Evaluation.

The definition of Stratified Semi-Naive Evaluation in Figure 2.3 makes a number of changes to Semi-Naive Evaluation. Instead of choosing any tuple from Δ as h at the beginning of an iteration, a tuple is selected from the set of minimal tuples found using the $min(\Delta)$ function. The h tuple may be constrained by negated goals remaining in β. (In fact only negated goals can exist in β when $h \leftarrow \beta$ is held in Δ. Therefore $\beta = \beta^-$ in Δ.) Querying the negated goals in Γ is safe at this time for reasons given previously. If h is contradicted by a tuple in Γ then h is not derived and is eliminated from Δ before the next iteration. Otherwise h is moved to Γ. The $conseq(h, \Gamma)$ function generates new tuples as before, however these are now constrained by their negated goals.

To optimise this evaluation technique for strongly stratified programs requires a subtle change to the $conseq(h, \Gamma)$ function. When a program is strongly stratified the head of any rule is stratified later than all its goals, as in (2). If all true tuples that satisfy the goals of $h_r \leftarrow \beta_r$ are either h – a minimally stratified element in Δ – or are in Γ, then given that $h_r \gg h$ and $\forall g \in \Gamma,\ h \underline{\gg} g$, then $\forall g \in \Gamma,\ h_r \gg g \Rightarrow h_r \notin \Gamma$. As a consequence, the test in the $conseq(h, \Gamma)$ function for h_r in Γ will always fail and so can be omitted. The definition of $conseq(h, \Gamma)$ that does not search Γ for the existence of rule heads is given in Figure 2.4. Although this optimisation is applied by the implementation of Starlog, all later definitions of the $conseq(h, \Gamma)$ function given in this chapter do not apply this optimisation. As such, they are more general purpose.

$$\begin{aligned}\mathit{letfun}\ \mathit{conseq}(h,\Gamma) = \{(h_r\theta \leftarrow \beta_r^-\theta) \mid (h_r \leftarrow \beta_r) \in \mathit{Rules} \wedge \\ h \in \beta_r^+\theta \ \wedge\ (\beta_r^+\theta - \{h\}) \subseteq \Gamma \ \wedge\ \beta_r^\lambda\theta \subseteq \Xi\}\end{aligned}$$

Figure 2.4: Modification to Stratified Semi-Naive Evaluation to optimise strongly stratified programs.

2.2.1 Example Evaluation

Figure 2.5 gives an updated version of the program from Figure 2.2. In this new version negation is used to ensure that for any new path created, an equivalent path has not already been derived whose cost is two units less. This example is contrived for sure, but succinctly demonstrates Stratified Semi-Naive Evaluation for programs with negation. To ensure we prioritise the cost of a path, tuples are stratified on their cost argument. Because costs can only be positive values, the paths with the lowest costs will be generated first.

Stratified Semi-Naive Evaluation of this program repeatedly selects a minimum stratified element from Δ (interpreted as the path with the lowest cost) to activate rules. In iteration 1 the first new tuples are produced from the $conseq(h,\Gamma)$ function. The new `path/3` tuples generated are constrained by unresolved negated goals. These tuples are added to Δ still constrained by these negated goals. In iteration 3 the first `path/3` tuple constrained by a negated goal is selected to activate relevant program rules. Before this tuple is considered true the negated goals must be resolved. By comparing the negated goal `p(a, c, 5)` with the true tuples in Γ it can be determined that no other paths exist from node `b` to `d` with a cost of 5. Therefore this tuple is true and can be used to activate rules. Iteration 5 is of interest as it demonstrates the failure of a negated goal. The tuple `path(a, d, 12)` is only considered true if `path(a, d, 10)` is false. However the tuple `path(a, d, 10)` which exists in Γ contradicts this goal. Therefore the tuple `path(a, d, 12)` is not derived in this iteration and is removed from Δ without activating any rules.

In Stratified Semi-Naive Evaluation delaying evaluation of negated goals until the rule's head is the minimum tuple in Δ may be unnecessary. Next we present an optimisation that avoids storing some false tuples in Δ.

2.2.2 Early Failure of Negated Goals

In some cases Semi-Naive Evaluation can be optimised to allow for the early failure of negated goals. In the previous definition (Figure 2.3), querying negated goals in Γ is delayed until the head tuple is minimal in Δ. Without inferring more information from the program, this is the first time when queries on negated goals can give definite results. If such queries were performed earlier it is possible that tuples generated in the future would contradict the results of this query. However if a negated goal is queried earlier and fails due to some tuple in Γ then, because tuples are never removed from Γ, the negated goal will always fail. By testing for failure of negated goals early (before tuples are added to Δ) Stratified Semi-Naive Evaluation may be optimised so that fewer irrelevant tuples are added to Δ. This reduces the space required for Δ and improves the performance of operations on Δ. Yet to be correct, negated goals which do not fail early will also require querying when tuples become minimal in Δ. In cases where early failure does not occur, the extra queries performed

Program P:

```
path(a,b,4).
path(a,d,C) ← C is 5 * 2.
path(b,c,3).
path(c,d,5).
path(From,To,Cost) ← path(From,X,C1), path(X,To,C2), C1 > C2,
                     Cost is C1 + C2, Prev is Cost − 2,
                     not(path(From,To,Prev)).
```

Stratification Order:

$\texttt{path(\_,\_,C1)} \ll \texttt{path(\_,\_,C2)} \Leftarrow \texttt{C1} < \texttt{C2}$

$\Delta_0 = \{\texttt{path(a,b,4)}, \texttt{path(a,d,10)}, \texttt{path(b,c,3)}, \texttt{path(c,d,5)}\}$ $\Gamma_0 = \emptyset$
choose min $h = \texttt{path(b,c,3)}$ $conseq(h, \Gamma_0) = \emptyset$
$\Delta_1 = \{\texttt{p(a,b,4)}, \texttt{path(a,d,10)}, \texttt{path(c,d,5)}\}$ $\Gamma_1 = \{\texttt{path(b,c,3)}\}$
choose min $h = \texttt{path(a,b,4)}$ $conseq(h, \Gamma_1) = \{(\texttt{path(a,c,7)} \leftarrow \texttt{not(path(a,c,5))})\}$
$\Delta_2 = \{\texttt{path(a,d,10)}, \texttt{path(c,d,5)}, (\texttt{path(a,c,7)} \leftarrow \texttt{not(path(a,c,5))})\}$ $\Gamma_2 = \{\texttt{path(b,c,3)}, \texttt{path(a,b,4)}\}$
choose min $h = \texttt{path(c,d,5)}$ $conseq(h, \Gamma_2) = \emptyset$
$\Delta_3 = \{\texttt{path(a,d,10)}, (\texttt{path(a,c,7)} \leftarrow \texttt{not(path(a,c,5))})\}$ $\Gamma_3 = \{\texttt{path(b,c,3)}, \texttt{path(a,b,4)}, \texttt{path(c,d,5)}\}$
choose min $(h \leftarrow \beta) = (\texttt{path(a,c,7)} \leftarrow \texttt{not(path(a,c,5))})$ $\beta^{\sim} \cap \Gamma_3 = \emptyset \;\; \Rightarrow h$ is a true tuple $conseq(h, \Gamma_3) = \{(\texttt{path(a,d,12)} \leftarrow \texttt{not(path(a,d,10))})\}$
$\Delta_4 = \{\texttt{path(a,d,10)}, \; (\texttt{path(a,d,12)} \leftarrow \texttt{not(path(a,d,10))})\}$ $\Gamma_4 = \{\texttt{path(b,c,3)}, \texttt{path(a,b,4)}, \texttt{path(c,d,5)}, \texttt{path(a,c,7)}\}$
choose min $h = \texttt{path(a,d,10)}$ $conseq(h, \Gamma_4) = \emptyset$
$\Delta_5 = \{(\texttt{path(a,d,12)} \leftarrow \texttt{not(path(a,d,10))})\}$ $\Gamma_5 = \{\texttt{path(b,c,3)}, \texttt{path(a,b,4)}, \texttt{path(c,d,5)}, \texttt{path(a,c,7)}, \texttt{path(a,d,10)}\}$
choose min $(h \leftarrow \beta) = (\texttt{path(a,d,12)} \leftarrow \texttt{not(path(a,d,10))})$ $\beta^{\sim} \cap \Gamma_5 \neq \emptyset \;\; \Rightarrow h$ is a false tuple (contradicted by $\texttt{path(a,d,10)} \in \Gamma_5$)
$\Delta_6 = \emptyset$ $\Gamma_6 = \{\texttt{path(b,c,3)}, \texttt{path(a,b,4)}, \texttt{path(c,d,5)}, \texttt{path(a,c,7)}, \texttt{path(a,d,10)}\}$

Figure 2.5: Stratified Semi-Naive Evaluation of an example path finding program.

$$\begin{aligned} \textit{letfun conseq}(h,\Gamma) = \{(h_r\theta \leftarrow \beta_r^-\theta) | (h_r \leftarrow \beta_r) \in \textit{Rules} \wedge \\ h \in \beta_r^+\theta \ \wedge \ (\beta_r^+\theta - \{h\}) \subseteq \Gamma \ \wedge \ \beta_r^\lambda\theta \subseteq \Xi \wedge \\ h_r\theta \notin \Gamma \wedge \\ \beta_r^\sim\theta \cap (\Gamma \cup \{h\}) = \emptyset\} \end{aligned}$$

Figure 2.6: Modification to Semi-Naive Evaluation for early failure of negated goals.

may make the program less efficient.

Modification to the Semi-Naive algorithm in Figure 2.3 for the early failure of negated goals involves adding an extra test when finding the consequences of a tuple. As shown in Figure 2.6, the negated goals (in the form $\beta_r^\sim$) are queried against the set of true tuples that are stratified before the rule's head ($\Gamma \cup \{h\}$).

Figure 2.7 gives an example how of early failure affects program evaluation. The example program creates a new p/2 tuple when there is an equivalent q/2 relation whose two arguments are not commutative. From inspection it can be determined that the program rule will fail because the provided q/2 relation is commutative. Stratified Semi-Naive Evaluation begins by adding the two q/2 facts to Δ. The q(a, b) tuple is selected from Δ in the first iteration to activate the (only) rule. After unifying this tuple with the positive goal in the rule, the negated goal not(q(b, a)) is evaluated by searching Γ. Because a contradicting tuple is not present, early failure does not occur. The head of the rule p(a, b) is added to Δ, still constrained by the unresolved negated goal. In the next iteration q(b, a) is selected from Δ and is unified with the positive goal in the program rule. In this case the negated goal is instantiated to not(q(a, b)). Because the contradicting tuple q(a, b) exists in Γ the negated goal fails early and so the head tuple of this rule is not added to Δ. In the final iteration the element (p(a, b) $\leftarrow$ not(q(b, a))) is selected from Δ. Before the head tuple is considered true the negated goal must be resolved. By searching Γ, the contradicting tuple q(b, a) is found. If early failure did not occur in iteration 1 then an extra tuple would have been present in Δ and an extra iteration would have been required to extract this and resolve the negated goal.

In practice, early failure provides optimisation as the efficiency of operations performed on a smaller Δ sets usually outweigh the extra tests. For this reason we consider early failure a valid optimisation and it is applied whenever possible.

2.3 Triggering Evaluation

To introduce the next evolution of Stratified Semi-Naive evaluation we first modify the bottom-up evaluated program and the Stratified Semi-Naive Evaluation technique.

2.3.1 Triggering Transformation

In the previous sections it has been shown that both Semi-Naive and Stratified Semi-Naive evaluation will activate a rule when a true tuple taken from Δ unifies with one of the positive body goals in the rule. In the *conseq*(h, Γ) functions of Figures 2.1, 2.3, 2.4 and 2.6 this is performed when $h \in \beta_r^+\theta$. However

Program P:
q(a,b).
q(b,a).
p(X,Y) $\leftarrow$ q(X,Y), not(q(Y,X)).

Stratification Order:
q(_,_) $\ll$ p(_,_)

$\Delta_0 = \{\mathtt{q(a,b)}, \mathtt{q(b,a)}\}$ $\Gamma_0 = \emptyset$
choose min $h = \mathtt{q(a,b)}$ $conseq(h, \Gamma_0) = \{(\mathtt{p(a,b)} \leftarrow \mathtt{not(q(b,a))})\}$ because $(\{\mathtt{q(b,a)}\} \cap (\Gamma_0 \cup \{h\}) = \emptyset) \Rightarrow$ Early Failure of p(a,b) does not occur
$\Delta_1 = \{\mathtt{q(b,a)}, (\mathtt{p(a,b)} \leftarrow \mathtt{not(q(b,a))})\}$ $\Gamma_1 = \{\mathtt{q(a,b)}\}$
choose min $h = \mathtt{q(b,a)}$ $conseq(h, \Gamma_1) = \emptyset$ because $(\{\mathtt{q(a,b)}\} \cap (\Gamma_1 \cup \{h\}) \neq \emptyset) \Rightarrow$ Early Failure of p(b,a) does occur
$\Delta_2 = \{(\mathtt{p(a,b)} \leftarrow \mathtt{not(q(b,a))})\}$ $\Gamma_2 = \{\mathtt{q(a,b)}, \mathtt{q(b,a)}\}$
choose min $(h \leftarrow \beta) = (\mathtt{p(a,b)} \leftarrow \mathtt{not(q(b,a))})\}$ $\beta^{\sim} \cap (\Gamma_2 \cup \{h\}) \neq \emptyset \quad \Rightarrow h$ is a false tuple (contradicted by $\mathtt{q(b,a)} \in \Gamma_2$)

Figure 2.7: Early failure for an example program.

another approach that simplifies the $conseq(h, \Gamma)$ function is to transform the input program to explicitly define the positive goals that activate a rule.

For now we assume that any positive goal can activate a rule. Each program rule can be transformed into a set of rules where each new rule variant in the set is activated by a different positive goal. The number of rule variants is the same as the number of positive goals in the original rule. The positive goal that activates each variant is called the *Trigger*[1] and requires distinction from other positive goals in the rule. The trigger of the rule $h \leftarrow \beta$ is identified by β^{τ} where the β^{τ} goal has been removed from β^{+}. Figure 2.8 shows how a single program rule is transformed into a set of triggered rules for both the general case and an example rule. In examples of rules, the trigger goals are given in bold font.

By transforming all the rules in a program to use triggers the $conseq(h, \Gamma)$ function can be simplified. This function now compares only the trigger goals (rather than all positive goals) with the newly derived input tuple h before a rule is activated. The updated $conseq(h, \Gamma)$ function is shown in Figure 2.9. (Note that this version of the $conseq(h, \Gamma)$ is for Stratified Semi-Naive Evaluation using the early failure of negated goals optimisation.) One difference to note is that the new definition of $conseq(h, \Gamma)$ satisfies all the positive goals in β_{τ}^{+} using the set $(\Gamma \cup \{h\})$. This is because the trigger goal has been removed from β_{τ}^{+}

[1]Although the trigger terminology is used in both relational and deductive database literature it differs here. In relational databases, triggers activate programs or procedures when database tables or other components are used [54]. Deductive databases often use triggers to enforce integrity constraints or to periodically interact with the external environment [95]. However in both cases, triggers are specifically designed and assigned to databases by users [53, 34].

$$\begin{array}{ll}
\text{(a)} & \text{Given } R = (h \leftarrow \beta) \\
& R_{triggered} = \{(h \leftarrow \beta_{new}) \mid a \in \beta^{+} \ \wedge\ \beta^{-}_{new} = \beta^{-} \ \wedge\ \beta^{\lambda}_{new} = \beta^{\lambda} \ \wedge \\
& \qquad\qquad \beta^{\tau}_{new} = a \ \wedge\ \beta^{+}_{new} = \beta^{+} - \{a\}\} \\
\text{(b)} & \text{Given } R = (\mathtt{p(X,Y)} \leftarrow \mathtt{q(X), r(Y), s(Z), not(t(Z))}. \\
& R_{triggered} = \{(\mathtt{p(X,Y)} \leftarrow \mathbf{q(X)}, \mathtt{r(Y), s(Z), not(t(Z))}), \\
& \qquad (\mathtt{p(X,Y)} \leftarrow \mathtt{q(X)}, \mathbf{r(Y)}, \mathtt{s(Z), not(t(Z))}), \\
& \qquad (\mathtt{p(X,Y)} \leftarrow \mathtt{q(X), r(Y)}, \mathbf{s(Z)}, \mathtt{not(t(Z))})\}
\end{array}$$

Figure 2.8: Adding triggers to rules for (a) the general case and (b) an example rule (where triggers are bold positive goals).

$$\begin{aligned}
letfun\ conseq(h,\Gamma) = \{(h_r\theta \leftarrow \beta^{-}_r\theta) \mid (h_r \leftarrow \beta_r) \in Rules \wedge \\
h = \beta^{\tau}_r\theta \ \wedge\ \beta^{+}_r\theta \subseteq (\Gamma \cup \{h\}) \ \wedge\ \beta^{\lambda}_r\theta \subseteq \Xi \wedge \\
h_r\theta \notin \Gamma \wedge \\
\beta^{\sim}_r\theta \cap (\Gamma \cup \{h\}) = \emptyset\}
\end{aligned}$$

Figure 2.9: Modification to Semi-Naive Evaluation to evaluate triggered rules.

and other positive goals are able to match with h. When this new $conseq(h,\Gamma)$ function is used the resulting evaluation technique is called "Triggering".

The number of unifications performed by the new $conseq(h,\Gamma)$ function in Figure 2.9 is identical to the the previous function given in Figure 2.6. For a program with N rules and an average of M positive goals per rule, Stratified Semi-Naive Evaluation will attempt to unify all positive goals in all rules with the new tuple h, requiring NM comparisons[2]. For every triggered rule, only one positive body goal (the trigger) is unified with the newly derived tuple h. But when using triggers, for each N rules in the original program, there are (on average) M new rule variants. Therefore triggered programs also perform NM comparisons in $conseq(h,\Gamma)$.

The Triggering transformation presented here is similar to the rewriting of mutually recursive rules in [85] for Basic Semi-Naive Evaluation. In both transformations new versions of rules are created with different evaluation patterns in the bodies. But instead of requiring only one (the ith) positive body goal to be satisfied by a newly derived tuple, BSN requires all positive goals left of the ith goal to unify with newly derived tuples.

With Stratified Semi-Naive evaluation modified for triggered rules, a number of optimisations are possible.

2.3.2 Optimising Triggered Programs

To optimise Triggering Evaluation we observe that any rule which has a positive goal whose satisfying tuples are produced *later* than the trigger tuple will always fail. This is because if any positive goals can not be satisfied when the rule's trigger goal is produced the rule will fail. Any rule that always fails can be safely removed from the program.

[2]Implementations of these evaluation techniques may not perform this many comparisons since functor matching between goals and tuples may be performed statically. However, static functor matching can be used to optimise any of the evaluation techniques presented here, independent of any other optimisation.

Removing rules from the triggered program improves the efficiency of the *conseq*(h, Γ) function. In the best case each of the N rules in the original program will have exactly one triggered rule. In this case, *conseq*(h, Γ) will perform only N unifications between trigger goals and any new tuple h. For many programs this is a significant improvement.

Determining the order that positive goals are satisfied is trivial when the stratification order between goals is determined by static information (such as the functor). But when the stratification order of tuples depends on the values of their arguments, the order that positive goals are satisfied is less obvious. This is because arguments in positive goals may be free variables over which there is no obvious ordering. Yet by using theorem proving techniques the orderings between free variables may be inferred from relationships within the rule. Built-in operations in a rule may define the ordering between variables that must occur for the rule to succeed. These orderings may be explicit, such as stating $X > Y$ in the rule, or implicit such as $X\, is\, Y + K$ in a rule implies that $X > Y$ if $K > 0$.

For Starlog, the stratification order between positive goals is proven using the *Simplify* theorem prover[79][3] In a nut shell, the background axioms encapsulate the stratification order as defined by the program's stratification priorities and the theorem to be validated is that one of the positive goals is stratified later than all others given the conditions (i.e. the builtin operations) within each rule.

To demonstrate when rules can be removed from a program an example program is given in Figure 2.10. In this example the triggered rule definitions and stratification order combine to find the trigger goals that are not the last goal satisfied. In step 1, the order of goals `q(X,Z)` and `r(Y,Z)` in rule R_1 is given by rule C in the stratification order, ruling out `q(X,Z)` as the last goal. Consequently rule R_1 is removed from the program. For step 2 the built-in operation in R_2 that ensures `W` $>$ `Z` combines with stratification rule A. The result is that `r(Y,Z)` is not the last goal satisfied making rule R_2 redundant. Finally, step 3 infers that `V` $>$ `W` from the built-in `V is W + 1` in R_3. When combined with rule B in the stratification order it is inferred that `s(X,W)` is not the last and rule R_3 is removed. By deduction the last positive goal to be satisfied, and therefore the only valid trigger for this rule, is `t(Y,V)`. The rule triggered by this positive goal (R_4) remains in the program.

In some cases it is difficult or impossible to find the order that the positive goals in a rule can be satisfied. When this occurs fewer triggered rules can be removed from the program. For rules where positive goals are unordered due to a partial stratification order, or where no ordering relationship between argument variables exists, finding the last goal satisfied is impossible. Alternatively, the ordering relations between positive goals may depend on complex formulas or information only available at run time. Either way, finding the order that positive goals are satisfied may not be practical. When a trigger goal can not be ruled out as being the last positive goal satisfied then the rule must remain in the program. Performance degrades when each rule in the original program requires more than one triggered variant because, for any tuple produced, all but one of the triggered rules will always fail.

[3]*Simplify* was used because it has a Java implementation and so reduces the prerequistes for running Starlog.

Program P:
$R_1 = (\mathtt{p(X,Y)} \leftarrow \mathbf{q(X,Z)},\ \mathtt{r(Y,Z)},\ \mathtt{s(X,W)},\ \mathtt{W > Z},\ \mathtt{V\ is\ W+1},\ \mathtt{t(Y,V)})$
$R_2 = (\mathtt{p(X,Y)} \leftarrow \mathtt{q(X,Z)},\ \mathbf{r(Y,Z)},\ \mathtt{s(X,W)},\ \mathtt{W > Z},\ \mathtt{V\ is\ W+1},\ \mathtt{t(Y,V)})$
$R_3 = (\mathtt{p(X,Y)} \leftarrow \mathtt{q(X,Z)},\ \mathtt{r(Y,Z)},\ \mathbf{s(X,W)},\ \mathtt{W > Z},\ \mathtt{V\ is\ W+1},\ \mathtt{t(Y,V)})$
$R_4 = (\mathtt{p(X,Y)} \leftarrow \mathtt{q(X,Z)},\ \mathtt{r(Y,Z)},\ \mathtt{s(X,W)},\ \mathtt{W > Z},\ \mathtt{V\ is\ W+1},\ \mathbf{t(Y,V)})$
Partial Stratification Order:
A. $\mathtt{r(\_,X)} \ll \mathtt{s(\_,Y)} \Leftarrow \mathtt{X < Y}$
B. $\mathtt{s(\_,X)} \ll \mathtt{t(\_,Y)} \Leftarrow \mathtt{X < Y}$
C. $\mathtt{q(\_,X)} \ll \mathtt{r(\_,X)}$

Optimisation stages:
1. Using rule C, $\mathtt{q(X,Z)} \ll \mathtt{r(Y,Z)}$
   $\Rightarrow$ $\mathtt{q(X,Z)}$ is not the last to be satisfied, R_1 can be removed.
2. Using rule A, $(\mathtt{r(Y,Z)} \ll \mathtt{s(X,W)}$ where $\mathtt{W > Z})$
   $\Rightarrow$ $\mathtt{r(Y,Z)}$ is not the last to be satisfied, R_2 can be removed.
3. Using rule B and the additional rule $(\mathtt{V\ is\ W+1} \Rightarrow \mathtt{V > W})$,
   $(\mathtt{s(X,W)} \ll \mathtt{t(Y,V)}$ where $\mathtt{V > W})$
   $\Rightarrow$ $\mathtt{s(X,W)}$ is not the last to be satisfied, R_3 can be removed.

By deduction $\mathtt{t(Y,V)}$ is the last positive goal satisfied and R_4 remains.

Figure 2.10: Removing redundant rules based on the satisfaction order of positive goals.

The concept of activating rules using triggers has been previously seen in Pseudo-Naive Evaluation, introduced in [101]. This paper outlines the requirements for automatically determining trigger *tokens* (a new tuple's predicate name and arity) but few details are given. However, because Pseudo-Naive Evaluation is a refinement of Naive rather than Semi-Naive Evaluation, the non-repetition property no longer holds. Therefore, though fewer rules are activated using triggers than in Naive evaluation, Pseudo-Naive Evaluation frequently re-applies rules to the same data and so may be dramatically less efficient. [101] also mentions the possibility and advantages of identifying multiple triggers for a rule – the same concept explored here where multiple versions of a rule each have a different trigger.

An alternative approach to optimising Semi-Naive Evaluation is to use left-to-right modularly stratified programs described in [94]. These process rule bodies in ascending stratification order such that earlier goals are solved before later ones [20]. This is the opposite approach to the Triggering optimisation presented here, where the first positive goal to be satisfied is the last in the stratification order. Yet the technique needed to find the stratification order of goals from a static program is identical. Here we have shown analysis of variable relationships (given by built-in operations in rules) can help clarify such orderings.

2.3.3 Example Evaluation

The path finding program previously demonstrated in Figure 2.5 has been transformed into an optimised triggered program in Figure 2.11. To make this transformation two triggered versions of the program rule were created where one

```
Program P:
path(a,b,4).
path(a,d,C) ← C is 5 * 2.
path(b,c,3).
path(c,d,5).
path(From,To,Cost) ← **path(From,X,C1)**, path(X,To,C2),
                      Cost is C1 + C2, Prev is Cost − 2,
                      not(path(From,To,Prev)).

Stratification Order:
path(_,_,C1) ≪ path(_,_,C2) ⇐ C1 < C2
```

Figure 2.11: Triggered version of the path finding program from Figure 2.5 with trigger goals in bold.

uses the positive goal path(From,X,C1) and the other uses path(X,To,C2) for the trigger goals. From the stratification order given with the program (where path/3 tuples with a lower cost are produced first) and the built-in constraint C1 > C2 that orders the positive goals, the rule activated by path(X,To,C2) will always fail to produce output. This rule is removed from the program so that only one triggered rule remains, as shown in Figure 2.5.

Evaluation of the optimised triggered program will have an identical sequence of tuple derivations as that of Stratified Semi-Naive Evaluation (Figure 2.5), however each application of the *conseq*(h, Γ) function is now more efficient. New path/3 tuples are no longer unified with both positive goals in the program's rule. Instead new tuples are unified with only the single trigger goal. This not only halves the number of comparisons performed on each tuple in Δ but also reduces the searching in Γ for tuples that have not yet been generated.

With the basic Triggering optimisation in place additional optimisations are now possible.

2.3.4 Early and Late Evaluation of Negated Goals

Early failure of negated goals has been addressed with respect to Stratified Semi-Naive Evaluation. It was shown that failure of negated goals may be detected before the head tuple becomes the minimum of Δ, but since tuples that satisfy the negated goals may still be produced, definite evaluation of these is not possible.

Yet in Triggering, negated goals stratified before the trigger can be evaluated correctly when the rule is activated. This is called *Early Evaluation.* A rule $h_r \leftarrow \beta_r$ is activated when tuple h (a minimal element in Δ) unifies with the rule's trigger β_r^T. For each negated goal $n \in \beta_r^{\sim}$, if it can be proven that when all positive goals are satisfied and all built-ins are true, $n \ll \beta_r^T$ (i.e. all true tuples capable of satisfying a negated goal have already been generated when the rule is activated) then n can be safely queried in Γ when h unifies with β_r^T. This is because tuples are added to Γ strictly in stratification order and any negated goal that is stratified before the trigger of the rule can only be satisfied by tuples in Γ.

The converse situation is where negated goals can only be evaluated late –

$$\begin{aligned}
letfun\ conseq(h,\Gamma) = \{(h_r\theta \leftarrow \beta^{-}_{new}) \mid{} & (h_r \leftarrow \beta_r) \in Rules \wedge \\
& h = \beta^{\tau}_{r}\theta \ \wedge\ \beta^{+}_{r}\theta \subseteq (\Gamma \cup \{h\}) \ \wedge\ \beta^{\lambda}_{r}\theta \subseteq \Xi \wedge \\
& h_r\theta \notin \Gamma \wedge \\
& (\forall c \in \beta^{\sim}_{r}\theta,\ (c \ll \beta^{\tau}_{r}\theta \ \Rightarrow c \notin \Gamma) \vee \\
& \qquad\qquad (c \approx \beta^{\tau}_{r}\theta \ \Rightarrow c \notin (\Gamma \cup \{h\}))) \wedge \\
& \beta^{-}_{new} = \{not(c) \mid c \in \beta^{\sim}_{r}\theta \ \wedge\ (c \gg \beta^{\tau}_{r}\theta \vee \\
& \qquad\qquad\qquad (c \approx \beta^{\tau}_{r}\theta \ \wedge\ c \notin (\Gamma \cup \{h\}))))\}
\end{aligned}$$

Figure 2.12: Modifications to Triggering Evaluation for early and late evaluation of negated goals. (Includes early failure for negated goals that are not definitely stratified before or after the trigger.)

when h_r, the head of a rule, becomes the minimum in Δ. Given the negated goal $n \in \beta^{\sim}_{r}$, *Late Evaluation* optimises programs when it is proven $n \gg \beta^{\tau}_{r}$ when all positive goals are satisfied and all built-ins are true (i.e. n can only be satisfied after the trigger goal is satisfied). In these cases the early failure optimisation will never optimise the program and the additional early tests are removed.

To maximise optimisation, the use of early evaluation, early failure or late evaluation of negated goals should be specific to the negated goal being evaluated. For example, different negated goals in the same rule may apply different optimisations depending on when they are satisfied.

Changes made to the definition of the $conseq(h,\Gamma)$ function allowing for early and late evaluation of negated goals are shown in Figure 2.12. Note that the $\approx$ operator used in this definition succeeds only when its operands are neither stratified before or after each other (i.e. $a \approx b \ \Leftrightarrow \neg(a \ll b) \wedge \neg(a \gg b)$). All negated goals that are stratified before the trigger goal ($\beta^{\tau}_{r}\theta$) use early evaluation and so are queried within the set of true tuples stratified before the trigger goal (Γ). In these cases, if no contradicting tuple exists then one will never exist so after the negated goal is queried it is discarded (i.e. does not appear in β^{-}_{new}). Negated goals stratified after the trigger goal require late evaluation since these can not fail or be evaluated early. These negated goals are added to β^{-}_{new} and are queried when the head tuple becomes the minimum of Δ. Negated goals that are not definitely stratified before or after the trigger are still queried within the set of true tuples ($\Gamma \cup \{h\}$) to check for early failure. If no contradicting tuple exists one may still be generated in the future. The negated goal is stored in β^{-}_{new} to await evaluation when the head tuple becomes minimal in Δ.

The effect of early evaluation of negated goals is that tuples in Δ have fewer negated goals constraining them. The result is that Δ will be smaller due to fewer redundant tuples existing and, in cases where all negated goals in a rule can be evaluated early, a simpler data structure can be used (since tuples in Δ are unconstrained, specifying the types and arguments of remaining negated goals in Δ is now unnecessary). Using late evaluation reduces the number of redundant searches in Γ for tuples that have not yet been produced.

The example program rule in Figure 2.13 demonstrates when each of the negation optimisations can be applied. Rule R contains only one positive goal which is the trigger – `q(X,Y)`. Using the stratification order provided, the first negated goal `not(r(X))` is satisfied by `r/1` tuples that are generated strictly before any `q/2` trigger tuples. This means that when this rule is activated by

Rule R:
`p(X,Y)` $\leftarrow$ `q(X,Y),not(r(X)),not(s(Y)),not(q(Y,X))`.

Stratification Order:
`q(_,A)` $\ll$ `q(_,B)` $\Leftarrow$ `A < B`
`r(_)` $\ll$ `q(_,_)`
`q(_,_)` $\ll$ `s(_)`

Negated Goal in R	Optimisation
`not(r(X))`	Early Evaluation
`not(s(Y))`	Late Evaluation
`not(q(Y,X))`	Early Failure

Figure 2.13: Example uses of early/late evaluation and early failure of negated goals based on the stratification order of goals.

a new `q/2` tuple all true `r/1` tuples will already exist in Γ. In this case we apply the early evaluation optimisation to `not(r(X))` and query Γ only when the rule is activated. The second negated goal, `not(s(Y))`, is satisfied by tuples generated after all `q/2` tuples. Late evaluation of this negated goal will delay querying Γ until the head tuple of this rule is the minimum tuple in Δ. The final negated goal in this rule, `not(q(Y,X))`, is unordered with respect to the trigger goal `q(X,Y)`. This means that it is unclear whether this negated goal is satisfied by tuples generated before or after the rule is activated. We apply the early failure optimisation to this negated goal to allow the rule to fail if a contradicting tuple exists in Γ when the rule is activated, and query Γ a second time when the head tuple of the rule becomes the minimum of Δ.

2.3.5 Optimisation for Non-Trigger Rule Heads

In the definition of Triggering Evaluation, rules are activated when the trigger goal successfully unifies with a minimal true tuple in Δ. However it is possible to generate tuples that do not trigger any rules. Programs can be optimised so that these tuples are never treated as triggers and instead are passive tuples used only to solve the remaining body goals (after a rule is triggered).

To detect if the head of a rule (including facts, which, for this optimisation, are considered rules without any positive goals in their bodies) is a non-trigger tuple in program P the following condition is used. Note that head tuples constrained by negated goals (i.e. when a negated goal is not evaluated early) must always be added to Δ to delay their evaluation. Given a rule $h \leftarrow \beta$, h is a non-trigger head iff:

$$(\forall n \in \beta^{\sim},\ \forall\theta,\ n\theta \ll \beta^{\tau}\theta)\ \wedge\ (\forall(h_i \leftarrow \beta_i) \in P, \nexists\theta,\ \beta_i^{\tau}\theta = h)$$

To ensure non-trigger head tuples are never treated as triggers, non-trigger heads are never added to Δ — the set of tuples waiting to activate rules. Instead non-trigger head tuples are added to Γ as soon as they are generated by a rule. The Γ set is exclusively used for non-trigger goal evaluation. The effect of adding tuples to Γ immediately is that tuples are no longer added to Γ in stratification order – non-trigger tuples may be added early. Yet the addition of future tuples to Γ will not affect the correctness of programs with stratified negation if the stratification order is enforced in rules. That is, for all rules $(h_r \leftarrow \beta_r) \in P$, it must be true that if $\beta_r^\lambda\theta \subseteq \Xi$ then $\forall n \in \beta_r^\sim, n\theta \ll h_r\theta$. Therefore this optimisation is valid only when each rule includes the stratification conditions on the argument variables used by the rule's head and all negated goals. To illustrate, if the stratification order between `p/1` and `q/1` is given as $\mathtt{p(X)} \ll \mathtt{q(Y)} \Leftarrow \mathtt{X} < \mathtt{Y}$ then any rule with a `q(Y)` as a head and `not(p(X))` in the body must include the condition that $\mathtt{X} < \mathtt{Y}$. The conditions from the stratification order can be added to Starlog programs without changing their evaluation[4].

In some cases the early addition of non-trigger tuples to Γ will increase the early failure rate of negated goals. Adding non-trigger tuples to Γ early allows some positive goals to be satisfied earlier. However the early satisfaction of positive goals does not alter the correctness since any consequence of a head tuple produced early is delayed until the head tuple is a minimal element in Δ.

Tuples from built-in predicates that cause external side-effects (such as printing to standard output) are never considered non-trigger rule heads. This is because although the program may never refer to built-in predicates in any rule bodies, these built-in predicates trigger external operations where the order of operations may be important.

Optimisation of non-trigger rule heads can not be expressed as an incremental modification to the previous definition of Triggering Evaluation given in Figure 2.9. Instead, all the optimisations described in this chapter (including the optimisation in the next section) are given in the definition in Figure 2.15. This figure represents the version of the bottom-up evaluation technique that is used to evaluate Starlog programs throughout the remainder of this thesis. To implement the optimisation for non-trigger rule heads the $\mathit{conseq}(h, \Gamma)$ function has been split to separate the new tuples that will be added to the Δ and Γ sets. In addition, non-trigger tuples are added Γ instead of Δ.

This optimisation improves evaluation in three ways. By adding tuples directly to Γ instead of processing head tuples through Δ first, the number of tuple insert and delete operations is reduced. Also, attempts to unify non-trigger head tuples with trigger goals are skipped since these will always fail. By reducing the number of tuples in Δ, finding the minimum element should be more efficient. However, as Δ gets smaller Γ gets larger. Tuples stored in Γ earlier than if they had been processed through Δ are usually not immediately used to satisfy goals. Therefore it is possible that the additional tuples in Γ (that remain unused) will saturate this set and can reduce efficiency of other searches in Γ. However by using specialised data structures to implement the Γ set the inefficiency of saturated data structures can be minimised (see Chapter 7).

An example of a program containing non-trigger heads is given in Figure 2.14. (This example program is also used to demonstrate optimisation of exclusive trigger heads discussed in the next section.) Notice that the stratification order between any negated goals and the head of a rule is explicit in each rule through the use of built-in tests. This satisfies the precondition for this optimisation mentioned previously. For each rule (including facts), if the head can not unify with any trigger goals in the program then the head is a non-trigger. In the example program, rule 1 is activated when a tuple matching `q(2,_)` is generated whereas rule 2 is activated by any `q/2` tuple. Because the first two rule

[4] If existential variables are used by negated goals the conditions on such variables must be included inside the `not(...)` structure.

heads q(1,2) and q(2,1) can unify with at least one of these trigger goals, these rule heads are not non-trigger goals. However, the rule heads r(3) and r(2) do not unify with any trigger goals, making these non-trigger heads. Although the rule head p(X,Y) is not used as a trigger in any rules, this rule head can not be optimised. The rule which produces p(X,Y) tuples includes the negated goal not(r(Y)) which is *not* stratified before the trigger of this rule. Consequently, the p(X,Y) rule head must wait in Δ until the unresolved negated goal can be correctly evaluated.

Evaluation of the program in Figure 2.14 is optimised by inserting new tuples generated by non-trigger rule heads directly into Γ instead of Δ. This occurs in iterations 0 and 2 when tuples r(3) and r(2) are added to Γ.

2.3.6 Optimisation for Exclusive Trigger Rule Heads

In contrast to non-trigger tuples, exclusive trigger tuples are those that are only ever used as triggers in a program. These tuples are never used as negated goals and any positive goal references are always as the trigger of the rule. Optimisation can occur when these tuples are omitted from the Γ set. For a given rule $h \leftarrow \beta$, h is an exclusive trigger head iff:

$$(\forall(h_i \leftarrow \beta_i) \in P, (\forall\theta\vartheta, h\theta \not\in \beta_i^{+}\vartheta) \wedge (\forall\theta\vartheta, h\theta \not\in \beta_i^{\sim}\vartheta))$$

Excluding a tuple from Γ optimises future searches in Γ. It should be noted that Γ was originally considered the model or set of all true tuples a program generates. By applying this optimisation this property no longer holds. However, as this optimisation is intended for a programming language implementation rather than a theorem prover or database system, there is no value in keeping all tuples indefinitely.

The definition of semi-naive evaluation given in Figure 2.15 implements the optimisation for exclusive trigger rule heads. In this figure, the *storeGamma*(h) function is used to update Γ when h is used as a trigger in some rules. Otherwise *storeGamma*(h) excludes h from Γ.

This optimisation is actually a form of garbage collection as any tuples that are not referenced by the program in the future are deleted (or, in this case, simply not stored). Chapter 8 discusses generalisations of this form of garbage collection in the future work section.

In Figure 2.14 the rule heads q(1,2), q(2,1) and p(X,Y) in the example program do not unify with any non-trigger body goals in any program rules. Therefore these are exclusive trigger heads. When evaluating the example program, in iterations 0, 1 and 2 tuples q(1,2), q(2,1) and p(2,1) are the minimum tuples in Δ, respectively. After these tuples have activated rules they are removed from Δ but not added to Γ. The result is that the Γ set is smaller and does not hold tuples that can not contribute to the derivation of other tuples.

An interesting property of combining exclusive trigger and non-trigger head optimisations is that together they can remove redundant tuples from a program. Tuples that are not used as triggers are never added to Δ (excluding those constrained by negated goals and those causing external side-effects). Tuples that are never referenced by a program in non-trigger goals are never added to Γ. As a consequence, tuples which satisfy both conditions are never added to Δ or Γ. Although not considered in this thesis, this optimisation could be further expanded to remove any tuples that are constrained by negated goals but are not used by the program. Furthermore, such an optimisation could be recursively applied so that the removal of one set of redundant tuples from a program can prompt the removal of others. The ultimate result of such an optimisation would be the removal of all rules that do not (eventually) result in a side-effect. For all the example programs in this thesis that do not generate side-effects, this optimisation would remove all the rules.

2.4 Conclusions

In this chapter it has been shown how bottom-up evaluation of stratified programs is possible. The Triggering Evaluation technique used to evaluate Starlog programs is based on Semi-Naive Evaluation which is used by many deductive database systems. However Triggering is optimised for stratified programs in ways not previously seen in the bottom-up evaluation literature. Triggering Evaluation can reduce the number of unifications between goals and new tuples by a factor of M, where M is the average number of positive goals in each rule. Additional optimisations of rule heads and negated goals are also possible.

Triggering Evaluation improves the efficiency of Starlog programs, making Starlog a more attractive language for programmers. In addition to improving bottom-up logic programming languages, new bottom-up evaluation techniques may benefit deductive databases, theorem provers and planners. The fact that Triggering Evaluation takes advantage of a stratification order makes this an appealing technique for applications where stratification is already necessary for well-founded negation.

The use of sets in the definitions of Triggering Evaluation does not prohibit the use of more advanced data structures. Indeed, to be more efficient, indexing techniques can be used to optimise operations performed over the various sets of tuples. The next chapter describes an indexing technique used to hold and access tuples efficiently during Triggering Evaluation.

Program P:
$\mathtt{q(1,2).}\ \ \mathtt{q(2,1).}\ \ \mathtt{r(3).}$ %Facts
$\mathtt{r(2)} \leftarrow \mathbf{q(2,\_)}.$ %Rule1
$\mathtt{p(X,Y)} \leftarrow \mathbf{q(X,Y)},\ \mathtt{X > Y},\ \mathtt{not(r(Y))}.$ %Rule2

Stratification Order:
$\mathtt{q(X,\_)} \ll \mathtt{r(X)}$
$\mathtt{q(X,Y)} \ll \mathtt{p(X,Y)}$
$\mathtt{r(Y)} \ll \mathtt{p(X,\_)} \leftarrow \mathtt{Y < X}$

Finding Non-Trigger Heads:
Head $\mathtt{q(1,2)}$ = trigger goal $(\mathbf{q(X,Y)})\theta$ $\Rightarrow$ $\mathtt{q(1,2)}$ is not a non-trigger head.
Head $\mathtt{q(2,1)}$ = trigger goal $(\mathbf{q(X,Y)})\theta$ $\Rightarrow$ $\mathtt{q(2,1)}$ is not a non-trigger head.
Head $\mathtt{r(3)} \neq$ any trigger goal $\Rightarrow$ $\mathtt{r(3)}$ is a non-trigger head.
Head $\mathtt{r(2)} \neq$ any trigger goal $\Rightarrow$ $\mathtt{r(2)}$ is a non-trigger head.
Head $\mathtt{p(X,Y)}$ is constrained by negated goal $\mathtt{not(r(Y))}$ $\Rightarrow$ not a non-trigger head.

Finding Exclusive Trigger Heads:
Head $\mathtt{q(1,2)} \neq$ any non-trigger body goal $\Rightarrow$ $\mathtt{q(1,2)}$ is an exclusive trigger head.
Head $\mathtt{q(2,1)} \neq$ any non-trigger body goal $\Rightarrow$ $\mathtt{q(2,1)}$ is an exclusive trigger head.
Head $\mathtt{r(3)}$ = negated goal $(\mathtt{r(Y)})\theta$ $\Rightarrow$ $\mathtt{r(3)}$ is not an exclusive trigger head.
Head $\mathtt{r(2)}$ = negated goal $(\mathtt{r(Y)})\theta$ $\Rightarrow$ $\mathtt{r(2)}$ is not an exclusive trigger head.
Head $\mathtt{p(X,Y)} \neq$ any non-trigger body goal $\Rightarrow$ $\mathtt{p(X,Y)}$ is an exclusive trigger head.

$\Delta_0 = \{\mathtt{q(1,2)},\mathtt{q(2,1)}\}$ $\Gamma_0 = \{\mathtt{r(3)}\}$
choose min $h = \mathtt{q(1,2)}$ $\mathit{conseq}(h,\Gamma_0) = \emptyset$
$\Delta_1 = \{\mathtt{q(2,1)}\}$ $\Gamma_1 = \{\mathtt{r(3)}\}$
choose min $h = \mathtt{q(2,1)}$ $\mathit{conseq}(h,\Gamma_1) = \{\mathtt{r(2)},(\mathtt{p(2,1)}\leftarrow_{\mathtt{not(r(1))}})\}$
$\Delta_2 = \{(\mathtt{p(2,1)}\leftarrow_{\mathtt{not(r(1))}})\}$ $\Gamma_2 = \{\mathtt{r(3)},\mathtt{r(2)}\}$
choose min $(h \leftarrow \beta) = (\mathtt{p(2,1)} \leftarrow \mathtt{not(r(1))})$ $\beta^{\sim} \cap \Gamma_2 = \emptyset \quad \Rightarrow h$ is a true tuple $\mathit{conseq}(h,\Gamma_2) = \emptyset$
$\Delta_3 = \emptyset$ $\Gamma_3 = \{\mathtt{r(3)},\mathtt{r(2)}\}$

Figure 2.14: Example of Non-trigger and Exclusive Trigger Heads optimisation.

```
Given Program $P$
$let\ Rules = \{(h \leftarrow \beta) \mid (h \leftarrow \beta) \in P \ \wedge\ \beta^{+} \neq \emptyset\}$
$let\ Facts = \{(h \leftarrow \beta) \mid (h \leftarrow \beta) \in P \ \wedge\ \beta^{+} = \emptyset\}$
$let\ \Xi =$ the set of all true, ground built-ins
$letfun\ conseqGamma(h, \Gamma) = \{h_r\theta \mid (h_r \leftarrow \beta_r) \in Rules \wedge$
                                  $((\forall n \in \beta_r^{\sim},\ \forall\vartheta,\ n\vartheta \ll \beta_r^{\tau}\vartheta) \wedge$
                                  $(\forall(h_i \leftarrow \beta_i) \in Rules, \nexists\vartheta,\ \beta_i^{\tau}\vartheta = h_r)) \wedge$
                                  $h = \beta_r^{\tau}\theta \ \wedge\ \beta_r^{+}\theta \subseteq (\Gamma \cup \{h\}) \ \wedge\ \beta_r^{\lambda}\theta \subseteq \Xi \wedge$
                                  $(\beta_r^{\sim}\theta \cap (\Gamma \cup \{h\})) = \emptyset$
$letfun\ conseqDelta(h, \Gamma) = \{(h_r\theta \leftarrow \beta_{new}^{-}) \mid (h_r \leftarrow \beta_r) \in Rules \wedge$
                                  $\neg((\forall n \in \beta_r^{\sim},\ \forall\vartheta,\ n\vartheta \ll \beta_r^{\tau}\vartheta) \wedge$
                                     $(\forall(h_i \leftarrow \beta_i) \in Rules, \nexists\vartheta,\ \beta_i^{\tau}\vartheta = h_r)) \wedge$
                                  $h = \beta_r^{\tau}\theta \ \wedge\ \beta_r^{+}\theta \subseteq (\Gamma \cup \{h\}) \ \wedge\ \beta_r^{\lambda}\theta \subseteq \Xi \wedge$
                                  $(\forall c \in \beta_r^{\sim}\theta,\ (c \ll \beta_r^{\tau}\theta \ \Rightarrow c \notin \Gamma) \vee$
                                              $(c \approx \beta_r^{\tau}\theta \ \Rightarrow c \notin (\Gamma \cup \{h\}))) \wedge$
                                  $\beta_{new}^{-} = \{not(c) \mid c \in \beta_r^{\sim}\theta \ \wedge\ (c \gg \beta_r^{\tau}\theta \ \vee$
                                                     $(c \approx \beta_r^{\tau}\theta \ \wedge\ c \notin (\Gamma \cup \{h\}))))\}$
$letfun\ min(\Delta) = \{(h \leftarrow \beta) \mid (h \leftarrow \beta) \in \Delta \wedge$
                                  $\nexists(h_i \leftarrow \beta_i) \in \Delta, h_i \ll h\}$
$letfun\ storeGamma(h) = \{h \mid (\forall(h_i \leftarrow \beta_i) \in Rules,$
                                  $(\forall\theta\vartheta, h\theta \notin \beta_i^{+}\vartheta) \ \wedge\ (\forall\theta\vartheta, h\theta \notin \beta_i^{\sim}\vartheta))$
$\Delta_0 = \{(h\theta \leftarrow \beta^{-}\theta) \mid (h \leftarrow \beta) \in Facts \ \wedge\ \beta^{-} \neq \emptyset \ \wedge\ \beta^{\lambda}\theta \subseteq \Xi\}$
$\Gamma_0 = \{h\theta \mid (h \leftarrow \beta) \in Facts \ \wedge\ \beta^{-} = \emptyset \ \wedge\ \beta^{\lambda}\theta \subseteq \Xi\}$
$< \Delta_{i+1}\ ,\ \Gamma_{i+1} > =$ if $\Delta_i \neq \emptyset$ then
                        choose $(h \leftarrow \beta) \in min(\Delta_i)$
                        if $\beta^{\sim} \cap \Gamma_i \neq \emptyset$ then
                          $< \Delta_i - \{(h \leftarrow \beta)\}\ ,\ \Gamma_i >$
                        else
                          $< (\Delta_i \cup conseqDelta(h, \Gamma_i)) - \{(h \leftarrow \beta)\}\ ,$
                            $\Gamma_i \cup storeGamma(h) \cup conseqGamma(h, \Gamma_i) >$
                        fi
                      else
                        $< \emptyset\ ,\ \Gamma_i >$    (Termination)
                      fi
```

Figure 2.15: Triggering Evaluation with all optimisations.

Chapter 3

Indexing

The previous chapter described Triggering Evaluation for Starlog programs. Two critical components of Triggering Evaluation are the Δ and Γ sets. Therefore, before any evaluation of Starlog rules can begin, the implementation of these sets must be defined.

Recall from the previous chapter that the Γ set holds a set of ground tuples, where a tuple is made up of predicate name and a number of primitive arguments. Similarly, the Δ set holds a set of ground tuples, however tuples in the Δ set can be constrained by unresolved negated goals.

For clarity a few terms used throughout this chapter are defined. An *index structure* is a description of the global database schema. Diagrams of index structures use variables to indicate how a tuple's arguments are stored. An *index node* is a component of the index structure that holds the different possible values of an argument. An *index branch* is a directed edge down which one value of an argument is stored. A *path* is a collection of branches used to index all arguments in a tuple. An index node is an abstraction of a *data structure* which is a concrete storage device used to hold the values of an argument. An index structure that stores ground Starlog tuples in data structures is called an *index structure instance*.

As with all database related applications, for Starlog programs to be efficient, efficient access to data is crucial [90]. Starlog programs use fast in-memory (or primary storage) databases for the Δ and Γ sets, rather than slower disk-based solutions. The majority of deductive database systems are disk-based [91] which allows for larger volumes of data that will persist when the application terminates. However, because Starlog is intended as a programming language, the loss of run time data when the program terminates is permitted, as it is in other language implementations. (Persistence of selected data could be restored by adding file reading and writing facilities to Starlog.) The efficiency of the in-memory database is improved by optimising tuple storage devices for how they are used. Statically defining the database schema also reduces run time overhead.

In this chapter, the discrimination tree origins of Starlog's index structures are discussed and reasons are given why dynamic discrimination trees are unnecessarily complex for storing Starlog tuples. Various optimisations for Starlog's index structures are described. Operations performed on index structure instances are discussed before comparisons are made between Starlog's indexes

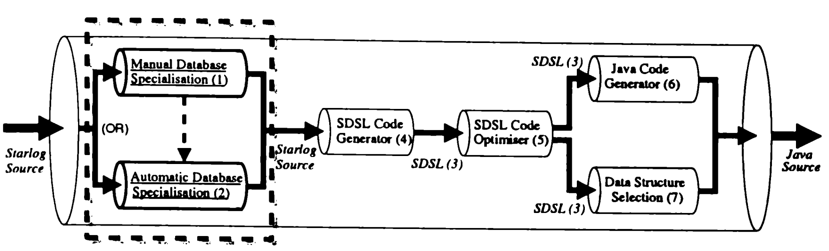


Figure 3.1: Specification of index structures in Starlog's compilation pipeline.

and relational database systems. Section 3.4 introduces a syntax to describe index structures. Section 3.5 goes on to describe one approach to automatically constructing efficient index structures based on how programs use tuples. Figure 3.1 shows where the specialisation of index structures occurs in the Starlog compilation pipeline.

3.1 Starlog's Index Structures and Discrimination Trees

Previous speed ups in indexing have been achieved for a variety of applications from the use of discrimination trees [76] (sometimes generalised to discrimination nets [17, 8] or refined to tabling tries [84]). Experiments using the automated deduction system OTTER showed that discrimination trees significantly improved indexing for generalisation retrieval and gave promising results for instance retrieval when compared with the alternative path indexing strategy [76] (although only the retrieval of instances is relevant to the Starlog database). Logic programming languages Quintus Prolog and XSB use tabling tries by extending the WAM instruction set to access terms [84]. This is an improvement over the much berated first-argument-indexing that was prevalent in early implementations of Prolog. However discrimination trees still require programmers to be aware of the indexing scheme to be able to take advantage of it. The theorem prover REVEAL uses Associative-Commutative (AC) discrimination nets to improve the efficiency of "many-to-one AC matching" [19, 18]. TRAM (Term Rewriting Abstract Machine), used to evaluate lazy functional languages OBJ3 and CafeOBJ, has been optimised by indexing rule heads in discrimination trees [81]. Discrimination nets are displayed to guide user queries performed on the COREL retrieval system [36] (not to be confused with the CORAL deductive database). More recently, discrimination trees have been incorporated into the KRHyper theorem prover to hold facts derived during Semi-Naive Evaluation [115]. Performance comparisons between regular discrimination trees and alternative indexing techniques (context trees and code trees) have been documented in [80] for a variety of theorem proving benchmarks.

To clarify the discussion on discrimination trees, an example is given in Figure 3.2 that shows how the following set of tuples is stored.

{p(a,b,c), p(b,b,c), q(a,2), q(c,1), q(c,2)}

To add tuples to a discrimination tree, each tuple is decomposed into a sequence of atomic components that describe the predicate and arguments of the tuple in a left-to-right order. For example, the tuple `p(a,b,c)` is decomposed into the following sequence of atoms: `p,a,b,c`. The discrimination tree itself is a multi-branched tree capable of storing these atoms. However discrimination trees do not hold duplicate branches in any sub-tree. When a sequence of atoms is added to a discrimination tree, each atom corresponds to a branch that is followed to find the next atom in the sequence. If there is no existing branch that represents the atom then a new branch is created (initially referencing an empty sub-tree). In [32] the process of building a discrimination tree from the set of literals in a relation is described as: "Write the literals in prefix notation. Then build a tree structure combining all initial segments of literals that are equal." A successful traversal from the root of the tree to a leaf indicates the presence of a relation with the values of the functor and arguments determined by the branches taken. For an excellent introduction to discrimination trees see [17].

In the example in Figure 3.2 the atoms indexed in the tree are given in the node at the end of each branch. That is, when searching for an atom (such as p) in a sub-tree (such as the `Root` node), the values associated with each branch are presented with the node at the end of the branch rather than beside the branch itself. This notation simplifies the layout of these diagrams. Note that in all tree diagrams in this thesis the tuples that are indexed at the end of each path are not explicitly stored in the leaf nodes – in spite of their inclusion in the diagrams (e.g. the inclusion of `p(a,b,c)` at the end of the left-most path of Figure 3.2). The tuples are included at the end of each path only as an aid to understanding. When a tree is searched, tuples must be recomposed using the atoms found in the path from the root to a leaf.

Discrimination trees are dynamic indexes capable of storing arbitrary relations. Each node of a discrimination tree is implemented by a dynamic data structure (e.g. KRHyper uses list or hash table implementations [115]) which we refer to as a *dynamic argument index* (or an *argument index* for short).

In spite of the well documented advantages of discrimination trees, regular discrimination trees are unnecessarily complex for Starlog tuples for two reasons. First, discrimination trees are capable of holding and indexing variable arguments [76, 17]. All arguments stored during Triggering Evaluation of Starlog programs are ground. As a result the tree structure and the search process is greatly simplified because special variable entries that unify with any goal term are not required, and additional filtering of query results to ensure unification of shared variables [80, 17] is unnecessary. Second, current implementations of Starlog do not allow arguments to contain nested structures (sub-terms) with their own functors and additional arguments for reasons discussed in Chapter 1. This means that every level of the tree will hold only simple terms, making the process of unifying these with goal arguments simpler – normally when a nested structure is unified with a variable, many levels of the tree may be used to build up the structure (i.e. functor + argument1 + argument2...) [17]. Without nested structures each level of the tree will consistently index the same arguments.

The use of discrimination trees as a base for Starlog's indexes may be criticised since the overriding advantage of discrimination trees is their ability to index nested structures and store variables – neither of which is necessary for Starlog. However discrimination trees are very effective for storing sets of rela-

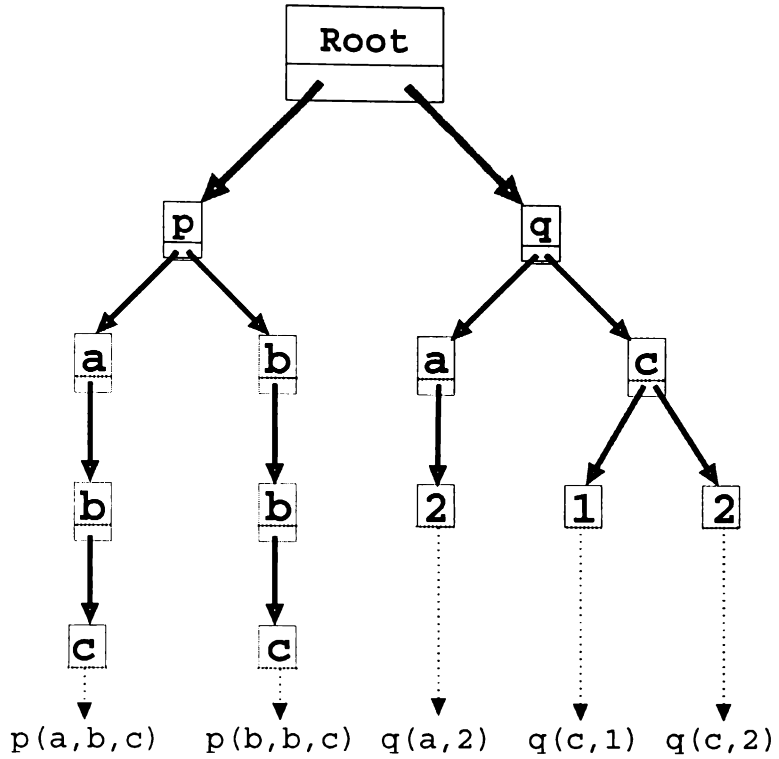


Figure 3.2: Example of indexing tuples in a discrimination tree.

tions since all duplicated relations share the same path ([17] describes relations as being "uniquified"). Moreover, the hierarchical nature of discrimination trees can be taken advantage of to minimise the overhead of backtracking through multiple solutions (see Section 3.5.1). This type of indexing also allows optimisation of operations on related tuples (see Chapter 5) in ways that would not be possible in other indexing frameworks.

The following sections present a series of optimisations for discrimination trees that improve their performance for Starlog.

3.1.1 Static Labelled Branches

In the last section it was stated that each node of a discrimination tree is implemented by a dynamic data structure that is capable of inserting and deleting values at run time. However, instead of using dynamic structures for every node in a discrimination tree, static indexes can be used for data with a small domain. In these indexes, each possible value is represented by a corresponding static labelled branch. As will be seen in Chapter 6, the implementation of and operations performed on statically labelled branches can be made very efficient.

Static branches are particularly useful for discriminating between tuples from different predicates. This is because the set of predicates used by a program is finite and usually small. Figure 3.3 (a) shows how static labelled branches are presented in diagrams of index structures. Static labelled branches are represented by curving, think, dashed arrow, with the label given in quotes. In Figure 3.3 (a), static labelled branches are used to distinguish between tuples with p and q as their functors.

Unlike dynamic argument indexes, the sub-indexes associated with static labelled branches can be customised for the set that they index. For example, because some predicates have more arguments, or arguments of a different type than others, each sub-index pointed to by a statically labelled branch can be specialised for the predicate it holds. Static indexes are also useful for discriminating between tuples in the Δ and Γ sets, and for customising dynamic argument indexes for specific data types. For example, if both integers and strings are expected for an argument then two static labelled branches can each reference a dynamic argument index holding a different argument type. The ability to store arguments of different types down different paths will be revisited later in this chapter and allows for further optimisations given in Chapter 8.

Although the labelled branches are static in each index node, the index nodes that they point to are not. When an index node is created, all labelled branches originating from this node point to a void (i.e. null) memory location. Labelled branches can be reassigned to index nodes when tuples are added.

By not storing nested terms and using static labelled branches to discriminate between predicates/sets with different sub-index structures, the structure of the discrimination tree can be statically defined. That is, for any predicate used by a program, how it is indexed (i.e. its path in the index structure) is already defined before any tuples are added to the index structure. Insert operations into a statically defined index structure are more efficient than those for a dynamic structure (such as a regular discrimination tree) because allowances do no have to be made for tuples from unknown predicates with unknown argument types.

3.1.2 Argument Reordering

The order that arguments are indexed in Starlog's index structures does not affect correctness of this system. During compilation the argument order can be entirely ignored so long as tuple arguments are unified with those of goals consistently. To achieve this, each predicate (distinct in Δ and Γ) has a map of how their arguments are arranged in the index structure. These maps are called *index path definitions* and will be explained later in this chapter. This is a significant departure from regular discrimination trees which have a strict left-to-right indexing order and would require transformation of predicates at the source level to achieve the same result.

Argument reordering also applies to static indexes. For example it is possible to change when predicate names are indexed within the index structure. When different predicates have arguments with similar properties (such as the same types and ranges of terms), these arguments can be indexed together in the same index node for greater efficiency. This frequently occurs when one predicate is a consequence of the other where variables are shared. Combining arguments from different predicates not only reduces the amount of space required for an index structure instance but is the basis of some code optimisations given in

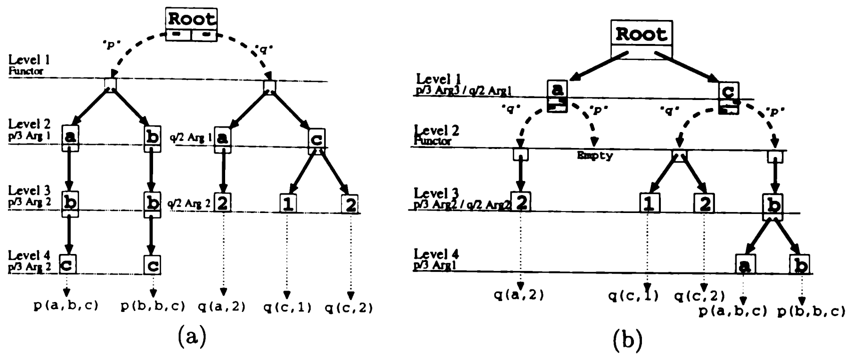


Figure 3.3: Two index structure instances with different argument orderings.

Chapter 5.

To illustrate how the order of the functor and arguments can change an index structure instance, Figure 3.3 shows two structures storing the same tuples with different argument orders. (Note that both structures use static labelled branches to index the functors of tuples.) Figure 3.3 (a) is a structure resulting from a simple left-to-right argument priority (starting with the functor). In (b) tuples of type p/3 are ordered initially on their third argument, then the functor, their second argument, and finally on their first argument. q/2 tuples are ordered on their first argument and functor before the second argument. The result, in this case, is that (b) is a smaller index instance because more arguments are combined together. Whether this optimised index structure leads to more efficient execution depends on how it is used by the program. This is discussed further in this and other chapters.

When arguments are reordered, index structures that include static labelled branches can be optimised. Static labelled branches can exist in conjunction with a dynamic argument index to reduce the size of an index structure. The values stored in a dynamic argument index are independent of any labelled branches that exist at the same index node. By allowing labelled branches in dynamic argument index nodes, some index paths may follow these labelled branches whereas others can follow branches in the dynamic argument index. The advantage of this is that index nodes may have a default dynamic argument index where the arguments of some tuples are stored. Labelled branches can be used to separate arguments from different predicates that are not stored in the default index (perhaps due to type conflicts). This system allows only those predicates that are considered exceptions to follow a labelled branch whereas other predicates do not perform the additional branching operation.

When this optimisation is applied to Figure 3.3 (b) only one labelled branch is required to discriminate p/3 tuples from q/2 tuples. Figure 3.4 shows the index where p/3 tuples are automatically indexed in level 2 of the index structure and q/2 tuples are considered the exception that require an additional labelled branch to reach.

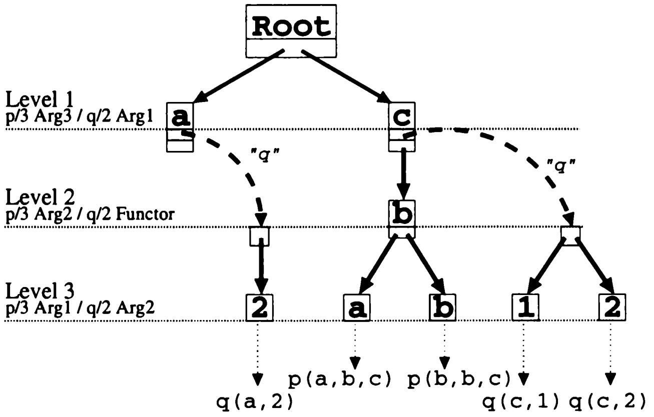


Figure 3.4: Labelled branch optimisation where p/3 is the default predicate.

3.1.3 Labelled Boolean Values

By indexing compatible arguments from different predicates in the same index nodes it is possible that all the arguments of one predicate will be indexed together with the arguments of another. When this occurs one predicate is said to be *subsumed* by another. To distinguish subsumed predicates in an index structure a static labelled branch could be inserted after the last argument is indexed. However, because all the arguments have been indexed, using a labelled branch that points to an index node is excessive since no additional values will be stored. Instead, a *labelled boolean value* is added to indicate the presence of a tuple from the subsumed predicate. The boolean value is *true* when a tuple of the particular type (identified by the boolean value's label) exists and *false* when the tuple is not present.

Figure 3.5 gives an index structure instance that uses labelled boolean values to discriminate two predicates. In diagrams of index structures, labelled boolean values are presented as a rectangle with rounded corners that is connected to the index structure by a dotted arrow. In this example the r/2 predicate subsumes the s/1 predicate after the first argument of each is indexed. This is common when arguments from one predicate contain a subset of the arguments from another predicate. A generalised program rule that may generate the r/2 and s/1 tuples in this way is:

```
s(X) <- ..., r(X,Y), ...
```

The labelled boolean value is placed in the index node after all arguments from s/1 tuples have been indexed, in order to distinguish s/1 tuples from r/2 tuples. The labelled boolean value is used only when searching for or inserting an s/1 tuple. When searching or inserting r/2 tuples the dynamic argument index holds the second argument.

When one predicate is a complete projection of and subsumed by another predicate, using a labelled boolean value is redundant. For example, this is the case if the index path for t(X) tuples is subsumed by the index path of u(X,Y) tuples, and the only rule producing t(X) tuples is:

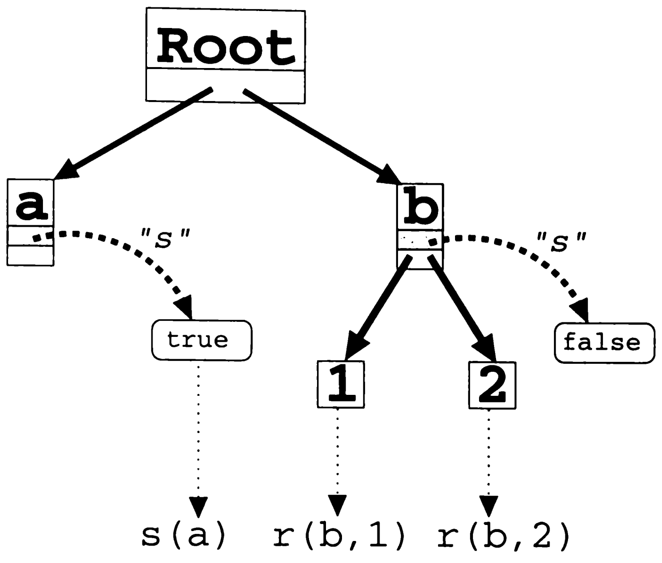


Figure 3.5: Using labelled boolean values to distinguish tuples subsumed in an index.

```
t(X) <- u(X,Y).
```

A labelled boolean value is redundant because the presence of one tuple implies the presence of the projected tuple and the labelled boolean value would always hold *true*. To optimise programs, subsumed, projected tuples need not be represented in the index structure. Any projected predicate referred to in a goal can be satisfied by searching for a subset of the arguments from the larger predicate on which the projection is based. Because a projected tuple has been subsumed, its arguments will occur in the index structure before any non-projected arguments. Therefore projected, subsumed tuples can be found by searching for the "non-projected" tuple but then truncating the search when the projected arguments have been found. It should be noted that this optimisation could be performed at source level by removing all references to the projected predicate and instead using the predicate on which the projection is based. Any arguments which do not appear in the projection are represented by arbitrary variables during queries. Although the removal of subsummed, projected tuples from an index structure is a valid optimisation, techniques for detecting these cases have not been investigated further in order to reduce the scope of the project.

3.1.4 Labelled Values for Functional Relationships

Another potential optimisation that reduces the number of index nodes in an index structure is to replace some index nodes with a single value. Dynamic argument indexes generally store multiple terms for an argument. However predicates may possess functional relationships between their arguments such that, given a subset of the arguments in a predicate, there is only one possible value for each of the remaining arguments (see [33] for a more formal description of functional dependency in the logic programming context). This relationship could be exploited in the index structure when the functionally dependent arguments are indexed after the arguments they depend on by representing each functional argument with a single labelled value rather than a complex argument index.

Unfortunately it has been found that proving functional relationships between groups of arguments from a static program is complicated by recursion and negation. This is discussed during the optimisation of destructive assignment in [23]. For this reason it is not yet possible to replace index nodes with single values. The future work section of Chapter 8 discusses how some common functional relationships may be detected using pattern matching on a program's source code and outlines necessary extensions to index structure implementations and the SDSL language (see Chapter 4) to facilitate this optimisation.

3.1.5 Multiple Argument Orders for Predicates

To increase the efficiency of searching, tuples may be indexed using multiple argument orders [17]. Although a single argument ordering may be the optimal choice for one mode of access used by the program, it can be inefficient for others. (Searching for tuples in index structure instances using argument orders is discussed later in this chapter.) Using argument orders that are specialised for each mode of access improves efficiency of searching (although it does increase the memory requirements of an index structure instance).

Alternative argument orderings are separated in the index structure using statically labelled branches. When a tuple is added down one index path it must be added down all relevant paths to ensure data integrity. If index paths end with the same sequence of arguments then it is possible to store these common arguments in the same index nodes, since they will contain the same set of tuples. Labelled branches that point to existing nodes in the index are used to join indexes together to form a graph, (as shown by the "*rejoin*" branch in the example index structure in Figure 3.6). This removes duplication of indexes thus reducing the number of insert operations and memory requirements.

However the benefit of optimised searching with multiple argument orders may still be outweighed by the additional insert operations or by the increased memory requirements. Whether the use of multiple argument orders improves performance is still dependent on how tuples are used by the program. Consequently the algorithms described later in this chapter which automatically define index structures do not create multiple argument orders. However the use of multiple argument orders in user-defined index structures is permitted.

To summarise, index structures are statically defined yet remain highly customisable. Customisation can occur (1) when choosing the order in which arguments are indexed, and (2) by combining or separating arguments from different predicates or sets. Many of the concepts discussed in this chapter are difficult to understand. The next section gives a detailed example to help clarify what is posible using Starlog's indexing structures.

Program:

```
p(X,Y,Z) <- q(X,Z), q(Z,Y), Z>Y, q(Y,N), Z>N, not(q(X,N)).
q(X,S) <- p(Y,X,Z), p(X,W,U), Z>U, p(V,X,T), Z>T, S is Z+1.
```

Stratification Order:

```
p(_,_,Z1)≫p(_,_,Z2)⇐Z1 > Z2        p(_,Y1,Z)≫p(_,Y2,Z)⇐Y1 > Y2
q(_,Z1) ≫ q(_,Z2) ⇐ Z1 > Z2         q(X1,Z) ≫ q(X2,Z) ⇐ X1 > X2
p(_,X,Z) ≫ q(X,Z)
```

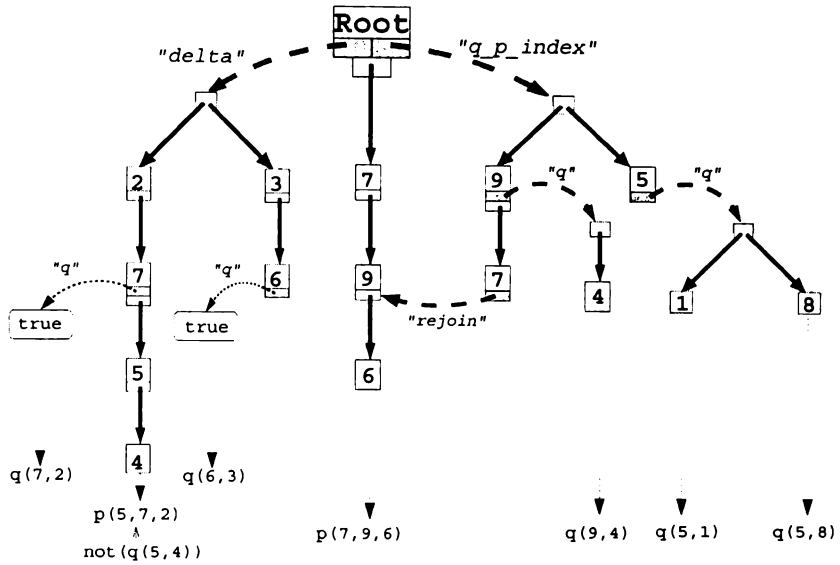


Figure 3.6: Example program and a possible index instance.

3.2 Example of an Index Structure for Starlog

To clarify the discussion on indexing, an example program and one possible index structure are given in Figures 3.6 and 3.7. The index structure shown here demonstrates what is possible using the indexing system that has been described previously in this chapter.

The top sections of Figure 3.6 give the definition of a Starlog program and its stratification order. Note that it is not necessary to understand the program or its stratification order to interpret this index structure. The program is included only as a reference to why such an index structure would exist. The lower section of Figure 3.6 gives an example of how a set of tuples could be stored in a Starlog index. To complete the example of the index structure, Figure 3.7 gives the schema for the index structure instance given in Figure 3.6.

A schema of an index structure provides a generalised model of all instances of an index structure. That is, the schema specifies how all tuples that can be generated by a program will be stored in a given index structure. A schema

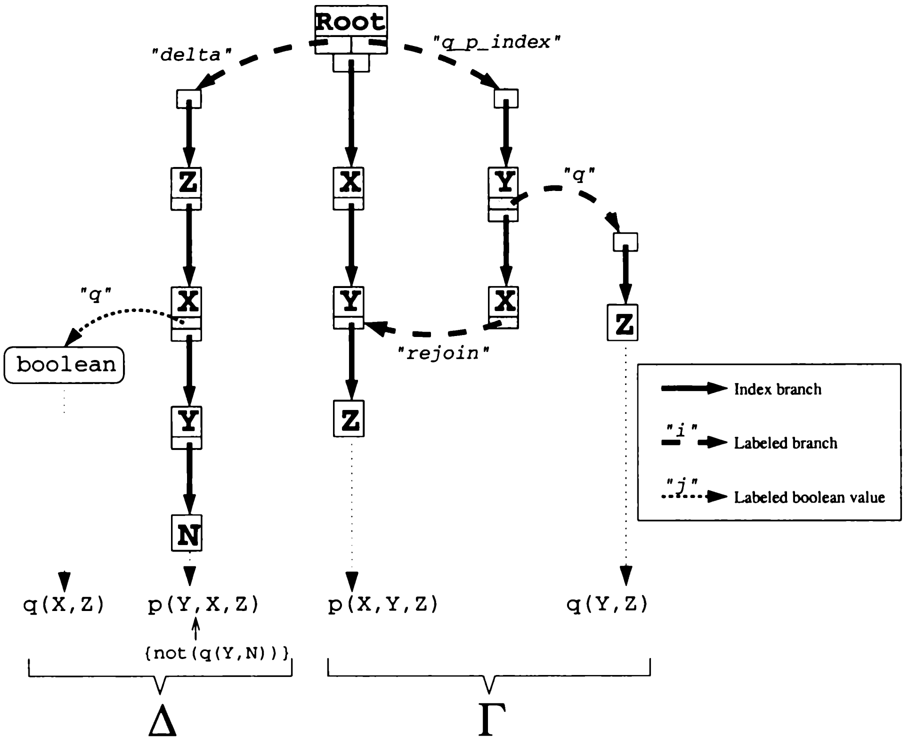


Figure 3.7: Index schema.

of an index structure is an index structure that holds a tuple from each of the program's predicates (distinct in Δ and Γ), where the tuple's arguments are replaced by variables capable of holding all possible bindings. Schemas are used in the remainder of this thesis to graphically describe index structures. For more examples of schemas of index structures see Appendix C.

The index structure instance in Figure 3.6 and schema in Figure 3.7 present an index structure capable of holding tuples from the `q/2` and `p/3` predicates. These tuples can exist in both the Δ and Γ sets. To distinguish between these two sets, all tuples that are stored in the Δ set are stored down the static labelled branch named "`delta`" that originates from the root node. All tuples in the Γ set are stored down the other paths that originate from the root node. This partitioning is indicated in Figure 3.7 using the Δ and Γ symbols.

Of the tuples in the Δ set (those stored down the "`delta`" path), the arguments of `q/2` tuples are subsumed by the arguments of `p/3` tuples. That is, when the arguments of `q/2` tuples are stored in the index, the path used to store `q/2` tuples is a subset of the path used to store `p/3` tuples. To distinguish `q/2` tuples from `p/3` tuples, a labelled boolean value exists at the point in the index structure where all the arguments of `q/2` tuples have been indexed. Consequently, for the Δ set, the boolean value labelled "q" is set to *true* to indicate the presence of `q/2` tuples, and *false* otherwise.

From the program, when `p/3` tuples are stored in the Δ set they are constrained by an unresolved negated goal (for reasons discussed in Chapter 2). To allow for the negated goal to be queried at some point in the future, the bindings of variables used by negated goals must be maintained. For this reason, although there are only three arguments in any `p/3` tuple, the binding of the extra variable that is required for the negated goal is also stored.

Tuples in the Γ set are located down those paths from the root node that do not follow the "`delta`" static labelled branch. All `q/2` tuples are stored down the "`q_p_index`", where the first argument is indexed, and then down the "q" static labelled branch, where the second argument is indexed. `p/3` tuples are indexed in the Γ set in two ways. The series of dynamic argument indexes that start at the root node index the first, the second and then the third arguments of `p/3` tuples. However, `p/3` tuples are also stored down the path starting with the "`q_p_index`" static labelled branch at the root node. Down this path, `p/3` tuples are indexed initially by their second argument, then by their first argument. To reduce the size of the index structure, indexing of the third argument of `p/3` tuples is performed in the same dynamic argument index for both paths.

3.3 Operations on Starlog Index Structures

We now outline how to perform a few of the more frequent operations using Starlog's indexing system.

3.3.1 Searching for Tuples in Starlog's Indexes

To search for a tuple in an index structure instance requires searching each level for argument bindings unifiable with those in the query term, starting from the *Root* index node and ending at a leaf. The order that the functor and arguments are indexed is given in the index path definitions of each predicate. Following the index path definition, when the value is an instantiated argument the index branch associated with this value is followed to its "child" index node (the sub-index). Here the next term identified by the index path definition is searched for. Similarly, statically labelled branches are followed when their label occurs in the index path definition. If at any time a branch is undefined in a dynamic index or the sub-index of a labelled branch is undefined (i.e. is a null value) the search process is aborted and fails.

When attempting to satisfy a goal which contains a free variable for an argument, the argument unifies with any term in the appropriate dynamic argument index. In fact, to ensure that all possible output is produced from the program's rules, all possible bindings for the argument must be used. The chosen technique to ensure every binding is considered is backtracking. Initially the free variable in the goal is unified with the first term in the appropriate argument index. The search process continues (as does any subsequent evaluation) with this variable binding until the rule succeeds or fails. When backtracking occurs the variable binding made previously is removed. Unification occurs between the variable and the next term in the argument index. This process continues until all possible terms in the argument index (and therefore all possible matching tuples) have been used.

When a leaf node is encountered during a search all the arguments of the tuple will have been found. If the leaf node is a labelled boolean value, a true value indicates the sought tuple is present in the index structure instance whereas false indicates its absence. Otherwise, if the leaf node is not a labelled boolean value, the sought tuple is present.

3.3.2 Inserting Tuples into Starlog's Indexes

The functor and arguments of tuples are inserted into an index structure instance in the order specified by the predicate's index path definition. If an argument value to be inserted does not exist in the argument index then a new index branch associated with this value is added. A new "child" index node (the sub-index) is created at the end of the new branch, in which the next term identified in the path definition is inserted. In the event that the value of the argument already exists in a dynamic index, the value's existing branch is followed to reach the relevant sub-index where the insertion of the next term continues.

When the path definition indicates a statically labelled branch is followed, the labelled branch will exist in the current index node reached during the insertion process (the index structure would be invalid if it did not exist). However if this is the first insertion to follow this labelled branch then its sub-index will be undefined (i.e. the branch will point to a null value). Before the insertion process can proceed a new index node is created as the sub-index of the labelled branch. When the sub-index of the labelled branch is defined, the next term specified by the index path definition is inserted into this index node.

After the last argument has been indexed, it may be necessary to distinguish the new tuple from any others that have been indexed using the same path. A tuple that is subsumed by others in the index must assign "*true*" to the labelled boolean value that represents this tuple.

3.3.3 Deleting Tuples from Starlog's Indexes

To delete a tuple we find the leaf which defines the presence of that tuple. In the case where the leaf is a labelled boolean value, the value is changed to $false$.

Individual argument values can be removed from dynamic argument indexes. When all argument values have been deleted from an argument index the index is referred to as *empty.*

Statically labelled branches can never be removed from their parent index – they originate from *static* indexes. However the sub-index beneath a labelled branch can be removed by setting it to a null value.

To remain efficient an index structure instance should remove any redundant index nodes that exist after tuples have been deleted (called *"dead" branches* in [17]). A *redundant* index node contains an empty argument index, all labelled boolean values hold $false$ and any labelled branches point only to null values. The process of deleting one redundant index node may cause the parent to become redundant. The single exception is that although the *Root* of an index structure instance may become redundant, it is never deleted.

To find the parents of index nodes requires maintaining references to previous index nodes encountered in the path. Because the definition of the index structure is static there is a finite and constant number of index nodes encountered when finding any tuple from a given predicate, making the use of a stack unnecessary. Instead a reference to each index node is maintained in a different variable. (This approach also leads to additional code optimisations discussed in Chapter 6).

When an index node J has multiple parents (i.e. when paths to the same predicate converge) deletion proceeds as normal. If J becomes redundant then all references to J in all parent indexes must be removed. To find all parents of a redundant index node extra searching may be required to follow all paths that include the redundant index node.

3.3.4 Comparisons to Relational Databases

Most relational databases are disk-based, allowing persistent storage in files [39]. In contrast Starlog uses an in-memory database. The values of a Starlog tuple's arguments are stored at each index, making it unnecessary to store full tuples as records. As a consequence, many of the issues associated with maintaining database files (even when kept in main-memory) are not relevant to the Starlog database.

Yet the concept of nested indexes is similar to hierarchical files in [74]. In this system records are indexed on various fields, one at a time, to form a tree structure. Indexing on one field may lead to a different subset of later fields in the same way that indexing over functors often results in a different set of arguments in the sub-indexes.

In general, Starlog tuples do not have a unique key field used to distinguish different records. Two Starlog tuples are distinct if any of their arguments (or functors) are different. Therefore it is often impossible to uniquely identify tuples using a single key value. Like Clustering Indexes in relational databases where a key value may refer to more than one record [39], a value stored in a dynamic index may represent more than one tuple. However, Starlog uses nested indexes (not to be confused with multilevel indexes used in relational

```
<index_path>  ::= index <set> <abstract_tuple_with_negation>
                   <branch_list> '.'
<set> ::= delta | gamma
<abstract_tuple_with_negation> ::= <abstract_tuple_definition>
                                     [ '<-'  <negated_goals>]
<abstract_tuple_definition> ::= <functor> [ '(' <variable>
                                              {',' <variable>} ')' ]
<negated_goals> ::= <negated_goal> {',' <negated_goal>}
<negated_goal> ::= 'not(' <abstract_tuple_definiton>
                            {',' <built_in>} ')'
<branch_list> ::= '[' <branch> {',' <branch>}
                                [',' <boolean_value> ] ']'
<branch> ::= <argument_index> | <static_branch> | <name_definition>
<argument_index> ::= <variable>
<static_branch> ::= 'branch(' <label> ')'
                  | 'join(' <label> ',' <label> ')'
<name_definition> ::= 'name(' <string> ')'
<boolean_value> ::= 'boolean(' <label> ')'
```

Figure 3.8: BNF grammar of index path definitions.

databases) to distinguish similar tuples – each index operates on one of the fields (arguments and functors) and points to a sub-index that indexes the next field. By indexing on all fields, tuples are uniquely identified.

In the same way that multiple, secondary indexes (or multiple hierarchy files) are used in relational databases to index records with multiple keys fields, multiple index paths are used in a Starlog index structure to access the different fields of a tuple more efficiently.

3.4 Index Structure Path Definitions

Every Starlog program requires an instance of an index structure to store tuples at run time. However the static nature of the index structure requires it to be defined at compile time. Although much of the design of index structures can be automated it should also be possible for programmers to specify the design for fine tuning. For this reason an index structure specification syntax is introduced.

Starlog index structure instances store all tuples from the same predicate (distinct in Δ or Γ) alike. Each predicate stored in Δ and Γ has an *index path definition* that lists which branches are taken to get from the root to a leaf when storing tuples from this group.

Figure 3.8 gives path definitions in BNF format. (Non-terminals `<variable>`, `<functor>`, `<built_in>`, `<label>` and `<string>` have not been defined since they conform to definitions in Appendix A, conform to Prolog's standard syntax, or are explained in the text.) Figure 3.9 gives an example of the path definitions used to describe the index structure in Figure 3.6.

By convention, programmer defined index path definitions are placed after the stratification priorities but before any program rules in a Starlog program file. This grouping allows immediate comparison between all path definitions, reducing design errors. The order of path definitions in a program's source code is irrelevant.

The structure of index path definitions is similar to the stratification priorities described in Chapter 1. An index path definition starts with the keyword `index`. Each path definition then uses the keywords **delta** or **gamma** to indicate which of the Δ or Γ sets will be indexed in this particular definition. Next, the predicate to be indexed is given as an abstract tuple definition where each argument is represented by a locally unique variable name (see Appendix A for more details on abstract tuple definitions). Because tuples in Δ may be constrained by negated goals, these are included in the abstract tuple definitions. In any negated goal, arguments which are existential variables and arguments bound by a value in the head tuple are represented using an underscore ("_"), whereas other arguments are represented by unique variables. When tuples from the same predicate can be constrained by different negated goals depending on the rule which generated them, each different constraint requires a different index path definition. After the predicate to be indexed has been sufficiently identified, the path through the index structure is given as a list.

There are four elements that can exist in the path: argument indexes, labelled boolean values, labelled branches, and index names. Argument indexes discriminate on the values of a tuple's arguments. To indicate when these occur in a path we use a variable that occurs in the abstract tuple definition. Since all variables in an abstract tuple definition are unique, each represents an argument without ambiguity.

Labelled boolean values distinguish tuples from each other. These only occur at the end of a path definition as `boolean_value/1` relations, where the value's label[1] is the parameter.

There are two types of labelled branches in path definitions. Labelled branches that point to a sub-index are identified using `branch/1` relations where the static label is the parameter. The second type of labelled branch also points to a sub-index however these branches point to sub-indexes that are already defined elsewhere in the index structure. This is useful when multiple index paths to the same predicate converge to reduce the size of the index. This second type of labelled branch points to an index specified in another path definition. `join/2` relations identify branches that point to existing indexes. The parameters used here are a label, to identify the branch, and the name assigned to the destination node in the index structure.

For nodes to be named so they can be referred to in `join/2` relations in other path definitions, `name/1` relations hold a globally unique name assigned to the index node that immediately follows it in a path definition.

Any predicate in either Δ or Γ may have any number of index path definitions. If a predicate does not have a path definition for Δ or Γ then tuples from this predicate will not be stored in this set. By omitting path definitions of predicates from a program, the exclusive trigger and non-trigger rule head optimisations from sections 2.3.5 and 2.3.6 can be achieved. Each path definition stores predicates according to one argument ordering whereas multiple path definitions allow indexing using different argument orderings.

[1]For implementation reasons, labels of any labelled boolean value or labelled branch must obey Java's variable naming rules where labels can consist of any combination of letters, numbers and underscore characters, but must begin with a letter.

```
index  delta   p(Y,X,Z) :- not(q(_,N))  [branch(delta),Z,X,Y,N].
index  delta   q(X,Z)     [branch(delta),Z,X,boolean(q)].
index  gamma   p(X,Y,Z)   [X,Y,name(lastarg_p),Z].
index  gamma   p(X,Y,Z)   [branch(q_p_index),Y,X,
                                 join(rejoin,lastarg_p),Z].
index  gamma   q(Y,Z)     [branch(q_p_index),Y,branch(q),Z].
```

Figure 3.9: Path definitions corresponding to index structure in Figure 3.6

Although index path definitions can be supplied by Starlog programmers, it is often preferable for the compiler to automatically infer appropriate definitions. This is especially useful during initial code development (when efficiency is of a lesser concern than correctness) or for Starlog users who do not know (or do not want to know) how their program stores or uses run time data. Next we present a heuristic for automatically defining an appropriate (and relatively efficient) index structure from a Starlog program file.

3.5 Automatic Construction of Index Structures

Constructing an index structure capable of storing all the tuples in a program is trivial. Each predicate appearing in a program may be distinguished by labelled branches at the root of the index structure. (For each predicate one labelled branch is used for tuples in Δ and one labelled branch for those in Γ.) Arguments can then be indexed one at a time in a simple left to right order. Although sufficient for correct execution, the resulting index structure will rarely be efficient since there is no sharing of arguments indexes between predicates, and no attention has been given to how predicates are used by the program.

Yet to construct a very efficient index structure at compile time is difficult. The most efficient index structures may use multiple index paths when there are different modes of access to a predicate. However, as has been described earlier, a tradeoff exists that is dependent on run time data. Therefore it is difficult to know when to apply this feature at compile time.

It is possible that by executing the program (perhaps using the naive index structure described previously) that run time statistics can be gathered to help make a more informed choice about when multiple indexes are useful. However there are two cases when this may prove inadequate or even misleading. When programs receive input from an external source, such as standard input or from a file, then run time performance may change as the input changes. Also, when programs are non-terminating, gathering run time characteristics for the entire run is impossible so a subset of the execution must be used (i.e. termination must be forced). However, depending on the subset, the run time data collected may not be representative of actual run time characteristics. Because of these problems we do not use run time data to influence the design of the index structure. Instead we use a very simple but effective heuristic to design the index structure from only the Starlog source code. In Chapter 7 the problems associated with running a program to measure performance reemerge when using run time data to help select efficient data structures. The effectiveness of such approaches is evaluated and the conclusions reached may have implications in the context of index structure design. However the use of run time data to influence the design of the index structure is considered beyond the scope of this project and is addressed in Chapter 8 as future work.

There are many ways that an index structure could be constructed based only on the program. Some approaches may prioritise space rather than time complexity or optimise index structures for some operations rather than others.

The following sections describe a simple approach to automatically constructing an efficient index structure by referring to the stratification order and mode information to find efficient argument index orders for all predicates in Δ and Γ. This system prioritises efficient searching while combining compatible indexes (to reduce space or facilitate code optimisations discussed in Chapter 5) is a secondary concern. Indexing the same tuples using multiple paths is not used. The Δ and Γ sets are separated by a labelled branch ("*delta*") at the root of the index structure. This allows optimised searching in each set, however there is no argument sharing between Δ and Γ requiring more operations to move tuples from Δ to Γ. Initially each predicate in Δ and Γ is treated independently of others while efficient argument index orderings are inferred. All argument index orders of predicates in Δ and Γ are then merged together into a single index structure using labelled branches to separate non-comparable arguments from different predicates and boolean values to distinguish subsumed predicates.

3.5.1 Efficient Argument Index Ordering for Δ

The efficient ordering of argument indexes for the Δ set is now discussed. As seen in Chapter 2, when a stratification order is present in a bottom-up program Δ stores new tuples and extracts tuples individually when they become minimal elements.

When inserting a tuple into or deleting a tuple from Δ, the efficiency differs only slightly when the order of arguments is changed because only one path is traversed through an index. However the efficiency of finding the minimum element in Δ is much more dependent on the order of arguments and so optimising Δ for this operations is important.

The minimal tuples in Δ are found by working through the argument stratification priority and taking the subset of tuples that are in the minimal strata at each level. If tuples are indexed in the same order as their argument stratification priority then finding the minimum tuples requires exploring a single path. An index order that ignores the argument stratification priorities will still locate minimal tuples but will require backtracking to explore alternative paths down which minimal tuples may exist. In the worst case, where the first element in the argument stratification priority is the last to be indexed, an exhaustive traversal of the Δ index may be required.

Figure 3.10 demonstrates of how changing the argument index ordering for a set of tuples affects the performance when finding the minimal element. The `p/2` tuples are stratified primarily on their second argument and then, assuming the second arguments of two of these tuples are identical, on their first argument. Index instance `(a)`, which indexes the arguments of `p/2` tuples left-to-right, requires exploring all the branches originating from the root node in order to

Stratification Order:
p(_,X) ≫ p(_,Y) ⇐ X > Y
p(X,Z) ≫ p(Y,Z) ⇐ X > Y

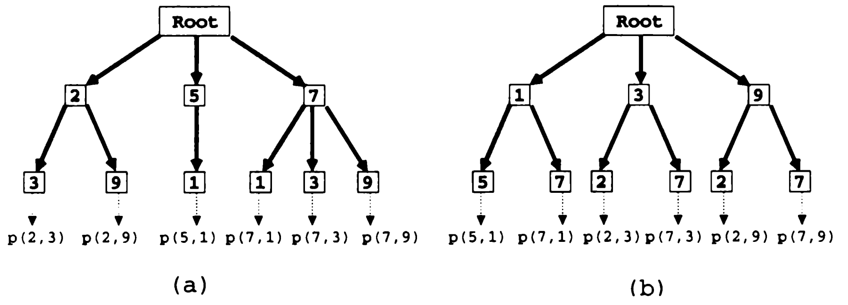

Root
2
5
7
3
9
1
1
3
9
p(2,3) p(2,9) p(5,1) p(7,1) p(7,3) p(7,9)
(a)
Root
1
3
9
5
7
2
7
2
7
p(5,1) p(7,1) p(2,3) p(7,3) p(2,9) p(7,9)
(b)

Figure 3.10: Finding the minimum stratified element using two different argument orders.

find the minimum value for the tuple's second argument. As a result, three paths are fully explored from the root to a leaf. However index instance (b), which indexes the two arguments from right-to-left, can conclusively determine the minimum argument at each level, thus requires exploring only one path.

3.5.2 Efficient Argument Index Ordering for Γ

Efficiently indexing of tuples in the Γ set requires a different approach. Operations in Γ involve tuple inserts and tuple searches. Searching for the values of a tuple's arguments in an index can take two forms: known value lookup and unknown value lookups. A known value lookup is used when the value of an argument in a goal has been instantiated. In an index this can be achieved using (at worst) a semi-deterministic operation (where the sought value either exists or does not). An unknown value lookup attempts to instantiate a variable with all possible values of an argument (one value at a time) that exist in an argument index. This operation uses backtracking to ensure all alternatives are considered.

The design of the index structure requires all arguments to be indexed sequentially in some order specific to each predicate. Considering the two forms of lookups, the most efficient strategy is to perform known value lookups earlier in the index structure before unknown value lookups.

To prove this, let us consider the case where A and B are dynamic argument indexes where the A index is used exclusively for unknown value lookups and the B index is used for known value lookups. An index structure that positions index A before B will perform an unknown value lookup first during a search. The unknown value search will instantiate variable X and continue searching B for a known value before backtracking occurs and X is instantiated with another value. In this situation when N instantiations of X are possible in A, N known value lookups will be performed in B. The alternative, where the B index occurs before A, will perform only one known value lookup in B, after which (assuming the known value lookup is successful) X is instantiated N times. When $N > 1$

there is a clear improvement in efficiency. Therefore, if we assume indexes hold more than one element (a reasonable assumption at this stage of compilation), performing known value lookups before unknown value lookups in the index structure will reduce the number of operations performed at run time.

Known value and unknown value lookups are used when an argument in a body goal is ground or free, respectively. This is equivalent to the *mode* of a variable before the goal has been satisfied. With the exception of trigger goals (which are always the first goal to be instantiated) and negated goals whose evaluation is delayed, the remaining body goals and built-ins are evaluated in a left-to-right order. Arguments occurring in a goal are bound if they are a constant or are a variable that has been ground previously in the rule. Otherwise arguments are free variables. When describing the mode of variables for this analysis, as is usual in the mode analysis literature, a program's variables are adorned with a '$-$' when a variable is free before the goal is satisfied or with a '$+$' to indicate it is ground. This mode information is inferred automatically and does not have to be provided by the programmer.

Using a moded Starlog program, an indexing order of arguments can be chosen to maximise the efficiency of queries in Γ. For any given predicate, a count is made of the number of ground occurrences of its arguments in the moded program. Arguments that are more often ground are indexed earlier in the index structure. Arguments that are ground the same number of times are ordered arbitrarily.

Next we define a syntax to describe these argument orders in preparation for the automatic definition of an index structure.

3.5.3 Argument Order Definitions

To express the order of each predicate's arguments in the index, a variation on the path definition syntax is used to create an *argument order definition.* By including argument order definitions in programs, programmers can control indexing of some or all predicates yet do not have to specify the complete index structure. Alternatively, argument order definitions can be generated automatically using the program analysis outlined in the previous sections. Later in this section we give the automated process as an algorithm.

Argument index orders may be considered an abstraction of path indexes without labelled boolean values and with (in general) fewer labelled branches. Examples of argument order definitions can be seen in Figure 3.12. (This example demonstrates the process of automatically defining these definitions and is discussed in detail later.) By convention, argument order definitions are included between the stratification priorities and the rules in a Starlog program's source code. An argument order definition is identified in a Starlog program by the keyword `order`. Like path definitions, the keywords `delta` and `gamma` are used to identify the set being indexed. The predicate to be indexed is given as an abstract tuple definition. The order that arguments should be indexed is given as a list of variables (corresponding to variables in the abstract tuple definition) and labelled branches. Labelled branches are used only to distinguish the Δ set and to distinguish between different predicates when tuples are stratified on their predicate names.

The algorithm in Figure 3.11 uses a moded bottom-up program and argument stratification priorities to infer efficient argument order definitions for all predicates, for both Δ and Γ. The algorithm takes each predicate appearing in the head of a rule and generates an efficient argument ordering for Γ by counting the number of ground occurrences each argument has when used in body goals. Arguments are sorted so those that are more often ground occur earlier in the argument order definition.

Argument order definitions for Δ are distinguished from Γ using a `branch(delta)` branch as the first element. Although this reduces index sharing between the Δ and Γ sets, it leads to a much simpler evaluation system than when these indexes are combined. (The class of program where indexes of Δ and Γ are combined are discussed in Chapters 5 and 8.) For each predicate, the argument order definition for the Δ set initially indexes arguments as they appear in the argument stratification priority. The static fields of a predicate that are used in the stratification order (such as the predicate names) that sometimes occur in the argument stratification priorities become the labels of labelled branches. Because some arguments do not appear in the argument stratification priority, the additional arguments are appended to the argument order definition in a left to right order. When a head predicate is constrained by unresolved negated goals, any negated goal arguments that are not bound to an argument in the head predicate must be stored separately in Δ. These arguments are appended to the end of the argument order definition. By including the extra arguments for negated goals after the head predicate's arguments and consistently indexing unordered head arguments left to right, any instances of a predicate that are constrained by negated goals and instances that are unconstrained will have the same argument order prefix. When constructing an index structure from the argument order definitions the shared prefix will result in shared indexes. Exact duplicates of argument order definitions are removed using the set insertion operation.

Figure 3.12 gives an example of automatically generating efficient argument order definitions. The modes of the non-trigger body goals have been summarised in the table. Arguments in Γ are ordered according to the number of ground occurrences in body goals, where those arguments that are more often ground occur first. Argument index orders for predicates stored in Δ are determined by the predicate's stratification priority. In this example all tuples are stratified primarily on one of their arguments (the last argument for `q/2` and `p/3` and the first argument of `r/2`) after which the predicate name is used (where, $\texttt{r(A,\_)} \ll \texttt{q(\_,A)} \ll \texttt{p(\_,\_,A)}$). The argument order definitions for Δ are distinguished from the Γ set using a `branch(delta)` term. Following this the arguments of each predicate are ordered according to the predicate's stratification priority, where variables representing arguments remain unchanged in the argument order definition, and constant values (such as predicate names) become the labels of labelled branches. The remaining arguments in a predicate that do not occur in the predicate's stratification priority now appear in the argument order definition in a left to right order. Finally, for the head predicate `r(X,Y)` that is constrained by unresolved negated goal `not(q(Z,Y))`, the additional argument (`Z`) that is necessary to instantiate the negated goal is appended to the argument order definition.

Algorithm

let P denote the set of moded triggered program rules

let T denote the set of all head predicates used in P together with any unresolved negated goals. All arguments are represented by different variable names (except arguments in negated goals that are already bound by a head argument)

let $head(x)$ give the only head predicate of $x \in T$, i.e. ignore unresolved negated goals

let $negbody(x)$ return the unresolved negated goals of $x \in T$

let $arity(y)$ be the arity of a predicate y

let $arg(y, i)$ be the ith argument of predicate y

let $args(y)$ be the set of all arguments in predicate y

let $strat(y)$ be the sequence of terms in the argument stratification priority for predicate y (where each term is either a constant or a unique variable name representing an argument of y)

let $ground(h, i, P)$ be the number of ground references to the ith argument of predicate h in program P

let $Vars$ be the set of all variable names, excluding "_"

let $sorted(X)$ be the descending sorted version of list X, where X contains elements structured (A, B) and in $sorted(X)$ these are sorted on field A

let $+$ be list/string concatenation

```
|let $ArgumentIndexOrders = \emptyset$
|for all $r \in T$ loop
|  let $h = head(r)$
|  let $G = \emptyset$
|  for $i = 0$ to $arity(h)$ loop
|    $G = G \cup \{(ground(h, i, P), arg(h, i))\}$
|  end loop
|  let $OrderGamma = \emptyset$
|  for all $(f, v) \in sorted(G)$ loop
|    $OrderGamma = OrderGamma + \{v\}$
|  end loop
|  $ArgumentIndexOrders = ArgumentIndexOrders \cup \{$``order gamma ''$+h+$
                                    `` ''$+OrderGamma\}$
|  let $OrderDelta = \{branch(\texttt{delta})\}$
|  for all $k \in strat(h)$ loop
|    if $k \in Vars$ then
|      $OrderDelta = OrderDelta + \{k\}$
|    else
|      $OrderDelta = OrderDelta + \{branch(k)\}$
|    fi
|  end loop
|  for all variables $v \in args(h) \wedge v \notin strat(h)$ loop
|    $OrderDelta = OrderDelta + \{v\}$
|  end loop
|  for all $not(n) \in negbody(r)$ loop
|    for all variables $u \in args(n)$ loop
|      if $u \in Vars$ then
|        $OrderDelta = OrderDelta + \{u\}$
|      fi
|    end loop
|  end loop
|  $ArgumentIndexOrders = ArgumentIndexOrders \cup \{$``order delta ''$+r+$
                                    `` ''$+OrderDelta\}$
|end loop
|exit
```

Figure 3.11: An algorithm to find efficient argument order definitions.

Moded Program Rules:
(The '+' and '−' preceding each variable represents the mode immediately before the goal has been satisfied assuming a left-to-right sideways information passing strategy)

p(X,Y,Z) ← q(**+X**,**+Y**), r(−Z,+Y), p(+Y,+Z,−W), +W < +Y.
q(X,Z) ← r(**+Y**,**+X**), p(−Z,+Y,−W), q(−V,+W), +W < +X.
q(X,Y) ← r(**+X**,**+Y**), p(−Z,+X,−W), +W < +Y.
r(X,Y) ← p(**+Y**,**+X**,**+Z**), not(q(+Z,+Y)), +Z < +Y, +Y < +X.

Argument Stratification Priorities:

```
stratify  p(_,_,A)   [A, p].
stratify  q(_,A)     [A, q].
stratify  r(A,_)     [A, r].
stratify  r << q.
stratify  q << p.
```

Goal arguments searched in Γ	Free	Ground
q/2 Arg 1	1	1
Arg 2	0	2
r/2 Arg 1	1	0
Arg 2	0	1
p/3 Arg 1	2	1
Arg 2	0	3
Arg 3	3	0

Argument Index Orders:

```
order  gamma  q(X,Y)                 [Y,X].
order  delta  q(X,A)                 [branch(delta),A,branch(q),X].
order  gamma  r(X,Y)                 [Y,X].
order  delta  r(A,Y) ← not(q(Z,_))   [branch(delta),A,branch(r),Y,Z].
order  gamma  p(X,Y,Z)               [Y,X,Z].
order  delta  p(X,Y,A)               [branch(delta),A,branch(p),X,Y].
```

Figure 3.12: Example of generating argument order definitions. (Triggers are shown in bold).

3.5.4 Combining Index Structures

Argument order definitions can combine to produce a single index structure. There are many approaches that will produce a correct index however we present a scheme that maximises sharing of argument indexes without contradicting the argument order definitions.

Each argument order definition can be interpreted as an index structure holding one predicate. Multiple index structures are combined by repeatedly applying a simple grafting algorithm similar to those used to extend decision trees [113]. The output is a set of index path definitions representing a single index structure.

The algorithm used to combine two index structures is given in Figure 3.13 but its operation is best demonstrated graphically. Figure 3.14 gives an example where four index structures are combined. The disjoint index structures that each hold a single predicate are given as argument index orders. The types of arguments in each predicate are given as a type declaration statement. It is assumed type declarations are inferred automatically from the program (see Chapter 1).

When combining two indexes, one index structure will be the *base structure* to which extra branches will be added. (Base structures are represented as a set of index path definitions.) The other index (a single path definition) is called the *new structure*.

Initially the base structure and new structures are arbitrarily selected from the set of argument order definitions. In the example in Figure 3.14 the `p/2` index is the first base structure and the `q/2` index is the first new structure. As shown in Figure 3.14 (a), the roots of both the base and new structures are merged together. When both the base and new structures have an argument index (represented by a variable in the argument index order) at the next level, the types of the arguments are compared. If the two arguments have comparable types then the two indexes are combined to increase sharing. In Figure 3.14 (a) the `X` arguments from both `p/2` and `q/2` are combined in this way. In the event that the two types of arguments are incompatible for sharing (as with arguments `Y` and `Z`) these must be separated using a labelled branch. The labelled branch "*id*0" is automatically generated for this purpose. Indexing of all remaining arguments in the new structure (e.g. `Z`) occurs below the new labelled branch.

In Figure 3.14 (b) the root and `X` index nodes of `r/2` are merged with those in the base structure (now indexing both `p/2` and `q/2` predicates). When attempting to merge the second index (holding `Z` arguments) with the base structure there is a clash of variable types between `Z` (a string) in `r/2` and `Y` (an integer) in the base structure. Rather than generating a new labelled branch to separate `r/2` tuples in the base index, the existing labelled branches in the base structure are explored. The branch labelled "*id*0" already specifies an index storing string values so this branch is shared between the `q/2` and `r/2` tuples. All arguments of `r/2` have now been indexed in the base structure. However, the `r/2` and `q/2` predicates now have exactly the same index paths (i.e. they are both indexed on their first argument, both follow the labelled branch "*id*0" and finally are both indexed on their second argument). This means that there is no way to distinguish `q/2` tuples from `r/2` tuples in the index structure – their index paths are subsumed by each other. To indicate the presence of tuples from each predicate, labelled boolean values are added to the final index nodes of subsumed tuples.

```
Algorithm
let $B$ denote set of path definitions in the base structure
let $n$ denote the new path definition
let $size(X)$ be the number of elements in the list of path definition X
let $X[i]$ denote the $i$th element in the list of path definition X
let $Vars$ denote the set of all variable names, excluding "_"
let $type(X)$ be the type of an argument represented by variable X
let $uniqueID$ generate a unique label each time it is used
let $-$ denote the set exclusion operator

let $i = 1$
let S = B
|main: loop while $i <= size(n)$
|  for all $s \in S$ loop
|    if $size(s) >= i$ then
|      if $s[i] \in Vars \land n[i] \in Vars$ then
|        if $type(s[i]) = type(n[i])$ then
|          $S = S - \{t \mid t \in S, t[i] \notin Vars \lor (t[i] \in Vars \land type(t[i]) \neq type(n[i]))\}$
|          $i = i + 1$
|          continue main
|        fi
|        for all $b \in S$ loop
|          if $b[i] = branch(A)$ then
|            if $b[i+1] \in Vars$ then
|              if $type(b[i+1]) = type(n[i])$ then
|                insert $branch(A)$ into $n$ before $n[i]$
|                $S = S - \{t \mid t \in S, t[i] \neq branch(A)\}$
|                $i = i + 1$
|                continue main
|              fi
|            fi
|          fi
|        end loop
|        let $A = uniqueID$
|        insert $branch(A)$ into $n$ before $n[i]$
|        $B = B \cup \{n\}$
|        exit
|      else if $s[i] = branch(A) \land n[i] = branch(A)$ then
|        $S = S - \{t \mid t \in S, t[i] \neq branch(A)\}$
|        $i = i + 1$
|        continue main
|      else
|        $S = S - \{s\}$
|      fi
|    else
|      if $size(n) = size(s)$ then
|        let $A = uniqueID$
|        add $boolean\_value(A)$ to the end of $n$
|      fi
|      let $C = uniqueID$
|      add $boolean\_value(C)$ to the end of $s$
|      $B = B \cup \{n\}$
|      exit
|    fi
|  end loop
|  $B = B \cup \{n\}$
|  exit
|end loop
|for all $s \in S$ loop
|  if $size(s) = size(n)$ then
|    let $A = uniqueID$
|    add $boolean\_value(A)$ to the end of $s$
|  fi
|  let $C = uniqueID$
|  add $boolean\_value(C)$ to the end of $n$
|  $B = B \cup \{n\}$
|end loop
|exit
```

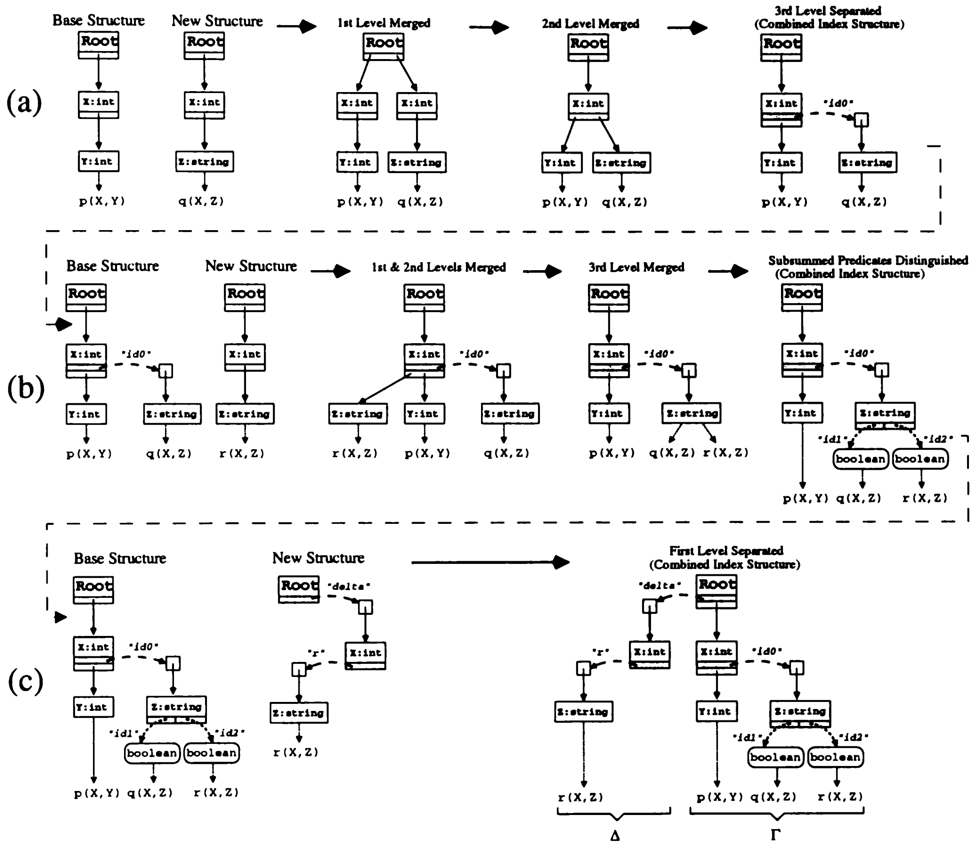
Figure 3.13: Merging Indexes Algorithm.

Argument Order Definitions:

```
order  gamma  p(X,Y)  [X,Y].
order  gamma  q(X,Z)  [X,Z].
order  gamma  r(X,Z)  [X,Z].
order  delta  r(X,Z)  [branch(delta),X,branch(r),Z].
```

Type Declarations:

```
type  p(integer,integer)
type  q(integer,string)
type  r(integer,string)
```



Combined Index Path Definitions:

```
index  gamma  p(X,Y)  [X,Y].
index  gamma  q(X,Z)  [X,branch(id0),Z,value(id1,boolean)].
index  gamma  r(X,Z)  [X,branch(id0),Z,value(id2,boolean)].
index  delta  r(X,Z)  [branch(delta),X,branch(r),Z].
```

Figure 3.14: Combining four example index structures.

The labels are automatically generated such that q/2 tuples are identified by the boolean value "*id1*", and r/2 tuples with "*id2*".

The final new index to be combined in Figure 3.14 (c) contains a labelled branch allowing distinction between the Γ and Δ sets. To combine this new index with the base index the roots are merged together. The labelled branch "*delta*" is added to the root node in the base structure when it does not already exist. The remaining index nodes from the new structure are added under the "*delta*" branch, making r/2 tuples in Δ distinct from all others.

The output of this algorithm is the set of index path definitions given at the end of Figure 3.14. These represent the final combined index derived in (c). It should be noted when using this index structure combining algorithm the order of arguments specified by the argument order definitions is always the same as those in the final index paths.

The order in which indexes are merged has an effect on the final index structure. Generally, indexes added earlier will have fewer labelled branches added to their paths, making accessing their associated tuples more efficient. Therefore, one way to optimise a program would be to sort the input indexes (or argument orders) so that those used more frequently occur earlier. Although the frequency of use of an index is associated with the number of times a predicate is used in the source code, it is also dependent on a program's run time characteristics (i.e. the amount of backtracking that occurs in a rule using a particular index). Moreover, statically labelled branches can be implemented very efficiently (see Chapter 6) such that the improvement in performance is only very slight. For these reasons the order that indexes are combined remains arbitrary.

3.6 Conclusions

It has been shown it is possible to create index structures that maintain those indexing properties of discrimination trees that are important to Starlog, yet whose structure is statically defined. Static definitions of these structures leads to more efficient implementations. However unlike discrimination trees, the order that a predicate's arguments are indexed can be selected to further improve efficiency.

The process of mapping predicates from both Δ and Γ to efficient Starlog index structures has been discussed. Starlog programmers can influence the design of the index structure by defining it directly using index path definitions, or by specifying argument order definitions. This is useful when programmers have an insight into the run time characteristics of their programs or when they wish to use advanced features such as multiple indexes. Alternatively, efficient argument orders can be defined automatically for tuples in both the Δ and Γ sets based on how predicates are used in the program. A single index structure can be created by merging all the argument order definitions. The final result will share indexes where possible, and ensure that different predicates are distinguished. It is noted that this is only one heuristic for determining an efficient index structure using simple program analysis and many alternatives are possible.

The use of an efficient index structure is paramount to efficient execution of bottom-up programs. A compiler that automatically infers an index structure frees the programmers from specifying the underlying representations of their data. As a consequence, Starlog programs remain abstract. In the next chapter we introduce a language that uses Starlog index structures to maintain run time data, and show how the use of these indexes allows high-level optimisation of programs.

Chapter 4

Starlog Data Structure Language (SDSL)

The Starlog Data Structure Language (SDSL) is an intermediate representation of Starlog programs that allows high level analysis and optimisation. In this chapter we describe SDSL in preparation for a description of Triggering Evaluation using Starlog index structures. Figure 4.1 shows SDSL to be the medium used to express Starlog programs at various stages of the compilation pipeline.

In a nut shell, SDSL provides facilities to "walk" through an index structure instance using semi-deterministic or non-deterministic operations whenever choice points are encountered. By examining the state of an index structure instance the state of the Δ and Γ sets can be determined, leading to correct evaluation of programs. SDSL also provides a selection of built-in operations for the manipulation and inspection of program variables.

SDSL exists at a level of abstraction between Starlog and the target language, Java. Unlike Starlog, whose programs consist of rules, SDSL programs consist of sequences of instructions. However, unlike Java, SDSL instructions can be deterministic, semi-deterministic or non-deterministic, where backtracking is used to access multiple solutions. Although SDSL instructions are polymorphic, type information is specified whenever a new program variable is declared.

Intermediate languages are frequently used during compilation or interpretation of programming languages. Early Lisp compilers transformed programs into LAP (Lisp Assembly Language) code which continued to use Lisp's high-level data structures before a final pass generated machine code [17]. [2] describes

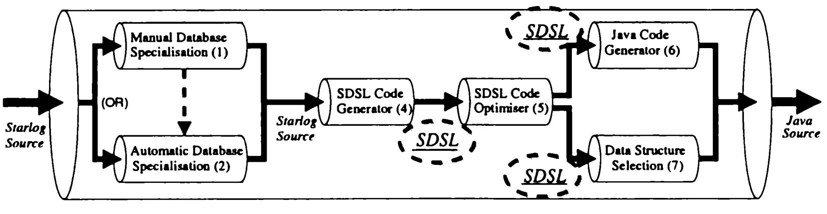


Figure 4.1: Use of SDSL in Starlog's compilation pipeline.

the three-address code intermediate representation where complex, built-up expressions are represented as binary and unary expressions with the aid of temporary variables. Some just-in-time Java compilers translate bytecode to these expressions to allow optimisation [5, 109]. The most well known intermediate language developed for the field of logic programming is WAM code. This instruction set implements the operations necessary to evaluate Prolog programs on a stack-based machine, with particular emphasis on the unification of terms [3]. Although originally interpreted by the Warren Abstract Machine, current versions of Prolog which generate WAM code tend to compile it to executable languages to improve efficiency. SDSL's point of departure from other intermediate languages is that SDSL uses the index structure instances described in the previous chapter to store run time data in addition to a stack (which is used for local program variables).

This chapter outlines the fundamental concepts and constructs of the SDSL language. The set of SDSL instructions is introduced with explanations about the purpose, syntax and semantics of each. The semantics of SDSL's instructions and control facilities are described using Prolog. Prolog is used because it has some of the properties of SDSL (such as backtracking and built-in operations) and because it is familiar to most logic programmers. How SDSL instructions are used to evaluate a Starlog program and their translation to executable code is given in Chapters 5 and 6.

4.1 SDSL concepts

Before the SDSL instruction set is given, some of the logic, control and documentation facilities are discussed.

4.1.1 Comments

Because SDSL is intended to be an intermediate language only produced by a compiler and interpreted by another compiler, annotating programs for programmers is not a priority. However, to clarify the SDSL programs presented in this thesis, a syntax for comments in SDSL programs is provided. As with Prolog, comments in SDSL begin with a "%" and continue to the end of the line.

4.1.2 Index Structure Definition

Recall from Chapter 3 that index structures can be described using a set of index path definitions. Index path definitions are included at the beginning of SDSL programs. After the index structure has been defined, the predicate that a particular index path belongs to is irrelevant, since each rule statically knows the types of branches to be followed to satisfy goals or assert rule heads. However in this thesis the predicate definitions remain in each of the index path definitions to clarify the link between the Starlog and SDSL programs.

4.1.3 Index Variables

The majority of instructions in SDSL provide a mapping from one index node to another in a Starlog index structure instance. To keep track of these nodes

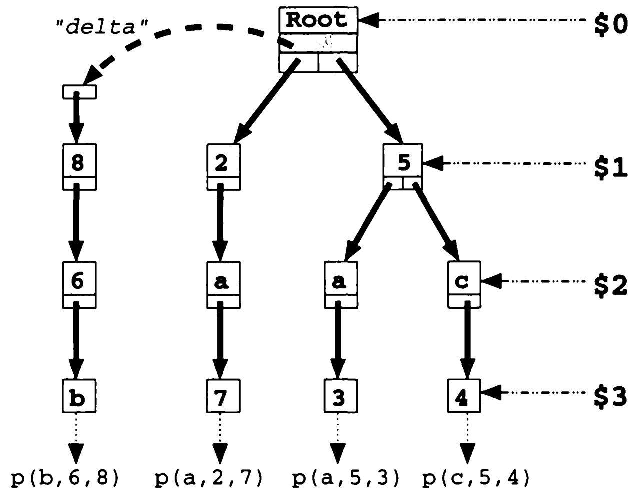


Figure 4.2: Example use of index variables $0 - $3 in a Starlog index.

in an SDSL program *index variables* are used. Index variables are distinguished from other variables in an SDSL program by a '$' prefix.

Like pointers in imperative programming languages, index variables reference nodes that already exist and are connected in the index structure instance. However, unlike pointers, index variables can not be updated after they have been assigned (although their bindings are removed when backtracking occurs).

Figure 4.2 shows the bindings of index variables that might result from traversing one path of an index structure instance. The convention is that all index variable names are numbers. The first index variable used during the traversal in Figure 4.2 is `$0` which points to the root index node. For programs to access the root of the index structure from any context the `$0` index variable is globally defined. This is the only global index variable. Index variables `$1`, `$2` and `$3` are specific to this traversal of the index structure instance.

4.1.4 Program Variables

Starlog rules use variables extensively to generalise their use. These variables also exist in SDSL programs.

In SDSL, program variables keep the variable names from the Starlog source program but append a rule identifier index. The original variable names can be useful for debugging programs. Rule identifiers are necessary to avoid name clashes if the scope of variables changes during optimisation (discussed later in this chapter). This is because variable names in Starlog programs have only a local context (i.e. they are only visible within a single rule). Optimisation of SDSL programs may change the scope of a program variable so that it is accessed by multiple rules.

To ensure that type information inferred from a Starlog program can be used in later phases of the compilation process, the types of program variables are given when a variable is first used in an SDSL program. Types are included after

```
SDSL                    Prolog
A { B } C               A, ((B, fail); true), C.
```

Figure 4.3: Semantics of SDSL code blocks in Prolog.

the variable name and separated by a colon (:). For example, the first use of an integer `X1` will be given as `X1:int`. Like index variables, the values assigned to program variables do not change except when the program backtracks over the first use of the variable.

4.1.5 Code Blocks

Code Blocks are used in SDSL to enforce exhaustive backtracking through non-deterministic operations. When a Starlog rule can succeed using different variable bindings, all valid bindings are made to ensure the program is complete with respect to the well-founded model. Code blocks are surrounded by the braces { and }. For an example of code blocks used in real SDSL programs see Figure 5.4.

The semantics of a code block is comparable to failure-driven-loops in Prolog programs [104, 82] as shown in Figure 4.3. In this Figure, variables `A`, `B` and `C` represent any number of instructions. An implicit fail statement exists before the closing brace of a code block to ensure backtracking occurs. When all operations in a code block have been performed exhaustively the code block is said to *succeed* and any operations following the closing brace are performed.

A sequence of code blocks is a useful way of distinguishing different sets of independent operations - such as independent rules. Each code block in the sequence is evaluated independently until all non-deterministic alternatives have been considered. At this point the next code block in the sequence is processed. Code blocks are independent of each other because variables created in them are local to the code block and can not be accessed by any external operations, and because backtracking is performed locally in each code block.

When backtracking through a series of SDSL instructions, any code block encountered is considered to be deterministic (containing no choice points) since any possible backtracking within the code block has already occurred. For this reason, backtracking skips backward over code blocks and continues searching for the last choice point.

Code blocks may also be nested. Nested code blocks are useful for optimising programs (as will be seen later in this chapter). Nested code blocks can access any variables created in any of their outer code blocks.

Figure 4.4 gives an example of how the presence and order of code blocks can be used to control the flow of an SDSL program. In Figure 4.4 (2) the order that the labels `<a>` to `<d>` are processed assumes that each label has exactly two non-deterministic alternatives (or two variable binding sets) that are explored using backtracking. Each label is annotated with either `(1)` or `(2)` to distinguish each binding set. Beginning at the start (or top) of the program, both binding sets for label `<a>` are found in the first code block. After entering the second code block the first binding for label `<b>` is found. The first nested code block containing `<c>` is entered where both binding sets are found before exiting this block. The second nested code block containing `<d>` is processed

```
     |{
     |  <a>
     |}
     |{
     |  <b>
(1)  |  {
     |    <c>
     |  }
     |  {
     |    <d>
     |  }
     |}


(2)  [ <a(1)>, <a(2)>, <b(1)>, <c(1)>, <c(2)>, <d(1)>, <d(2)>,
                       <b(2)>, <c(1)>, <c(2)>, <d(1)>, <d(2)> ]
```

Figure 4.4: Example code blocks and their processing sequence.

SDSL	Prolog
`A not{ B } C`	`A, \+ (B), C.`

Figure 4.5: Semantics of SDSL negated code blocks in Prolog.

where again all binding sets are found. When the end of the second top-level code block is encountered backtracking is enforced. The last unexplored choice point remaining is the second variable binding of `<b>`. After making this binding the following code blocks containing `<c>` and `<d>` are processed as they were in the first pass.

Any variables created in `<a>` are local to this block and can not be accessed in any other code blocks. But any variables created and bound in `<b>` can be accessed by nested code blocks `<c>` and `<d>`.

4.1.6 Negated Code Blocks

For negated goals to be evaluated correctly in an SDSL program a different form of code block is required. *Negated Code Blocks* (or "not-blocks") are identified in an SDSL program by the annotated set of braces `not{` and }.

The semantics of a negated code block are that the block fails if there exists any variable bindings where all the internal operations succeed, and succeeds otherwise. Until a negated block is proven false by some variable binding, backtracking occurs within the block to consider all non-deterministic alternatives. The fact that only one contradicting variable binding is necessary for the negated block to fail means the exhaustive searching is not always required. Negated code blocks have the same semantics as the negation-as-failure operator in Prolog [77] as shown in Figure 4.5.

Like regular code blocks, any variables created within a negated code block can not be accessed by external operations. This enforces the restricted scope of existential variables in negated goals.

4.2 SDSL Instructions

In SDSL there are 15 instruction. A summary of the SDSL instructions appears in Table 4.1 where each instruction's name, syntax and a brief description of its function are given. The remainder of this section gives a detailed description of each instruction including their purpose, syntax, valid determinism, input and output variables, and semantics.

To aid understanding, the semantics of the first 14 SDSL instructions are described using operations performed on a Prolog database system. (The 15th instruction is used to perform builtin operations on variables which do not require such a database.) A Prolog database is chosen to describe semantics because its facilities (such as backtracking and untyped variables) are already well documented allowing a simplified discussion. To be consistent with the Starlog index structure, the Prolog database holds each argument element in a `data(Parent,Value,Subindex)` dynamic predicate where `Parent` is the unique identifier of the parent index node, `Value` is a value held in the dynamic index and `Subindex` is the unique identifier of the index node associated with `Value`. To model statically labelled branches and boolean values in a Prolog database we use two different dynamic predicates. A `branch(Parent,Label,Subindex)` defines a static branch (labelled `Label`) that points to a sub-index whereas `boolean(Parent,Label,Value)` holds the boolean `Value` identified by `Label`. In this data model each sub-index is functional given its parent and value or static label, and boolean values are functional given their parent index and static labels.

For the purposes of describing semantics, the asserts and retracts used to update the database hold to the "immediate update" view where predicates may be updated while in use, and the changes are immediately noticed by all running copies of the predicate [82]. This specification is necessary because Triggering Evaluation frequently adds tuples to the Γ and Δ sets and removes elements from the Δ while non-deterministic searches are in progress. The alternative "logical" view requires a more complex index structure to record when values are inserted and deleted. When used for strongly stratified programs, Triggering Evaluation inserts and deletes tuples which are irrelevant to existing searches (i.e. all new tuples produced are stratified later than their contributing goals, and all values deleted from Δ have been previously found during searches), therefore the "immediate update" view is sufficient.

To generate new, globally unique identifiers for index nodes the following clauses are used. Note that it is an error for `unique/1` to be called with a ground argument.

```
:- mode unique(-).
unique(ID) :- retract(current_num(Old)),
              ID is Old+1, assert(current_num(ID)), !.
unique(0) :- assert(current_num(0)).
```

With the exception of the builtin instruction, SDSL instructions perform actions on an index structure instance. To illustrate these actions we introduce

Name	Syntax	Description
Look	`look:<DET>(<PARENT INDEX>, <VALUE>, <SUBINDEX>)`	Searches for known value specified by `<VALUE>` in `<PARENT INDEX>`
Scan	`scan:<DET>(<PARENT INDEX>, <VALUE>, <SUBINDEX>)`	Binds variable `<VALUE>` with values in in `<PARENT INDEX>`
Insert	`insert:<DET>(<PARENT INDEX>, <VALUE>, <SUBINDEX>)`	Inserts the value specified by `<VALUE>` into `<PARENT INDEX>`.
Minimum	`minimum:<DET>(<PARENT INDEX>, <VALUE>, <SUBINDEX>)`	Binds `<VALUE>` to the minimum value from `<PARENT INDEX>`.
Next-Minimum	`nextminimum:<DET>(<PARENT INDEX>, <VALUE>, <SUBINDEX>)`	Binds `<VALUE>` to the next smallest unseen value in `<PARENT INDEX>`.
Delete	`delete:<DET>(<PARENT INDEX>, <VALUE>)`	Removes `<VALUE>` from `<PARENT INDEX>`.
Empty	`empty:<DET>(<SUBJECT INDEX>)`	Tests if an index `<SUBJECT INDEX>` is empty.
Follow	`follow:<DET>(<PARENT INDEX>, <LABEL>, <SUBINDEX>)`	Follows a statically labelled branch `<LABEL>` in the `<PARENT INDEX>`.
Establish	`establish:<DET>(<PARENT INDEX>, <LABEL>, <SUBINDEX>)`	Creates a new sub-index associated with `<LABEL>` in the `<PARENT INDEX>`.
Prune	`prune:<DET>(<PARENT INDEX>, <LABEL>)`	Removes the sub-index associated with `<LABEL>` in the `<PARENT INDEX>`.
Is-Pruned	`ispruned:<DET>(<PARENT INDEX>, <LABEL>)`	Tests if a sub-index of `<LABEL>` in the `<PARENT INDEX>` does not exists.
Link	`link:<DET>(<PARENT INDEX>, <LABEL>, <SUBINDEX>)`	Joins the labelled branch `<LABEL>` to an existing `<SUBINDEX>`.
Test	`test:<DET>(<PARENT INDEX>, <LABEL>, <BOOLEAN VALUE>)`	Tests the truth value of a labelled boolean value `<LABEL>` against `<BOOLEAN VALUE>`.
Set	`set:<DET>(<PARENT INDEX>, <LABEL>, <BOOLEAN VALUE>)`	Sets the truth value of a labelled boolean value `<LABEL>` to `<BOOLEAN VALUE>`.
Built-in	`builtin:<DET>(<BUILTIN NAME>, <INPUT VARS>, <OUTPUT VARS>)`	Performs built-in operation specified by `<BUILTIN NAME>`.

Table 4.1: SDSL instruction summary table.

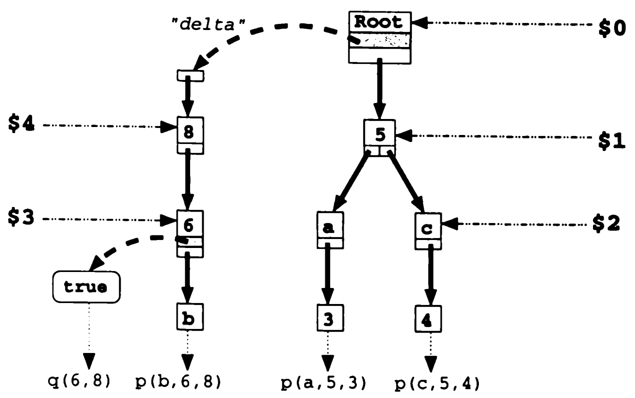


Figure 4.6: Example index structure instance for demonstrating SDSL instructions.

an example index structure instance in Figure 4.6. Index nodes that are used in the examples have been assigned to index variables **$0**, **$1**, **$2**, **$3** and **$4**.

In this section we categorise SDSL instructions into four groups: (1) those instructions that perform operations on the dynamic argument indexes in an index node, (2) those which operate on the statically labelled branches, (3) those which operate on the statically labelled boolean values and (4) those which do not perform operations on index nodes (the built-ins).

4.2.1 Dynamic Argument Index Instructions

Look

Purpose: A *look* instruction is used to find the sub-index associated with a known value in a dynamic argument index.

Syntax: `look:<DET>(<PARENT INDEX>, <VALUE>, <SUB-INDEX>)`

Determinism: Look instructions have two possible modes of determinism for `<DET>`. They can be `det` (deterministic) when the look operation is known to succeed, or `semidet` (semi-deterministic) if the look operation can fail to find the known value.

Input and Output Parameters: The `<PARENT INDEX>` parameter is a previously defined index variable pointing to the index node to be searched. The `<VALUE>` parameter is the known value to be searched for. This value can be in the form of a constant or a ground program variable. The final parameter, `<SUB-INDEX>`, is the output of the operation. This must be the name of a new (unbound) index variable that, when the operation succeeds, will be bound to the sub-index associated with the `<VALUE>`.

Semantics: The semantics of a look instruction can be compared to a known record search in a Prolog database where the parent index and value are always ground. The look instruction returns only one sub-index as each data record is functional given a ground parent and value. It is an error to call `look/3` with unbound `Parent` or `Value` arguments, or with a ground `Subindex` argument.

```
:- mode look(+,+,-).
look(Parent,Value,Subindex) :- data(Parent,Value,Subindex).
```

Examples: Using the Starlog index from Figure 4.6 we demonstrate a look instruction that is performed on the node pointed to by `$1`.

```
look:semidet($1,c,$4)
```

This instruction searches the values in the dynamic index `$1`. In this case the sought value, `c`, exists. The operation succeeds with the output index variable `$4` bound to the sub-index containing `c`. Here is another example of a look instruction.

```
look:semidet($1,b,$5)
```

This operation fails because no `b` value exists in the index pointed to by `$1`. Failure of an operation initiates backtracking through the preceding instructions.

Scan

Purpose: A *scan* instruction is used to find possible bindings for a free program variable (and the associated sub-indexes) in a given index.

Syntax: `scan:<DET>(<PARENT INDEX>, <VALUE>, <SUB-INDEX>)`

Determinisms: There are three possible determinisms for scan instructions. A scan that is looking for exactly one binding and that can not fail is `det`. A scan looking for exactly one binding but may fail is `semidet`. Scans that are `nondet` (non-deterministic) find multiple bindings for a variable using backtracking to find each case. (Usually scans need to be declared `nondet`.)

Input and Output Parameters: Like the look instruction, the `<PARENT INDEX>` parameter is a previously defined index variable pointing to the index node to be searched. However, unlike the look instruction, the `<VALUE>` parameter is a new output program variable that is bound when the operation succeeds. (New program variables created here must also specify their type.) The `<SUB-INDEX>` output parameter is a new index variable that points to the sub-index associated with the binding of `<VALUE>`.

The order that values are retrieved from an index during a non-deterministic scan is unspecified as this will depend on the data structure implementation. The only condition is that all values stored in the argument index for the duration of the scan must be returned exactly once.

Semantics: A scan operation is equivalent to searching the Prolog database for an unknown value given a known parent index. It is an error for `scan/3` to be called with ground `Value` or `Subindex` arguments.

```
:- mode scan(+,-,-).
scan(Parent,Value,Subindex) :- data(Parent,Value,Subindex).
```

Examples: When a non-deterministic scan instruction is performed on the index node referenced by `$1` in Figure 4.6, all values and sub-indexes contained in the argument index are found. Such an instruction would appear as follows:

```
scan:nondet($1,X:string,$6)
```

We assume that this instruction first binds the new variable `X` to `a` and `$6` to the sub-index containing `a`. The instructions following this scan are performed until backtracking occurs. When this non-deterministic scan instruction is encountered during backtracking, the program variable `X` is bound to `c` and index variable `$6` now points to the sub-index containing `c`. The instructions following the scan are repeated using the new variable bindings. This time when failure occurs and this scan instruction is backtracked through, the scan operation fails because there are no unused bindings for `X`, and backtracking continues.

Insert

Purpose: *Insert* instructions are used to add values to a dynamic argument index. When a new value is inserted then a new sub-index associated with this value is created. In this way the sub-index is initialised for the insert operations that follow. If a value is already present in the target index, the duplicate value to be inserted is ignored.

Syntax: `insert:<DET>(<PARENT INDEX>, <VALUE>, <SUB-INDEX>)`

Determinisms: The determinism of insert operations can only be `det` (deterministic). The `<DET>` parameter is included in the syntax definition for consistency.

Input and Output Parameters: Like look or scan instructions, inserts are performed on the argument index in the index node pointed to by the ground index variable `<PARENT INDEX>`. The `<VALUE>` parameter is the known value (constant or ground program variable) to be added. The output of this operation is the `<SUB-INDEX>` – a new index variable that points to the index node associated with `<VALUE>`. This may be a new sub-index created by this operation or (when attempting to a add duplicate value) an existing sub-index.

Semantics: The insert instruction is equivalent to the following clauses in the Prolog database. If a sub-index already exists then it is returned. Otherwise a sub-index is created. All insert operations must provide ground parent indexes (`Parent`) and ground insert values, while the output sub-index must be unbound before the operation is performed.

```
:- mode insert(+,+,-).
```

```
insert(Parent,Value,Subindex) :- data(Parent,Value,Subindex),!.
insert(Parent,Value,Subindex) :- unique(Subindex),
                                 assert(data(Parent,Value,Subindex)).
```

Note that the use of assert/1 under the "immediate update" view makes no assumptions that a newly inserted value will be returned by any scans that are still in progress (that is, non-deterministic scans which have not exhausted all the values in an index). Triggered, strongly stratified programs can be structured in such a way that the exclusion or inclusion of new values in the set of results does not affect its correctness since any new values added will represent tuples that are irrelevant to the existing scans (see Chapter 5).

Examples: To insert a new value (and create a new sub-index) in the root node in Figure 4.6 we use the following instruction.

```
insert:det($0,7,$7)
```

The new entry in the root index node ($0) is indexed by the value 7 and contains a new (empty) sub-index. Although not shown in the Prolog semantics for clarity, the new index node is based on a template that all child indexes of $0 use. This template describes the implementation of the sub-index including any labelled branches or boolean values that exist. The output index variable $7 is bound to the new sub-index created during this operation. Now consider the instruction:

```
insert:det($0,5,$8)
```

This instruction attempts to add a duplicate value to the root index node ($0). Because the value 5 already exists in the dynamic argument index we do not create a new value or sub-index. Instead, the operation succeeds by binding the output index variable $8 to the existing sub-index containing 5. In this way arguments with identical values are combined together to maximise sharing and each value stored in an argument index is unique.

Minimum

Purpose: The *minimum* instruction is required by Stratified Semi-Naive Evaluation (and consequently is required by Triggering Evaluation) to ensure tuples in Δ are processed in stratification order. This instruction selects and outputs the minimum element that exists in an index node's argument index. A predefined order over the values held in an argument index is used to find a minimal element.

Syntax: `minimum:<DET>(<PARENT INDEX>, <VALUE>, <SUB-INDEX>)`

Determinisms: When used in triggered programs, minimum instructions are nondet (non-deterministic) where each time they are encountered during backtracking they rescan the subject argument index and recompute the current minimum value (since previous minimum values are usually deleted during evaluation). A minimum operation fails when the argument index it is searching becomes empty. Both det (deterministic) and semidet (semi-deterministic) versions of the minimum instruction are also possible where one, or at most one, minimum element is output respectively, however these have not yet been used in any SDSL programs.

Input and Output Parameters: The `<PARENT INDEX>` input parameter is the index variable pointing to the index node to be searched. The `<VALUE>` output parameter is a new program variable that is bound to the minimum value in the index after the operation succeeds. The `<SUB-INDEX>` output parameter is a new index variable that points to the index node associated with the minimum value in the `<PARENT INDEX>`.

Semantics: The Prolog database would perform a minimum operation of this kind using the following clause.

```
:- mode minimum(+,-,-).
minimum(Parent,Value,Subindex) :- data(Parent,Value,Subindex),
                        \+ (data(Parent,Value2,_), Value2 < Value).
```

Note that this is a very inefficient implementation ($O(N^2)$ where N is the number of elements in the argument index) but is logically correct. The implementations of this operation used in compiled programs can be much more efficient (see Appendix B).

Examples: The following instruction finds the minimum value (and sub-index) in the index node pointed to by `$1` in Figure 4.6.

```
minimum:notdet($1,Y:string,$9)
```

Assuming the strings indexed at `$1` are ordered lexicographically, this operation will bind program variable `Y` to value `a` and the index variable `$9` to the sub-index associated with value `a`. If index node `$1` remains unchanged then the minimum value will remain `a` each time this instruction is encountered during backtracking. This example will be continued in conjunction with an example of the delete operation in the next subsection.

Next-Minimum

Purpose: The *next-minimum* instruction searches a dynamic argument index and returns values one at a time in ascending order. Unlike the minimum instruction, a next-minimum operation maintain a reference to the last element found and will return the next smallest value from an argument index. In this way a next-minimum operation is equivalent to an ordered scan. This instruction is necessary when the index structure used by an SDSL program shares argument indexes between both the Δ and Γ sets. (See Chapter 5 for details of its application.)

Syntax: `nextminimum:<DET>(<PARENT INDEX>, <VALUE>, <SUB-INDEX>)`

Determinisms: Next-minimum instructions are always `nondet` (non-deterministic) to ensure all values are found. A next-minimum operation fails after it has returned all values from an argument index.

Input and Output Parameters: The <PARENT INDEX> input parameter is the index variable pointing to the index node to be searched and the <VALUE> output parameter is a new program variable that is bound to the *next* minimum value in the argument index when the operation succeeds. The <SUB-INDEX> output parameter is a new index variable that points to the index node associated with the minimum value in the <PARENT INDEX>.

Semantics: The semantics of the next-minimum instruction are equivalent to the following Prolog clauses. Note that, unlike scan instructions, when next-minimum instructions are used to optimise programs (see Chapter 5) it is impossible for multiple instances of these instructions to be working on the same index node at the same time. This simplifies the implementation of this instruction in the underlying data structures and simplifies the semantic definition given here. Like the minimum instruction, the Prolog clauses given to describe the semantics of next-minimum instructions are correct but inefficient.

```
:- mode nextminimum(+,-,-).
nextminimum(Parent,Value,Subindex) :-
            retract(next_min(Parent,OldValue)), !,
            data(Parent,Value,Subindex), Value > OldValue,
            \+ (data(Parent,Value2,_),
                 Value2 > OldValue, Value2 < Value),
            assert(next_min(Parent,Value)).
nextminimum(Parent,Value,Subindex) :- data(Parent,Value,Subindex),
            \+ (data(Parent,Value2,Subindex), Value2 < Value),
            assert(next_min(Parent,Value)).
```

Examples: When the following instruction is applied to the index structure instance in Figure 4.6 all values indexed at $1 will be found.

```
nextminimum:notdet($1,Y:string,$9)
```

When strings in $1 are ordered lexicographically Y will be bound to value a in the first iteration. The index variable $9 is bound to the sub-index associated with value a. When this instruction is encountered during backtracking then (assuming index $1 has not been not updated) Y is bound to c – the next lowest element in the index – and $9 is updated to the sub-index associated with c. When encountered during backtracking a second time this operation fails as all values have been explored.

Delete

Purpose: To remove values (and their associated sub-indexes) from a dynamic argument index the *delete* instruction is used. Usually deletions are performed on "leaf" sub-index nodes that represents a tuple, or when the sub-index is redundant (see Section 3.3.3). (To test whether an index is redundant before deleting it requires the *empty*, *is-null* and *test* instructions introduced later in this section.)

Syntax: `delete:<DET>(<PARENT INDEX>, <VALUE>)`

Determinisms: The determinism of all delete instructions is **det** (deterministic).

Input and Output Parameters: The **<PARENT INDEX>** parameter is a previously defined index variable referencing an index node. The parameter **<VALUE>** is the known value to be removed from **<PARENT INDEX>**. **<VALUE>** is either a constant or a ground program variable.

Semantics: The semantics of the delete operation when performed on the Prolog database are defined by the following clause. Note that this implementation explicitly deletes any remaining sub-indexes, labelled branches and labelled boolean values held beneath the deleted node. Such resource management could be performed by an automatic garbage collector.

```
:- mode delete(+,+).
delete(Parent,Value) :- retract(data(Parent,Value,Subindex)),
                        ( data(Subindex,Value,_),
                          delete(Subindex,Value), fail
                        ;
                          true ),
                        ( branch(Subindex,Label,_),
                          retract(branch(Subindex,Label,_)), fail
                        ;
                          true ),
                        ( boolean(Subindex,Label,_),
                          retract(boolean(Subindex,Label,_)), fail
                        ;
                          true ).
```

Examples: The following delete instruction can be performed on the index in Figure 4.6.

```
delete:det($1,a)
```

Used by itself, this delete instruction will remove the sub-index associated with the value a from the index node referenced by $1 even when the sub-index contains many tuples. When used in conjunction with a minimum operation, deleting values from an index node can change the minimum value found in each iteration. Consider:

```
minimum:nondet($1,Y:string,$9)
delete:det($1,Y)
```

In Figure 4.6, when the argument values held in the index node $1 are ordered lexicographically, the first minimum value bound to Y is a. Using this binding of Y, the delete operation removes a from the argument index at $1 and removes any associated sub-indexes. When backtracking occurs over these instructions the non-deterministic minimum operation searches index node $1 again and finds now, with a removed, the minimum value in the argument index is c. The process of finding the next minimum value and deleting it from an index node can continue until the argument index is empty. This is the mechanism used to process tuples in the Δ set in stratification order.

Examples of how an index node is tested for redundancy are given as the testing instructions are introduced.

Empty

Purpose: The *empty* instruction succeeds only when the argument index of the subject index node contains no elements. This instruction helps determine whether an index node is redundant and should be deleted.

Syntax: `empty:<DET>(<SUBJECT INDEX>)`

Determinisms: The determinism of an empty instruction is **semidet** (semi-deterministic) where it succeeds if the argument index in question is empty, and fails otherwise.

Input and Output Parameters: The index node whose contents are being examined is specified by the `<SUBJECT INDEX>` index variable. This variable must have been instantiated previously.

Semantics: The empty SDSL instruction is equivalent to the following clause in the Prolog database.

```
:- mode empty(+).
empty(Index) :- \+ data(Index,_,_).
```

Examples: The following instruction can be performed on the index structure instance given in Figure 4.6.

```
empty:semidet($1)
```

This operation fails because the index node pointed to by **$1** is not empty. Empty instructions are used to test the redundancy of an index node before its deletion. For example:

```
scan:nondet($1,Y:string,$9)
empty:semidet($9)
delete:det($1,Y)
```

This set of instructions processes all sub-indexes of **$1** that are found during a scan operation and tests if each sub-index is empty. If the sub-index is empty the delete instruction will remove it from its parent index.

4.2.2 Labelled Branch Instructions

Follow

Purpose: A *follow* instruction is used when a static labelled branch should be followed to a sub-index. This instruction is used when searching for tuples.

Syntax: `follow:<DET>(<PARENT INDEX>, <LABEL>, <SUB-INDEX>)`

Determinisms: Follow instructions that are `semidet` (semi-deterministic) fail when a sub-index associated with a label does not exist (i.e. before the labelled branch has been initialised using the *establish* instruction, described later). Alternatively when the sub-index of a labelled branch is known to exist a `det` (deterministic) version is available.

Input and Output Parameters: The `<PARENT INDEX>` is a previously defined (ground) index variable whose index node must contain the labelled branch identified by `<LABEL>`. Labels are constant values. The naming convention of labels adhere Java's variable naming conventions for implementation reasons (discussed in Chapter 6). The `<SUB-INDEX>` output variable is a new index variable that points to the index node associated with the `<LABEL>` branch.

Semantics: Following a labelled branch in the Prolog database can be achieved using the following clause.

```
:- mode follow(+,+,-).
follow(Parent,Label,Subindex) :- branch(Parent,Label,Subindex).
```

Examples: In Figure 4.6, the following instruction is used to follow the "`delta`" labelled branch in the index node referenced by `$0`.

```
follow:semidet($0,delta,$10)
```

The output index variable `$10` is bound to the sub-index at the end of the "`delta`" branch. If no sub-index existed at the end of the "`delta`" branch then this operation would fail.

Establish

Purpose: An *establish* instruction creates and returns a new sub-index for a labelled branch if one does not already exist. Given a labelled branch, before any follow instruction can succeed on this branch an establish operation must have been performed. If a sub-index does exist then the establish operation simply follows the labelled branch and returns the associated sub-index (equivalent to a follow instruction). This instruction is used when adding a new tuple to an index structure instance where the tuple's index path definition contains a labelled branch.

Syntax: `establish:<DET>(<PARENT INDEX>, <LABEL>, <SUB-INDEX>)`

Determinisms: The determinism of an establish instruction is always `det` (deterministic).

Input and Output Parameters: Like the follow instruction, the `<PARENT INDEX>` is a previously defined (ground) index variable referencing an index node that contains a labelled branch identified by `<LABEL>`. The label is a constant value. The `<SUB-INDEX>` is a new index variable that points to the index node associated with the `<LABEL>` branch.

Semantics: Establishing a labelled branch in the Prolog database uses the following clauses. The second clause completes the initialisation of a labelled branch if a sub-index does not exist.

```
:- mode establish(+,+,-).
establish(Parent,Label,Subindex) :-
                                branch(Parent,Label,Subindex),!.
establish(Parent,Label,Subindex) :- unique(Subindex),
                            assert(branch(Parent,Label,Subindex)).
```

Examples: In Figure 4.6, to establish the "`delta`" labelled branch in the index pointed to by `$0` the following instruction is used:

```
establish:det($0,delta,$11)
```

Because the sub-index associated with "`delta`" already exist in Figure 4.6, index variable `$11` is bound to this existing sub-index. If no sub-index were beneath the "`delta`" labelled branch this instruction would generate a new sub-index according to a sub-index template.

Prune

Purpose: To remove an index node from the end of a labelled branch we use a *prune* instruction. After this operation has been performed the labelled branch will still exist, but it will not point to an index node. Prune instructions are used to remove redundant index nodes that are pointed to by labelled branches.

Syntax: `prune:<DET>(<PARENT INDEX>, <LABEL>)`

Determinisms: The determinism of a prune instruction is always `det` (deterministic).

Input and Output Parameters: The `<PARENT INDEX>` must be a previously defined (ground) index variable referencing an index node with a `<LABEL>` branch. The `<LABEL>` is a constant.

Semantics: In the Prolog database a prune instruction would be equivalent to the following clause.

```
:- mode prune(+,+).
prune(Parent,Label) :- (retract(branch(Parent,Label,_)) ; true),!.
```

Examples: In Figure 4.6 the "`delta`" branch can be removed using the following instruction.

```
prune:det($0,delta)
```

After this instruction is performed the "`delta`" branch will no longer point to an index node and any subsequent follow instructions will fail. (To create another index node the establish instruction must be used.)

A sequence of SDSL instructions which will remove the "`delta`" branch only when it becomes redundant is given below.

```
follow:semidet($0,delta,$9)
empty:semidet($9)
prune:det($0,delta)
```

This sequence of SDSL instructions will not change the index structure instance in Figure 4.6 since the index pointed to by the "`delta`" labelled branch is not redundant.

Is-Pruned

Purpose: During deletion of nodes it may be necessary to determine when an index pointed to by a labelled branch is redundant. The *is-pruned* instruction tests a labelled branch to see if a sub-index exists. Although this is the equivalent to a negated follow instruction, a new instruction is used to optimise both SDSL program size and its implementation.

Syntax: `ispruned:<DET>(<PARENT INDEX>, <LABEL>)`

Determinisms: Because this operation performs a test, the determinism of this instruction is always `semidet` (semi-deterministic).

Input and Output Parameters: `<PARENT INDEX>` represents a previously defined (ground) index variable referencing an index node containing a `<LABEL>` branch. `<LABEL>` is a constant.

Semantics: The is-pruned instruction is equivalent to the following clause in the Prolog database.

```
:- mode ispruned(+,+).
ispruned(Parent,Label) :- \+ branch(Parent,Label,_).
```

Examples: An is-pruned instruction performed on the example index in Figure 4.6 is given.

```
ispruned:semidet($0,delta)
```

This instruction would fail since the "`delta`" branch on the root node has a valid sub-index.

Link

Purpose: A *link* instruction is used to allow multiple indexing paths within an index structure to merge. This instruction takes two existing index variables and assigns a labelled branch as a one directional link from one to the other.

Syntax: `link:<DET>(<PARENT INDEX>, <LABEL>, <SUB-INDEX>)`

Determinisms: Link instructions are always `det` (deterministic).

Input and Output Parameters: The `<PARENT INDEX>` and `<SUB-INDEX>` are two previously generated (ground) index variables. The `<LABEL>` is a constant. After the operation is complete, following the `<LABEL>` branch in the `<PARENT INDEX>` will lead to the `<SUB-INDEX>`.

Semantics: In the Prolog database, a link instruction can be implemented as follows.

```
:- mode link(+,+,+).
link(Parent,Label,Subindex) :-
                                branch(Parent,Label,Subindex),!.
link(Parent,Label,Subindex) :-
                                assert(branch(Parent,Label,Subindex)).
```

Examples: The example index structure instance in Figure 4.6 does not contain two index nodes that require linking. However the following example instruction would still be valid.

```
link:semidet($0,delta,$3)
```

The wisdom in performing such an operation is questionable since the index structure instance will be mutilated, but the outcome would be that the "`delta`" labelled branch would now point to the subject of index variable`$3`.

4.2.3 Labelled Boolean Value Instructions

Test

Purpose: *Test* instructions are used to determine the truth value of labelled boolean values. This operation succeeds when the labelled boolean value is equal to the sought truth value and fails otherwise. This operation is used to detect the presence of a tuple represented by a labelled boolean value during tuple searches, and can help determine whether an index node is redundant before deleting it.

Syntax: `test:<DET>(<PARENT INDEX>, <LABEL>, <BOOLEAN VALUE>)`

Determinisms: The determinism of test operations is always `semidet` (semideterministic) since tests that can not fail and tests that have multiple solutions are of no use.

Input and Output Parameters: The `<PARENT INDEX>` parameter is a previously defined (ground) index variable that references an index node containing a labelled boolean flag identified by the `<LABEL>` constant. The `<BOOLEAN VALUE>` is the constant value `true` or `false` depending on how it is being used.

Semantics: In Prolog a test instruction would be performed as follows. The second clause ensures that all uninitialised boolean values hold `false`.

```
:- mode test(+,+,+).
test(Parent,Label,Value) :- boolean(Parent,Label,V2),!,
                            Value = V2.
test(Parent,Label,false).
```

Examples: In Figure 4.6, to test the truth value of the labelled boolean value "id0" in the index node referenced by $3 requires the following instruction:

```
test:semidet($3,id0,true)
```

In this case the boolean value labelled "id0" is true, therefore this operation succeeds.

Test instructions are used to determine if an index node containing labelled boolean values is redundant and can be safely deleted. The following sequence of instructions performs this operation on the index in Figure 4.6.

```
scan:nondet($4,Y:int,$9)
empty:semidet($9)
test:semidet($9,id0,false)
delete:det($4,Y)
```

This sequence of instructions will not remove any index nodes from the index in Figure 4.6 primarily because the only sub-index of $4 is not empty. However even if the dynamic argument index were empty, the test instruction would fail since the labelled boolean value is true.

Set

Purpose: The *set* instruction is used to modify the contents of labelled boolean values. By setting a labelled boolean value to `true` the presence of a tuple can be added to an index structure instance. Setting a labelled boolean value to `false` removes the tuple it represents from the index structure instance.

Syntax: `set:<DET>(<PARENT INDEX>, <LABEL>, <BOOLEAN VALUE>)`

Determinisms: All set operations are `det` (deterministic).

Input and Output Parameters: The `<PARENT INDEX>` input parameter is a ground index variable pointing to an index node. This index node contains a labelled boolean value identified by the `<LABEL>` constant. The `<BOOLEAN VALUE>` is the value `true` or `false` depending on the intended use of this operation.

Semantics: To update a labelled boolean value in the Prolog database would require the following clause. By insisting that only one value exist is in the database by retracting any previous values, the first `boolean/3` relation encountered is always the most recent assignment.

```
:- mode set(+,+,+).
set(Parent,Label,Value) :- retract(boolean(Parent,Label,_)), !,
                           assert(boolean(Parent,Label,Value)).
set(Parent,Label,Value) :- assert(boolean(Parent,Label,Value)).
```

Examples: To change the truth value of the labelled boolean value "id0" in Figure 4.6 we use:

```
set:det($3,id0,false)
```

This operation assigns the value of the "id0" labelled boolean value in the index pointed to by `$3` to `false`. Future test instructions which search for a `true` value in this labelled boolean value would now fail as a result of this set operation.

4.2.4 Built-ins

Purpose: The purpose of this instruction is to execute *built-in* operations. More precisely, built-in operations allow programs to test the values of program variables, declare and assign values to new variables or generate side effects within SDSL programs. (See Figure 4.2 for the complete range of available operations.)

Syntax:

```
builtin:<DET>(<BUILTIN NAME>, <INPUT VARIABLES>, <OUTPUT VARIABLES>)
```

Determinisms: The particular built-in operation specified by the `<BUILTIN NAME>` parameter dictates whether the operation can be `det` (deterministic), `semidet` (semi-deterministic) or `nondet` (non-deterministic).

Input and Output Parameters: The `<BUILTIN NAME>` is the description of a built-in (a constant) taken from a library. The input parameters to be used by the built-in operation are given in `<INPUT VARIABLE LIST>`. This is a comma-separated list enclosed in symbols "`[`" and "`]`", of previously defined (ground) program variables or constant values. The `<OUTPUT VARIABLE LIST>` is a list of new program variables that will be bound during the built-in operation. As with all occurrences of new program variables, types are included.

Semantics: The names of built-ins are usually overloaded for different input variable types, for different determinisms and for different modes. Figure 4.2 gives a list of all built-in operations currently supported with their types, modes, determinisms and Prolog semantics. Some of the uses of determinism for built-in operations requires additional explaination. Semi-deterministic arithmetric operations (i.e. add, divide, multiply, sqrt, and subtract) are used to test that the inputs form a valid equation. For example, `builtin:semidet(add,[A,B,C],[])` is true iff the equation `C = A+B` is true. The non-deterministic version of the square-root (sqrt) operation returns both the positive and negative square roots the input (when they exist).

Examples: Some examples of using SDSL built-ins are given here. The semantic definitions of each built-in given in Figure 4.2 should be used to interpret each example.

```
builtin:semidet(greaterThan, [X,5], [])
builtin:det(add, [X,Y], [Z:int])
builtin:semidet(add, [X,Y,42], [])
builtin:nondet(sqrt, [X], [W:int])
builtin:det(print, [''hello world''], [])
```

Instruction	Prolog[1] Semantics	Comment
builtin:det(add,[(int)A,(int)B],[(int)C])	C is A+B	
builtin:det(add,[(double)A,(double)B],[(double)C])	C is A+B	
builtin:det(add,[(string)A,(string)B],[(string)C])	concat(A,B,C)	String concatenation
builtin:semidet(add,[(int)A,(int)B,(int)C],[])	C is A+B	
builtin:semidet(add,[(double)A,(double)B,(double)C],[])	C is A+B	
builtin:semidet(add,[(string)A,(string)B,(string)C],[])	C is A+B	
builtin:det(assign,[(int)A],[(int)B])	A = B	
builtin:det(assign,[(double)A],[(double)B])	A = B	
builtin:det(assign,[(string)A],[(string)B])	A = B	
builtin:det(divide,[(int)A,(int)B],[(int)C])	C is A//B	Integer division
builtin:det(divide,[(double)A,(double)B],[(double)C])	C is A/B	
builtin:semidet(divide,[(int)A,(int)B,(int)C],[])	C is A//B	Integer division
builtin:semidet(divide,[(double)A,(double)B,(double)C],[])	C is A/B	
builtin:det(equals,[(int)A],[(int)B])	A = B	
builtin:det(equals,[(double)A],[(double)B])	A = B	
builtin:det(equals,[(string)A],[(string)B])	A = B	
builtin:semidet(equals,[(int)A,(int)B],[])	A =:= B	
builtin:semidet(equals,[(double)A,(double)B],[])	A =:= B	
builtin:semidet(equals,[(string)A,(string)B],[])	A = B	
builtin:semidet(fail,[],[])	fail	Unconditional failure
builtin:semidet(greaterThan,[(int)A,(int)B],[])	A > B	
builtin:semidet(greaterThan,[(double)A,(double)B],[])	A > B	
builtin:semidet(greaterThanOrEqual,[(int)A,(int)B],[])	A >= B	
builtin:semidet(greaterThanOrEqual,[(double)A,(double)B],[])	A >= B	
builtin:det(input,[],[(string)C])	read(C)	Inputs string from standard input
builtin:det(multiply,[(int)A,(int)B],[(int)C])	C is A*B	
builtin:det(multiply,[(long)A,(long)B],[(long)C])	C is A*B	
builtin:det(multiply,[(double)A,(double)B],[(double)C])	C is A*B	
builtin:semidet(multiply,[(int)A,(int)B,(int)C],[])	C is A*B	
builtin:semidet(multiply,[(double)A,(double)B,(double)C],[])	C is A*B	
builtin:semidet(notEquals,[(int)A,(int)B],[])	A =\= B	
builtin:semidet(notEquals,[(double)A,(double)B],[])	A =\= B	
builtin:semidet(notEquals,[(string)A,(string)B],[])	A \== B	
builtin:det(print,[(string)A],[])	print(A)	Outputs string to standard output
builtin:det(sqrt,[(int)A],[(int)B])	B is sqrt(A), round(B,0)	Integer square root
builtin:det(sqrt,[(double)A],[(double)B])	B is sqrt(A)	
builtin:semidet(sqrt,[(int)A],[(int)B])	B is sqrt(A)	Fails when A is negative
builtin:semidet(sqrt,[(double)A],[(double)B])	B is sqrt(A)	Fails when A is negative
builtin:semidet(sqrt,[(int)A,(int)B]),[]	(B is sqrt(A) ; B is -sqrt(A))	
builtin:semidet(sqrt,[(double)A,(double)B],[])	(B is sqrt(A) ; B is -sqrt(A))	
builtin:nondet(sqrt,[(int)A],[(int)B])	(B is sqrt(A) ; B is -sqrt(A))	
builtin:nondet(sqrt,[(double)A],[(double)B])	(B is sqrt(A) ; B is -sqrt(A))	
builtin:det(subtract,[(int)A,(int)B],[(int)C])	C is A-B	
builtin:det(subtract,[(double)A,(double)B],[(double)C])	C is A-B	
builtin:semidet(subtract,[(int)A,(int)B,(int)C],[])	C is A-B	
builtin:semidet(subtract,[(double)A,(double)B,(double)C],[])	C is A-B	
builtin:semidet(subtract,[(string)A,(string)B,(string)C],[])	C is A-B	

[1] The semantics of these operators, functions and comparators are equivalent to those given in the definition of Arity/Prolog in [73].

Table 4.2: SDSL built-in operations.

4.3 Summary

With the features of SDSL described, the implementation of triggered programs which operate on a Starlog's index structures can be explained. The next chapter describes the layout of SDSL's code blocks, negated code blocks and instructions necessary to correctly evaluate a given program using Triggering Evaluation. In addition, the structure of SDSL programs can allow for many high-level optimisations which would be difficult to realise in other representations.

Chapter 5

SDSL Triggered Programs

In Chapter 4 the SDSL language was described with vague references to how each instruction can contribute to the evaluation of a bottom-up program. We now discuss specifics of how Starlog is compiled to SDSL.

This chapter gives an overview of the steps necessary to evaluate a bottom-up programs using Triggering Evaluation. A description of each of these steps is given with example SDSL code provided for the less trivial steps. A complete Starlog example program and the equivalent SDSL program are introduced in Figure 5.4 to demonstrate the integration of each step in a complete SDSL program. Finally, an extensive set of SDSL code optimisations is given which reduce both code size and run time overhead of Starlog programs. Each optimisation is described for the general and an example case, and when possible, comparisons are made between the SDSL optimisations and code optimisations used by other languages. Figure 5.1 indicates the stages in the Starlog compilation pipeline that are discussed in this chapter.

For clarity, SDSL triggered programs are first described for index structures that do not combine the arguments of the Δ and Γ sets. That is, the Δ set is distinguished from Γ by a labelled branch before any arguments are indexed. When Δ and Γ share argument indexes the resulting program may be more efficient. However the operations required on such index structures are more complicated. Details of the extra requirements are given later in this chapter.

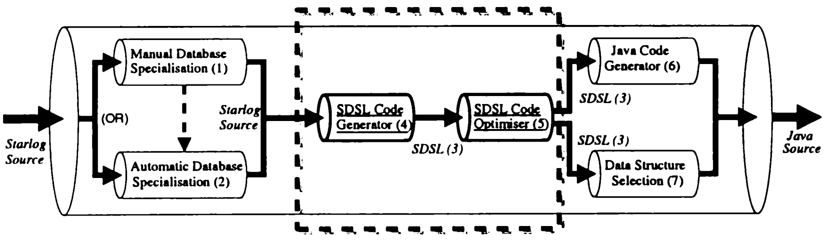


Figure 5.1: SDSL code generation and optimisation in Starlog's compilation pipeline.

5.1 Triggering Evaluation Overview

In Chapter 2 the Triggering Evaluation algorithm was introduced and, through a series of optimisations, it evolved to become the algorithm given in Figure 2.15. This chapter shows how this algorithm is implemented using SDSL. To introduce Triggering Evaluation using SDSL we will divide the process into a number of steps. Each of these steps is represented by a series of SDSL instructions and/or code blocks which appear in an SDSL program in roughly the order given here.

The first step to evaluating a triggered bottom-up program (1) adds facts (rules without positive goals) to the Δ set. Next, (2) the Δ set is repeatedly searched for a minimal stratified tuple h (which may be constrained by unresolved negated goals such that $h \leftarrow \beta^-$). Because of the static nature of the index structure, the predicate that tuple h belongs to is determined by the path followed to find it. Consequently the static code used to process each tuple depends on the path followed to find h. (3) Based on the predicate that the minimum tuple belongs to (where tuples with the same functor but constrained by different negated goals are considered from different predicates) the unresolved negated goals of the minimum tuple are queried in Γ. (4) If the minimum tuple h is not contradicted by a negated goal from β^- then it is added to Γ. h, the new unconditionally true tuple, is now used to activate rules. Because rules are activated only when the tuple h matches the trigger goal, only rules where the trigger is from the same predicate as h need to be considered. (The subset of rules activated by tuples from each predicate is statically defined.) (5) For each rule, tuple h is matched with the trigger tuple. If matching is successful the other goals in the body are queried one at a time. If the conjunction of all goals is true, the head of the rule is inserted into Δ (constrained by unresolved negated goals where necessary). (6) When all relevant rules have been activated by tuple h, h is deleted from Δ so that a new minimum tuple will be found in the next iteration. The details of each of these steps is given over the next few sections.

It should be noted that although SDSL does allow different versions of some instructions with different determinisms, the determinisms of instructions used for Triggering Evaluation are fixed for each instruction. All *look* instructions are semi-deterministic, *scans* are non-deterministic, *minimums* are non-deterministic and *follows* are semi-deterministic. The determinisms of *built-in* instructions is specific to the operation performed, however the most general determinism will be used when there is more than one alternative. For example, built-ins will be `nondet` if a non-determinstic version of the built-in exists, otherwise will be `semidet` if a semi-determinisitc version exist. The inclusion of the multiple determinisms for instructions in the previous chapter allows for future optimisations.

5.2 Adding Facts to Δ

Triggered programs begin by adding all facts to Δ that are true at the start of the SDSL program. The definition of a fact in Starlog is a rule with no positive body goals. To allow semi-deterministic and non-deterministic built-in operations to fail or give multiple variable bindings without affecting the rest of the program, each fact is evaluated in a separate code block. The built-in operations in the fact's body are represented by their equivalent SDSL built-in operations (see Chapter 4) in the code block. All negated goals constraining facts are delayed until the fact becomes a minimal element in Δ. The head of a fact is added to Δ using insert, establish or set (to true) SDSL instructions depending on the contents of the fact's path definition (see Section 3.3.2 for how tuples are inserted into an index structure instance).

To illustrate this process, the first code block of the SDSL program in Figure 5.4 shows how a fact is added to the Δ set for a given index structure.

5.3 Finding and Deleting Minimum Tuples

We have seen in Chapter 2 that a key step in the evaluation of stratified programs with negation is to extract tuples from the Δ set in stratifiaction order. That is, at each top level iteration of either Stratified Semi-naive or Triggering Evaluation a minimal stratified element in Δ is found (refered to as h). In Chapter 1 it was shown that the stratification order of tuples can be determined by their argument values, the tuple's functor, or any sequence of these. In this section we show how we can extract elements from Δ in stratification order when this set has been automatically defined in a Starlog index structure using techniques from Chapter 3.

In Chapter 3 the automatic definition of argument orders arranged the arguments and functors in the Δ set according to their stratification priority. That is, arguments or functors which are more significant to determining the order of tuples are indexed first. It was shown that this reduces the amount of searching required to locate a minimal tuple. The index structures derived automatically using techniques from Chapter 3 maintain these argument/functor orders.

To repeatedly find the minimum tuple in Δ we perform an in-order traversal of the Δ set in the index structure instance, deleting the tuple after it is found. For this operation to be performed repeatitively it is encapsulated in a code block which enforces backtracking and therefore exploration of all tuples in Δ. To perform such a traversal requires different SDSL instructions depending on the properties of the index nodes encountered.

Starting from the root index node of the Δ set, if an argument index is encountered which holds arguments that are involved in the stratification order for some tuples (i.e. if the argument indexed appears in an argument stratification priority) then a non-deterministic *minimum* instruction is used to locate the minimum value of this argument. If an argument index is encountered whose values do not affect the stratification order of tuples then a non-deterministic *scan* instruction is used to return argument values in an arbitrary order. The variables used to hold the binding of these arguments are automatically generated so that they are distinct from other program variables used in rules. By convention we label these variables as `MinVar#` where `#` is a globally unique integer value identifying the argument.

Labelled branches and labelled boolean values in Δ represent static information about each tuple – usually the predicate name. Each labelled branch is traversed independently using a semi-determinisitic *follow* instruction which fails when the sub-index referenced by the branch does not exist (i.e. there are no tuples represented in the sub-index). Likewise, labelled boolean values which indicate the presence of tuples in Δ are queried independently using semi-deterministic *test* (for true) instructions. To ensure independence of the follow and test instructions each appears in a new code block so that the failure of one set of instructions does not affect others. When stratification priorities specify the stratification order of predicate names, care must be taken to order the independent code blocks accordingly. That is, if a predicate p is stratified before predicate q then the follow or test instructions which search for p tuples must occur in the SDSL program before instructions that search for q tuples.

Using a consistent and predetermined order for processing the minimum tuples in Δ ensures fairness with respect to the stratification order. By testing the contents of the argument index, all labelled branches and all labelled boolean values in a consistent order, the evaluation of any sub-set of rules which can produce an infinite chain of tuples is interleaved with the evaluation of the remaining rules. Of course, other stratification orders can be specified that allow programs to be evaluated unfairly.

After a tuple has been identified as the minimum in Δ in some code block it is used to activate rules (see the next section). When rule activation is complete it is necessary to delete the tuple before the next iteration. At the end of the code block which identifies the minimum tuple in Δ, a *delete* instruction can remove values from argument indexes which represent tuples. After the *delete* is performed, the value will no longer exist in the argument index and so the tuple will not be found in successive searches.

Although we use *delete* instructions to remove each minimal value from an argument index, we can optimise the deletion of unordered values found using non-deterministic *scans* by removing the entire index node after all sub-indexes have been explored. Ordinarily, multiple delete operations are necessary to remove all elements from an argument index. However if the entire index node is removed after the scan is complete only 1 instruction is required in the SDSL code. Another important advantage of this optimisation is that because deletion of elements in an argument index does not occur during a *scan*, the data structures representing argument indexes and their associated *scan* operations are simplified (i.e. the size of the data structure is not reduced during *scans*). To allow the deletion of the entire index node to occur after the scan is complete, the *scan* instruction and any subsequent instructions associated with the values it finds are encapsulated in a code block. When the code block exits after all values have been found by the *scan*, the index node where the *scan* was performed is removed using a single instruction.

To remove a tuple represented by a labelled boolean value, the boolean value is assigned `false` using a *set* instruction.

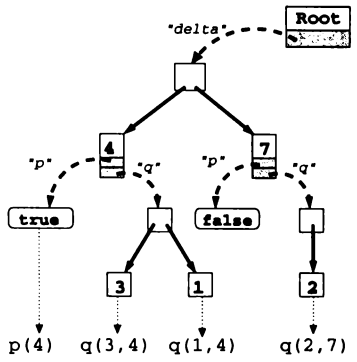
The process of removing tuples from an index node may cause the parent nodes to become redundant and so be deleted. To determine if a node is redundant we test that (1) the argument index is empty using the *empty* instruction, (2) all labelled boolean values are `false` using the *test* (for false) instruction, and (3) all labelled branches do not reference any index nodes using the *is-pruned* instruction. In the event that the redundant index node is the sub-index associated with a labelled branch, the sub-index referenced by the branch is removed using the *prune* instruction. By convention the index node referenced by the "`delta`" labelled branch is not removed even when it becomes redundant because an empty Δ set will cause termination in the next iteration.

This is a confusing process that is best explained using an example. In Figure 5.2 the Δ set holds tuples from two predicates. (Program rules have

Stratification Priorities:

```
stratify p(X) [X,p].
stratify q(_,X) [X,q].
stratify p << q.
```

Example Index Structure Instance:



SDSL code to find and delete minimum tuples from Δ:

```
| {
|    follow:semidet($0, delta, $1)
|    minimum:nondet($1, MinVar0:int, $2)  % Found minimum argument value
|    {
|       test:semidet($2, p, true)         % Test for p(MinVar0)
|       {
|          % Found h tuple as p(MinVar0)
|          ...
|       }
|       set:det($2, p, false)             % Delete p(MinVar0) tuple
|       ispruned:semidet($2, q)           % Test if $2 is redundant
|       test:semidet($2, p, false)
|       delete:det($1, MinVar0)           % Delete $2
|    }
|    {
|       follow:semidet($2, q, $3)         % Test for q(_,MinVar0)
|       {
|          scan:nondet($3, MinVar1:int, $4) % Finding q(MinVar1,MinVar0)
|          {
|             % Found h as tuple q(MinVar1,MinVar0)
|             ...
|          }
|       }
|       prune:det($2, q)                  % Delete all q(_,MinVar0)
|       ispruned:semidet($2, q)           % Test if $2 is redundant
|       test:semidet($2, p, false)
|       delete:det($1, MinVar0)           % Delete $2
|    }
| }
```

Figure 5.2: Finding and deleting the minimum element in Δ using SDSL.

been omitted in this example due to their irrelevance when finding and deleting a minimum tuple.) We arrange each tuple in Δ where arguments or functors that occur earlier in the stratification priorities occur earlier in the index structure. In Figure 5.2 the stratification order of arguments and predicate names are represented in the index structure instance using a left-to-right order, where values or predicate names that are on left branches are stratified before those to their right (with the exception of the first argument of `q/2` tuples, which are unordered). The SDSL code necessary to repeatedly find and remove minimal elements from Δ is given below the index structure instance.

Using the given index structure instance, the code first locates the Δ set using a follow instruction from the root index node (`$0`). The first argument indexed in Δ is used to stratify tuples from both `p/1` and `q/2` predicates. Therefore a *minimum* instruction is used to locate the smallest value for the variable `MinVar0` (in this case it will be bound to 4) and binds the new index variable `$2` to its sub-index. At the index node referenced by `$2` the set of tuples is split into two subsets (representing tuples from the two different predicates) using a labelled branch and a labelled boolean value. Because each predicate is processed independently each is encapsulated in its own code block. The order that predicates are processed depends on how they are stratified. In this case, the `p/1` predicate is explicitly stratified before the `q/2` predicate, as shown in the stratification priority. The presence of a `p/1` tuple is determined using a *test* instruction on the labelled boolean value that represents `p/1` tuples. If a `p/1` tuple is found (i.e. if the boolean value representing it is `true`) then, after it has been used to activate rules (see the next section), it is deleted by setting the labelled boolean value to `false`. Any index nodes which are made redundant by the removal of this `p/1` tuple are found and removed from the index structure instance. The *ispruned* and *test* instructions check for the absence of a sub-index beneath the `q` labelled branch and test that the `p` labelled boolean value is `false` in the index node referenced by `$2`. The index node referenced by `$2` is removed by a *delete* instruction performed on the parent of `$2`. When the end of the code block is reached backtracking is enforced, however because there are no non-deterministic choice points in the block where `p/1` tuples are found this code block is exited.

The next code block in the sequence is entered which searches for the presence of `q/2` tuples. `q/2` tuples are found using two instructions: a *follow* instruction is used to locate `$3` – the sub-index where `q/2` tuples are stored – followed by a non-deterministic *scan* operation to find all possible binding of `Minvar1` – `q/2`'s first variable. A *scan* is used here rather than a *minimum* since the values of `Y` do not affect the stratification order of `q/2` tuples so can be processed in an arbitary order. After all `q(_,MinVar0)` tuples have been found and used to activate rules they can be deleted from Δ. Rather than delete these tuples individually when they are found during the *scan*, the entire index node is removed after the *scan* is complete. This is achieved using a *prune* instruction because the index node to be deleted is referenced by a labelled branch. Any index nodes that are made redundant from the deletion of `q/2` tuples are found and deleted using the same set of instructions as in the previous code block.

5.4 Delayed Querying of Negated Goals

Before the minimal tuple in Δ (h) is used to activate rules, any unresolved negated goals which constrain this tuple must be evaluated. The reasons for delaying evaluation of the negated goals of a rule are given in Chapter 2. Note that the delayed evaluation of negated goals is not necessary when they can be optimised using early evaluation (see Section 2.3.4).

To satisfy negated goals they are queried in the Γ set. To ensure negated goals fail when all internal instructions succeed, the SDSL instructions associated with each negated goal are encapsulated in a negated code block. Using the path defintion in Γ of the goal's predicate as a guide, semi-deterministic *look* and non-deterministic *scan* instructions are used when performing known value and unknown value searches for argument values, respectively. Semi-deterministic *follow* instructions are used to follow labelled branches that occur in the goal's path definition. Semi-deterministic *test* instructions are used to determine whether labelled boolean values which represents a tuples contain the value `true`. For more details about querying an index structure instance see Section 3.3.1. Built-in operations that are included with negated goals are translated to their SDSL equivalents and included in the negated code block.

The SDSL program in Figure 5.4 gives an example of the delayed querying of a negated goal. Approximately two-thirds of the way through the SDSL program a negated goal is queried inside a negated code block. Note that this example program also attempts early failure of the negated goal in another negated code block earlier in the program.

5.5 Adding True Tuples to Γ

After all unresolved negated goals associated with h (the minimum tuple in Δ) have been successfully evaluated, h is an unconditionally true tuple and can be added to Γ. However if h is an exclusive trigger head (as described in Section 2.3.6) the program is optimised by omitting this step.

The process of adding a tuple into an index structure is described in detail in Section 3.3.2. In this case the index path definition associated with the predicate of h in Γ determines the instructions used during the insertion process. More precisely, *insert* instructions are used to insert argument values into index nodes, *establish* instructions ensure that sub-indexes associated with labelled branches exist, and *set* instructions assign `true` to any labelled boolean value. Because all these operations are deterministic there is no need to isolate these instructions in a separate code block.

The SDSL program in Figure 5.4 has two occurrences where minimal tuples are added to Γ. These are commented for ease of reference.

5.6 Activation of Rules

Given that h is a new, true tuple, h is now used to activate program rules (denoted as $h_r \leftarrow \beta_r$). Each rule is encapsulated in its own code block to prevent interference with others. The code blocks for all rules activated by h are ordered arbitrarily because the output of one rule (h_r) will be irrelevant to the evaluation of the other rules because h_r is always stratified after the positive body goals of rules activated by h, and the evaluation of negated goals stratified after their trigger goal are delayed. The first stage of activating a program rule using Triggering Evaluation is to match h with the trigger goal in each rule (β^{τ}).

Because the predicate names of all goals and tuples are static fields, some of the matching can be performed at compile time. It should be noted that with all negated goals associated with h now resolved, the predicate of a tuple or goal reverts to its traditional interpretation as the static elements of a term (i.e. the combination of the functor and arity). In SDSL triggered programs, separate code blocks are used to deal with new tuples from each predicate (from Section 5.3). For each of these code blocks the only rules that can be successfully activated by an h tuple are those whose trigger goal is from the same predicate as h. Since the predicates of goals and any h tuple are known in the predicate's code block at compile time, the set of rules to be activated in each code block is reduced, resulting in more efficient compiled programs.

Assuming the predicate of a rule's trigger goal and the predicate of h are identical, the arguments of the trigger goal are now matched with the ground arguments of h. Arguments in h are represented by bound program variables in the SDSL program. Matching arguments which are free variables in a trigger goal with arguments in h can be achieved at a syntactic level by replacing all occurances of the trigger goal's variable with the variable in h throughout the SDSL representaion of the rule. Matching arguments with constant terms in the trigger goal and arguments in h requires using a built-in equality test, such as `builtin:semidet(equals,[X,5],[])` where `X` is the argument in h and `5` is the constant argument in a trigger goal.

After the SDSL code to ensure a match between h and the trigger goal, the remaining positive and negated goals (those in β^{+}, β^{-}) are queried in Γ, and built-in operations (β^{λ}) are evaluated to determine if the rule's body is true. Searching for tuples matching a single goal in an index structure instance has been described previously in Section 3.3.1, and more recently in this chapter when describing the evaluation of negated goals. However this discussion needs to be extended for evaluating a set of goals. All SDSL instructions necessary to evaluate the rule body are added sequentially one after the other so that the failure of any element in the body will invoke backtracking though all instructions in the rule's body. With the exception of the trigger goal and negated goals which are evaluated late (see Section 2.3.4), body goals and built-in operations are satisfied in a left-to-right order (a continued assumption from Chapter 3). With the previously noted exceptions, program variables are only free the first time they are encountered in a left-to-right ordering because all goals and built-in operations bind new program variables when they are satisfied. The binding patterns of variables affects the instructions used in the SDSL program such that when searching in Γ for the value of a program variable, a *scan* instruction is used the first time the variable is encountered (since it is free), however a *look* instruction is used for all subsequent searching (since the program variable is bound by the *scan*). Similarly, the variables used in built-in operations appear in the set of output parameters when they are first encountered however will occur in the set of input parameters when the variable is bound.

The instructions necessary to satisfy negated goals in a rule body are evaluated in negated code blocks. The use of negated code blocks restricts the scope of existential variables created within the negated goal. However if evaluation of the negated goal is optimised by late evaluation (see Section 2.3.4), the negated goal does not need to be queried with the other body goals, and instead is delayed.

Following the instructions which evaluate the body of a rule, the rule's head (h_r) is inserted into Δ. Using the head predicate's index path definition in the Δ set as a guide, tuples are inserted into the index structure instance using the same set of instructions as previously described when adding new tuples to Γ. When h_r is a non-trigger head then it is inserted into Γ rather than Δ. This is achieved by following the index path definition for h_r in Γ instead of that for Δ. (See Section 2.3.5 for more details and the preconditions of this optimisation.)

When all instructions necessary for the evaluation of a rule body and assertion of its head are complete, the rule's code block is closed. This enforces backtracking within the rule so that multiple outputs can be generated from a rule when it is activated by a new tuple.

Any side-effects associated with h tuples (i.e. side effects generated by the head of a rule) are performed in a new code block, and included beside the code blocks that activate and evaluate rules. Tuples from output predicates use the SDSL built-in `print` instruction to output terms to the standard output. For example, if the current minimal tuple in Δ is found as `print(MinVar0)` then `MinVar0` is printed by including incuding the SDSL built-in operation `builtin:det(print,[MinVar0],[])` in a new code block together with the code blocks for the rules that `print(MinVar0)` activates. Input request tuples use the SDSL built-in instruction `builtin:det(input,[],[Input:string])` in a new code block to collect input from the user. (Note that an SDSL print instruction can be inserted before the input instruction when a prompt is required.) When input is received it is bound to the `Input` variable and inserted into the index structure according to the index path definition of the input's result tuple. In this way user input can be referred to by other rules in the program.

To illustrate the evaluation of a rule in SDSL we provide Figure 5.3. The given rule in this figure has a trigger tuple (shown in bold) with a constant value for its first argument. The remainder of the body contains a positive goal, a built-in operation and a negated goal. To clarify the choice of SDSL instructions the mode of each variable is given before each variable using '+' and '-' symbols for ground and free, respectively. The index path definitions are sufficient to express this rule in SDSL.

Before the SDSL code is given for the program rule we must assume that h, the minimum tuple found in Δ, is from predicate `p/2` and its first argument is represented by the variable `MinVar0` and the second by variable `MinVar1` (such that it forms the term `p(MinVar0,MinVar1)`). Because of this assumption, the SDSL code which represents the rule must exist inside the code block where minimum tuples in Δ belonging to the `p/2` predicate are processed. The SDSL representation of the rule is encapsulated in a code block so that its evaluation is independent of other rules.

The first SDSL instruction in Figure 5.3 ensures a match between the first argument of the trigger goal (3) and the first argument of the minimum $p/2$ tuple (`MinVar0`). This is necessary because neither term is a free variable and is achieved using a semi-deterministic built-in which succeeds only when the two terms are equal. The second argument of the trigger goal is a free variable, therefore a second equality test is unnecessary to prove it matches with `MinVar1`.

Stratification Priorities:

```
index gamma q(X,Y) [X, Y].
index gamma r(X)   [X, boolean(r)].
index delta s(X,Y) [branch(delta), X, Y].
```

Starlog rule with modes:

```
s(Y,Z) <- p(3,Y), q(-Y,+Z), +W is -Z+1, not(r(-W)).    % Rule 1
```

SDSL code for Rule 1:
(assumes p(MinVar0,MinVar1) has been found as a minimal element in Δ)

```
| {
|   builtin:semidet(equals, [MinVar0,3], []) % Match trigger goal with
|                                            %   minimum delta tuple:
|                                            %     MinVar0 = 3,
|                                            %     MinVar1 = Y
|
|   look:semidet($0, MinVar1, $1)      % Query gamma for q(MinVar1,Z1)
|   scan:nondet($1, Z1:int, $2)
|
|   builtin:det(add, [Z1,1], [W1:int]) % Perform built-in W1 is Z1+1
|
|   not{                               % Begin negated goal
|        look:semidet($0, W1, $3)      % Query gamma for r(W1)
|        test:semidet($3, r, true)
|   }
|
|   establish:det($0, delta, $4)       % Insert s(MinVar1,Z1)
|   insert:det($4, MinVar1, $5)        %     into delta
|   insert:det($5, Z1, $6)
| }
```

Figure 5.3: Example Starlog rule and equivalent SDSL code.

Instead, matching the free variable Y with MinVar1 is achieved statically by replacing the variable Y with MinVar1 everywhere in the rule. After matching *h* with the trigger goal, the remainder of the body is evaluated.

The first non-trigger goal requires searching the Γ set for a q/2 tuple. Using the index path definition for q/2 tuples in Γ, tuples from this predicate are indexed from the root of the index structure initially on their first argument and then on their second. The first argument of the q/2 goal (MinVar1) is ground by the instantiation of the trigger goal. Therefore a semi-deterministic *look* instruction is used to determine if Γ holds this value. If successful, the next instruction searches in the sub-index returned by the *look* for the second argument of the q/2 goal. Recall from Chapter 4 that new program variables appearing in SDSL programs append a rule identifier to distinguish variables if their scope changes due to optimisation. This is the case for the Z variable in the rule which has become Z1 in the SDSL code. Because this variable is unbound in the rule a non-deterministic *scan* instruction is necessary to explore all variable bindings within the argument index.

The next element to be evaluated from the body of the rule is the addition operation. The Starlog operation is mapped to its SDSL equivalent built-in using the predefined mapping from Table 4.2. By analysing the binding patterns of the variables the correct sequences of input and output variables is used.

The final element in the body is a negated goal. Negated goals are evaluated inside negated code blocks which invert the success or failure of the internal operations. This negated goal succeeds if there are no r/1 tuples whose argument is W1. To query Γ for the presence of r/1 tuples we follow the r/1 predicate's index path definition. The path definition indicates that the argument bindings of r/1 tuples are stored in the root index. Because the sought argument (W1) is ground a semi-deterministic *look* instruction is used. The final element in the path definition indicates that a labelled boolean value specifies the presence of r/1 tuples. This labelled boolean value is queried using a *test* instruction which succeed when the value is true.

When all body goals have been satisfied the head of the rule is asserted in Δ. The index path definition associated with the head predicate (s/2) directs the insertion operations. According to the path definition, s/2 tuples stored in Δ are stored down the static branch labelled as "delta". Because there is no guarantee that a sub-index exists for this labelled branch an *establish* instruction is used. The first and second arguments of the head tuple are inserted into the index structure instance separately using *insert* instructions.

5.7 Example Triggered SDSL Program

To demonstrate the integration of each step of a Triggered SDSL program we present a complete example program that is compiled to SDSL in Figure 5.4.

This program generates tuples from the two predicates p/1 and q/1. The stratification order specified by the stratification priorities indicates that all tuples are initially stratified on their argument value and then on their predicate name where q/1 tuples are stratified before p/1 tuples when their arguments are equal.

The index definitions given at the start of the SDSL program have been generated automatically using techniques from Chapter 3. (A diagram of the index structure used in this program is given in Figure 5.5.) The first code block in the SDSL program uses the path definition of q/1 tuples in Δ to store fact q(0). Program rules are repeatedly applied in the second code block. The first set of operations find the minimum argument value in the Δ set and assign this value to MinVar0. These operations fail only when Δ becomes empty.

Given that Δ contains at least one tuple, the program checks to see if there is a q/1 tuple in Δ with the value of MinVar0 as its argument. The q/1 predicate is processed before p/1 as specified by p $\gg$ q in the stratification order. To determine whether a q/1 tuple exists with the value of MinVar0 as an argument, the boolean value labelled id3 in the sub-index associated with the MinVar0 branch is tested. Minimum q/1 tuples are immediately added to Γ by inserting the value of MinVar0 into the first index of Γ and setting the labelled boolean value id2 to true. Rule 1 is the first (and only) rule that is activated by a new q/1 tuple. The negated goal not(q(Y), Y>1) in the body of Rule 1 is evaluated by searching for q(Y) tuples in Γ where Y > 1, within a negated code block. This search allows for early failure of the negated goal rather than conclusive evaluation because the goal is unstratified with respect to the trigger goal. Because there are no other goals in Rule 1, the remaining program variable operations are translated into their built-in equivalents.

Once the body of Rule 1 is proven true the rule's head tuple is added to Δ by following the path definition of p(X) in Δ. To ensure that a sub-index exists beneath the delta labelled branch an *establish* instruction is used. When all program rules activated by a q/1 tuple have been processed, the minimum q/1 tuple is deleted from Δ. This is achieved by first setting the boolean value labelled id3 to false and then, if the index node containing id3 has become redundant, deleting the index node.

After q/1 tuples have been processed the program attempts to find a minimum p/1 tuple. If a minimum tuple from predicate q/1 exists it will be constrained by the unresolved negated goal not(q(Y), Y>10) that occurs in the body of the only rule capable of producing q/1 tuples.

This negated goal is evaluated by searching Γ for a contradicting tuple within a negated code block. This query could be omitted if the early evaluation optimisation were applicable (where the negated goal can be proven to be stratified before the rule's trigger).

When no contradicting tuple exist, the minimum q/1 tuple is considered unconditionally true and is added to Γ by inserting MinVar0 into the root node of Γ and setting the sub-index's boolean value labelled id0 to true.

The remaining positive body goal in Rule 2, q(X), is searched for in Γ given that X has been matched with the argument of the trigger goal (now represented by MinVar0 in the SDSL program). The additional operations are translated into their built-in equivalents. The output q/1 tuple from Rule 2 is added to Δ using the *establish*, *insert* and *set* instructions. Finally the p/1 tuple found as the minimum of Δ is deleted from this set by setting the boolean value representing it to false and then removing the value of MinVar0 from the argument index in Δ if its sub-index has become redundant.

Next we discuss the modifications necessary to SDSL triggered programs that allow Δ and Γ to share index nodes.

Starlog program:

```
stratify  p(X)  [X, p].
stratify  q(X)  [X, q].
stratify  q << p.
q(0).
p(Z) <- q(X), not(p(Y), Y>1), Z is 2*X.    % Rule 1
q(Z) <- p(X), q(X), X<10, Z is X+1.        % Rule 2
```

SDSL Program:

```
| index gamma p(X)                  [X, boolean_value(id0)]
| index delta p(X) <- not(q(_), _>1)  [branch(delta), X, boolean_value(id1)]
| index gamma q(X)                  [X, boolean_value(id2)]
| index delta q(X)                  [branch(delta), X, boolean_value(id3)]
|
| {                                           % Add fact q(0) to delta
|   establish:det($0, delta, $1)
|   insert:det($1, 0, $2)
|   set:det($2, id3, true)
| }
| {
|   follow:semidet($0, delta, $3)             % Find minimum timestamp value
|   minimum:nondet($3 MinVar0:int, $4)
|   {
|     test:semidet($4, id3, true)             % Test if a minimum q/1 tuple exists
|     {
|       insert:det($0, MinVar0, $5)           % Add minimum tuple to gamma
|       set:det($5, id2, true)
|       {                                     % Evaluate Rule 1
|         not{                                % Test negated goal for early failure
|            scan:nondet($0, Y1:int, $6)
|            test:semidet($6, id2, true)
|            builtin:semidet(greaterThan, [Y1,1], [])
|         }
|         builtin:det(multiply, [MinVar0,2], [Z1:int])
|         establish:det($0, delta, $7)        % Add head to delta
|         insert:det($7, Z1, $8)
|         set:det($8, id1, true)
|       }
|     }
|     set:det($4, id3, false)                 % Delete minimum from delta
|     test:semidet($4, id1, false)
|     delete:det($3, MinVar0)
|   }
|   {
|     test:semidet($4, id1, true)             % Test if a minimum p/1 tuple exists
|     {
|       not{                                  % Evaluate unresolved negated goal late
|          scan:nondet($0, MinVar1:int, $9)
|          test:semidet($9, id2, true)
|          builtin:semidet(greaterThan, [MinVar1,1], [])
|       }
|       insert:det($0, MinVar0, $10)          % Add minimum tuple to gamma
|       set:det($10, id0, true)
|       {
|         look:semidet($0, MinVar0, $11)      % Evaluate Rule 2
|         test:semidet($11, id2, true)
|         builtin:semidet(greaterThan, [10,MinVar0], [])
|         builtin:det(add, [MinVar0,1], [Z2:int])
|         establish:det($0, delta, $12)       % Add head to delta
|         insert:det($12, Z2, $13)
|         set:det($13, id3, true)
|       }
|     }
|     set:det($4, id1, false)                 % Delete minimum from delta
|     test:semidet($4, id3, false)
|     delete:det($3, MinVar0)
|   }
| }
```

Figure 5.4: Example Starlog and equivalent SDSL program.

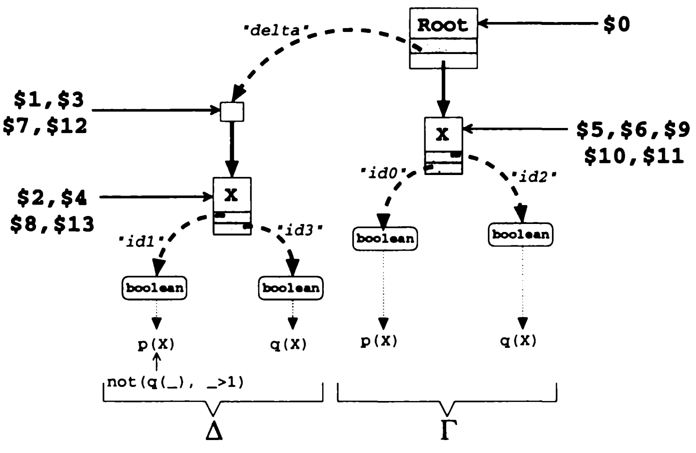


Figure 5.5: Index structure schema used by example program in Figure 5.4.

5.8 SDSL Triggered Programs with Combined Δ and Γ Sets

In Chapter 3 the advantages and disadvantages of combining index nodes for the Δ and Γ sets were discussed. Sharing index nodes between these two sets can lead to a smaller index structure and fewer instructions needed to move tuples from one set to another, however each set is no longer specialised for its particular mode of access. Because the degree of optimisation is difficult to predict – and in many cases may be detrimental to performance – we disallowed sharing index nodes between Δ and Γ in automatically generated index structures.

However programmers may wish to manually define an index structure where such index node sharing occurs when they have an insight into the performance of their program. For the correct evaluation of these programs the translation of programs from Starlog to SDSL must be modified. Although this section explores the requirements for SDSL programs with shared indexes, the remaining thesis does not consider these programs further. It is included here for completeness.

When the Δ and Γ sets are combined in the index structure, finding minimum values of each argument from Δ is more difficult than when the sets are separated. This is because the minimum value in any index may point to a sub-index that does not contain any tuples from Δ, and instead holds only Γ tuples. In these cases the next lowest minimum value must be repeatedly explored until a value is located that holds at least one Δ tuple. To perform this type of searching requires using *next-minimum* instructions rather than *minimum* instructions when finding the minimum value for an argument. *Next-minimum* instructions return all values in an argument index in ascending order, one value at a time. In this way an index can hold both the argument values of tuples in Γ and the argument values in Δ, however the *next-minimum* operation restricts searches to unseen values (i.e. only those in Δ).

To illustrate the difference between SDSL programs that share index nodes between Δ and Γ and those that do not, we provide Figure 5.6. This figure is the same example program used in Figure 5.4 but uses a different index structure. A diagram of the index structure is given in Figure 5.7. The major distinction between the two SDSL programs in these figures is the use of a *next-minimum* instruction rather than using a *minimum* when finding the lowest value of a tuple's argument. As written, the SDSL program which shares index nodes between Δ and Γ is no smaller than the original SDSL program where these sets are disjoint. However the new program does contain more redundant instructions than the original which can be removed using optimisation techniques from the next section. By applying these code optimisation techniques to the two programs the new program with index nodes shared between Δ and Γ will contain three fewer instructions than when the original is optimised (22 versus 25). Whether this leads to a more efficient execution depends on the program since the complexities of operations performed on shared indexes are likely to increase.

Following the guidelines in the previous sections, triggered programs can be implemented in SDSL. However these programs may contain redundant operations which reduce efficiency. The next sections discuss a number of optimisations that can be applied to SDSL programs.

5.9 Optimisations

The automatic generation of SDSL programs from Starlog programs may generate sub-optimal code. That is, the SDSL program is likely to repeat instructions or perform redundant operations. We now present a series of optimisations to remove redundant or repeated instructions from SDSL programs.

Each optimisation is demonstrated on the example SDSL program in Figure 5.4. To assist with understanding this example, a diagram of the program's index structure is provided in Figure 5.5. The index variables used throughout the program are included in this diagram with references to the index nodes they will hold. As the various optimisations are applied, new SDSL programs are produced. To reduce space the newly derived, optimised programs do not include the index definitions as these remain unchanged in all versions.

All the optimisations presented in this section are safe for SDSL programs that implement Triggering Evaluation as presented earlier in this chapter. Such programs are said to follow the *Triggering Template.* In spite of this, the conditions that must be satisfied before the optimisation is considered safe are explicitly defined to allow optimisation of more general SDSL programs.

5.9.1 Removal of Repeated Instructions Within Code Blocks

Automatically generated code blocks frequently repeat operations. This often occurs within rules where tuples satisfying different goals share indexes. By removing repeated instructions, programs may be optimised without affecting their correctness.

SDSL instructions do not have to be identical to consider one a repetition of another. Any instruction that will always generate the same output as a previous instruction is redundant. For example, after a `scan` instruction instantiates the program variable `X` and binds `$7` to the sub-index, any `look` instruction

Starlog program:

```
q(0).
p(Z) <- q(X), not(p(Y), Y>1), Z is 2*X.  % Rule 1
q(Z) <- p(X), q(X), X<10, Z is X+1.      % Rule 2
```

SDSL Program:

```
| index gamma p(X)                     [X, boolean_value(id0)]
| index delta p(X) <-- not(q(_), _>1) [X, boolean_value(id1)]
| index gamma q(X)                     [X, boolean_value(id2)]
| index delta q(X)                     [X, boolean_value(id3)]
|
| {                                                  % Add fact q(0) to delta
|   insert:det($0, 0, $1)
|   set:det($1, id3, true)
| }
| {
|   nextminimum:nondet($0, MinVar0:int, $2)          % Find minimum timestamp
|   {
|     test:semidet($2, id3, true)                    % Test if a minimum q/1 tuple exists
|     {
|       insert:det($0, MinVar0, $3)                  % Add minimum tuple to gamma
|       set:det($3, id2, true)
|       {                                            % Evaluate Rule 1
|         not{                                       % Test negated goal for early failure
|            scan:nondet($0, Y1:int, $4)
|            test:semidet($4, id2, true)
|            builtin:semidet(greaterThan, [Y1,1], [])
|         }
|         builtin:det(multiply, [MinVar0,2], [Z1:int])
|         insert:det($0, Z1, $5)                     % Add head to delta
|         set:det($5, id1, true)
|       }
|     }
|     set:det($2, id3, false)                        % Delete minimum from delta
|     test:semidet($2, id0, false)
|     test:semidet($2, id1, false)
|     test:semidet($2, id2, false)
|     delete:det($0, MinVar0)
|   }
|   {
|     test:semidet($2, id1, true)                    % Test if a minimum p/1 tuple exists
|     {
|       not{                                         % Evaluate unresolved negated goal late
|          scan:nondet($0, MinVar1:int, $6)
|          test:semidet($6, id2, true)
|          builtin:semidet(greaterThan, [MinVar1,1], [])
|       }
|       insert:det($0, MinVar0, $7)                  % Add minimum tuple to gamma
|       set:det($7, id0, true)
|       {
|         look:semidet($0, MinVar0, $8)              % Evaluate Rule 2
|         test:semidet($8, id2, true)
|         builtin:semidet(greaterThan, [10,MinVar0], [])
|         builtin:det(add, [MinVar0,1], [Z2:int])
|         insert:det($0, Z2, $9)                     % Add head to delta
|         set:det($9, id3, true)
|       }
|     }
|     set:det($2, id1, false)                        % Delete minimum from delta
|     test:semidet($2, id0, false)
|     test:semidet($2, id2, false)
|     test:semidet($2, id3, false)
|     delete:det($0, MinVar0)
|   }
| }
```

Figure 5.6: Example Starlog and equivalent unoptimised SDSL program with all index nodes shared between Δ and Γ.

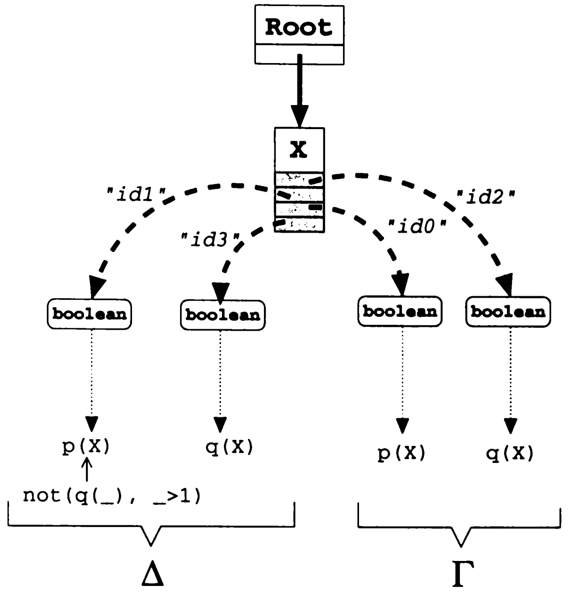


Figure 5.7: Index structure used by example program in Figure 5.6.

performed on the same index searching for ground variable X will find the same sub-index held by $7. Therefore the look instruction is redundant.

When a repeated instruction is removed from a program, subsequent instructions that use the output of the removed instruction must be updated. Instructions that use the output of a removed instruction are modified to use the output of the instruction that is equivalent to that which was removed. That is, if the operations of instruction Q are later repeated by instruction R then any instructions that use the output of instruction R must instead use the output of Q when R is removed. To allow instructions that originally used the output of R to use the output of Q, the output variables of Q must be accessible. Therefore two instructions can be optimised only when the later (repeated) instruction lies within the variable scope of the first instruction. Variables in SDSL programs are only accessible in the code block where they first occur, or in any code block which is nested within the code block where the variable first occurs. Therefore this is the scope of the repeated instruction optimisation.

Two instructions are repetitions of each other if they always produce the same output. However if the data accessed by these instructions is updated between the repeated instructions then their output may differ. Therefore repeated SDSL instructions that operate on some value are removed only when no other instructions "interfere" with the value between the repeated instructions. Interference between instructions occurs when the first instruction adds or modifies data which is examined by the second instruction. Interference is detected when there is an intersection between the values added or removed and those searched for in the same argument indexes, same labelled branches or same labelled boolean values. By analysing constraints on the variables in an SDSL program, potential intersections can be identified statically and this optimisation can be restricted to safe cases.

However such safety conditions are unnecessary for SDSL programs that follow the Triggering Template. In these programs there is no single code block where the contents of an argument index, labelled branch or labelled boolean value are modified in a way which affects repeated searches. For proof we consider all the steps in which the index structure instance is modified by Triggering Evaluation. The addition of facts to Δ occurs before either Δ or Γ are queried, therefore there are no prior search instructions which are repetitions of later instructions. New tuples (h) are added to Γ after they have been found as a minimum tuple in Δ, but only after all negated goals are satisfied. This update of Γ does not interfere with repeated instructions because the instructions prior to this insertion code involve searching Δ – a set unaffected by the addition of tuples in Γ – and searching Γ to satisfy the unresolved negated goals associated with h ($h \leftarrow \beta^{-}$). Although the instructions used to perform negated goals appear inside a negated code block, for the moment we will assume that the instructions appear outside of the block, which can happen as a result of other code optimisations. The instructions used to add h to Γ do not interfere with those used to solve negated goals since $h \gg n$ where $n \in \beta^{\sim}$ such that all newly added h tuples are irrelevant to n. The addition of new tuples into Δ as the output of a rule also does not interfere with any later searches because the instructions to add tuples to Δ always appear at the end of a code block. Likewise, the code to delete the minimum tuple from Δ occurs toward the end of a code block where the only other searches test the redundancy of an index node, a process which is not repeated previously in the code block.

Figure 5.9 shows the complete set of repeated instruction optimisations that can be performed on an SDSL program. These optimisations are given as rewrite rules which map programs with a repeated instruction to those without that repeatition. Here $T \longrightarrow U$ means that T is rewritten to U [22]. The brackets { and } represent code blocks of any type. Variables of type $\mathbf{S_i}$ represent sequences of SDSL instructions (which may include nested code blocks). The non-interference operator is denoted as # such that $V\#W$ is true if all the SDSL instructions in set V do not interfere with the SDSL instructions in set W, and is false otherwise.

The non-interference operator (#) is defined in Figure 5.8. Informally stated, V interferes with W if there is an update instruction (either an *insert*, *delete*, *establish*, *prune*, *link* or *set*) in V and a read instruction (a *look*, *scan*, *minimum*, *nextminimum*, *empty*, *follow*, *ispruned* or a *test*) in W which access the same value in an argument index, the same labelled branch or the same labelled boolean value. For example, if a *delete* instruction removes value X from an argument index then any subsequent *scan* instructions operating on the same argument index will not find X. Likewise, any *look* instruction searching for X will fail. In this example the *delete* interferes with the *scan* and the *look* instructions. (Depending on the update instruction, *insert*, *establish* and *set* instructions are sometimes considered read instructions since they access data and can produce different output when interfered with, as indicated in Figure 5.8.)

To clarify the optimisations in Figure 5.9 the first optimisation is explained in detail. This optimisation removes a `look` instruction if an equivalent `look` instruction appears earlier in a code block. The pattern of SDSL instructions

$$
\begin{aligned}
V\#W \Leftrightarrow & (\forall\ \texttt{insert} : \texttt{det}(\$E, F, \$G) \in V, \nexists\ \texttt{look} : \texttt{R}(\$E, F, \$J) \in W\ \wedge \\
& \qquad \nexists\ \texttt{scan} : \texttt{R}(\$E, I, \$J) \in W\ \wedge \\
& \qquad \nexists\ \texttt{minimum} : \texttt{R}(\$E, I, \$J) \in W\ \wedge \\
& \qquad \nexists\ \texttt{nextminimum} : \texttt{nondet}(\$E, I, \$J) \in W\ \wedge \\
& \qquad \nexists\ \texttt{empty} : \texttt{semidet}(\$E) \in W)\ \wedge \\
& (\forall\ \texttt{delete} : \texttt{det}(\$E, F, \$G) \in V, \nexists\ \texttt{look} : \texttt{R}(\$E, F, \$J) \in W\ \wedge \\
& \qquad \nexists\ \texttt{scan} : \texttt{R}(\$E, I, \$J) \in W\ \wedge \\
& \qquad \nexists\ \texttt{insert} : \texttt{det}(\$E, F, \$J) \in W\ \wedge \\
& \qquad \nexists\ \texttt{minimum} : \texttt{R}(\$E, I, \$J) \in W\ \wedge \\
& \qquad \nexists\ \texttt{nextminimum} : \texttt{nondet}(\$E, I, \$J) \in W\ \wedge \\
& \qquad \nexists\ \texttt{empty} : \texttt{semidet}(\$E) \in W)\ \wedge \\
& (\forall\ \texttt{establish} : \texttt{det}(\$E, L, \$G) \in V, \nexists\ \texttt{follow} : \texttt{R}(\$E, L, \$J) \in W\ \wedge \\
& \qquad \nexists\ \texttt{ispruned} : \texttt{semidet}(\$E, L) \in W)\ \wedge \\
& (\forall\ \texttt{prune} : \texttt{det}(\$E, L) \in V, \nexists\ \texttt{follow} : \texttt{R}(\$E, L, \$J) \in W\ \wedge \\
& \qquad \nexists\ \texttt{establish} : \texttt{det}(\$E, L, \$H) \in W\ \wedge \\
& \qquad \nexists\ \texttt{ispruned} : \texttt{semidet}(\$E, L) \in W)\ \wedge \\
& (\forall\ \texttt{link} : \texttt{det}(\$E, L, \$G) \in V, \nexists\ \texttt{follow} : \texttt{R}(\$E, L, \$J) \in W\ \wedge \\
& \qquad \nexists\ \texttt{ispruned} : \texttt{semidet}(\$E, L) \in W)\ \wedge \\
& (\forall\ \texttt{set} : \texttt{det}(\$E, L, T) \in V, \nexists\ \texttt{test} : \texttt{semidet}(\$E, L, S) \in W\ \wedge \\
& \qquad \nexists\ \texttt{set} : \texttt{det}(\$E, L, S) \in W)
\end{aligned}
$$

Figure 5.8: Logical definition of the non-interference operator.

relevant to this optimisation is given as:

$\{\mathbf{S_0}\ \texttt{look}:M(\$A, B, \$C)\ \mathbf{S_1}\ \texttt{look}:N(\$A, B, \$D)\ \mathbf{S_2}\}$

The interpretation of this pattern is that within a single code block (denoted by { and }) there must exist two `look` instructions. The use of variable $\mathbf{S_0}$ (which can represent any number of instructions) means that the first `look` instruction does not have to be the first instruction in the code block. Variable $\mathbf{S_1}$ allows the two `look` instructions to be separated in the code by zero or more intermediate instructions. $\mathbf{S_2}$ at the end of the sequence means the second `look` instruction does not have to be the last instruction in the code block. The sharing of index variable $\$A$ and program variable B between the two `look` instructions ensures these operations are performed on the same index node and are searching for the same value. The use of different variables M and N to represent the determinisms of the `look` instructions means that the determinism can be different for each instruction. The conditions for this optimisation are given by $\exists\theta\ (\$C = \$D\theta) \wedge (\mathbf{S_1}\#[\texttt{look} : \texttt{N}(\$A, B, \$D)])$. The first part of this conjunction creates the variable substitution θ in which index variable $\$D$ is assigned to index variable $\$C$ (this is not really a condition because it can never fail but it is a necessary step). The second part of the conjunction ensures that the second `look` instruction is not interfered with by any instructions in $\mathbf{S_1}$. If interference does occur then this condition is false and the optimisation is not safe. The output of this optimisation is given by $\{\mathbf{S_0}\ \texttt{look}:M(\$A, B, \$C)\ \mathbf{S_1}\ \mathbf{S_2}\theta\}$ which is the original code block but with the second `look` instruction removed, and all the occurrences of its output ($\$D$) replaced in $\mathbf{S_2}$ with $\$C$ by applying variable substitution θ.

Identical Instruction Optimisations

$$\frac{\exists\theta\ (\$C = \$D\theta) \wedge (\mathtt{S_1}\#[\mathtt{look}:N(\$A, B, \$D)])}{\{\mathtt{S_0}\ \mathtt{look}:M(\$A, B, \$C)\ \mathtt{S_1}\ \mathtt{look}:N(\$A, B, \$D)\ \mathtt{S_2}\} \longrightarrow \{\mathtt{S_0}\ \mathtt{look}:M(\$A, B, \$C)\ \mathtt{S_1}\ \mathtt{S_2}\theta\}}$$

$$\frac{\exists\theta\ (\$C = \$D\theta) \wedge (\mathtt{S_1}\#[\mathtt{insert}:\mathtt{det}(\$A, B, \$D)])}{\{\mathtt{S_0}\ \mathtt{insert}:\mathtt{det}(\$A, B, \$C)\ \mathtt{S_1}\ \mathtt{insert}:\mathtt{det}(\$A, B, \$D)\ \mathtt{S_2}\} \longrightarrow \{\mathtt{S_0}\ \mathtt{insert}:\mathtt{det}(\$A, B, \$C)\ \mathtt{S_1}\ \mathtt{S_2}\theta\}}$$

$$\frac{\exists\theta\ (\$C = \$D\theta) \wedge (\mathtt{S_1}\#[\mathtt{minimum}:\mathtt{nondet}(\$A, B, \$D)]}{\begin{array}{c}\{\mathtt{S_0}\ \mathtt{minimum}:\mathtt{nondet}(\$A, B, \$C)\ \mathtt{S_1}\ \mathtt{minimum}:\mathtt{nondet}(\$A, B, \$D)\ \mathtt{S_2}\} \longrightarrow \\ \{\mathtt{S_0}\ \mathtt{minimum}:\mathtt{nondet}(\$A, B, \$C)\mathtt{S_1}\ \mathtt{S_2}\theta\}\end{array}}$$

$$\frac{\exists\theta\ (\$C = \$D\theta) \wedge (\mathtt{S_1}\#[\mathtt{follow}:\mathtt{semidet}(\$A, L, \$D)])}{\begin{array}{c}\{\mathtt{S_0}\ \mathtt{follow}:\mathtt{semidet}(\$A, L, \$C)\ \mathtt{S_1}\ \mathtt{follow}:\mathtt{semidet}(\$A, L, \$D)\ \mathtt{S_2}\} \longrightarrow \\ \{\mathtt{S_0}\ \mathtt{follow}:\mathtt{semidet}(\$A, L, \$C)\ \mathtt{S_1}\ \mathtt{S_2}\theta\}\end{array}}$$

$$\frac{\exists\theta\ (\$C = \$D\theta) \wedge (\mathtt{S_1}\#[\mathtt{establish}:\mathtt{det}(\$A, L, \$D)])}{\begin{array}{c}\{\mathtt{S_0}\ \mathtt{establish}:\mathtt{det}(\$A, L, \$C)\ \mathtt{S_1}\ \mathtt{establish}:\mathtt{det}(\$A, L, \$D)\ \mathtt{S_2}\} \longrightarrow \\ \{\mathtt{S_0}\ \mathtt{establish}:\mathtt{det}(\$A, L, \$C)\ \mathtt{S_1}\ \mathtt{S_2}\theta\}\end{array}}$$

$$\frac{\mathtt{S_1}\#[\mathtt{test}:\mathtt{semidet}(\$A, L, V)]}{\{\mathtt{S_0}\ \mathtt{test}:\mathtt{semidet}(\$A, L, V)\ \mathtt{S_1}\ \mathtt{test}:\mathtt{semidet}(\$A, L, V)\ \mathtt{S_2}\} \longrightarrow \{\mathtt{S_0}\ \mathtt{test}:\mathtt{semidet}(\$A, L, V)\ \mathtt{S_1}\ \mathtt{S_2}\}}$$

$$\frac{\mathtt{S_1}\#[\mathtt{set}:\mathtt{det}(\$A, L, V)]}{\{\mathtt{S_0}\ \mathtt{set}:\mathtt{det}(\$A, L, V)\ \mathtt{S_1}\ \mathtt{set}:\mathtt{det}(\$A, L, V)\ \mathtt{S_2}\} \longrightarrow \{\mathtt{S_0}\ \mathtt{set}:\mathtt{det}(\$A, L, V)\ \mathtt{S_1}\ \mathtt{S_2}\}}$$

$$\frac{\exists\theta\ O = P\theta}{\{\mathtt{S_0}\ \mathtt{builtin}:M(B, I, O)\ \mathtt{S_1}\ \mathtt{builtin}:M(B, I, P)\ \mathtt{S_2}\} \longrightarrow \{\mathtt{S_0}\ \mathtt{builtin}:M(B, I, O)\mathtt{S_1}\ \mathtt{S_2}\theta\}}$$

Equivalent Instruction Optimisation

$$\frac{\exists\theta\ (\$C = \$D\theta) \wedge (\mathtt{S_1}\#[\mathtt{look}:N(\$A, B, \$D)])}{\{\mathtt{S_0}\ \mathtt{scan}:\mathtt{M}(\$A, B:T, \$C)\ \mathtt{S_1}\ \mathtt{look}:\mathtt{N}(\$A, B, \$D)\ \mathtt{S_2}\} \longrightarrow \{\mathtt{S_0}\ \mathtt{scan}:\mathtt{M}(\$A, B:T, \$C)\ \mathtt{S_1}\ \mathtt{S_2}\theta\}}$$

$$\frac{\exists\theta\ (\$C = \$D\theta) \wedge (\mathtt{S_1}\#[\mathtt{look}:N(\$A, B, \$D)])}{\{\mathtt{S_0}\ \mathtt{insert}:\mathtt{det}(\$A, B, \$C)\ \mathtt{S_1}\ \mathtt{look}:\mathtt{N}(\$A, B, \$D)\ \mathtt{S_2}\} \longrightarrow \{\mathtt{S_0}\ \mathtt{insert}:\mathtt{det}(\$A, B, \$C)\ \mathtt{S_1}\ \mathtt{S_2}\theta\}}$$

$$\frac{\exists\theta\ (\$C = \$D\theta) \wedge (\mathtt{S_1}\#[\mathtt{look}:N(\$A, B, \$D)])}{\begin{array}{c}\{\mathtt{S_0}\ \mathtt{minimum}:\mathtt{nondet}(\$A, B:T, \$C)\ \mathtt{S_1}\ \mathtt{look}:\mathtt{N}(\$A, B, \$D)\ \mathtt{S_2}\} \longrightarrow \\ \{\mathtt{S_0}\ \mathtt{minimum}:\mathtt{nondet}(\$A, B:T, \$C)\ \mathtt{S_1}\ \mathtt{S_2}\theta\}\end{array}}$$

$$\frac{\exists\theta\ (\$C = \$D\theta) \wedge (\mathtt{S_1}\#[\mathtt{insert}:\mathtt{det}(\$A, B, \$D)])}{\{\mathtt{S_0}\ \mathtt{look}:\mathtt{M}(\$A, B, \$C)\ \mathtt{S_1}\ \mathtt{insert}:\mathtt{det}(\$A, B, \$D)\ \mathtt{S_2}\} \longrightarrow \{\mathtt{S_0}\ \mathtt{look}:\mathtt{M}(\$A, B, \$C)\ \mathtt{S_1}\ \mathtt{S_2}\theta\}}$$

$$\frac{\exists\theta\ (\$C = \$D\theta) \wedge (\mathtt{S_1}\#[\mathtt{insert}:\mathtt{det}(\$A, B, \$D)])}{\{\mathtt{S_0}\ \mathtt{scan}:\mathtt{N}(\$A, B:T, \$C)\ \mathtt{S_1}\ \mathtt{insert}:\mathtt{det}(\$A, B, \$D)\ \mathtt{S_2}\} \longrightarrow \{\mathtt{S_0}\ \mathtt{scan}:\mathtt{N}(\$A, B:T, \$C)\ \mathtt{S_1}\ \mathtt{S_2}\theta\}}$$

$$\frac{\exists\theta\ (\$C = \$D\theta) \wedge (\mathtt{S_1}\#[\mathtt{insert}:\mathtt{det}(\$A, B, \$D)])}{\begin{array}{c}\{\mathtt{S_0}\ \mathtt{minimum}:\mathtt{nondet}(\$A, B:T, \$C)\ \mathtt{S_1}\ \mathtt{insert}:\mathtt{det}(\$A, B, \$D)\ \mathtt{S_2}\} \longrightarrow \\ \{\mathtt{S_0}\ \mathtt{minimum}:\mathtt{nondet}(\$A, B:T, \$C)\ \mathtt{S_1}\ \mathtt{S_2}\theta\}\end{array}}$$

$$\frac{\exists\theta\ (\$C = \$D\theta) \wedge (\mathtt{S_1}\#[\mathtt{establish}:\mathtt{det}(\$A, L, \$D)])}{\begin{array}{c}\{\mathtt{S_0}\ \mathtt{follow}:\mathtt{semidet}(\$A, L, \$C)\ \mathtt{S_1}\ \mathtt{establish}:\mathtt{det}(\$A, L, \$D)\ \mathtt{S_2}\} \longrightarrow \\ \{\mathtt{S_0}\ \mathtt{follow}:\mathtt{semidet}(\$A, L, \$C)\ \mathtt{S_1}\ \mathtt{S_2}\theta\}\end{array}}$$

$$\frac{\exists\theta\ (\$C = \$D\theta) \wedge (\mathtt{S_1}\#[\mathtt{follow}:\mathtt{semidet}(\$A, L, \$D)])}{\begin{array}{c}\{\mathtt{S_0}\ \mathtt{establish}:\mathtt{det}(\$A, L, \$C)\ \mathtt{S_1}\ \mathtt{follow}:\mathtt{semidet}(\$A, L, \$D)\ \mathtt{S_2}\} \longrightarrow \\ \{\mathtt{S_0}\ \mathtt{establish}:\mathtt{det}(\$A, L, \$C)\ \mathtt{S_1}\ \mathtt{S_2}\theta\}\end{array}}$$

$$\frac{\mathtt{S_1}\#[\mathtt{set}:\mathtt{det}(\$A, L, V)]}{\{\mathtt{S_0}\ \mathtt{test}:\mathtt{semidet}(\$A, L, V)\ \mathtt{S_1}\ \mathtt{set}:\mathtt{det}(\$A, L, V)\ \mathtt{S_2}\} \longrightarrow \{\mathtt{S_0}\ \mathtt{test}:\mathtt{semidet}(\$A, L, V)\ \mathtt{S_1}\ \mathtt{S_2}\}}$$

$$\frac{\mathtt{S_1}\#[\mathtt{test}:\mathtt{semidet}(\$A, L, V)]}{\{\mathtt{S_0}\ \mathtt{set}:\mathtt{det}(\$A, L, V)\ \mathtt{S_1}\ \mathtt{test}:\mathtt{semidet}(\$A, L, V)\ \mathtt{S_2}\} \longrightarrow \{\mathtt{S_0}\ \mathtt{set}:\mathtt{det}(\$A, L, V)\ \mathtt{S_1}\ \mathtt{S_2}\}}$$

$$\frac{\mathtt{S_1}\#[\mathtt{minimum}:\mathtt{nondet}(\$A, L, \$C)]}{\{\mathtt{S_0}\ \mathtt{notempty}:\mathtt{nondet}(\$A)\ \mathtt{S_1}\ \mathtt{minimum}:\mathtt{nondet}(\$A, L, \$C)\ \mathtt{S_2}\} \longrightarrow \{\mathtt{S_0}\ \mathtt{minimum}:\mathtt{nondet}(\$A, L, \$C)\ \mathtt{S_1}\ \mathtt{S_2}\}}$$

Figure 5.9: Optimisations to remove repeated SDSL instruction within code blocks where $\mathtt{S_i}$ represents zero or more instructions or nested code blocks, and θ represents a set of variable bindings.

To demonstrate the effect of removing repeated instructions, this optimisation is applied to the example program from Figure 5.4. Figure 5.10 gives the new optimised version of this program. In this case two ***establish*** instructions and a *look* instruction have been removed from the program. The ***establish*** instructions were removed using the 15th optimisation rule whereas the *look* instruction was removed using the 10th optimisation rule from Figure 5.9. This would yield a small performance improvement. In general, applying this optimisation to more complex SDSL programs containing rules with more goals can result in much greater optimisation.

This optimisation is a variant of *local common sub-expression elimination* performed during optimisation of imperative languages, where re-computation of expressions is avoided by using previously computed values [2]. This variation differs from the traditional optimisation because the degree of optimisation depends on the sharing of index nodes or boolean values, not just variables. Generally, when groups of related tuples share index nodes then more code optimisation is possible.

5.9.2 Factorisation of Repeated Prefixes Across Code Blocks

Another optimisation that can be applied to SDSL programs factorises out common sequences of instructions occurring at the beginning of different code blocks. Including these common instructions only once in the program reduces code size and improves performance.

For this optimisation, two code blocks that exist in the same parent code block must begin with the same sequence of instructions. The largest sequence of instructions shared between the code blocks is identified as the *prefix*.

Prefixes do not have to be identical to be considered common. As seen in the previous optimisation, two instructions are equivalent if they always produce the same output. Yet because the scope of variables is restricted to the code block where they were initialised, the set of non-identical but equivalent instructions is reduced in this optimisation.

When factorising two code blocks that are not consecutive, the intermediate code blocks are moved so that they occur after the factorised block. This means that the order code blocks are evaluated changes when non-consecutive blocks are factorised. For general SDSL programs, reordering code blocks may compromise the semantics of the original program.

As with the removal of repeated instructions, to ensure correctness of general SDSL programs we restrict factorising to cases where any code blocks that are moved do not interfere with those that they are migrated over. Cases of interference can be detected by finding an intersection between the set of tuples generated by one code block and the set of tuples accessed by another. The non-interference operator is defined in Figure 5.8.

However, if programs follow the Triggering template, code blocks that can be factorised never interfere with any other blocks. This is because the order of many code blocks appearing in an SDSL triggered program is arbitrary and therefore can be reordered without consequence. The only code blocks that have a necessary order process tuples from each predicate in stratification order. However these are immune to reordering: if two predicate code blocks (A and C) are to be factorised then they will share the first i instructions. Any intermediate code block (B) that exists between A and C, whose order with C is significant

```
| {                                              % Add fact q(0) to delta
|   establish:det($0, delta, $1)
|   insert:det($1, 0, $2)
|   set:det($2, id3, true)
| }
| {
|   follow:semidet($0, delta, $3)                % Find minimum timestamp value
|   minimum:nondet($3 MinVar0:int, $4)
|   {
|     test:semidet($4, id3, true)                % Test if a minimum q/1 tuple exists
|     {
|       insert:det($0, MinVar0, $5)              % Add minimum tuple to gamma
|       set:det($5, id2, true)
|       {                                        % Evaluate Rule 1
|         not{                                   % Test negated goal for early failure
|            scan:nondet($0, Y1:int, $6)
|            test:semidet($6, id2, true)
|            builtin:semidet(greaterThan, [Y1,1], [])
|         }
|         builtin:det(multiply, [MinVar0,2], [Z1:int])
|
|        <Deleted establish:det($0, delta, $7),  $7 = $3>
|
|         insert:det($3, Z1, $8)                 % Add head to delta
|         set:det($8, id1, true)
|       }
|     }
|     set:det($4, id3, false)                    % Delete minimum from delta
|     test:semidet($4, id1, false)
|     delete:det($3, MinVar0)
|   }
|   {
|     test:semidet($4, id1, true)                % Test if a minimum p/1 tuple exists
|     {
|       not{                                     % Evaluate unresolved negated goal late
|          scan:nondet($0, MinVar1:int, $9)
|          test:semidet($9, id2, true)
|          builtin:semidet(greaterThan, [MinVar1,1], [])
|       }
|       insert:det($0, MinVar0, $10)             % Add minimum tuple to gamma
|       set:det($10, id0, true)
|       {
|
|        <Deleted look:semidet($0, MinVar0, $11),  $11 = $10>
|
|         test:semidet($10, id2, true)           % Evaluate Rule 2
|         builtin:semidet(greaterThan, [10,MinVar0], [])
|         builtin:det(add, [MinVar0,1], [Z2:int])
|
|        <Deleted establish:det($0, delta, $12),  $12 = $3>
|
|         insert:det($3, Z2, $13)                % Add head to delta
|         set:det($13, id3, true)
|       }
|     }
|     set:det($4, id1, false)                    % Delete minimum from delta
|     test:semidet($4, id3, false)
|     delete:det($3, MinVar0)
|   }
| }
```

Figure 5.10: Example SDSL program after the removal of repeated instructions.

$$\frac{\exists\theta\ (\mathbf{Prefix_0} \equiv \mathbf{Prefix_1}\theta) \wedge ((\mathbf{S_0} \cup \mathbf{X})\#\mathbf{Prefix_1})}{\{\mathbf{Prefix_0}\ \mathbf{S_0}\}\ \mathbf{X}\ \{\mathbf{Prefix_1}\ \mathbf{S_1}\} \longrightarrow \{\mathbf{Prefix_0}\ \{\mathbf{S_0}\}\ \{\mathbf{S_1}\theta\}\}\ \mathbf{X}}$$

$$\frac{\exists\theta\ (\mathbf{Prefix_0} \equiv \mathbf{Prefix_1}\theta) \wedge ((\mathbf{S_0} \cup \mathbf{X})\#\mathbf{Prefix_1})}{\texttt{not}\{\mathbf{Prefix_0}\ \mathbf{S_0}\}\ \mathbf{X}\ \texttt{not}\{\mathbf{Prefix_1}\ \mathbf{S_1}\} \longrightarrow \texttt{not}\{\mathbf{Prefix_0}\ \texttt{not}\{\texttt{not}\{\mathbf{S_0}\}\ \texttt{not}\{\mathbf{S_1}\theta\}\}\}\ \mathbf{X}}$$

Figure 5.11: Factorisation of identical prefixes of code blocks in an SDSL programs where **X** represents zero or more code blocks of any type.

(C is the code block which would be migrated over B during factorisation), will share the first i instructions with C. If it did not, the order between B and C would be arbitrary since the order would be based on some other argument values rather than the static predicate ordering. If B shares the first i instructions with C, and A shares the first i instructions with C (since A and C were to be factorised), then A, B and C can be factorised together and their suffixes remain in their original order. Because we are only considering triggered programs in this thesis, no further investigation into detecting code block interference has been undertaken.

Figure 5.11 defines this optimisation for code blocks where the variables used in $\mathbf{Prefix_1}$ are replaced by those in $\mathbf{Prefix_0}$. Notice that this optimisation preserves the order of instruction sequences $\mathbf{S_0}$ and $\mathbf{S_1}$. Depending on the types of code blocks involved, this optimisation generates different output. In these rules, **X** represents zero or more intermediate standard or negated code blocks. Factorisation of negated code blocks is based De Morgan's laws. The safety condition restricting reordering of code blocks that interfere with each other is given as $V\#W$ where V and W are sets of code blocks that will be reordered after optimisation. Such safety conditions are redundant when following the Triggering Template.

To determine if code blocks share a common prefix the instructions in each prefix must be equivalent. Instructions that are identical and whose program and index variables match are obviously equivalent because they produce the same output. However, non-identical instructions that produce the same output are also considered equivalent. Figure 5.12 gives factorisation rules for non-identical instructions and the conditions under which they are considered equivalent. For the first factorisation rule, an *establish* instruction is equivalent to a *follow* instruction that appears later in the code block when the two operations are performed on the same index node and same labelled branch. Similarly, a *set* instruction that updates a labelled boolean value to V is equivalent to a *test* instruction that tests the same labelled boolean value for V in a later code block. The third optimisation transforms a *look* instruction into a *scan* and an equality test when the *scan* operation is already performed in another code block. (Note that this optimisation trades a *look* instruction for N equality tests, where N is the number of elements held in the index. If N is sufficiently large then efficiency may deteriorate.) The final three rewrite rules give three equivalent optimisations for factorising negated code block.

The results of applying these optimisations to the example SDSL program in Figure 5.10 is given in Figure 5.13. The optimised program has factorised out an *establish* instruction (by matching it with an equivalent *follow*). In the case of this program this optimisation will have very little effect on the run time performance.

$$\begin{array}{c}\dfrac{\exists\theta\ (\mathbf{Prefix_0} \equiv \mathbf{PrefixB}\theta) \wedge (\$A = \$C\theta) \wedge (\$B = \$D\theta) \wedge ((\mathbf{S_0} \cup \mathbf{X})\#(\mathbf{Prefix_1} \cup \texttt{follow} : \texttt{semidet}(\$C, L, \$D)\})))}{\{\mathbf{Prefix_0}\ \texttt{establish} : \texttt{det}(\$A, L, \$B)\ \mathbf{S_0}\}\ \mathbf{X}\ \{\mathbf{Prefix_1}\ \texttt{follow} : \texttt{semidet}(\$C, L, \$D)\ \mathbf{S_1}\} \longrightarrow} \\ \{\mathbf{Prefix_0}\ \texttt{establish} : \texttt{det}(\$A, L, \$B)\ \{\mathbf{S_0}\}\ \{\mathbf{S_1}\theta\}\}\ \mathbf{X}\end{array}$$

$$\begin{array}{c}\dfrac{\exists\theta\ (\mathbf{Prefix_0} \equiv \mathbf{Prefix_1}\theta) \wedge (\$A = \$C\theta) \wedge ((\mathbf{S_0} \cup \mathbf{X})\#(\mathbf{Prefix_1} \cup \texttt{test} : \texttt{semidet}(\$C, L, V)))}{\{\mathbf{Prefix_0}\ \texttt{set} : \texttt{det}(\$A, L, V)\ \mathbf{S_0}\}\ \mathbf{X}\ \{\mathbf{Prefix_1}\ \texttt{test} : \texttt{semidet}(\$C, L, V)\ \mathbf{S_1}\} \longrightarrow} \\ \{\mathbf{Prefix_0}\ \texttt{set} : \texttt{det}(\$A, L, V)\ \{\mathbf{S_0}\}\ \{\mathbf{S_1}\theta\}\}\ \mathbf{X}\end{array}$$

$$\begin{array}{c}\dfrac{\exists\theta\ (\mathbf{Prefix_0} \equiv \mathbf{Prefix_1}\theta) \wedge (\$A = \$D\theta) \wedge (\$C = \$I\theta) \wedge ((\mathbf{S_0} \cup \mathbf{X})\#(\mathbf{Prefix_1} \cup \texttt{look} : N(\$D, H, \$I)))}{\{\mathbf{Prefix_0}\ \texttt{scan} : \texttt{nondet}(\$A, E : T, \$C)\ \mathbf{S_0}\}\ \mathbf{X}\ \{\mathbf{Prefix_1}\ \texttt{look} : N(\$D, H, \$I)\ \mathbf{S_1}\} \longrightarrow} \\ \{\mathbf{Prefix_0}\ \texttt{scan} : \texttt{nondet}(\$A, E : T, \$C)\ \{\mathbf{S_0}\}\ \{\texttt{builtin} : \texttt{semidet}(\mathit{equals}, [E, H], [])\ \mathbf{S_1}\}\theta\}\ \mathbf{X}\end{array}$$

$$\begin{array}{c}\dfrac{\exists\theta\ (\mathbf{Prefix_0} \equiv \mathbf{PrefixB}\theta) \wedge (\$A = \$C\theta) \wedge (\$B = \$D\theta) \wedge ((\mathbf{S_0} \cup \mathbf{X})\#(\mathbf{Prefix_1} \cup \texttt{follow} : \texttt{semidet}(\$C, L, \$D)))}{\texttt{not}\{\mathbf{Prefix_0}\ \texttt{establish} : \texttt{det}(\$A, L, \$B)\ \mathbf{S_0}\}\ \mathbf{X}\ \texttt{not}\{\mathbf{Prefix_1}\ \texttt{follow} : \texttt{semidet}(\$C, L, \$D)\ \mathbf{S_1}\} \longrightarrow} \\ \texttt{not}\{\mathbf{Prefix_0}\ \texttt{establish} : \texttt{det}(\$A, L, \$B)\ \texttt{not}\{\texttt{not}\{\mathbf{S_0}\}\ \texttt{not}\{\mathbf{S_1}\theta\}\}\}\ \mathbf{X}\end{array}$$

$$\begin{array}{c}\dfrac{\exists\theta\ (\mathbf{Prefix_0} \equiv \mathbf{Prefix_1}\theta) \wedge (\$A = \$C\theta) \wedge ((\mathbf{S_0} \cup \mathbf{X})\#(\mathbf{Prefix_1} \cup \texttt{test} : \texttt{semidet}(\$C, L, V)))}{\texttt{not}\{\mathbf{Prefix_0}\ \texttt{set} : \texttt{det}(\$A, L, V)\ \mathbf{S_0}\}\ \mathbf{X}\ \texttt{not}\{\mathbf{Prefix_1}\ \texttt{test} : \texttt{semidet}(\$C, L, V)\ \mathbf{S_1}\} \longrightarrow} \\ \texttt{not}\{\mathbf{Prefix_0}\ \texttt{set} : \texttt{det}(\$A, L, V)\ \texttt{not}\{\texttt{not}\{\mathbf{S_0}\}\ \texttt{not}\{\mathbf{S_1}\theta\}\}\}\ \mathbf{X}\end{array}$$

$$\begin{array}{c}\dfrac{\exists\theta\ (\mathbf{Prefix_0} \equiv \mathbf{Prefix_1}\theta) \wedge (\$A = \$D\theta) \wedge (\$C = \$I\theta) \wedge ((\mathbf{S_0} \cup \mathbf{X})\#(\mathbf{Prefix_1} \cup \texttt{look} : N(\$D, H, \$I)))}{\texttt{not}\{\mathbf{Prefix_0}\ \texttt{scan} : \texttt{nondet}(\$A, E : T, \$C)\ \mathbf{S_0}\}\ \mathbf{X}\ \texttt{not}\{\mathbf{Prefix_1}\ \texttt{look} : N(\$D, H, \$I)\ \mathbf{S_1}\} \longrightarrow} \\ \texttt{not}\{\mathbf{Prefix_0}\ \texttt{scan} : \texttt{nondet}(\$A, E : T, \$C)\ \texttt{not}\{\texttt{not}\{\mathbf{S_0}\}\ \texttt{not}\{\texttt{builtin} : \texttt{semidet}(\mathit{equals}, [E, H], [])\ \mathbf{S_1}\}\theta\}\}\ \mathbf{X}\end{array}$$

Figure 5.12: Factorisation of common prefixes for non-identical instructions where **X** represents zero or more code blocks of any type.

Factorisation of common prefixes in SDSL is similar to the construction of code trees which have been used to implement bottom-up systems. However, unlike SDSL programs which are statically defined at run time, code trees are dynamically changing programs which represent "the dynamically changing databases of clauses" [110].

In other ways the factorising optimisation is similar to the *if-optimisation* used when compiling imperative programming languages. The if-optimisation combines the bodies of two or more *if* statements if their truth conditions are identical. Performance improves because only one *if* test is performed. Common prefixes in SDSL code blocks are equivalent to repeated *if* statements because the code immediately following any SDSL prefix is executed only when all operations in the prefix are successful.

5.9.3 Deletion Completion

A common artifact of Triggering Evaluation is the insistence that deletions of tuples in Δ occur at the end of each predicate's code block. Yet it is usually unnecessary to delete tuples individually. Optimisation occurs by combining the deletion code after the *complete* set of predicates have used the common arguments in trigger tuples[1]. After an argument binding has been used by all predicates it is deleted.

The combining of deletion code frequently causes auxiliary code to become redundant. For example, if a delete operation is performed on a value in an index node and then the index node itself is entirely deleted, the first delete operation is unnecessary. Instead, by deleting all references to the index node

[1]Because this optimisation assumes that a minimum argument value is only used once by all predicate code blocks, this optimisation is unsafe for programs that are not strongly stratified, and programs which share indexes between Δ and Γ.

```
{                                              % Add fact q(0) to delta

  establish:det($0, delta, $1)                 % Factorised prefix

  {
    insert:det($1, 0, $2)
    set:det($2, id3, true)
  }
  {
    minimum:nondet($1 MinVar0:int, $4)         % Find minimum timestamp value
    {
      test:semidet($4, id3, true)              % Test if a minimum q/1 tuple exists
      {
        insert:det($0, MinVar0, $5)            % Add minimum tuple to gamma
        set:det($5, id2, true)
        {                                      % Evaluate Rule 1
          not{                                 % Test negated goal for early failure
             scan:nondet($0, Y1:int, $6)
             test:semidet($6, id2, true)
             builtin:semidet(greaterThan, [Y1,1], [])
          }
          builtin:det(multiply, [MinVar0,2], [Z1:int])
          insert:det($1, Z1, $8)               % Add head to delta
          set:det($8, id1, true)
        }
      }
      set:det($4, id3, false)                  % Delete minimum from delta
      test:semidet($4, id1, false)
      delete:det($1, MinVar0)
    }
    {
      test:semidet($4, id1, true)              % Test if a minimum p/1 tuple exists
      {
        not{                                   % Evaluate unresolved negated goal late
           scan:nondet($0, MinVar1:int, $9)
           test:semidet($9, id2, true)
           builtin:semidet(greaterThan, [MinVar1,1], [])
        }
        insert:det($0, MinVar0, $10)           % Add minimum tuple to gamma
        set:det($10, id0, true)
        {
          test:semidet($10, id2, true)         % Evaluate Rule 2
          builtin:semidet(greaterThan, [10,MinVar0], [])
          builtin:det(add, [MinVar0,1], [Z2:int])
          insert:det($1, Z2, $13)              % Add head to delta
          set:det($13, id3, true)
        }
      }
      set:det($4, id1, false)                  % Delete minimum from delta
      test:semidet($4, id3, false)
      delete:det($1, MinVar0)
    }
  }
}
```

Figure 5.13: Example SDSL program after common prefix factorisation.

$$\{\texttt{minimum} : \texttt{nondet}(\$A, B, \$C)\ \mathbf{S_0}\ (\forall i)\{\mathbf{Pred_i}\ \texttt{delete} : \texttt{det}(\$A, B)\}\} \longrightarrow$$
$$\{\texttt{minimum} : \texttt{nondet}(\$A, B, \$C)\ \{\mathbf{S_0}\ (\forall i)\{\mathbf{Pred_i}\}\}\ \texttt{delete} : \texttt{det}(\$A, B)\}$$

$$\{\texttt{minimum} : \texttt{nondet}(\$A, B, \$C)\ \{\mathbf{S_0}\ (\forall i)\{\mathbf{Pred_i}\ \texttt{delete} : \texttt{det}(\$C, D)\}\ \mathbf{S_1}\}\ \texttt{delete} : \texttt{det}(\$A, B)\} \longrightarrow$$
$$\{\texttt{minimum} : \texttt{nondet}(\$A, B, \$C)\ \{\mathbf{S_0}\ (\forall i)\{\mathbf{Pred_i}\}\ \mathbf{S_1}\}\ \texttt{delete} : \texttt{det}(\$A, B)\}$$

$$\{\texttt{minimum} : \texttt{nondet}(\$A, B, \$C)\ \{\mathbf{S_0}\ (\forall i)\{\mathbf{Pred_i}\ \texttt{prune} : \texttt{det}(\$C, D)\}\ \mathbf{S_1}\}\ \texttt{delete} : \texttt{det}(\$A, B)\} \longrightarrow$$
$$\{\texttt{minimum} : \texttt{nondet}(\$A, B, \$C)\ \{\mathbf{S_0}\ (\forall i)\{\mathbf{Pred_i}\}\ \mathbf{S_1}\}\ \texttt{delete} : \texttt{det}(\$A, B)\}$$

$$\{\texttt{minimum} : \texttt{nondet}(\$A, B, \$C)\ \{\mathbf{S_0}\ (\forall i)\{\mathbf{Pred_i}\ \texttt{set} : \texttt{det}(\$C, D, E)\}\ \mathbf{S_1}\}\ \texttt{delete} : \texttt{det}(\$A, B)\} \longrightarrow$$
$$\{\texttt{minimum} : \texttt{nondet}(\$A, B, \$C)\ \{\mathbf{S_0}\ (\forall i)\{\mathbf{Pred_i}\}\ \mathbf{S_1}\}\ \texttt{delete} : \texttt{det}(\$A, B)\}$$

$$\{\texttt{minimum} : \texttt{nondet}(\$A, B, \$C)\ \{\mathbf{S_0}\ (\forall i)\{\mathbf{Pred_i}\ \texttt{empty} : \texttt{semidet}(\$C)\}\ \mathbf{S_1}\}\ \texttt{delete} : \texttt{det}(\$A, B)\} \longrightarrow$$
$$\{\texttt{minimum} : \texttt{nondet}(\$A, B, \$C)\ \{\mathbf{S_0}\ (\forall i)\{\mathbf{Pred_i}\}\ \mathbf{S_1}\}\ \texttt{delete} : \texttt{det}(\$A, B)\}$$

$$\{\texttt{minimum} : \texttt{nondet}(\$A, B, \$C)\ \{\mathbf{S_0}\ (\forall i)\{\mathbf{Pred_i}\ \texttt{ispruned} : \texttt{semidet}(\$C, D)\}\ \mathbf{S_1}\}\ \texttt{delete} : \texttt{det}(\$A, B)\} \longrightarrow$$
$$\{\texttt{minimum} : \texttt{nondet}(\$A, B, \$C)\ \{\mathbf{S_0}\ (\forall i)\{\mathbf{Pred_i}\}\ \mathbf{S_1}\}\ \texttt{delete} : \texttt{det}(\$A, B)\}$$

$$\{\texttt{minimum} : \texttt{nondet}(\$A, B, \$C)\ \{\mathbf{S_0}\ (\forall i)\{\mathbf{Pred_i}\ \texttt{test} : \texttt{semidet}(\$C, D, E)\}\ \mathbf{S_1}\}\ \texttt{delete} : \texttt{det}(\$A, B)\} \longrightarrow$$
$$\{\texttt{minimum} : \texttt{nondet}(\$A, B, \$C)\ \{\mathbf{S_0}\ (\forall i)\{\mathbf{Pred_i}\}\ \mathbf{S_1}\}\ \texttt{delete} : \texttt{det}(\$A, B)\}$$

Figure 5.14: Deletion Completion optimsation.

will make it a candidate for the automatic garbage collection processes of the target environment. (Note that this form of optimisation would not be applicable in environments without automatic garbage collection.) Redundant instructions that delete tuples by removing argument values from sub-indexes, setting labelled boolean variables to false or by pruning labelled branches are unnecessary given that their parent index node is scheduled for deletion (before the program accesses their values again). Likewise, any additional tests to check if an index node is redundant before deleting it are unnecessary after this optimisation. This is because after all predicate code blocks have used an argument value from Δ which determines the stratification order of tuples, this value is redundant and should be deleted so that a new minimum value is selected in the next iteration.

Figure 5.14 gives the definition of this optimisation as rewrite rules. In this definition variables $\mathbf{S_0}$, $\mathbf{S_1}$ and $\mathbf{Pred_i}$ represent any number of instructions or nested code blocks. The first rewrite rule replaces all identical *delete* instructions appearing at the end of all predicate code blocks with a single delete instruction performed after all predicates have been processed. After this optimisation, any instructions associated with the deletion of a tuple in a predicate code block are redundant and are removed by the remaining six rewrite rules. For example, the second rule removes all *delete* operations that are performed on the index node $\$C$ when this index node itself is subsequently removed by a *delete* instruction performed on the index node $\$A$ (where $\$A$ is the parent of $\$C$). Similarly, the third rule removes all *prune* instructions that are performed on an index node that is deleted. If tuples are stratified on more than one argument then this optimisation can be repeatedly applied.

Deletion Completion has been applied to the example program from Figure 5.13 and the new optimised program is given in Figure 5.15. In this case the six SDSL instructions that remove tuples one argument at a time from the Δ set have been replaced by one delete instruction performed after all the predicate's code blocks.

5.9.4 Removal of Redundant Code Block Nesting

When an SDSL program has been automatically generated it may encapsulate instructions in two or more nested code blocks where one would be sufficient. The addition of extra code blocks not only reduces run time performance of programs but obfuscates cases where prefix factorisation is possible.

Rules used to remove unnecessary code block nesting are given in Figure 5.16. In each rule, $\mathbf{T_0}$ and $\mathbf{T_1}$ represent a sequence of at least one instruction or code block. Applying the first rewrite rule from Figure 5.16 removes double nesting of code blocks. This clarifies cases where factorisation is possible and so can be applied before factorisation for increased optimisation.

The second rule removes any extra unnecessary code block nesting at the end of a parent code block to reduce run time overhead. This optimisation is valid because all operations at the end of a code block are exhaustively performed (before backtracking to the previous operations) whether they are inside a code block or not. Because this transformation extracts instructions from code blocks it may prevent the factorising of code blocks. For this reason the second rule is best applied after factorisation has occurred. (More on the ordering of optimisations later.)

The last three rules perform general code clean up of cases where empty code blocks are created. When an empty standard code block ({}) occurs in a program this can be deleted outright since it will always succeed. However the occurrence of an empty negated code block is equivalent to a **fail** statement in Prolog where the negated code block can never succeed because none of the internal instructions can fail. The final rewrite rule generalises this optimisation for programs where instructions follow an empty negated code block. Because the empty negated code block will always fail, any instructions that occur after the block ($\mathbf{T_0}$) will never be executed. Therefore removing these instructions is correct and will reduce code size. Figure 5.17 shows the example program from Figure 5.15 after all redundant code block nestings have been removed.

Note that no rule exists to remove extra nesting at beginning of a code block (i.e. code blocks of the form $\{\{\mathbf{T_0}\}\mathbf{T_1}\}$). This is because these code blocks specify that all operations in the nested code block must be exhaustively performed via backtracking before any of the operations that follow are performed. By removing nesting from the beginning of a code block the order of operations is changed and the program's correctness may be compromised.

The first rewrite rule in Figure 5.16 is similar to *loop collapsing* performed when compiling imperative languages where a single loop replaces multiple nested loops that have the same escape conditions. The second rewrite rule has properties of a *tail recursion optimisation* because the last nested code block in a sequence is now immediately evaluated rather than entering and accumulating extra overhead from a new, nested code block. The remaining three rewrite rules exhibit behavior equivalent to *dead code removal* where code blocks and instructions that will never have an effect on the state of the program are removed.

```
| {                                              % Add fact q(0) to delta
|   establish:det($0, delta, $1)                 % Factorised prefix
|   {
|     insert:det($1, 0, $2)
|     set:det($2, id3, true)
|   }
|   {
|     minimum:nondet($1 MinVar0:int, $4)         % Find minimum timestamp value
|     {
|       {
|         test:semidet($4, id3, true)              % Test if a minimum q/1 tuple exists
|         {
|           insert:det($0, MinVar0, $5)            % Add minimum tuple to gamma
|           set:det($5, id2, true)
|           {                                      % Evaluate Rule 1
|             not{                                 % Test negated goal for early failure
|                scan:nondet($0, Y1:int, $6)
|                test:semidet($6, id2, true)
|                builtin:semidet(greaterThan, [Y1,1], [])
|             }
|             builtin:det(multiply, [MinVar0,2], [Z1:int])
|             insert:det($1, Z1, $8)               % Add head to delta
|             set:det($8, id1, true)
|           }
|         }
|       }
|       {
|         test:semidet($4, id1, true)              % Test if a minimum p/1 tuple exists
|         {
|           not{                                   % Evaluate unresolved negated goal late
|              scan:nondet($0, MinVar1:int, $9)
|              test:semidet($9, id2, true)
|              builtin:semidet(greaterThan, [MinVar1,1], [])
|           }
|           insert:det($0, MinVar0, $10)           % Add minimum tuple to gamma
|           set:det($10, id0, true)
|           {
|             test:semidet($10, id2, true)         % Evaluate Rule 2
|             builtin:semidet(greaterThan, [10,MinVar0], [])
|             builtin:det(add, [MinVar0,1], [Z2:int])
|             insert:det($1, Z2, $13)              % Add head to delta
|             set:det($13, id3, true)
|           }
|         }
|       }
|     }

|     delete:det($1, MinVar0)                    % Combined Delete instruction

|   }
| }
```

Figure 5.15: Example SDSL program after Deletion Completion.

$$\{\{T_0\}\} \longrightarrow \{T_0\}$$

$$\{T_0\ \{T_1\}\} \longrightarrow \{T_0\ T_1\}$$

$$\{\} \longrightarrow < \textit{empty} >$$

$$\texttt{not}\{\} \longrightarrow \texttt{builtin:semidet(fail,[],[])}$$

$$\texttt{not}\{\}\ T_0 \longrightarrow \texttt{builtin:semidet(fail,[],[])}$$

Figure 5.16: Rules to remove redundant code block nesting where $\mathbf{T_0}$ and $\mathbf{T_1}$ are sequences of at least one instruction and/or code blocks.

```
| {                                                % Add fact q(0) to delta
|   establish:det($0, delta, $1)                   % Factorised prefix
|   {
|     insert:det($1, 0, $2)
|     set:det($2, id3, true)
|   }
|   minimum:nondet($1 MinVar0:int, $4)             % Find minimum timestamp value
|   {
|     {
|       test:semidet($4, id3, true)                % Test if a minimum q/1 tuple exists
|       insert:det($0, MinVar0, $5)                % Add minimum tuple to gamma
|       set:det($5, id2, true)
|       not{                                       % Test negated goal for early failure
|          scan:nondet($0, Y1:int, $6)
|          test:semidet($6, id2, true)
|          builtin:semidet(greaterThan, [Y1,1], [])
|       }
|       builtin:det(multiply, [MinVar0,2], [Z1:int])
|       insert:det($1, Z1, $8)                     % Add head to delta
|       set:det($8, id1, true)
|     }
|     test:semidet($4, id1, true)                  % Test if a minimum p/1 tuple exists
|     not{                                         % Evaluate unresolved negated goal late
|        scan:nondet($0, MinVar1:int, $9)
|        test:semidet($9, id2, true)
|        builtin:semidet(greaterThan, [MinVar1,1], [])
|     }
|     insert:det($0, MinVar0, $10)                 % Add minimum tuple to gamma
|     set:det($10, id0, true)
|     test:semidet($10, id2, true)                 % Evaluate Rule 2
|     builtin:semidet(greaterThan, [10,MinVar0], [])
|     builtin:det(add, [MinVar0,1], [Z2:int])
|     insert:det($1, Z2, $13)                      % Add head to delta
|     set:det($13, id3, true)
|   }
|   delete:det($1, MinVar0)}                       % Combined Delete instruction
| }
```

Figure 5.17: Example SDSL program after redundant code nesting has been removed.

$$\begin{aligned}\{\mathtt{A}:\mathtt{det}(B,C,D)\ \ \mathbf{S_0}\} &\longrightarrow \mathtt{A}:\mathtt{det}(B,C,D)\ \{\mathbf{S_0}\}\\ \mathtt{not}\{\mathtt{A}:\mathtt{det}(B,C,D)\ \ \mathbf{S_0}\} &\longrightarrow \mathtt{A}:\mathtt{det}(B,C,D)\ \mathtt{not}\{\mathbf{S_0}\}\end{aligned}$$

Figure 5.18: Extraction of deterministic instructions from the beginning of code blocks.

5.9.5 Extracting Deterministic Instructions from Code Blocks

Programs are sometimes optimised when deterministic instructions are removed from the beginning of code blocks. Because deterministic instructions that exist at the beginning of a code block do not recompute their outputs when encountered during backtracking, these can be safely moved from inside the code block to immediately before the code block. One advantage of moving deterministic instructions out of a code block is that the scope of the instruction's output is increased which may enable further optimisation of repeated instruction. Another advantage is if all instructions in a code block are deterministic then the presence of the code block is redundant and can prompt removal of redundant code block nesting. This optimisation is particularly useful for optimising negated code blocks, or for extracting all instructions that initialise facts inside a code block. The definition of this optimisation is given in Figure 5.18. A SDSL instruction is deterministic iff its determinism property is set to `det`.

Note that this optimisation does not remove arbitrary deterministic instructions from code blocks – only those at the beginning of a code block. This is because dependencies can exist between the variable bindings used by deterministic instructions in the middle of a code block and those used in previous instructions, making reordering instructions unsafe. To perform a more sophisticated optimisation that detects when instruction reordering is safe would require building a dependency graph for all variables in each code block. Although such an optimisation is reasonably simple to understand, its implementation is complicated and so it has not been explored further.

The results of this optimisation (together with all other optimisations) can be seen Figure 5.19. In this example the instructions used to add the `q(0)` fact to Δ have been removed from their code block. (The empty code block has been removed using the previous optimisation.)

This optimisation is a limited form of the *hoisting* optimisation that moves invariant operations out of loops in imperative programs.

5.9.6 Order of Optimisation

With the exception of redundant nested code block removal, the order that optimisations should be applied has been ignored. In many cases the order does not matter as the optimisations are commutative and will produce the same result no matter which order they are applied. (This is the case for redundant instruction removal and prefix factorisation.) But to ensure maximal optimisation for non-commutative optimisations the best approach is to repeatedly apply all optimisations in a predetermined order until no further transformations are possible (i.e. the program definition reaches a fixed-point). Because no combination of optimisations are complementary, optimisation will never enter an infinite loop and a fixed-point will be found. This is true because all optimi-

```
| {                                                 % Add fact q(0) to delta
|   establish:det($0, delta, $1)                    % Factorised prefix
|   insert:det($1, 0, $2)
|   set:det($2, id3, true)
|   minimum:nondet($1 MinVar0:int, $4)              % Find minimum timestamp value
|   {
|     {
|       test:semidet($4, id3, true)                 % Test if a minimum q/1 tuple exists
|       insert:det($0, MinVar0, $5)                 % Add minimum tuple to gamma
|       set:det($5, id2, true)
|       not{                                        % Test negated goal for early failure
|          scan:nondet($0, Y1:int, $6)
|          test:semidet($6, id2, true)
|          builtin:semidet(greaterThan, [Y1,1], [])
|       }
|       builtin:det(multiply, [MinVar0,2], [Z1:int])
|       insert:det($1, Z1, $8)                      % Add head to delta
|       set:det($8, id1, true)
|     }
|     test:semidet($4, id1, true)                   % Test if a minimum p/1 tuple exists
|     not{                                          % Evaluate unresolved negated goal late
|        scan:nondet($0, MinVar1:int, $9)
|        test:semidet($9, id2, true)
|        builtin:semidet(greaterThan, [MinVar1,1], [])
|     }
|     insert:det($0, MinVar0, $10)                  % Add minimum tuple to gamma
|     set:det($10, id0, true)
|     test:semidet($10, id2, true)                  % Evaluate Rule 2
|     builtin:semidet(greaterThan, [10,MinVar0], [])
|     builtin:det(add, [MinVar0,1], [Z2:int])
|     insert:det($1, Z2, $13)                       % Add head to delta
|     set:det($13, id3, true)
|   }
|   delete:det($1, MinVar0)}                        % Combined Delete instruction
| }
```

Figure 5.19: Example program after all optimisations have been applied.

sations can only reduce code size or replace expensive instructions with simple ones, so no cycles are possible.

By carefully ordering the optimisations the number of iterations performed to find the fixed-point can be minimised. Figure 5.20 gives one order that the optimisations may be applied which, in practice, quickly produces significantly optimised code for SDSL programs. The number of iterations performed before the program code reaches a fixed-point will depend on the degree and types of optimisation possible in the SDSL program. Therefore there are many other orders apart from Figure 5.20 in which may be equally or more efficient.

The result of applying all optimisations repeatedly to the example program in Figure 5.4 can be seen in Figure 5.19. By applying the optimisations in the order specified in Figure 5.20 three top-level iterations are required for the program to reach a fully optimised state. The original program was 54 lines long comprising of 34 SDSL instructions. The optimised program is only 35 lines long with 25 SDSL instructions. In practice, executing these two programs (using the same data structures for each argument index) reveals the optimised program to be 8% faster than the unoptimised variant. Although an improvement, the change in execution speed will not usually be proportional to the code size reduction. The two major reasons for this is that some instructions are performed more than others (due to differing numbers of non-deterministic alternatives), and some instructions are more computationally expensive than others. In this case both programs are occupied for a disproportional amount of time constructing the same data structures in memory.

Table 5.1 demonstrates the effect of the code optimisations on code size and run time performance on a set of programs. Details of each example program are included in Appendix C (where these and other programs are used to demonstrate the various data structure selection techniques given in Chapter 7). Note that optimal data structures are selected for each argument index to maximise the effect of the code optimisations. In some cases the example data sets used in these experiments differ from those described in Appendix C so that the code size is not overwhelmed by the number of facts. Table 5.1 shows that the application of the optimisations described in this chapter reduces code size and run times for all these example programs. However the degree of optimisation varies greatly as some programs share more index nodes between different sets of tuples or contain rules with greater redundancy than others – in some cases run times improve by less than 1% however other programs result are more than 20% faster.

Although the SDSL code optimisations do reduce run times, these results are still disappointing considering the amount of work required to perform each optimisation. It is believed that the disappointing results are due to the high degree of optimisation already achieved in the example Starlog programs. This due to the programmer's extensive experience and knowledge of how Starlog programs are compiled influencing their design. For example, these programs do not include redundant predicates and impose strict stratification orders to maximise efficiency. Consequently these Starlog programs may not be considered typical. It is believed that programs produced by less experienced programmers will experience far greater optimisation using the techniques given in this chapter.

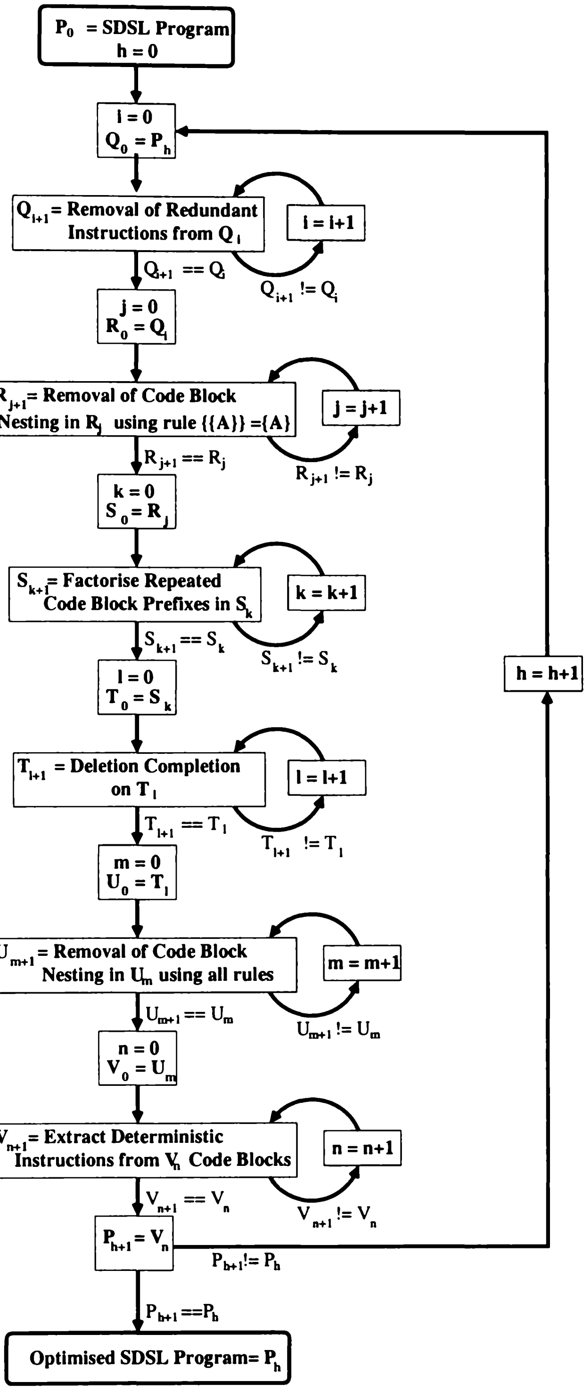


Figure 5.20: Flow diagram showing one order that optimisations may be applied.

Program	Applicable Optimisations	Unoptimised Code Size	Unoptimised Run Time	Optimised Code Size	Optimised Run Time
Hamming	Removal of Code Block Nesting Extracting Det Instructions	25 lines	16 ms	21 lines	13 ms
Primes	Redundant instruction Removal Factorisation of Code Blocks Deletion Completion	68 lines	12.3 sec	53 lines	12.1 sec
Shortest Path	Factorisation of Code Blocks Removal of Code Block Nesting	74 lines	31 ms	69 lines	28 ms
Pacal's Triangle	Redundant instruction Removal Factorisation of Code Blocks Removal of Code Block Nesting	70 lines	44 ms	66 lines	41 ms
Transitive Closure	Redundant instruction Removal Factorisation of Code Blocks Removal of Code Block Nesting	134 lines	496 ms	120 lines	492 ms
Game of Life	Redundant Instruction Removal Factorisation of Code Blocks Deletion Completion Removal of Code Block Nesting	250 lines	766 ms	173 lines	662 ms
N-Queens	Redundant Instruction Removal Factorisation of Code Blocks Deletion Completion Removal of Code Block Nesting	381 lines	245 ms	299 lines	193 ms

Table 5.1: Performance comparisons for optimised and unoptimised SDSL programs.

5.10 Conclusions

In this chapter we have shown that triggered programs which use Starlog index structures can be represented in SDSL. The SDSL programs are efficient because they allow some matching of terms to be performed at compile time, and because specialised code is produced to evaluate each rule for each trigger goal. These properties are a consequence of the static definition of the index structures.

After a triggered SDSL program has been defined we can apply a number of optimisations. These reduce the number of redundant or repeated instructions, or replace expensive instuctions with simple ones. Although these optimisations are given with reference to triggered programs, all the optimisations are applicable to general SDSL programs which may implement different evaluation techniques. The optimisations presented here (1) remove redundant operations which occur in the same code block, (2) combine common prefixes to factorise instructions performed in multiple code blocks, (3) remove redundant instructions associated with index nodes that will be deleted, (4) remove redundant code block nesting and (5) extract deterministic instructions from the beginnings of code blocks. These are the only opportunities for optimisations that are currently apparent for SDSL programs. However it is possible that more may be discovered in future work. Using an example SDSL program the effect of each optimisation is shown on the program's source code. It has also been shown that run times are reduced as a result of these optimisations.

Although the optimisations presented here are restricted to SDSL programs they would be applicable to other langauges with similar semantics or control facilities. For example, the non-interference condition upheld when factorising common prefixes is valid for the *if-optimisation* in imperative languages. Therefore, by better understanding the interference conditions if-optimisations may be more widely used. Another interesting application of these optimisation may improve the performance of Prolog programs. Optimisations on code blocks (such as prefix factoring) are applicable to Prolog since the semantics of a code block is equivalent to Prolog's failure-driven-loop. The same is true of optimisations performed on negated code blocks since their semantics are equivalent the negation-as-failure operator. The effectiveness of these optimisations in other contexts is yet to be determined.

The next stage of compilation is to translate SDSL programs into an executable form. In the next chapter we discuss Java implementations of SDSL programs.

Chapter 6

Compilation of SDSL to Java

The Starlog Data Structure Language (SDSL) can be used as an intermediate representation of bottom-up programs, allowing high-level analysis and optimisation. However, to run programs, SDSL must be compiled to an executable form. In this chapter the process of compiling SDSL programs to Java is discussed. Figure 6.1 shows this stage in Starlog's compilation pipeline.

In the context of the Mercury compiler, [58] argues that compilers of logic languages should target high-level, imperative programming languages (in their case, they choose C). This is because such languages are high-level enough to make code generation easy but low-level enough to express low-level optimisations. In this thesis the chosen target language for the Starlog compiler is Java. Java provides a comfortable platform to reach from the SDSL intermediate form. Although Java is procedural, the addition of object orientation and garbage collection are useful facilities. For example, object orientation allows all index nodes to have a consistent interface for indexing argument values, yet remain expandable for the more specific labelled branches and boolean values. Java's platform independence is also an attractive feature for propagating Starlog. The current popularity of Java is also an important factor for this project because other researchers will be expected to modify and improve the compiler in the future. By using a language which is already being taught to new many researchers reduces the obstacles that these researchers face when contributing

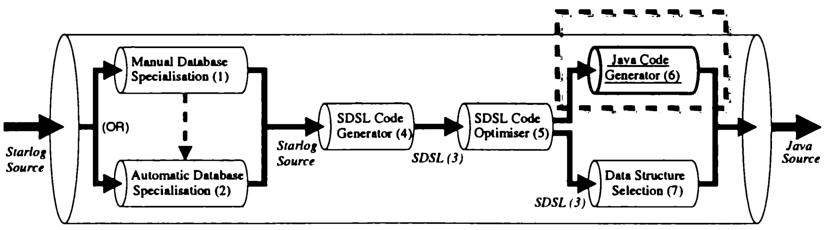


Figure 6.1: Java code generation in Starlog's compilation pipeline.

to the Starlog compiler.

Although using Java as a target simplifies many aspects of the Starlog compiler, Java programs are often criticised for lacking the run time efficiency of other languages. Consequently, compiled Starlog programs may not be as efficient as if another language were targeted (e.g. C). However Starlog programs compiled to Java will still be a massive improvement over the Starlog interpreters previously used. Therefore the use of Java, although probably not the optimal target language, is considered sufficiently efficient.

Yet Java compiler technology is currently evolving and improving. This means that Starlog programs will benefit from any efficiency improvements developed for Java. On the other hand, some libraries and methods are deprecated in each revision and will not be supported in later versions of the language. For this reason we are careful to use only the core Java library (`java.lang`) which is unlikely to change.

Care has been taken to provide efficient Java implementations of SDSL instructions. Priority has been given to reducing the number of objects generated by the Java programs since our early experiments showed that their creation is expensive in both initialisation time and memory (a fact acknowledged in [38] and [103]). To improve performance, primitive data types are used instead of objects where possible, in spite of complicating many of the instruction translations. Because casting objects is another source of inefficiency the Java code avoids this where possible.

Of course Java is not the only suitable target language to which SDSL programs can be compiled. This guide to compilation would be relevant when compiling SDSL to most modern procedural languages such as C, C++, Pascal or Smalltalk. The use of classes is necessary only for implementing the index nodes of the index structure that extend existing data structures. Without automatic garbage collection, memory locations need to be explicitly deleted when they become obsolete.

This chapter initially discusses the implementation of index structures in Java, provides general discussions about index and program variables, and then gives the Java code translations of code blocks and SDSL instructions. Finally, to demonstrate the complete compilation process, the Java implementation is given of the fully optimised program generated in the previous chapter (Figure 5.19).

6.1 Implementation of Index Structures

Each node of an index structure is represented in Java by an object. To allow customisation of nodes, each type of node that can appear in an index structure is represented by a different class. Each class representing a node in the index structure is based on the `Node` interface. This interface, shown in Figure 6.2[1], dictates methods that all index nodes must implement. Classes which implement the Node interface include `look`, `insert` and `delete` methods for all the primitive types required by Starlog programs (currently ints, strings and doubles). In addition, methods exist to find the sub-index associated with the

[1]The interface shown in Figure 6.2 is a simplification of the actual interface used. The actual interface includes additional method names to allow experimentation with garbage collection.

```
public interface Node{
    Node look(int N);
    Node look(String N);
    Node look(double N);
    Node insert(int N, NodeFactory nf);
    Node insert(String N, NodeFactory nf);
    Node insert(double N, NodeFactory nf);
    boolean delete(int N);
    boolean delete(String N);
    boolean delete(double N);
    Node minimum();
    int minimumValue_int();
    String minimumValue_String();
    double minimumValue_double();
    NodeIterator getIterator();
    boolean isEmpty();
}
```

Figure 6.2: The Node interface.

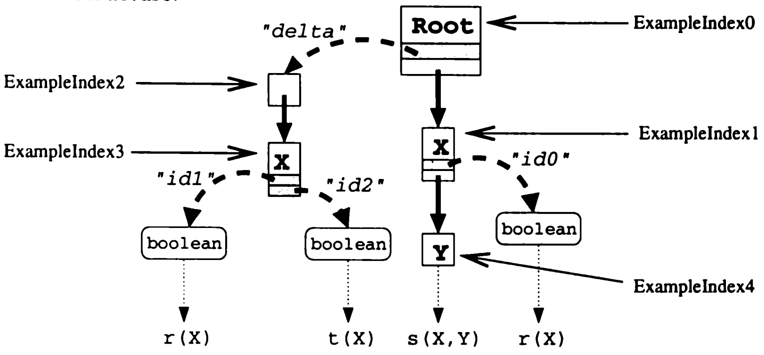
minimum value of an argument, and methods to return the minimum value of the expected type. These are necessary when finding a minimal element in the Δ set. To allow nested *scan* operations on an argument index the `getIterator` method produces a new `NodeIterator` object (introduced later in Figure 6.4). Finally, the `isEmpty` method tests if the argument index contains no values. Although the operation of some of these methods is transparent, all will be described in greater detail as necessary.

To store argument values in each index node, a data structure class implements the methods of the `Node` interface. These implementations differ depending on their design and purpose. For example, a balanced binary tree allows efficient searching in large quantities of data whereas an unsorted list is more efficient for very small data sets. The details of some data structures are described in Chapter 7 with various approaches to their selection. For now it is sufficient to assume that argument values are available through the methods in the `Node` interface.

To allow specialisation of index nodes, each index node extends the data structure class in which its argument values are stored. Statically labelled branches and labelled boolean values are implemented as fields (or member variables) within these index node objects. The static labels of these become the variable names. Naturally, labelled boolean values become boolean primitives. Labelled branches take on the types of other index node classes, all derived from the `Node` interface.

Figure 6.3 demonstrates the implementation of index nodes in Java. The names of each class are derived from the SDSL program's file name concatenated with a unique identifier. The data structures used in this example are balanced binary trees for all indexes that hold argument values, and special empty nodes for those that do not. (Again, the selection of these data structures is discussed in Chapter 7.)

Index Structure:



Index Path Definitions:

```
index gamma r(X)   [X, boolean(id0)]
index delta r(X)   [branch(delta), X, boolean(id1)]
index gamma s(X,Y) [X, Y]
index delta t(X)   [branch(delta), X, boolean(id2)]
```

Java Classes:

```
class ExampleIndex0 extends BalancedBinaryTree{
  public ExampleIndex2 delta;
}

class ExampleIndex1 extends BalancedBinaryTree{
  public boolean id0;
}

class ExampleIndex2 extends BalancedBinaryTree{
}

class ExampleIndex3 extends EmptyNode{
  public boolean id1;
  public boolean id2;
}

class ExampleIndex4 extends EmptyNode{
}
```

Figure 6.3: Example index structure and Java implementation.

6.2 Index Variables

Index variables are renamed from their SDSL representation to a "Java friendly" syntax. The '$' that prefixes SDSL index variables is replaced with the identifier "`node`". For example the SDSL variables $0, $1 and $2 become `node0`, `node1` and `node2`, respectively.

With the exception of the the root (`node0`), all index variables are generated as the results of SDSL operations. All new index variables used in the Java program belong to the `Node` interface. Declaring index variables as `Node` (e.g. `Node node1 = ...`) avoids specifying an index variable's derived class when it is initialised, and so reduces the amount of object casting.

In logic programming, variables can only be bound to a single value unless backtracking removes the binding. To ensure index variables are not reassigned after they have been initialised, index variables are declared `final` when assigned a value.

6.3 Program Variables

The names of program variables used in an SDSL program remain unchanged in the equivalent Java program. The convention that Java variable names always begin with a lowercase letter conflicts with the variable naming requirements of Starlog, SDSL and most other logic programming language. However since the Java code produced by the Starlog compiler is not intended for human interpretation, Java's variable naming conventions are not enforced. However, when displaying Java code in this thesis, variable names will conform to the Java variable naming conventions to aid interpretation. The original Starlog variable names can be reproduced by converting the first character of a program variable to uppercase.

For efficiency, program variables are typed. In an SDSL program, the first occurrence of a program variable includes type information (that is either inferred from or declared in the original Starlog program). This type information is used during Java's declaration of program variables such that a variable `X` :<*type*> in an SDSL program becomes <*type*> `X` in Java.

As with index variables, program variables are declared `final` to enforce a single assignment.

6.4 Code Blocks

A code block controls the flow of an SDSL program and has an effect on the scope of variables. In Java, each code block is implemented as a compound statement block (enclosed in braces { `<code>` }) that will execute once. Any variables declared within the block are not accessible outside. Code blocks are also used to enforce exhaustive backtracking. The Java implementation of this this backtracking is performed during the translation of non-deterministic instructions and so is discussed in the following sections.

The translation of a code block from SDSL to Java is given as follows.

SDSL:
{ S }

$\downarrow$

Java:
```
{
    <translation of S>
}
```

6.5 Negated Code Blocks

The Java implementations of negated code blocks are more complex. A negated goal must exhaustively execute all internal instructions. If the conjunction of instructions always fails the negated goal succeeds and the instructions following the negated goal are executed. Otherwise, if there exists a solution to the negated goal such that all internal operations succeed, the program must backtrack to the last non-deterministic choice point encountered before the negated goal.

Like regular code blocks, negated code blocks are implemented as compound statement blocks. However, each block is labelled because in some cases outer blocks must be "broken" out of from inner blocks. Since Java does not support a `goto` statement, this can only be achieved using labelled blocks or loops.

As shown in the translation below, the block begins by executing the instructions in the negated goal ($\mathbf{S_0}$). If all instructions succeed then the negated goal fails and backtracking occurs by breaking out of the negated block and not executing those instructions that immediately follow. Any loops or compound statements associated with the instructions in $\mathbf{S_0}$ are closed after the **break** statement. (In this translation $\mathbf{S_1}$ represents all the instructions or nested code blocks which occur after the negated goal, but exist in the same code block as the negated goal.) Assuming the negated goal succeeds, the instructions occurring after it ($\mathbf{S_1}$) are executed. The extra unlabelled statement block around $\mathbf{S_0}$ ensures that variables declared in the negated goal are not visible to or conflict with those of later instructions.

SDSL:
`not`{ $\mathbf{S_0}$ } $\mathbf{S_1}$

$\downarrow$

Java:
```
negCodeBlock<id>: {
    {
        <translation of S0>
          break negCodeBlock<id>;
        <closing brackets associated with S0>
    }
    <translation of S1>
}
```

6.6 SDSL Instructions

The Java representation of each SDSL instructions is now given.

6.6.1 Dynamic Argument Index Instructions

Look

An SDSL *look* instruction is represented in Java as a call to the `look` method in an object implementing the `Node` interface. The object on which the `look` method is called is the subject index variable used in the *look* instruction. The input to this method is the known value to be searched for. The output is a new `Node` object corresponding to the sub-index and is assigned to the output index variable. By overloading `look` methods for all types, the SDSL compiler does not need to specify the correct method to call. When the SDSL *look* instruction is semi-deterministic a test is required to detect when the operation fails. When the output of the `look` method is `null` no matching value has been found in the subject index. *Look* instructions known to be deterministic skip this test.

The translation of both deterministic and semi-deterministic SDSL *look* instructions to Java follows.

SDSL:
`look` : `semidet($`A`, `B`,$`C`)` $\mathbf{S_0}$
↓
Java:
`final Node node`C = `node`A`.look(`B`);`
`if (node`C == `null){`
    <translation of $\mathbf{S_0}$>
`}`
	SDSL:
	`look` : `det($`A`, `B`,$`C`)` $\mathbf{S_0}$
	↓
	Java:
	`final Node node`C = `node`A`.look(`B`);`
	<translation of $\mathbf{S_0}$>

Scan

Because *scan* operations can be nested, it must be possible for multiple *scans* to be active at any one time. Each instance of a *scan* must maintain a reference to the previous values returned so that each element is returned at most once by each *scan*. If the data set is modified while a *scan* operation is active there is no requirement for the *scan* to return any newly added values and there is no requirement for the *scan* to omit any newly deleted values.

To perform each *scan* an iterator is used. Each *scan* operation creates an instance of a class implementing the `NodeIterator` interface, shown in Figure 6.4, that is specialised for the argument index's data structure implementation. `NodeIterator` objects must provide definitions for the `hasNext` method, which determines whether there are any more values that can be returned from the argument index, the `nextNode` method, which returns a sub-index from the argument index and locates the next element ready for the next iteration, and all relevant `nextValue` methods. The `nextValue` methods are used to return each value, given the expected type. Note that not all data structures have suitable definitions for all data types. For example, array values can not be accessed using `double` or `String` arguments. Other data structures may be optimised for one type of input data. In these cases the irrelevant methods have default implementations to appease the Java compiler.

```
public interface NodeIterator{
    boolean hasNext();
    Node nextNode();
    int nextValue_int();
    String nextValue_String();
    double nextValue_double();
}
```

Figure 6.4: The NodeIterator interface.

The order that an iterator's methods are called is important for correctness. The first method that must be called is `hasNext` to guard the iterator against accessing a data structure whose values have all been seen, or attempting to access an empty data structure. Next, the appropriate `nextValue` method is called based on the type of the expected output. Finally the `nextNode` method is called which has the side effect of updating the iterator object.

The concept of iterators is based on Java's `Iterator` interface[2]. However the `NodeIterator` was defined to allow Starlog programs more specialised (and therefore more efficient) methods of access to the values and sub-indexes within data structures (e.g. casting is avoided). Therefore `NodeIterator` does not extend `Iterator`.

The translation of an SDSL *scan* instruction into Java is given below. For each *scan* operation a new <*id*> value is automatically generated to identify the iterator. Having a different name for each iterator ensures that one iterator can not interfere with the running of another. For a non-deterministic *scan* instruction, a loop is used to access all values in an index. This loop is a choice point in the program and will be iterated through when backtracking occurs. A *scan* instruction that is semi-deterministic uses an `if` statement in place of the loop. Deterministic *scan* instructions do not require any loops or conditions for the operation to succeed. The variable $\mathbf{S_0}$ represents the set of instructions and code blocks following the *scan*.

```
            SDSL:
scan : nondet($A, B : <type>, $C) S0

               ↓

            Java:
final NodeIterator iterator <id> = nodeA.getIterator();
while (iterator <id> .hasNext()){
    final <type> B = iterator <id> .nextValue_ <type> ();
    final Node nodeC = iterator <id> .nextNode();
    <translation of S0>
}
```

---

[2]The `Iterator` interface first appeared in Java 1.2. Previous to this the `Enumerator` interface served the same purpose, although both are currently supported [38].

SDSL:

```
scan : semidet($A, B : <type>, $C) S_0
```

$\downarrow$

Java:

```
final NodeIterator iterator <id> = nodeA.getIterator();
if (iterator <id> .hasNext()){
    final <type> B = iterator <id> .nextValue_<type> ();
    final Node nodeC = iterator <id> .nextNode();
    <translation of S_0>
}
```

SDSL:

```
scan : det($A, B : <type>, $C) S_0
```

$\downarrow$

Java:

```
final NodeIterator iterator <id> = nodeA.getIterator();
final <type> B = iterator <id> .nextValue_<type> ();
final Node nodeC = iterator <id> .nextNode();
<translation of S_0>
```

Insert

The semantics of the SDSL *insert* instruction require a new value to be added to an argument index only when the value does not already exist. Otherwise the existing sub-index associated with the value is returned. This operation is performed within the `insert` methods implemented by classes derived from the `Node` interface.

However using `insert` methods require creating auxiliary objects. When a new value is successfully added to an argument index, a new sub-index is created associated with the value. Yet because `insert` methods are not specialised for each index node, they do not automatically know the class of the sub-index to be created. (To create specialised implementations of the `insert` method would require the compiler rewriting every insertion method for whatever data structures implement the `Node` interface. This approach is not modular when new data structures are added, and would complicate the automatic data structure selection process discussed in Chapter 7.) Instead, when calling an `insert` method, the type of the sub-index is passed as a `NodeFactory` object. A `NodeFactory` object has one method, `newNode()`, which generates a new `Node` object of a specific type. When an `insert` method finds that the insert value does not exist in the argument index it needs to create a new sub-index of the appropriate type. To do this the `insert` method calls the `newNode` method from the `NodeFactory` object it has been passed. The definitions of `NodeFactory` classes, that together generate all possible sub-indexes, are created automatically during SDSL compilation. Instances of `NodeFactory` objects are created for all sub-indexes when the program is initialised and reused whenever an *insert* occurs. In this way the overhead of these extra objects is minimised. For simplicity, instances of each `NodeFactory` are given names corresponding to the type of index node they generate. For example, the node factory generating `ExampleIndex0` index nodes from Figure 6.3 would be named `ExampleIndex0Factory`.

When inserting a value into an argument index, the correct `NodeFactory` object must be passed to the `insert` method. To determine which node factory should be used, the class of each index variable is maintained during compilation of SDSL instructions. By simulating execution of instructions on a model of the index structure, the class of each new output index variable is determined. When the class of the sub-index is found, the appropriate `NodeFactory` object is derived by simply appending the sub-index class name with the "Factory" extension.

Given that the appropriate node factory object exists for a new index variable, an SDSL *insert* instruction can be translated to Java as follows.

SDSL:

`insert` : `det`($\$A$, B, $\$C$)

$\downarrow$

Java:

`final Node node`C = `node`A`.insert(`B, <Node factory object for generating C index>);

Minimum

Each class that implements the `Node` interface defines a set of methods to find the minimum elements. To find the minimal sub-index the `minimum` method is called. If the argument index is empty this method will return a null value. When a minimum element does exist in the argument index, its value is found by calling a `minimumValue` method. As with *scan* operations, there are different methods to allow different types of values to be returned. By appending the expected variable type to the `minimumValue_` prefix the appropriate output is returned.

For a *minimum* operation to be non-deterministic these method calls are encapsulated in a loop. The failure (or escape) condition is when the argument index becomes empty.

Unlike *scans*, *minimum* instructions do not require iterator objects for nested searching. This is because non-deterministic *minimum* operations only reference the current state of an argument index and do not keep a record of previously returned results. (The implementation of the *next-minimum* instruction would be very close to that of *scans*, however we do not include the translation since it is unused by the Starlog programs described in this thesis.) When following the Triggering Template (see Chapter 5), minimum elements are deleted from the argument index after they are found so that a new minimum element is found in each iteration.

The Java implementations of the variants of the *minimum* instruction are given below.

SDSL:
```
minimum : nondet($A, B : <type>, $C) S0
```
$\downarrow$

Java:
```
Node nodeC;
while ((nodeC = nodeA.minimum()) != null){
    final <type> B = nodeA.minimumValue_<type> ();
    < translation of S0>
}
```

SDSL:
```
minimum : semidet($A, B : <type>, $C) S0
```
$\downarrow$

Java:
```
final Node nodeC = nodeA.minimum();
if (nodeC != null){
    final <type> B = nodeA.minimumValue_<type> ();
    < translation of S0>
}
```

SDSL:
```
minimum : det($A, B : <type>, $C) S0
```
$\downarrow$

Java:
```
final Node nodeC = nodeA.minimum();
final <type> B = nodeA.minimumValue_<type> ();
< translation of S0>
```

Delete

The Java implementation of a *delete* instruction involves calling the `delete` method defined in an index node class. Java's run time garbage collection ensures that the sub-indexes of any deleted value are deleted when all references to them are removed. Thus run time garbage collection simplifies the implementation of the `delete` methods. The translation of a *delete* instruction follows.

SDSL:
```
delete : det($A, B)
```
$\downarrow$

Java:
```
nodeA.delete(B);
```

Empty

An *empty* SDSL instruction becomes a call to the `isEmpty` method defined in an index node class. Depending on the boolean output of this method the instructions following this may or may not be executed.

```
        SDSL:
empty : semidet($A) S0

        ↓

        Java:
if (nodeA.isEmpty()){
    <translation of S0>
}
```

6.6.2 Labelled Branch Instructions

In Java each static labelled branch and labelled boolean value is represented in an index node's class as a field variable. To find the sub-index or boolean value we access these fields. However, the fields in the derived classes are not immediately accessible because all index variables are declared as `Node` objects, which do not have these fields. Therefore index variables must be cast to their derived classes to make their fields accessible. Since object casting is expensive, it is more efficient to use casting only when the fields are required, rather than every time an index variable is assigned. For this reason the Java implementations of labelled branch instructions and labelled boolean value instructions perform object casting.

The derived class of each index variable is found during compilation by creating a model of the index structure and simulating the execution of instructions on this model.

Follow

The *follow* instruction simply accesses a field within an index node object. A semi-deterministic *follow* instruction will fail if the sub-index of a branch has not been initialised (i.e. is a null value). A deterministic *follow* instruction does not require testing the contents of the sub-index.

```
        SDSL:
follow : semidet($A, L, $C) S0

        ↓

        Java:
final Node nodeC = ((<index type>)nodeA).L;
if (nodeC != null){
    <translation of S0>
}
```

```
        SDSL:
follow : det($A, L, $C) S0

        ↓

        Java:
final Node nodeC = ((<index type>)nodeA).L;
<translation of S0>
```

Establish

To ensure a labelled branch is initialised an *establish* instruction is used. This instruction examines the field variable representing the labelled branch, and if its value is null then a new sub-index is created.

The Java code for an SDSL *establish* instruction is shown below. This code appears to be sub-optimal since it makes an unnecessary number of casts. Although the use of a temporary variable could improve performance it would obfuscate the code. Moreover, it is believed to be an unnecessary optimisation since Java compilers perform Common Subexpression Elimination [2] on the code and introduce such temporary variables automatically.

```
        SDSL:
establish : det($A, L, $C)

          ↓

        Java:
if ((<index type>)nodeA).L == null){
    ((<index type>)nodeA).L = new <sub-index type> ();
}
final Node nodeC = ((<index type>)nodeA).L;
```

Prune

To remove the sub-index associated with a labelled branch the *prune* instruction is used. In Java this translates into setting an index node's field variable to null. Java's run time garbage collection reclaims the removed sub-indexes when no references remain. The translation of *prune* instructions to Java follows.

```
        SDSL:
prune : det($A, L)

          ↓

        Java:
((<index type>)nodeA).L = null;
```

Is-Pruned

Testing if a labelled branch is empty is achieved using an *is-pruned* instruction. The Java implementation of this instruction is a `null` check on the field variables corresponding with the labelled branch.

```
        SDSL:
ispruned : semidet($A, L) S0

          ↓

        Java:
if (((<index type>)nodeA).L == null){
    <translation of S0>
}
```

Link

The *link* instruction is used to reference an existing index node with a labelled branch. In Java this becomes a simple assignment of the field variable representing a labelled branch to an existing index node.

SDSL:

`link` : `det`($\$A$, L, $\$C$)

$\downarrow$

Java:

((<*index type*>)`node`A).L = `node`C;

6.6.3 Labelled Boolean Value Instructions

Test

Testing the contents of a labelled boolean value is achieved with the *test* instruction. In Java, the boolean value held in a labelled field variable is compared with the query value.

SDSL:

`test` : `semidet`($\$A$, L, V) $\mathbf{S_0}$

$\downarrow$

Java:

```
if (((<index type>)nodeA).L == V){
    <translation of S0>
}
```

Set

The *set* instruction updates a labelled boolean value in a given index node. This becomes a simple assignment statement when translated to Java.

SDSL:

`set` : `det`($\$A$, L, V)

$\downarrow$

Java:

((<*index type*>)`node`A).L = V;

6.6.4 Built-ins

Each *built-in* instruction used in SDSL programs has a Java implementation. *Built-in* instructions are used in the bodies of Starlog rules or used to perform side-effects associated with the heads of some rules.

To support modularity, each of the basic types of *built-ins* (e.g. add, subtract, multiply etc.) is implemented as a class of static, final methods, all of which make up the *Builtin* package. The static, final declaration of these methods allows the Java compiler to perform in-lining code optimisation for most instances. The method names correspond to the determinism of the *built-in* and the input variables are overloaded for all types applicable to the operation.

For compilation purposes, *built-ins* can be divided into deterministic, semi-deterministic and non-deterministic operations, and then further categorised by the number of output variables they produce (none, one or many). Although some of these cases are rarely used, the Java representations of all possible types of built-ins are now discussed.

Deterministic *built-in* operations with no output simply become `void` method calls, as follows. These *built-ins* usually perform side-effects, such as printing to the standard output.

SDSL:

`builtin` : `det`(B, [I_0, ... I_n], [])

$\downarrow$

Java:

B.`det`(I_0, ... I_n);

For deterministic *built-in* operations with a single output variable, the output becomes the method's returned variable.

SDSL:

`builtin` : `det`(B, [I_0, ... I_n], [O : <*type*>])

$\downarrow$

Java:

`final` <*type*> O = B.`det`(I_0, ... I_n);

When a deterministic *built-in* has multiple output variables, these are packaged together in a *variable return object*, whose class has been constructed specifically for this *built-in*. The outputs are stored as public field variables and named **arg0**, **arg1**, **arg2** and so on, in accordance with the order, types and cardinality of the output variables in the SDSL *built-in* instruction. These field variables are re-mapped to program variables before they are used. The additional objects and re-mapping of variables makes *built-ins* with multiple outputs more expensive to evaluate, however such operations are rarely defined or used.

SDSL:

`builtin` : `det`(B, [I_0, ... I_n], [O_0 : <*type*$_0$>, ... O_m : <*type*$_m$>]) $(m > 0)$

$\downarrow$

Java:

`final` B`VariableReturnObject output`<*id*> = B.`det`(I_0, ... I_n);
`final` <*type*$_0$> O_0 = `output`<*id*> .arg0;
...
`final` <*type*$_m$> O_m = `output`<*id*> .arg*m*;

A semi-deterministic operation needs to communicate its success or failure back to the parent program. A *built-in* that has no output takes advantage of the unused return value and returns a boolean value indicating success or failure.

SDSL:

`builtin` : `semidet`(B, [I_0, ... I_n], []) $\mathbf{S_0}$

$\downarrow$

Java:

```
if (B.semidet(I0, ... In)){
    < translation for S0>
}
```

Semi-deterministic *built-ins* that output a single value already return values from their respective Java methods when successful. To indicate that such a method has failed reserved values are used. (For integers the reserved value is the fringe value `Integer.MIN_VALUE`, the reserved double value is `Double.MIN_VALUE`, and the reserved string is `null`.) The parent program does not need to know the reserved values of all types because each class which implements this type of *built-in* instruction provides a `valid` method that is overloaded to take all possible output types from the *built-in*. By passing the value returned from the *built-in* to the `valid` method, the success or failure of the *built-in* can be determined.

SDSL:

`builtin` : `semidet`(B, [I_0, ... I_n], [O : <*type*>]) $\mathbf{S_0}$

$\downarrow$

Java:

```
final <type> O = B.semidet(I0, ... In);
if (B.valid(O)){
    < translation for S0>
}
```

A semi-deterministic *built-in* operation which has multiple output variables returns a variable return object when successful and a null value upon failure.

SDSL:

`builtin` : `semidet`(B, [I_0, ... I_n], [O_0 : <*type*$_0$>, ... O_m : <*type*$_m$>]) $\mathbf{S_0}$ ($m > 0$)

$\downarrow$

Java:

```
final BVariableReturnObject output<id> = B.semidet(I0, ... In);
if (output<id> != null){
    final <type0> O0 = output<id> .arg0;
    ...
    final <typem> Om = output<id> .argm;
    < translation for S0>
}
```

Non-deterministic *built-ins* use iterators – similar to those used to *scan* through an index – to return all valid variable bindings. The appropriate iterator object is automatically generated when a non-deterministic *built-in's* method is called. Although all iterator classes used for *built-ins* are derived from a `BuiltinIterator` interface (see Figure 6.5) and must provide a `hasNext` method, each iterator specifies its own `next` method whose output type is customised for the *built-in* operation. The names of all classes derived from the `BuiltinIterator` interface consist of the *built-in* operation's name concatenated with "`BuiltinIterator`" in order to simplfy the Java translations.

```
public interface BuiltinIterator{
    boolean hasNext();
    // The output of the next() method is specific
    //  to the built-in so is not defined here.
}
```

Figure 6.5: The BuiltinIterator interface.

If the *built-in* has only one output variable then the `next` method in the iterator returns a variable of a primitive type, as shown here. (Notice that the iterator is declared as B`BuiltinIterator` so that the `next` method is accessible.)

```
        SDSL:
builtin : nondet(B, [I_0, ... I_n], [O : <type>]) S_0

          ↓

        Java:
final BBuiltinIterator biIterator<id> = B.nondet(I_0, ... I_n);
while(biIterator<id> .hasNext()){
    final <type> O = biIterator<id> .next();
    < translation for S_0>
}
```

Otherwise, when a non-deterministic *built-in* instruction has multiple output variables, the output from the iterator's `next` method is a variable return object from which variable bindings are extracted. Again, the creation of extra iterator objects and variable return objects will affect the performance of programs. However, non-deterministic *built-ins* are rarely defined and non-deterministic built-ins that return multiple output variables are even more scarce.

```
        SDSL:
builtin : nondet(B, [I_0, ... I_n], [O_0 : <type_0>, ... O_m : <type_m>]) S_0
                                                                    (m > 0)
          ↓

        Java:
final BBuiltinIterator biIterator<id> = B.nondet(I_0, ... I_n);
while(biIterator<id> .hasNext()){
    final BVariableReturnObject output<id> = biIterator<id> .next();
    final <type_0> O_0 = output<id> .arg0;
    ...
    final <type_m> O_m = output<id> .argm;
    < translation for S_0>
}
```

6.7 Example of Compiled SDSL Code

Compilation of an SDSL program to Java is demonstrated using the fully optimised program defined in the previous chapter (Figure 5.19). Although this is a compact example in both Starlog and SDSL, its Java implementation is much larger. It is for this reason we do not give the translation of more complex example programs that demonstrates translation of all the instructions.

The first lines of any Java program declare which external classes are necessary to run the program. In addition to the `java.lang` standard package, programs translated from SDSL must import the `builtin` and `datastructures` packages. These include custom made classes that streamline the compilation of SDSL programs. The `builtin` package contains classes that implement all *built-in* operations possible in an SDSL program, whereas the `datastructures` package contains a library of data structure implementations (including their iterators) on which each node of the index structure is based.

Each type of node in the index structure is implemented as a class that extends a class from the `datastructures` package. (Selecting which data structure to extend is discussed in the next chapter.) The names of each new class are created automatically from the program's file name (here called `ExampleProgram`) and an identifier. In this program, class `ExampleProgram0` is the root node (shown in the index structure diagram of Figure 5.5). `ExampleProgram1` is the sub-index of the root holding Γ tuples, each identified with a boolean field. The labelled branch "`delta`" in the root node is implemented as a field of type `ExampleProgram2`. The sub-index of `ExampleProgram2` is `ExampleProgram3`, which contains two boolean fields that identify each of the tuples in Δ.

The next set of classes are the node factories used to generate correct sub-indexes during an *insert*. Each node factory implements a `newNode` method that returns a new object from the appropriate class.

The final class contains the `main` method where instructions from the SDSL program are implemented. The initialisation of this program involves generating instances of node factories that can be passed as a parameter to *insert* operations, and the declaration and initialisation of the root node of the index structure instance. The root node of any SDSL program (named `node0`) is initialised before the first code block to give it a global scope. Initialisation is achieved using a call to the `newNode` method in the appropriate node factory object.

Each instruction and control element in an SDSL program is now translated into its Java equivalent using the previously described mappings. In this example we have included the original instruction as a comment before its Java representation to aid understanding.

As mentioned previously, there is a clash between the variable naming conventions of Java and Starlog. The program variables used in this example conform with Java's convention that every variable begins with a lowercase letter. Oridinarily the automatically generated Java code will use the Starlog variable naming conventions.

```
import java.lang.*;
import builtin.*;
import datastructures.*;

// Index Structure Node class definitions
class ExampleProgram0 extends SortedListNode{
  public ExampleProgram2 delta;
}

class ExampleProgram1 extends EmptyNode{
    public boolean id0;    public boolean id2;
}

class ExampleProgram2 extends SortedListNode{
}

class ExampleProgram3 extends EmptyNode{
    public boolean id1;    public boolean id3;
}

// Node Factory class definitions
class ExampleProgram0Factory implements NodeFactory{
  public Node newNode(){
    return new ExampleProgram0();
  }
}

class ExampleProgram1Factory implements NodeFactory{
  public Node newNode(){
    return new ExampleProgram1();
  }
}

class ExampleProgram2Factory implements NodeFactory{
  public Node newNode(){
    return new ExampleProgram2();
  }
}

class ExampleProgram3Factory implements NodeFactory{
  public Node newNode(){
    return new ExampleProgram3();
  }
}

// Program implementation class
public class ExampleProgram{
  public static void main(String[] argv){
    final NodeFactory exampleProgram0Factory = new ExampleProgram0Factory();
    final NodeFactory exampleProgram1Factory = new ExampleProgram1Factory();
    final NodeFactory exampleProgram2Factory = new ExampleProgram2Factory();
    final NodeFactory exampleProgram3Factory = new ExampleProgram3Factory();

    final Node node0 = exampleProgram0Factory.newNode();


    // Beginning of SDSL Code
    {
      //establish:det($0,delta,$1)
      if (((ExampleProgram0)node0).delta == null){
        ((ExampleProgram0)node0).delta = new ExampleProgram2();
      }
      final Node node1 = ((ExampleProgram0)node0).delta;
      //insert:det($1,0,$2)
      final Node node2 = node1.insert(0,exampleProgram3Factory);
      //set:det($2,id3,true)
      ((ExampleProgram3)node2).id3 = true;
      //minimum:nondet($1,MinVar0:int,$4)
      Node node4;
      while ((node4 = node1.minimum()) != null){
        final int minVar0 = node1.minimumValue_int();
        {
          {
            //test:semidet($4,id3,true)
            if (((ExampleProgram3)node4).id3 == true){
              //insert:det($0,MinVar0,$5)
              final Node node5 = node0.insert(minVar0,exampleProgram1Factory);
              //set:det($5,id2,true)
              ((ExampleProgram1)node5).id2 = true;
              //not{
```

```
              negCodeBlock5:{
                {
                  //scan:nondet($0,Y1:int,$6)
                  final NodeIterator iterator6 = node0.getIterator();
                  nondetLoop6:while(iterator6.hasNext()){
                    final int y1 = iterator6.nextValue_int();
                    final Node node6 = iterator6.nextNode();
                    //test:semidet($6,id2,true)
                    if (((ExampleProgram1)node6).id2 == true){
                      //builtin:semidet(greaterThan:semidet,[Y1,1],[])
                      if (greaterThan.semidet(y1, 1)){
                        break negCodeBlock5;
                      }
                    }
                  }
                }
                // not}
                //builtin:det(multiply:det,[MinVar0, 2],[Z1:int])
                final int z1 = multiply.det(minVar0, 2);
                //insert:det($1,Z1,$8)
                final Node node8 = node1.insert(z1,exampleProgram3Factory);
                //set:det($8,id1,true)
                ((ExampleProgram3)node8).id1 = true;
              }
            }
          }
          //test:semidet($4,id1,true)
          if (((ExampleProgram3)node4).id1 == true){
            // not{
            negCodeBlock7:{
              {
                //scan:nondet($0,MinVar1:int,$9)
                final NodeIterator iterator8 = node0.getIterator();
                while(iterator8.hasNext()){
                  final int minVar1 = iterator8.nextValue_int();
                  final Node node9 = iterator8.nextNode();
                  //test:semidet($9,id2,true)
                  if (((ExampleProgram1)node9).id2 == true){
                    //builtin:semidet(GreaterThan:semidet,[MinVar1, 1],[])
                    if (greaterThan.semidet(minVar1, 1)){
                      break negCodeBlock7;
                    }
                  }
                }
              }
              // not}
              //insert:det($0,MinVar0,$10)
              final Node node10 = node0.insert(minVar0,exampleProgram1Factory);
              //set:det($10,id0,true)
              ((ExampleProgram1)node10).id0 = true;
              //test:semidet($10,id2,true)
              if (((ExampleProgram1)node10).id2 == true){
                //builtin:semidet(greaterThan:semidet,[10, MinVar0],[])
                if (greaterThan.semidet(10, minVar0)){
                  //builtin:det(add:det,[MinVar0, 1],[Z2:int])
                  final int z2 = add.det(minVar0, 1);
                  //insert:det($1,Z2,$13)
                  final Node node13 = node1.insert(z2,exampleProgram3Factory);
                  //set:det($13,id3,true)
                  ((ExampleProgram3)node13).id3 = true;
                }
              }
            }
          }
        }
        //delete:det($1,MinVar0)
        node1.delete(minVar0);
      }
    }
  }
}
```

6.8 Conclusions

The translation scheme for compiling SDSL programs into Java source code has been given. The compiled programs take advantage of Java's object orientation and class hierarchies to specialise the index structure nodes, which are based on predefined data structures. This has the effect of simplifying the SDSL compiler while promoting a modular design. For example, to add a new data structure to the index library or a new *built-in* operation to the language, a new class can be added to the appropriate package. The compiler's code remains unchanged.

Effort has been made to produce efficient Java programs from SDSL by reducing the number of expensive operations performed whenever possible. In Chapter 8 the efficiency of some compiled Starlog programs is shown to be competitive with that of hand-coded programs. To further improve the efficiency of compiled SDSL programs a different target language could be chosen. Most of the translations presented in this chapter would be relevant when compiling to other imperative programming languages. However targeting languages which are not object oriented or do not provide automatic garbage collection would make compilation more complicated.

The next chapter discusses the process of Data Structure Selection that allow Java programs produced by the SDSL compiler to be further optimised.

Chapter 7

Data Structure Selection

During the design of the index structure it was assumed that the values of each argument were stored in a set. Later, during compilation of SDSL programs to Java, argument indexes were refined to use a common interface. As yet the implementation of the indexes has not been addressed.

Substituting different data structures into each index can alter the efficiency of programs. As shown in the compilation pipeline in Figure 7.1, using SDSL – a data structure independent language – the assignment of data structures can be delayed until after the SDSL program is compiled. This situation has two advantages. The implementation of each index can be based on properties of the program, thus more efficient data structures may be selected. In other languages, programs depend on data structures which may have been selected prematurely by the programmer, perhaps before the use of each data structure has been adequately defined [97]. The second advantage is that changing programs to use different data structures is simpler than in other languages. In most other languages, changing data structures usually requires large-scale rewriting of programs [22, 98], which reduces the ability to experiment with different the data structures. (Although changing data structures can be simplified when different data structures have a common interface.)

In this chapter we argue that the best solution for declarative languages is to allow the compiler to select the data structures. The concept of automatic data structure selection has previously been experimented with for some other languages and applications. For instance, [70] describes a compiler for the Algol-like language SAIL which automatically chooses its data structures

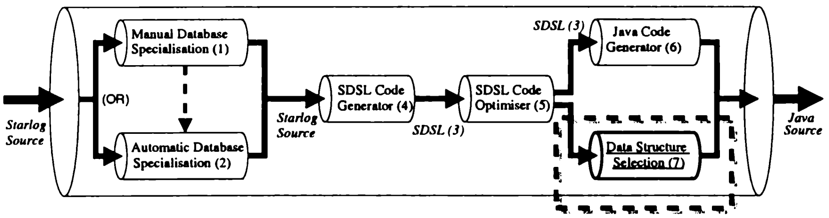


Figure 7.1: Data structure selection in Starlog's compilation pipeline.

by estimating the costs of operations. The feasibility of automatic selection of high-level data structures such as sets, lists and triples is demonstrated, however no evaluation of its effectiveness is given. More details of SAIL's data structure selection technique are discussed later in this chapter.

SETL, a "very high level programming language supporting set theoretical syntax and semantics", selects between bit arrays, hashed bit arrays or hashed bit arrays with dynamic links by propagating the local hashing requirements of operations to a global context [97, 9]. This language uses abstract data structures explicitly in its programs but automatically assigns their implementations to optimise their hashing requirements [98]. The algorithm to select data structures for SETL programs assigns a local "base" implementation to each data set mentioned in a program. "Base" implementations are said to allow access to data in an especially efficient manner. Global analysis then merges equivalent "base" implementations with respect to the combination of operations performed on each set. This technique does not take into account any characteristics of the data set to make its selection. The SETL data structure selection technique has been applied to several test programs and has been said to have "produced very satisfactory results" [98]. SETL has been extended to the SETM language which uses more data structures but uses type inference to select data structures where each data structure is considered a sub-type [16].

More recently, work on data structure selection performed at run time has been documented in [21]. The technique described uses Markov processes with random choices. Although the results of this technique are shown to be promising, the differences in performance are from simulations and do not take into account all overhead incurred by the selection technique. Later in this chapter a similar run time selection technique is described and evaluated.

There has also been research into optimising programs with sparse matrices into programs that use dense data structures (see [13] and [14]). This involves not only a change to the data structure implementation but also to the program itself.

In spite of automatic data structure selection being recognised as a desirable feature for high-level programming languages, it appears this is the first work of its kind for a logic programming language. We begin this chapter by describing the data structures implemented for Starlog's argument indexes and discuss why generic data structures and manual selection of data structures are unsuitable for Starlog. We then give automated strategies for selecting implementations and report on experiments that use these strategies.

7.1 Data Structure Implementations

Before details of the data structure selection strategies are discussed, the set of data structures used to implement argument indexes is given. The six data structures given in Figure 7.1 represent common data structure paradigms. To allow comparison, the worst case time complexities are given for the operations used by Starlog programs.

For a detailed discussion about these data structures see Appendix B. Although there are many other relevant data structure implementations – and in future releases of the compiler more will appear – for now these six data structures are all that are necessary to test the concept of automatic data structure

Data Structure	*look*	*scan*	*insert*	*delete*	*minimum*	*empty*
Empty Data Structure	No operations are valid for the Empty Data Structure					
Unsorted List	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(1)$
Sorted List	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(1)$	$O(1)$
Balanced Tree	$O(logN)$	$O(NlogN)$	$O(logN)$	$O(logN)$	$O(logN)$	$O(1)$
Hash Table*	$O(1)$	$O(N)$	$O(1)$	$O(1)$	$O(N)$	$O(1)$
Flexible Array	$O(1)$	$O(M)$	$O(1)$	$O(1)$	$O(M)$	$O(M)$

* Time complexities for the Hash Table assume a perfect hashing function and a constant sized table.

N is the number of elements stored in the data structure.
M is the difference between the maximum and minimum integers represented in the Flexible Array.

Table 7.1: Available data structure implementations and a summary of their time complexities.

selection.

In general the chosen data structures provide robustness and predictable complexities rather than striving for optimal efficiency in a few cases. For example, it is well known that searching a regular binary tree has an $O(logN)$ complexity when the data set is randomly distributed but the complexity becomes $O(N)$ when the data is sorted. Yet a balanced binary tree will have similar complexities irrespective of the data set. Therefore our binary tree implementation performs balancing.

An important requirement of the data structures is that no duplicate elements are stored. This is because Starlog's index structures, like discrimination trees, need to combine equal terms to optimise the term retrieval process. To ensure duplicate values are not introduced, data structures perform additional tests during insert operations. Although in some cases (such as the unsorted list) the duplicate checking requirement adds a significant overhead, for most data structures it can be achieved by adding lightweight equality checks to the insert operations. Optimised versions of insert operations could be implemented which do not perform a duplicate check. The conditions for using these optimised insert operations are outlined in Chapter 8 in the future work section.

Although examples of these and other data structures appear in the Java standard libraries, these existing implementations were not used because standard implementations do not have all the functionality required for argument indexes. For example, to ensure duplicates are not added during an insert often requires an additional search that, in general, would double the operation's complexity. Also the methods to find the minimum element do not exist for all standard data structures. Another problem with data structures from standard libraries is that they are often too general at the cost of efficiency. Standard data structures store objects of any type and incur overhead for methods that are unnecessary for an argument index. The data structure implementations described here were constructed from the ground up with the goal of efficiently indexing arguments. As a result the implementations are lean and have known and predictable complexities.

7.2 Problems with Generic Data Structures and Manual Data Structure Selection

High-level applications (including deductive databases) often avoid data structure selection by using generic data structures to store run time data. For example, both standard Prolog and XSB use hash tables to index all predicates [84, 96]. The CORAL deductive database uses hash tables for all in-memory relations and B-trees for persistent relations [89].

Although these data structures scale well for large data sets, they are often less efficient than more specialised implementations. To illustrate, the use of a hash table is unnecessarily complex if the data set contains only a few elements. Usually the inefficiency of generic data structures for small data sets is not a great concern since programs that generate small data sets are usually sufficiently efficient. However this is not true for Starlog's nested index structures where, although each argument index may only hold a few elements, there can be a many instances of an argument index. Because using generic data structures in Starlog index structures can result in very inefficient programs, specialised data structures are preferable.

In low-level programming languages it is always possible for programmers to specify the data structures used to store run time data. This is a good idea because programmers usually have an intuition about the data stored in low-level programs and can make informed decisions that an automated system may not. For example, a programmer will know from their program specification that the size of a data set will remain unchanged after a quicksort has been applied. However for a compiler to infer this same property from the quicksort's source code requires proof by induction. Because compilers are not yet capable of proving such properties an automated selection technique might assume a dynamic structure is necessary. This selection may be inferior to that made by the programmer who will know a structure with a static size is sufficient.

However as languages become more abstract the programmer is less likely to know details of their program's run time data. This is especially true for declarative languages where programmers describe a program specification [49] and are not required to understand the execution model in order to produce correct programs [78]. (A current problem with Prolog is that programmers must be aware of the indexes and instantiation patterns when constructing a query to generate an efficient program [96, 22].) Furthermore, applications which represent data as relations deliberately hide the underlying implementations from users. As a result, programmers find that declarative programs are easy to write, yet hard to optimise.

Another factor is that human error can contribute to a poor selection of data structures. Programming languages often provide abstract data structures either built-in to the language (e.g. lists in Lisp) or as external modules (e.g. Java's Collection classes). When selecting a predefined data structure the programmer is usually unaware of the specifics of the implementation and so may be surprised by its performance, or lack thereof. Although it is prudent for programmers to be familiar with the implementations of any data structures that they use, it is often impractical and instead programmers rely on documentation which may be insufficient to assess run time performance.

The index structures used by Starlog programs compound the problems of manual data structure selection. Each argument index in a Starlog index structure holds only the argument values of tuples *given* the previously indexed arguments. For example, if predicate `fib(X,Y)` holds elements in the Fibonacci series where `Y` is the Fibonacci value and `X` is its index in the series, given a value of `X` there will be at most one `Y` value. It is more difficult for programmers to predict the properties of an argument's data set relative to those arguments already indexed than properties of the argument's data set itself. Also, the issue of manual data structure selection is complicated because the order that arguments are indexed is decided by the compiler (see Chapter 3).

Above all, automatic data structure selection is in keeping with the philosophy of declarative programming, where programmers are abstracted from the underlying mechanics of their programs and instead are only concerned with its logic.

7.3 Automatic Selection

An alternative approach to generic data structures and manual selection is to allow the compiler to select the data structures. We present five techniques for automatically selecting data structures.

The efficiency of a data structure in a particular situation depends on two factors: (1) how the data structure is used (i.e. the operations performed on it) and (2) what is stored (i.e. the characteristics of the data set). For example, an unsorted list would seem a very inefficient choice for most situations, however, if the data set is very small, or if the only operations performed on it are scans, then the unsorted list is the best choice. For now we assume that (2), details about the data, can only be accurately determined by running the program.

How a data structure is used can be predicted from a program's source code. SDSL programs contain all the operations that can be performed on argument indexes. We introduce two techniques that analyse an SDSL program and draw conclusions about the operations performed at run time. These two techniques make different assumptions about the frequency of operations appearing in a programs and the characteristics of the data.

In general, to select more efficient data structures it is necessary to have more information about the run time characteristics of the program. We have developed two data structure selection techniques that execute the program and record statistics. The first technique uses a single program run to collect the number of operations performed as well as details about the data sets stored in each argument index. This data is entered into a model which computes the best data structure for each index. The second technique records multiple execution times of the program using different combinations of data structures and applies regression analysis. The regression analysis calculates how each data structure contributes to the total complexity and the data structures resulting in the lowest predicted run time are selected.

A different approach to automatic data structure selection is to delay the selection process until run time and use actual properties of the program rather than estimates or averages. We present a data structure which is capable of analysing its performance and changing its underlying implementation on the fly to improve efficiency.

From this point onward, efficiency only refers to the time complexity of programs. This is the priority for programs which do not require prohibitive amounts of memory to function. Chapter 8 discusses modifications to the data structure selection techniques to minimise the memory usage of programs.

Each of the data structure selection techniques is discussed over the next few sections. The data structure selection techniques are applied to seven example programs in Appendix C to compare their performance in realistic applications. After analysing the performance of each selection technique on each example program we evaluate each selection technique in the context of Starlog programs. The machines used during all the experiments in this chapter were AMD Athlon(TM) XP 1600+ with 256MB of main memory and 256 KB of cache. These were running Linux version 2.4.17 and use the Java compiler from the Java(TM) 2 SDK, Standard Edition Version 1.3.1 and the Java HotSpot(TM) Client VM (build 1.3.1).

7.3.1 Static Selection Techniques

Static data structure selection is the process of choosing data structures using only the program's source code as a guide. In other words, the program is not executed to collect run time performance information.

Set-Based Approach

The first static selection technique analyses a program's SDSL code to determine the set of instructions performed on an argument index. This set of instructions is mapped to what is considered the most appropriate data structure. The advantage of this approach is that the analysis required is very simple.

The data structures associated with each set of instructions were chosen after analysis of each data structure's performance using general assumptions about an instruction's frequency and what constitutes a typical data set from a Starlog program. The general assumptions were based on data sets sampled from previous Starlog programs, experience with implementations of Starlog programs, and knowledge of each data structure's implementation.

Table 7.2 gives the chosen mapping from a set of instructions to a data structure. Notice that all valid sets must contain an insert operation and at least one "read" type instruction (look, scan or minimum). Any other combination of instructions is assigned an empty data structure since an index that never has an item added or an index whose data is never examined is redundant.

The flexible array data structure is absent from this list. This is because the complexity of some operations depends on the maximum and minimum elements in the array. Flexible arrays are not used because these values are not estimated by this selection strategy.

This selection technique is the closest to that implemented in the SETL compiler (see [9, 97, 98]) in that only the set of instructions performed on an index affects the choice. However the technique given here is considerably simpler than that used for SETL since the effect of assigning each data structure does not have to be propagated throughout the program. Instead, the set-based data structure selection technique performs a global analysis which takes all instructions in the program into account before choosing data structures.

Due to the very general assumptions made by this strategy, the data structures selected for a program are not expected to be optimal. Indeed the main

Set of Instructions	Selected Data Structure
[insert, look]	Hash Table
[insert, look, empty]	Hash Table
[insert, look, delete]	Hash Table
[insert, look, delete, empty]	Sorted List
[insert, minimum]	Sorted List
[insert, minimum, empty]	Sorted List
[insert, minimum, delete]	Sorted List
[insert, minimum, delete, empty]	Sorted List
[insert, scan]	Unsorted List
[insert, scan, empty]	Unsorted List
[insert, scan, delete]	Unsorted List
[insert, scan, delete, empty]	Unsorted List
[insert, look, scan]	Hash Table
[insert, look, scan, empty]	Hash Table
[insert, look, scan, delete]	Hash Table
[insert, look, scan, delete, empty]	Hash Table
[insert, look, minimum]	Balanced Tree
[insert, look, minimum, empty]	Balanced Tree
[insert, look, minimum, delete]	Sorted List
[insert, look, minimum, delete, empty]	Sorted List
[insert, scan, minimum]	Sorted List
[insert, scan, minimum, empty]	Sorted List
[insert, scan, minimum, delete]	Sorted List
[insert, scan, minimum, delete, empty]	Sorted List
[insert, look, scan, minimum]	Balanced Tree
[insert, look, scan, minimum, empty]	Balanced Tree
[insert, look, scan, minimum, delete]	Balanced Tree
[insert, look, scan, minimum, delete, empty]	Balanced Tree
Other	Empty Data Structure

Table 7.2: Sets of instructions and data structures.

advantage of this approach is the simple analysis required.

Static Cost Analysis

A more sophisticated approach to static data structure selection observes the number of occurrences of each instruction type in an SDSL program. The number of occurrences in the source code may be an indication of the frequency that operations are performed at run time. For example, if there are two look instructions and only one scan instruction in a program then a conclusion may be drawn that look operations are performed more often than scans. Of course, due to non-deterministic loops in the program, the number of occurrences of instructions can be unrelated to the actual number of operations performed.

The infinite combinations of instructions that can exist in a program means that mapping every case to a data structure is impossible. Instead we use a quantitive approach where the time cost of each data structure in each argument index is estimated and the data structure with the lowest estimated cost is selected. To estimate the time cost, each type of operation has been benchmarked using what is considered typical data. The assumptions made during these benchmarks are as follows.

- All data structures hold an average of 100 elements.
- The data set consists of randomly distributed integers between 0 and 200.
- All look operations have a 50% success rate.
- All insert operations have a 0% duplicate rate.

In retrospect, the last assumption is particularly inaccurate for most programs, however it was made before the significance of duplicated inserts was known. Whether any of these assumptions approximate the behaviour of typical Starlog programs will be seen in the next section.

Based on these assumptions, Table 7.3 gives the time estimates for each operation in each data structure. These values are dependent on the architecture of the subject machine and, for improved accuracy, should be recalculated when this selection technique is migrated to different machines. However, in general, the relationships between time estimates should remain relatively unchanged on most machines.

The operation times in Table 7.3 were calculated using Java's `System.currentTimeMillis()` built-in method, where precision was improved by performing each operation multiple times (between 10,000 and 1,000,000 times depending on the execution time of the operation). Care was taken to subtract any loop overhead required for multiple operations and the time required to access any input data. The times for operations on the Empty Data Structure are set to infinity so that this data structure is not chosen if there are any operations performed on it.

Selecting the data structure with the lowest cost estimate is outlined in Figure 7.2. To estimate the total time cost of using a data structure for an argument index, the number of occurrences of each instruction is multiplied by the estimated cost of the operation it performs. The sum of all the individual operation costs is the total cost estimate of the data structure. The data structure with

Data Structure:	look	scan	insert	delete	minimum	empty
Empty Data Structure	∞	∞	∞	∞	∞	∞
Flexible Array	0.194	105.0	0.692	0.345	0.540	0.310
Unsorted List	7.09	38.5	9.84	5.58	17.6	0.0732
Sorted List	6.85	38.6	10.6	8.53	0.268	0.0727
Balanced Tree	1.03	264.0	6.18	11.9	0.674	0.0733
Hash Table	1.10	78.0	4.16	1.50	36.7	0.270

Cost values are microseconds (μs).

Table 7.3: Instruction costs used for static cost analysis.

$$selected(index) = argmin_{w \in DS}\ totalCost(index, w)$$
$$totalCost(index, ds) = \sum_{o \in Ops} (occurrences(index, o) * cost(ds, o))$$

Let DS be the set of all available data structures.
Let Ops be the set of all data structure operations.
Let $occurrences(A, B)$ denote the number of occurrences of instruction of type B operating on index A in the program's source.
Let $cost(A, B)$ denote the average time taken to perform operation B on data structure A. (This is equivalent to a table lookup in Table 7.3.)

Figure 7.2: Formula for selecting a data structure using static cost analysis.

the minimum cost estimate is selected to implement the argument index. In the event that two or more data structures have the same cost estimate the first implementation is chosen as listed in Table 7.3. This ensures the empty data structure is used when appropriate.

To improve this selection strategy it was speculated that the number of occurrences of an operations could be modified according to the number of non-deterministic loops the operation was embedded in. That is, more emphasis could be placed on instructions that are inside loops than those that are not. The number of occurrences of each instruction in the program is equivalent to each non-deterministic loop succeeding exactly once. Depending on the data set, this would be an under-estimate in some cases and an over-estimate for others (when the loop never succeeds or is never entered). By making the assumption that all non-deterministic loops would iterate either 2, 5 or 10 times, the results were promising for the few programs for which this was true. However the programs that had loops which iterate fewer times than these estimates made increasingly poor selections. For this reason and because emphasising instructions performed within non-deterministic loops is actually a form of data set prediction (which our static selection strategies do not attempt), it has not been pursued further.

Static cost analysis is designed to make a better selection than set-based selection by inferring more information from the program. However it requires modelling the time costs of operations using very general assumptions about the data set. Unlike set-based selection which is designed to make a conservative selection, the effectiveness of this approach is dependent on how well the cost model fits the actual properties of the program.

7.3.2 Dynamic Selection Techniques

Without execution it is difficult to infer the run time properties of a program. Therefore a different approach to automatic data structure selection is to execute the program and record details about its operation to aid the selection process. By observing the properties of an executing program, more accurate measures of the effect and significance of each operation can be found. Moreover, properties of the data set can be accurately measured to improve the selection.

This approach is not without problems. The first is that it requires the program to terminate in a reasonable time. Without the program terminating the selection process can not continue. To avoid problems with non-terminating programs (or programs with very long execution times) artificial termination conditions can be added or termination can be forced so that data structure selection can proceed. However, changing the termination condition of a program may alter the properties of a program – now it may not be considered a typical run since the properties of the program and its data sets may change dramatically without this termination.

A second disadvantage of dynamic selection techniques is that to execute a program adds additional overhead to the compilation process. For the regression based selection technique that requires multiple runs, the overhead can be excessive for program development. Unfortunately, this can not be avoided since decreasing the run time of a program alters the run time properties and reduces the precision of program timing.

One major difference from the static analysis techniques is that the dynamic selections are specific to the data sets used during the trial executions. For programs which change their data sets from run to run (such as those that take input from an external source) the data structure selection may be based on non-typical data and consequently may produce inefficient programs. Alternatively, tuning programs for particular data set may be considered as an advantage since it can select more efficient data structures when the trial data sets are typical for the program.

We now introduce two different techniques for dynamic data structure selection.

Cost Analysis using a Single Run

To accurately estimate the time cost of using a data structure for an argument index the run time properties of the argument index are needed. Properties such as the number and type of operations performed and various measures of the data sets held in each argument index can help estimate the cost.

Values collected at run time are entered into formulas which model the time complexity of each operation. To approximate the time complexity of a data structure's operations, each operation has been executed many thousands of times in isolation[1] for a variety of data sets and operation properties. The formulas for each operation were inferred by varying a number of parameters and applying regression analysis. The parameters necessary to estimate the time complexity of operations are:

[1]Programs generating statistics on data structure operation times were given large volumes of memory (150MB) upon initialisation to avoid excessive overhead from memory allocation and run time garbage collection. Code was added that ensured no other processes were active during the experiments to avoid skewed results.

- the average size of the data sets (used for the flexible array, both lists, the balanced tree and the hash table)
- the average log_2 of the size of the data sets (used for the balanced tree)
- the average of the maximum and minimum values in the data sets (used for the flexible array)
- the look operation's success rate
- the insert operation's duplicate rate

After experimenting with different data sets, significant differences in efficiency became apparent when storing ordered data compared with randomly distributed data. As a consequence, detecting when an index stores ordered data is necessary for the model to be accurate. Therefore the two additional parameters used are:

- the proportion of inserts adding a maximum value
- the proportion of inserts adding a minimum value

Special case formulas are required to estimate the costs associated with ordered data sets. The proportion that each formula (i.e. that for ascending data, descending data and random data) contributes to the total cost is given by the proportion of each insert type (i.e. maximum value inserts, minimum value inserts and intermediate value inserts).

The model which estimates time costs for random data sets is given in Table 7.4. Tables 7.5 and 7.6 give any new formulas necessary to model the costs of ascending and descending data sets. For ascending and descending data sets, delete operations were timed removing the minimum and maximum elements in the set, respectively. This is the typical behaviour of triggered programs.

Some of the cost formulas given in Table 7.4 may strike the reader as peculiar and may require explanation. The time to scan through all values in a sorted list is consistently longer than for an unsorted list despite the code being identical for both. This is attributed to the caching effect (described in Appendix B) that favours the unsorted list structure. Another feature of the model is that most data structures are more efficient performing look operations that succeed rather than a failing operation. This is due to the additional (sometimes exhaustive) searching required. Similarly, the time for each insert decreases as more duplicate values are inserted due to the additional object creation that occurs when a new value is inserted. The `max` function (distinct from the Max variable used in Tables 7.4, 7.5 and 7.6) is necessary to model the initial capacities of hash tables and flexible arrays.

The empty data structure is not included in Table 7.4. The empty data structure is used only when no operations (or no useful operations) are performed on an index. The number and type of the operations gathered from a program run is not a conclusive indication of all operations possible for an index. In some cases the measured program run may not have exercised all the operation types in the program. Selecting an empty data structure for an argument index where valid data is stored will compromise the correctness of the program. To avoid this, static set-based data structure selection is used

Data Structure:	look	scan	insert
Flexible Array	0.19	$0.10 + 0.60N + 0.21 * \mathtt{max}(10, Max, (Max - Min))$	$0.80 - 0.53D$
Unsorted List	$0.24 + (0.089 - 0.039R)N$	$0.50 + 0.38N$	$1.2 + (0.044 - 0.022D)N$
Sorted List	$0.17 + 0.067N$	$0.54 + 0.38N$	$1.2 + 0.047N$
Balanced Tree	$0.12 + (0.15 - 0.023R)(log_2 N)$	$32 + 0.36N(log_2 N)$	$2.0 - 1.6D + (0.55 - 0.070D)(log_2 N)$
Hash Table	$0.8 + (0.0021 - 0.0021R)N$	$8.6 + 0.78 * \mathtt{max}(75, N)$	$1.0 + (0.41 - 0.41D)(log_2 N)$

Data Structure:	delete	minimum	isEmpty
Flexible Array	0.35	$0.28 + 0.13\frac{\mathtt{max}(10, Max, (Max - Min))}{N}$	$0.14 + 0.093\frac{\mathtt{max}(10, Max, (Max - Min))}{N}$
Unsorted List	$0.35 + 0.054N$	$0.40 + 0.17N$	0.083
Sorted List	$0.52 + 0.078N$	0.27	0.083
Balanced Tree	$5.5 + 0.97(log_2 N)$	$0.31 + 0.056(log_2 N)$	0.083
Hash Table	1.7	$5.9 + 0.36N$	0.28

Cost values are microseconds (μs).
N represents the number of elements stored in the data structure.
Max is the maximum value stored in the data structure.
Min is the minimum value stored in the data structure.
R is the lookup success rate between 0 and 1
D is the duplicate insert rate between 0 and 1
$\mathtt{max}(A, B, ...)$ is a function which selects the maximum value of its arguments.

The machines used for measurements were AMD Athlon(TM) XP 1600+ with 256MB of main memory and 256 KB of cache, running Linux version 2.4.17 with the Java compiler from Java(TM) 2 SDK, Standard Edition Version 1.3.1 and the Java HotSpot(TM) Client VM (build 1.3.1).

Table 7.4: Instruction cost formula using random data for cost analysis after a single run.

Data Structure:	insert	delete
Flexible Array	$0.80 - 0.45D$	-
Unsorted List	$1.2 - 0.93D + 0.044N$	0.35
Sorted List	$1.1 - 0.85D$	$0.35 + 0.16N$
Balanced Tree	$2.0 - 1.6D + (0.82 - 0.15D)(log_2 N)$	$4.3 + 0.85(log_2 N)$

Table 7.5: Amendments to instruction cost formula in Table 7.4 for ascending data sets.

Data Structure:	insert	delete
Flexible Array	$0.67 - 0.34D$	-
Unsorted List	$1.2 - 0.93D + 0.044N$	0.35
Sorted List	$1.1 - 0.85D + 0.094N$	0.35
Balanced Tree	$2.0 - 1.6D + (0.82 - 0.15D)(log_2 N)$	$4.3 + 0.85(log_2 N)$

Table 7.6: Amendments to instruction cost formula in Table 7.4 for descending data sets.

to assign empty data structures to such argument indexes since this approach considers all the operations in the program's source code. Run time statistics are used to select data structures for the remaining indexes.

With the time cost formula of each operation established, data is collected and entered into the model. To gather run time statistics a new version of the program is produced by the SDSL compiler. The new program counts the number of operations of each type performed on an argument index by incrementing an integer variable immediately before the operation. In addition, a count is made of the number of successful look operations. The number of inserts that attempt to add duplicate, maximum and minimum values to an index are also recorded by searching the index before the insert operation is performed. (Note that this additional search does not affect the program's operation count.)

The new version of the program also examines the program's argument indexes regularly (before each new trigger tuple is extracted from the Δ set) to determine properties of the data sets. Using special purpose query functions that can recursively traverse the index structure, a record of the number of values (N), the log of the number of values (log_2N), and the maximum (Max) and minimum (Min) values are determined for each index. Due to nesting of argument indexes within the index structure, many instances of an argument index can exist at one time. In these cases these values are averaged over all instances. Finally, when the program terminates, the average of each measurement is calculated. An alternative strategy could examine each index after an operation is performed on it. This alternative strategy may produce more accurate statistics from the indexes but has not been pursued further due to time constraints.

The statistics collected at run time are entered into the models in Tables 7.4, 7.5 and 7.6. The time cost of each operation is found using the argument index's average N, log_2N, Max, Min, R (look operation's success rate) and D (insert operation's duplicate rate) values. Where applicable, the costs of random, ascending and descending data sets are combined in the proportion that each type of insert occurs. The cost of each operation is multiplied by the number of times the operation is actually performed. The total cost estimate of a data structure in a particular argument index is found by summing the cost estimates of all its operations. Finally, when the time cost has been estimated for all data structures, the data structure with the lowest estimate is selected. This process is applied to all argument indexes independently. Figure 7.3 formalises this process.

In theory, this selection technique is capable of accurately selecting efficient data structures since the model takes into account many factors of an executing program. However it is possible to select inefficient data structures if the time cost formulas given by Tables 7.4, 7.5 and 7.6 do not represent the run time behaviour accurately. Moreover, if the N, log_2N, Max and Min values collected at run time do not adequately represent each index then data structure selection may be mislead. This can occur when there are multiple instances of an index, each holding very different data sets, whose individual run time properties are lost through averaging.

The time cost formula given in Tables 7.4, 7.5 and 7.6 are machine specific. To maintain accuracy when migrating this data structure selection technique to other platforms additional benchmarks and analysis are required to discover

$$
\begin{aligned}
selected(index) &= argmin_{w \in DS}\ totalCost(index, w) \\
totalCost(index, ds) &= \textstyle\sum_{o \in Ops}\ (performed(index, o) * cost(ds, o, index)) \\
cost(ds, o, index) &= ascendingInserts(index) * ascendingCost(ds, o, index) \\
&\quad + descendingInserts(index) * descendingCost(ds, o, index) \\
&\quad + randomInserts(index) * randomCost(ds, o, index)
\end{aligned}
$$

Let DS be the set of all available data structures.
Let Ops be the set of all data structure operations.
Let $performed(A, B)$ denote the number of B operations performed on index A at run time.
Let $ascendingInserts(A)$, $descendingInserts(A)$ and $randomInserts(A)$ be the proportion of inserts that add a maximum, minimum and intermediate values to index A, respectively.
Let $ascendingCost(A, B, C)$, $descendingCost(A, B, C)$ and $randomCost(A, B, C)$ be the time cost of performing operation B on data structure A, using parameters from index C, from the ascending, descending and random data set models. (This is equivalent to table lookups on Tables 7.4, 7.5 and 7.6 where the equations are solved by substituting the parameters of index C).

Figure 7.3: Formula for selecting a data structure using data gathered from a single run.

each formula. However, depending on the chosen platform, the existing formulas could be used and may make a reasonable choice of data structures. Currently there is no frame of reference for such migration and clearly more research is required to determine how universal the time cost formulas are.

Data structure selection using cost analysis has been suggested in [70] for the programming language SAIL. The SAIL compiler uses a time cost model similar to that in Figure 7.4, albeit for a different set of instructions. Sample executions are used to determine the frequency of each operation, however the parameters of the data sets are found by interrogating the user. Although no explanation is given, the motivation for this interaction may be that recording the properties of actual data sets may yield misleading results, and that users are more likely to know the parameters of each data set in the SAIL programs. The latter is not true for Starlog programs because the nesting of argument indexes change the parameters of an argument's data sets. The system described in [70] does not extend the data structure model for ordered data sets – perhaps because these do not often occur or are not detected in SAIL programs.

Regression Analysis

An alternative to estimating how data structures contribute to the total run time is to measure the effect of each data structure using different runs. By recording the run times of a program using different data structures and applying regression analysis, the effect of a data structure used for an argument index can be inferred. In general, the more runs performed the more accurate the measurement is.

To measure the run time of a compiled program a new version of the program is produced by the SDSL compiler. The new version starts a timer shortly after initialisation and stops it immediately before termination[2]. The run times and data structures used in each argument index are output to a predefined file.

To determine the contribution of each data structure to the total run time the data structures are varied. Depending on the chosen analysis scheme, the data structures implementing each argument index are either randomly chosen, or one argument index is varied while the others are left constant. The process of assigning data structures to argument indexes and measuring the run times that result is repeated until sufficient data is gathered. Empty data structures are not assigned to argument indexes in this way as attempting to hold data in these affects the correctness of the program. Instead argument indexes that do not perform any useful combination of instructions are discovered using set-based data structure selection and implemented by empty data structures.

As shown in Figure 7.4, each run of the program produces a run time together with a description of the data structures used for each argument index. In this figure and throughout this chapter the abbreviations *FA* is used for a flexible array, *UL* for an unsorted list, *SL* for a sorted list, *BT* for a balanced tree and *HT* for a hash table.

To allow the contribution of each data structure to be modelled by a linear formula, each data structure in each argument index is represented by a numeric variable where '1' indicates the use of the data structure and '0' when the data structure is not used. However, because each of these variables is a linear combination of the others (e.g. `FA = 1-UL-SL-BT-HT`), one variable can be removed without loss of information. Consequently some variable representing a data structure in each argument index is always set to '0' in every record – even when the data structure is used. The data structure whose variable is consistently '0' is the reference that all other data structures will be measured against. The use of reference data structures makes the linear model more robust since it does not have co-linear attributes, and means that fewer records are required to infer coefficients.

To demonstrate regression analysis, the two argument indexes given in Figure 7.4 use the flexible array as a reference and so this data structure is always represented with a '0'. All variables representing the argument index's data structures are '0' when the flexible array is used.

When records such as those in Figure 7.4 are analysed using linear regression (where the run time is the subject), coefficients of each argument index's data structure variables are calculated. Each coefficient corresponds to the cost of using the associated data structure for the argument index. The times found from the coefficients are relative to the reference data structure where a negative coefficient indicates the associated data structure is faster than the reference data structure. For example, when linear regression is applied to the records in Figure 7.4 the result is the following equation.

$$\begin{aligned} time = {} & 0 * \texttt{Index0\_FA} - 400 * \texttt{Index0\_UL} - 1900 * \texttt{Index0\_SL} - 800 * \texttt{Index0\_BT} - 800 * \texttt{Index0\_HT} \\ & +0 * \texttt{Index1\_FA} + 100 * \texttt{Index1\_UL} - 400 * \texttt{Index1\_SL} + 1000 * \texttt{Index1\_BT} - 600 * \texttt{Index1\_HT} \\ & +3400 \end{aligned}$$

To minimise the run time generated from this equation, the data structure with the lowest coefficient is selected for each argument index. In this example, the

[2]It is necessary to record times from inside the program as the initialisation times of Java programs vary, making external timing difficult.

Argument Index 0					Argument Index 1					
FA	UL	SL	BT	HT	FA	UL	SL	BT	HT	RunTime
0	1	0	0	0	0	0	0	1	0	4000ms
0	0	1	0	0	0	0	0	1	0	2500ms
0	1	0	0	0	0	0	0	0	0	3000ms
0	0	0	0	1	0	0	1	0	0	2200ms
0	0	0	0	0	0	1	0	0	0	3500ms
0	0	0	1	0	0	0	0	0	1	2000ms
0	0	0	0	0	0	0	1	0	0	3000ms
0	0	0	1	0	0	0	0	0	0	2600ms
0	0	0	0	1	0	0	0	0	1	2000ms
0	0	1	0	0	0	1	0	0	0	1600ms

Figure 7.4: Example records used in regression analysis for two indexes.

most efficient data structure for argument index 0 is the sorted list (with a coefficient of -1900) and argument index 1 is best implemented by a hash table (coefficient of -600). Notice that the selected combination of data structures is not in the original sample.

A criticism of this approach is that linear regression assumes the input variables are independent. In spite of using a reference data structure, the binary variables representing the use of each data structure in each argument index are slightly correlated with the other variables from the argument index. More precisely, when one data structure variable is set to '1' all other variables relevant to the argument index are set to '0'. After analysis it was found that any two data structure variables have a R-squared value of less than 0.0625 rather than 0.0 that results from completely independent variables. Although correlation does affect the accuracy of the linear regression models produced, such a low value is considered insignificant for our purposes. Moreover, we will see that the linear models generated are accurate in practice.

To select efficient data structures for the argument indexes of a program requires executing the program multiple times. In general, the more run times collected, the more accurate the cost estimates will be. Of course, executing the program multiple times can be very inconvenient for the programmer so it is often useful to know the minimum number of times a program should be run in order to make a reasonable selection. An important consideration is that the number of variables increases with the number of argument indexes in a program. For the regression analysis to distinguish the contribution of each variable, the number of records must be more than the number of independent variables (other sources claim that the number of records should be at least ten times the number of independent variables to generate accurate results but here only the minimum number of records is sought). A record produced from a trial execution contains $DS * Indexes$ variables (excluding the run time) where DS is the number of data structures and $Indexes$ is the number of argument indexes where this data structure selection is performed (this excludes those argument indexes which are assigned empty data structures). However variables associated with reference data structures are redundant since their values are always '0'. Having a reference data structure for each argument index means that we can reduce the minimum number of required records to $((DS-1)*Indexes)+1$. This number of records is adequate when the data structures used in each run are carefully chosen so that the presence of each is equally represented. But when data structures are randomly selected it is recommended that more records are used. This is because equal representation of each data structure cannot be guaranteed when a random selection of data structures is made. From previous experience it is recommended that at least $DS * Indexes$ records are generated when timing randomly selected data structures.

The requirement of running a program multiple times during compilation has a number of disadvantages. The most obvious is the additional compile time which can make this selection technique impractical in many situations. However another disadvantage of regression analysis is that it requires each run of the program to be identical except for the data structures implementing each argument index. This technique becomes inaccurate when timing each run is imprecise (usually due to very short run times) or when other processes use resources in an inconsistent manner. Moreover, programs which contain random elements or programs which use other external elements that change over time are problematic. For example, a quick sort program which uses a random element for its pivot values, or whose data is generated externally may have different characteristics from run to run. To avoid such problems even more trial runs are necessary to find the average-case characteristics of the program.

Programs that are most unsuitable for regression analysis are those which require user interaction. A program that needs users to interact with it will require the the same input for each run. To repeat this process a minimum of $((DS-1)*Indexes)+1$ times is not only tedious but may be impossible – for example real-time games that rely on reflexes. To avoid this problem without changing the program's source code, external scripts might be used to mimic the actions of a user.

Data structure selection using regression analysis also has many advantages. First, this approach scales well. If a rapid compile time is not essential then data structure selection can be improved using more program runs. If optimal data structures are desired then this approach can execute the program for all combinations of data structures (although this is seldom feasible for programs with more than a few argument indexes). Another advantage is that the regression analysis is automatically configured to the machine it is operating on. This is an attractive feature for Starlog which, when compiled to Java, is already platform independent. Finally, this technique prioritises argument indexes where the choice of data structure makes the greatest difference to the total run time. Therefore, with only the minimum number of runs, the most efficient data structures are usually found for the most significant argument indexes.

7.3.3 Data Structure Selection at Run Time

In cases where selecting efficient data structures at compile time is difficult, it is possible to delay the choice until run time. Selecting data structures at run time simplifies the compilation process and can select more efficient data structures than compile time approaches when the run time properties inferred by the compiler are not accurate. However selecting data structures at run time incurs additional overhead such that the resulting program is unlikely to be optimal.

To allow the choice of data structure to be delayed until run time, argument indexes of a program are implemented by *dynamic data structures* that are capable of changing their underlying implementation. The dynamic data structure is implemented as a 'wrapper' object whose data may be stored in a flexible array, unsorted list, sorted list, balanced tree or hash table. Initially dynamic data structures store data in an unsorted list however a more sophisticated approach could initialise these to an implementation selected by a compile time method. At run time, when the dynamic data structure detects its performance is sub-optimal, the underlying data structure is re-targeted for improved efficiency. To ensure data integrity when changing implementations, all elements stored in the original data structure are copied into the new data structure.

To determine when the dynamic data structure's underlying implementation is sub-optimal, statistics are collected. These statistics include the actual number of elements in the data set, the maximum and minimum elements inserted, the number of each type of operation performed, the success rate of look operations and the duplicate rate of inserts. Using these statistics this technique estimates the time costs of each data structure using formulas developed for single run cost analysis (for now only Table 7.4 is used because accurately calculating the cost of ordered data sets is considered too expensive). The cost estimates also take into account overhead incurred from copying existing data from one representation to another. The data structure with the lowest cost estimate is considered the most efficient.

Run time overhead is incurred whenever the underlying data structure changes due to the reinsertion of data. However the process of recording and analysing statistics is another source of inefficiency. Whenever an operation is performed, at least one integer value is incremented so that the frequency of operations of each type is recorded. During insert and delete operations the number and properties of elements stored in the data structure are maintained. Furthermore, many arithmetic operation are required to estimate the performance of each data structures using the run time statistics.

Precautions are made during run time data structure selection to avoid thrashing. Time cost estimates for all data structures are recalculated only when the number of elements doubles or halves. This reduces the possibility of the dynamic data structure oscillating between two or more implementations, and reduces the frequency of implementation changes as the number of elements to be reinserted increases.

Deciding when run time data structure selection should be used instead of a compile time approach is difficult. Run time selection is useful when the run time properties of an argument index are either unknown, can not be inferred or can not be adequately represented at compile time. For example, if 50 is the average number of elements in an argument index measured at run time, this value will not accurately represent a data set when one instance of the argument index holds 3 elements on average while another holds 97. However to detect such inadequacies is difficult. Static data structure selection techniques especially have no means to measure the accuracy of the assumptions made at compile time. This means that it is best for the programmer to decide when to use run time data structure selection.

Due to the increased overhead associated with run time data structure selection, this technique is considered a secondary approach to compile time selection. This means it should be employed when the programmer considers the data structure selections made at compile time to be inefficient or too specific to the trial runs analysed by the dynamic selection techniques.

7.4 Evaluation of Automatic Data Structure Selection

In [80] it has been stated that asymptotic worst-case or average-case analysis of indexing techniques used when theorem proving is "not a very realistic enterprise". Based on a survey of indexing techniques they go on to say "the only reasonable method is to apply a statistical analysis of the empirical behaviour of the different techniques on benchmarks corresponding to real runs of real systems on real problems"[3]. Following this recommendation, the various data structure selection techniques used have been evaluated using real Starlog programs.

Appendix C gives seven example programs and the results obtained by the data structure selection techniques described in this chapter. These are the Hamming number program, prime number program, shortest path, Pascal's triangle, transitive closure, game of life, and the N-queens program. The seven example programs in Appendix C are sufficient to evaluate the automatic data structure selection techniques. To begin with, the index structures vary in complexity from programs with only one argument index to complicated programs with 13 or more. In total, the data structure selection techniques are applied to 75 individual argument indexes. By examining the tables of data collected during runs of each program, the combinations of operations performed on individual argument indexes are shown to be extremely diverse, as are the data sets held in each. By comparing the performance of the data structure selection techniques across all the example programs a number of conclusions are drawn.

When all example programs in Appendix C are considered, the average run time resulting from set-based selection is 185% slower than the fastest observed run time. A normalised average has also been calculated as an alternative means of comparison. The normalised average is a measure of run times with respect to the best and worst run times produced for each experiment. Normalised average values are interpreted on a linear scale between 0% and 100% where 0% is the fastest (best case) run time and 100% is the slowest (worst case) run time. When using set-based selection the normalised average is 6%. Set-based selection chose very inefficient data structures for the two shortest path programs. The reason for these poor selections are partly caused by the set-based approach being unable to choose a flexible array – a data structure necessary for maximum efficiency in the shortest path programs. The effectiveness of this technique indicates that automatic data structure selection based only on the program source is feasible, and supports the results of experiments using the SETL compiler that are briefly discussed in [98].

Relatively speaking, static cost analysis frequently selected inefficient data structures. On average run times are 203% slower than the fastest run times. The normalised average indicates that static cost analysis produces programs with run times 22% along the scale between the fastest (0%) and slowest (100%) run times. The poor selections are the result of assumptions made by static cost analysis being inconsistent with the actual run time properties. For example,

[3]However [80] also notes that one should not draw too many conclusions from such analysis.

the assumption that argument indexes hold around 100 elements is almost always contradicted by the actual average number of elements. In particular, a disproportional number of argument indexes hold only one element at run time due to functional relationships between arguments. Also, assuming that a flexible array will have a 50% density is almost always inaccurate. Generally, when data sets from Starlog programs are stored in an array they are either very dense (at or close to 100%) or very sparse (less than 10%) but infrequently in between. Without knowing the density of a data set, the efficiency of a flexible array can not be accurately estimated by this selection strategy. This problem is highlighted by the Hamming number program where, by assuming 50% density, the flexible array selected by static cost analysis is actually the worst case. To avoid such inaccuracy the flexible array should not be assigned by static cost analysis. However this will result in using sub-optimal data structures when the flexible array actually is the most efficient. Because of the problems associated with accurately representing a typical data set, static cost analysis is not considered a useful technique for data structure selection.

When a single run of a program is used to collect run time properties, in general, the resulting data structures selected are very efficient. The average run time resulting from this selection technique is 36% slower than the fastest run time. The normalised average is 5%. For the two programs where this technique selects relatively inefficient data structures (prime number and N-queens), the cost estimate model is inaccurate. Yet inaccuracy of the cost estimate model is inevitable because each formula (and each parameter entered into a formula) is only an approximation of the actual behaviour. To improve the reliability of data structure selection using this technique the time cost formulas could be improved (perhaps by using special case formulas when only one element is stored) or by taking more run time parameters into account (e.g. the variance of the data set). Alternatively the method by which run time parameters are computed could be refined. The results presented in this thesis would be of interest to researchers of the SAIL compiler which also uses a single run and cost analysis to select data structures. The more advanced properties of the cost models described in this chapter (such as the detection of ordered data sets and failure rates of instructions) could be incorporated into SAIL's data structure selection technique to improve accuracy. However the results of our experiments have shown that – even with many parameters – modelling the run time behaviour of data structures is difficult.

Regression analysis consistently selected efficient data structures for all the example programs. When all regression results are considered the average run times are 39% slower than the fastest observed run times. The normalised average for regression-based selections lies at 4% between the fastest and slowest run times. Unlike the other data structure selection techniques, regression analysis prioritises argument indexes which make the greatest contribution to the total run time. Consequently, less dominant indexes may be assigned arbitrary data structures when their contribution to the total run time is lost in the margin of error. As shown in Appendix C, to improve selection using regression requires analysing more combinations of data structures. Using specifically chosen combinations of data structures has mixed results when compared with random combinations. For most example programs, analysing chosen combinations of data structures makes a better selection than when random combinations are analysed. This is impressive since there were far fewer chosen records than randomly selected records. However, in other cases, when the measured run times have a significant margin of error, there are not enough specifically chosen records to make an accurate selection. To improve regression analysis, both chosen and random combinations of data structures should be used. By first measuring the run times of the chosen combinations and then adding records from random combinations, the chosen combinations quickly find an efficient selection of data structures while the random combinations should compensate for the margin of error.

In all cases data structures selected at run time are not as efficient as those selected at compile time. This is evident from the example programs where the run times generated using dynamic data structures were seldom near optimal. Indeed, in half of the examples, data structure selection at run time produced the worst run times of any of the automatic selection techniques. However a trend apparent in these examples is that the performance of the dynamic data structure deteriorates as the complexity of the index structure increases. This is attributed to complex examples with more argument indexes having fewer elements in each argument index. Consequently the frequency that a data structure's performance is evaluated by the dynamic data structure is increased, resulting in additional overhead. It seems that evaluating performance when the number of elements doubles or halves is too frequent when there is a low number of elements. However further research to find the best conditions for run time data structure selection has not been attempted in this work.

7.5 Conclusions

The choice of data structures can greatly affect the efficiency of a program – in some of our example programs the run times varied by more than a factor of 20 depending on the data structures. Rather than use a generic data structure which will often be sub-optimal, or rely on manual selection which is problematic for high-level languages, we allow the compiler to choose the most appropriate data structures.

This chapter has discussed and evaluated a variety of techniques for automatic data structure selection. The first techniques presented are designed to select data structures during compilation of a program, and can be divided into two categories – those which requires the program to be executed (dynamic), and those which do not (static). By executing a program more information can be collected about its run time characteristics and, in general, more efficient data structures are likely to be selected. A key difference between static and dynamic selection techniques is that the data structures chosen using dynamic techniques will be specifically tuned for the program's trial runs.

For comparison, the possibility of delaying data structure selection until run time has been investigated. Although this approach is useful in some situations (e.g. when the data sets held in an index vary greatly) the overhead of choosing data structures at run time had a significant impact on performance. It is believed that alternative strategies for data structure selection at run time may reduce this overhead, however it will never be removed completely.

The compile time data structure selection techniques presented here are effective for the example Starlog programs. None of the selection techniques consistently chose inefficient data structures. Based on the evaluation in this chapter a number of recommendations can be made regarding data structure selection. The most accurate, consistent and robust technique of those evaluated is regression analysis – it seems there is no substitute to running a program to determine its bottlenecks. Although this technique does not always choose the optimal data structure for every index, it always prioritises those indexes that most affect the run time. Regression analysis is also scales well, allowing users to decide the degree of optimisation they require for their programs. For these reasons regression analysis is the primary technique choosen for selecting data structures for Starlog programs.

The greatest disadvantage of regression analysis is that it requires multiple program runs and therefore is unsuitable in situations where a fast compile time is required. A fast compile time is particularly desirable when programs are being developed where the efficiency of a program is secondary to its correctness. It is recommended that the simpler set-based approach is used during program development since it is the more effective of the two static selection techniques. It is also immune to changes in the data sets which frequently occur during program testing and, unlike regression analysis, can be applied to non-terminating programs. After using the set-based selection technique to ensure that a program is correct, the more sophisticated regression analysis technique may be used (via compiler switch) to assign data structures prior to the release of the program.

The data structure selection techniques did not attempt to optimise the memory usage of a program. However all the techniques presented in this chapter could be modified to prioritise memory usage instead of run time. Chapter 8 describes such modifications in the future work section.

Automatic data structure selection is the last stage in the compilation of Starlog programs. In the next chapter the efficiency of compiled Starlog programs is compared with that of equivalent hand-coded programs.

Chapter 8

Conclusions and Future Work

The Starlog programming language is of interest because it is logically pure and abstract. Its purity extends to all language facilities, including negation, input and output, aggregation, and destructive assignment. Therefore logic programming using Starlog is distinguished from Prolog whose logical semantics have been compromised to include these facilities. Logical purity allows programs to have a declarative semantics and to be proven correct. Abstraction allows programmers to be oblivious to the execution details of their programs. Thus, abstract languages tend to be easier to learn, represent programs more efficiently and usually allow easier interpretation of programs by programmers than lower level languages. In Starlog programs run time data is abstracted using relations. Relations have seen widespread success in the field of databases and have been accepted by users as a natural form of data abstraction since Codd's landmark paper in 1970. However abstraction in programming languages is usually achieved at the price of run time efficiency.

This thesis has shown how it is possible to compile Starlog programs to an executable form (Java). The compilation process can be completely automated. Moreover, the compilation process presented here is focused on efficient execution at all stages.

Four important features of the Starlog compilation process that improve efficiency are (1) the use of Triggering Evaluation, (2) the use of static index structures, (3) automatic construction of efficient instances of these static index structures, and (4) the automatic selection of efficient low-level data structures to implement the static index structures. As well as providing efficiency to Starlog these features are also applicable to a variety of other fields.

As shown in Chapter 2, Triggering Evaluation can optimise the evaluation of rule systems which obey a stratification order. In [84] the use of top-down evaluation in the XSB system was justified by its raw speed. Top-down evaluation is used for XSB in spite of decidability problems when solving negated queries and infinite recursing queries. If the efficiency of bottom-up evaluation is improved then it may become more competitive with top-down techniques and allow such problems to be avoided. Triggering Evaluation improves run time performance of bottom-up evaluation when programs are monotonic and obey a stratification order. In general, the more complete the stratification order, the more optimisation Triggering Evaluation allows. Triggering Evaluation would be of interest as an addition to forward-chaining theorem-provers (such as SATCHMO [69]) and planners (such as TLPLAN [7]) as well as the monotonic deductive database systems Aditi, COL, LogicBase, DECLARE, Hy+, LOLA, and Starburst [91].

The static nature of Starlog's index structures allows an efficient implementation of programs in three ways: (1) Starlog's index structures can be statically defined in the target language, (2) the order that arguments and predicate names are indexed can be optimised at compile time to improve access to tuples and (3) when programs search or update instances of Starlog's index structures specialised code is used for trigger/tuple matching and for rule evaluation.

Further reduction of run time overhead is possible when Starlog index structures are automatically constructed using techniques from Chapter 3. The reordering of arguments from each predicate can result in an unbounded reduction in searching. Reordering arguments can improve the efficiency of any program that uses discrimination trees (such as XSB programs) and could be achieved in other systems using a source transformation. By sharing index nodes between different predicates within an automatically constructed index structure, the size of the index is reduced, and when related predicates share indexes, further optimisations are possible (as described in Chapter 5). These optimisations would be applicable to other systems which use static discrimination trees.

Chapter 7 argues that both manual selection and generic data structures produce less efficient programs than automatic selection for abstract languages. The automatic data structure selection techniques presented in Chapter 7 are also useful to any languages or applications where data structures are abstracted. These include logic and functional languages, most modern imperative languages, and all database applications.

Automatically selecting implementations for data structures after their usage has been determined is an approach uncommon in compilers. To achieve this in an object-oriented target language, the Starlog compiler uses a very general data structure interface which specifies the necessary methods that the parent program may require. Based on the number and type of the methods which are used, a previously developed implementation can be selected. This strategy allows automated systems to avoid premature selection of implementations which, after optimisations have been applied, may have very different preconditions or usage patterns. Allowing the compiler to delay its selection of implementations is useful for any facilities that can have multiple implementations – not just data structures.

8.1 Comparison of Efficiency

To rate the efficiency of compiled Starlog programs Table 8.1 compares the example Starlog programs from Appendix C with hand-coded implementations. The hand-coded programs were collected from web-sites which are unrelated to the Starlog project and so should be reasonable subjects for comparison. Note that these hand-coded programs may not be the most efficient implementations available, however they represent the typical efforts of programmers to solve each problem. Due to the differences in programming style between Java and

Starlog, some of the algorithms used by the hand-coded programs are very different from the Starlog programs.

To avoid the contentious issues of comparing the run time behaviour of programs written in different languages, all the hand-coded implementations are written in Java and executed on the same platform as that used to evaluate the Starlog programs. For consistency between the Starlog implementations and the hand-coded examples, some of the hand-coded programs have been modified to remove user interfaces and other superfluous features.

The efficiency of hand-coded programs is a bold target for compiled code to strive for. In general the hand-coded programs have used customised evaluation techniques and/or data structures which, by the general purpose nature of the Starlog compiler, are unavailable to compiled Starlog programs. Table 8.1 summarises the differences between the Starlog and hand-coded programs. A few details about the significant differences are discussed here.

The complexity of the Starlog prime number program is an order of magnitude larger than the hand-coded example (the exact complexity is difficult to determine). This is because the Starlog prime number program investigates N^2 integers when inferring multiples rather than the N multiples that are investigated by the hand-coded program.

Both the hand-coded shortest path program and the transitive closure programs use an array of integers for an adjacency matrix to determine paths in the graph. Therefore these require very little overhead for storing data.

To generate Pascal's Triangle the hand-coded program uses a ragged, two dimensional array of integers resulting in very little overhead.

The hand-coded Game of Life holds only the current and previous generations of cells in two integer arrays whereas the Starlog implementation maintains all previous generations. However, like the Starlog implementation, the hand-coded program is capable of representing cells on an infinite board.

When the N-Queens program is hand-coded in Java a top-down, depth-first search is used to find all solutions. Using recursive calls and in-place updates on a single integer array that represents the chess-board, the hand-coded program is very efficient when compared to the Starlog solution. The Starlog N-Queens program performs a breadth first search and generates copies of the board at each node of the search tree.

In spite of such differences in algorithms and data structures there are some promising results in Table 8.1. The efficiencies of the Hamming Number, Shortest Path and Game of Life programs when written in Starlog are competitive with the equivalent hand-coded programs (and in two cases are even more efficient).

Although these results give an indication of the performance of Starlog programs, these results do not conclusively prove that Starlog programs are as efficient as equivalent programs in other languages. To prove this would require analysing the complexity of Starlog programs and comparing that with alternative implementations. Such an evaluation is beyond the scope of this thesis.

To further improve the efficiency of programs produced by the Starlog compiler (so that more Starlog programs may become competitive with their hand-coded equivalents) the compiler could be extended in a number of ways. Several potential extensions are described in the next section as future work with comments about which of the example programs they would benefit.

Program	Starlog		Hand-Coded Java			
	Code Size	Run Time	Code Size	Run Time	Source	Differences
Hamming Numbers	3 Rules	12 ms	168 lines	71 ms	http://gauss.ececs.uc.edu/Users/ Franco/Streams/stream_explain.html	-
Primes	3 Rules	11,894 ms	36 lines	108 ms	http://www.macs.hw.ac.uk/ pjbk/cs3/3PD1/java/Nsieve.java	A,P,M
Shortest Path (Random Graph)	1 Rule	145 ms	142 lines	119 ms	http://www.cs.rpi.edu/ puninj/XMLJ/ projects/stuproj/turned/XMLProject/	A,P,M,F
Shortest Path (Chain Graph)	1 Rule	89 ms	140 lines	346 ms	http://www.cs.rpi.edu/ puninj/XMLJ/ projects/stuproj/turned/XMLProject/	A,P,M,F
Pascal's Triangle	3 Rules	55 ms	24 lines	0.7 ms	http://occs.cs.oberlin.edu/faculty/jdonalds/150/lecture21.html	P,M,F
Transitive Closure (Random Graph)	1 Rule	464 ms	33 lines	45 ms	Adapted from [99]	A,P,M
Transitive Closure (Chain Graph)	1 Rule	242 ms	32 lines	11 ms	Adapted from [99]	A,P,M
Game of Life (Rabbit Pattern)	3 Rules	728 ms	124 lines	178 ms	http://www.aceshardware.com/articles/technical/ java_vs_c/files/infilife.java	A,P,G
Game of Life (Traffic Light)	3 Rules	298 ms	127 lines	159 ms	http://www.aceshardware.com/articles/technical/ java_vs_c/files/infilife.java	A,P,G
N-Queens	7 Rules	266 ms	96 lines	2 ms	http://webster.cs.uga.edu/ arford/NQ/NQueens.java	A,P

Key to Differences:
A - Algorithmic differences which affects the complexity
P - Arrays of primitive types used in hand-coded program
M - Single data structure equivalent to combined Δ and Γ in hand-coded program
G - Garbage collection of redundant relations in hand-coded program
F - Functional relationships exploited in hand-coded program

Table 8.1: Comparison between Starlog programs and similar hand-coded Java programs.

8.2 Future Work

Although Starlog programs can be compiled with the techniques given in this thesis, there are additional optimisations and extensions that could further improve the quality of the compiled code.

8.2.1 Detecting Functional Relationships

In Chapter 3 it was stated that dynamic argument indexes could be replaced by single values if functional relationships were proven between arguments held in the index structure. Programs that use specialised data structures that hold only one element would be more efficient because operations performed on a single value are more efficient than those performed on more sophisticated data structures.

It is reported in [33] that finding functional relationships is undecidable in the general case. However the statistics gathered from data sets during data structure selection (Section 7.3.2) may be useful for identifying potential functional relationships. Using the number of elements held in each argument index during run time, argument indexes that consistently hold one element are likely to be functionally dependent on the previously indexed arguments. A special purpose functional dependency analysis tool could then be used to confirm the relationship.

This technique for detecting potential functional relationships is similar to the dynamic invariant detection performed by the Diakon tool [41, 42]. Diakon uses program runs to infer program properties which may be later proven correct. For Starlog, the invariant to be proven is that each argument index holds exactly one element.

Alternatively, specialised data structures which hold one element can be used to automatically garbage collect redundant tuples (see the next section for more garbage collection issues). In some programs (e.g. those which implement destructive assignment) only the most recent values added to an argument index are ever accessed. In these cases there is no reason to maintain previous values and any previous value stored can be overwritten with a new value. Although it is difficult to determine when previous values of an argument are irrelevant to a program, some occurrences could be detected using pattern matching within the Starlog source code to find rules that constitute the destructive assignment program (or a close approximation).

The detection of functional relationships would improve the efficiency of two of the example programs described in Table 8.1. The Shortest Path program would benefit from the observation that there exists at most one minimal cost value between any two nodes. Similarly, Pascal's Triangle would benefit when it is known that for each valid row and column in the triangle there is exactly one binomial coefficient. In both cases, the use of specialised data structures would improve performance.

8.2.2 Memory Usage and Garbage Collection

When optimising Starlog programs the time complexity took priority over the memory usage. This is usually the preference for applications which do not require a prohibitive amount of memory to function. However the ability to control the memory usage is essential when large data sets are produced or when running Starlog programs on platforms with low memory. The two approaches to reduce the memory used by Starlog programs are to (1) reduce the memory requirements of Starlog index structures and (2) perform garbage collection.

Starlog index structures occupy less space when more predicates share index nodes. To increase the sharing of index nodes, the arguments from each predicate can be ordered so that arguments with compatible types appear at the same level of the index structure. This is an alternative heuristic to that described in Chapter 3 which prioritises efficient searching. A single Starlog index structure can then be constructed using the existing algorithm from Chapter 3 that merges independent index structures.

A second approach to reducing the memory requirements of Starlog index structures is to reduce the memory requirements of each argument index. Chapter 7 gave techniques to select data structures which minimised the time complexity of programs, however all of the techniques could be modified to select data structures based on their memory requirements ([70] discusses this modification for one selection technique). To minimise the memory requirements the time cost estimate tables are replaced with memory cost estimate tables or, in the case of regression analysis, programs are created which can measure their own memory usage. With the exception of the set-based selection technique, all the data structure selection techniques can prioritise both the run time and memory requirements when given a user-defined space/time tradeoff.

Garbage collection is another technique for reducing the memory requirements of programs. Moreover, garbage collection in Starlog is essential to reduce memory requirements to that of other languages so that, for example, Starlog may be used for server applications which must guarantee reasonable memory usage over long time periods. [2] defines *garbage* as "storage that a program allocates but can not refer to". Unreferenced data occurs in the Java implementation of Starlog programs when program and index variables are reclaimed, and when sections of a Starlog index structure instance are pruned, however such data is reclaimed periodically by Java's automatic garbage collection process.

Yet there is a another form of data which is not referred to by Starlog programs but can not be reclaimed by Java's automatic garbage collector. Tuples in Γ become garbage when their presence or absence can no longer contribute to the production of new tuples. Typically, tuples become garbage when the constraints within all rules makes them irrelevant. These can be safely deleted from Γ without affecting the correct evaluation of the program. However, in other cases, tuples never become garbage because they contribute to one or more rules for the life of the program.

To describe garbage collection of tuples in Γ it must be understood how they are used. Tuples in Γ are accessed as the result of non-trigger goal queries. We have seen in Section 2.3.6 that tuples from predicates that are never queried as non-trigger goals are always garbage in Γ and can be safely omitted from this set. Although research continues on general purpose garbage collection of tuples, no satisfactory solution has been discovered which is both thorough and practical. The only way to conclusively prove which tuples are used by a program is to run the program and so far our attempts to approximate the usage of tuples is limited to a few simple cases. Clearly more research is necessary to find a practical solution to the garbage collection problem.

An alternative strategy would be to allow the programmer to annotate their programs with the garbage collection properties of tuples. For example, data regarding the persistence of each predicate could be consulted to determine which tuples should be removed from Γ. Unfortunately such an approach exposes the programmer to complex details about the execution of their programs, breaking the declarative semantics of Starlog. Moreover, programmers can make erroneous assumptions regarding the garbage collection of tuples and so may compromise the correctness of their program.

Without garbage collection, there are some programs that use unboundedly more memory than when garbage collection is used. Garbage collection would reduce the memory requirements and run time of the Game of Life program. For the Game of Life to function only the current and previous generations of cells are required at any time. Therefore all preceding generations can be safely discarded. On average, our example programs maintain 75 generations of cells that could be discarded. The Pascal's Triangle Starlog program could also benefit from garbage collection since only the previous row of binomial coefficients is necessary to calculate the next row. (Although this is an optimisation not implemented in the hand-coded Pascal's Triangle program.)

8.2.3 Sharing Index Nodes between Δ and Γ

Chapters 3, 4 and 5 discussed the possibility of sharing dynamic argument indexes between the Δ and Γ sets. In these chapters it was concluded that, although the index structure would be smaller and moving tuples from Δ to Γ would be more efficient, the shared argument indexes can not be specialised for one type of access. Instead, to minimise the cost of all operations, these argument indexes would have to be implemented by generic data structures, which ultimately may result in less efficient programs (see Chapter 7).

However the advantages of sharing index nodes between Δ and Γ could be exploited with more analysis. By gathering the frequency and type of operations performed on each argument index and properties of the data set held in each argument index, cost estimates can be made for programs with shared index nodes and for those without. The run time properties of each argument index can be found using techniques employed during data structure selection. By comparing the cost estimate of the programs with argument indexes shared between Δ and Γ and the cost estimate for programs where Δ and Γ are distinct, the most efficient version can be selected.

By sharing argument indexes between Δ and Γ the Starlog versions of the Prime Number, Shortest Path, Pascal's Triangle and Transitive Closure programs would become closer to the hand-coded versions identified in Table 8.1. Yet it is unclear whether such modification would result in faster run times for these programs. For example the Shortest Path program written in Starlog is already competitive with the hand-coded implementation and combining argument indexes may actually reduce its efficiency.

8.2.4 Skipping Duplicate Detection

One of the requirements of the SDSL insert instruction is that it searches the dynamic argument index for the value to be inserted before it actually inserts the value. This requirement maximises sharing within argument indexes. However such duplicate detection adds to the run time overhead of the operation. To avoid the extra overhead it may be possible to use a different version of the insert instruction which skips searching for duplicate values in an argument index (we shall refer to this instruction as *insert-unique*).

An insert-unique instruction can only be used in place of an insert instruction when it is impossible to insert a duplicate value. To prove this property requires sophisticated static analysis: Given a set of rules with the same head predicate, it must not be possible for corresponding arguments in the heads to overlap values. Furthermore, any single rule which includes non-deterministic searches must not be able to reproduce the same values for an argument in the rule's head. This requires global analysis of all rules in the program using theorem proving techniques to ensure there is not an overlap of argument values in a argument index.

The effect of skipping the duplicate detection phase when inserting values into an argument index is different for each data structure. For example, testing for the presence of a value requires almost no overhead for an array, however requires an exhaustive search when performed on an unsorted list.

8.2.5 Automatic Use of Multiple Indexing Orders

One of the advantages of using Starlog's index structures that was discussed in Chapter 3 was the use of multiple indexing orders for predicates. The benefit of multiple indexing orders is that different index paths can be specialised for different modes of access. This can result in an unbounded reduction of the time complexity of searches (e.g. an operation that has $O(N^i)$ complexity may become $O(N)$ using a different indexing order). However, because each tuple is stored in multiple ways, the cost of inserting a tuple into the index structure is proportional to the number of index paths travelled. Consequently, the automatic construction of Starlog's index structures described in Chapter 3 does not allow multiple index orders because the degree of optimisation (if any) is difficult to predict.

Multiple indexing orders for predicates can be used to optimise programs if we can predict both the number of searches which would benefit and the number of additional inserts required. The analysis required to make such predictions is similar to that used to automatically select data structures (e.g. the number of occurrences of each instruction in the SDSL program is an indication of which operations are performed at run time, or the number of operations performed at run time can be measured). All of the techniques described in Chapter 7 could be modified for this purpose. If running the program is required to estimate the number of searches which would be improved and the number of additional inserts required by multiple indexes, a rudimentary Starlog index structure (such as one which does not use multiple indexes) would be sufficient during this analysis.

8.2.6 Parallel Execution

Two reasons to use parallel execution in a system are (1) because the nature of the application demands it (e.g. for multiple, independent, real-time processes) or (2) to improve time efficiency. Reason (1) is not important for Starlog because the computations performed in rules can be easily interleaved, simulating parallel processes. Therefore, in this section we are only concerned with using parallel execution to make Starlog programs faster.

The declarative nature of Starlog programs makes few requirements on the order that rules and goals are evaluated. So long as the stratification order (usually a partial order) is obeyed then program evaluation will be correct with respect to the well-founded semantics. However to implement Starlog programs in SDSL the partial order is arbitrarily strengthened when converted to a series of SDSL code block (see Chapter 5) to simplify serial execution on a Von Neumann architecture. The implementations of Starlog programs in SDSL (and subsequently in Java) follow this total order to evaluate rules and goals.

However if explicit parallel execution is desired then there is no need to totally order the evaluation of Starlog programs. When Starlog programs are translated to SDSL, parallelisation can occur in three places: (1) the solving of goals in the body of a rule, (2) the evaluation of rules activated by the same trigger tuple, and (3) the selection and use of multiple minimal tuples in Δ as trigger tuples. Parallel evaluation of goals (1) is an application of *And-Parallelism*[1] [50, 92]. The remaining opportunities for parallelisation of Starlog programs ((2) and (3)) correspond to *Or-Parallelism* [50, 92]. Note that this is only a preliminary assessment of how parallelisation can be exploited by Starlog programs and further instances may become clear in time. In fact there is ample scope for another PhD thesis on the parallelisation of Starlog.

When compiled to Java, each parallel code block can be evaluated by a separate thread. The use of threads allows the operating system to dispatch jobs to different processors as they become available, potentially making the whole program run much faster [38].

8.3 Summary

New programming languages have the potential to introduce greater productivity to each new generation of programmers. Traditionally this productivity has manifested itself in either the efficient and clear representation of programs or through a program's efficient run time behaviour. In Starlog we value both these properties. The design of the Starlog language allows declarative and abstract representations of programs, whereas the compilation techniques given in this thesis are focused on efficient run time behaviour. Algorithms in Starlog are often much different from the equivalent algorithms in other languages. In some cases the Starlog programs are more elegant and readily understandable by programmers, leading to faster development and better quality programs.

The goal of developing compilation techniques for Starlog has been achieved. Moreover, the compilation of Starlog is focused on efficient execution of programs. Results given in this thesis indicate that some (but not all) compiled Starlog programs are more efficient than equivalent hand-coded programs. However it remains to be proven if such results are typical. Nonetheless, this thesis has shown that run time efficiency does not have to be sacrificed when compiling a declarative and abstract language.

[1]The Bernstein condition [12] would restrict or complicate the parallel solving of goals that share unbound variables.

Appendix A

Starlog Syntax Reference

Starlog programs consist of rules, to define the logic of the program, and stratification priorities, to specify the stratification order (and therefore the behaviour) of the program.

Stratification Priorities

The stratification order for the program is specified using stratification priorities. By convention the stratification priorities are included at the beginning of a Starlog program. The order of stratification priorities is irrelevant. There are two types of stratification priorities.

Stratification Orders of Predicates

Each predicate in a program appears in at most one stratification priority. A stratification priority for a predicate takes the form "`stratify` *Pred Order.*".

Pred represents an ***abstract tuple definition*** where all the arguments of the predicate are represented by locally unique variable names. (Any arguments irrelevant to the stratification order can be represented by the anonymous variable "_".)

Order is a list of comma-separated terms that represents the stratification order of the *Pred* predicate. Elements in the list are either variable names corresponding to arguments in *Pred* or constant symbols. Elements earlier in the list are more significant than those later in the list.

Any predicate not used in a stratification priority is stratified before those which do appear in a stratification priority.

Stratification Orders of Constants

Constant symbols used in the predicates' stratification priorities are explicitly ordered. Constants are ordered using a "`stratify` *Const*1 << *Const*2" statement where the constant symbol *Const*1 is ordered before constant symbol *Const*2.

Rules

A Starlog rule is represented as "*H* `<-` *B*." where *H* is the rule's head and *B* is the rule's body. When *B* is empty the rule is referred to as a *fact* and is represented by only the rule's head as "*H*.".

Rule Heads

H is a non-variable term whose functor and arguments specify the tuples that the rule produces as output. Tuples from some "built-in" predicates perform additional operations or side-effects when generated as the head of a rule. Current built-in predicates include:

- `print(N)` which outputs the value `N` to the standard output.
- `print_string(Str,N)` outputs `Str` to the standard output and is stratified by `N` when necessary.
- `input_request(Prompt,N)` outputs a command-line prompt specified by `Prompt` and waits for input. These tuples are stratified by `N` when necessary.
- `input_request(Prompt,N,Key)` is the same as the previous predicate however the `Key` argument is used to associate the request with its input.

Rule Bodies

B – the body of a rule – is either a single goal or a comma separated conjunction of goals (e.g. "*Goal1*, *Goal2*, *Goal3*."). There are three types of goals: built-in operations, positive goals, and negated goals.

Built-in Operations

Built-in operations in the body of a rule perform operations on the rule's local variables. The set of available built-in operations is given below. For the most part the semantics of these built-in operations coincide with those in Prolog. (The "`C is sqrt(A)`" operation for Starlog differs from that of Prolog because Starlog produces both the positive and negative square roots of its input.) Because built-in operations are evaluated in a left-to-right order, the binding orders of variables must be considered when writing rules.

```
  C is A      A =\= B     C is -A    C is A + B
C is A - B  C is A * B  C is A / B  C is A // B
   A > B       A < B       A >= B      A =< B
                   C is sqrt(A)
```

`A` and `B` represent either ground variables or constants.
`C` represents either a ground or free variable, or a constant.

Positive Goals

Positive goals are predicates whose arguments are either constant symbols or variables. A positive goal is satisfied when a tuple is produced by the program that matches with the predicate.

Negated Goals

Negated goals are predicates encapsulated in a "`not(...)`" structure. Negated goals are satisfied when no tuples that match the predicate can be produced by the program. Variables occurring in negated goals which do not occur elsewhere in the rule are automatically existentially quantified. Built-in operations may be performed on these existential variables inside the "`not(...)`" structure.

Local Variables in Rules

There are some constraints on the use of local variables in rules. All variables occurring in the head of a rule must also occur in a positive context in the body. That is, variables in the head must be either an argument in a positive goal or the output of a built-in operation in the body.

All variables used to determine the stratification order of negated goals must either appear in a positive context in the body, or be constrained by built-in operations that ensure the negated goal is stratified before the rule's head.

Stratification of Rules

All rules in a Starlog program must be strongly stratified with respect to the program's stratification priorities. To ensure this some rules require built-in operations (such as $A>B$) which explicitly define orderings for arguments and, consequently, defines the stratification order between goals and the heads of rules.

Comments

Comments in Starlog follow the Prolog convention. A comment is preceded by a '%' symbol and continues until the end of the line.

Backus-Naur Form

The BNF of the Starlog language is given in Figure A.1. Non-terminal symbols `<variable>`, `<constant>` and `<functor>` are undefined but correspond to their usual definitions in Prolog. The BNF is incapable of describing constraints on the use of variables and the stratification of rules that were previously described in this section. The use of comments is also omitted from this definitions as, like Prolog, comments can occur in almost any context.

```
<program> ::= { <stratification_priority> } { <fact_or_rule> }
<stratification_priority> ::= <predicate_ordering> |
                              <constant_ordering>
<predicate_ordering> ::= ``stratify'' <abstract_tuple_definition>
                         <element_list> ``.''
<abstract_tuple_definition> ::= <functor> [``('' <variable>
                                {``,'' <variable> } ``)'' ]
<element_list> ::= ``['' [ <argument> {``,'' <argument> } ] ``]''
<constant_ordering> ::= ``stratify'' <constant> ``<<'' <constant> ``.''
<fact_or_rule> ::= <fact> | <rule>
<fact> ::= <predicate> ``.''
<rule> ::= <predicate> ``<-'' <rule_body> ``.''
<rule_body> ::= <goal> {``,'' <goal> }
<goal> ::= <built_in> | <positive_goal> | <negated_goal>
<positive_goal> ::= <predicate>
<negated_goal> ::= ``not('' <predicate> {``,'' <built_in> } ``)''
<predicate> ::= <functor> [``('' <argument> {``,'' <argument> } ``)'']
<argument> ::= <variable> | <constant>
<built_in> ::= <equals> | <not_equals> | <negated> | <addition> |
               <subtraction> | <multiplication> | <division> |
               <int_division> | <greater> | <less> |
               <greater_equal> | <less_equal> | <square_root>
<equals> ::= <argument> ``is'' <argument>
<not_equal> ::= <argument> ``=\='' <argument>
<negated> ::= <argument> ``is -''<argument>
<addition> ::= <argument> ``is'' <argument> ``+'' <argument>
<subtraction> ::= <argument> ``is'' <argument> ``-'' <argument>
<multiplication> ::= <argument> ``is'' <argument> ``*'' <argument>
<division> ::= <argument> ``is'' <argument> ``/'' <argument>
<int_division> ::= <argument> ``is'' <argument> ``//'' <argument>
<greater> ::= <argument> ``>'' <argument>
<less> ::= <argument> ``<'' <argument>
<greater_equal> ::= <argument> ``>='' <argument>
<less_equal> ::= <argument> ``=<'' <argument>
<square_root> ::= <argument> ``is sqrt('' <argument> ``)''
```

Figure A.1: BNF grammar of Starlog.

Appendix B

Data Structure Implementations

In this appendix we present the six data structures that are used to implement argument indexes. Each of the following sections describes the design of a data structure and, when relevant, gives an insight into how each of the *insert*, *look*, *scan*, *delete*, *minimum* and *isempty* operations are performed. A commentary on the time complexities of each valid operation is provided. When describing complexities of operations N will refer to the number of items in the data structure, unless otherwise stated. For the source code of these implementations see `www.cs.waikato.ac.nz/~rjc4/`.

B.1 Empty Data Structure

Empty Data Structures do not hold any argument values. Instead these objects implement those nodes in an index structure on which no operations are performed. Although all operations in the `Node` interface have default implementations in the empty data structure, calling these has no effect. Empty data structures are required for the leaf nodes of an index structure so that all argument indexes point to objects of type `Node`. The empty data structure class is often extended to hold labelled branches or boolean values for specific index nodes.

B.2 Unsorted List

An unsorted dynamically-linked list is a very simple way to store data of any type. The implementation encapsulates each data value in an object with a "next" handle to locate the next element in the list. As usual with linked lists, the "next" handle of the last element in the list holds a `null` value.

Inserting a value into an unsorted list is not as efficient as other list implementations due to the duplicate detection phase. To ensure that a value to be added does not already exist, the new value is compared with those elements already in the list (giving a $O(N)$ complexity). If the new value is not a duplicate it is inserted onto the end of the list.

To search for a specific value in an unsorted list requires a linear search ($O(N)$) where an equality test is performed on the sought value and each value in the list until they are equal, or the end of list is reached (indicating failure). When the value is present in the list then, on average, half the elements will have been processed.

An iterator for an unsorted list maintains a "current position" handle pointing to the next element to be returned. Each time the `nextNode` method is called the "current position" handle is incremented to the next element in the list. When the "current position" element becomes `null` all elements have been visited. Consequently, scan operations on lists are very efficient since they require only one variable update per element.

The deletion of an element from an unsorted list initially involves finding the parent of the element to be deleted ($O(N)$) and reassigning the "next" handles to skip over the deleted node. Java's automatic garbage collection process will reclaim any non-referenced list element.

To find the minimum element in an unsorted list requires an exhaustive search of all elements ($O(N)$). The value of each element in the list is compared with the current minimum value, which is updated whenever a lower value is encountered.

Finally, to test if an unsorted list is empty a null test is performed on the list's start (or root) handle. This is a very inexpensive operation requiring only one variable test ($O(1)$).

B.3 Sorted List

In contrast to unsorted lists, sorted lists maintain the list invariant where the value of an element is greater than the values of all elements later in the list (i.e. values in the list are descending[1]). This has a number of advantages over unsorted lists which improves the performance of some operations.

On average, search operations have less complexity for sorted lists than unsorted lists. Although the worst case for a search operation remains $O(N)$, the average case of searching for a value which is not present in the sorted list will require processing only half the elements. Searches can be truncated when the current value in the sorted list becomes less than the sought value due to the sorted list invariant. At each node visited an extra test is necessary on to detect when truncation can occur. This improvement is also used to optimise the duplicate detection phase of insert operations.

Although the complexities of the scan, delete and empty operations are equivalent to the unsorted list, the minimum operation is much improved. Instead of performing an exhaustive search on all elements, the sorted list maintains a handle to the last (minimum) element in the list. Therefore finding the minimum element in a sorted list is a single variable fetch ($O(1)$).

One interesting disadvantage a sorted list has over an unsorted list occurs when memory architecture is considered. When inserting values into an unsorted list new elements are always added to the end of the list. If consecutive inserts are performed the elements will occupy adjacent memory locations. The

[1]A descending list is used because experience has shown that these are frequently more efficient than ascending lists for arguments in Starlog programs. This is because arguments are more often processed from the lowest to the highest value in Triggering Evaluation.

advantage is that when a memory block containing an unsorted list element is cached there is a greater chance that elements from the tail of the list will be cached also, making processing the list faster. When considering sorted lists, because new elements are often inserted between existing elements their memory layout is likely to be much more random and, as a result, caching a memory block has less effect. The effect of memory caching is seen when modelling time complexities for some of the data structure selection strategies.

B.4 Balanced Binary Tree

Binary trees are used frequently in applications where large number of records are stored which must be searched efficiently. To improve the robustness and reduce the worst case complexities of binary trees, balancing is performed. The balanced binary tree implementation used for Starlog's argument indexes is a red-black tree derived from that described in [99]. (Note that this implementation does not require parent pointers.)

The complexity of searching for or inserting a value into a balanced binary tree is $O(logN)$ in spite of the balancing performed on each node visited during the insert.

To scan through all values in the balanced binary tree is expensive when compared to other data structures. To scan through values in a tree structure usually requires a stack holding the sequence of parent nodes visited in the traversal so backtracking can occur when all nodes in a sub-tree have been visited. However, due to ongoing experiments involving modularly stratified programs, an iterator was developed that would continue to work correctly when new data is added to the data structure[2]. Using the stack approach, when new data is added to the tree and balancing occurs the parent nodes held in the stack may not exist in the same locations. As a result the scan may fail to return some values or repeatedly return others. To avoid such situations the balanced tree iterator maintains a reference to the last *value* returned. A call to the `nextNode` method searches the tree for the next largest value. The result is an in-order traversal that is unaffected by re-balancing. The searching that occurs with every call to `nextNode` makes scanning a balanced tree expensive – for each element in the tree $O(logN)$ nodes are processed.

Deletion of a node in a red-black tree involves replacing the deleted node with its successor node and traversing both up and down the tree to restore balance. Deletes have $O(logN)$ complexity. However, to maintain the sequence of parent nodes visited when searching for the node to be deleted, a stack is used. Due to the overhead of objects in Java, adding elements to and accessing elements in the stack are expensive operations.

To find the minimum element stored in the balanced binary tree, $O(logN)$ nodes are visited during repeated "left" branch searches.

[2]If an insert is performed on a data structure while a scan is in progress there is no requirement that the new data be returned by the scan, however all data that exists during the entire scan must still be returned.

B.5 Hash Table

The implementation of the hash table used here is known as a *double hash table.* A double hash table initially indexes records on a primary hash of their key field. However when a collision occurs a secondary hashing function generates a fixed increment for each additional probe. In this way the hash table avoids problems with clustering and, on average, requires fewer probes than linear probing [99]. This property allows the computational complexity to be less dependent on the data set and therefore is easier to model.

The hash table itself is a fixed sized array of elements where each element holds the actual value stored (the key field) and the relevant sub-index. Initially the table can hold 101 elements.

To insert an element into the table the two hashing functions calculate the initial location and the increment for each probe, and step through the table. Assuming a perfect hashing function where no collisions occur, the complexity of an insert is $O(1)$ [72]. If ineffective hashing functions are used that always return the same value the complexity rises to $O(N)$. In practice, carefully chosen hash functions (like those implemented for this data structure) result in a complexity between these two extremes. To avoid the hash table becoming saturated with data it will double[3] its size when it becomes more than 75% full. All values in the original table are individually inserted into the new table. Although resizing the table becomes less frequent as its size increases, each additional resize is more expensive than the last. The result is that insert operations which require a table resize have an $O(N)$ complexity but when this is amortised over all insert operations a typical insert is $O(logN)$.

To find an element in a hash table using a perfect hashing function is $O(1)$. However if the hashing functions are ineffective then the complexity becomes $O(N)$. In practice, the average number of probes required is usually a low constant if the table is regularly resized before it is saturated.

To scan through all elements in a hash table requires visiting all locations in the table. With a table size doubling at 75% capacity, the table will be between 37.5% and 75% full (except when there are less than 38 elements). Therefore, in most cases, finding each additional element in the table can be approximated by a constant ($O(1)$).

Deleting an element from a hash table is achieved by reassigning the element's table location to a special deleted record. The deleted record is used instead of a null value to allow known value searches to skip over it and continue to probe locations later in the table. An insert operation overwrites the deleted record with a valid record if one is encountered during probing. When resizing a table the deleted records are ignored so they do not appear in the new table. The complexity of a delete operation is proportional to the search process which finds the relevant record ($O(1)$ to $O(N)$ depending on the hashing functions).

Because the data in a hash table is unordered, finding the minimum element requires an exhaustive search. Complexity is related to the number of table locations and the number of elements in the table. If we assume that the number of table locations is roughly proportional to the number of elements then the complexity is $O(N)$.

[3]To allow the size of the table to be "probably prime" (i.e. have few multiples) the size of the original table is doubled and then decremented. This avoids infinite-loops when probing.

A hash table is empty if it does not contain any records. To test for emptiness the table is searched for an entry that is not null and does not contain an empty object resulting from a delete operation. Because the search can terminate when the first element is found the complexity of an empty operation is proportional to the number of elements in the hash table ($O(M - N)$ where M is the size of the table). However, when the hash table is resized regularly so that it is always between 37.5% and 75% full, the complexity of testing for an empty data structure is approximated with a constant ($O(1)$).[4]

B.6 Flexible Array

Arrays have been a part of programming languages since they were introduced to COBOL (and later versions of Fortran) in the 1950s. Arrays are still common in modern programming languages due to their simplicity and efficient access properties. In terms of data access and update speed, bit arrays have unparallelled efficiency making them an excellent choice for many applications.

However bit arrays have several preconditions that can make them unsuitable in many situations. First, the index values must be of integer type making them unsuitable for indexing non-integer values. The second precondition is that arrays in Java are indexed from zero making it impossible to use negative integers as a index. Finally, the size of an array must be known in advance. In some situations it is difficult to predict the maximum size of the data set until it is generated. With the exception of the first, these preconditions can be overcome by adding "wrapper" code around an array which controls how and where data is stored (similar to a Java vector). The resulting data structure is called a Flexible Array.

The flexible array implementation is initialised to hold up the values between 0 and 9. When values are inserted that can not be indexed in this array the array is resized to accommodate the new value. (The new array is twice as large as it needs to be to hold the new value in order to optimise inserting other values of a similar magnitude.) To allow negative integers to be stored, an offset value is added to the array's indexess. (Note that this offset is at most 0. This is because the offset is only used to accommodate negative integers and not to improve the performance of an array holding only high positive values. Whenever the offset value changes the array is resized and the original values are copied into their appropriate location in the new array.)

One disadvantage of the array data structure (flexible or otherwise) is that memory usage is very inefficient for sparse data sets [13]. That is, when there are large gaps between stored records the array requires memory for all the unused values between these records. When the difference between the maximum and minimum value in the array becomes sufficiently large there will not be enough memory, irrespective of the number of actual records stored in the array. Moreover, when memory is sufficient to hold the sparse data set in the ar-

[4]An alternative approach would be to maintain a count of the number of elements in the table. This differs from the count which is used to detect when resizing should occur as this count includes the deleted records held in the table. Although such a change would improve the isEmpty operation it would increase overhead in the insert and delete operations. Since isEmpty operations are rarely performed in programs compared to inserts and deletes this approach was not implemented.

ray, operations that involve searching multiple locations (i.e. scans, minimums and isemptys) are very inefficient.

The insert operation performed on the flexible array initially compares the new value (modified by the offset value) with the upper and lower bounds of the array. If the new value falls outside this range then an array resize is required. The process of resizing the flexible array involves initialising a new array and then copying the contents of the original array in to the new array (optimised using a built-in array copy method). When the size of the array is sufficient, and the value does not already exist, the new value (modified by the offset) is inserted into its appropriate array location. Inserting a new value into an array of sufficient size is very efficient ($O(1)$) since there is no searching required. Resizing an array is $O(M)$ where M is the the size of the original array, but the frequency of resizes depend on the data set.

To search for an item in a flexible array the value to be searched for is modified by the offset, compared with the upper and lower bounds of the array, and then used to access a location in the array ($O(1)$).

The iterator for a flexible array maintains a reference to the next location to be visited during a scan. After a record has been retrieved, this reference is updated to the next record in the array by incrementally searching through the array locations. When the end of the array is reached the scan is complete. The worst case complexity of a scan operation is $O(M)$ where M is the size of the array.

Deleting a record from a flexible array involves locating the sought value in the array and setting the object at this location to a null value ($O(1)$).

To find the minimum element in a flexible array requires searching multiple locations. Starting from location zero, all locations in the array are searched until one is found which holds a record. In the worst case the number of locations visited when finding the minimum element will be $O(M)$ with M the size of the array.

The isempty operation performs the identical search as the minimum operation to find the first array location containing a record. Consequently the complexity is $O(M)$.

Appendix C

Case Studies in Automatic Data Structure Selection

To demonstrate the effectiveness of the data structure selection techniques we present seven example programs. The programs use increasingly complex index structures from a program with only one argument index through to one with 13 argument indexes. A brief explanation of each program's purpose and origin is included before the Starlog source code is given. The performance results and the data structure selections made by each technique are given for each example program. An evaluation of the data structure selection techniques is given in Chapter 7 based on this appendix.

To strengthen the argument that generic data structures can be inefficient in Starlog programs, the run times of programs using generic data structures are included with the graphed results. In these experiments the balanced tree is considered the most general-purpose data structure available since it scales well for most operations. Therefore is used as the generic data structure.

All tuples in the example programs have integer arguments. This ensures that the flexible array, which can only hold integer values, can be used in all argument indexes. More general programs with arguments of other types would restrict the use of flexible arrays.

The example programs included in this section do not generate any side-effects. Consequently, their usefulness is questionable since they do not communicate their results to the user. However the Starlog source code (and in many cases the index structures) required for each program are simplified without side-effects. Moreover, inconsistent completion times of side-effects can distort the results of program benchmarking. If programmers wish to produce side-effects from these programs additional program rules need to be added, however the accuracy of dynamic data structure selection techniques can not be guaranteed.

C.1 Hamming Number Program

The Hamming number series was first described by R. W. Hamming and is frequently used in textbooks that teach recursive programming [119]. Hamming numbers are defined as any number that has only 2, 3 and 5 as its prime factors. More precisely:

```
% Hamming Number Program
%-----------------------------------------------------------------------
% Generates all numbers between 1 and 100,000 whose prime factors are
% only 2, 3 and 5, in ascending order.

stratify hamming(N) [N].             % Order hamming numbers on their value

hamming(1).                                   % Initial Fact
hamming(New) <- hamming(Old),                 % Generates multiples of 2
                New is Old*2, New < 100000.
hamming(New) <- hamming(Old),                 % Generates multiples of 3
                New is Old*3, New < 100000.
hamming(New) <- hamming(Old),                 % Generates multiples of 5
                New is Old*5, New < 100000.
```

Figure C.1: Hamming number program.

$$HammingNumbers = \{2^i * 3^j * 5^k \mid i \geq 0 \wedge j \geq 0 \wedge k \geq 0\}$$

What is interesting about Hamming numbers is not so much the series itself, but the method by which values are generated. The Hamming number series can be computed as follows: the first Hamming number is 1 and all other Hamming numbers are the product of a previously found Hamming number and integers 2, 3 or 5. To generate the complete series in ascending order, each Hamming number is considered in ascending order and multiplied by 2, 3 and 5. Each of these new Hamming numbers is inserted into its appropriate location in the ordered series. Because each Hamming number is greater than its Hamming factors, all Hamming numbers can be generated in order.

Figure C.1 shows the Starlog program to generate Hamming numbers to be very compact. This is because it uses Starlog's internal database and the stratification order to ensure each Hamming number is processed in order. Other programming languages require a queue to be explicitly defined making the program more complicated (see [64, 112, 119] for examples of alternative representations).

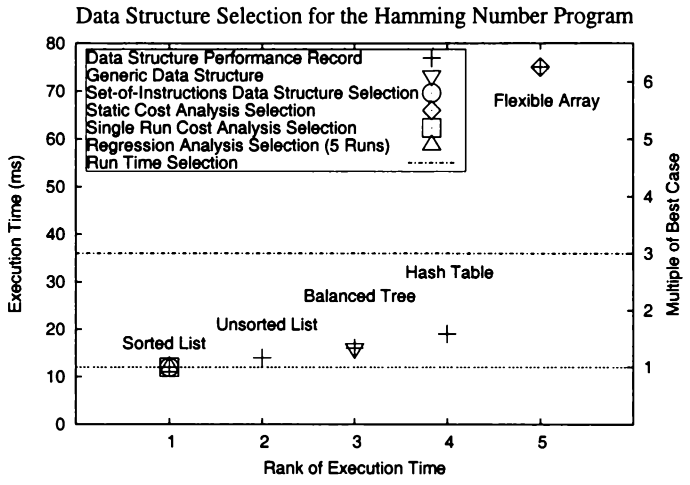
To allow termination there is an upper limit for the Hamming numbers produced. All three rules that generate new Hamming numbers will succeed only when the new value is less than 100,000. Termination is essential for dynamic data structure selection and to evaluate the performance of the program. Therefore all programs evaluated in this section terminate.

The SDSL version of this program stores all program tuples in a single index. This is because there are no non-trigger goals in any rules, making a Γ set unnecessary. In SDSL form the Hamming number program is optimised so that the three rules that share the same trigger goal are factorised together.

The performance results of the Hamming number program are given in Figure C.2. In this Figure (and in the performance results of other programs) the first table gives details of the operations performed by the program. For each operation type, the 'occurrences' column represents the number of occurrences of the operation in the SDSL program source. The number of occurrences is necessary for data structure selection using static cost analysis. The 'performed'

Index	look		scan		insert		delete		minimum		empty	
	occurrences	performed	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.
Index0	0	0	0	0	4	717	1	312	1	313	0	0

Index	Average # of Elements (N)	Average $log_2 N$	*Max* Value	*Min* Value	Successful Looks (R)	Duplicate Inserts (D)	Max Value Inserts	Min Value Inserts
Index0	53.7	5.47	46407	19655	-	56.5%	30.1%	0.1%



Worst Data Structure Selection:	Flexible Array	(75ms)
Generic Data Structure:	Balanced Tree	(16ms)
Best Data Structure Selection:	Sorted List	(12ms)
Set-of-Instructions Selection:	Sorted List	(12ms)
Static Cost Analysis Selection:	Flexible Array	(75ms)
Single Run Cost Analysis Selection:	Sorted List	(12ms)
Regression Analysis Selection (5 runs):	Sorted List	(12ms)
Run Time Selection:		(36ms)

Figure C.2: Hamming number program performance.

column gives the actual number of operations performed at run time. These values are used when making a selection with single-run cost analysis. The second table gives details about the data set and additional properties of the program's operations also required for single-run cost analysis.

The graphs show the results of data structure selection using the automated techniques described in Chapter 7. For each graph, the run times resulting from different data structures have been sorted and ranked for clarity. The two measures of run times are the time in milliseconds and the multiple of the best run time. The performance of the data structures selected using the various techniques are highlighted in the graphs with large symbols. A selection technique that chooses efficient data structures will highlight values in the lower left corner whereas a poor choice will have a run time in the upper right corner of the graph. Finally, the data structures which give the best and worst run times are given followed by the data structures chosen by the automated selection techniques.

To generate the graph in Figure C.2 the single argument index was implemented by all data structures in the library and the run time of the each program measured. The fastest data structure for this program is the sorted list which resulted in a run time of $12ms$. The slowest data structure is the flexible array which took $75ms$ to complete. From analysis of the operations performed and the data set generated during execution these results are reasonable: Although there are 312 Hamming numbers between 0 and $100,000$, the index holds an average of 53.7 elements at any one time (see Table 2 in Figure C.2). This means that data structures that scale well for large volumes of data (i.e. balanced binary trees or hash tables) are unnecessary and less efficient than lists. Also, a minimum operation is frequently performed on the index. Unsorted lists are less inefficient than sorted lists for minimum operations since unsorted lists require exhaustive searching. The flexible array data structure is very inefficient because the data set is sparse. The 53.7 elements stored have an average maximum value of $46,407$ and a minimum value of $19,655$. The minimum operations on a flexible array is inefficient since many empty locations may have to be searched before the first (minimal) element is found.

The effectiveness of the data structure selection techniques for the Hamming program is shown in the graph in Figure C.2. Set-based selection, single run cost analysis and regression analysis all choose the optimal data structure (the sorted list) from the library. The regression analysis requires at least one run time from each data structures to make any selection so it is not surprising that it makes a good selection. Static cost analysis selects the worst case data structure (flexible array) for this program. This is attributed to the Hamming number data set being inconsistent with the assumptions made during static cost analysis. This is particularly true for the density of the data set where static cost analysis assumes a 100 elements are stored with values between 0 and 200. Run time data structure selection initially used an unsorted list to store Hamming numbers, until 32 elements are stored when the implementation changes to a sorted list. However, due to the overhead of recording statistics, calculating cost estimates and copying data from one data structure to another, the resulting run time is relatively high at $36ms$.

To test the robustness of the data structure selection techniques the termination condition of the Hamming program was modified. By changing the program so that it generated Hamming numbers up to 1,000 and, in another

```
% Prime Number Program
%-----------------------------------------------------------------------
% Generates prime numbers between 2 and 10,000 by finding all multiple
% values and then using negation to find values that are not multiple
% values.

stratify num(N)   [N,num].          % Order all tuples on their arguments
stratify mult(N)  [N,mult].
stratify prime(N) [N,prime].
stratify num << prime.              % Ensure primes are stratified late
stratify mult << prime.

num(2).
num(M) <- num(N), M is N+1,          % Generate all numbers in range
          M < 10000.

mult(M) <- num(N), prime(P), N >= P, % Generate multiple values
           M is N*P, M < 10000.

prime(N) <- num(N), not(mult(N)).    % Deduce prime numbers
```

Figure C.3: Prime number program.

test, up to 400,000,000, the programs' run times were decreased and increased respectively. However the order of efficiency of the data structures remained unchanged. This is because the set of Hamming numbers has a logarithmic growth rate related to the maximum value of the set. Because the size of the data sets do not change dramatically for this range of values, the efficiency of each dynamic data structure remains in its relative position. (The flexible array becomes more inefficient as the upper limit increases due to the increasingly sparse data set – but is consistently the least efficient data structure.) Testing the Hamming number program with more extreme termination conditions (i.e. outside of the range 1,000 to 400,000,000) is impossible due to the lack of precision in program timing at the low end, and the insufficient precision of Java's integers at the high end.

C.2 Prime Number Program

The prime number program used in these experiments was introduced and discussed in Chapter 1. The Starlog source code is repeated here in Figure C.3 for reference. The index structure that stores the run time data of this program contains two argument indexes as shown in Figure C.4. To optimise this program, indexes holding the arguments of `num/1`, `mult/1` and `prime/1` tuples are automatically combined using techniques discussed in Chapter 3. Note that `mult/1` tuples do not need to be added to the Δ set because they are not used as triggers in any rules.

The results of automated data structure selection are shown in Figure C.5.

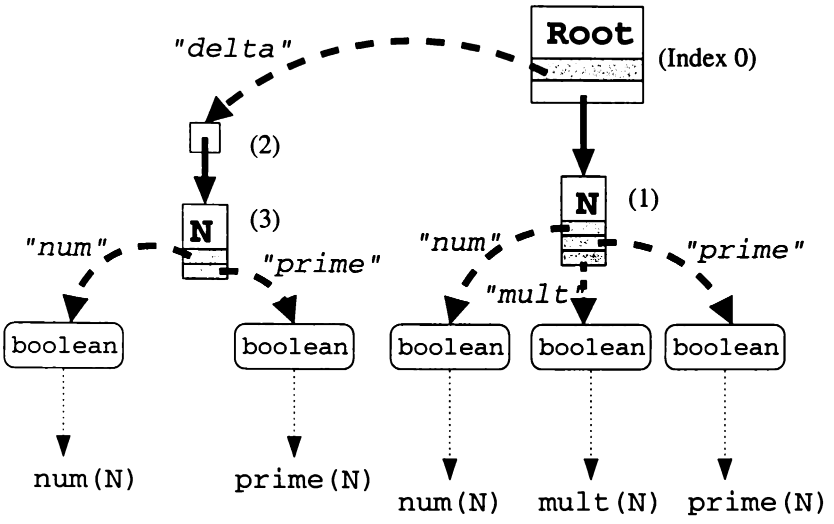


Figure C.4: Index structure schema used for the prime number program.

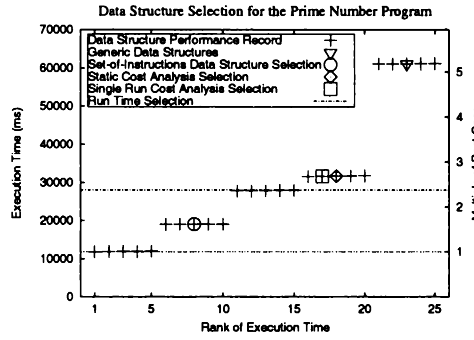
The two tables at the top of Figure C.5 indicate that the two indexes have very different properties. Index 0, which holds the arguments of tuples in the Γ set, performs look, scan and insert operations. Index 2, containing arguments of tuples in Δ, performs inserts, deletes and minimum operations. The data set held in Index 0 contains 8,611 elements on average, compared to Index 2's one element. When comparing run time statistics, the different sizes of the data sets is a significant factor. The run times produced by this program are dominated by the cost of operations on Index 0. Consequently, the choice of data structure for Index 0 is paramount for efficiency, whereas the data structure selected for Index 2 is all but inconsequential to the total run time. The dominance of Index 0 is characterised by the step-wise graphs in Figure C.5 where all run times with the same data structure for Index 0 are almost identical, irrespective of Index 2's data structure. The most efficient data structures for Index 0 are those with efficient insert and look operations, making the flexible array very effective, followed by the hash table, the sorted list, the unsorted list and finally the balanced tree. The balanced tree, whose insert and look operations are not particularly inefficient for the number of elements in Index 0, is handicapped by slow scan operations which is a secondary bottleneck.

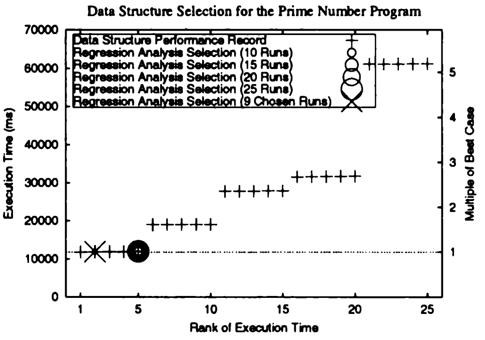
Set-based data structure selection makes a conservative choice of data structures based on only the operations in the program. By selecting a hash table for Index 0 the run time of the program is 61% slower than the fastest data structure combination. The static cost analysis makes the poorest selection of all the selection techniques (at 168% slower than the fastest data structure combination). Once again this is attributed to assumptions made by the selection technique being inconsistent with the run time properties of the program.

Using a single run to aid cost analysis also makes a poor selection (again 168% slower than the fastest combination). This technique selects an unsorted

Index	look		scan		insert		delete		minimum		empty	
	occurrences	performed	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.
Index0	2	11227	1	9998	4	28206	0	0	0	0	0	0
Index2	0	0	0	0	2	9998	1	9998	1	9999	0	0

Index	Average # of Elements (N)	Average $log_2 N$	*Max* Value	*Min* Value	Successful Looks (R)	Duplicate Inserts (D)	Max Value Inserts	Min Value Inserts
Index0	8611	13.0	9931	2.0	100%	64.6%	0.47%	0.0%
Index2	1.0	0.0	5001	5001	-	0%	100%	0.0%





Worst Data Structure Selection:	[Balanced Tree, Unsorted List]	(66,645ms)
Generic Data Structures:	[Balanced Tree, Balanced Tree]	(66,546ms)
Best Data Structure Selection:	[Flexible Array, Balanced Tree]	(11,803ms)
Set-of-Instructions Selection:	[Hash Table, Sorted List]	(19,055ms)
Static Cost Analysis Selection:	[Unsorted List, Flexible Array]	(31,652ms)
Single Run Cost Analysis Selection:	[Unsorted List, Sorted List]	(31,608ms)
Regression Analysis Selection (10 runs):	[Flexible Array, Sorted List]	(11,995ms)
Regression Analysis Selection (15 runs):	[Flexible Array, Sorted List]	(11,995ms)
Regression Analysis Selection (20 runs):	[Flexible Array, Sorted List]	(11,995ms)
Regression Analysis Selection (25 runs):	[Flexible Array, Sorted List]	(11,995ms)
Regression Analysis Selection (9 chosen runs):	[Flexible Array, Flexible Array]	(11,894ms)
Run Time Selection:		(28,022ms)

Figure C.5: Prime number program performance.

list for Index 0 because when analysing the data collected at run time it appears that the scan operations on Index 0 dominate the total run time. However, when the actual run times are compared this is not the case. This discrepancy shows the actual performance of the data structures for Index 0 does not corresponding to that the cost estimate model. This is either because the parameters gathered from the single run do not adequately represent the execution of the program (perhaps due to averaging) or the formulas themselves are inaccurate for this case.

To thoroughly test the regression analysis technique, the number of data structure combinations are varied. Using randomly selected data structures for each index, the minimum number of run times that can be analysed is 10. The number of measured runs is increased to improve accuracy. Finally, the program is run using carefully chosen combinations of data structures (where each data structure is equally represented in each index). The regression analysis performs very well and consistently selected the optimal data structure (flexible array) for Index 0. However the data structure selected for Index 2 is never optimal. This is because the contribution that Index 2 makes to the total run times is so small it is often lost in the "noise" within the data.

Using run time data structure selection, the execution time of the prime number program is 137% slower than the fastest combination of data structures. Although the data structures selected are reasonably efficient – initially an unsorted list followed by a hash table for Index 0 and constantly an unsorted list for Index 2 – the overhead of run time selection reduces efficiency.

C.3 Shortest Path Program

The shortest path program finds the shortest path between two points in a directed graph. This program is based on the shortest path program given in [45] which itself is based on Dijkstra's algorithm.

The Starlog source code for the shortest path program is given in Figure C.7. In this program a directed graph is represented as a set of weighted edges `edge(X,C,Y)` where `X` is the origin node identifier, `Y` is the destination node identifier and `C` is the cost associated with the edge. The minimum cost of reaching a node is given by `cost(X,C)` where `X` is the node and `C` is the cost incurred travelling the shortest path. To begin the program, a node is selected from where all paths will originate. This is achieved by adding a `cost/2` fact containing the originating node identifier and a zero cost value. When all possible shortest paths have been found from the origin node to other connected nodes in the graph the program will terminate. The index structure automatically generated for this program contains seven argument indexes as shown in Figure C.6.

To test the effectiveness of the data structure selection techniques, this program has been applied to two graphs. The first is a randomly defined graph with 1,000 nodes. The conventions of this graph are that each node has 20 edges originating from it to pseudo-random nodes, with cost values between 1 and 20. The second graph is a cyclic chain where each of 2,000 nodes is connected to the next.

The results of data structure selection for the shortest path program with a random graph are given in Figure C.8. Although set-based selection is de-

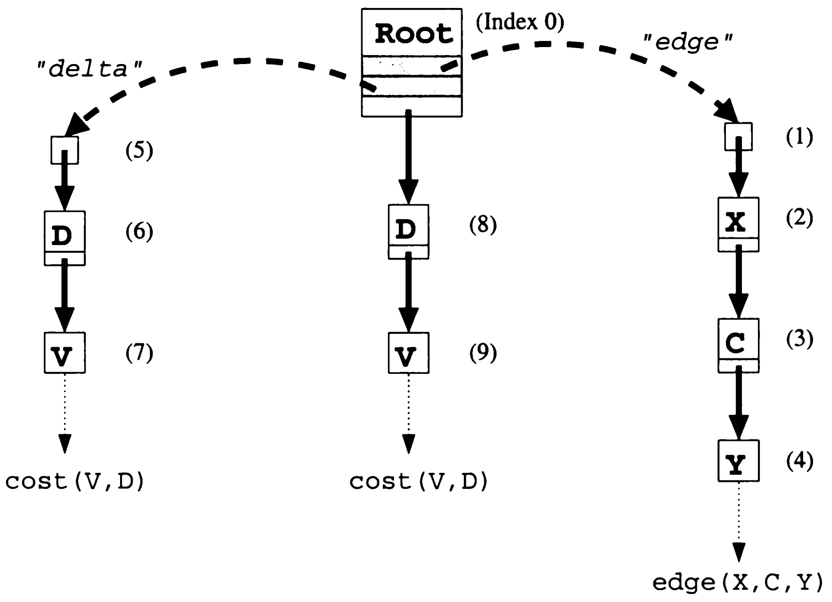


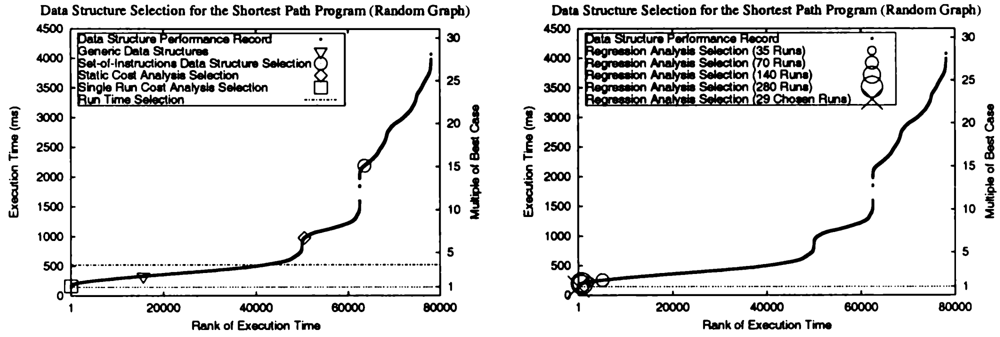
Figure C.6: Index structure schema used for the shortest path program.

```
% Shortest Path Program
%-----------------------------------------------------------------------
% Program to calculate the shortest path between a given node and any
% other nodes in a directed graph. The graph is defined as a set of
% weighted edges between nodes, represented in the program as
% edge(X,C,Y) where X is the origin node, Y is the destination node
% and C is the cost of the edge.
% The output is the minimum cost incurred to reach each connected node
% represented by cost(X,C) where X is the node reached and C is the
% cost.

stratify cost(_,C)   [C].   % Order cost tuples on their cost value.

edge(1,5,2).                % Sample graph (not used for benchmarks)
edge(1,7,3).
edge(3,2,4).
edge(3,8,2).

cost(1,0).                  % Node of origin (with zero cost)

cost(U,NewC) <- cost(V,D), edge(V,C,U),   % Find minimum cost to get
                NewC is D+C,              % to connected nodes.
                not(cost(U,W), W < NewC).
```

Figure C.7: Shortest path program.

Index	look		scan		insert		delete		minimum		empty	
	occurrences	performed	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.
Index0	1	9001	0	0	1	401	0	0	0	0	0	0
Index1	1	401	0	0	20000	20000	0	0	0	0	0	0
Index2	0	0	1	9	20000	20000	0	0	0	0	0	0
Index3	0	0	1	9000	20000	20000	0	0	0	0	0	0
Index5	0	0	0	0	2	9001	1	1326	1	1327	0	0
Index6	0	0	1	1326	2	9001	1	9001	0	0	1	9001
Index8	0	0	1	8600	1	401	0	0	0	0	0	0

Index	Average # of Elements (N)	Average $log_2 N$	*Max* Value	*Min* Value	Successful Looks (R)	Duplicate Inserts (D)	Max Value Inserts	Min Value Inserts
Index0	350	8.27	961	1.0	95.5%	0%	24.6%	0.2%
Index1	20.0	4.32	20	1.0	2.2%	99.9%	5.1%	5.0%
Index2	1000	9.97	1000	1.0	-	0%	100%	0.01%
Index3	1.0	0.0	499	499	-	0%	100%	100%
Index5	612	8.86	1274	663	-	85.3%	0.1%	11.2%
Index6	4.68	1.98	816	199	-	0%	55.0%	24.7%
Index8	1.0	0.0	147	147	-	0%	100%	100%

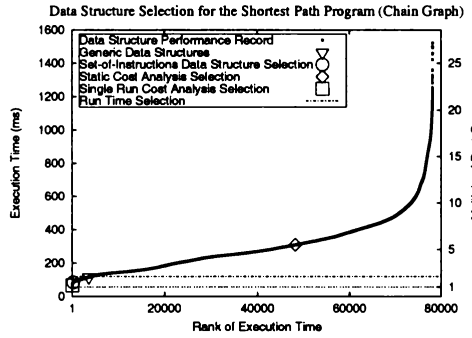


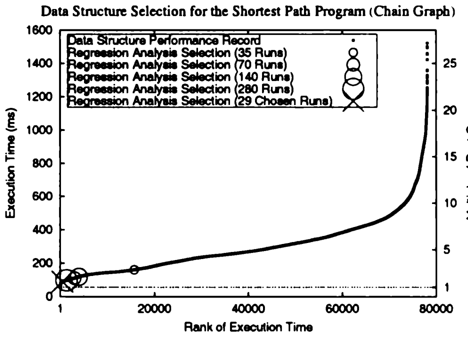
Worst Data Structure Selection:	[SL,BT,UL,FA,UL,FA,BT]	(4,076ms)
Generic Data Structures:	[BT,BT,BT,BT,BT,BT,BT]	(331ms)
Best Data Structure Selection:	[FA,FA,SL,SL,FA,SL,SL]	(145ms)
Set-of-Instructions Selection:	[HT,HT,UL,UL,SL,UL,UL]	(2190ms)
Static Cost Analysis Selection:	[FA,FA,FA,FA,FA,UL,UL]	(975ms)
Single Run Cost Analysis Selection:	[FA,FA,FA,SL,FA,UL,UL]	(164ms)
Regression Analysis Selection (35 runs):	[HT,FA,SL,UL,BT,HT,FA]	(228ms)
Regression Analysis Selection (70 runs):	[FA,FA,FA,SL,FA,HT,HT]	(254ms)
Regression Analysis Selection (140 runs):	[FA,HT,FA,SL,FA,UL,HT]	(197ms)
Regression Analysis Selection (280 runs):	[FA,SL,FA,SL,FA,UL,HT]	(199ms)
Regression Analysis Selection (29 chosen runs):	[FA,FA,SL,SL,FA,SL,SL]	(145ms)
Run Time Selection:		(523ms)

Figure C.8: Shortest path program performance (random graph).

Index	look		scan		insert		delete		minimum		empty	
	occurrences	performed	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.
Index0	1	2000	0	0	1	2000	0	0	0	0	0	0
Index1	1	2000	0	0	2000	2000	0	0	0	0	0	0
Index2	0	0	1	1999	2000	2000	0	0	0	0	0	0
Index3	0	0	1	1999	2000	2000	0	0	0	0	0	0
Index5	0	0	0	0	2	2000	1	2000	1	2001	0	0
Index6	0	0	1	2000	2	2000	1	2000	0	0	1	2000
Index8	0	0	1	0	1	2000	0	0	0	0	0	0

Index	Average # of Elements (N)	Average $log_2 N$	*Max* Value	*Min* Value	Successful Looks (R)	Duplicate Inserts (D)	Max Value Inserts	Min Value Inserts
Index0	1000	9.53	1000	1.0	0%	0%	100%	0.0%
Index1	2000	11.0	1999	0.0	100%	0%	100%	0.0%
Index2	1.0	0.0	1.0	1.0	-	0%	100%	100%
Index3	1.0	0.0	1000	1000	-	0%	100%	100%
Index5	1.0	0.0	1000	1000	-	0%	100%	100%
Index6	1.0	0.0	1001	1001	-	0%	100%	100%
Index8	1.0	0.0	499	499	-	0%	100%	100%





Worst Data Structure Selection:	[FA,UL,FA,UL,UL,UL,SL]	(1,522ms)
Generic Data Structures:	[BT,BT,BT,BT,BT,BT,BT]	(115ms)
Best Data Structure Selection:	[SL,FA,SL,SL,SL,SL,SL]	(56ms)
Set-of-Instructions Selection:	[HT,HT,UL,UL,SL,UL,UL]	(84ms)
Static Cost Analysis Selection:	[FA,FA,FA,FA,FA,UL,UL]	(309ms)
Single Run Cost Analysis Selection:	[FA,FA,SL,SL,SL,SL,SL]	(66ms)
Regression Analysis Selection (35 runs):	[UL,HT,HT,HT,SL,FA,UL]	(160ms)
Regression Analysis Selection (70 runs):	[BT,BT,SL,SL,UL,FA,SL]	(110ms)
Regression Analysis Selection (140 runs):	[BT,HT,SL,SL,UL,BT,SL]	(118ms)
Regression Analysis Selection (280 runs):	[SL,HT,SL,SL,UL,FA,SL]	(97ms)
Regression Analysis Selection (29 chosen runs):	[UL,BT,SL,SL,UL,SL,SL]	(89ms)
Run Time Selection:		(119ms)

Figure C.9: Shortest path program performance (chain graph).

signed to make a conservative choice, it makes a poor choice of data structures (1400% slower than the most efficient data structures) for this program. One explanation is that the optimal selection of data structures use flexible arrays but the set-based selection technique is not capable of assigning these to indexes, thereby forcing the use of a less efficient data structure. The static cost analysis technique also makes a poor selection (572% slower than the fastest combination of data structures) because it selects a flexible array for Index 3 which holds only one element. Using a single run to gather run time data is very effective in this case and chooses near optimal data structures for every index. Regression analysis is also very effective for this program where all selections are only 75% slower than the fastest run time. When measuring the performance of randomly selected data structures, accuracy tends to improve with additional runs. Regression analysis using chosen data structures is noteworthy in this case for choosing the optimal data structure combination.

When data structure selection is applied to the cyclic chain graph (Figure C.9), set-based selection and static cost analysis select the same combinations of data structures as for the random graph. However the performance of these data structures has changed. The data structures chosen by set-based selection are now very efficient giving a run time close to the best case (only 50% slower). The efficiency of data structures selected using static cost analysis has improved slightly for the chain graph, giving a run time 452% slower than the fastest run time. This suggests that the assumptions made during static cost analysis are modelled more closely when finding the shortest path in the chain graph than the random graph. By using a single run to gather data at run time, the optimal data structure is selected for all but one index, resulting in a run time 18% slower than the fastest. Regression analysis measuring randomly selected data structures improves from 185% to only 73% slower than the fastest combination as more runs are measured. Choosing data structures deliberately so that all are represented in each index makes a further improvement to regression analysis however the resulting run time does not eclipse that found using the single run cost analysis technique.

Run time data structure selection out performed both compile time static analysis techniques for both example graphs. However, when compared to the other selection techniques, this is more likely to be the result of poor selections being made by the static analysis rather an effective run time selection.

C.4 Pascal's Triangle Generation

Pascal's triangle is a commonly referenced structure used for expansion of polynomial algebraic formula and combinatorics. The program to generate Pascal's triangle is given in Figure C.11. Each binomial coefficient `N`, is identified by its `I`th row, and `J`th column as a `pascal(I,J,N)` tuple. Elements at the beginning and end of each row are generated with 1 as their coefficient. All other coefficients are generated as the sum of the two coefficients in the previous row that occupy the same column and the previous column. To ensure termination a limit is imposed on the number of rows generated. For the purposes of these experiments 20 rows are generated. The index structure stores run time data in seven argument indexes shown in Figure C.10.

The results of data structure selection are shown in Figure C.12. The step-

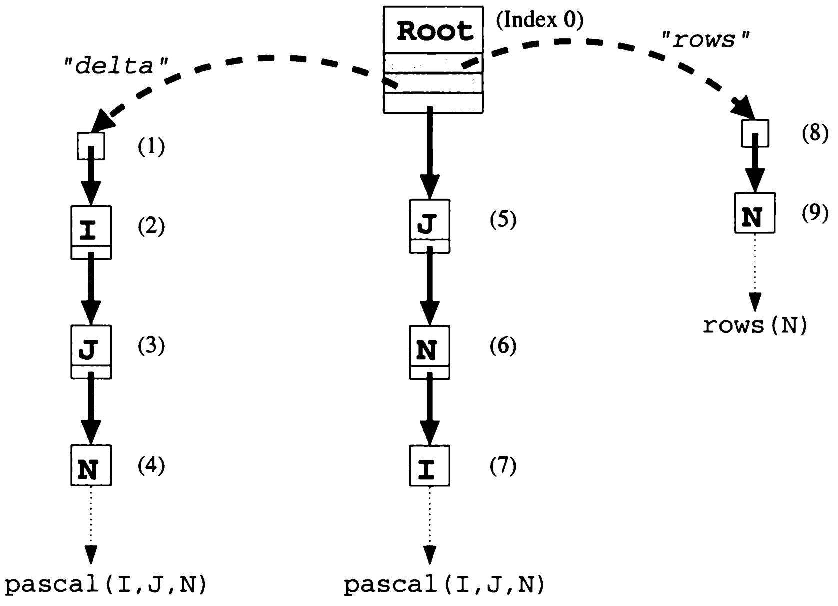


Figure C.10: Index structure schema used for the Pascal's triangle generation program.

wise nature of the program's run times shown in the graphs indicates that execution is dominated by a few indexes and the remaining indexes are almost inconsequential by comparison. However the profile of these graphs also suggests that most data structures are equally efficient when they implement the dominant indexes and only one data structure is particularly inefficient. By analysing the raw data produced by the programs it was found that when either Index 3 or 5 are implemented by a flexible array the run time increases significantly and is responsible for the first step which puts run times over 350ms. When both Indexes 3 and 5 are implemented by flexible arrays the second step occurs putting run times over 600ms.

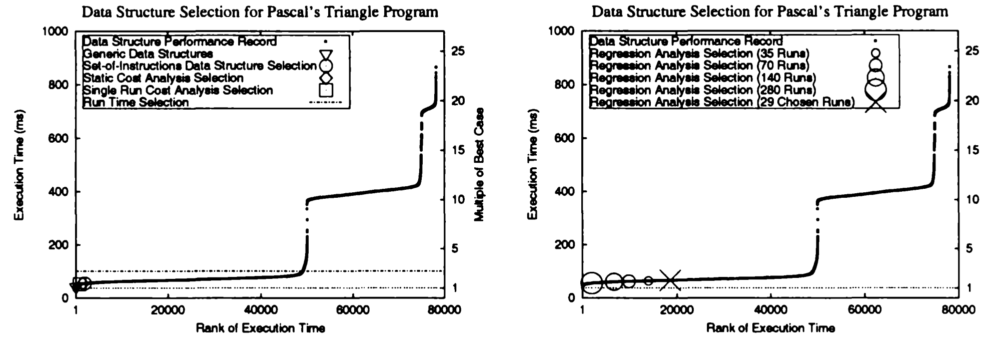
Set-based selection, static cost analysis and cost analysis using a single run all select efficient combinations of data structures for this program. The run times produced by their selections are very close to optimal (all these techniques produced run times that are less than 50% slower than the fastest data structure combination). However the regression analysis did not perform as well in these experiments. Although the regression analysis consistently avoids implementing Indexes 3 and 5 with flexible arrays, it has more difficulty selecting efficient data structures for the remaining indexes (producing run times between 78% and 48% slower than the fastest run time). This highlights a limitation of regression analysis where the dominant effect of some indexes makes selection of others difficult. To select more efficient data structures for all indexes using regression analysis requires increasing the number of measured runs, as shown in the graph. The run time selection technique produces an inefficient program when compared to the other techniques (173% slower than the fastest run time).

```
% Pascal's Triangle
%-----------------------------------------------------------------------
% Generates X rows of pascals triangle where X is the number specified
% in rows(X). A pascal(I,J,N) tuple represents each element in the
% triangle where I is the row, J is the column number and N is the
% binomial coefficient. In this system the triangle is generated as
% follows:   1
%            1 1
%            1 2 1
%            1 3 3 1
%            1 4 6 4 1
%            ...

stratify pascal(I,J,_) [I,J]. % Order elements on their row and
                                   % then column indexes.

rows(20).                        % Number of rows to generate.

pascal(0,0,1).                   % Row 0 (root of the triangle).

pascal(I,0,1) <- pascal(J,0,1), rows(Nn), % Generate a 1 as the first
                 Nn >= J, I is J+1.       % element in any row.

pascal(I,I,1) <- pascal(J,J,1), rows(Nn), % Generates a 1 as the last
                 Nn >= J, I is J+1.       % elements in any row.

pascal(Iv,Jh,N) <- pascal(Jv,Jh,N1), rows(Nn), % Generates row elements
                   Nn >= Jv, Iv is Jv+1,       % as the sum of the two
                   J is Jh-1, pascal(Jv,J,N2), % elements in the
                   N is N1+N2.                 % previous row.
```

Figure C.11: Pascal's triangle program.

Index	look		scan		insert		delete		minimum		empty	
	occurrences	performed	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.
Index0	1	231	0	0	1	253	0	0	0	0	0	0
Index1	0	0	0	0	4	253	1	22	1	254	0	0
Index2	0	0	0	0	4	253	1	253	1	253	1	253
Index3	0	0	1	253	4	253	0	0	0	0	0	0
Index5	0	0	1	210	1	253	0	0	0	0	0	0
Index6	1	1540	0	0	1	253	0	0	0	0	0	0
Index8	0	0	3	297	1	1	0	0	0	0	0	0

Index	Average # of Elements (N)	Average $log_2 N$	*Max* Value	*Min* Value	Successful Looks (R)	Duplicate Inserts (D)	Max Value Inserts	Min Value Inserts
Index0	14.0	3.78	13.0	0.0	90.9%	91.3%	17.0%	8.7%
Index1	1.82	0.827	14.8	13.9	-	91.3%	100%	0.4%
Index2	7.75	2.77	10.7	3.72	-	0%	92.1%	8.7%
Index3	1.0	0.0	14682	14682	-	0%	100%	100%
Index5	6.59	2.55	12967	1.0	-	8.3%	100%	17.0%
Index6	1.91	0.909	9.85	8.86	13.6%	0%	100%	91.7%
Index8	1.0	0.0	20.0	20.0	-	0%	100%	100%



Worst Data Structure Selection:	`[HT,FA,FA,FA,FA,SL,UL]`	(865ms)
Generic Data Structures:	`[BT,BT,BT,BT,BT,BT,BT]`	(42ms)
Best Data Structure Selection:	`[UL,UL,UL,UL,UL,UL,UL]`	(37ms)
Set-of-Instructions Selection:	`[HT,SL,SL,UL,UL,HT,UL]`	(55ms)
Static Cost Analysis Selection:	`[FA,FA,FA,UL,UL,FA,UL]`	(54ms)
Single Run Cost Analysis Selection:	`[FA,SL,FA,SL,SL,FA,UL]`	(52ms)
Regression Analysis Selection (35 runs):	`[FA,SL,HT,UL,UL,UL,HT]`	(64ms)
Regression Analysis Selection (70 runs):	`[FA,SL,UL,UL,UL,UL,HT]`	(62ms)
Regression Analysis Selection (140 runs):	`[BT,SL,UL,UL,UL,BT,SL]`	(60ms)
Regression Analysis Selection (280 runs):	`[BT,UL,BT,UL,UL,UL,FA]`	(55ms)
Regression Analysis Selection (29 chosen runs):	`[FA,FA,FA,UL,SL,FA,FA]`	(66ms)
Run Time Selection:		(101ms)

Figure C.12: Pascal's triangle program performance.

During run time data structure selection the flexible array was avoided for Indexes 3 and 5 however the additional overhead incurred at run time makes this approach inefficient for the Pascal's triangle program.

C.5 Transitive Closure

The transitive closure of a directed graph calculates the set of nodes reachable from any node in the graph. The program to determine the closure is given in Figure C.14 and is efficiently represented in Starlog. The single rule finds two existing paths which share a source and destination node and creates a new path that spans both existing paths. To ensure termination and efficiency for finite graphs, a new path is created only if it does not already exist. Paths are represented by a `path(X,Y,T)` tuple where `X` and `Y` are the source and destination node identifiers and `T` is the program iteration in which the path was created. Ordering paths on their `T` value ensures that the negation is stratified, and that existing paths will not be regenerated in later iterations.

To improve the performance of this program `path(X,Y,T)` tuples are indexed in two ways in the Γ set. Arguments are ordered `X`, then `Y`, then `T` in one index path and `T`, then `X`, then `Y` in another, as shown in the index structure in Figure C.13. The advantage of indexing these tuples in different ways is that each is optimised for the two types of searching that occurs at run time. The disadvantage is that the two indexes must be updated with any new data. Although the use of multiple indexing paths for predicates can not be inferred using the techniques given in Chapter 3, this makes an interesting example for experiments with data structure selection.

The transitive closure program is applied to two graphs. The first is a pseudo-randomly defined graph containing 200 nodes and 260 edges. The second is a cyclic chain of 50 nodes where each node has a single connection to the next in the chain.

Because the index structure used for the transitive closure program contains eight argument indexes, it is impractical to explore all combinations of data structures in order to rate the data structure selection technique. (With eight argument indexes there are 390,625 combinations of data structures.) Instead, for the transitive closure program and for programs with even more argument indexes, a random sample of all data structure combinations is used. Depending on time constraints, the samples contain between 30,000 and 50,000 data structure combinations.

Figures C.15 and C.16 give the results of data structure selection for the random graph and chain graph, respectively. For the random graph, set-based data structure selection chooses reasonably efficient data structures with a run time 67% slower than the fastest run time in the sample. The data structures selected by static cost analysis are slightly less efficient at 77% slower than the fastest combination of data structures. Using a single run to aid cost analysis, a very efficient combination of data structures is selected – almost identical to the best case found in the sample and only 4% slower. When transitive closure is applied to the chain graph, set-based selection (whose data structure selection is 140% slower than the fastest run time) is out performed by static cost analysis (only 126% slower than the fastest run time). Using a single run to aid cost analysis for the chain graph again results in an efficient combination of data

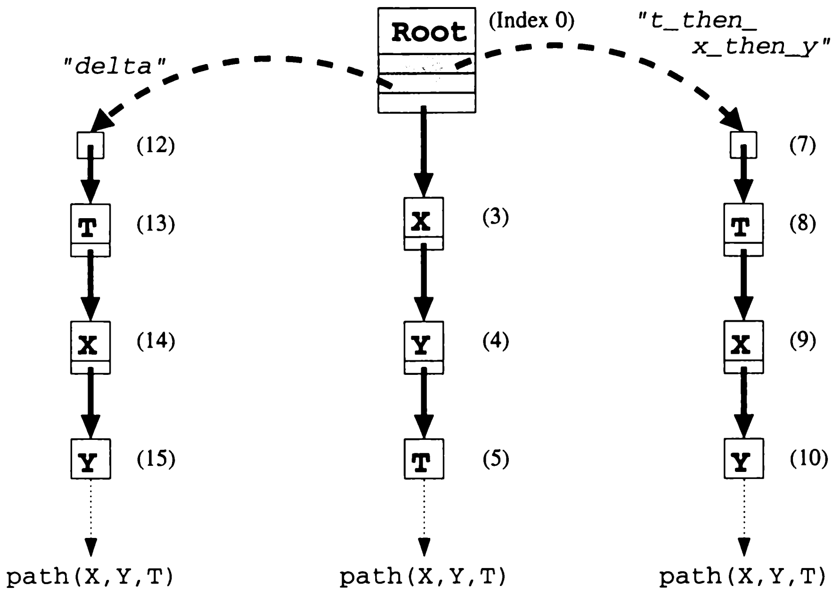


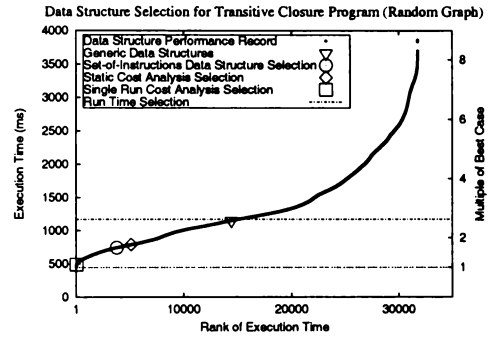
Figure C.13: Index structure schema used for the transitive closure program.

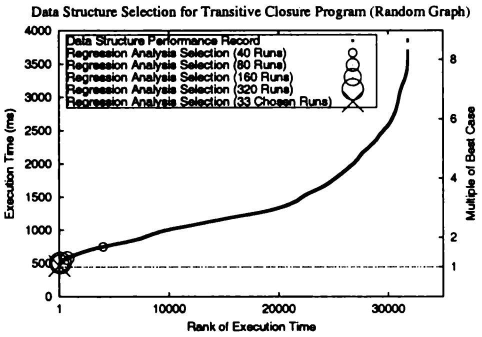
```
% Transitive Closure
%----------------------------------------------------------------
% Finds all paths that exist between any two nodes in a directed graph.
% path(X,Y,T) is the path between nodes X and Y that was found during
% iteration T of the program.

stratify path(_,_,T) [T].

path(1,2,0).                  % Sample graph (not used for benchmarks)
path(1,3,0).
path(3,4,0).
path(3,2,0).

path(X,Y,TNew) <- path(X,Z,T), path(Z,Y,T2), T >= T2,  % Generates a
                  not(path(X,Y,T3), T >= T3),         % new paths if one
                  Tnew is T+1.                         % does not exist.
```

Figure C.14: Transitive closure program.

Index	look		scan		insert		delete		minimum		empty	
	occurrences	performed	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.
Index0	6	352965	1	6359	1	6359	0	0	0	0	0	0
Index3	6	1293837	1	5093	1	6359	0	0	0	0	0	0
Index7	2	12718	0	0	1	6359	0	0	0	0	0	0
Index8	1	6359	1	6359	1	6359	0	0	0	0	0	0
Index9	1	274839	1	2038	1	6359	0	0	0	0	0	0
Index12	0	0	0	0	5	47103	1	6	1	7	0	0
Index13	0	0	1	6	264	47362	0	0	0	0	0	0
Index14	0	0	1	604	264	47362	0	0	0	0	0	0

Index	Average # of Elements (N)	Average $log_2 N$	Max Value	Min Value	Successful Looks (R)	Duplicate Inserts (D)	Max Value Inserts	Min Value Inserts
Index0	127	6.04	163	0.0	99.6%	97.6%	1.1%	5.2%
Index3	9.0	2.22	129	37.8	33.7%	42.6%	6.8%	11.7%
Index7	2.50	1.15	1.67	0.0	100%	99.9%	100%	4.1%
Index8	106	5.81	163	0.0	32.0%	90.5%	1.1%	100%
Index9	4.17	1.36	113	47.0	20.9%	0%	9.5%	100%
Index12	1.0	0.0	2.5	2.5	-	100%	100%	0.0%
Index13	101	6.60	196	0.0	-	98.8%	1.0%	1.7%
Index14	12.3	2.78	160	32.8	-	86.6%	0.5%	14.6%



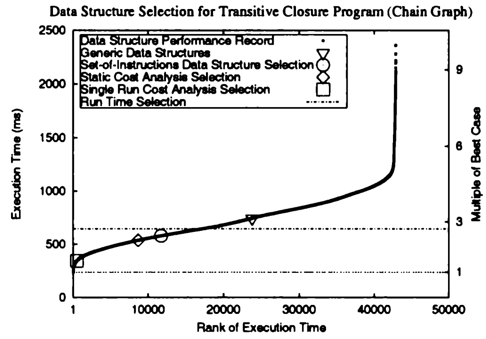


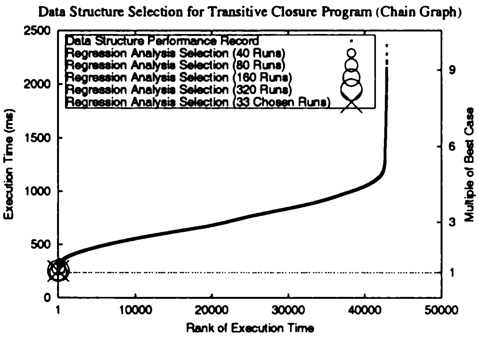
Worst Selection (in sample):	[UL,UL,HT,FA,UL,FA,UL,HT]	(3,858ms)
Generic Data Structures:	[BT,BT,BT,BT,BT,BT,BT,BT]	(1,150ms)
Best Selection (in sample):	[FA,FA,UL,SL,FA,SL,FA,FA]	(446ms)
Set-of-Instructions Selection:	[HT,HT,HT,HT,HT,SL,UL,UL]	(743ms)
Static Cost Analysis Selection:	[HT,HT,FA,UL,UL,FA,FA,FA]	(790ms)
Single Run Cost Analysis Selection:	[FA,FA,FA,UL,FA,SL,FA,FA]	(490ms)
Regression Analysis Selection (40 runs):	[HT,HT,FA,UL,BT,HT,FA,FA]	(750ms)
Regression Analysis Selection (80 runs):	[FA,FA,HT,FA,HT,UL,FA,BT]	(581ms)
Regression Analysis Selection (160 runs):	[FA,FA,SL,FA,FA,HT,FA,HT]	(503ms)
Regression Analysis Selection (320 runs):	[FA,FA,HT,SL,FA,HT,FA,UL]	(508ms)
Regression Analysis Selection (33 chosen runs):	[FA,FA,FA,SL,FA,SL,FA,FA]	(464ms)
Run Time Selection:		(1,169ms)

Figure C.15: Transitive closure program performance (random graph).

Index	look		scan		insert		delete		minimum		empty	
	occurrences	performed	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.
Index0	6	465505	1	4799	1	4799	0	0	0	0	0	0
Index3	6	699431	1	4798	1	4799	0	0	0	0	0	0
Index7	2	9598	0	0	1	4799	0	0	0	0	0	0
Index8	1	4799	1	4799	1	4799	0	0	0	0	0	0
Index9	1	122739	1	2388	1	4799	0	0	0	0	0	0
Index12	0	0	0	0	5	103613	1	8	1	9	0	0
Index13	0	0	1	8	54	103662	0	0	0	0	0	0
Index14	0	0	1	399	54	103662	0	0	0	0	0	0

Index	Average # of Elements (N)	Average $log_2 N$	Max Value	Min Value	Successful Looks (R)	Duplicate Inserts (D)	Max Value Inserts	Min Value Inserts
Index0	43.8	4.94	42.9	0.0	100%	99.0%	1.0%	3.0%
Index3	14.1	2.58	29.6	12.5	73.7%	47.9%	6.9%	9.5%
Index7	3.5	1.54	2.63	0.0	100%	99.8%	100%	1.0%
Index8	43.8	4.94	42.9	0.0	49.7%	91.7%	1.4%	100%
Index9	3.13	1.09	23.0	18.5	51.9%	0%	8.3%	100%
Index12	1.0	0.0	3.5	3.5	-	100%	100%	0.0%
Index13	49.9	5.64	48.9	0.0	-	99.6%	0.8%	4.2%
Index14	11.5	2.51	32.5	15.6	-	95.4%	3.9%	8.7%





Worst Selection (in sample):	[SL,UL,UL,HT,SL,FA,SL,BT]	(2360ms)
Generic Data Structures:	[BT,BT,BT,BT,BT,BT,BT,BT]	(743ms)
Best Selection (in sample):	[FA,FA,FA,UL,FA,SL,FA,SL]	(237ms)
Set-of-Instructions Selection:	[HT,HT,HT,HT,HT,SL,UL,UL]	(578ms)
Static Cost Analysis Selection:	[HT,HT,FA,UL,UL,FA,FA,FA]	(536ms)
Single Run Cost Analysis Selection:	[FA,FA,FA,UL,FA,SL,FA,FA]	(343ms)
Regression Analysis Selection (40 runs):	[FA,HT,SL,SL,FA,UL,FA,SL]	(327ms)
Regression Analysis Selection (80 runs):	[FA,FA,SL,SL,FA,UL,HT,SL]	(274ms)
Regression Analysis Selection (160 runs):	[FA,FA,SL,SL,FA,SL,FA,SL]	(242ms)
Regression Analysis Selection (320 runs):	[FA,FA,UL,SL,FA,SL,FA,SL]	(253ms)
Regression Analysis Selection (33 chosen runs):	[FA,FA,FA,FA,FA,FA,FA,SL]	(250ms)
Run Time Selection:		(644ms)

Figure C.16: Transitive closure program performance (chain graph).

structures – 45% slower than the best in the sample.

Regression analysis performed well for both the random and the chain graphs, typically choosing increasingly efficient data structures as more runs are analysed. This is not surprising because the run times generated by the program are not dominated by a few indexes, therefore allowing efficient data structures to be found for all indexes.

Using run time data structure selection produced inefficient run times for both graphs and were out performed by the selections made by all compile time techniques.

C.6 Game of Life

Conway's game of life (an implementation of cellular-automata) was first proposed in [46]. The rules of the game are that a cell on a two dimensional board lives to the next generation if it has either two or three neighbouring cells, and a new cell is produced if there are three cells around an empty location.

The Starlog implementation of the game of life given in Figure C.18 is unusual because it uses negation to test the number of cells that exist around a location. A cell, represented as `cell(X,Y,T)`, lives to the next generation (`T+1`) if it has 2 (different) neighbouring cells but not a third. A new cell is created at a location if it has 3 (different) neighbouring cells but not a forth. The neighbours of a cell (represented by `neighbour(X,Y,N,T)`) are precomputed where each of the eight locations around the cell are given an identifier as the `N` argument. In this way it is possible to distinguish neighbours originating from different cells. To ensure termination, an upper limit is imposed on the rules so that no cells are created beyond the 150th generation.

As shown in Figure C.17, the index structure used for the game of life contains 14 argument indexes. However, to reduce the number of data structure combinations, indexes 10 and 11 which only hold one element are implemented by unsorted lists. This allows this analysis of data structure selection to focus on the more interesting indexes.

To test data structure selection, the game of life is given two different starting patterns that generate different data sets. The first is known as the *rabbits* pattern (see [60]) which begins as an arrangement of nine cells and, as the name suggests, grows quickly over time. The second pattern, known as the *traffic light* (see [60]), is composed of 12 oscillating cells that never increase or decrease in number.

When the data structure selection techniques are applied to the game of life, all compile time techniques select reasonable combinations of data structures for both the rabbits and the traffic light patterns (see Figures C.19 and C.20). The set-based selection technique chose data structures which were 22% and 38% slower than the fastest sampled run times. Static cost analysis selected more efficient data structures for the rabbits pattern than the set-based approach but a slightly less efficient combination for the traffic light. Using a single run to aid cost analysis, a very efficient combination of data structures was selected for both patterns.

Although the smooth rate of change in the run time graphs indicates the performance of the life program is not dominated by a few indexes, the regression analysis had mixed results. For the rabbits pattern, when a small number

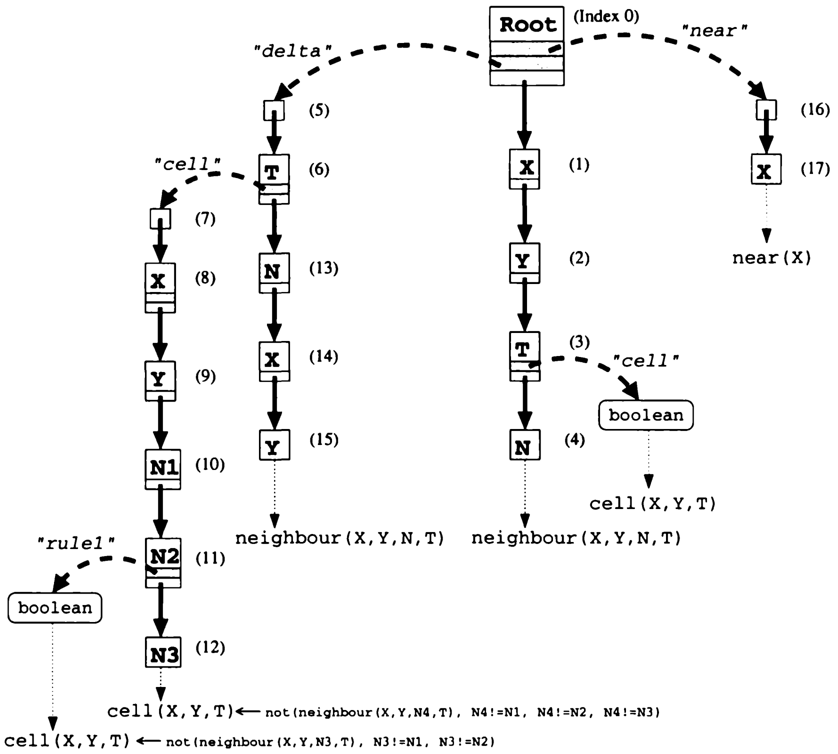


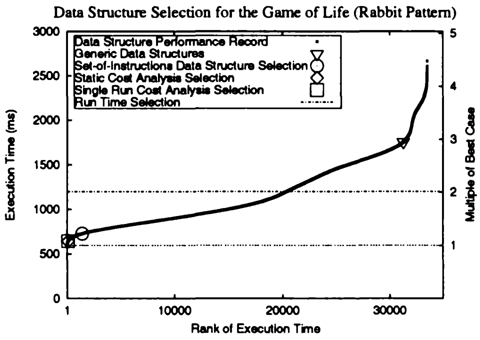
Figure C.17: Index structure schema used for the game of life.

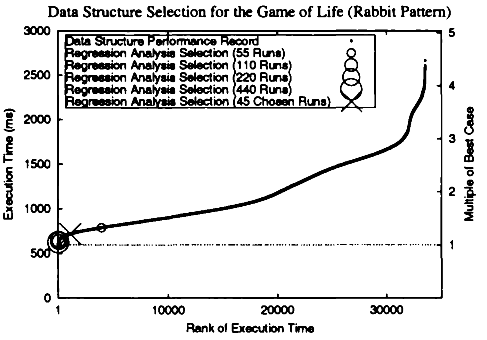
```
% Game of Life
%-----------------------------------------------------------------
% Implementation of Conway's Game of Life. Each live cell in on the
% board is a cell(X,Y,T) tuple where X and Y are its spacial
% coordinates and T is the generation that the cell is alive. The eight
% neighbours of a cell are labelled [-4,-3,-2,-1,1,2,3,4] and a cell
% which has two unique neighbouring cells will live to the next
% generation. If an empty location has three neighbouring cells a new
% cell will be created in the next generation.

stratify cell(_,_,T) [T].
stratify neighbour(_,_,_,T) [T].


cell(49,50,0).               % Sample board (not used for benchmarks)
cell(50,50,0).
cell(51,50,0).

cell(X,Y,TNew) <- cell(X,Y,T), T < 150, neighbour(X,Y,N1,T),
                  neighbour(X,Y,N2,T), N2 > N1,     % A cell with 2 but
                  not(neighbour(X,Y,N3,T),          % not 3 neighbours
                      N3 =\= N1, N3 =\= N2),        % lives on.
                  TNew is T+1.

cell(X,Y,TNew) <- neighbour(X,Y,N1,T), T < 150, neighbour(X,Y,N2,T),
                  N2 > N1,  neighbour(X,Y,N3,T), N3 > N2,  % A new
                  not(neighbour(X,Y,N4,T), N4 =\= N1,      % cell is
                      N4 =\= N2, N4 =\= N3), TNew is T+1.   % born.

neighbour(XNew,YNew,N,T) <- cell(X,Y,T), near(DX), near(DY),
                            N is (DX*3)+DY, N =\= 0,     % Generates
                            XNew is X+DX, YNew is Y+DY.   % neighbours

near(-1).                    % Used when generating neighbours.
near(0).
near(1).
```

Figure C.18: Game of life program.

Index	look		scan		insert		delete		minimum		empty	
	occurrences	performed	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.
Index0	2	77467	0	0	4	81806	0	0	0	0	0	0
Index1	2	77467	0	0	4	81806	0	0	0	0	0	0
Index2	2	77467	0	0	4	81806	0	0	0	0	0	0
Index3	0	0	5	296120	1	72640	0	0	0	0	0	0
Index5	0	0	0	0	13	77476	1	151	1	152	0	0
Index6	0	0	0	0	3	72640	1	1200	1	151	0	0
Index7	0	0	1	151	11	77476	0	0	0	0	0	0
Index8	0	0	1	16836	11	77476	0	0	0	0	0	0
Index9	0	0	2	27878	2	77467	0	0	0	0	0	0
Index13	0	0	1	1200	3	72640	0	0	0	0	0	0
Index14	0	0	1	18288	3	72640	0	0	0	0	0	0

Index	Average # of Elements (N)	Average $log_2 N$	Max Value	Min Value	Successful Looks (R)	Duplicate Inserts (D)	Max Value Inserts	Min Value Inserts
Index0	21.1	4.31	63.3	43.2	100%	100%	0.5%	0.9%
Index1	14.6	3.71	59.5	45.2	100%	99.1%	2.2%	1.8%
Index2	21.0	4.13	66.4	37.4	100%	62.2%	100%	1.0%
Index3	2.01	0.799	0.0	-2.11	-	0%	100%	42.2%
Index5	1.0	0.0	75.0	75.0	-	99.8%	100%	0.0%
Index6	7.95	2.98	3.97	-3.97	-	98.3%	12.5%	13.9%
Index7	15.5	3.86	62.1	44.8	-	97.0%	2.9%	3.1%
Index8	5.26	2.28	56.6	49.0	-	81.3%	15.6%	16.4%
Index9	1.03	0.0331	1.66	0.205	-	64.0%	100%	18.7%
Index13	15.1	3.80	61.6	44.5	-	74.8%	3.3%	100%
Index14	3.18	1.56	56.0	48.9	-	0%	25.2%	100%



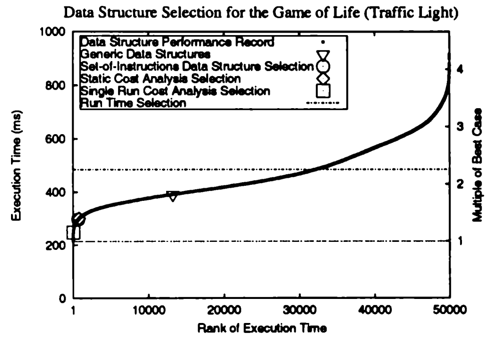


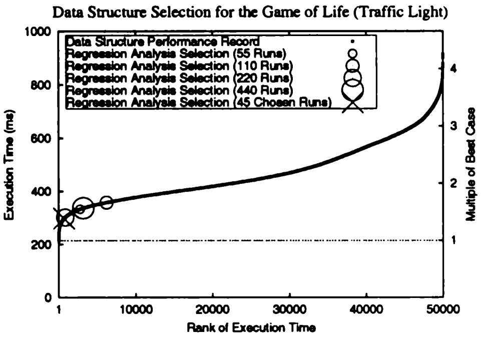
Worst Selection (in sample):	[UL,UL,UL,HT,UL,BT,UL,BT,HT,FA,HT]	(2,663ms)
Generic Data Structures:	[BT,BT,BT,BT,BT,BT,BT,BT,BT,BT,BT]	(1,754ms)
Best Selection (in sample):	[FA,FA,SL,UL,BT,SL,UL,UL,UL,FA,SL]	(595ms)
Set-of-Instructions Selection:	[HT,HT,HT,UL,SL,SL,UL,UL,UL,UL,UL]	(727ms)
Static Cost Analysis Selection:	[FA,FA,FA,UL,FA,FA,FA,FA,UL,UL,UL]	(643ms)
Single Run Cost Analysis Selection:	[FA,FA,FA,UL,SL,FA,FA,UL,FA,FA,UL]	(645ms)
Regression Analysis Selection (55 runs):	[FA,FA,SL,FA,HT,HT,UL,HT,UL,SL,HT]	(789ms)
Regression Analysis Selection (110 runs):	[FA,FA,SL,SL,HT,UL,SL,HT,UL,SL,SL]	(640ms)
Regression Analysis Selection (220 runs):	[FA,FA,SL,SL,FA,UL,SL,FA,UL,SL,SL]	(643ms)
Regression Analysis Selection (440 runs):	[FA,FA,FA,UL,FA,SL,SL,UL,UL,SL,SL]	(628ms)
Regression Analysis Selection (45 chosen runs):	[UL,HT,SL,SL,UL,SL,SL,SL,SL,SL,SL]	(718ms)
Run Time Selection:		(1,197ms)

Figure C.19: Game of life program performance (rabbit pattern).

Index	look		scan		insert		delete		minimum		empty	
	occurrences	performed	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.
Index0	2	1800	0	0	4	16212	0	0	0	0	0	0
Index1	2	1800	0	0	4	16212	0	0	0	0	0	0
Index2	2	1800	0	0	4	16212	0	0	0	0	0	0
Index3	0	0	5	37800	1	14400	0	0	0	0	0	0
Index5	0	0	0	0	16	1812	1	151	1	152	0	0
Index6	0	0	0	0	3	14400	1	1200	1	151	0	0
Index7	0	0	1	151	14	1812	0	0	0	0	0	0
Index8	0	0	1	2719	14	1812	0	0	0	0	0	0
Index9	0	0	2	1800	2	1800	0	0	0	0	0	0
Index13	0	0	1	1200	3	14400	0	0	0	0	0	0
Index14	0	0	1	7200	3	14400	0	0	0	0	0	0

Index	Average # of Elements (N)	Average $log_2 N$	*Max* Value	*Min* Value	Successful Looks (R)	Duplicate Inserts (D)	Max Value Inserts	Min Value Inserts
Index0	10.9	3.44	58.6	48.7	100%	99.9%	1.4%	1.5%
Index1	5.95	2.57	56.6	50.7	100%	99.5%	5.4%	5.4%
Index2	53.0	5.25	73.0	0.0	100%	48.1%	100%	8.1%
Index3	1.5	0.465	0.0	-1.25	-	0%	100%	58.3%
Index5	1.0	0.0	75.0	75.0	-	91.7%	100%	0.7%
Index6	7.95	2.98	3.97	-3.97	-	91.7%	12.5%	19.8%
Index7	6.01	2.57	57.5	50.5	-	49.9%	41.7%	29.2%
Index8	1.50	0.497	55.0	52.0	-	0%	75.2%	66.6%
Index9	1.0	0.0	0.993	0.993	-	0%	100%	100%
Index13	5.96	2.55	57.1	50.2	-	50.0%	16.7%	100%
Index14	1.49	0.497	54.6	51.7	-	0%	50.0%	100%





Worst Selection (in sample):	[BT,UL,UL,HT,FA,HT,SL,UL,FA,FA,HT]	(915ms)
Generic Data Structures:	[BT,BT,BT,BT,BT,BT,BT,BT,BT,BT,BT]	(391ms)
Best Selection (in sample):	[SL,BT,SL,SL,UL,UL,SL,UL,SL,UL,SL]	(215ms)
Set-of-Instructions Selection:	[HT,HT,HT,UL,SL,SL,UL,UL,UL,UL,UL]	(297ms)
Static Cost Analysis Selection:	[FA,FA,FA,UL,FA,FA,FA,FA,UL,UL,UL]	(301ms)
Single Run Cost Analysis Selection:	[FA,FA,FA,SL,SL,FA,SL,SL,FA,UL,SL]	(248ms)
Regression Analysis Selection (55 runs):	[SL,FA,SL,UL,SL,UL,SL,BT,HT,UL,SL]	(333ms)
Regression Analysis Selection (110 runs):	[HT,FA,SL,UL,HT,FA,HT,BT,UL,HT,SL]	(358ms)
Regression Analysis Selection (220 runs):	[HT,FA,SL,SL,FA,FA,FA,HT,UL,SL,SL]	(302ms)
Regression Analysis Selection (440 runs):	[HT,FA,SL,SL,FA,BT,FA,BT,UL,UL,SL]	(337ms)
Regression Analysis Selection (45 chosen runs):	[SL,SL,SL,UL,UL,UL,SL,UL,FA,SL,SL]	(298ms)
Run Time Selection:		(485ms)

Figure C.20: Game of life program performance (traffic light pattern).

of measured runs are analysed, the regression analysis selects relatively inefficient combinations of data structure when compared to the other selection techniques. However, predictably, as the number of runs increases the results of regression analysis improve. When regression analysis is applied the traffic light pattern the results are inconsistent where larger numbers of measured runs do not necessary yield the best selection. Such inconsistencies are due to additional degrees of error being introduced into the sample measurements, evident by the regression's low multiple R-squared value of 0.56 for the largest sample of run times compared to 0.89 for that of the transitive closure program. Although the cause of this additional error can not be substantiated, it is probably related to the extra memory requirements of the more complex index structure prompting a more active (yet inconsistent) garbage collection process.

Delaying the choice of data structure until run time produced inefficient programs for both patterns. Indeed, the run times produced were significantly longer than any of the compile time techniques.

C.7 N-Queens

The N-queens problem is a common example used for demonstrating advantages of logic programming. The Starlog implementation of the N-queens program is given in Figure C.22. (For an alternative declarative representation of the N-queens program see [52].) The program builds a search tree where a new queen is positioned on a chess-board where it can not attack any other previously positioned queen. A positioned queen is represented as `q(X,D,I,J)` where `X` is the identifier of the node in the search tree, `D` is the node's depth (which corresponds to the number of queens already positioned) and `I` and `J` are the coordinates of the queen on the board. The `child(X,D,Y,E)` represent the branches of the search tree where `X` and `Y` are the identifiers of the parent node and child nodes respectively and `D` and `E` are their depths. To test if two queens can attack each other `attack(X,Y)` relations are used. To reduce the run time of this program a 6 by 6 chess board is used rather than the usual 8 by 8. To hold run time data the index structure contains 13 argument indexes, as shown in Figure C.21. To simplify Figure C.21, the binding patterns for predicates `solution/2`, `child/2`, `fail/2`, `node/2` have been omitted.

The step-wise trend exhibited by the graphs of Figure C.23 indicate that the run time of N-queens program is dominated by a few indexes. From analysis of the data, the use of flexible arrays for Indexes 2 and 19 is responsible for each of the steps. Set-based selection and static cost analysis proved very effective for this program, both choosing data structures within 25% of the fastest run time in the sample. The selection made by static cost analysis was more efficient than its set-based alternative indicating the actual properties of the data sets are similar to those assumed in the static cost table. By using a single run to aid cost analysis, an uncharacteristically poor selection was made when compared to the selections made by the previous techniques. The chosen data structure combination was 40% slower than the fastest run time. The reason for the poor performance of the chosen data structure combination is because sub-optimal data structures were selected for a few indexes – namely Indexes 14, 17, 19 and 20. Because all these indexes store only one element at run time it is believed that the formulas used to estimate costs are inaccurate for this case.

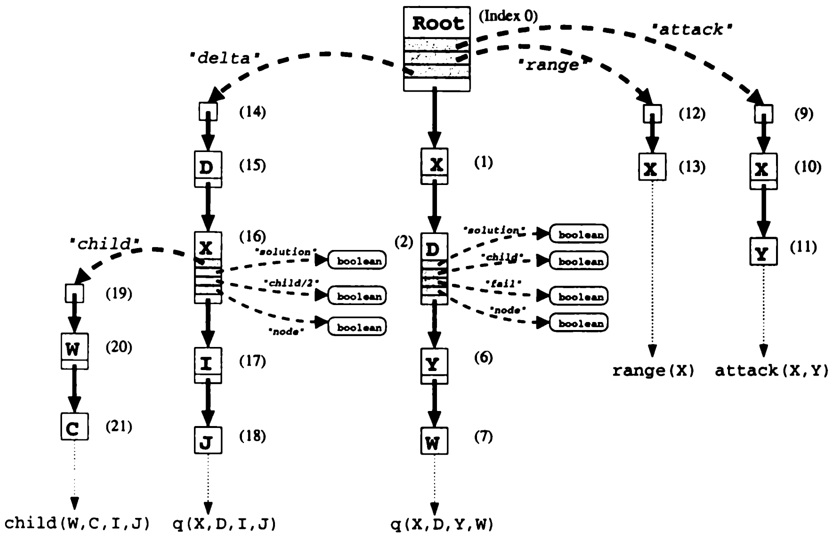


Figure C.21: Index structure schema used for the N-queens program.

Although regression analysis ensured the most significant indexes were not implemented by worst case data structures, the dominance of Indexes 2 and 19 reduces the effectiveness of this selection technique. Furthermore, additional error may be introduced to the measured run times due to the large memory requirement of this index structure (a phenomenon more obvious in the game of life program). As a result, the data structures selected for most indexes by regression analysis are more inconsistent for this program than the other example programs. In general, to improve the accuracy of this selection technique additional runs must be performed although, as shown by the graphed results, an improvement is not always guaranteed.

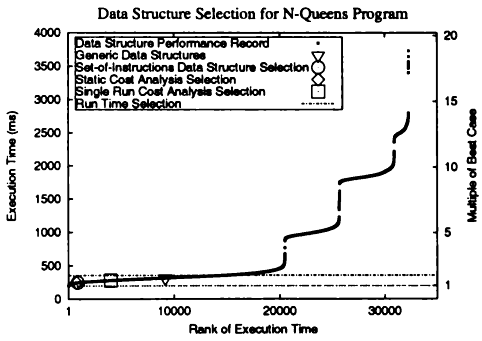
Run time selection of data structures was ineffective when compared with the compile time selection techniques (but still only 79% slower than the fastest run time). In this program, recalculating time cost estimates and subsequent reassignment from one data structure to another are frequent because of the many, small data sets used throughout execution (see Table 2 of Figure C.23). This results in additional overhead that reduces the efficiency of the program.

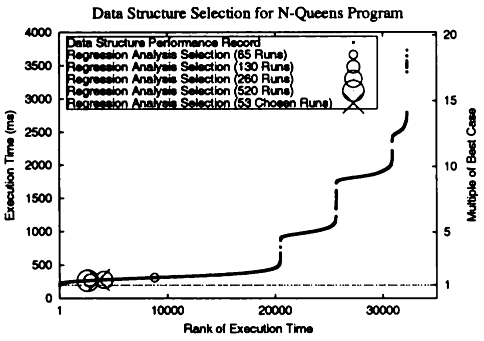
```
% N-Queens Program
%---------------------------------------------------------------------
% Calculates all the positions that 6 queens can fit on a 6x6 board
% without attacking each other. attack(I,J) tuples determine all
% locations where a queen at location (I,J) conflicts with another
% queen at location (0,0). A positioned queen is given as
% q(ID,Depth,I,J) where ID is the ID of the node in the search tree,
% depth is the number of queens already positioned, and (I,J) is the
% location of the queen. child(ID1,Depth1,ID2,Depth2) are the branches
% of the search tree where ID1 is the root node where Depth1 queens
% have been positioned, and ID2 is a child node where Depth2 queens
% have been positioned. fail(X,D) nodes are generated where two queens
% conflict.

stratify child(_,_,_,D) [D].            % Order all tuples by the
stratify q(_,D,I,_) [D, I].             % number of queens already
stratify node(X,D) [D].                 % positioned and then q/4
stratify fail(X,D) [D].                 % by its column index.
stratify solution(X,D) [D].
stratify child(X,D) [D].

range(0). range(1). range(2).            % All row/column values on
range(3). range(4). range(5).            % board.

% Positions on a [-6..6]x[-6..6] board that a queen can attack (0,0)
attack(0,1). attack(0,2). attack(0,3). attack(0,4). attack(0,5).
attack(0,-1). attack(0,-2). attack(0,-3). attack(0,-4). attack(0,-5).
attack(1,0). attack(2,0). attack(3,0). attack(4,0). attack(5,0).
attack(1,1). attack(2,2). attack(3,3). attack(4,4). attack(5,5).
attack(1,-1). attack(2,-2). attack(3,-3). attack(4,-4). attack(5,-5).

child(1,-1).                             % First node id=1,depth= -1

node(X,D) <- q(X,D,_,_).                 % Each node in search tree.

solution(X,D) <- node(X,D),              % All non-attacking queens
                 not(fail(X,D)), D is 5.  % positioned.

child(X,D) <- node(X,D),                 % Not all non-attacking
              not(fail(X,D)), D =\= 5.   % queens positioned.

q(Y,E,E,I) <- child(X,D), E is D+1,       % Position a new queen in
              range(I), gensym(X,I,Y).   % every column of row E.

child(X,D,Y,E) <- child(X,D), E is D+1,   % Creates new search tree
                  range(I), gensym(X,I,Y).      % node.

q(X,D,I,J) <- child(W,C,X,D), C < D,     % Copy previous queens
              q(W,C,I,J).                % from previous node.

fail(X,D) <- q(X,D,I,J), q(X,D,K,L),     % Find two previously
             K < I, Id is I-K,            % previously positioned
             Jd is J-L, attack(Id,Jd).   % queens that attack.
```

Figure C.22: N-Queens program source for 6x6 board.

Index	look		scan		insert		delete		minimum		empty	
	occurrences	performed	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.	occ.	perf.
Index0	6	6067	0	0	5	6063	0	0	0	0	0	0
Index1	6	6066	0	0	5	6063	0	0	0	0	0	0
Index2	0	0	3	5020	2	5016	0	0	0	0	0	0
Index6	0	0	3	15426	1	4122	0	0	0	0	0	0
Index9	1	8052	0	0	31	31	0	0	0	0	0	0
Index10	1	8052	0	0	40	40	0	0	0	0	0	0
Index12	0	0	1	149	6	6	0	0	0	0	0	0
Index14	0	0	0	0	5	4275	1	7	1	8	0	0
Index15	0	0	1	7	5	4275	0	0	0	0	0	0
Index16	0	0	1	895	2	4122	1	4122	1	5017	0	0
Index17	0	0	1	5016	2	4122	0	0	0	0	0	0
Index19	0	0	1	894	1	894	0	0	0	0	0	0
Index20	0	0	1	894	1	894	0	0	0	0	0	0

Index	Average # of Elements (N)	Average $log_2 N$	*Max* Value	*Min* Value	Successful Looks (R)	Duplicate Inserts (D)	Max Value Inserts	Min Value Inserts
Index0	178	4.79	4988	0.857	100%	85.2%	0.5%	0.1%
Index1	1.0	0.0	0.714	0.714	100%	85.2	100%	100%
Index2	2.14	0.987	329	0.143	-	0%	85.1%	17.9%
Index6	1.0	0.0	1.56	1.56	-	0%	100%	100%
Index9	11.0	3.46	5.0	-5.0	100%	64.5%	25.8%	25.8%
Index10	3.0	1.58	3.0	-3.0	14.5%	0%	62.5%	65.0%
Index12	6.0	2.58	5.0	0.0	-	0%	100%	16.7%
Index14	1.0	0.0	2.0	2.0	-	99.8%	50.0%	79.1%
Index15	128	5.49	34908	20647	-	79.1%	0.5%	21.3%
Index16	2.95	1.34	3.91	1.95	-	0%	21.7%	100%
Index17	1.0	0.0	1.71	1.71	-	0%	100%	100%
Index19	1.0	0.0	3968	3968	-	0%	100%	100%
Index20	1.0	0.0	1.29	1.29	-	0%	100%	100%





Worst Selection (in sample):	[UL,FA,FA,HT,HT,HT,SL,BT,SL,UL,FA,FA,UL]	(3,730ms)
Generic Data Structures:	[BT,BT,BT,BT,BT,BT,BT,BT,BT,BT,BT,BT,BT]	(313ms)
Best Selection (in sample):	[BT,UL,UL,UL,BT,UL,UL,UL,BT,SL,SL,SL,UL]	(198ms)
Set-of-Instructions Selection:	[HT,HT,UL,UL,HT,HT,UL,SL,UL,SL,UL,UL,UL]	(247ms)
Static Cost Analysis Selection:	[FA,FA,UL,UL,FA,FA,UL,FA,UL,SL,UL,UL,UL]	(245ms)
Single Run Cost Analysis:	[FA,FA,SL,UL,FA,FA,UL,SL,HT,SL,FA,SL,FA]	(278ms)
Regression Selection (65 runs):	[HT,UL,BT,UL,BT,BT,FA,HT,SL,UL,FA,SL,BT]	(311ms)
Regression Selection (130 runs):	[BT,BT,SL,FA,FA,HT,BT,UL,BT,FA,BT,BT,BT]	(268ms)
Regression Selection (260 runs):	[BT,FA,UL,FA,HT,SL,HT,BT,HT,FA,FA,BT,BT]	(279ms)
Regression Selection (520 runs):	[BT,FA,UL,UL,HT,FA,HT,UL,HT,FA,SL,UL,FA]	(266ms)
Regression Selection (53 chosen runs):	[HT,HT,SL,UL,HT,SL,UL,SL,HT,UL,UL,BT,BT]	(275ms)
Run Time Selection:		(354ms)

Figure C.23: N-Queens program performance.

Bibliography

[1] M. Abadi. Temporal logic in programming. In *International Symposium of Logic Programming*, pages 4–16, 1987.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing, Reading, Massachusetts, 1986.

[3] Hassan Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[4] V. Alexiev. Mutable object state for object-oriented logic programming: A survey. Technical Report TR93–15, University of Alberta, 1993.

[5] B. Alpern, A. Cocchi, D. Lieber, M. Mergen, and V. Sarkar. Jalapeno - a compiler supported Java virtual machine for servers. In *Workshop on Compiler Support for Software System (WCSSS 99)*, May 1999.

[6] D. Aquilino, Patrizia Asirelli, C. Renso, and Franco Turini. An operator for composing deductive databases with theories of constraints. In *Logic Programming and Non-monotonic Reasoning*, pages 57–70, 1995.

[7] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[8] Leo Bachmair, Ta Chen, and I. V. Ramakrishnan. Associative-commutative discrimination nets. In *Proceedings of the International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 61–74. Springer-Verlag, 1993.

[9] David John Bacon. *SETL for internet data processing.* PhD thesis, 1998.

[10] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3), 1987.

[11] Chitta Baral and Michael Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19/20:73–148, 1994.

[12] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(10):757–763, October 1966.

[13] Aart J. C. Bik and Harry A. G. Wijshoff. On automatic data structure selection and code generation for sparse computations. In *1993 Workshop on Languages and Compilers for Parallel Computing*, pages 57–75, Portland, Ore., 1993. Springer Verlag.

[14] Aart J. C. Bik and Harry A. G. Wijshoff. Advanced compiler optimizations for sparse computations. *Journal of Parallel and Distributed Computing*, 31(1):14–24, 1995.

[15] Dominic Frank Julian Binks. *Declarative Debugging in Godel.* PhD thesis, University of Bristol, 1995.

[16] J. Cai, P. Facon, F. Henglein, R. Paige, and E. Schonberg. Type analysis and data structure selection. In B. Moller, editor, *Constructing Programs from Specifications.* North-Holland, 1991.

[17] Eugene Charniak, Christopher K. Riesbeck, and Drew V. McDermott. *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, New Jersey, 1980.

[18] Ta Chen and Siva Anantharaman. STORM: A many-to-one associative-commutative matcher. In *6th International Conference on Rewriting Techniques and Applications*, pages 414–419, 1995.

[19] Ta Chen, I.V. Ramakrishnan, Siva Anantharaman, and Jacques Chabin. Experiments with associative-commutative discrimination nets. In *International Joint Conference on Artificial Intelligence*, pages 348–354, 1995.

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

[21] Tyng-Ruey Chuang and Wen L. Hwang. A probabilistic approach to the problem of automatic selection of data representations. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 190–200. ACM Press, 1996.

[22] Roger Clayton, John G. Cleary, Bernhard Pfahringer, and Mark Utting. Optimising tabling structures for bottom-up logic programming. In *Pre-proceedings of the International Workshop on Logic Based Program Development and Transformations*, pages 57–75, 2002.

[23] J. Cleary, M. Utting, and R. Clayton. Data structures considered harmful. In John Lloyd, editor, *Proceedings of the Australasian Workshop on Computational Logic*, pages 111–120, 2000.

[24] John Cleary and Mark Utting. Verification of Starlog programs. In *The Australasian Workshop on Computational Logic (AWCL)*, 2001.

[25] John G. Cleary, Roger Clayton, Mark Utting, and Bernhard Pfahringer. A semantics and implementation of stratified logic programs. Technical Report 03/2004, University of Waikato, Hamilton, New Zealand, 2004.

[26] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[27] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[28] Alain Colmerauer and Philippe Roussel. The birth of Prolog. In *The second ACM SIGPLAN conference on History of Programming Languages*, pages 37–52. ACM Press, 1993.

[29] Thomas Conway, Fergus Henderson, and Zoltan Somogyi. Code generation for Mercury. In *International Logic Programming Symposium*, pages 242–256, 1995.

[30] C. J. Date. *An Introduction to Database Systems, Sixth Edition.* Addison-Wesley Publishing, Reading, Massachusetts, 1995.

[31] R. E. Davis. Logic programming and Prolog: A tutorial. In Ellis Horowitz, editor, *Programming Languages: A Grand Tour, Third Edition*, pages 493–502, Rockville, MD, 1987. Computer Science Press.

[32] Hans De Nivelle. An algorithm for the retrieval of unifiers from discrimination trees. *Journal of Automated Reasoning*, 20(1–2):5–25, 1998.

[33] Suamya K. Debray and David S. Warren. Towards banishing the Cut from Prolog. *IEEE Transactions on Software Engineering*, 16(3):335–349, March 1990.

[34] S. Deiters and U. Griefahn. Propagation rule compiler: Tool specification. Technical Report IDEA.DE.22.O.001, University of Bonn, Germany, November 1994. ESPRIT project P6333 (IDEA).

[35] M. A. Derr, S. Morishita, and G. Phipps. The Glue-Nail deductive database system: Design, implementation, and evaluation. *Very Large Data Base Journal*, 3(2):123–160, 1994.

[36] M. Kathryn Di Benigno, George R. Cross, and Cary G. de Bessonet. COREL: a conceptual retrieval system. In *Proceedings of the 9th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 144–148. ACM Press, 1986.

[37] S. Dietzen and F. Pfenning. A declarative alternative to "assert" in logic programming. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 372–386, San Diego, USA, 1991. The MIT Press.

[38] Bruce Eckel. *Thinking in Java.* Prentice-Hall, New Jersey, 1998.

[39] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, Second Edition.* Addison-Wesley Publishing, Menlo Park, California, 1994.

[40] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[41] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

[42] Michael Dean Ernst. *Dynamically discovering likely program invariants.* PhD thesis, 2000.

[43] K. Frenkel. An interview with the 1986 A. M. Turing Award recipients – John E. Hopcroft and Robert E. Tarjan. *CACM*, 30(3):214–223, March 1987.

[44] M Fujita, S Kono, H Tanaka, and T Moto-oka. Tokio: logic programming language based on temporal logic and its compilation to Prolog. In *Proceedings on Third international conference on logic programming*, pages 695–709. Springer-Verlag New York, Inc., 1986.

[45] Harald Ganzinger and David McAllester. Logical algorithms. *Lecture Notes in Computer Science*, 2401:209–233, 2002.

[46] Martin Gardner. Mathematical games. *Scientific American*, 223:120–123, October 1970.

[47] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[48] Fosca Giannotti, Dino Pedreschi, and Carlo Zaniolo. Semantics and expressive power of nondeterministic constructs in deductive databases. *Journal of Computer and System Sciences*, 62(1):15–42, 2001.

[49] Steve Gregory. A declarative approach to concurrent programming. In *Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 79–93. Springer-Verlag, September 1997.

[50] Gopal Gupta and Vitor Santos Costa. Cuts and side-effects in and-or parallel Prolog. *Journal of Logic Programming*, 27(1):45–71, 1996.

[51] Antonella Guzzo and Domenico Sacca. Modelling the future with Event Choice DATALOG. In *Proceedings of the Joint Conference on Declarative Programming*, pages 57–70, September 2002.

[52] Jiawei Han, Ling Liu, and Tong Lu. Evaluation of declarative n-queens recursion: A deductive database approach. *Information Sciences*, 105(1-4):69–100, 1998.

[53] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *Proceedings of the 15th International Conference on Data Engineering*, pages 266–275. IEEE Computer Society Press, 1999.

[54] I. T. Hawryszkiewycz. *Relational database design: An introduction.* Prentice-Hall, Sydney, 1990.

[55] F. Henderson. Strong modes can change the world. Technical Report 93/25, University of Melbourne, Melbourne, Australia, 1993.

[56] F. Henderson, T. Conway, Z. Somogyi, and D. Jeffery. The Mercury language reference manual. Technical Report 96/10, University of Melbourne, Melbourne, Australia, 1996.

[57] F. Henderson, Z. Somogyi, and T. Conway. Determinism analysis in the Mercury compiler. In *Australian Computer Science Conference*, pages 337–346, 1996.

[58] Fergus Henderson, Thomas Conway, and Zoltan Somogyi. Compiling logic programs to C using GNU C as a portable assembler. In *ILPS'95 Post-conference Workshop on Sequential Implementation Technologies for Logic Programming*, pages 1–15, Portland, Or, 1995.

[59] Fergus Henderson and Zoltan Somogyi. Compiling Mercury to high-level C code. In *Computational Complexity*, pages 197–212, 2002.

[60] Al Hensel. A brief illustrated glossary of terms in conway's game of life, 1995. Archived at www.radicaleye.com/lifepage/glossary.html.

[61] David B. Kemp, Divesh Srivastava, and Peter J. Stuckey. Magic sets and bottom-up evaluation of well-founded models. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 337–354, San Diego, USA, 1991. The MIT Press.

[62] David B. Kemp, Divesh Srivastava, and Peter J. Stuckey. Bottom-up evaluation and query optimization of well-founded models. *Theoretical Computer Science*, 146(1–2):145–184, 1995.

[63] W. Kiebling, H. Schmidt, W. Straub, and G. Dunzinger. DECLARE and SDS: Early efforts to commercialize deductive database technology. *Very Large Data Base Journal*, 3(2):211–243, 1994.

[64] Ulrike Klusik, Rita Loogen, and Steffen Priebe. Controlling parallelism and data distribution in Eden. In *Proceedings of the Second Scottish Functional Programming Workshop*, pages 53–64, 2001.

[65] Feliks Kluzniak and Stanislaw Szpakowicz. *Prolog for Programmers*. Academic Press, London, 1985.

[66] Robert Kowalski. Algorithm = Logic + Control. In *Communications of the ACM*, volume 22, pages 424–436, 1979.

[67] Gabriel M. Kuper. Aggregation in constraint databases. In *Principles and Practice of Constraint Programming*, pages 166–173, 1993.

[68] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Second, Extended Edition, Berlin Heidelberg, 1987.

[69] Donald W. Loveland, David W. Reed, and Debra Sue Wilson. SATCHMORE: SATCHMO with Relevancy. *Journal of Automated Reasoning*, 14(2):325–351, 1995.

[70] James Low and Paul Rovner. Techniques for the automatic selection of data structures. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, pages 58–67. ACM Press, 1976.

[71] Lunjin Lu and John G. Cleary. An operational semantics of Starlog. In *Principles and Practice of Declarative Programming*, pages 294–310, 1999.

[72] Yannis Manolopoulos, Yannis Theodoridis, and Vassilis J. Tsotras. *Advanced Database Indexing.* Kluwer Academic Publisher, Boston, 2000.

[73] Claudia Marcus. *Prolog Programming.* Addison-Wesley Publishing, Reading, Massachusetts, 1986.

[74] James Martin. *Computer Data-Base Organization, Second Edition.* Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[75] David McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002.

[76] William McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[77] Lee Naish. Negation and control in Prolog. *Lecture Notes in Computer Science*, 238, 1986.

[78] Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

[79] Greg Nelson. *Techniques for program verification.* PhD thesis, Stanford University, Palo Alto, CA, 1980.

[80] Robert Nieuwenhuis, Thomas Hillenbrand, Alexandre Riazanov, and Andrei Voronkov. On the evaluation of indexing techniques for theorem proving. *Lecture Notes in Computer Science*, 2083:257–271, 2003.

[81] Kazuhiro Ogata, Shigenori Ioroi, and Kokichi Futatsugi. Optimizing term rewriting using discrimination nets with specialization. In *Proceedings of the 1999 ACM symposium on Applied Computing*, pages 511–518. ACM Press, 1999.

[82] Richard O'Keefe. *The Craft of Prolog.* The MIT Press, Cambridge, MA, 1990.

[83] T. C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *Proceedings of the ACM Symposium on Principle of Database Systems*, pages 11–21, 1989.

[84] I. V. Ramakrishnan, Prasad Rao, Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. Efficient tabling mechanisms for logic programs. In *International Conference on Logic Programming*, pages 697–711, 1995.

[85] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. In *Proceedings of the 16th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Brisbane*, 1990.

[86] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Controlling the search in bottom-up evaluation. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 273–287, Washington, USA, 1992. The MIT Press.

[87] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL—control, relations and logic. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 238–250, 1992.

[88] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Efficient bottom-up evaluation of logic programs. In J. Vandewalle, editor, *The State of the Art in Computer Systems and Software Engineering*, 1992.

[89] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. Implementation of the CORAL deductive database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 167–176, 1993.

[90] Raghu Ramakrishnan and S. Sudarshan. Top-down vs. bottom-up revisited. In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 321–336. MIT Press, 1991.

[91] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[92] Desh Ranjan, Enrico Pontelli, and Gopal Gupta. On the complexity of or-parallelism. *New Generation Computing*, 17(3):285–307, 1999.

[93] Louiqa Raschid and Jorge Lobo. A semantics for a class of stratified production system programs. *Journal of Automated Reasoning*, 12(3):305–349, 1994.

[94] Kenneth A. Ross. Modular stratification and magic sets for DATALOG programs with negation. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems*, pages 161–171. ACM Press, 1990.

[95] F. Sadri and F. Toni. Active behaviour in deductive databases. Technical report, Imperial College, 1996.

[96] Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. XSB: An overview of its use and implementation. In *Workshop on Programming with Logic Databases (Informal Proceedings), ILPS*, page 164, 1993.

[97] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. Automatic data structure selection in SETL. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 197–210. ACM Press, 1979.

[98] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. An automatic technique for selection of data representations in SETL programs. *ACM Transactions on Programming Languages and Systems*, 3(2):126–143, 1981.

[99] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley Publishing, Reading, Massachusetts, 1992.

[100] Dietmar Seipel and Helmut Thone. DISLOG - a system for reasoning in disjunctive deductive databases. In *Deductive Approach to Information Systems and Databases*, pages 325–343, 1994.

[101] Don Smith and Mark Utting. Pseudo-naive evaluation. In *Proceedings of the Tenth Australasian Database Conference*, Singapore, 1999. Springer-Verlag.

[102] Z. Somogyi, F. Henderson, T. Conway, A. Bromage, T. Dowd, D. Jeffery, P. Ross, P. Schachte, and S. Taylor. Status of the Mercury system. In *JICSLP'96 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, pages 207–218, 1996.

[103] Dennis M. Sosnoski. Java performance programming, part 1: Smart object-management saves the day. *Java World*, November 1994.

[104] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, Cambridge, Massachusetts, 1986.

[105] Rodney W. Topor. Views of objects and rules. In *Australasian Database Conference*, pages 1–13, 1994.

[106] Jeffrey D. Ullman. *Database and Knowledge-base Systems*, volume 1. Computer Science Press, 1988.

[107] J. Vaghani, K. Ramamohanaroa, D. B. Kemp, Z. Somogyi, and P. J. Stuckey. Design and overview of the Aditi deductive database system. In *Proceeding of the Seventh International Conference on Data Engineering*, pages 240–247, 1991.

[108] Jayen Vaghani, Kotagiri Ramamohanarao, David B. Kemp, Zoltan Somogyi, Peter J. Stuckey, Tim S. Leask, and James Harland. The Aditi deductive database system. *Very Large Data Bases Journal*, 3(2):245–288, 1994.

[109] Raja Vallee-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the SOOT framework: Is it feasible? In *Computational Complexity*, pages 18–34, 2000.

[110] A. Voronkov. Implementing bottom-up procedures with code trees: a case study of forward subsumption. Technical Report 88, Uppsala University, Uppsala, Sweden, 1994.

[111] A. Voronkov. The anatomy of Vampire (implementing bottom-up procedures with code trees). *Journal of Automated Reasoning*, 15(2):237–265, 1995.

[112] O. Wadge and W. Du. Chronolog(z): Linear-time logic programming. In *Proceedings of the Fifth International Conference on Computing and Information*, pages 545–549. IEEE Computer Society Press, 1993.

[113] Geoffrey I. Webb. Decision tree grafting. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 846–851, 1997.

[114] Daniel S. Weld. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999.

[115] Christoph Wernhard. System description: KRHyper. Technical report, Institut fur Informatik, Universitat Koblenz-Landau, 2003.

[116] Michael Winikoff. Hitch hikers guide to Lygon 0.7. Technical Report 96/36, University of Melborne, 1996.

[117] Carl Roger Witty. The Ontic inference language. Master's thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1995.

[118] Jens E. Wunderwald. Memoing evaluation by source-to-source transformation. In *Logic Program Synthesis and Transformation*, volume 1048 of LNCS, pages 17–32. Springer, 1995.

[119] C. K. Yuen. Hamming numbers, lazy evaluation, and eager disposal. *ACM SIGPLAN Notices*, 27(8):71–75, 1992.

[120] Carlo Zaniolo, Natraj Arni, and KayLiang Ong. Negation and aggregates in recursive rules: the LDL++ approach. In *Deductive and Object-Oriented Databases*, pages 204–221, 1993.