KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

TECHNIQUES FOR MORE EFFICIENT ILP DATA MINING ENGINES

Promotoren :
Prof. Dr. B. DEMOEN
Prof. Dr. ir. G. JANSSENS

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Remko TRONÇON

October 2006

KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

TECHNIQUES FOR MORE EFFICIENT ILP DATA MINING ENGINES

Jury :
Prof. Dr. ir. G. De Roeck, voorzitter
Prof. Dr. B. Demoen
Prof. Dr. ir. G. Janssens
Prof. Dr. ir. M. Bruynooghe
Prof. Dr. ir. F. Piessens
Prof. Dr. V. S. Costa (Universidade Federal do Rio de Janeiro)
Prof. Dr. P. Van Roy (Université Catholique de Louvain)

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Remko TRONÇON

U.D.C. 681.3*D34

October 2006

D/2006/7515/70
ISBN 90–5682–736–7

Abstract

The goal of data mining is to find rules (or hypotheses) that describe non-trivial relations, patterns or properties of large quantities of data, thus helping in understanding the data better. Inductive Logic Programming (ILP) is a relational data mining technique based on first order logic. Logic provides a powerful yet natural formalism for representing knowledge, allowing ILP to learn concepts that cannot be learned using less powerful data mining techniques. However, because of its high expressivity, the space of all possible hypotheses is also very complex, due to which the search for good hypotheses becomes a complex task.

One of the most important factors in the execution of ILP algorithms is the engine underlying the algorithm. This engine is responsible for evaluating candidate hypotheses (or queries) on the data, and provides primitives to the ILP algorithm for guiding the evaluation of queries. In this work, we present different techniques for optimizing the engines used by ILP algorithms.

We combine two existing, independent, and successful optimization techniques for query evaluation: the once transformation, which aims to avoid redundant execution within a single query, and query packs, which avoid redundancy in the execution of multiple queries.

The general approach to query evaluation is to compile the query to a more efficient version instead of executing the query directly. We study alternatives to this approach, and propose a more performant compilation technique, together with a lazy variant that only compiles parts of queries as they are needed.

Analysis and debugging of query execution is an important part of the development of more efficient query execution techniques. We present a trace-based technique for debugging and analyzing the execution step of ILP algorithms.

We present a study of trading off extra memory for execution time on different levels of ILP execution. These techniques include predicate tabling and program specialization, together with more ILP algorithm-specific techniques.

Dankwoord

Vanaf de eerste dag ik aan mijn doctoraat begon, keek ik al uit naar dit eigenste moment: het moment waarop ik mijn dank aan al de mensen die mij deze vijf jaar bijgestaan hebben eindelijk eens publiek mag uiten. Iedereen die ik ooit ontmoet heb heeft waarschijnlijk wel een steentje bijgedragen in dit werk. Hoewel ik niets liever zou doen dan al deze mensen bij naam vernoemen, ben ik helaas door plaatsgebrek (en mijn niet zo extensief geheugen) verplicht om het lijstje te beperken. Als je er niet tussen staat, weet toch dat ik je hulp apprecieer!

Eerst en vooral wil ik de mensen bedanken die al die jaren het dichtste bij mijn doctoraat gestaan hebben: mijn promotoren. Zonder Bart Demoen was er van dit boek geen sprake. Door zijn schijnbaar oneindige kennis, zijn vindingrijkheid, en zijn ervaring met mij te willen delen, is het maken van dit doctoraat een peulschil geweest. Ik ben er van overtuigd dat de lessen die hij mij geleerd heeft nog lang hun nut zullen bewijzen. Anderzijds was er de moederfiguur van mijn doctoraat, Gerda Janssens. Naast haar wetenschappelijke hulp kon ik doorheen die jaren altijd bij haar terecht als het niet meer helemaal meezat, of als ik gewoon eens een afwisselend gesprek wou over andere dingen. Zij is op alle gebieden een belangrijke steunpilaar geweest. Hoewel het vanzelfsprekend lijkt dat deze mensen dicht bij mijn doctoraat gestaan hebben, heeft ervaring mij geleerd dat iemand zelden zoveel hulp aangeboden krijgt als ik van elk van hen gekregen heb.

Als we een stapje verder gaan komen we bij Maurice Bruynooghe. Hij heeft mij de eerste belangrijke lessen van de wetenschapswereld geleerd, en heeft mij geïntroduceerd in de boeiende onderzoekswereld van (inductief) logisch programmeren. Een andere persoon die in diezelfde cruciale periode aan mijn zijde stond was Wim Vanhoof. Het feit dat iemand met wie ik op dezelfde golflengte sta het wetenschappelijk zo ver kunnen brengen heeft is iets wat mij al die jaren gesteund heeft.

En dan zijn er mijn hipP en ACE kompanen, Henk Vandecasteele en Jan Struyf. Henk heeft mij de donkerste hoekjes van hipP leren kennen, en heeft mij op regelmatige basis verbaasd met zijn ervaringen over efficiënte implementaties van virtuele machines. Bij Jan ben ik tot vervelens toe op bezoek geweest met al mijn vragen over ACE en ILP in het algemeen, en heeft die, zoals we Jan kennen, steeds met veel geduld beantwoord (zelfs als ik voor de derde keer hetzelfde vroeg).

De rest van mijn collega's hebben mij al die jaren ook het leven heel aangenaam gemaakt, en indirect met hun gesprekken geholpen bij mijn doctoraat. Ik denk hierbij in het bijzonder aan Tom(s), Anneleen, Tom(ma), en Yvette, maar er zijn er nog veel meer die mij minstens even veel geholpen hebben.

Ondertussen zijn we aan de grens gekomen van wetenschap en vriendschap. Hoewel ik technisch gezien Andrew een collega mag noemen, is die veel meer dan dat. Hij is over de jaren een van mijn beste vrienden geworden, met wie ik talloze middagen, namiddagen, en avonden tijdens de week gespendeerd heb, keuvelend over doctoraten, 'geek talk', emoties, films, muziek, en veel meer. Ook met Bettie heb ik menig middaggesprek gehad over alle belangrijke dingen van het leven, wat het leven van een doctoraatsstudent al wat lichter maakt.

Iets verder van de wetenschap, maar minstens even belangrijk doorheen dit werk, staan mijn vrienden en vriendinnen. Of ze nu in de buurt was of niet, Inge heeft mij altijd als beste vriendin bijgestaan doorheen de jaren. Ook Jeroen en Joris ben ik op diezelfde manier enorm veel dank verschuldigd. Met Jan en Gilles kon ik regelmatig even aan de realiteit ontsnappen naar het land van de muziek, om daarna herboren terug aan de slag te kunnen.

Ondanks het feit dat mijn familie er gans mijn leven al onvoorwaardelijk voor mij geweest is, zijn ze hiervoor nooit voldoende bedankt geweest. Wat mijn mama en papa voor mij betekend hebben is niet in woorden te vatten. Ze hebben mij alles gegeven wat ik ooit kon wensen, zonder hiervoor iets terug te krijgen, en zonder hun steun zou ik nooit geraakt zijn in mijn leven waar ik nu ben. Ook Ruben, Mimie en Papoe, en de rest van mijn familie hebben mij als persoon gevormd, en zijn altijd een steun geweest door er gewoon te zijn.

En tenslotte is er altijd die onzichtbare drijfkracht voor mij geweest, uit dewelke ik alle moed kon putten die ik nodig had om dit werk te kunnen vervolledigen.

Bedankt !

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Data Mining and Inductive Logic Programming

The amount of information stored in digital media is increasing very rapidly. The main goal of keeping information is to be able to extract new information from it: companies are collecting information about the shopping behavior of their customers, in order to make more effective marketing campaigns; search engines keep information about previous searches, in order to present personalized results; pharmaceutical companies keep information about experiments performed with newly designed drugs, in order to predict their behavior, ... Automatically extracting new information from large sets of data is the goal of *Data Mining* (Fayyad, Piatetsky-Shapiro, and Smyth 1996), also referred to as *Knowledge Discovery*. Data mining is a relatively recent topic, combining older techniques such as statistics, databases, and artificial intelligence. The task of data mining can be two-fold:

- *Predictive data mining* extracts information that allows to predict a target attribute of previously unseen data. For example, consider a database of credit card transactions, containing for each transaction information about whether it is fraudulent or not. The goal of a predictive data mining task is to discover rules that can predict if a new transaction is fraudulent.

- *Descriptive data mining* discovers rules 'describing' the data, typically in the form of interesting patterns. For example, a shop might want to mine the database of its customers in order to find classes of people with certain shopping behavior. Such information can help in deciding which products to allocate close to each other in the shop.

A widely used technique for data mining is *attribute-value learning*, where all the data to be analyzed is stored in one table (or 'relation'). Every row in a table represents a data instance, and every column an attribute of the data. The

simplicity of this representation has made attribute-value learning a very popular approach, for which many efficient techniques have been developed in the past. However, in practice, most data is stored using multiple relations, making attribute-value learning too limited to discover interesting patterns. Although under some very restricted conditions, learning over relational data can be reduced to attribute-value learning, this transformation is computationally too expensive for some real-life applications (De Raedt 1998).

A more powerful approach that overcomes these problems is *relational data mining*, which uses data stored in multiple tables directly. Not only does this avoid the (often infeasible) step of transforming data into one giant table, it also allows to learn rules that are not expressible in attribute-value systems. The most widely used approach to relational data mining is *Inductive Logic Programming* (Muggleton and De Raedt 1994), which uses *Logic Programming* (Kowalski 1974) as its foundation. In ILP, every instance of the data is represented as a logic program, predicates are used to encode the database relations, and the rules that have to be learned are expressed as queries (or sets of queries). ILP also makes it possible to express knowledge about the problem domain using logical predicates, providing a natural and expressive way of encoding knowledge. These factors make that ILP has been applied in a variety of more complex domains where classical learning techniques failed. Amongst the target application domains of ILP are medical applications (Davis, Burnside, Dutra, Page, Ramakrishnan, Costa, and Shavlik 2005; Srinivasan, Muggleton, Sternberg, and King 1996; Lavrač, Džeroski, Pirnat, and Krizman 1993), biology (Page and Craven 2003), chemistry (Blockeel, Dzeroski, Kompare, Kramer, Pfahringer, and Van Laer 2004), pharmaceutics (King, Muggleton, Lewis, and Sternberg 1992; Muggleton, King, and Sternberg 1992), software engineering (Bratko and Grobelnik 1993), aeronautics (Feng 1992), and CAD (Dolsak and Muggleton 1992).

1.2 Motivation and Contributions

Because ILP is based on first order logic, the hypotheses learned by ILP algorithms can represent complex rules. However, because of this powerful representation, the space of all possible hypotheses is also very complex. ILP algorithms generally use a generate-and-test approach to search for the target hypothesis: in each step (also called 'iteration') of the algorithm, a set of queries is generated and evaluated on the data set; the best queries are selected and refined in the next step of the algorithm, and the process continues until a certain condition is met. Because of the generate-and-test nature of ILP algorithms, evaluating all candidate hypotheses on the data typically is the bottleneck of the search problem. The goal of our work is to optimize this evaluation step by improving the performance of the engines underlying ILP algorithms. Previous research has shown that the ILP engine is an important factor in ILP algorithm execution (Blockeel, Dehaspe, Demoen, Janssens, Ramon, and Vandecasteele 2002). On one hand, the engine needs to provide primitives allowing the algorithm to

guide the evaluation process, and on the other hand, the engine itself needs to be designed for efficient execution. In this text, we present multiple contributions in this area:

- In the past, different query transformations were developed to improve ILP execution time. One of the most effective transformations is the *once transformation*. Independent of these transformations, the *query pack* execution mechanism was developed to optimize the execution of large sets of similar queries. Both independent optimizations yield significant speedups. The goal of our **first contribution** is to develop a new execution mechanism combining both the once transformation and the query pack execution mechanism, in order to get the best of both worlds. For this purpose, we introduce the notion of an *adpack*.

- Because queries need to be evaluated multiple times, the classical approach is to compile the query to a more efficient form, and execute the more efficient version. This approach indeed shows a significant improvement over dynamically interpreting queries. However, compiling a query is an expensive operation, and experiments point out that the time needed to compile a query dominates the total time of the evaluation process. As our **second contribution**, we present an optimized query interpretation scheme, which matches the speed of the compile-and-run approach, without requiring the expensive compilation step.

- As our **third contribution**, we present *control flow compilation*, a more flexible alternative for the compile-and-run approach. Control flow compilation combines a simple compilation step with fast execution. Besides improved performance, this approach also allows to reuse previously developed built-in instructions such as those required for execution of query packs. Moreover, the high degree of flexibility of this new form of compilation allows us to develop a lazy variant of this scheme that compiles parts of queries only when they are needed. This approach reduces the total compilation time even more.

Engine optimizations as the ones presented above are typically of a low-level nature, making them hard to debug. Tracing bugs in these execution mechanisms is made even harder by other factors, including the total size of the code base of the ILP algorithms. On the other hand, these factors also influence the feasibility of performing analysis of the ILP execution phase in order to detect bottlenecks. In our **fourth contribution**, we present an algorithm-independent way of performing automated debugging and analysis of ILP execution.

Our **final contribution** consists of investigating the benefits from trading memory for execution speed. We look at existing approaches of doing this on different levels of ILP execution, and make a qualitative comparison with other techniques that do not sacrifice memory for execution.

1.3 Organization of the Text

In this chapter (**Chapter 1**), we briefly introduced Inductive Logic Programming as a relational Data Mining technique, and highlighted some of the properties of ILP.

In **Chapter 2**, we present a technical overview of the concepts on which this text is based. We describe Prolog as a logic programming language, and discuss its implementation aspects. We also give a more in-depth description of ILP, together with concrete examples of ILP algorithms. Finally, we discuss two existing techniques for optimizing ILP execution.

In **Chapter 3**, we describe a new approach for optimizing ILP query execution by combining two existing independent techniques. This combination aims at yielding the advantages of both techniques, thus improving query execution time.

In **Chapter 4**, we develop alternatives for the classically used compile-and-run approach to query evaluation. The goal of these alternatives is to reduce the compilation time of queries, which encompasses a large share of the total query execution process.

In **Chapter 5**, we discuss an algorithm- and engine independent approach for analyzing and debugging ILP query execution. For debugging, an automated technique is presented that reduces the total time needed for a bug to expose to a minimum.

Chapter 6 studies trading off space for time on different levels of ILP execution. The discussed techniques store previously computed answers of complex predicates and of queries, such that these can be reused later.

The conclusions of the work presented in this text are given in **Chapter 7**. We give a high-level overview of the different techniques, and discuss their combination from a global point of view. We finally suggest possible directions for future work on the techniques developed throughout this text.

A graphical overview of the different chapters of this text can be seen in Figure 1.1.

1.4 Bibliographical Note

Some parts of this work have been published before. The following list contains the key articles. A complete publication list of the author can be found at the end of this text (page IX).

- **Chapter 3: Combining query packs with the once transformation**
  - R. Tronçon, H. Vandecasteele, J. Struyf, B. Demoen, and G. Janssens, *Query optimization: Combining query packs and the once-transformation*, Inductive Logic Programming, 13th International Conference, ILP 2003, Szeged, Hungary, Short Presentations (Horvath, T.

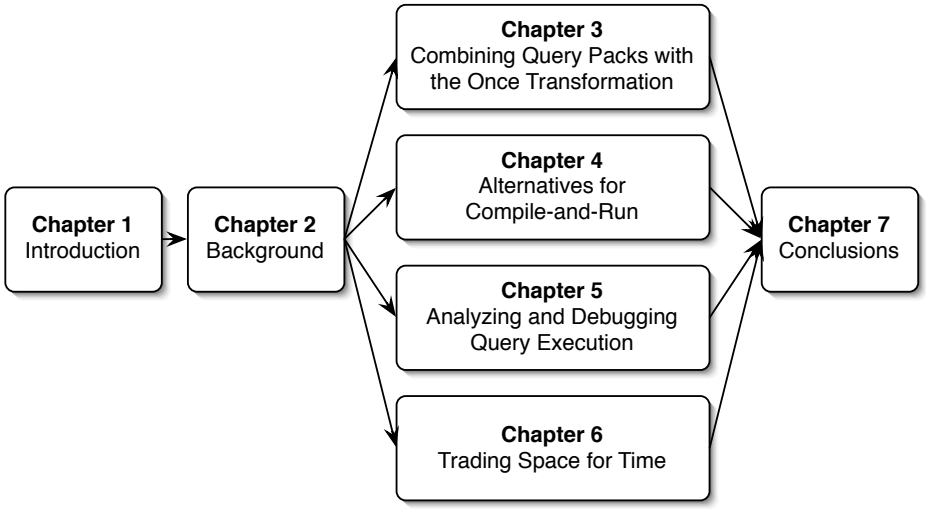


Figure 1.1: Organization of the text.

and Yamamoto, A., eds.), pp. 105-115, 2003.
http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=40938

- **Chapter 4: Alternatives for Compile-and-Run**
  - R. Tronçon, G. Janssens, and B. Demoen, *Alternatives for compile & run in the WAM*, Proceedings of CICLOPS 2003: Colloquium on Implementation of Constraint and LOgic Programming Systems (Lopes, R. and Ferreira, M., eds.), pp. 45-58, 2003.
    http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=41065
  - R. Tronçon, G. Janssens, B. Demoen, and H. Vandecasteele, *Fast frequent querying with lazy control flow compilation*, Theory and Practice of Logic Programming, 2006.
    http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=41995
- **Chapter 5: Analyzing and Debugging Query Execution**
  - R. Tronçon, and G. Janssens, *Analyzing and debugging ILP data mining query execution*, Proceedings of the Sixth International Workshop on Automated Debugging, AADEBUG 2005 (Jeffery, C. and Choi, J., Lencevicius, R., eds.), pp. 105-109, 2005.
    http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=41742
- **Chapter 6: Trading Space for Time**
  - W. Vanhoof, R. Tronçon, and M. Bruynooghe, *A fixed point semantics for logic programs extended with cuts*, Logic Based Program Syn-

thesis and Transformation, LOPSTR 2002, Revised Selected Papers (Leuschel, M., ed.), vol 2664, Lecture Notes in Computer Science, pp. 238-257, 2003
http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=40744

– R. Tronçon, B. Demoen, and G. Janssens *When tabling does not work*, Proceedings of CICLOPS 2006: Colloquium on Implementation of Constraint and LOgic Programming Systems, (Guo, H.-F. and Pontelli, E., eds.), pp. 18-31, 2006.

Chapter 2

Background: From Logic Programming to Inductive Logic Programming

In this chapter, we describe the fundamental concepts on which our work is based. Section 2.1 starts by giving a brief description of the concepts, syntax, and notational conventions of logic programs as the foundation of Inductive Logic Programming. In Section 2.2, we look at Inductive Logic Programming in more detail. Besides the general approach, we briefly discuss two specific ILP algorithms. Section 2.3 focuses on existing techniques for optimizing ILP execution, which form the basis of a large part of this work. Finally, we briefly describe the ACE/hipP data mining system in Section 2.4, which we use as the implementation platform for our developments.

2.1 Logic Programming

Logic Programming (Kowalski 1974) is a programming paradigm based on first order logic. The main motivation of logic programming is that by using logic to describe programs, they become easy to read, write, and maintain. Prolog is a concrete example of such a logic programming language, which we use as our platform for this text. We assume that the reader is familiar with Prolog and logic programming in general. In this section, we give a brief summary of the terminology and concepts of logic programming and Prolog. More information on Prolog itself can be found in the multitude of available literature on the language (Clocksin and Melish 2003; Bratko 2001; Sterling and Shapiro 1994; O'Keefe 1990). For a more in depth study of logic programming in general (including theoretical properties etc.), we refer to (Lloyd 1987).

We start by giving the basic syntax of Prolog in 2.1.1. Section 2.1.2 explains how logic programs are executed. Finally, Section 2.1.3 discusses the

```
parent(ann,bob).
parent(bob,chris).

ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

Figure 2.1: Example logic program.

implementation aspects of logic programming languages.

2.1.1 Syntax

A *logic program* is built out of a set of *predicates*, which describe logical relations. Each predicate is formed by a series of *clauses* of the form

$$H\text{: }-B_1, B_2, \ldots, B_n.$$

Such a clause represents the rule 'If B_1 and B_2 and ... and B_n hold, then H holds'. H is called the *head*, and $B_1, \ldots, B_n$ the *body* of the clause. Each B_i is a *body literal* (or *literal* for short). In the context of executing logic programs, a literal can also be called a *goal*. A clause without body literals is called a *fact*, and is written as 'H.'. A set of literals separated by ',' symbols is called a *conjunction*. Literals can also be separated by ';' symbols, in which case we talk about a *disjunction*. The reading of a disjunction $B_1; B_2$ is 'B_1 or B_2'. Figure 2.1 contains an example logic program describing the ancestor relations in a certain family. The program consists of a predicate `ancestor/2` (i.e. a predicate with name `ancestor` and arity 2), which is defined in terms of another predicate `parent/2`. `X`, `Y` and `Z` are *variables*, and `ann`, `bob` and `chris` are *constants*. Variables are always denoted by an uppercase letter or capitalized word, whereas constants are denoted by an lowercase letter or uncapitalized word. Together with *compound terms* (e.g. `f(a,X)`), variables and constants are all *terms*.

2.1.2 Semantics

Logic programs are 'executed' by running a *query* against them. A query has the same form as the body of a clause, and is denoted by:

$$?-\ Q_1, \ldots, Q_n.$$

Running this query consists in evaluating whether all of its subparts Q_i hold in the logic program. For example, the query

$$?-\ \texttt{ancestor(ann, X)}.$$

asks whether or not Ann is an ancestor of someone. Ann is indeed the ancestor of someone, and the query is therefore said to *succeed*. Moreover, an *answer substitution* is returned stating that this query holds for `X=bob` (and/or `X=chris`).

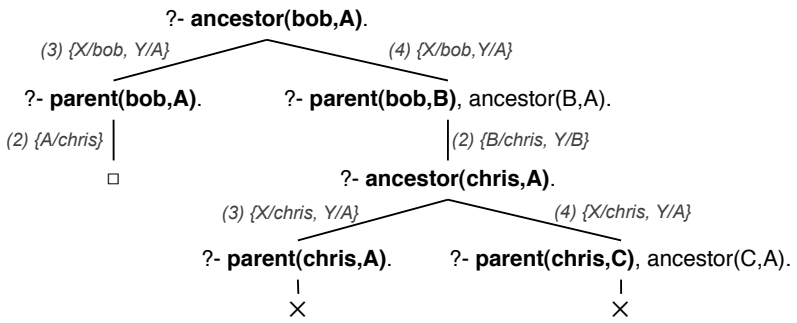


Figure 2.2: Execution tree for an example query against the program in Figure 2.1. The resolved goal and index of the resolving clause is shown, together with the unifier.

Contrary to the previous query, the query

$$?- \texttt{ancestor}(\texttt{chris}, \texttt{X}).$$

fails.

Concretely, the process of executing a query consists of selecting a literal of the query to resolve, substituting it with the body of one of the clauses in the logic program whose head matches with the literal, and repeating this process until no more literals remain (i.e. the query succeeds), or a selected literal cannot be resolved (i.e. the query fails). The result of such a query execution can be represented as a tree, where different branches indicate there was a choice in the clause to resolve the selected literal with. Such a choice is called a *choice point*. Whenever one of the branches fails, execution *backtracks* to the nearest choice point and tries the alternative branch(es). Prolog provides a built-in operation `!/0`, called the *cut*, which prunes away all the choicepoints up to (and including) the head of the clause in which the cut appears.

The execution tree for an example query against the program from Figure 2.1 is shown in Figure 2.2. The resolved goal is typeset in bold, and the index clause with which the goal is resolved is shown together with the resulting unifier. Prolog always selects the leftmost literal in a query for resolving, and selects resolving clauses from the program in a top-to-bottom order.

2.1.3 Implementation

A Prolog implementation (also called an *engine*) is a program that evaluates a given query against a given logic program. The *Warren Abstract Machine* (Warren 1983) describes a standardized design for implementing Prolog engines. The WAM proposes an internal memory layout, a compilation scheme for programs and queries, and a virtual instruction set for executing the compiled programs

and queries. In the this section, we briefly describe the most important aspects of the WAM. For an extended explanation of the WAM, we refer to the in-depth tutorial (Aït-Kaci 1991).

The internal memory layout of the WAM consists of four stacks:

- **Heap stack**: This is an array of *cells* containing the data structures created and used throughout the execution of the program. Cells on the heap contain a tag indicating which type of value the cell contains (e.g. variable, term reference, constant, ...).
- **Environment stack**: This stack contains *activation records* (also called *frames*) for every predicate call. Each frame contains amongst others a set of local (temporary) variables and a pointer to the top of the choice point stack at activation time.
- **Choice point stack**: For every choice point created during the execution process, a record is pushed on this stack. Besides a pointer to the code of the alternative to be tried upon failure, it also contains a copy of information of the current activation record, a pointer into the trail stack, and a pointer into the heap stack. This information is used to restore the original state upon backtracking.
- **Trail stack**: Every time a variable is bound, a record is pushed on this stack. This information is used to undo bindings upon backtracking.

Additionally, the WAM has a series of registers for various purposes: argument registers (denoted `A1`, `A2`, ...) for predicate argument passing, temporary registers (`X1`, `X2`, ...) for storing results of computations, a program counter pointing to the instruction being executed, and a continuation pointer pointing to the code to be executed after the current predicate is finished.

Before executing queries against a logic program, each predicate in the program is compiled into a set of instructions. These instructions can be divided into different types. The most important are:

- **Choice instructions**: These instructions control the way choice points are handled in the execution: `try*` instructions create a choice point, `retry*` instructions are executed when a (non-final) alternative of the choice point is executed, and `trust*` predicates handle the last alternative of a choice point (and destroy the choice point).
- **Control instructions**: These instructions control the aspects involving the calling of predicates. To call predicates, `call` and `execute` are used (depending on whether or not the environment needs to be saved); `allocate` and `deallocate` build and destroy an activation record; `proceed` continues execution after a predicate is finished. Some of these instructions occur together frequently, and are therefore combined into one instruction.

This is the case with `deallocate` and `execute`, which are combined into `deallex`.

- **Indexing instructions**: The `switch_on_*` instructions determine which code is to be executed, depending on the value of incoming arguments.
- **Cut instructions**: The `gettbreg`, `puttbreg`, `getpbreg` and `putpbreg` instructions handle the procedural aspects of the cut.
- **Get & put instructions**: These are used to construct and deconstruct terms on the heap. Besides the `get*` and `put*` instructions, there is also the `bldtvar` instruction, which builds a new temporary variable on the heap.

Notice that our description of the WAM above slightly diverges from the original WAM in some points:

- The original WAM interleaves the choice point stack and the environment stack, and therefore only uses 3 memory arrays.
- The WAM uses different instructions for handling cut.
- Merged instructions such as `deallex` are not described in the original WAM.
- The `bldtvar` instruction does not exist in the original WAM.

2.2 Inductive Logic Programming

2.2.1 Introduction

We briefly introduced Inductive Logic Programming in Chapter 1 as an approach to data mining, based on logic programming. We now describe ILP in more detail.

Given a *data set* E and *background knowledge* B of the target problem domain. The goal of an ILP algorithm is to find a hypothesis (or set of hypotheses) that describes the data in E. The hypothesis describing the data is itself a first order logic formula. The *data set* E consists of a (large) number of *examples*, each of which being a logic program. The background knowledge B describes knowledge about the target problem domain through predicate definitions. The background knowledge can therefore be seen as a common set of predicates that apply to every example in the data set.

In essence, ILP algorithms search through the space of all possible hypotheses (called the *hypothesis space*) for the target hypothesis. The *language bias* of an ILP algorithm imposes constraints on the structure of a hypothesis, narrowing the total (infinite) search space of all possible hypotheses. The language bias $\mathcal{L}$ imposes constraints on the form of valid hypotheses, thus limiting the number

$\mathcal{Q}$:= `Initialize`
repeat
    `Remove` H from $\mathcal{Q}$
    `Choose` refinements $r_1, \ldots, r_k$ to apply on H
    Apply refinements $r_1, \ldots, r_k$ on H to get $H_1, \ldots, H_n$
    Add $H_1, \ldots, H_n$ to $\mathcal{Q}$
    Evaluate $\mathcal{Q}$
    `Prune` $\mathcal{Q}$
until `Stop-criterion`($\mathcal{Q}$)

Figure 2.3: Generic ILP Algorithm.

of hypotheses that need to be considered. ILP algorithms search through the hypothesis space using a generate-and-test approach: hypotheses conforming to the language bias $\mathcal{L}$ are generated, and are then evaluated on the data set. Evaluating a hypothesis on the data set consists of evaluating it as a query on every example of the data set. A query Q is said to *cover* a given example E_i if the query Q succeeds on the logic program consisting of the example E_i and background knowledge B. The coverage of a query will be the deciding factor whether or not the hypothesis represented by the query is 'good enough', either as a final hypothesis, or as a temporary hypothesis to extend further in later iterations of the algorithm. Because the hypotheses generated by ILP algorithms are used as queries in the evaluation step, we will often use 'query' as a replacement for 'hypothesis' throughout this text.

Figure 2.3 shows the generic ILP algorithm as presented in (Muggleton and De Raedt 1994). $\mathcal{Q}$ is a set of *candidate hypotheses*, hypotheses that currently best describe the target concept to be learned. In every iteration of the algorithm, a hypothesis is selected from $\mathcal{Q}$, and is extended to yield a new set of hypotheses. This step is called the *refinement* step. The size of the refinements r_i depends on the *lookahead* setting of the algorithm: for a higher lookahead setting, the refinements will be larger, and the refined hypotheses will therefore be more complex. The newly generated set of hypotheses is then added to $\mathcal{Q}$, after which all the candidate hypotheses are evaluated. Depending on how well certain hypotheses describe the data set, they may or may not be pruned from $\mathcal{Q}$. This process is repeated until the hypotheses in the queue satisfy a certain *stop criterion*.

The algorithm from Figure 2.3 is generic in the `Initialize`, `Remove`, `Choose`, `Prune` and `Stop-criterion` operations. This means that a concrete instance of this algorithm has to fill in the actual implementation of these steps. We consider two such instances of ILP algorithms in the next sections: Tilde, a decision tree learner, and Warmr, a frequent pattern discovery algorithm. For a more detailed overview of various ILP algorithms, we refer to (Džeroski and Lavrač 2001).

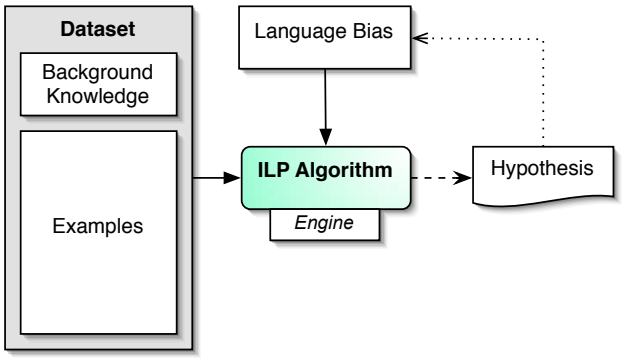


Figure 2.4: Overview of the ILP data mining process.

An overview of the ILP data mining process is depicted in Figure 2.4. The dataset, consisting of the background knowledge and the examples, is used by the ILP algorithm to find a good hypothesis describing the examples. The ILP algorithm uses the language bias to guide the search through the hypothesis space. If the hypothesis learned by the algorithm is unsatisfactory, the user can make changes to the language bias in order to find a better hypothesis. The typical ILP data mining process therefore consists of different cycles, where an algorithm is run on the data set with varying settings and language bias.

2.2.2 Tilde

Decision trees are trees containing a test in every internal node of the tree, and a class in every leaf. The goal of a decision tree is to be able to classify a new example by applying the tests contained in the nodes of the tree. To classify a certain example, one performs the test of the root node, and depending on the outcome of the test, chooses the appropriate branch of the node to continue the tests. After applying the tests of the subsequent nodes, a leaf with the target classification of the example is reached. An example decision tree for a simplified poker game is depicted in Figure 2.5. Classifying the given example results in a path from the root to a leaf, classifying it as a *pair*.

The TILDE algorithm (Blockeel and De Raedt 1998) builds first-order decision trees from a set of given examples. TILDE is a first-order extension of propositional decision tree learners such as C4.5 (Quinlan 1993) and CART (Breiman, Friedman, Olshen, and Stone 1984), whose internal node tests are limited to propositional tests only.

Figure 2.6 shows a high-level description of the TILDE algorithm. At the core of the algorithm is the GROW-TREE function which, given a set of examples and an accumulated query covering these examples, builds a decision tree

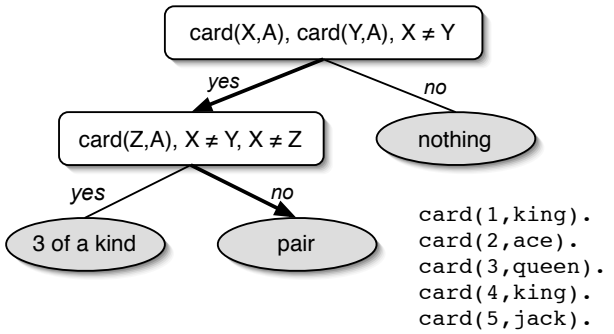


Figure 2.5: Decision tree for a simplified poker game. The classification of the given example is indicated by arrows.

function TILDE(E) :
    return GROW-TREE(E,`true`)

function GROW-TREE(E, Q) :
    $\mathcal{Q}_r$ = REFINE(Q)
    Q_b = OPTIMAL-SPLIT($\mathcal{Q}_r$,E)
    if STOP-CRITERION(Q_b, E) :
        return leaf(INFO(E))
    else :
        $E_+ := \{e \in E \mid Q_b \text{ covers } e\}$
        $E_- := \{e \in E \mid Q_b \text{ does not cover } e\}$
        T_{left} := GROW-TREE(E_+,Q_b)
        T_{right} := GROW-TREE(E_-,Q)
        return node($Q_b - Q$, T_{left}, T_{right})

Figure 2.6: The TILDE algorithm.

Example 1
`buys(beer).`
`buys(chocolate).`
`buys(nuts).`

Example 2
`buys(cheese).`
`buys(bread).`
`buys(wine).`
`buys(chocolate).`

Example 3
`buys(beer).`
`buys(cheese).`
`buys(nuts).`
`buys(bread).`

Pattern	Frequency
`buys(beer),buys(nuts)`	2/3
`buys(bread),buys(cheese)`	2/3
`buys(beer),buys(wine)`	0/3

Figure 2.7: Market basket analysis example.

for these examples. This is done by refining the accumulated query, and selecting the best query from the resulting set. The selection function `Optimal-Split` chooses the query Q_b that divides the set of queries into sets that give the best 'improvement'. By dividing the examples into homogenous sets, the classification process will reach a leaf very fast, and therefore TILDE will generate the smallest trees. After selecting the query with optimal split, a subtree is built for both outcomes of Q_b (i.e. success or failure): for the examples covered by the query, Q_b is further refined in the left subtree; the right subtree will refine the original query Q further on the remainder of the examples. Both trees are combined into an internal node with as corresponding test the selected query Q_b minus Q, the query accumulated so far.

Notice that during the execution of GROW-TREE, REFINE is applied on exactly one query, after which the resulting queries are evaluated. Hence, all the queries evaluated during one iteration of the algorithm will be refinements of the same query, and will therefore have an identical prefix.

2.2.3 Warmr

Discovering frequently occurring patterns in a large set of data is a very popular data mining task. One example is *market basket analysis*, where one analyzes the purchases of all customers to find out which combination of products are often bought together. Figure 2.7 shows a concrete basket analysis for a set of 3 customers. The example also shows three patterns, together with their *frequency* (the relative number of examples in the data set the pattern covers). For example, beer and nuts are often bought together, whereas beer and wine never occur in one basket.

WARMR (Dehaspe and Toivonen 1999) is an ILP algorithm that discovers such patterns in a set of examples. It is a first-order upgrade of the propositional APRIORI (Agrawal, Mannila, Srikant, Toivonen, and Verkamo 1996) algorithm. The goal of WARMR is the following: given a set of examples E and a minimum support frequency *minfreq*, find all queries that cover at least $\mathit{minfreq} \cdot |E|$

function WARMR(E, *minfreq*) :
    $d := 1$ *# Current level*
    $\mathcal{Q}_1 := \{\texttt{true}\}$ *# Candidate queries*
    $\mathcal{F} := \emptyset$ *# Frequent queries*
    $\mathcal{I} := \emptyset$ *# Infrequent queries*
    while not empty(Q_d) :
        $\mathcal{F} := \mathcal{F} \cup \{Q \in \mathcal{Q}_d \mid \text{FREQ}(Q, E) \geq minsup\}$
        $\mathcal{I} := \mathcal{I} \cup \{Q \in \mathcal{Q}_d \mid \text{FREQ}(Q, E) < minsup\}$
        $\mathcal{Q}_{d+1} :=$ REFINE($\mathcal{Q}_d$,$\mathcal{F}$,$\mathcal{I}$)
        $d := d + 1$
    return $\mathcal{F}$

function FREQ(Q,E) :
    return $\#\{e \in E \mid Q \text{ covers } e\}/|E|$

Figure 2.8: The WARMR algorithm.

examples. The high-level WARMR algorithm can be seen in Figure 2.8. The algorithm proceeds level-wise through the query space. Every level has a set of generated candidate queries, which are the result from refining queries from the previous iteration. Based on the frequency of the query, every candidate query is added to the set of frequent or infrequent queries. The algorithm finishes when no more frequent queries are found.

The refinement step in WARMR depends on the set of frequent and infrequent queries found so far. This is because WARMR avoids generating queries that are either equivalent to previously generated queries, or refinements of infrequent queries (since those queries will be infrequent as well). However, these extra checks are computationally very expensive, which makes the refinement step of WARMR potentially a time-consuming step (contrary to the refinement step of other algorithms such as TILDE). Another difference with TILDE is that not one query, but a set of queries is refined during every iteration. This means that the queries evaluated do not necessarily have an identical prefix, although they do show similarities, as they typically are descendants of a selected set of queries.

2.3 Optimizing ILP Execution

The run time of the generic algorithm from Figure 2.3 can be approximated by the following formula:

$$T_{run} = \sum_{i=1}^{n} \underbrace{|R_i| \cdot T_{gen}}_{\text{refinement}} + \underbrace{|E_i| \cdot |R_i| \cdot T_{exec}}_{\text{evaluation}}$$

with n the number of iterations of the ILP algorithm, T_{run} the *(algorithm) run time* of the algorithm, T_{gen} the average time needed to generate a refinement

of a query, T_{exec} the time needed to execute a query, $|R_i|$ the number of queries generated in iteration i, and $|E_i|$ the number of examples to consider in iteration i (the latter being equal to the total number of examples for WARMR). We call the component $|E_i| \cdot |R_i| \cdot T_{exec}$ the *(query) evaluation time*.

Looking at the above formula, one can see that improving the efficiency of the algorithm can be done on different levels. The impact of the first half of the formula can be reduced by reducing $|R_i|$, which is achieved by designing a good language bias for the target hypotheses. The factor T_{gen} is usually very small (with the exception of a few algorithms like WARMR). The second half of the formula is the most important, as it depends on the total number of examples $|E_i|$, which can typically grow very large for real-life data mining problems. Besides $|E_i|$, the other factors influencing the evaluation are the number of candidate queries to be evaluated and the time needed to execute them on the examples. $|E_i|$ can be reduced by applying sampling techniques on the data set, thus selecting only a subset of relevant examples (Srinivasan 1999). The size of $|R_i|$ depends on the language bias, and can be reduced by restricting the hypothesis space further. Reducing the factor T_{exec} is the main goal of this work.

The following sections describe two techniques for optimizing query evaluation in the context of ILP data mining. Different optimizations for reducing the query evaluation time exist. On one hand, there are query transformations such as the ones from (Costa, Srinivasan, Camacho, Blockeel, Demoen, Janssens, Struyf, Vandecasteele, and Van Laer 2002) which, given a single query, optimize the evaluation of the query by transforming it into an equivalent query with better performance. One such transformation is the *once transformation*, which we describe in more detail in Section 2.3.1. Another class of techniques are *multi query optimizations* (as opposed to single query optimizations). The fundamental approach for this class of optimizations is the query pack approach, described in Section 2.3.2.

2.3.1 Once transformation

Because ILP algorithms only use information about the coverage of queries on an example (i.e. whether a query succeeds or not), and not the actual binding of the query variables itself, we can cut away all the choice points created during the evaluation of the query once it is successful. Therefore, we end every query with a cut, as is illustrated in the query below. In the future, however, we will omit writing the trailing cut.

Now consider the following query:

$$?- \ \mathtt{p(X,Y,Z)}, \overbrace{\mathtt{q(X,Y,U)}, \mathtt{r(Y,V)}}^{a}, \overbrace{\mathtt{q(X,Z,W)}}^{b}, !.$$

If one knows (e.g. through program analysis) that `p/3` grounds its first argument and leaves its second and third argument independent of each other, one can see that parts a and b of the query do not share any unbound variables

function ONCE-TRANSFORM(Q) :
    return ONCE-TRANSFORM(Q,$\emptyset$)

function ONCE-TRANSFORM(Q,$\mathcal{V}$) :
    if $Q = \emptyset$:
        return $\emptyset$
    else :
        $Q_{result} := \emptyset$
        $\mathcal{P}$:= PARTITION-INDEPENDENT(Q,$\mathcal{V}$)
        for each q in $\mathcal{P}$:
            (q_{prefix}, q_{suffix}) = SPLIT-PREFIX(q)
            q_{result} := ONCE-TRANSFORM(q_{suffix}, $\mathcal{V} \cup vars(q_{prefix})$)
            $Q_{result} := Q_{result} \cup \text{once}((q_{prefix}, q_{result}))$
        return Q_{result}

Figure 2.9: Once-transformation.

anymore. This means that the execution of a does not influence the execution of b: if the calls to `q(X,Y,U)` and `r(Y,V)` succeed, and `q(X,Z,W)` fails, it is useless to try alternative solutions for `q(X,Y,U)`,`r(Y,V)`, as they have no influence on the success of `q(X,Z,W)`. Such redundant backtracking can have a big influence on the total execution time, especially if part a has many solutions. This needless backtracking can be avoided by applying the *once transformation* (Costa, Srinivasan, Camacho, Blockeel, Demoen, Janssens, Struyf, Vandecasteele, and Van Laer 2002) on the query. This transformation first identifies all independent subparts of a query, and embeds them in a `once/1` call. The definition of `once/1` is simply

```
once(G) :- call(G), !.
```

meaning that `once/1` cuts away all the remaining choice points after the execution of its argument has finished. For the above query, this would result in:

$$?- \ \texttt{p(X,Y,Z)},\ \texttt{once((q(X,Y,U),r(Y,V)))},\ \texttt{q(X,Z,W)}.$$

Notice that the `once` is omitted for the last goal, since its choice points will be cut away by the trailing cut anyway. In this example, `once/1` only occurs at the top level of the query. However, the once transformation can also generate nested `once`s to avoid backtracking inside an independent subpart of the query.

A high-level implementation of the once transformation can be seen in Figure 2.9. The algorithm first partitions the query Q into a set of independent subparts of the query (assuming that the variables in $\mathcal{V}$ do not cause dependencies). For every part, a prefix that grounds variables in the rest of the query is split off, and the once transformation is called recursively on the remainder. The recursive call to the once transformation assumes the variables occurring in

the prefix are ground in the suffix, and therefore disregards all these variables. Both the prefix and the new suffix are finally combined and embedded in a `once`, and the result is added to the transformed query.

Notice that PARTITION-INDEPENDENT can be implemented in a way that it leaves the order of the goals in the query intact, but it can also be implemented such that goals in a query are reordered to yield more independent parts.

The experiments from (Costa, Srinivasan, Camacho, Blockeel, Demoen, Janssens, Struyf, Vandecasteele, and Van Laer 2002) indicate that applying the once transformation on queries can result in significant speedups: for certain algorithms and datasets, measurements indicated improvements up to several orders of magnitude.

2.3.2 Query Packs

Queries generated in a specific iteration of an ILP algorithm are all refinements of one or more queries from a previous iteration. A consequence of this is that the generated queries are very similar. Suppose, for example, that the following query is chosen for refinement

$$?- \ \mathtt{p(X,Y)}, \mathtt{q(Y,Z)}.$$

and that the refinements `r(Z)` and `s(Z)` are applied to it. This results in the following queries to be evaluated:

$$?- \ \mathtt{p(X,Y)}, \mathtt{q(Y,Z)}, \mathtt{r(Z)}.$$
$$?- \ \mathtt{p(X,Y)}, \mathtt{q(Y,Z)}, \mathtt{s(Z)}.$$

Executing these queries separately means that the execution of both queries has a part in common: the call to (`p(X,Y)`,`q(Y,Z)`) generates the same answers for the variables `X`,`Y` and `Z` in both queries. To share the execution of these goals, both queries can be laid out in a disjunction by applying left factoring on the queries:

$$?- \ \mathtt{p(X,Y)}, \mathtt{q(Y,Z)}, (\mathtt{r(Z)}; \mathtt{s(Z)}).$$

While this indeed shares the execution of the common prefix, there is still redundancy: since we are only interested in the success of the queries, we want to avoid that, after the success of one of the branches of the disjunction, it is ever executed again. To achieve this, *query packs* are introduced in (Blockeel, Dehaspe, Demoen, Janssens, Ramon, and Vandecasteele 2002) as an alternative kind of disjunction with special semantics: the disjunction 'cuts away' every succeeding branch, thus avoiding it is executed again after success. We denote a query pack disjunction by $;_p$. For example, transforming the above set of queries into a query pack results in:

$$?- \ \mathtt{p(X,Y)}, \mathtt{q(Y,Z)}, (\mathtt{r(Z)};_p \mathtt{s(Z)}).$$

A high-level execution algorithm for a query pack is given in Figure 2.10. For

```
function EXECUTE-PACK(Q, θ) :
    for each σ ∈ EXECUTE-CONJ(conj(Q)θ)
        for each Q_child ∈ children(Q) :
            if EXECUTE-PACK(Q_child, σ) :
                children(Q) := children(Q) \ Q_child
        if children(Q) = ∅ :
            return success
    return fail
```

Figure 2.10: Query pack execution.

$$? - \ \mathtt{p(X,Y)}, \mathtt{q(Y,Z)}, (\mathtt{r(Z)} ;_p \mathtt{s(Z)}).$$

```
    pack_init 1
    ...
    call p/2
    ...
    call q/2
    ...
    pack_try 1
L1: ...
    call r/1
    pack_success
L2: ...
    call s/1
    pack_success
```

Pack	Children
1	@L1
	@L2

Figure 2.11: Compiled query pack and its pack table.

every solution of the prefix of the pack (a conjunction), the children pack-ors of the query pack are executed. Every successful child pack is permanently removed from the set of children from the pack. When the pack has no more children, success of the pack is propagated upwards.

Pack implementation results (Demoen, Janssens, and Vandecasteele 1999) indicate that the overhead of a high-level implementation of the query pack execution mechanism destroys the benefits of the approach. To gain speedups from query packs, their execution mechanism has to be implemented in the core of the execution engine. This is done by introducing dedicated WAM choice instructions for dealing with the special semantics of the query pack disjunction. For example, the compiled version of the above query pack can be seen on the left of Figure 2.11. `pack_init` initializes the data structures used during the execution of the pack. These data structures are needed to keep track of which branches of the pack still need to be executed. The `pack_try` instruction

creates a special pack choice point, and starts executing the first child branch of the pack. Pointers to the code of the different child branches are stored in a data structure called the pack table, as depicted on the right of Figure 2.11. After success of a branch, the `pack_success` instruction removes the current branch from the pack table, and backtracks to the pack choice point, which then continues with the next branch.

The experiments from (Blockeel, Dehaspe, Demoen, Janssens, Ramon, and Vandecasteele 2002) indicate that query packs improve the speed of query execution drastically (typically around an order of magnitude).

2.4 The ACE/hipP Data Mining System

The *ACE Data Mining System* (ACE 2006) is a combined ILP system for ILP data mining, developed at the K.U.Leuven. It provides implementations of several ILP algorithms, including the TILDE and WARMR algorithms described in Sections 2.2.2 and 2.2.3. The largest part of ACE is written in Prolog, and works on top of *hipP* (hipP 2006), a high performance Prolog engine designed specifically for ILP algorithms.

hipP is a WAM-based Prolog interpreter, implemented in C, and with a Prolog compiler written in Prolog itself. hipP provides support for several ILP-specific extensions such as the query packs described in Section 2.3.2.

Chapter 3

Combining Query Packs with the Once Transformation

3.1 Introduction

In the previous chapter, we showed two optimizations for executing queries in ILP. On one hand, query packs (Section 2.3.2) optimize the execution of a set of similar queries by sharing the execution of common parts, while on the other hand query transformations such as the once transformation (Section 2.3.1) transform a single query to optimize its execution. Both approaches were developed independently, and each one yields significant speedups (up to several orders of magnitude for certain applications). The question therefore arises whether combining both approaches would give the best of both (indpedendent) worlds with respect to optimizing execution. However, combining query packs with query transformations is difficult, because query transformations usually have a negative effect on the structure of the pack. Since the transformations alter the form of queries (e.g. by reordering goals or by introducing `once/1` predicates), it can be that queries no longer have common parts of which execution can be shared by a query pack.

In this chapter, we focus on combining query packs with the once transformation. While there exist several other query transformations (Costa, Srinivasan, Camacho, Blockeel, Demoen, Janssens, Struyf, Vandecasteele, and Van Laer 2002), the once transformation is the most interesting one to consider, since it gives the highest speedups yet is the hardest one to combine with the query pack execution mechanism. In order to combine query packs with the once transformation, we introduce the concept of an *adpack*, which is a variant of a query pack with special support for once transformed queries.

The organization of this chapter is as follows: Section 3.2 starts by explaining

the intuition behind our proposed approach, by using a simple illustrative example. In Section 3.3, we give the definition and the detailed execution mechanism of adpacks. Section 3.4 illustrates the execution mechanism on an elaborated example. Techniques for transforming a set of queries into an adpack are discussed in Section 3.5. Section 3.6 gives an in-depth description of the techniques used for efficient execution of adpacks. Further optimizations to the basic techniques are briefly discussed in Section 3.7. We then perform an experimental evaluation of the proposed techniques in Section 3.8, after which we conclude in Section 3.9.

3.2 Intuition

We start by sketching the intuition behind our approach to combine query packs with the once transformation through an example.

Suppose that an ILP algorithm has to evaluate the set of queries shown in Figure 3.1(a). A first observation is that both queries have the calls to `p/2` and $\texttt{q}_1\texttt{/2}$ in common, which results in repeated execution of these goals when the queries are executed separately. Transforming these queries into a query pack yields the pack from Figure 3.1(b), which shares the execution of the queries' common prefix. On the other hand, observe that the calls to $\texttt{q}_1\texttt{/2}$ and $\texttt{q}_2\texttt{/2}$ in the first query do not share any variables with the call to `r/2` (under the assumption that a literal always grounds its free variables). Hence, the execution of `r/2` is independent of its two predecessors, and backtracking to either $\texttt{q}_1\texttt{/2}$ or $\texttt{q}_2\texttt{/2}$ would yield no new solutions to `r/2`. Applying the once transformation on this query ensures that execution backtracks directly to `p/2` if a solution for `r/2` cannot be found. This is illustrated in Figure 3.1(c). However, while this avoids redundant execution present in both the original queries, execution of the common prefix is not shared as in the query pack version.

In a simple attempt to combine query packs with the once transformation, one can apply the query pack transformation on the once transformed queries from Figure 3.1(c), which would result in Figure 3.1(d). While this indeed shares the execution of `p/2`, the literal `q/2` is still executed separately for both queries.

To gain the benefits from both the once transformation and query packs, we introduce the notion of an *adpack*. An adpack of a set of once transformed queries is obtained by constructing a query pack as before, without taking into account the `once`s, and then marking the scope of every `once` by `activate/1` and `deactivate/1` goals. For our example, this would result in the pack shown in Figure 3.1(e). Notice that we use $;_a$ as the adpack counterpart of the pack-or notation $;_p$. A `deactivate/1` goal indicates that all alternatives up to the corresponding (i.e. with the same identifier as argument) `activate/1` goal are not relevant for finding solutions for the remainder of the current branch. As in query packs, the adpack-or node cuts away branches when all their children are successful. Additionally, it also prohibits execution from entering branches when they are in the scope of a `once` and their execution has no influence on

```
?- p(X,Y), q1(X,A), q2(A,B), r(Y,C).
?- p(X,Y), q1(X,A), s(Y,A).
```

(a) Original queries.

```
?- p(X,Y), q1(X,A),
        ( q2(A,B), r(Y,C)
        ;p s(Y,A)).
```

(b) Query pack.

```
?- p(X,Y), once((q1(X,A), q2(A,B))), r(Y,C).
?- p(X,Y), q1(X,A), s(Y,A).
```

(c) Once-transformed queries.

```
?- p(X,Y),
        ( once((q1(X,A), q2(A,B))), r(Y,C).
        ;p q1(X,A), s(Y,A)).
```

(d) Query pack of once transformed queries.

```
?- p(X,Y), activate(1), q1(X,A),
        ( q2(A,B), deactivate(1), r(Y,C)
        ;a s(Y,A)).
```

(e) ADPack: Query pack adorned with `activate/1` and `deactivate/1`.

Figure 3.1: Different transformations on a set of queries.

the success of the remainder of the branch.

For example, consider execution of the adpack in Figure 3.1(e). The `deactivate` for `once`((`q`$_1$(`X`,`A`),`q`$_2$(`A`,`B`))) first only removes remaining alternatives for `q`$_2$, as the other branch with `s` still needs to be able to backtrack to `q`$_1$ when necessary. When `r` fails, we backtrack to the closest adpack-or node. As long as `s` fails, we further explore the alternatives for q$_1$, but without considering the (`q`$_2$,`r`) branch (because of the blocked once). Notice that it is as if the (`q`$_2$,`r`) branch is removed from the pack. This removal is not definitive, and will be undone as soon as the execution backtracks to a call before the `activate` (i.e. before the original `once`). When `s` succeeds, the branch is pruned from the adpack-or, leaving only the (`q`$_2$,`r`) branch. This means that the `once` can now have its full scope, and the remaining alternatives for `q`$_1$ are removed. When execution finally backtracks to `p`, the (`q`$_1$,`q`$_2$`,r`) branch is executed again, and the temporary removal is undone.

To summarize, we can say that the success of a branch permanently removes the branch from the adpack, whereas a `deactivate` only temporarily removes a branch from the adpack.

3.3 ADPack Execution

In this section, we describe the high-level execution of adpacks. First, we start by defining the exact syntax of an adpack.

Definition 1 (ADPack) *An adpack is a term of the following form:*

$$
\begin{array}{rcl}
\mathit{ADPack} & := & \mathit{Subgoal}\ \mathit{ADPackOr} \mid \mathit{Subgoal} \\
\mathit{ADPackOr} & := & \mathit{ADPack}\ ;_a \mathit{ADPack}\ [\,;_a \mathit{ADPack}\,]+ \\
\mathit{Subgoal} & := & \mathit{Literal} \mid \texttt{activate}(\mathit{ID}) \mid \texttt{deactivate}(\mathit{ID}) \\
 & & \mid (\mathit{SubGoal}\texttt{,}\mathit{SubGoal}) \\
\mathit{ID} & := & i \text{ (with } i \text{ a natural number)} \\
\mathit{Literal} & := & \text{a term}
\end{array}
$$

Additionally, the adpack should satisfy the following restrictions:

- *All `activate/1` and `deactivate/1` literals occur in pairs. The `activate` and `deactivate` in such a pair have the same unique identification number as argument.*

- *For each `activate`/`deactivate` pair, both literals should always be a part of the same path from the root of the adpack to a leaf, with the `activate` preceding the `deactivate`. Additionally, both are not located on the same branch, and as such are separated by at least one adpack-or*[1]*.*

[1] An `activate`/`deactivate` pair on the same branch can be replaced by a `once/1`.

- *`activate`s and `deactivate`s are properly nested: If `deactivate`(i) comes before a `deactivate`(j) on the path from the root of the adpack to a leaf, then the `activate`(i) comes after the `activate`(j) on that path.*

An example adpack can be seen in Figure 3.1(e). An adpack consists of several conjunctions (*branches*), grouped together in adpack-ors by $;_a$. A branch that does not end with an adpack-or is called a *leaf* of the adpack.

Execution of an adpack can be described in terms of two phases: *forward execution*, where the goals of the adpack are executed, and *backtracking*, which occurs when a goal fails or the end of a branch is reached. During the execution of an adpack, we remember for each branch whether it is open or closed (temporarily disabled), and whether the branch is successful. Initially, every branch is open and unsuccessful.

The *forward execution* of an adpack consists of executing the goals on the branches as normal, except for the two special goals `activate/1` and `deactivate/1`. In these particular cases, the following actions have to be taken:

- `activate`(*Id*)
  Let *DeactBranch* be the branch containing the `deactivate` corresponding to this `activate` (i.e. the `deactivate` with the same *Id*). If *DeactBranch* is still unsuccessful, do the following:

  1. Open all branches on the path from the current branch to *DeactBranch*.
  2. Remember the current choicepoint, and associate it with this `activate`.

  Finally, continue execution.

- `deactivate`(*Id*)
  Cut away all choicepoints on this branch, close the current branch, and continue execution.

When forward execution reaches an adpack-or, the set of children that have to be tried is determined. This is exactly the set of child branches that are marked open, but are still unsuccessful. A child of this set is chosen, it is removed from the set, a choicepoint is created (which enables backtracking such that the remainder of the set can be executed), and the selected child is executed.

Finally, when a leaf of the adpack finishes, success of the current branch is registered, all its choicepoints are cut away, and execution backtracks to the adpack-or node to which this branch belongs.

When *backtracking* occurs in the WAM, the only thing that has to be done is restoring a previous state by (amongst others) undoing bindings, and to select the next alternative to be tried. However, when backtracking to an adpack-or, more complicated actions have to be taken. We distinguish the following 4 situations (in order) when backtracking to an adpack-or, with their corresponding actions:

1. *There is still a branch that has to be tried:* Remove the untried branch from the set of children that have to be tried, and execute it.

2. *All branches are successful:* Mark success of the branch to which this adpack-or belongs, cut away all its choicepoints, and backtrack further.

3. *All branches are closed or successful:* First, the most recent relevant choicepoint has to be determined. This is exactly the corresponding choicepoint (saved during the forward execution) of the last `activate` on the parent branch with a corresponding `deactivate` on a closed, unsuccessful branch. If no such `activate` is found, the choicepoint of the parent adpack-or is taken.

   If the most recent relevant choicepoint is the previous adpack-or, close the parent branch, cut away all its choicepoints, and backtrack to the previous adpack-or. Otherwise, cut away all alternatives up to the most recent relevant choicepoint, and backtrack to it.

4. *There is still an open, unsuccessful branch:* Since this branch has been tried before, it has failed without deactivation or success. No special actions have to be taken here: backtrack to the previous choicepoint.

A reference implementation of this execution mechanism is available in the form of the meta-interpreter from Appendix B (page 137).

3.4 Elaborated Example

We now illustrate the execution of adpacks on a larger example, covering most aspects of the execution. The example adpack and the fact database on which we run the pack are given in Figure 3.2.

Executing the first branch of the adpack binds the variables `X` and `Y` to 1 (due to the calls to `a` and `b`). Since the branches containing the `deactivate`s of `activate`s 1 and 2 are initially unsuccessful, and since all branches are initially open, the two `activate` goals on the branch do nothing but remember the last choicepoint, being the choicepoint of `a` for both `activate`s. At the end of the first branch, the first adpack-or is reached, and its first child is chosen for execution. After the succeeding calls to `c` and `d`, the end of the query is reached, and so the branch is marked as successful (denoted by a dot), and execution backtracks to the parent adpack-or:

```
?- a(X), activate(1), activate(2), b(X,Y),
      •  once(c(Y)), d(Y)
   → ;a e(Y,Z),
            (   deactivate(2), f(X)
            ;a g(Y,Z), deactivate(1), d(X) )).
```

```
?- a(X),b(X,Y), once(c(Y)), d(Y).
?- a(X), once((b(X,Y), e(Y,Z))), f(X).
?- a(X), once((b(X,Y), e(Y,Z), g(Y,Z))), d(X).
```

(a) Set of 3 once transformed queries

```
a(1).   d(1).
a(2).
        e(1,1).
b(1,1). e(2,1).
b(2,1).
b(1,2). f(2).

c(1).   g(2,1).
c(2).
```

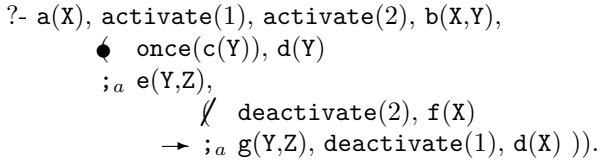
(b) Example program

```
?- a(X), activate(1), activate(2), b(X,Y),
        (   once(c(Y)), d(Y)
        ;a  e(Y,Z),
                (   deactivate(2), f(X)
                ;a  g(Y,Z), deactivate(1), d(X) )).
```

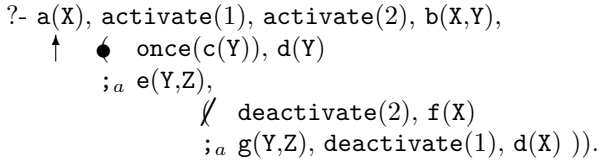
(c) ADPack of the 3 queries

Figure 3.2: Elaborated Example.

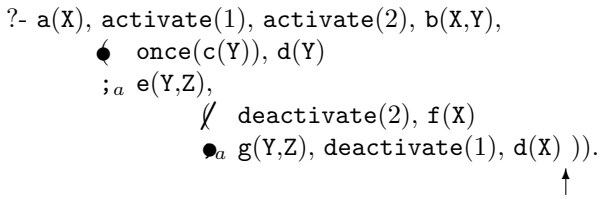
The following branch is executed, resulting in a successful call to `e` (thus binding `Z` to 1) and encountering the next adpack-or. Again, the first branch of the latter is executed, causing the branch to be closed immediately due to the `deactivate`. The next call to `f` fails, causing execution to backtrack back to the parent adpack-or:



Execution of the last branch fails immediately due to the call to `g`, returning execution to the parent adpack-or. Since all branches of the current adpack-or have been tried and not all of them are closed, normal backtracking occurs, returning the execution in the parent adpack-or (since `e` has no alternatives). There, the same situation arises, resulting in backtracking to the nearest choicepoint, `b(X,Y)`:

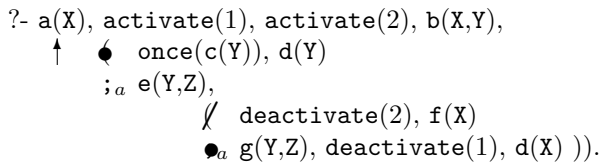


Forward execution now binds `Y` to 2, and then arrives in the first adpack-or again. Now, only the second branch is still open for execution (since the first branch is already successful), and is therefore executed. After calling `e` (again binding `Z` to 1), we arrive in another adpack-or with one open branch. Indeed, the first branch is still closed, because execution didn't backtrack over `activate(2)` (the start of the corresponding `once`) yet, so the alternative solutions are not interesting for this branch. Entering the only open branch, execution now reaches the end of the branch, and so it is marked as successful:

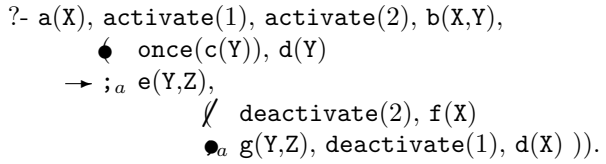


After backtracking to the parent adpack-or, execution is in an adpack-or where all branches are closed (or successful). Execution should now backtrack to a point where an `activate` opens one of the children branches of this adpack-or. Since there is no such backtrack point on the parent branch, we close the parent branch as well, and look higher up in the parent adpack-or for such a

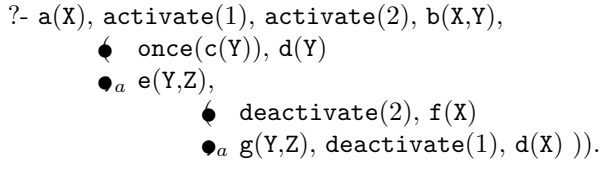
backtrack point. There, a relevant choicepoint is searched for on the parent branch again. Since the `deactivate` of `activate` 1 lies on a branch that is already successful, it is not interesting to reactivate this `activate`. However, `activate` 2 still corresponds to an unsuccessful branch. By backtracking to the choicepoint immediately before `activate` 2, all the closed branches on the path from `activate`(2) to `deactivate`(2) will be reopened during forward execution. Therefore, execution backtracks to the call to `a(X)` (the most recent choicepoint before the most recent relevant `activate`, `activate`(2)):



Now, forward execution restarts, binding `X` to 2, executing `activate`(1) (which does nothing since it corresponds to a successful branch), then arriving in `activate`(2). This `activate` opens all branches from the corresponding `deactivate` up to the current branch. Indeed, this `activate` corresponds to the begin of a once, and since we backtracked to a point before the once, backtracking inside the once is allowed again. After binding `Y` to 1, execution arrives in the top-level adpack-or:



Execution follows the single path of open branches, finally reaching the end of the last unsuccessful branch, which is now marked as successful. Since all child branches of the adpack-or are successful, its parent branch is marked for success also, and execution arrives in the top-level adpack-or. All the latter's children are also successful now, and therefore the top-level branch is marked for success, ending execution of this adpack:



function CONSTRUCT-ADPACK($\mathcal{Q}$) :
    $\mathcal{P} := \emptyset$
    for each $Q \in \mathcal{Q}$
        Q_{once} := ONCE-TRANSFORM(Q)
        Q_{ad} := FLATTEN-ONCES(Q_{once})
        $\mathcal{P}$:= MERGE-IN-ADPACK(Q_{ad}, $\mathcal{P}$)
    return POST-PROCESS($\mathcal{P}$)

Figure 3.3: An algorithm for constructing an adpack, given a set of queries $\mathcal{Q}$.

3.5 Transformation

This section describes how a set of queries generated by an ILP system is transformed into an adpack, before the adpack is evaluated on a set of examples. We consider two approaches: the approach from Section 3.5.1 applies the once transformation on each query separately, and then converts this set into an adpack, whereas the approach from Section 3.5.2 proposes a once transformation that works directly on a query pack of untransformed queries.

3.5.1 Query-based transformation

The query-based transformation works as follows: iterate over the set of given queries, transform each query with the once transformation, and merge the transformed query in an accumulating adpack.

The CONSTRUCT-ADPACK algorithm (Figure 3.3) implements this idea. It takes as input a set of queries Q and starts with an empty adpack P. In each iteration of the main loop, it applies the once transformation to a query Q from $\mathcal{Q}$. The resulting query Q_{once} is then flattened out into Q_{ad}, transforming every `once` into a conjunction delimited by unique `activate`/`deactivate` pairs. Finally, Q_{ad} is added to the accumulator pack $\mathcal{P}$, possibly causing new branching in the accumulator adpack. A postprocessing step makes sure that consecutive `activate`s with consecutive corresponding `deactivate`s are collapsed into one `activate`/`deactivate` pair, and that `activate`s with their corresponding `deactivate`s on the same branch are transformed back into onces. A more detailed description of this transformation can be found in (Struyf 2004; Tronçon, Vandecasteele, Struyf, Demoen, and Janssens 2003).

3.5.2 Pack-based transformation

Because the queries that have to be analyzed by the once transformation have a lot of goals in common, the analysis is also similar. Moreover, ILP systems such as ACE use a pack representation internally to represent the set of queries that are to be evaluated, which needs to be flattened out before it can be used in the query-based transformation from the previous section.

function TRANSFORM-PACK($\mathcal{P}$,$\mathcal{V}$) :
  $\mathcal{Q}_a$ = ANNOTATE-QUERIES($\mathcal{P}$,$\mathcal{V}$)
  $\mathcal{P}_a$ = QUERIES-TO-PACK($\mathcal{Q}_a$)
  $(\overline{\mathcal{P}_a}, \mathcal{S})$:= SPLIT-INDEPENDENT($\mathcal{P}_a$,$\mathcal{V}$)
  $\mathcal{P}_{\text{result}} := \emptyset$
  for each $\mathcal{P}_i$ in $\overline{\mathcal{P}_a}$:
    $(\mathcal{G}_{\text{prefix}}, \mathcal{P}_{\text{tail}})$:= SPLIT-PREFIX($\mathcal{P}_i$)
    $\mathcal{P}_{\text{newtail}}$:= TRANSFORM-PACK($\mathcal{P}_{\text{tail}}$, $\mathcal{V} \cup \mathit{vars}(\mathcal{G}_{\text{prefix}})$)
    $\mathcal{P}_{\text{result}} := \mathcal{P}_{\text{result}} \cup$ ADD-ACTIVATE-DEACTIVATES($\mathcal{G}_{\text{prefix}} \cup \mathcal{P}_{\text{newtail}}$)

Figure 3.4: Algorithm to transform query pack $\mathcal{P}$ into an adpack. The variables $\mathcal{V}$ are not taken into account when computing independence classes.

```
?- p(X), q(Y),
        (   r(Z)
        ;p  s(X)).
```

(a) Original pack $\mathcal{P}$.

```
?- p(X)-C1, q(Y)-C2, r(Z)-C3.
?- p(X)-C1, q(Y)-C1, s(X)-C1.
```

(b) Annotated queries $\mathcal{Q}_a$.

```
?- p(X)-C1,
        (   q(Y)-C2, r(Z)-C3
        ;p  q(Y)-C1, s(X)-C1).
```

(c) Annotated pack $\mathcal{P}_a$.

Figure 3.5: Example adpack, with every independent sub-pack marked.

In this section, we describe a pack-based once transformation, as an alternative for the query-based transformation described in Section 3.5.1. By performing the once transformation on a query pack instead of on each query separately, we avoid redundant parts of the transformation.

A high-level description of the pack-based transformation can be seen in Figure 3.4. Each iteration in the algorithm consists of two phases: in the first phase, the incoming query pack is analyzed to determine the independent parts of the pack (which are packs themselves); the second phase consists of calling the transformation on every independent part recursively, and embedding every part in `activate`/`deactivate` pairs (one `deactivate` for every leaf of the sub-pack).

To partition the pack into independent parts, we annotate each of its goals with a class, where goals of different classes have independent execution. An example of such an annotated pack can be seen in Figure 3.5. All goals in the first query are independent of each other, whereas those of the second are

function ANNOTATE-QUERIES($\mathcal{P}$,$\mathcal{V}$,goals,class,last) :
  for each $goal_i \in conj(\mathcal{P})$:
    goals[i] := $goal_i$
    class[i] := i
    c := class[$min(\{j \mid vars(\text{goals}[j]) \cap vars(goal_i) \backslash \mathcal{V} \neq \emptyset\})$]
    for last[c] $\leq k \leq i$:
      class[k] := c
    last[c] := i
  if $children(\mathcal{P}) = \emptyset$:
    return CREATE-ANNOTATED-QUERY(goals,class)
  else :
    return $\bigcup_{\mathcal{P}_j \in children(\mathcal{P})}$ ANNOTATE-QUERIES($\mathcal{P}_j$,$\mathcal{V}$,g,class,last)

Figure 3.6: ANNOTATE-QUERIES.

assigned to the same class by the analysis. Notice how the structure of the resulting annotated pack differs from the original pack: where the execution of `q(Y)` was shared in the original pack, it is now split up because `q(Y)` belongs to a different independence class in the two queries contained in the pack. By generating a list of annotated queries (in ANNOTATE-QUERIES) and combining them back into a pack, the result is an annotated pack with goals belonging to exactly one independence class.

Contrary to the query-based algorithm, the once transformation is integrated with the adpack transformation algorithm. The ANNOTATE-QUERIES from Figure 3.6 implements the once transformation as defined in Figure 2.9 (page 18), which assumes that every goal grounds its arguments. ANNOTATE-QUERIES also assumes that every goal in the pack has an index i, starting from the root of the pack. ANNOTATE-QUERIES traverses the pack depth first, keeping track of the independence class of each goal of the query it is accumulating. The independence class of each goal is stored in 'class', whereas the current query is accumulated in 'goals'. As class identifier, we use the index of the first goal belonging to the class. For every goal in the accumulated query, the first goal sharing variables with the current goal is determined, and all the goals between the current goal and the first sharing goal are set to the same independent class. Finally, when a leaf of a pack is reached, the accumulated query is annotated with the independence classes, and the process continues for the other queries in the pack.

We now illustrate the pack-based adpack transformation on the following pack:

```
?- a(X), b(X,Y),
        (   c(Y), d(Y)
        ;p e(Y,Z),
                (   f(X)
                ;p g(Y,Z), d(X) )).
```

Applying ANNOTATE-QUERIES on this pack assigns all goals to the same independence class. Therefore, the first step boils down to splitting off the first goal `a(X)`, and continuing with the remainder of the pack, ignoring the now ground variable `X`. Annotating the queries in the second iteration results in the following annotated pack:

```
?- b(X,Y)-C1,
        (   c(Y)-C1, d(Y)-C1
        ;p e(Y,Z)-C1,
                (   f(X)-C3
                ;p g(Y,Z)-C1, d(X)-C4 )).
```

The variables that have to be ignored are striked out. Splitting up this pack into independent sub-packs (SPLIT-INDEPENDENT) yields the following three query packs:

```
?- b(X,Y),
        (   c(Y), d(Y)
        ;p e(Y,Z),                      ?- f(X)          ?- d(X)
                ( ...
                ;p g(Y,Z), ... )).
```

where the '...' denotes an open end, where a transformed (independent) pack will follow in the final pack. Applying the adpack transformation recursively on the two packs on the right yields the original packs. After splitting of the first goal of the pack on the left, the annotation step returns the following annotated pack:

```
?-      (   c(Y)-C1, d(Y)-C2
        ;p e(Y,Z)-C1,
                ( ...
                ;p g(Y,Z)-C1, ... )).
```

which splits up in the following packs:

```
                                        ?- e(Y,Z),
?- c(Y), ...    ?- d(Y).                        ( ...
                                                ;p g(Y,Z), ... )).
```

Applying the adpack transformation recursively on these three packs results in the same packs. At this point, the packs need to be reassembled, adding `activate/deactivate` pairs and onces as necessary. Because the goal `c(Y)` is followed by an independent pack, it needs to be embedded in a `once/1`. The third pack also has open ends, but these open ends were introduced in an earlier step, and are therefore left untouched. This yields the following pack:

```
?-      (   once(c(Y)), d(Y)
        ;a e(Y,Z),
                ( ...
                ;a g(Y,Z), ... )).
```

After prepending `b(X,Y)` to this pack, the open ends need to be replaced by `f(X)` and `d(Y)`. Since the execution of these goals are independent of the current pack, they follow a `deactivate`. We therefore add a `deactivate` for each open end, and add a corresponding `activate` before `b(X,Y)`:

```
?- activate(1), activate(2), b(X,Y),
        (   once(c(Y)), d(Y)
        ;a e(Y,Z),
                ( deactivate(1), ...
                ;a g(Y,Z), deactivate(2), ... )).
```

Appending the goals `f(X)` and `d(X)`, and prepending `a(X)` to the pack results in the final pack:

```
?- a(X), activate(1), activate(2), b(X,Y),
        (   once(c(Y)), d(Y)
        ;a e(Y,Z),
                ( deactivate(1), f(X)
                ;a g(Y,Z), deactivate(2), d(X) )).
```

The Annotate-Queries algorithm described above is order preserving: goals are not reordered to yield more independent parts in the pack. The algorithm can be adapted to reorder goals, at the risk of less sharing of execution in the pack structure.

3.6 Efficient Execution

As mentioned in Section 2.3.2, previous implementation results suggest that special execution mechanisms such as query packs have to be implemented in the internals of the system to yield significant speedups. Therefore, we compile an adpack to specialized WAM instructions, just as is done in the query packs approach. The compilation process is discussed in Section 3.6.1. The actual execution of the newly introduced WAM instructions is covered in 3.6.3, and makes use of the data structures described in 3.6.2.

3.6.1 Compiling

Each adpack that is to be executed is handed to the compiler, which compiles it to WAM instructions. This code makes use of six new WAM instructions:

- `adpack_start` initializes all the data structures

- `adpack_try` performs the forward execution of an adpack-or
- `adpack_retry` handles the backtracking to an adpack-or
- `adpack_activate` and `adpack_deactivate` activate and deactivate parts of the adpack
- `adpack_success` registers the success of a branch of an adpack-or

The implementation of these instructions is described described in 3.6.3. Besides code, the compiler also generates information about the structure of the adpack, and stores this together with the code. When the loader loads the adpack code, it creates and initializes the adpack datastructures based on this information. We omit the details on how the adpack structure information is stored in practice.

Compiling the example from Figure 3.2 results in the following WAM code:

```
 1 adpack_start 2 4 3 0x800121    17 adpack_success
 2 allocate 5                     18 putpval Y3 A1
 3 putpvar Y2 A1                  19 putpval Y4 A2
 4 call a/1                       20 call e/2
 5 adpack_activate 0              21 adpack_try 2
 6 adpack_activate 1              22 adpack_deactivate 1
 7 putpval Y2 A1                  23 putpval Y2 A1
 8 putpvar Y3 A2                  24 call f/1
 9 call b/2                       25 adpack_success
10 adpack_activate 2              26 putpval Y3 A1
11 adpack_try 1                   27 putpval Y4 A2
12 putpval Y3 A1                  28 call g/2
13 call c/1                       29 adpack_deactivate 0
14 adpack_deactivate 2            30 putpval Y2 A1
15 putpval Y3 A1                  31 call d/1
16 call d/1                       32 adpack_success
```

Not all information about the execution of the adpack is present in the code of the adpack. The `adpack_try` instruction looks up information about the branches belonging to the corresponding adpack-or in a datastructure, and uses this to determine the location in the code where execution needs to proceed (i.e. the code of the first branch of the adpack-or). Before jumping to this location, it creates a choicepoint with `adpack_retry` as alternative. This instruction is not stored in the code of the adpack itself, as the operations it needs to perform are independent of its location in the code: while execution in the WAM always proceeds to the next instruction after a `retry*` instruction, the `adpack_retry` uses the adpack datastructures to determine where execution has to proceed. This is explained in more detail in Section 3.6.3.

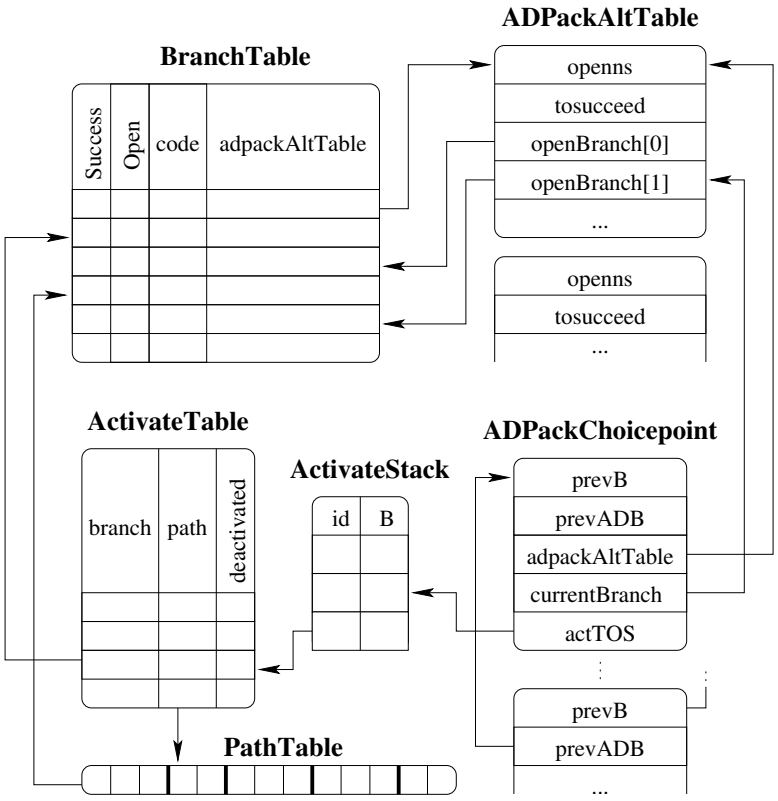


Figure 3.7: Overview of the data structures in the implementation of adpack execution.

3.6.2 Data structures

To execute an adpack, we make use of the data structures depicted in Figure 3.7. These data structures are designed for executing adpacks as efficiently as possible. The description of these data structures is as follows:

- **BranchTable:** This table contains for each branch in the adpack the following information: a flag indicating whether this branch was completed successfully (*success*, initially false); a flag indicating whether the branch can be entered for execution. (*open*, initially true); the location of the code block for this branch (*code*); and finally a reference to the alternatives table of the adpack-or to which this branch belongs (*adpackAltTable*).
- **ADPackAltTable:** Each adpack-or has a corresponding ADPackAltTable. This table contains the number of branches that are still unsuccessful (*tosucceed*), the number of open, non-successful branches (*openns*), and for each of those open, non-successful branches an entry in *openBranch* containing its index in the BranchTable.
- **ActivateTable:** This table contains for each `activate`/`deactivate` pair a flag indicating whether the `deactivate` has been triggered (*deactivated*), the branch on which the `deactivate` resides (*branch*), and a reference to a path in the PathTable (*path*). The row on which a `deactivate` is located in the ActivateTable is the unique identification number of the `activate`/`deactivate` pair (as described in the definition in Section 3.3).
- **PathTable:** This table contains for each `activate`/`deactivate` pair a sequence of branches describing the path from the `activate` to the `deactivate`. Each path is referenced from an entry in the ActivateTable.
- **ActivateStack:** Each time an `activate` is performed, the unique identification number of the `activate`/`deactivate` pair to which this activate belongs (*id*) is pushed on the ActivateStack, together with a reference to the most recent choicepoint at the time the `activate` is triggered (B).
- **ADPackChoicepoint:** When an adpack-or is executed, an ADPackChoicepoint is created, containing a reference to the previous choicepoint (*prevB*), a reference to the previous ADPackChoicepoint (*prevADB*), a reference to the ADPackAltTable of this adpack-or (*adpackAltTable*), the index of the currently executed branch in the openBranch table of the corresponding ADPackAltTable (*currentBranch*, initially 0), and the top of the ActivateStack at the time the execution of the adpack-or started (*actTOS*). The pointer to the code to be executed upon backtracking to this choicepoint always points to the `adpack_retry` instruction.

We impose an extra constraint on the unique identification number associated with each `activate`/`deactivate` pair: for each path from the root to a leaf of the adpack, the identification numbers of the `activate`s on that path should increase. Numbering the `activate`/`deactivate` pairs by depth-first traversal satisfies this condition.

3.6.3 Executing the ADPack Instructions

In this section, we present the details of every WAM instruction.

3.6.3.1 `adpack_start <#adpack-or> <#branches> <#activates> <data>`

`adpack_start` indicates that an adpack is executed. This instruction has to reset all data structures, using information on the total number of disjunctions (`<#adpack-or>`), the total number of branches in the adpack (`<#branches>`), the total number of `activate`/`deactivate` pairs (`<#activates>`), and the location of all data structures (`<data>`). The instruction should ensure that:

- All AltTables have all their children in `openBranch[]`, and that `openns` and `tosucceed` are initialized to the total number of children. The necessary information to do this can be derived from the BranchTable.
- The `open` flag of all branches in the BranchTable is set to true, and the `success` flag to false.
- The ActivateStack is emptied.
- The `deactivated` flag of all `activate`/`deactivate` pairs in the ActivateTable is set to false.
- A dummy ADPackChoicepoint is put on the choicepoint stack.

The implementation of this instruction is omitted, as it can be easily reconstructed.

3.6.3.2 `adpack_activate <id>`

This instruction activates the `activate`/`deactivate` pair `<id>`, opens the path from the `activate` to the `deactivate`, and pushes a record on the ActivateStack.

```
if (!BranchTable[*ActivateTable[id].path]->success) {
   if (ActivateTable[id].deactivated) {
      ActivateTable[id].deactivated = false;

      /* Open path */
      if (!(br=&BranchTable[ActivateTable[id].branch])->open) {
         /* Open first branch */
         br->open = true;
         br->adpackAltTable->openBranch[br->adpackAltTable->
                openns++] = ActivateTable[id].branch

         /* Open rest of path */
         path = ActivateTable[id].path;
         while (*path && !(br=&BranchTable[*path])->open) {
```

```
            br->open = true;
            br->adpackAltTable->openBranch[
                 br->adpackAltTable->openns++] = *path;
            path++;
         }
      }
   }

   /* Pop choicepoints of more recent activates */
   while ((ActivateStack-1)->id >= id)
      ActivateStack--;

   /* Push most recent choicepoint */
   ActivateStack->id = id;
   ActivateStack->B = B;
   ActivateStack++;
}
```

3.6.3.3 `adpack_deactivate <id>`

This instruction registers deactivation of an `activate`/`deactivate` pair, and closes the current branch. The only argument of `adpack_deactivate` is the unique number of the `activate`/`deactivate` pair.

```
/* Register deactivation */
BranchTable[ADB->adpackAltTable->
  openBranches[ADB->currentBranch]].open = false;
ActivateTable[id].deactivated = true;

/* Cut up to previous ADPackChoicepoint */
B = ADB;
```

Notice that `ADB` is a register containing a reference to the previous ADPack-Choicepoint.

3.6.3.4 `adpack_success`

This instruction denotes the end of a branch. It registers success of the current branch, and backtracks.

```
/* Register success & decrement #tosucceed */
BranchTable[ADB->adpackAltTable->
  openBranches[ADB->currentBranch]].success = true;
ADB->adpackAltTable->tosucceed--;

/* Cut up to previous ADPackChoicepoint */
B = ADB;
```

```
/* Backtrack */
fail;
```

3.6.3.5 adpack_try <id>

The `adpack_try` instruction corresponds to the part from Section 3.3 describing the forward execution of an adpack-or. This instruction puts an ADPackChoicepoint on the choicepoint-stack, using the first argument of the instruction to determine the corresponding `adpackAltTable` of the adpack. The `currentBranch` field is initialized to 0, the `actTOS` field is set to the current top of the ActivateStack, and the address of the `adpack_retry` instruction is used as alternative for the choicepoint. Finally, the program counter is set to the code for the first branch.

```
/* If there are no open nodes, backtrack */
if (adpackAltTable->openns =< 0)
    fail;

/* Make new choicepoint */
prevB = B
B++;
B->prevB = prevB;
B->prevADB = ADB;
B->adpackAltTable = (address of AdpackAltTable #id);
B->currentBranch = 0;
B->actTOS = /* top of */ ActivateStack;
B->alt = @adpack_retry;
ADB = B;

/* Set program counter */
PC = BranchTable[ADB->adpackAltTable->openBranch[0]].code;
```

3.6.3.6 adpack_retry

This instruction handles backtracking to an adpack-or. Using the information stored in the current ADPackChoicepoint, it is determined which action to take. The 4 cases correspond to the ones in Section 3.3.

```
tbl = ADB->adpackAltTable;

/* Temporarily remove branch & set current branch */
if (BranchTable[tbl->openBranches[ADB->currentBranch]].success
  || !BranchTable[tbl->openBranches[ADB->currentBranch]].open) {
    tbl->openns--;
    tbl->openBranches[ADB->currentBranch] =
        tbl->openBranches[tbl->openns]
```

```
}
else {
    ADB->currentBranch++;
}

/* Reset stack */
/* top of */ ActivateStack = ADB->actTOS;
if (ADB->currentBranch < tbl->openns) {        /* Case 1 */
  /* Try another branch */
  PC = BranchTable[tbl->openBranches[tbl->currentBranch]].code;
}
else {
  ADB = ADB->prevADB;

  if (ADB->adpackAltTable->tosucceed == 0) {   /* Case 2 */
    /* Mark branch as successful */
    goto adpack_success;
  }
  else if (ADB->adpackAltTable->openns == 0) { /* Case 3 */
      /* Pop until a non-successful deactivated branch */
      while (ActivateStack > ADB->actTOS) {
          ActivateStack--; // Pop element
          if (ActivateTable[ActivateStack->id].deactivated
              && ! BranchTable[ADB->adpackAltTable->openBranch[
                          ADB->currentBranch]].success) {
              /* Cut & backtrack */
              B = ActivateStack->B;
              fail;
          }
      }

      /* Close branch */
      BranchTable[ADB->adpackAltTable->openBranches[
          ADB->currentBranch]].open = false;
      B = ADB;
      fail;
  }
  else {                                       /* Case 4 */
    B = B->prevB;
    fail;
  }
}
```

3.7 Optimizations

Experiments on ILP applications indicate that `activate`s often come in batches. Consider the following part of an adpack taken from adpack execution for a real-life application:

$$\ldots, \texttt{p(X)}, \texttt{activate}(1), \texttt{activate}(2), \ldots, \texttt{activate}(100), \texttt{q(X,Y)}, \ldots$$

After `p(X)` has succeeded for the first time, 100 `activate`s are executed, and `q(X,Y)` is called. If the latter fails, another alternative for `p(X)` is tried, and the first `activate` pops all the later `activate`s one by one from the stack. We have observed that such a scenario happens frequently in practice. The occurrences of these batches of `activate`s is inefficient both in time (popping the stack one by one, plus a different emulator cycle for each `activate`) and space (all `activate` records on the stack have the same choicepoint field).

To avoid this inefficiency, we add a new instruction `adpack_activate_range`, which has as its two arguments the begin and the end of the `activate` id range. This of course imposes the extra constraint on the compiler to have a consecutive numbering on the `activate`s.

Additionally, we store variable-length records on the ActivateStack, where each record has a backtrack point, a pointer to the previous record, and a variable list of `activate` id's which belong to this record. Popping the stack in `adpack_activate` and `adpack_activate_range` can now be simplified to checking the last element of each record, and popping the whole record if its id is more recent than the id of the current `activate`. Pushing elements on this stack can be done in two ways:

- Each time an `adpack_activate` instruction puts something on the stack, it checks if the choicepoint of the top element is the same as the current choicepoint. If so, the `activate` instruction adds the new `activate` (or in the case of `adpack_activate_range`, batch of `activate`s) to the current record. Otherwise, a new record is created and pushed on the ActivateStack.

- `adpack_activate` always creates a new record on the `ActivateStack` containing one `activate`, and `adpack_activate_range` stores its batch of `activate`s in one record. In this case, we assume that the compiler statically determines which `activate` instructions can be grouped together. This means that there is no overhead due to extending records on the stack, and all the dynamic checks are omitted. The downside of this approach is that there could be non-consecutive groups of `activate`s that can be grouped together on the stack (because they share the same most recent choicepoint at run-time), but that this cannot be detected statically.

3.8 Evaluation

We evaluate the adpack execution mechanism through various experiments. All experiments were run on a Pentium III 1.1 GHz with 2Gb of RAM, running Linux.

Both the query-based transformation and the pack-based transformation were implemented in the ACE system. Both transformations were implemented in Prolog. However, the ONCE-TRANSFORM step from the query-based transformation (Figure 3.3, page 32) makes use of the built-in once transformation from ACE, which is written in C. The adpack compiler itself was implemented in Prolog as an extension of the hipP compiler. We have implemented the additional WAM instructions and required data structures in the hipP engine.

As a first experiment, we compare both adpack transformations by running TILDE on the Mutagenesis dataset (Appendix A.1), Carcinogenesis (Appendix A.2), and a version of Bongard (Appendix A.3) with 10000 examples. After generating a set of queries to be evaluated, TILDE transforms the set into an adpack, compiles the adpack, and executes it on the examples. In this experiment, we only measure the time needed to transform the queries. The lookahead setting of TILDE was set to vary between 0 and 2 to compare the transformation of queries of increasing complexity. The results of this experiment can be seen in Figure 3.8. Because the pack-based transformation takes advantage of the similarity of queries, we expected this transformation to be faster than the query-based transformation. However, contrary to this intuition, the query-based approach performs better over the whole line of experiments, even with increasing query sizes. Closer investigation reveals that the major bottleneck of the pack-based transformation is the QUERIES-TO-PACK step from Figure 3.4. Transforming the flattened annotated version of every pack (and sub-pack) into a pack takes 70% of the total pack-based adpack transformation time. This means that the fact that the once transformation step of the query based approach is implemented in C is not the reason the pack-based approach is slower than the query based approach.

To compare the overall performance of adpacks, we compare runs of TILDE using regular query execution ('Query'), using once transformed query execution ('Once'), using query pack execution ('Pack'), and finally using adpack execution ('ADPack'). For adpack execution, we use the query based transformation, as this is the fastest alternative. Figure 3.9 shows the results of running TILDE with different lookahead settings on the same datasets as the previous experiment. For each setting, the graph shows the query execution time 'Execute', the compilation time 'Compile' (only relevant for query packs and adpacks, the other settings use meta-calls) and the transformation time 'Transform' (only relevant for once transformed queries and adpacks). All bars are relative to the total time for 'Queries', which is shown together with the number of refinement steps and the total number of queries evaluated. Above the other bars, the speedup in execution time over 'Queries' is shown. The experiments are

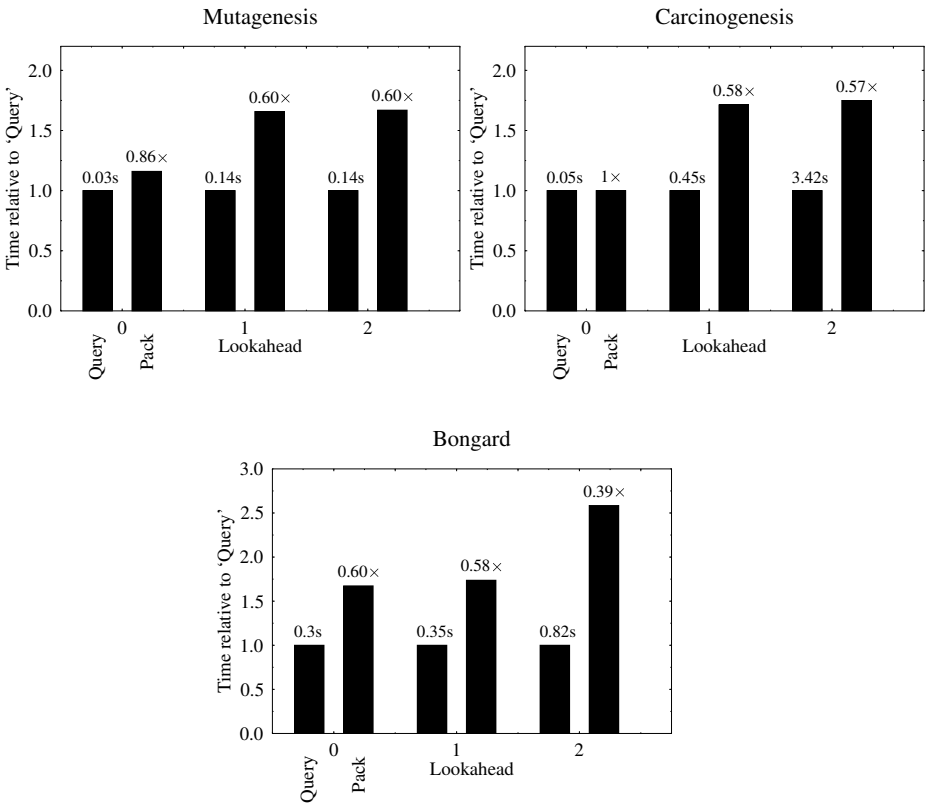


Figure 3.8: Query-based and pack-based transformation times for running TILDE on Mutagenesis, Carcinogenesis and Bongard.

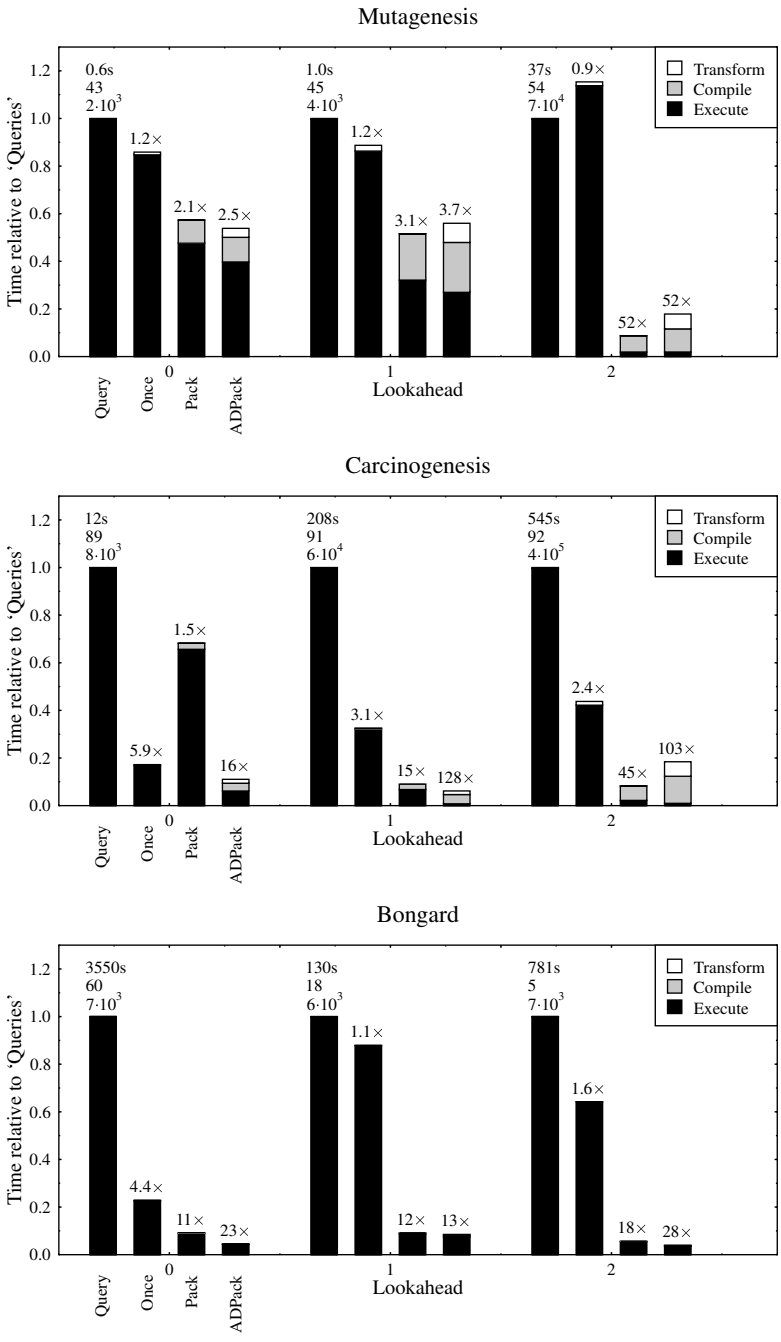


Figure 3.9: Experiments on the data sets Mutagenesis, Carcinogenesis and Bongard.

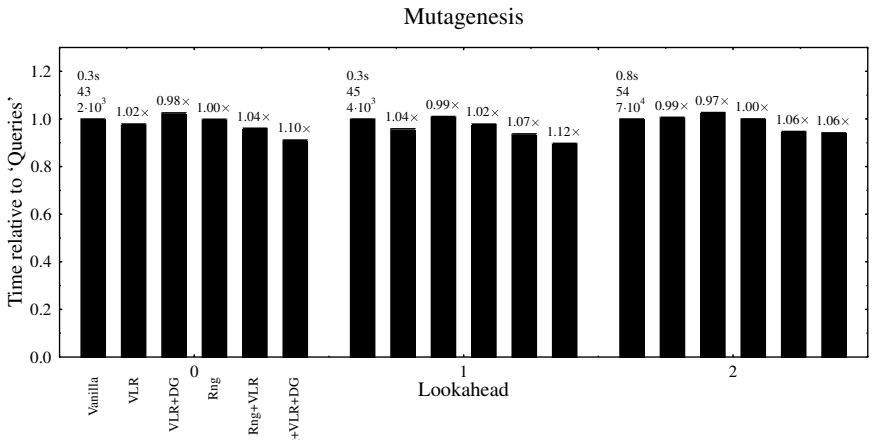


Figure 3.10: Effect of various optimizations on the execution time.

performed for varying values of TILDE's lookahead parameter.

In all experiments, query pack execution performs better than queries and the speedup increases as lookahead increases. Similar results were obtained in (Blockeel, Dehaspe, Demoen, Janssens, Ramon, and Vandecasteele 2002). Applying the once transformation on queries yields a significant improvement in execution time for the Carcinogenesis and Bongard data sets (without lookahead), but the improvement is smaller than that reported in (Costa, Srinivasan, Camacho, Blockeel, Demoen, Janssens, Struyf, Vandecasteele, and Van Laer 2002) ($> 100\times$ for Carcinogenesis). This is due to the fact that the search strategy of TILDE generates fewer non-deterministic queries than Aleph (Srinivasan 2005), the algorithm used in the experiments from the earlier work. Also note that the performance of the once transformed queries is worse for higher lookahead values. The reason is that with higher lookahead, larger components are generated that cannot be partitioned by the once transformation (Struyf 2004).

The execution time of adpacks is better than all other settings, but is close to query packs in most cases (especially when the once transformation does not perform well). The best improvement over query packs is obtained on Carcinogenesis ($\pm 10\times$). For some applications (e.g., Mutagenesis with lookahead ≥ 1), the higher transformation[2] and compilation times of the adpack version can make regular query pack execution the best choice (best total time). However, the proportion of time spent during transformation and compilation decreases as the number of examples increases as can be seen with the Bongard data set with 10000 examples.

[2]The transformation time for adpacks is higher than that of the once transformed queries because the queries must also be merged in the adpack (cfr. Figure 3.3).

As a final experiment, we compare the impact of the optimizations introduced in Section 3.7. Figure 3.10 shows the execution time for 6 different versions of the ADPack execution mechanism on the Mutagenesis dataset, where each of the versions is formed by using a combination of following changes (each described in more detail in 3.7):

- **Variable-length records (VLR):** Store variable-length records on the ActivateStack. This not a real optimization, just a change of data structure. This change is needed for the following optimizations.
- **Dynamic Grouping (DG):** Accumulate `activate`s with the same corresponding choicepoint in the top record of the ActivateStack. Only start a new record if the choicepoint of the `activate` (or group of `activate`s) is different than the one of the record on top of the ActivateStack.
- **Use `activate_range` (Rng):** let the compiler statically group subsequent `activate`s together, and generate `activate_range` instructions for these groups.

As can be seen in Figure 3.10, variable-length records are a slightly more efficient representation for the ActivateStack. The extra checks needed for dynamic grouping are not compensated for in execution time in absence of the grouping of `adpack_range`. This is due to the fact that, when `activate`s are statically grouped together, the check only has to be performed once for each group of `activate`s, instead of for each `activate` separately. Introducing `activate_range` does not improve execution time in itself, but combined with dynamic grouping it results in the best execution time. Overall, however, the difference in execution time of the variations are small.

3.9 Conclusions

In this chapter, we introduced adpacks as an approach to combine query pack execution with the once transformation. The experiments in Section 3.8 show that the evaluation of queries benefits from adpack execution, compared to using only query packs or the once transformation. However, the overhead caused by transforming a set of queries into an adpack causes query packs to outperform adpacks in some experiments. Applying the once transformation on a pack as a whole (instead of each query separately) does not help, because this transformation relies heavily on the construction of query packs. A possible remedy for this is to modify the pack structure directly (instead of creating new packs during the transformation phase). While we expect most of the high-level transformation algorithm and data structures to be reusable for this, altering a query pack directly at the engine level requires a significant effort. Another open issue is the impact of reordering goal literals in the adpack transformation to yield more independent parts in the adpack, how this affects the structure of the adpack, and whether or not execution benefits from this. However, note that

the potential benefit (if any) can not compensate for the high transformation times at this point.

Avoiding redundant backtracking such as is done by the once transformation is also the goal of techniques such as *backjumping* (Gaschnig 1979) and *intelligent backtracking* (Bruynooghe and Pereira 1984). These approaches use run-time checks to determine what parts of the search tree can be pruned. The run-time overhead of these optimizations make them unsuitable for the ILP context. However, especially the non-reordering once transformation still suffers from redundant backtracking sometimes, and a static version of more advanced backtracking techniques might therefore be interesting. How this is implemented, and how this can be combined with query pack execution is a topic for further research.

Chapter 4

Alternatives for Compile-and-Run

4.1 Introduction

When a set of queries have been constructed dynamically by an ILP algorithm, they need to be evaluated on a series of examples from a dataset. At least two approaches of doing this come to mind: either the queries are meta-called directly for every example, or the queries are first compiled to WAM instructions, after which the resulting code is executed. *Meta-calling* consists of passing the query to `call/1`, which looks up the code entry point of the top-level functor of its argument, initializes the argument registers to the values of the arguments of the functor, and jumps to the entry point of the corresponding predicate. Retrieving the top-level functor and its arguments for every call introduces overhead during execution. This overhead becomes even bigger when a conjunction is meta-called, since this requires for each conjunct that both arguments of the `','/2` functor have to be retrieved, and that both arguments have to be in turn meta-called. The alternative to meta-calling a query is transforming it into a clause (by using the query as the body and adding a head, e.g. `query/0`), and *compiling* the resulting clause. By compiling this clause, the predicates that have to be called are computed in advance, and are directly embedded in the compiled code. Moreover, conjunctions and disjunctions in queries are encoded in the code itself: conjunctions are simply sequences of instructions, while disjunctions are executed using choice instructions (`try*`, `retry*`, `trust`*). Although precomputing all this information introduces an initial overhead, it is compensated by the large reduction in dynamic overhead when the code is run multiple times.

Because queries need to be evaluated many times, the compilation approach has been identified to be the best in the context of ILP in the past (Blockeel, Dehaspe, Demoen, Janssens, Ramon, and Vandecasteele 2002). Although compilation requires an initial cost, the repeated execution of the more efficient

code compensates for this. Moreover, in the case of special execution mechanisms such as query packs (Section 2.3.2) and adpacks (Chapter 3), adapting the meta-call to handle these query packs is difficult and inefficient, and therefore compilation is needed (Blockeel, Dehaspe, Demoen, Janssens, Ramon, and Vandecasteele 2002; Demoen, Janssens, and Vandecasteele 1999). However, experiments indicate that, even though compilation improves the total evaluation time of queries (i.e. compilation and execution time), the compilation time often dominates the total time of an ILP run. For example, in the Mutagenesis and Carcinogenesis experiments depicted in Figure 3.9 (page 47), compilation consumes the largest part of the query evaluation step for the highest lookahead settings. This raises the question whether the amount of code that needs to be compiled can be reduced, whether compilation can be simplified, or even avoided altogether.

In this chapter, we study alternatives for the compile-and-run approach for query evaluation. While meta-calling introduces a dynamic overhead, its dynamic nature also has advantages over compiled queries. Amongst others, meta-calling does not require the complex step of constructing the arguments of a goal before calling it. A first direction we therefore take is fine-tuning the meta-call and exploiting its advantages, trying to reach the same execution speed as compiled code (yet omitting the compilation step). The second approach consists of simplifying the compilation step, while still keeping the advantages of meta-call in the generated code. Finally, the amount of code to be compiled is reduced by only compiling code when it is strictly necessary.

The contributions of this chapter are:

- A *specialized meta-call* for query execution, implemented as a series of special WAM instructions. This embedded meta-call provides a fast query execution mechanism without requiring a compilation step.

- *Control flow compilation*, which is a hybrid approach between meta-calling and classical compilation. This scheme incorporates the best of both worlds: it has the fast execution times of compiled code, without needing the expensive compilation step (which is a dominating factor in practical ILP settings). Another advantage is that it is easy to support query packs in this approach.

- *Lazy control flow compilation*, which is a Just-In-Time (JIT) version of the control flow compilation scheme, where unreachable parts of the code are not compiled. This reduces both the compilation time and the code size.

An overview of these approaches in terms of execution and compilation is shown in Figure 4.1.

The organization of this chapter is as follows: In Section 4.2, we investigate and optimize meta-calling of conjunctive and disjunctive queries. In Section 4.3,

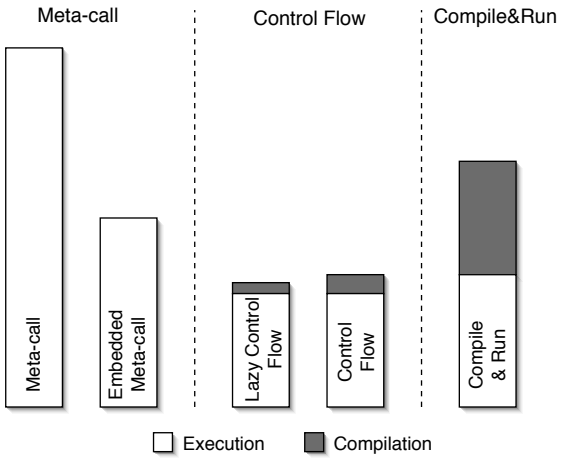


Figure 4.1: Overview of the query execution mechanisms presented in this chapter.

control flow compilation is introduced and evaluated on both artificial and real life benchmarks. A lazy variant of this scheme is introduced in Section 4.4. Both control flow compilation variants are then adapted to a practical ILP setting, by extending them to the query pack execution mechanism in Section 4.5. This extension is evaluated on real life ILP benchmarks. Section 4.6 discusses memory management implementation issues of the approaches described in this chapter. Finally, we present our conclusions of the described approaches in Section 4.7, and discuss future work.

4.2 Meta-calling

4.2.1 Specializing the meta-call

Figure 4.2 shows the implementation of `call/1` as a built-in instruction `metacall`. This instruction fetches the goal to be called from the first argument register, initializes the argument registers with all arguments of the goal, and then jumps to the entry point of the code for the top-level functor. When a conjunction is meta-called, a lot of steps have to be taken: for every conjunction of two goals, both conjuncts are put in argument registers, and the code for `','/2` (Figure 4.3) is executed, which in turn uses meta-call to execute the goals in both argument registers.

In a first attempt to avoid the overhead of meta-calling conjunctions, we introduce a more specialized approach of meta-calling conjunctive queries. Consider the predicate `conj_call` from Figure 4.4. This special version of `call/1`

instruction METACALL :
    $Goal$:= $\texttt{ARG}_1$
    for $1 \leq i \leq$ arity(functor($Goal$)) :
        $\texttt{ARG}_i$:= arg($Goal$, i)
    CONTP := PC + 1
    PC := entry_point(functor($Goal$))
    continue

Figure 4.2: Pseudo-code for the `metacall` instruction. PC, CONTP, and $\texttt{ARG}_i$ represent the program counter, continuation pointer, and argument registers respectively.

```
allocate 3
getpvar Y2 A2
call call/1
putpval Y2 A1
deallex call/1
```

Figure 4.3: Simplified WAM compiled version of `','/2`. Support for cut is omitted.

```
conj_call((A,B)) :- !,
    call(A),
    conj_call(B).
conj_call(G) :-
    call(G).
```

Figure 4.4: `conj_call/1`: A specialized `call/1` for conjunctive queries.

Query
?- b, f([g(1), g(2), ...,g(60)]).

	Program
1	b.
2	b.
	⋮
20	b.
21	f(_) :- fail.

Compiled Query
allocate 2
call b/0
putlist A1
set_struct 2
build_list 3
set_functor g/1
build_int 1
⋮
set_struct 2
build_list 3
set_functor g/1
build_int 60
deallex f/1

Figure 4.5: Example program and query.

Experiment	Time
Compiled code	94.6
Meta-call (`call/1`)	10.2
Specialized meta-call (`conj_call/1`)	8.1

Table 4.1: Timings (in μs) for executing the query from Figure 4.5 using different approaches.

takes advantage of the fact that conjunctions passed to `conj_call` are always right-linear. This saves an extra indirection when calling the first part of the conjunction.

Now consider the program and the query from Figure 4.5. The query consists of a call to a highly non-deterministic predicate `b` (with 20 clauses), and a (failing) call with a very large argument. Executing this query on the example program using the three approaches discussed so far results in the timings from Table 4.1. We see that the meta-call is faster than the compiled call, and that the specialized `conj_call/2` is even faster. The explanation is that the compiled version of a query first constructs the arguments of its calls on the heap before calling (as can be seen in the WAM code from Figure 4.5), whereas meta-calling a goal uses the term which has already been constructed on the heap. In this case, the (very large) term that is passed as an argument to `f/1` is constructed over and over again in the compiled version, whereas it is never constructed in

```
      gettbreg A2
      switchonterm struct=L1 else=L2   △
L1:   get_structure A1 ','/2           △
      unitvar A1                       △
      allocate 3                       △
      unipvar Y2                       △
      puttbreg A2
      call call/1                      △
      putpval Y2 A1                    ◯
      deallex conj_call/1              ◯
L2:   execute call/1                   △
```

Figure 4.6: Compiled WAM code of `conj_call/1`. The △ and ◯ symbols indicate the instructions that can be merged into one instruction.

the meta-called versions (as it was already preconstructed when the query was generated).

One could think that the situation can be easily improved for the compiled version, by compiling it as if its code where:

$$\text{?- } \texttt{X} = \texttt{[g(1), g(2), \ldots ,g(60)],b, f(X)}.$$

However, such a transformation, although correct, can make performance also worse: if `b` fails, the term has been constructed in vain. Moreover, the memory requirement can become arbitrarily larger if this transformation is performed systematically in a Prolog program.

From these results, we can conclude that compiling a query before executing it is not a priori the fastest alternative, even without taking into account the overhead of compilation. However, executing a query using the normal meta-call or `conj_call/1` is in practical situations still slower than executing compiled code. We improve upon this in the following section.

4.2.2 Embedding the meta-call

To get more speedup from a specialized meta-call such as `conj_call/1`, it should be implemented in the internals of the Prolog system. One could choose to implement the specialized call completely in the host language of the system. However, it is easier to implement a series of new WAM instructions, and to use those to implement `conj_call`. Also, this approach results in the same performance as implementing the specialized call directly in the host language of the system.

The WAM compiled version of the `conj_call/1` predicate from Figure 4.4 can be seen in Figure 4.6. We can now make a new instruction `mc_switch` that performs all the actions from the instructions labeled with △, and another new

```
L1   mc_switch L2          △
L2   mc_continueconj L1    ◯
```

Figure 4.7: Compiled WAM code of `conj_call/1`, using specialized instructions.

```
conjdisj_call((A,B)) :- !,
    call(A),
    conjdisj_call(B).
conjdisj_call((A;B)) :- !,
    ( conjdisj_call(A) ; conjdisj_call(B) ).
conjdisj_call(A) :-
    call(A).
```

Figure 4.8: `conjdisj_call/1`: A specialized `call/1` for queries consisting of conjunctions and disjunctions.

instruction `mc_continueconj` that performs the actions of the instructions labeled with ◯. The remaining (cut-related) instructions are no longer of any use, and are therefore dropped. Using these instructions, we can make a new version of `conj_call`, as shown in Figure 4.7. The instruction `mc_switch` distinguishes 2 types of terms in the first argument register: a `','/2` term and a goal. In the latter case, the goal is simply called. When the argument is a conjunction, an environment is allocated, and the first argument of `','/2` is called with the label of `mc_continueconj` (passed through the argument of `mc_switch`) as its continuation. When execution reaches `mc_continueconj`, the environment created by `mc_switch` is deallocated, the continuation pointer is restored, and `mc_switch` is executed with the second argument of `','/2`. Pseudo-code of the `mc_switch` and `mc_continueconj` instructions is presented at the end of this section.

We now extend `conj_call/2` to a predicate that can handle disjunctions as well. The resulting predicate is shown in Figure 4.8. Notice that, since the disjunctions might in turn contain conjunctions in both arguments, we cannot assume that the first argument of `';'/2` is a simple goal as we could with `','/2`.

To implement `conjdisj_call/1` in the WAM, the `mc_switch` instruction has to be extended, and an extra instruction `mc_continuedisj` is introduced. The resulting code can be seen in Figure 4.9. The only extension `mc_switch` needs is the ability to handle a third type of term in the first argument register: a `';'/2` term. In this case, a choice-point is created with the label of `mc_continuedisj` (passed through the second argument of `mc_switch`) as an alternative, after which `mc_switch` is executed on the first argument of `';'/2`. In `mc_continuedisj`, the choice-point is removed, and `mc_switch` is executed with the second argument of `';'/2`.

For subsequent goals in a conjunction, the embedded meta-call deallocates

```
L1   mc_switch L2 L3
L2   mc_continueconj L1
L3   mc_continuedisj L1
```

Figure 4.9: Compiled WAM code of `conjdisj_call/1`, using specialized instructions.

Query				Experiment	Time[a]
G[b]	B[c]	D[d]	T[e]		
5	5	4	3905	Meta-call	2.14
				Compile & Run	0.38
				Embedded Meta-call	0.36
10	10	4	111110	Meta-call	64.94
				Compile & Run	10.82
				Embedded Meta-call	13.35
5	5	6	19531	Meta-call	59.20
				Compile & Run	9.93
				Embedded Meta-call	14.50

[a]Execution time of the query (in ms.)
[b]Number of goals in a branch
[c]Branching factor of each disjunction
[d]Nesting depth of disjunctions
[e]Total number of goals ($= G \sum_{n=0}^{n=D} B^n$)

Table 4.2: Experiments for artificial disjunctions.

the current environment (in `mc_continueconj`), and immediately allocates a new environment (in `mc_switch`). Such redundancy can be avoided by adding an extra test: `mc_continueconj` checks the next functor, and skips the deallocate and allocate if it is again a conjunction. The pseudo-code for the three new instructions is shown in Figure 4.10.

4.2.3 Evaluation

To compare the embedded meta-call with the normal meta-call and the compile-and-run approach, we perform a set of artificial experiments. For each artificial experiment, a query was generated with the following parameters:

- G: the number of goals in a branch,
- B: the branching factor in a disjunction,
- D: the nesting depth of disjunctions.

For example, for the values $G = 2$, $B = 3$ and $D = 1$, the following query is generated:

```
instruction MC_SWITCH (ConjLabel,DisjLabel) :
    Goal := ARG_1
    if functor(Goal) = ','/2 :
        allocate_environment(1)
        E[0] := arg(Goal, 2)
        ConjGoal := arg(Goal, 1)
        for 1 ≤ i ≤ arity(functor(ConjGoal)) :
            ARG_i := arg(ConjGoal, i)
        CONTP := ConjLabel
        PC := entry_point(functor(ConjGoal))
    else if functor(Goal) = ';'/2 :
        ARG_1 := arg(Goal, 2)
        create_choicepoint(DisjLabel)
        ARG_1 := arg(Goal, 1)
        PC := PC
    else :
        for 1 ≤ i ≤ arity(functor(Goal)) :
            ARG_i := arg(Goal, i)
        PC := entry_point(functor(Goal))
    continue

instruction MC_CONTINUECONJ (SwitchLabel) :
    Goal := ARG_1
    if functor(Goal) = ','/2 :
        E[0] := arg(Goal, 2)
        ConjGoal := arg(Goal, 1)
        for 1 ≤ i ≤ arity(functor(ConjGoal)) :
            ARG_i := arg(ConjGoal, i)
        CONTP := PC
        PC := entry_point(functor(ConjGoal))
    else :
        ARG_1 := E[0]
        deallocate_environment()
        PC := SwitchLabel
    continue

instruction MC_CONTINUEDISJ (SwitchLabel) :
    pop_choicepoint()
    PC := SwitchLabel
    continue
```

Figure 4.10: Pseudo-code for the `mc_switch`, `mc_continueconj` and `mc_continuedisj` instructions. `PC`, `CONTP`, `E` and $\texttt{ARG}_i$ represent the program counter, continuation pointer, environment pointer, and argument registers respectively.

```
?- a(A,B,C), a(C,D,E), ( a(E,F,G), a(G,H,I)
                       ; a(E,J,K), a(K,L,M)
                       ; a(E,N,O), a(O,P,Q) ).
```

For $G = 1$, $B = 2$ and $D = 2$, the generated query has nested disjunctions:

```
?- a(A,B,C), ( a(C,D,E), ( a(E,F,G) ; a(E,H,I) )
             ; a(C,J,K), ( a(K,L,M) ; a(K,N,O) ) ).
```

The definition of `a/3` was taken to be `a(_,_,_)` to minimize the time spent outside of the query execution. For each generated query, the average spent on executing the query was measured over a large number of runs. All experiments (including all other experiments from this chapter) were run on a Pentium III 1.1 GHz with 2 GB main memory running Linux, with a minimum of applications running. The resulting timings can be seen in Table 4.2. As the results show, the embedded metacall indeed results in a significant speedup over the original meta-call (up to 6 times faster). However, except for the first experiment, classical compiled code is still faster than the embedded meta-call. Therefore, if a query is executed multiple times (i.e. for different examples), compile-and-run can outperform execution that uses the embedded meta-call.

4.2.4 Conclusion

We illustrated that meta-call has advantages over compiled code: it does not require a (costly) compilation step, nor does it need to spend as much time in setting up arguments to goal calls. The dynamic overhead introduced by the meta-call can be significantly reduced by implementing a specialized version directly in the internals of the system, yielding up to 6 times faster execution. However, besides the fact that it cannot always compete with compiled code speed-wise, implementation of an embedded meta-call suffers from other drawbacks as well. Compiled code can make use of built-in instructions for optimizing execution of arithmetic, tests, etc. Extending the embedded meta-call to perform built-in operations requires adding more checks in the `mc_switch` instruction, which is cumbersome and slows down the execution, regardless of whether a query contains built-ins or not. Another problem is that it is hard to implement special execution mechanisms such as query packs using embedded meta-call. Previous results showed that, for efficient execution of query packs, the structure of the pack needs to be known beforehand (Blockeel, Dehaspe, Demoen, Janssens, Ramon, and Vandecasteele 2002; Demoen, Janssens, and Vandecasteele 1999). This means that it cannot be integrated with a meta-call that does not use any form of compilation (or analysis) of the query pack. Moreover, integrating packs with the embedded meta-call would necessitate reimplementing the complete pack execution mechanism in the embedded meta-call instructions themselves. These properties make the embedded meta-call less attractive for further development.

4.3 Control Flow Compilation

The major reason why meta-call can be competitive with running a compiled query is that the code for the compiled query contains instructions for setting up the arguments of the called goals (i.e. the `put` instructions). These require costly emulator cycles in compiled code, whereas setting up the arguments for the meta-called goal happens in the same emulator cycle as the call itself. Moreover, compilation itself is costly due to the non-linear allocation tasks such as assigning variables to environment slots, managing argument registers, ...

It would be interesting to combine the advantage of meta-interpretation (avoiding to set up arguments to goals using `put` instructions) with a simple form of compilation without expensive operations such as register allocation. Such a simple compiler would amongst others have the advantage that it can inline built-ins, and would be easy to extend. For this purpose, we introduce *control flow compilation*. The idea is to generate code for a query which describes the flow of control, but where the goals themselves are still meta-called, in the sense that their arguments have been preconstructed on the heap before the execution has started. The code generated by control flow compilation looks very much like ordinary code, but it does not contain any instructions related to arguments of goals nor variables. We illustrate the idea in Section 4.3.1 by a sequence of steps that will lead to a simplified form of our desired compilation scheme. We then present the actual control flow compilation technique in Section 4.3.2, and evaluate it in Section 4.3.3.

4.3.1 Intuition

Consider the query in Figure 4.11(a). We can flatten this query out into a structure from Figure 4.11(b), containing every literal of the query as one of its arguments. We can now write a predicate `call_query/1` which, given the constructed term as an argument, executes the original query. The Prolog code for this program is shown in Figure 4.11(c). Calling the query from Figure 4.11(d) results in the same execution as executing the original query. Notice how this code reflects the structure of the query, but calls the individual goals using meta-call.

Compiling the `call_query/1` predicate results in the WAM code from Figure 4.12(a). This block of instructions starts by allocating the environment on the stack, and putting the argument of the predicate (the `query/4` term) into the second variable slot of the environment, `Y2`. Then, each group of instructions labeled with △ and ◯ represents the call to `arg/3`, immediately followed by a call to `call/1`. Except for the varying integer argument for `arg/3` and the `call` or `deallex` instruction in the end, these groups are identical. We can therefore merge the instructions labeled △ and ◯ into new instructions `arg_call` and `arg_deallex` respectively. These new instructions fetch a given argument from the structure in `Y2`, put it in the first argument register for `call/1`, and finally either call `call/1` (for `arg_call`) or deallocate the environment and execute `call/1` (for `arg_deallex`). Using these new instructions, the new code

$$? - \ \mathtt{a}(\mathtt{X},\mathtt{Y}),(\mathtt{b}(\mathtt{Y},\mathtt{Z});\mathtt{c}(\mathtt{Y},\mathtt{Z})).$$

(a) Original query.

$$\mathtt{query}(\mathtt{a}(\mathtt{X},\mathtt{Y}),\mathtt{b}(\mathtt{Y},\mathtt{Z}),\mathtt{c}(\mathtt{Y},\mathtt{Z}))$$

(b) Flattened query structure.

```
call_query(Query) :-
    arg(Query,1,G1),
    call(G1),
    ( arg(Query,2,G2),
      call(G2)
    ; arg(Query,3,G3),
      call(G3)
    ).
```

(c) Generated `call_query/1` predicate.

$$?\text{- } \mathtt{Q} = \mathtt{query}(\mathtt{a}(\mathtt{X},\mathtt{Y}),\mathtt{b}(\mathtt{Y},\mathtt{Z}),\mathtt{c}(\mathtt{Y},\mathtt{Z})),\ \mathtt{call\_query}(\mathtt{Q}).$$

(d) Transformed query.

Figure 4.11: Different steps in the argcall transformation.

```
        allocate 3
        getpvar Y2 A1
        put_int A2 1              △
        putpval Y2 A3             △
        builtin_arg_3 A3 A2 A1    △
        call call/1               △
        trymeorelse L1
        put_int A2 2              ◯
        putpval Y2 A3             ◯
        builtin_arg_3 A3 A2 A1    ◯
        deallex call/1            ◯
L1:     trustmeorelsefail
        put_int A2 3              ◯
        putpval Y2 A3             ◯
        builtin_arg_3 A3 A2 A1    ◯
        deallex call/1            ◯
```

(a) Classical WAM code.

```
        allocate 3
        getpvar Y2 A1
        arg_call 1
        trymeorelse L1
        arg_deallex 2
L1:     trustmeorelsefail
        arg_deallex 3
```

(b) Using `arg_call` and `arg_deallex`.

Figure 4.12: Compiled code for `call_query/1` from Figure 4.11(c). The △ and ◯ symbols indicate the instructions that can be merged into one instruction.

for `call_query` becomes the code in Figure 4.12(b). What remains is only the control flow of the original query.

Using the ideas above, we develop a query compiler which, given a query, flattens the query into a term and generates code encoding the control flow of the original query. This compiler is very simple and lightweight, as all it has to do is analyze the structure of the query and generate instructions accordingly. After the code has been emitted, the query can be executed by calling the newly created predicate with the constructed term as its argument. We have therefore achieved our goal of combining a simple form of compilation with meta-calling of goals.

Although the approach described in this section can be implemented and used in practice (Tronçon, Janssens, and Demoen 2003), we refine it further in the next section to avoid the need of the `query/N` structure. Doing so has several advantages:

- The compilation step no longer needs to construct a `query/N` structure for every query.
- During execution of `arg_call` and `arg_deallex`, there are two indirections that have to be followed to get to the called goal, as is illustrated in the example in Figure 4.13. Omitting the `query/N` structure removes one indirection, and therefore speeds up the `arg_call` and `arg_deallex`

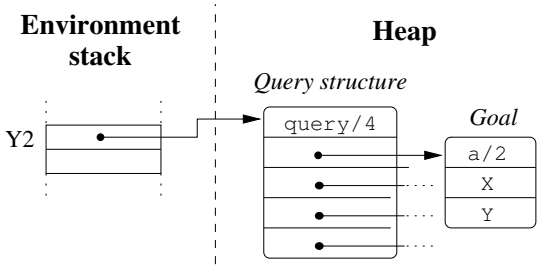


Figure 4.13: Memory layout for a `query/4` structure. The `arg_call` instruction needs to follow two indirections to get to `a(X,Y)`.

instructions.

- Extra goals can easily be added to queries after compiling the query, allowing a lazy compilation scheme to be developed.

4.3.2 Control Flow Compilation

As explained in Section 4.3.1, the main idea of control flow compilation is to compile only the control flow instructions, and to use a special type of meta-call instruction to actually call the goals. For that purpose, we introduce two new WAM instructions `cf_call` and `cf_deallex`, whose argument points to a heap data structure (the goal) that is to be meta-called. Hence, control flow code only contains the control flow instructions (`try*`, `retry*`, ...), `cf_call`, and `cf_deallex` instructions.

For example, control flow compiling the query

`query` :- `a(X,Y)`, (`b(Y,Z)` ; `c(Y,Z)`, `d(Z,U)`; `e(a,Y)`).

results in the code in the left part of Figure 4.14. Notice that, because queries are dynamically generated by the ILP system, the query itself is a term on the heap, and we use `&a(X,Y)` to represent the pointer to its sub-term `a(X,Y)`. On the right of Figure 4.14 is the classical compiled code for the same query. Before calling each goal, the compiled code first sets up the arguments to the goal, whereas the control flow compiled code uses a reference to the sub-term of the query to indicate the goal that is called. The most important aspect is that the control flow code saves emulator cycles, because it contains no instructions related to the arguments of the goals that are called. Moreover, the fact that these kinds of instructions are no longer necessary has other positive consequences: (1) it makes the expensive (non-linear) argument register allocation step unnecessary, saving compilation time, and (2) it makes it easy to incrementally add new code to existing parts of code. The latter is very interesting because it makes introducing laziness in the compilation process possible, as explained in Section 4.4.

```
query :- a(X,Y), ( b(Y,Z) ; c(Y,Z), d(Z,U); e(a,Y) ).
```

	Control flow code	Compiled code
	allocate 2	allocate 4
		bldtvar A1
		putpvar Y2 A2
	cf_call &a(X,Y)	call a/2
	trymeorelse L1	trymeorelse L1
		putpval Y2 A1
		bldtvar A2
	cf_deallex &b(Y,Z)	deallex b/2
L1:	retrymeorelse L2	retrymeorelse L2
		putpval Y2 A1
		putpvar Y3 A2
	cf_call &c(Y,Z)	call c/2
		putpval Y3 A1
		bldtvar A2
	cf_deallex &d(Z,U)	deallex d/2
L2:	trustmeorelsefail	trustmeorelsefail
		putpval Y2 A2
		put_atom A1 a
	cf_deallex &e(a,Y)	deallex e/2

Figure 4.14: Control flow compiled code vs. classical compiled code.

Compiled code	**Control Flow code**		
	No Inlining	Built-ins	Special Built-ins
`...`	`...`	`...`	`...`
`call a/2`	`cf_call &a(X,Y)`	`cf_call &a(X,Y)`	`cf_call &a(X,Y)`
`putpval Y2 A1`		`putarg &X A1`	
`putpval Y3 A2`		`putarg &Y A2`	
`b_smaller A1 A2`	`cf_call &(X<Y)`	`b_smaller A1 A2`	`cf_smaller &X &Y`
`...`	`...`	`...`	`...`

Figure 4.15: Built-in inlining for (`a(X,Y)`, `X < Y`).

Contrary to compiled code, control flow code cannot exist on its own, since it contains external references to terms on the heap. Therefore, an implementation must take the following garbage collection issues into consideration: (1) the terms of a query must be kept alive as long as its control flow compiled code can be executed; (2) when terms representing goals of a control flow compiled query are moved to another place in memory, the references in the code must be adapted as well. The extensions needed for the garbage collector to handle these issues are discussed in more detail in Section 4.6.2. Another fact that must be taken into account is that the execution of control flow compiled code can bind variables in the original term representing the query. This prevents recursive use of control flow compiled code, and requires the variable bindings to be undone before calling the query again. However, since control flow compilation is targeted towards compiling queries, recursive calls do not occur. The variable bindings caused by the execution of a query are undone by backtracking after the query finished.

To speed up execution, the classical compilation scheme typically inlines smaller predicates (such as tests) using dedicated instructions implemented in the system. This is illustrated by the first column of Figure 4.15: the WAM compiler emits instructions to initialize the argument registers, and instead of emitting a `call` to a (WAM-compiled) `'<'/2` predicate, it emits a built-in instruction to do the test. Since control flow compilation also emits WAM instructions, the same built-ins can be used for control flow compiled code as for classical compiled code. These built-in instructions typically use argument registers for their arguments, so the compiler just needs to emit extra instructions to move data structures on the heap into the correct argument registers. These are illustrated in the third column of Figure 4.15, where the `putarg` instructions move references to data structures on the heap into the relevant argument registers for the built-in instruction. In the spirit of the `cf_call` instruction, the extra emulator cycles needed for filling the argument registers can be omitted by defining special versions of each built-in that, instead of argument registers, have references to the heap as their parameters. An example of such a built-in is the `cf_smaller` instruction from Figure 4.15.

Another possible optimization is creating specialized versions of the `cf_call`

and `cf_deallex` instructions for goals with a fixed arity. For example, goals with arities smaller than 4 occur very frequently in ILP applications, and we therefore introduce `cf_call_4` and `cf_deallex_4` instructions for calling these goals. Unfolding the loop to set up all the arguments to the call improves execution speed.

A final addition to the set of control flow instructions is the `cf_unifyhead` instruction. This instruction is used whenever parameters need to be passed to the control flow compiled code. Instead of using an atom `query/0` as a head (cfr. Figure 4.14), any term can be used, possibly containing variables occurring in the body of the query. Before executing the control flow compiled code of the body, the instruction `cf_unifyhead` unifies the argument registers with the variables in the term representing the head of the query.

Figure 4.16 shows pseudo-code for the `cf_call` instruction, the specialized `cf_call_4` instruction, the `cf_smaller` builtin, and the `cf_unifyhead` instruction. A control flow compiler supporting these built-in instructions can be found in Appendix C.

4.3.3 Evaluation

To evaluate our approach, we added support for control flow code to the hipP system and implemented a separate control flow compiler for queries. The new compiler was written in Prolog, as was the existing classical compiler. For the built-in predicates that are frequently used in ILP applications (e.g. `'<'/2`, `'>'/2`, `'='/2`, `'\='/2`, ...), we implemented special control flow instructions (such as `cf_smaller` from Figure 4.15), and these built-ins are inlined by the control flow compiler. The heap garbage collector of hipP was modified to support control flow compiled code (see Section 4.6.2).

Two kinds of experiments are discussed: the benchmarks in Table 4.3 show the potential gain in an artificial setting, whereas the results in Table 4.4 are obtained from a real world application.

The artificial experiment consists of generating the same kind of parameterized queries as in Section 4.2.3. We measure both the time required to compile a query and to execute it. Table 4.3 shows that control flow compilation is clearly faster than compile-and-run: the compilation times are one order of magnitude better, while the execution times also show improvement. The compilation in the control flow approach is much faster because it does not need to perform expensive tasks such as assigning variables to environment slots. The better execution times are explained by the fact that only one emulation cycle per call is needed as no arguments have to be put in registers. Doubling the G parameter more or less doubles the timings. For larger queries, namely for $G = 10$, $B = 10$, $D = 4$, and for $G = 5$, $B = 5$, $D = 6$, control flow compilation becomes up to a factor 16 faster than compile-and-run. If the query is executed a sufficient number of times, meta-call is outperformed by control flow compilation (e.g. for $G = 5$, $B = 5$, $D = 4$, this number is 15). Since in ILP, each query is run on thousands of examples, these results indicate that control flow compilation is

```
instruction CF_CALL (Struct) :
    for 1 ≤ i ≤ arity(functor(Struct)) :
        ARG_i := arg(Struct, i)
    CONTP := PC + 1
    PC := entry_point(functor(Struct))
    continue

instruction CF_CALL_4 (Struct) :
    ARG_1 := arg(Struct, 1)
    ARG_2 := arg(Struct, 2)
    ARG_3 := arg(Struct, 3)
    ARG_4 := arg(Struct, 4)
    CONTP := PC + 1
    PC := entry_point(functor(Struct))
    continue

instruction CF_SMALLER (Ref1, Ref2) :
    if test_smaller(Ref1, Ref2) :
        PC := PC + 1
        continue
    else :
        fail

instruction CF_UNIFYHEAD (Struct) :
    for 1 ≤ i ≤ arity(functor(Struct)) :
        if not unify(ARG_i, arg(Struct, i)) :
            fail
    PC := PC + 1
    continue
```

Figure 4.16: Pseudo-code for the `cf_call`, `cf_call_4`, `cf_smaller`, and `cf_unifyhead` instructions. `PC`, `CONTP`, and $\texttt{ARG}_i$ represent the program counter, continuation pointer, and argument registers respectively.

Query				Experiment	Time	
G[a]	B[b]	D[c]	T[d]		Comp.[e]	Exec.[f]
5	5	4	3905	Meta-call	-	2.14
				Compile & Run	288.0	0.38
				Control Flow	30.8	0.20
10	5	4	7810	Meta-call	-	4.20
				Compile & Run	668.0	0.73
				Control Flow	62.6	0.36
5	10	4	55555	Meta-call	-	33.68
				Compile & Run	6368.0	5.64
				Control Flow	457.4	3.15
10	10	4	111110	Meta-call	-	64.94
				Compile & Run	13876.0	10.82
				Control Flow	847.8	5.72
5	5	6	19531	Meta-call	-	59.20
				Compile & Run	11596.0	9.93
				Control Flow	758.0	5.40

[a]Number of goals in a branch
[b]Branching factor of each disjunction
[c]Nesting depth of disjunctions
[d]Total number of goals ($= G \sum_{n=0}^{n=D} B^n$)
[e]Compilation time of the query (in ms.)
[f]Execution time of the query (in ms.)

Table 4.3: Experiments for artificial disjunctions.

Dataset	Experiment	Comp.[a]	Exec.[b]	Total
Mutagenesis	Meta-call	-	1.43	1.43
	Compile & Run	1.30	1.06	2.36
	Control Flow	0.21	1.02	1.23
Bongard	Meta-call	-	24.70	24.70
	Compile & Run	4.98	21.34	26.32
	Control Flow	0.91	21.18	22.09
Carcinogenesis	Meta-call	-	108.81	108.81
	Compile & Run	17.50	65.30	82.80
	Control Flow	2.24	59.51	61.75
HIV	Meta-call	-	50291.30	50291.30
	Compile & Run	1196.54	12918.43	14114.97
	Control Flow	191.10	12030.82	12221.92

[a] Total compilation time of all queries (in seconds)
[b] Total execution time of all queries (in seconds)

Table 4.4: Experiments for conjunctions from a real world application.

the best suited alternative.

The real world experiment consists of running TILDE on the Mutagenesis (Appendix A.1), Carcinogenesis (Appendix A.2), Bongard (Appendix A.3), and HIV (Appendix A.4) datasets. During the execution of TILDE, queries are generated that need to be run on a subset of the examples. These queries consist only of conjunctions, and every query is executed separately on the examples. Table 4.4 shows the compilation time and execution time for all queries in the control flow compilation approach with the corresponding times of the compile-and-run and the meta-call approach.

In the TILDE runs, control flow compilation gains a factor 5 to 8 over usual compilation. For all datasets, control flow compiled code also outperforms both the classical compiled code and the meta-called queries. Meta-call is slower than control flow compiled code because of the extra emulator cycles spent in testing the incoming goal upon each call.

We conclude that control flow compilation is the fastest approach for executing the queries on these datasets. The main reason for this is that the share of query compilation in the total execution time of the ILP algorithm is reduced significantly. Moreover, control flow compiled code contains less instructions, and therefore saves emulator cycles as well.

The results are more pronounced for the artificial benchmarks than for the TILDE ones for several reasons. The artificial queries are longer than the typical TILDE queries. We observed that short queries make the timings unreliable. During the artificial benchmarks, the time spent in the called goals is very small, whereas in the TILDE experiments much more time is spent on executing the called predicates. Therefore, the effect of control flow on the *exec* timing

decreases.

4.3.4 Conclusion

The main goal of control flow compilation was to reduce high compilation times, without slowing down execution itself. Our experiments show that control flow compilation achieves this goal: compilation times are reduced by an order of magnitude, while the execution becomes even slightly faster. Moreover, the new compilation scheme is flexible, and allows for extensions such as lazy compilation (Section 4.4) and query packs (Section 4.5).

4.4 Lazy Control Flow Compilation

4.4.1 Technology

We observed that, during the execution of ILP algorithms on real-life datasets, large parts of the queries generated by the query generation process are never executed. Hence, unnecessary time is spent in compiling this unreachable code. With a lazy compilation scheme that only compiles code when it is actually reached, this redundancy can be removed. Control flow compilation is particularly suited for this dynamic kind of code, since existing compiled code can be extended without needing to alter the latter because of e.g. argument register allocation (as is the case with classical compilation). In this section, we extend the control flow compilation approach, and develop a lazy variant.

In (Aycock 2003), *lazy compilation* is identified as a kind of *just-in-time* (JIT) compilation or *dynamic compilation*, which is characterized as translation that occurs after a program begins execution. Our lazy variant implicitly calls the control flow compiler when execution reaches a part of the query that is not yet compiled. We restrict the discussion in this section to queries with conjunctions and disjunctions; the extension to query packs is presented in Section 4.5.

As before, the query that needs to be evaluated is represented by a term on the heap. We introduce a new WAM instruction `lazy_compile`, whose argument is a pointer to the term on the heap that needs compiling when execution reaches this instruction.

Consider the query

```
query :- a(X,Y), b(Y,Z).
```

The initial lazy compiled version of `query/0` is

```
allocate 2
lazy_compile &(a(X,Y),b(Y,Z))
```

The `lazy_compile` instruction points to a conjunction. Its execution replaces itself by the compiled code for the first conjunct, namely a `cf_call`, and adds for the second conjunct another `lazy_compile` instruction, resulting in:

```
allocate 2
cf_call &a(X,Y)
lazy_compile &b(Y,Z)
```

Execution continues with the newly generated `cf_call` instruction. When this call succeeds, the next `lazy_compile` instruction is executed, after which the compiled code is identical to code generated without laziness:

```
allocate 2
cf_call &a(X,Y)
cf_deallex &b(Y,Z)
```

Notice that lazy compilation overwrites the `lazy_compile` instruction with a `cf_` instruction, and that when the query has been executed completely for the first time, the resulting code is the same as the code produced by non-lazy control flow compilation. If the query did not succeed, a part of the query will remain uncompiled.

As a second example, consider the lazy compilation of the query from Figure 4.14 (page 65):

```
q :- a(X,Y), ( b(Y,Z) ; c(Y,Z), d(Z,U); e(a,Y) ).
```

Initially, the code is

```
allocate 2
lazy_compile &(a(X,Y),(b(Y,Z);c(Y,Z),d(Z,U);e(a,Y)))
```

The `lazy_compile` changes the code to:

```
allocate 2
cf_call &a(X,Y)
lazy_compile &(b(Y,Z);c(Y,Z),d(Z,U);e(a,Y))
```

Now, `lazy_compile` needs to compile a disjunction. Where normal (control flow) compilation would generate a `trymeorelse` instruction, we generate a lazy variant for it. The `lazy_trymeorelse` instruction has as its argument the second part of the disjunction, which will be compiled and executed upon failure of the first branch. The instruction is immediately followed by the code of the first branch, which is initially again a `lazy_compile`:

```
allocate 2
cf_call &a(X,Y)
lazy_trymeorelse &(c(Y,Z),d(Z,U);e(a,Y))
lazy_compile &b(Y,Z)
```

Execution continues with the `lazy_trymeorelse`: a special choice point is created such that on backtracking the remaining branches of the disjunction will be compiled in a lazy way. To achieve this, the failure continuation of the choice point is set to a new `lazy_disj_compile` instruction, which behaves similarly to `lazy_compile`. Then, execution continues with the first branch:

```
        allocate 2
        cf_call &a(X,Y)
        lazy_trymeorelse &(c(Y,Z),d(Z,U);e(a,Y))
        cf_deallex &b(Y,Z)
```

Upon backtracking to the special choice point created in `lazy_trymeorelse`, the `lazy_disj_compile` instruction resumes compilation, and replaces the corresponding `lazy_trymeorelse` by a `trymeorelse` instruction with the address of the code to be generated as argument:

```
        allocate 2
        cf_call &a(X,Y)
        trymeorelse L1
        cf_deallex &b(Y,Z)
    L1: lazy_retrymeorelse &(e(a,Y))
        lazy_compile &(c(Y,Z),d(Z,U))
```

Here, `lazy_retrymeorelse` (the lazy variant of `retrymeorelse`) behaves similar to `lazy_trymeorelse`, but instead of creating a special choice point, it alters the existing choice point. It is immediately followed by the code of the next part of the disjunction, which after execution looks as follows:

```
        allocate 2
        cf_call &a(X,Y)
        trymeorelse L1
        cf_deallex &b(Y,Z)
    L1: lazy_retrymeorelse  &(e(a,Y))
        cf_call &c(Y,Z)
        cf_deallex &d(Z,U)
```

Upon backtracking, `lazy_retrymorelse` is overwritten, and a `trustmeorelse` is generated for the last branch of the disjunction, followed by a `lazy_compile` for this branch:

```
        allocate 2
        cf_call &a(X,Y)
        trymeorelse L1
        cf_deallex &b(Y,Z)
    L1: retrymeorelse L2
        cf_call &c(Y,Z)
        cf_deallex &d(Z,U)
    L2: trustmeorelsefail
        lazy_compile &e(a,Y)
```

After the execution of the last branch, we end up with the full control flow code.

The amount of code compiled during one step in the JIT compilation process can be varied to yield different compilation granularities:

```
?- a(X), b(Y), X > Y.

a(1).
a(2).
b(1).
```

Control Flow code
`cf_call &a(X)`
`cf_call &b(Y)`
`cf_greater_int 1 &Y`

Figure 4.17: Overly specialized control flow compiled code.

- *Per goal:* Every time the JIT compiler is called, it compiles exactly one goal, and then resumes execution. This is the granularity used throughout the example above.
- *Per conjunction:* Every time the JIT compiler is called on a query, all the goals in a conjunction (up to a disjunction) are compiled at once. This avoids frequent switching between the compiler and the execution by compiling bigger chunks.
- *Per disjunction:* All the branches of a disjunction are compiled up to the point where a new disjunction occurs. This approach is reasonable from an ILP viewpoint: the branches of a disjunction represent different queries, and since the success of each query is recorded, all branches will be tried (and thus compiled) eventually.

Apart from the overhead of switching between compilation and execution, these approaches might also generate different code depending on the execution itself. When a goal inside a disjunction fails, the next branch of the conjunction is executed, and newly compiled code is inserted at the end of the existing code. When in a later stage the same goal succeeds, the rest of the branch is compiled and added to the end of the code, and a jump to the new code is inserted. These jumps cost extra emulator cycles and decrease locality of the code. Lazy compilation per goal can in the worst case have as many jumps as there are goals in the disjunctions. Compiling per conjunction can have as many jumps as there are disjunctions. If a disjunction is completely compiled in one step, each branch of the disjunction ends in a jump to the next disjunction.

Just as for control flow compilation, special control flow instructions for built-in predicates can be used in the lazy variant. Care must be taken though: typically, specialized built-ins are emitted depending on the type of arguments (e.g. specialized built-ins for unifying arguments with integers); however, as compilation is now interleaved with execution, arguments of a goal might have been bound after starting the execution of the query, which could make the emitted built-in overly specialized, thus generating code that becomes erroneous after backtracking or when run on another example. Figure 4.17 illustrates this issue. After having compiled and executed the first two goals, binding `X` and `Y` to 1, the control flow compiler needs to compile (`X > Y`). Because `X` is bound to 1, an optimizing compiler emits a specialized test comparing `Y` to 1. However,

this code will fail even after selecting a new solution for `a/1`, although the query should succeed. The compiler should therefore not emit specialized built-ins depending on the instantiation and/or type of the arguments, or it should keep track of the state of the goal arguments in the original query. In our implementation, we chose for the former approach.

Finally, notice that this lazy control flow compilation can be used to exploit the incremental nature of a query generation process such as the one from ILP. By constructing queries with an open end, and letting the compiler generate a `lazy_compile` instruction for such open ends, these open ends can be instantiated by a later query generation phase. For example, compiling the query

```
query :- a(X,Y), b(Y,Z), End1.
```

results in the following code:

```
allocate 2
cf_call &a(X,Y)
cf_deallex &b(Y,Z)
lazy_compile &End1
```

The `lazy_compile` is implemented to do nothing if its argument is an unbound variable, and so the query executes as before. In the following iteration, the refinement step of the ILP algorithm can now unify the variable `End1` with (`c(Z)`, `End2`), resulting in the following query:

```
query :- a(X,Y), b(Y,Z), c(Z), End2.
```

By instantiating the variable `End1`, the compiled code for the query becomes

```
allocate 2
cf_call &a(X,Y)
cf_deallex &b(Y,Z)
lazy_compile &(c(Z),End2)
```

Executing the query will now only compile the code increment, whereas previously the whole query needed to be recompiled. However, as the experiments from Section 4.4.2 show, control flow compilation times are very low, and therefore the incremental compilation approach cannot yield any significant speedups in total query evaluation time with respect to the use of (lazy) control flow compilation.

4.4.2 Evaluation

In this section, we measure the overhead of the new lazy compilation scheme. The artificial queries from Table 4.5 have no unreachable parts, and as such provide a worst case for lazy compilation overhead. In practice, queries have unreachable parts, and so the total overhead of the lazy compilation scheme will be compensated by the smaller compilation time. The experiments of Table 4.5

Query			Experiment	Time
G[a]	B[b]	D[c]		Total[d]
5	5	4	Per Goal	55
			Per Conjunction	34
			Per Disjunction	32
			No Laziness	28
10	5	4	Per Goal	111
			Per Conjunction	60
			Per Disjunction	59
			No Laziness	59

[a]Number of goals in a branch
[b]Branching factor of each disjunction
[c]Nesting depth of disjunctions
[d]Total Compilation + Execution time of the query (in ms.)

Table 4.5: Lazy compilation for several kinds of disjunctions.

use only the first two benchmarks from Table 4.3. The other benchmarks of Table 4.3 give similar results. Timings are given for the different settings of lazy compilation. The timings report the time needed for one execution of the query, thus including the time of its lazy compilation. These timings are then compared with the time of performing non-lazy control flow compilation of the query and executing it once[1]. Lazy compilation per goal clearly has a substantial overhead, whereas the other settings have a small overhead. We also measured the execution times for the three lazy alternatives once they are compiled: they were all equal, and are therefore not included in the table.

The main message here is that the introduction of laziness in the control flow compilation does not degrade performance much, and that it opens perspectives for query packs compilation: (1) lazy compilation is fast; (2) in real life benchmarks, some branches are never compiled due to failure of goals, whereas in our artificial setting all goals in the queries succeed.

4.5 Lazy Control Flow Compilation for Query Packs

4.5.1 Technology

So far, we restricted our (lazy) control flow compilation approach to queries containing conjunctions and 'ordinary' disjunctions. However, in the context of ILP, it is more interesting to optimize the execution of query packs. Since query packs are essentially a special kind of disjunction, implemented using dedicated

[1]Notice that these timings are slightly higher than the sum of *Comp.* and *Exec.* in Table 4.3. This is due to the fact that both experiments are run in different circumstances.

Dataset	Unused[a]	Experiment	Comp.[b]	Exec.[c]	Total[d]
Mutagenesis	17%	Compile & Run	0.52	0.11	0.63
		Control Flow	0.07	0.10	0.17
		Lazy Control Flow	-	-	0.14
Bongard	51%	Compile & Run	1.91	1.17	3.08
		Control Flow	0.28	1.15	1.43
		Lazy Control Flow	-	-	1.37
Carcinogenesis	32%	Compile & Run	7.39	4.63	12.02
		Control Flow	0.81	3.81	4.62
		Lazy Control Flow	-	-	4.34
HIV	74%	Compile & Run	209.47	191.68	401.15
		Control Flow	27.13	178.53	205.66
		Lazy Control Flow	-	-	186.22

[a]Total % of the query code that is never executed
[b]Total compilation time of all queries (in seconds)
[c]Total execution time of all queries (in seconds)
[d]Total query evaluation time (= Comp. + Exec.)

Table 4.6: Experiments for query packs from a real world application.

WAM instructions, we can easily extend control flow compilation to support query packs by emitting different control flow instructions.

As experiments in Section 4.4 pointed out, the choice of the actual lazy compilation variant does not matter with respect to the overhead introduced (except for lazy compilation per goal). Because the query pack data structures store information about all the branches of every disjunction, it is more convenient to use the lazy compilation variant that compiles one complete disjunction at once. As explained in Section 4.4, this means that each branch of a disjunction ends in a jump. This corresponds to the actual implementation of query packs, where all branches of disjunctions end with a `pack_try` instruction. This means that lazy compilation of query packs does not introduce extra emulator cycles because of `jump` instructions in the code.

Notice that the actual implementation of query packs assumes that the complete structure of the pack is known beforehand, and uses this information to initialize the query pack data structures. Because the complete structure is no longer known beforehand, the implementation of query packs needs some modifications in order to allow incrementally adding new disjunctions to the data structures.

4.5.2 Evaluation

We evaluate (lazy) control flow compilation for query packs by running TILDE, and letting it generate query packs instead of conjunctions (as we did for Table 4.4). The experiments are performed on the same ILP datasets used in Table 4.4 (page 70).

Table 4.6 shows both the total compilation time and the execution time for compile-and-run and control flow compilation; for lazy control flow compilation, no distinction can be made, and so the total time for compilation and execution is given. Additionally, we give for each dataset the amount of query goals that are never reached by the query execution. Comparing the timings for the query packs with the timings for the sets of queries in Table 4.4 (page 70), we see that the query packs are considerably faster.

Comparing control flow compilation with compile-and-run, we see that control flow compilation is up to an order of magnitude faster than traditional compilation, even though the hipP system already has a compiler that is optimized for dealing with large disjunctions (Vandecasteele, Demoen, and Janssens 2000) (in particular for the classification of variables in query packs). The execution times show the same characteristics as in the experiments with the conjunctions in Table 4.4: control flow has a faster execution than classical compilation. For the ILP application, the total time must be considered: the total time of control flow is up to a factor 3 faster than compile-and-run. Notice that this factor is higher for the query packs than for the conjunctions. The timings show that, for our benchmarks, the compilation time in compile-and-run is systematically larger than the execution time for all the examples such that the impact of improving the compilation has a larger effect on the total times.

Table 4.5 shows that lazy compilation has some overhead, but it is compensated in all experiments by avoiding the compilation of failing parts in the query packs. The time gained by not compiling unused parts of queries corresponds roughly to the measured amount of unreached goals.

The timings indicate that lazy control flow compilation is the best approach for query packs.

4.6 Memory management issues

In this section, we briefly discuss some memory management issues when dealing with (lazy) control flow compiled code.

4.6.1 Locality

Because the execution of control flow compiled code needs to fetch the data for its calls from the heap, the compiled code should be located as close as possible to the data it consumes, in order to have good locality of data and therefore to improve caching. This can be achieved by allocating control flow compiled code on the heap, and by extending the heap garbage collector to support this new data structure (see Section 4.6.2). Because of the dynamic nature of lazy compiled code, control flow blocks can be scattered across the heap during execution; the heap garbage collector moves these blocks closer to each other during collection, which improves locality.

The locality of the query goals themselves also has an impact on execution time. During the query generation phase, other data is allocated on the heap,

Dataset	Experiment	Comp.[a]	Exec.[b]
Mutagenesis	Compile & Run	0.52	**0.11**
	Control Flow (No Copy)	0.08	**0.10**
	Control Flow (With Copy)	0.07	**0.10**
Bongard	Compile & Run	1.91	**1.17**
	Control Flow (No Copy)	0.33	**1.19**
	Control Flow (With Copy)	0.28	**1.15**
Carcinogenesis	Compile & Run	7.39	**4.63**
	Control Flow (No Copy)	0.70	**4.11**
	Control Flow (With Copy)	0.81	**3.81**
HIV	Compile & Run	209.47	**191.68**
	Control Flow (No Copy)	27.81	**193.90**
	Control Flow (With Copy)	27.13	**178.53**

[a]Total compilation time of all queries, including the time to copy the query (in seconds)
[b]Total execution time of all queries (in seconds)

Table 4.7: Impact of locality on execution times

which can lead to a situation where the goal terms of the query (which are used during execution of the control flow compiled code) are scattered across the heap. We therefore create a copy of the (possibly scattered) term representing the complete query before compiling it. This ensures that all the terms used in the compiled code are allocated together on the heap. The impact of this copying step is illustrated in Table 4.7, showing the query execution time when running Tilde on the same datasets as in Table 4.6. Without copying the goals, the execution time of control flow compiled code becomes slower than code executed using the classical approach. Copying the term before compiling it sometimes introduces a slight overhead in some of the smaller benchmarks, but this is compensated by the gain in execution time.

4.6.2 Extending the garbage collector

As mentioned in Section 4.3.2, the garbage collector needs to be extended in order to correctly support control flow code: goals referenced from control flow compiled code need to be kept alive together with the code itself, and the references in the code need to be modified whenever the goals are relocated. Additionally, because we choose to keep the control flow code itself on the heap, the heap garbage collector also needs to take references between control flow compiled code blocks: if the argument of a choice instruction (`try*`, `retry*`, ...) points to another block, the collector needs to update this reference whenever the target block is moved. We therefore introduce modifications of the mark-and-copy garbage collector of hipP, in order to support lazy compiled control flow code.

During the marking phase of the garbage collection process, all reachable

blocks of code and the goals they reference need to be marked. The entry points of all control flow compiled predicates are added to the root set, and the marking phase traverses all the code blocks belonging to the predicate by following references to other blocks in the choice instructions. Terms referenced by the `cf_call` and `cf_deallex` instructions are also marked, as they need to be kept alive as long as the code using them is alive. When the scan pointer detects code blocks during the copying phase, all references to other code blocks (in the choice instructions) or terms (in the `cf_call` and `cf_deallex` instructions) are copied to the to-space, and the copied blocks are replaced by forwarding pointers.

4.7 Conclusions

In this chapter, alternatives for the classical compile-and-run approach for evaluating dynamically generated queries were investigated. The main motivation underlying this investigation was the large share the compilation step takes in the total query evaluation time. Using an embedded meta-call removes the need of a costly compilation step of queries altogether, yet it suffers from some drawbacks: although executing queries using the embedded meta-call is 5 times faster than with the normal meta-call, execution is sometimes still slower than compiled code; moreover, extending the meta-call further to support built-in operations and special execution mechanisms (such as query packs) is very cumbersome, and leads to even more slowdowns. Control flow compilation, a hybrid between meta-calling and compilation, uses a compilation step that is an order of magnitude faster than classical compilation, without affecting the execution time. The benefits of control flow compilation versus classical compilation are clear and are confirmed in the context of real world applications from the ILP community. Moreover, the lazy variant provides additional speedup in the total time by not compiling unreached parts of the query.

Traditionally, Prolog implementations have implemented a form of JIT, where compilation to WAM code or machine code happens at consult time. YAP (Damas and Costa 2003) goes one step further and compiles a predicate to abstract machine code at the first call to that predicate. BinProlog (Tarau 1992) switches back and forth between a compiled and an interpreted form of dynamic predicates, based on the relative frequency of modification and execution of the predicate. hipP uses a less flexible version of this scheme, and compiles a dynamic predicate when it is first called. The granularity of these JIT compilation forms is always a predicate, while control flow compiled code can have a finer grained granularity up to a literal. On the other hand, control flow compilation cannot be used for compiling recursive predicates.

YAP, which is used by the Aleph ILP system (Srinivasan 2005), provides other implementation techniques for speeding up the evaluation of many queries against many examples. In particular, tabling and dynamic indexing can speed up the query execution phase considerably. Our control flow compilation scheme

is orthogonal to these techniques, and can be combined with them. Especially when tabling is used (see Chapter 6), it is important to spend little time on compiling the queries, as tabling avoids repeated execution of the same goal (or prefix of a query). So, we expect that control flow compilation is beneficial in combination with tabling.

Within the ILP setting, the applications of (lazy) control flow compilation can be extended further. Extending control flow compilation to support adpacks (Chapter 3) is straightforward, and we expect this to yield the same speedups as for query packs. However, introducing laziness requires more work, and the impact of laziness needs to be investigated. In (Ramon and Struyf 2004), a technique for efficient theta-subsumption is proposed which uses query pack execution. It has to be investigated whether lazy control flow compilation reduces the compilation time enough in the particular setting that executes the query pack only once, or that a pure meta-call based approach for the query packs performs better. Finally, we mentioned in Section 4.4 that lazy control flow compilation can be used to exploit the incremental nature of queries. Instead of compiling a refined query from scratch, the compiled code of the original query can be reused so that only the refinement has to be compiled. Because control flow compilation is very fast, further exploring this is not very interesting at this time, but it may be worth investigating this when compilation times *do* become significant again.

Chapter 5

Analyzing and Debugging Query Execution

5.1 Introduction

The development of new execution mechanisms for ILP happens mainly in the engine used by the ILP algorithm. These optimized execution strategies typically require a low level implementation to yield significant benefits. For example, the adpack execution mechanism from Chapter 3 required the introduction of new dedicated WAM instructions, together with a set of new data structures which these instructions use and manipulate. The embedded meta-call and the control flow compilation schemes from Chapter 4 also make use of new WAM instructions, implemented in the core of the system. Because of their low-level nature, finding bugs in the implementation of these execution mechanisms is very hard. While tracing bugs in these implementations of execution mechanisms might still be feasible for small test programs, many bugs only appear during the execution of the ILP algorithm on real life data sets. Several factors make debugging in this situation difficult:

- *The size of the ILP system itself.* Real life ILP systems (such as ACE) often have a very large code base. For example, ACE consists of over 150000 lines of code. This makes it very hard to use standard tracing to detect bugs.

- *The complexity/size of the ILP problem.* With large datasets, it can take a very long time (hours, even days) before a specific bug occurs. When debugging, one typically performs multiple runs with small modifications to pin-point the exact problem, and so long run times make this approach infeasible.

- *The high complexity of the hypothesis generation phase.* While the evaluation of hypotheses is often the bottleneck, some algorithms (such as

rule learners) have a very expensive hypothesis generation phase. For these algorithms, it can take a very long time for the bug in the execution mechanism to expose itself, even when the time spent on executing these queries is small.

- *Non-determinacy of ILP algorithms.* If an ILP algorithm makes random decisions, the exact point in time where the bug occurs changes from run to run. It is even possible that the bug does not occur at all in certain runs.

Not only do these properties hinder the tracing of bugs, they also pose problems for analysis of query execution. When developing new execution mechanisms, analyses of the execution phase of an ILP algorithm can provide very useful information to guide the development. Such useful analyses include:

- Statistics about the structure of the evaluated queries, such as the average query length, amount of (ad)pack-ors, maximum number of branches in (ad)pack-ors, the amount of activate/deactivate pairs, ...
- The total number of goals called and backtracked to during the execution of queries, query packs, and adpacks.
- The total memory used by data-structures used while executing (ad)packs.

These statistics not only serve in detecting potential bottlenecks in the execution of queries, they can also be used to compare various execution mechanisms. However, similar problems as the ones for debugging query execution mechanisms arise when one wants to gather information about query execution:

- Due to the size of the ILP system, adding the necessary code to perform the analyses is tedious.
- Gathering statistics from ILP algorithms with a complex hypothesis generation phase takes a very long time. If one wants to gather extra statistics, the ILP algorithm has to be run all over again, including the complex hypothesis generation phases.
- Due to non-determinacy of ILP algorithms, analysis results might differ from one run to another, making a fair comparison impossible.

An additional problem is that it is impossible to compare execution mechanisms from different ILP engines. For example, the query pack implementation is only available in hipP, and can therefore only be used by algorithms implemented on top of this engine (such as the algorithms from the ACE system). On the other hand, YAP (Damas and Costa 2003) provides an execution mechanism based on tabling (see Chapter 6), which can only be used by algorithms built on top of YAP (e.g. Aleph (Srinivasan 2005)). This means that there is no way to compare both the query packs approach with the tabling-based approach objectively, as they are implemented in different engines, each engine performing

differently. Moreover, if an ILP algorithm is implemented for a specific engine (or is part of a larger engine-dependent system), it not possible to compare the performance of different engines for the algorithm in question.

In the past, traces of execution have been used to understand misbehavior of programs (Ducassé 1999a; Ducassé 1999b) and to profile programs (Jahier and Ducassé 2002). In this chapter, we present a trace based approach for analyzing the runs of an ILP algorithm, without modifying the actual implementation of the algorithms. This approach is an effective algorithm- and engine-independent approach for easy and fast debugging of the underlying query execution engines, and provides an easy way to evaluate and compare the impact of different execution mechanisms. Moreover, we explain how debugging using these traces can be simplified by automatically limiting execution to the part causing a bug to appear. While we mainly focus on analyzing execution mechanisms, these trace-based approaches can help in analyzing behavior of entire ILP algorithms as well.

The organization of this chapter is as follows: Section 5.2 discusses the collection of the run-time information necessary for our trace based approaches. Section 5.3 then discusses using these traces to allow fast and easy debugging of query execution. Section 5.4 describes analyzing the gathered information to monitor the queries and their execution. Finally, we give some conclusions and future work in Section 5.5.

5.2 Gathering Run-time Information

Consider the generic ILP algorithm from Figure 2.3 (page 12). The target of engine optimizations as the ones described throughout this work is the *Evaluate* step, which takes a set of hypotheses to be evaluated, and evaluates them on the current dataset. The other steps that characterize the algorithm such as finding suitable refinements for queries are not important from an engine implementor's point of view. However, the latter are the most complex parts of the algorithm, and encompass most code of the algorithm itself. We extract enough information from an ILP run necessary to be able to reproduce the *Evaluate* step, without running the ILP algorithm itself. More specifically, we only need to know the queries that the algorithm runs, and on which example each query is evaluated. How and why these queries were generated and selected is irrelevant for reconstructing the execution step.

To extract the desired information, we modify the *Evaluate* step from the ILP algorithm to record all evaluated queries to a file, which we call the *trace* of the algorithm. An example of such a trace after running a modified algorithm can be seen in Figure 5.1. This trace represents a run of an ILP algorithm that executed 4 queries: 2 queries that were executed on all 5 examples, and 2 extensions of the first query, which were only executed on the first and the last example. Notice that this trace is no longer dependent of the concrete algorithm

```
query((atom(X,'c')), [1,2,3,4,5]).
query((atom(X,'h')), [1,2,3,4,5]).

query((atom(X,'c'),atom(Y,'o'), bond(X,Y)), [1,5]).
query((atom(X,'c'),atom(Y,'c'), bond(X,Y)), [1,5]).
```

Figure 5.1: Example trace of an ILP run.

```
% Run all queries from 'Trace' on 'Dataset'
simulate(Trace, Dataset) :-
    read(Trace, Input),
    ( Input == end_of_file ->
        true
    ;
        Input = query(Query, Examples),
        evaluate_query(Examples, Query, Dataset),
        simulate(Trace, Dataset)
    ).

% Evaluate a query on a set of examples
evaluate_query([], _, _).
evaluate_query([E|Es], Query, Dataset) :-
    load_example(Dataset, E),
    (call(Query), fail ; true),
    evaluate_query(Es, Query, Dataset).
```

Figure 5.2: `simulate/2`: A simple trace simulator.

itself, in the sense that it is just a sequence of queries the algorithm evaluated on the examples.

The gathered information can now be run through a trace simulator which, using the example database and background knowledge of the ILP algorithm, can now simulate the execution step of the ILP algorithm. A trivial trace simulator is shown in Figure 5.2, and does nothing but run the original queries on the corresponding examples. Such a simulator is not very useful in itself, except for debugging purposes (such as described in Section 5.3). However, extending this trivial simulator can yield more interesting applications, such as the query analyzers described in Section 5.4.1 and Section 5.4.2.

The information in the trace can be extended even further. While the analyses described in this chapter do not need to know the origin of queries, this information is useful for the analysis from Chapter 6, where we need to distinguish the prefix and the refinement of a query. Such information might also be useful

for other applications, such as debugging or visualizing ILP algorithms, or researching new query representations for incremental query compilation schemes such as mentioned in Section 4.4.

The sizes of the traces can grow quite large in practice. For example, running the TILDE algorithm on 4150 examples of the HIV data set results in a trace of 1113594 queries, encompassing a total of 235 MB. For our purposes, this does not pose any problems, since the debugging and analyses presented in this chapter only use sequential access to this data. However, when needing random access to the trace (e.g. for a visualizing 'trace browser'), the traces need smarter storage methods (e.g. building indexes on the traces).

5.3 Debugging Query Execution

When developing optimizations for query evaluation, different execution mechanisms are investigated. If a new execution mechanism theoretically yields the same final results as the existing ones, inconsistencies can be detected by running the ILP algorithm using each execution mechanism, and comparing the final results. For example, for TILDE, one can compare the learned decision trees to determine whether or not two runs are consistent with each other. However, an inconsistent result only indicates that there is a bug in the execution *somewhere*, but it does not show *where*. To be able to determine this, the complete ILP algorithm has to be run using both the debugger of the host language of the ILP algorithm (e.g. Prolog), and the debugger of the host language of the execution engine (e.g. C). Because of the size and complexity of ILP systems, debugging on both levels simultaneously is very hard and time-consuming in practice. Moreover, testing execution mechanisms by comparing the output of the algorithm only works when the algorithm has deterministic behavior: if the decisions it makes are based on a random factor, the outputs of the algorithm can (slightly) differ, and comparing runs is not possible. This makes locating bugs in the implementation of optimizations even harder. Using execution traces for debugging solves many of these problems: trace execution is deterministic, and a trace simulator is so small that the focus of the debugging process is purely on the optimization itself. Moreover, traces can speedup debugging even more drastically by limiting execution to the part of the trace causing the bug, as we show in this section.

When two runs of a deterministic ILP algorithm produce different results, this means that the query evaluation process selected different queries at some point. If the only difference between both runs is a query optimization scheme, this means that the optimization caused a query to succeed or fail where it did not before, meaning there is a bug in the optimization (assuming that optimizations preserve success or failure of queries). Testing optimizations can therefore be reduced to comparing the success of query with and without the optimizations scheme. This can be achieved by simply running the trace through a simulator that records query successes, and runs every set of queries with and

without the optimization enabled. Not only can such a simulator detect bugs this way, it can also pinpoint exactly in which query the bug occurs.

Due to the size of the trace, it might still be that a big part of the execution needs to be analyzed to find the bug. A bug occurring in a query is often also dependent on previously executed queries, which means that the trace cannot just be reduced to a single query to be able to reproduce and locate the bug. However, because the trace contains all the information determining the execution, locating a bug through traces can be turned into a *data slicing* (Chan and Lakhotia 1998) problem. The goal of data slicing is to take input data (i.e. a trace) that causes a bug to manifest itself, and reduce this data as much as possible to yield a smaller subset of data still exposing the bug. The standard approach to data slicing is simply to use binary search: split your data in two, test both halves, and continue with the half that still reproduces the bug. However, binary search might be too coarse-grained to find a bug, and as such fail to reduce the trace sufficiently. For example, if a bug occurs in the last query of the trace because of the execution of the first query, neither of both halves would reproduce the bug. *Delta Debugging* (Zeller and Hildebrandt 2002) is an automated data slicing technique that overcomes these issues.

Given a set of data $\mathcal{D}$ which causes a bug to appear. We denote this as $test(\mathcal{D}) = \text{fail}$. $\mathcal{D}_g \subseteq \mathcal{D}$ is a *global minimal data slice* if

$$test(\mathcal{D}_g) = \text{fail} \wedge \forall \mathcal{D}' \subseteq \mathcal{D} \cdot (|\mathcal{D}'| < |\mathcal{D}_g| \Rightarrow test(\mathcal{D}') \neq \text{fail})$$

In other words, $\mathcal{D}_g$ is the smallest possible subset of the original slice still reproducing the bug. Computing a global minimal data slice is infeasible in practice, since it requires testing of all $2^{|\mathcal{D}|}$ subsets of $\mathcal{D}$, which has exponential complexity. A less strict condition is the one of the *local minimum data slice* $\mathcal{D}_l$, for which no smaller subset exists that exposes the bug:

$$test(\mathcal{D}_l) = \text{fail} \wedge \forall \mathcal{D}' \subset \mathcal{D}_l \cdot test(\mathcal{D}') \neq \text{fail}$$

However, testing whether $\mathcal{D}_l$ is indeed a local minimum still requires $2^{|\mathcal{D}_l|}$ tests. An approximation to the local minimal slice is an *n-minimal data slice* $\mathcal{D}_n$, which is a slice for which no n elements can be removed without making the bug disappear:

$$test(\mathcal{D}_n) = \text{fail} \wedge \forall \mathcal{D}' \subset \mathcal{D}_n \cdot (|\mathcal{D}_n| - |\mathcal{D}'| \leq n \Rightarrow test(\mathcal{D}') \neq \text{fail})$$

The delta debugging algorithm (Zeller and Hildebrandt 2002), depicted in Figure 5.3, finds a 1-minimal data slice of $\mathcal{D}$, i.e. a slice for which no one element can be removed without making the bug disappear. Notice that even smaller slices might be constructed by removing more than one element. The algorithm works by dividing the data set in n (more or less) equal subsets, and checking if one of them still exposes the bug. If so, the process continues with this subset. If no subset exposes the bug, but a complement of one of the subsets does, the process continues with the complement and increases granularity (such that the subsets in the next step are equally large). Otherwise, the granularity is increased if possible, or the process stops.

```
function DDebug(D) :
    return DDebug(D,2)

function DDebug(D,n) :
    Δ_{i=1}^{n} := Partition(D,n)
    if ∃Δ_i, test(Δ_i) = fail :
        return DDebug(Δ_i, 2)                          – 'Reduce to subset'
    else if ∃Δ_i, test(D − Δ_i) = fail :
        return DDebug(D − Δ_i, max(n − 1, 2))          – 'Reduce to complement'
    else if n < |D| :
        return DDebug(D, min(|D|, 2n))                 – 'Increase granularity'
    else :
        return D                                       – 'Done'
```

Figure 5.3: DDebug: The Delta Debugging algorithm. Finds a 1-minimal subset of $\mathcal{D}$ that causes the bug to appear.

Step	Call		Queries 1	2	3	4	Result
1	DDebug($\{1,2,3,4\}$,2)	Δ_1	•	•			√
		Δ_2			•	•	√
		Increase granularity					
2	DDebug($\{1,2,3,4\}$,4)	Δ_1	•				√
		Δ_2		•			√
		Δ_3			•		√
		Δ_4				•	√
		Δ_1^{-1}		•	•	•	×
		Reduce to complement					
3	DDebug($\{2,3,4\}$,3)	Δ_1		•			√
		Δ_2			•		√
		Δ_3				•	√
		Δ_1^{-1}			•	•	√
		Δ_2^{-1}		•		•	×
		Reduce to complement					
4	DDebug($\{2,4\}$,2)	Δ_1		•			√
		Δ_2				•	√
		Done: $\{2,4\}$ *is 1-minimal*					

Figure 5.4: Example run of the delta debugging algorithm on a trace with 4 queries.

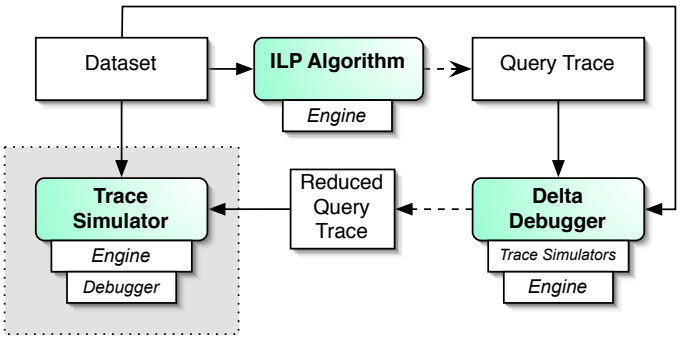


Figure 5.5: Overview of the debugging process.

In our case, the data $\mathcal{D}$ corresponds to a trace, and every Δ_i represents a set of queries. Testing a Δ_i consists of running the trace with queries Δ_i through a trace simulator, and checking the output of the simulator for consistent results. For example, suppose that we have a query trace with 4 queries exposing a bug. Applying the delta debugging algorithm on the set of queries in the trace results in the steps from Figure 5.4. Notice that some tests are repeated: a smart implementation can memorize tests, and reuse their answers.

In the worst case, the DDEBUG algorithm needs to perform $|\mathcal{D}|^2+3|\mathcal{D}|$ tests. However, this worst case seldom occurs in practice. In the optimal case where there is only one element in the slice causing the bug to appear, the number of tests is bound by $2 \cdot log_2(|\mathcal{D}|)$.

We have implemented and used the delta debugging approach in the development of new execution mechanisms in hipP. An overview of the debugging process can be seen in Figure 5.5. The traces generated by the ILP algorithm are fed to the delta debugger, which trims it down to a smaller trace. The resulting trace is then fed into a trace simulator, and the engine (i.e. hipP) can then be manually debugged using the host language debugger (i.e. `gdb`).

We implemented two types of delta debuggers, which differ in the type of test they perform to detect when the execution of a trace exposes a bug. The simplest type of delta debugger is is one that runs a trace through a trace simulator run in a separate hipP engine, and checks whether the process terminates successfully or not. This test can be used for bugs that cause an engine to fail (e.g. due to a segmentation fault). The second type of test compares the trace execution of two engines to check for inconsistent results. First, the queries from the original trace are adorned with extra goals, recording for every query in the trace on which examples it succeeds. The test of the delta debugger then consists of calling hipP and running the resulting trace through both a plain trace simulator (see Figure 5.2) and a simulator with the (buggy) optimization enabled. The resulting logs of both runs are compared, and if they differ, the test fails.

The delta debugger can be set to use different granularities: it can either

Trace	Granularity	Time	Tests	Resulting Trace		
				It[a]	Qu[b]	R[c]
1	Iterations	16.2s	10	1	137	822
	Queries	27.1s	26	1	1	6
	Queries∘Iterations	18.9s	24	1	1	6
	Examples∘Queries	27.6s	29	1	1	1
2	Iterations	78.0s	53	2	181	10942
	Queries	177.3s	157	2	2	236
	Queries∘Iterations	120.0s	136	2	2	236
	Examples∘Queries	180.4s	171	2	2	2
3	Iterations	138.1s	105	3	398	17235
	Queries	360.2s	338	3	3	265
	Queries∘Iterations	226.0s	271	3	3	265
	Examples∘Queries	371.1s	413	3	3	3

[a]Total number of iterations in the trace.
[b]Total number of queries in the trace.
[c]Total number of query runs necessary to reproduce the bug.

Table 5.1: Delta debugger execution time and number of tests performed for different granularities on three traces, together with statistics about the resulting traces. Traces are trimmed to the minimal amount of failing **I**terations, **Q**ueries or **E**xamples. Combinations of these granularities are denoted by ∘.

choose to find failing iterations in a trace, which gives fast results, but also less compact traces; it can prune the trace on the level of the queries themselves, giving a minimal trace; and, it can trim down the number of examples on which every query is run, reducing the number of times a query needs to be called to expose a bug.

Table 5.1 shows the execution time of the delta debugger using different combinations of granularities. For our experiments, we used a trace from a TILDE run on the Mutagenesis data set, with a lookahead setting of 2. The trace consists of 53 iterations of the algorithm, encompassing a total of 12908 queries. This trace was modified to get three variants: the first trace triggers a bug in the last query of the last iteration; the second trace triggers the same bug, yet only if the first query of the first iteration was executed before the query containing the bug; the third trace triggers the same bug whenever the first query and another query from the middle of the trace was executed. For each of these traces, the delta debugger was run using different granularities. Combinations of granularities are denoted by ∘, where $G_1 \circ G_2$ means applying delta debugging with granularity G_1 on the trace resulting from delta debugging with granularity G_2. The delta debugger successfully minimized all three traces to the minimal trace needed to reproduce the bug, being a trace of 1, 2 and 3 queries respectively. The results show that applying the delta debugging first on the level of iterations, and then pruning further on the query level requires less

```
% Measure the number and max. length of queries in a trace
analyze(Trace, NbQueries, MaxLength) :-
    read(Trace, Input),
    ( Input == end_of_file ->
        NbQueries = 0,
        MaxLength = 0
    ;
        Input = query(Query, Examples),
        analyze_query(Query, Length),
        simulate(Trace, NbQueries1, MaxLength1),
        NbQueries is NbQueries1 + 1,
        MaxLength is max(Length, MaxLength1)
    ).

% Measure the length of a query
analyze_query((G,Gs), Length) :- !,
    analyze_query(Gs, Length1),
    Length is Length1 + 1.
analyze_query(G, 0).
```

Figure 5.6: `analyze/3`: A simple query analyzer measuring the total number of queries evaluated (`NbQueries`) and the maximum length of queries (`MaxLength`).

tests than immediately pruning the complete trace on the query level. Pruning on the iteration level gives a first 'rough' version of the trimmed down trace, after which one can decide to prune further on the query level.

5.4 Query Analysis

To be able to get an insight in the behavior of ILP algorithms with respect to query execution, knowing the characteristics of the queries and their execution is particularly interesting. Modifying the ILP system to record all interesting information is a cumbersome job because of the size of the ILP system. In this section, we discuss performing execution analysis using the traces gathered in Section 5.2. Structural query analysis (Section 5.4.1) gives an insight in ILP algorithms by extracting structural properties from the queries generated by the algorithm. Structurally analyzing traces not only provides information on various aspects (such as the average length) of evaluated hypotheses, it can also be used to analyze the structure of transformed queries, query packs, ad-packs, ... and helps getting insight in why certain execution mechanisms perform better or worse than others. Dynamic profiling of queries (Section 5.4.2) provides information on the run-time behavior of query execution mechanisms.

```
statistic(nbqueries, [
    event(query_begin(AIn, AOut), (
        incr_assoc(nbqueries,AIn,AOut)))]).

statistic(maxlength, [
    event(query_begin(AIn, AOut), (
        put_assoc(length,AIn,0,AOut))),
    event(query_goal(Goal,AIn,AOut), (
        incr_assoc(length,AIn,AOut))),
    event(query_end(AIn, AOut), (
        get_assoc(length,AIn,Length),
        get_assoc(maxlength,AIn,MaxLength1),
        MaxLength is max(Length,MaxLength1),
        put_assoc(maxlength,AIn,MaxLength,AOut)))]).
```

Figure 5.7: Event-based definition of the `NbQueries` and `MaxLength` statistics from Figure 5.6.

5.4.1 Structural Query Analysis

Basically, all a structural query analyzer needs to do is scan through every query of the trace, while keeping counters for various statistics throughout the process. For example, measuring both the maximum length and total number of queries in a trace can be done using the simple analyzer from Figure 5.6 (which is a variant of the base simulator from Figure 5.2, page 86). To collect more query-level statistics, more counters have to be added to `analyze_query`, whereas introducing additional global statistics involves extending the top level `analyze` predicate. However, the number of interesting statistics quickly grows, due to which the analyzer quickly becomes hard to maintain. We therefore propose an event-based approach for designing the trace analyzer. We define a set of events in the trace, indicating points of interest in the analysis process. Such events include the start and end of a new query, the occurrence of a single goal in a query, the occurrence of a pack-or in a query pack, ... A particular statistic can then be defined by specifying the actions that have to be taken at certain events. These actions typically consist of simply incrementing or resetting counters. Figure 5.7 shows the definition of the 2 statistics measured in Figure 5.6 as a combination of event actions. The actions are always passed a current state of counters `AIn` (here represented as an associative list), and return a new state `AOut`. Certain events can also be passed extra arguments, such as is the case with `query_goal`, which is passed the actual goal to be analyzed. The actual query analyzer now has to collect all the defined statistics, scan through the trace, and at every event execute all the necessary code, and finally return the values of the requested statistics. For example, collecting both statistics from Figure 5.7 from the trace of Figure 5.1 can be done as follows:

# Goals	17149
# Queries	7613
Depth	7
Min. Goals per Query	3
Avg. Goals per Query	7.83
Max. Goals per Query	11
Min. Branching Factor	2
Avg. Branching Factor	42.37
Max. Branching Factor	91

Table 5.2: Structural analysis results for a specific query pack

```
?- analyze_trace('example.trace',[nbqueries,maxlength],Result).
Result = [4,3]
```

We omit the code of `analyze_trace/4`. The analysis can be extended further by not only keeping simple counters, but by also keeping statistics for every iteration. For example, a list of all query pack analyses can be maintained throughout the analysis process, such that reports can be made on a per-pack basis. A report for a separate query pack looks like Table 5.2.

By defining statistics in terms of events in the trace analysis process, the code of the analyzer itself can remain simple, and does not need to be modified whenever we want to measure additional statistics of the trace.

5.4.2 Dynamic Query Profiling

Run-time information of query execution can be used for a variety of purposes: predicting or explaining behavior of execution mechanisms on a data set, comparing different execution mechanisms, detecting bottlenecks in query execution of data sets in order to identify targets for further optimization, ...

Trace-based dynamic analysis essentially works by executing all queries from the trace using the simulator from Figure 5.2, except that the query is first instrumented with profiling calls before it is executed. One interesting dynamic aspect to monitor is the set of inputs of Byrd's 'Box Model' (Byrd 1980) for every goal of the query. This measures how many times a goal was called or was backtracked to. This information can be useful in determining how well execution mechanisms that try to avoid redundant backtracking (such as the once transformation from Section 2.3.1 or adpacks from Chapter 3) actually perform. Measuring this can be done by embedding all goal calls in a predicate `profile_call/2` (Figure 5.8) that calls the query goal and increment the necessary counters. For example, the query

$$\texttt{atom(X, 'c')}, \texttt{atom(Y, 'o')}, \texttt{bond(X, Y)}.$$

```
profile_call(Goal) :-
    increment_call
    assert(in_call),
    call(Goal),
    ( in_call ->
        retract(in_call)
    ;
        increment_redo
    ).
profile_call(_) :-
    retract(in_call),
    fail.
```

Figure 5.8: `profile_call/2`: Measure the number of calls and redos in a query goal.

	QPacks	ADPacks
Examples	13	
Min. Call	59452	19296
Avg. Call	447692.23	61240.77
Max. Call	1469595	137569
Tot. Call	5819999	796130
Min. Redo	1205	545
Avg. Redo	7933.31	2411.54
Max. Redo	24084	6348
Tot. Redo	103133	31350

Table 5.3: Collected statistics for a query pack

is transformed into the following query:

$$\texttt{profile\_call}(\texttt{atom}(\texttt{X}, \texttt{'c'})), \texttt{profile\_call}(\texttt{atom}(\texttt{Y}, \texttt{'o'})), \texttt{profile\_call}(\texttt{bond}(\texttt{X}, \texttt{Y})).$$

These results can be collected for different variations of queries (or sets of queries) to compare different execution mechanisms. For example, comparing the query packs and the adpacks approach this way results in Table 5.3. This table shows that the goal of adpacks, being to reduce the number of calls, is indeed reached.

5.4.3 Implementation

An overview of the implementation of both the structural analyzer and the dynamic profiler can be seen in Figure 5.9. The structural analyzer works solely on the query trace, whereas the dynamic profiler needs the dataset to evaluate the

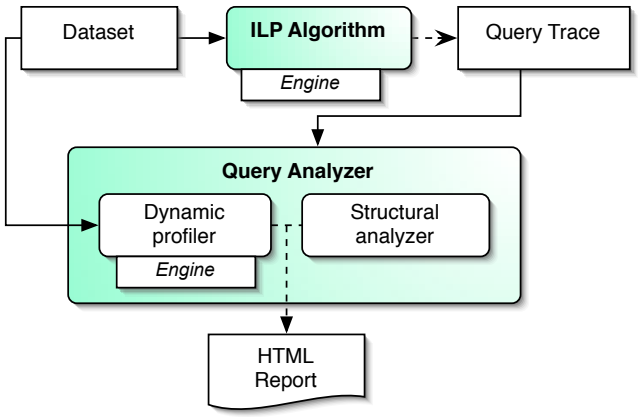


Figure 5.9: Overview of the query analyzer.

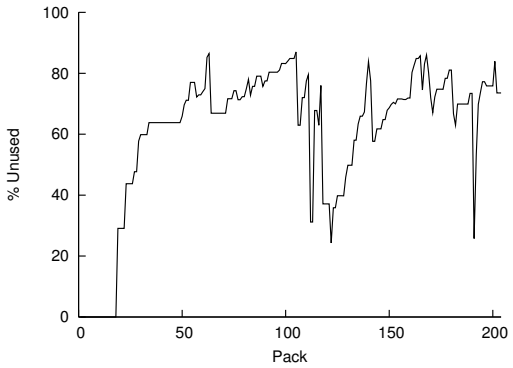


Figure 5.10: Percentage of unreached goals.

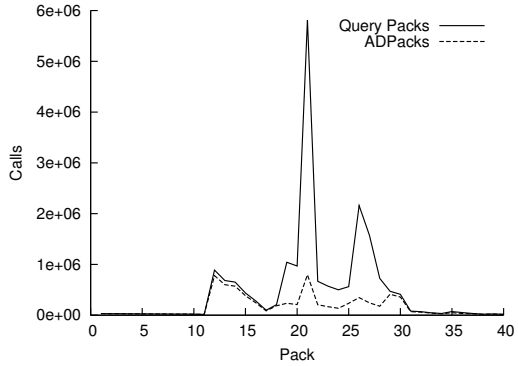


Figure 5.11: Total number of calls to goals in a pack for Query Packs and ADPacks.

(annotated) queries from the trace on using the engine. For both analyses, the queries are first transformed into different execution variants (once transformed queries, query packs, adpacks), and every variant is analyzed both structurally and dynamically. The resulting information from both analyses are combined into an HTML report that gives a high-level overview of the ILP algorithm run, comparing different execution mechanisms, and illustrating the behavior of the queries over the complete run. Besides tables of measured statistics, the results are also plotted out like in Figure 5.11. This figure illustrates the decrease in calls over the whole trace that adpacks provide over query packs. From the global overview, one can get more detailed information about certain packs in the trace. For example, the 21^{st} pack from Figure 5.11 shows a peak in the trace, which can be analyzed closer by requesting the statistics for this individual pack.

As was mentioned before, the dynamic analysis can also be used to detect bottlenecks in execution. Figure 5.10 illustrates for the HIV data set the percentage of goals of a query pack that are never reached. For the HIV data set, this runs up to 85% of a pack, which means that the largest part of this query pack is generated and compiled in vain. This information is useful for the ILP experts for tweaking the language bias of the data set (which controls the query generation process) to yield more relevant queries.

5.5 Conclusions

In this chapter, we illustrated a trace based approach to debugging and analyzing query execution mechanisms for ILP algorithms. Using a trace based approach yields several advantages in this context:

- The specific workings of the ILP algorithm do not have to be known, as the traces are algorithm independent, yet provide enough information for

performing a perfect simulation of the query execution of the algorithm itself.

- By altering the traces (e.g. reducing the number and size of queries), execution can be 'shortcut', allowing bugs to be exposed very fast without having to go through the whole trace. Reducing the trace to a minimal subset that still reproduces the bug can be done automatically through the use of delta debugging.
- With trace-based execution, time is only spent on the execution of queries. Therefore, a complex query generation phase of an ILP algorithm does not affect the total execution time of a trace, and so debugging can be done faster.
- It is not necessary to have full knowledge of the code base of the ILP system, which can in practice become very large.

Besides being useful in debugging, the query traces can also be used for analyzing and comparing query execution mechanisms. Performing a structural analysis of the trace provides useful information on the number and structure of the queries executed by a run of an ILP algorithm. The dynamic profiling uses a source-level transformation on the trace to collect run-time statistics of the execution. While this approach is flexible and easy to implement, it introduces significant run-time overhead.

We have used the analysis and debugging techniques described in this chapter extensively during the development of the techniques of the other chapters in this text. The delta debugging approach made it possible to rapidly trace bugs in the low level implementation of adpacks (Chapter 3) and control flow compilation techniques (Chapter 4). The analysis techniques also made it possible to study the efficiency of tabling techniques compared to the query pack execution mechanism, as described in Chapter 6.

One application of ILP query traces that was not explored in this chapter is visualization. Visualizations of query execution help in understanding what happens in the query execution step of an algorithm on a concrete data set. Moreover, this approach can also be applied to visualization of ILP algorithms themselves (instead of just their query execution) by extending the traces to contain information concerning the other steps from Figure 2.3. Adding even more algorithm-dependent data can serve in making visualizations for specific algorithms. The fact that a trace-based visualizer does not need any changes to the ILP system itself is even more interesting here, because visualizations typically need to call back to the system to control the execution of algorithms. The other advantages of traces apply here as well: time-consuming steps can be avoided, and visualizations are implementation-independent.

In (Jahier and Ducassé 2002), a general trace-based monitor is proposed for program analysis. This monitor is built around a functional `fold`-like predicate which accumulates information based on events generated by the program,

which is similar to the event-based approach we use in our structural analysis. Their approach does not use explicit traces, but interleaves execution of the monitor with the execution of the program. Modifying our approach to perform analyses online is also possible, but the advantage of not spending time in the algorithm itself would be lost. Similarly, (Ducassé 1999b) proposes an on-line approach for debugging using tracers. In the context of debugging ILP query execution, this approach avoids the need to trace through the ILP system itself while debugging. However, besides the longer execution times and the possibility of the bug not occurring when the algorithms are non-deterministic, automated debugging by slicing the trace is no longer possible.

Chapter 6

Trading Space for Time

6.1 Introduction

Because the candidate hypotheses generated by an ILP algorithm are refinements of queries from previous iterations, the queries that need to be executed on the dataset show a lot of similarity. Techniques such as query packs (Section 2.3.2) and adpacks (Chapter 3) avoid redundancy in the execution process by exploiting these similarities in queries. These techniques change the way (sets of) queries are executed, and as such avoid executing the same goals multiple times. A different approach to avoiding redundancy is to store computation results in memory. By remembering answers to single goals, parts of queries, or even complete queries, repeated execution can be reduced to fetching and applying the previously computed answers from memory. However, the catch here is that to be able to reuse answers computed in a previous step, one needs space to store these answers in memory.

In this chapter, we discuss the possibilities, feasibility, advantages, and disadvantages of remembering results of computations at different levels of the execution of ILP algorithms. Section 6.2 introduces program specialization and tabled execution, techniques used in the other sections of this chapter. Section 6.3 focuses on avoiding recomputation during the execution of predicates from the background knowledge. Section 6.4 discusses remembering results of queries and parts of queries. In Section 6.5, we look at some specific ILP algorithms, and try to avoid recomputation by taking advantage of some of their properties in combination with memory usage. Finally, we present our conclusions in Section 6.6. An overview of the techniques introduced in this chapter is shown in Table 6.13 (page 124).

6.2 Background: Specialization & Tabling

6.2.1 Top-Down & Bottom-Up Specialization

The idea of *specialization* is the following: given a program P with inputs S and D. A specialization of P with respect to S is a program P_S such that, for all input D:

$$P_S(D) = P(S, D)$$

Input S is called *static* input (known at specialization time), and D is called *dynamic* input (unknown at specialization time).

Consider for example a predicate `pow/3`, for which `pow(X,N,Y)` computes $\texttt{X}^{\texttt{N}}$ and unifies the result with `Y`. The definition of this predicate is:

```
pow(X,0,Y) :- !, Y = 1.
pow(X,N,Y) :-
    N1 is N - 1,
    pow(X,N1,Y1),
    Y is Y1 * X.
```

Specializing this program for a known input $\texttt{N} = 3$ eliminates the recursive call in the predicate, resulting in

```
pow_3(X,Y) :- Y is X*X*X.
```

By factoring out certain computations from the program (e.g. tests, recursive calls, ...) and replacing computations by their result, specialization generally increases the speed of the original program. While the resulting program size is in certain cases smaller than the size of the original program, specialization might have the exact opposite effect as well (e.g. when unfolding many loops). This is where the tradeoff in specializing programs lies.

Automated specialization (also called *partial evaluation*) is a well known optimization technique (Jones, Gomard, and Sestoft 1994), for which a variety of approaches have been developed within programming paradigms such as imperative (Andersen 1993), object-oriented (Dean, Chambers, and Grove 1995), and especially in the context of functional (Weise, Conybeare, Ruf, and Seligman 1991) and logic programming (Lloyd and Shepherdson 1991; Leuschel and Bruynooghe 2002). Most specialization techniques for logic programming are based on the *top-down* evaluation mechanism (Gallagher 1993), where the static input is a partially instantiated query. The information from the partially instantiated query is used to eliminate certain computations during the specialization process. The result of this specialization process is then recorded in the specialized program, which can then be evaluated using instances of the initial query. While top-down specialization is driven by (instances of) particular queries, *bottom-up specialization* (Vanhoof, De Schreye, and Martens 1999; Leuschel and Schreye 1996) provides a query-independent approach to program specialization. Instead of propagating data provided in a query downwards in

Example 1
father(chris,dirk)
mother(bea,chris)
father(andrew,bea)
mother(anneleen,bea)
greatgrandparent(andrew,dirk)
greatgrandparent(anneleen,dirk)

Example 2
mother(caroline,dina)
father(bob,caroline)
father(andre,bob)
mother(ann,bob)
greatgrandparent(andre,dina)
greatgrandparent(ann,dina)

Background Knowledge
grandfather(X,Y) :- father(X,Z), father(Z,Y).
grandfather(X,Y) :- father(X,Z), mother(Z,Y).
grandmother(X,Y) :- mother(X,Z), father(Z,Y).
grandmother(X,Y) :- mother(X,Z), mother(Z,Y).

Figure 6.1: A data set describing family relationships.

```
greatgrandparent(X,Y) :- grandfather(X,Z), father(Z,Y)
greatgrandparent(X,Y) :- grandfather(X,Z), mother(Z,Y)
greatgrandparent(X,Y) :- grandmother(X,Z), father(Z,Y)
greatgrandparent(X,Y) :- grandmother(X,Z), mother(Z,Y)
```

Figure 6.2: Definition of `greatgrandparent/2`.

a program, this technique propagates information from facts and predicates in the program upwards, thus precomputing as much relations as possible.

6.2.2 Tabling

Tabled execution is an alternative for classical execution that remembers and reuses answers to goal calls. When a tabled goal is called, execution first checks whether answers for that goal have already been computed before. If this is the case, these answers are returned to the caller, and the goal in question is not executed. If the goal has not been called before, it is executed, and the resulting answers are stored in the goal table.

Tabled execution was originally developed in the XSB system (Warren et al. 2005). Since then, other implementations have emerged, such as YapTab (Rocha, Silva, and Costa 2000) for the YAP system (Damas and Costa 2003). Tabled execution has been successfully applied in various domains such as program analysis, model checking, and parsing.

6.3 Memorizing Background Knowledge

6.3.1 Motivation

Consider the example data set from Figure 6.1, describing family relationships between people. The data set contains `father/2`, `mother/2`, and `greatgrandparent/2` facts, describing *'... is father/mother/great grandparent of ...'* relationships for two families. The background knowledge consists of the definition of the *'... is grandfather/grandmother of ...'* relations, coded in the `grandfather/2` and `grandmother/2` predicates. Suppose that we want our ILP algorithm to learn the definition of the predicate `greatgrandparent/2` in terms of the predicates `father/2`, `mother/2`, `grandfather/2`, and `grandmother/2`. A possible solution is the one seen in Figure 6.2.

During the evaluation phase of the ILP algorithm, the predicates `grandfather/2` and `grandmother/2` are called repeatedly if candidate hypotheses contain these relationships (which they should at some point in time if we want to reach the definition from Figure 6.2). However, because the answer to this predicate remains the same for a given example over the whole learning run, the same computations will be performed over and over again. For the toy example described above, the amount of overhead in calling the background predicates every time is not so big (only 2 calls to predicates in the data set itself). However, background knowledge predicates can become quite complex in practical ILP settings. For example, the background knowledge of the Mutagenesis data set (Figure A.2, page 134) contains definitions of higher-level molecular structures such as `phenanthrene/1`. Calling these predicates is very expensive.

The aim of this section is to study techniques for reducing the time spent in background knowledge by remembering their answers. The widely used ad-hoc approach described in Section 6.3.2 and the specialization approaches from Section 6.3.3 take a data set as input, and transform it into a more efficient version. By using this more efficient data set, the ILP algorithm performs better during all of its runs. This is schematically depicted in Figure 6.3, (which is a modification of the standard ILP process overview from Figure 2.4 on page 13). The technique of Section 6.3.4, on the other hand, leaves the original data set unaltered, and dynamically remembers answers to predicates in the background knowledge at run-time.

6.3.2 Manual precomputation

In practice, whenever a data set contains complex background knowledge predicates, the complex predicates are manually identified, all its answers are precomputed for every example, and the precomputed answers are added to the example. Queries execute faster against the resulting data set, because the answers to the complex background predicates do not need to be computed any longer. For example, consider the background predicates `grandfather/2` and `grandmother/2` from Figure 6.1. Both predicates have exactly one answer for

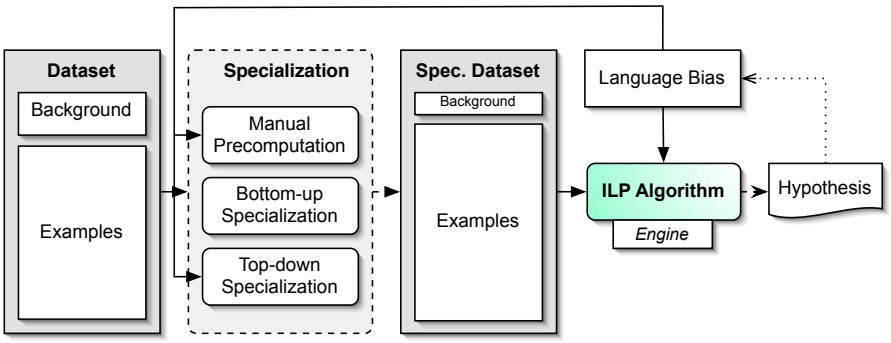


Figure 6.3: Transforming a data set to a specialized, more efficient version.

Example 1
father(chris,dirk)
mother(bea,chris)
father(andrew,bea)
mother(anneleen,bea)
greatgrandparent(andrew,dirk)
greatgrandparent(anneleen,dirk)
grandfather(andrew,chris)
grandmother(anneleen,chris)

Example 2
mother(caroline,dina)
father(bob,caroline)
father(andre,bob)
mother(ann,bob)
greatgrandparent(andre,dina)
greatgrandparent(ann,dina)
grandfather(andre,caroline)
grandmother(ann,caroline)

Figure 6.4: Dataset from Figure 6.1 with precomputed answers for `grandfather/2` and `grandmother/2`.

Example
p(a).
p(b).
q(b).

Background knowledge
r(X) :- p(X), !, q(X).

Figure 6.5: A data set with background knowledge containing cuts.

every example. Adding the answers as facts to the examples results in the new data set from Figure 6.4. Executing queries against this data set avoids the need to compute the answers to the background predicates for every call.

This ad-hoc way of selecting predicates and adding their precomputed answers to the data set is very simple and effective. However, when background predicates contain impure constructs such as cuts, simply computing all answers top-down is unsafe and yields potentially incorrect results. The data set from Figure 6.5 has such a background predicate with a cut. A top-down search for solutions of the query `r(X)` yields no result, and so no facts are precomputed for `r(X)`. However, the query `r`(b) succeeds on the original data set, whereas it fails on a data set with the answers of `r(X)` precomputed. Taking the cut into account requires dynamically checking the call pattern, and creating a specialized version for the different possible call patterns of a predicate. For the above example, this would result in the following specialized version of `r/1`:

```
r(X) :- nonvar(X), X = b.
```

6.3.3 Predicate Specialization

The goal of specializing background knowledge is the same as the precomputation from Section 6.3.2: for every example in the data set, create new versions of background knowledge predicates, where parts of the execution of the predicates are precomputed. However, instead of simply computing all answers of background knowledge predicates, we apply specialization techniques on the background knowledge. Specializing the data set is done by treating each example in the data set as a separate logic program, and applying specialization on the combination of the example and the background knowledge. The result is a new example (logic program) containing the data from the original example, and specialized versions of background knowledge predicates specific to the original example.

Because we want to create a specialized version of the data set, independent of the ILP algorithm that will be used and the queries that will be executed against the data set, bottom-up (i.e. query-independent) specialization is the most promising specialization flavor to use for specializing the data set. Bottom-up specialization is achieved by using a specific bottom-up semantics for logic programs to infer facts from already inferred facts. For specialization to be usable in a practical ILP setting, the bottom-up semantics used needs to be

Answers[a]	Experiment	Exec.[b]	Size[c]
1	Original	0.21	819
	Bottom-up	7.75	4433
	Top-down	0.03	2750
	Manual precomputation	0.01	877
2	Original	32.73	1837
	Bottom-up	4156.20	13948
	Top-down	×	×
	Manual precomputation	0.01	1957

[a]Number of answers to the query `benzene(X)`.
[b]Execution time of the query (`benzene(X)`,`fail`) (in ms).
[c]Total size of the (specialized) example (in bytes).

Table 6.1: Performance and size of two examples from the Mutagenesis data set after specializing `benzene/1`.

able to take into account all procedural aspects of Prolog, most importantly the cut. Several semantics have been proposed that take into account a specific procedural aspect of Prolog (Proietti and Pettorossi 1991; Debray and Mishra 1988; Spoto 2000). The semantics from (Vanhoof, Tronçon, and Bruynooghe 2003) supports a combination of the procedural aspects of Prolog, including support for cuts and preservation of order and multiplicity of predicates.

To estimate the potential benefit from applying bottom-up specialization on background knowledge predicates, we integrated the prototype of the semantics from (Vanhoof, Tronçon, and Bruynooghe 2003) with the bottom-up specialization framework from (Vanhoof, De Schreye, and Martens 1999), which initially only supported definite programs. The resulting bottom-up specializer was used to specialize the `benzene/1` predicate from the Mutagenesis data set (Appendix A.1). A partial definition of `benzene/1` is given in Figure A.2 (page 134). Two examples with varying complexity from this data set were selected as the target of the specialization: the first example consists of 6 `atom/4` facts, 6 `bond/3` facts, and has exactly one answer for the query `benzene(X)`; the second example contains 15 `atom/4` facts, 16 `bond/3` facts, and returns two answers to the query `benzene(X)`. Both examples were specialized using the bottom-up specializer. We compare the resulting specialized data set with a manually preprocessed data set where all answers to `benzene/2` were collected and added to the data set. Additionally, we compare both approaches with a top-down specialized version of the examples with respect to the call `benzene(X)`. Because of the presence of cuts, we used the Mixtus (Sahlin 1993) specializer to specialize the examples, as it supports full Prolog.

Table 6.1 reports on the performance of queries on the original data set, and compares it with the performance of a manually precomputed version, a bottom-up specialized version, and a top-down specialized version of the data set. The performance of an example is tested by querying all solutions of the predicate `benzene(X)`. The total size of the resulting specialized example is in-

dicated as well. In the experiments, the bottom-up specializer generates code that performs worse than the original example. The reason for this is that the number of clauses explodes by propagating facts upwards, as the code size increase illustrates. The top-down specializer performs better on the simple example, but runs out of memory for the second example. The reason here is that not all tests can be removed at specialization time, and the number of tests also explodes with increasing complexity of the example. Overall, the resulting example after manually precomputing answers to `benzene(X)` performs best.

6.3.4 Predicate Tabling

Precomputing answers for queries beforehand suffers from the fact that the queries run on the data set by an ILP algorithm are not known beforehand. In the ad-hoc approach taken in practice (Section 6.3.2), one assumes that precomputing the answers to the most general call to the background predicates is enough. The problem with this is that answers to predicates that are never called are stored in memory, and that these answers can potentially be incorrect in the presence of cuts. Taking the cuts into account requires hand-crafting a specialized version of a predicate for each possible call pattern. Applying specializers that support cuts (Section 6.3.3) automate this data set preprocessing step while preserving correctness, but introduces an explosion of derived answers to predicates. Instead of precomputing answers of predicates and adding them to the data set in advance, one can simply table the complex background predicates during execution. This approach ensures that only useful answers are stored and that the answers are correct (assuming that the tabling mechanism supports cuts), at the cost of a dynamic overhead for every run of the algorithm (as opposed to the one-time cost of transforming a data set prior to using it).

To compare the performance of tabling with the ad-hoc approach from Section 6.3.2, we set up an experiment to run Tilde on the Mutagenesis data set. Because hipP does not have support for tabling, we used YAP in the experiment. On the other hand, an implementation for the Tilde algorithm is only available in the ACE/hipP system. Therefore, we recorded query traces (Chapter 5) of Tilde runs with various lookahead using the ACE/hipP system, and ran these traces through a trace simulator written in YAP. During the execution of the trace on the Mutagenesis data set, all calls to background predicates occurring in the queries were tabled. The timings of Table 6.2 compare the total time of executing all queries from the Tilde traces on the original data set, with the background predicates tabled and untabled respectively. Additionally, the execution time for running the trace on the data set with manually precomputed answers of the background predicates is also shown. The time needed to precompute all answers for the background knowledge predicates present in the language bias (`benzene/1`, `phenanthrene/1`, ...) is also included in the results. Untabled execution is clearly infeasible: the execution time explodes with more complex queries, causing the experiment with lookahead setting 2 to last more than a week. For tabled execution, the experiment with lookahead 1 is faster

Experiment	Preproc.[a]	Execution[b]		
		LA0	LA1	LA2
No optimization	-	10740	35640	×[c]
Tabled execution	-	164	161	277
Manual precomputation	683	1	1	21

[a]Time needed for precomputing all answers to the complex background knowledge predicates (in seconds)
[b]Query execution time (in seconds)
[c]Experiment was canceled after 1 week.

Table 6.2: TILDE execution times for tabled execution on the Mutagenesis dataset.

than the experiment with no lookahead. This can be explained by the fact that less solutions of background predicates are required to find an optimal query. However, TILDE still performs significantly better on the manually precompiled version of the data set than on the original one with tabled execution.

6.3.5 Conclusions

Different approaches can be used to avoid redundant execution of complex background predicates during the evaluation of queries. The approach commonly used in practice is to identify these predicates manually, precomputing all their answers for every example, and adding these example-specific answers to the data set. However, this approach is not robust with respect to the occurrence of cuts in background knowledge. A more general approach consists of applying specialization techniques on the data set as a whole. Unfortunately, although these approaches support the use of cuts, both the query-independent bottom-up specialization technique as the query-directed top-down technique do not scale well with increasing complexity, making them unusable. Another approach is leaving the data set unaltered, but tabling the execution of background predicates. The advantage of this approach is that it supports cuts, and that only the necessary answers are stored in memory. The downside is that there is a run-time overhead of computing the answers, whereas in the other approaches, these answers are precomputed prior to executing ILP algorithms.

6.4 Tabling Conjunctions

6.4.1 Introduction

Queries in ILP algorithms generally consist of a prefix and a refinement part. The prefix is a query from a previous iteration that was selected by the algorithm for further extension, and the refinement itself is a new conjunction. The set of evaluated queries is therefore partitioned into sets of queries with an identical prefix. This means that during the execution of these queries over an example,

the same answers for the prefix are computed over and over again. To avoid this recomputation, it makes sense to remember the computed answers to prefixes, and reuse them when subsequent queries with the same prefix are executed. We call this approach *prefix tabling*. Because the answers stored by prefix tabling are only used within the same iteration of the ILP algorithm, they can be removed from memory in the next iteration.

Another consequence of the incremental nature described above is that, since the prefix of a query is itself a query from the previous iterations, each prefix has been executed before (and either yielded an answer or failed). Remembering the results of queries can therefore avoid execution of prefixes in the next iterations. We call this approach *query tabling*. Because the answers to queries depend on answers from previous iterations of the ILP algorithm, all answers from the previous iterations need to be stored in memory as long as there are queries depending on them. Therefore, contrary to prefix tabling, the extra memory used for remembering the answers cannot be freed after every iteration.

Recall that query packs (Section 2.3.2) also exploit the incremental nature of ILP query execution by sharing execution of common prefixes across queries. The main advantage of query pack execution over storing answers to queries or prefixes is that query packs do not introduce extra memory overhead for every example in the data set. The downside is that answers to queries from previous iterations are not remembered, and that the answers to prefixes have to be recomputed at every iteration.

In this section, we discuss several approaches to use tabling to store and reuse answers to queries and their prefixes. The prefix and query tabling approaches are also presented in (Rocha, Fonseca, and Costa 2005). We make a qualitative evaluation of the potential benefits of these approaches, and compare them to query pack execution.

6.4.2 Motivating Example

We illustrate the advantages of prefix tabling, query tabling, and query packs using the example from Figure 6.6. This example shows the queries executed in 3 iterations of an ILP algorithm, and two data set examples on which the queries are executed. Table 6.3 shows for every query in every iteration how many goals were called or backtracked to.

First, let us consider the execution of the queries on the first example. Executing the prefix of the first query of the second iteration (i.e. query 3), which is a refinement of a query from the first iteration, results in a call and a redo for `a`, and 2 calls to `b` before reaching the solution {`X`=2,`Y`=1}. The remainder of the query execution consists of a call and a redo for `c` and 2 calls for `d`, resulting in a final total of 6 calls and 2 redos for the first query (indicated by the 8 in the column for query 3 in Table 6.3). However, in the second query, the same prefix is executed. By remembering the solution to the prefix, execution of the calls to `a` and `b` can be replaced by fetching the solution for the prefix from

Iteration 1	
1	a(X), b(X,Y)
2	m(X), n(X,X)

Iteration 2	
3	(a(X),b(X,Y)), c(Y,Z),d(Z)
4	(a(X),b(X,Y)), c(Y,Z),e(Z)

Iteration 3	
5	(a(X),b(X,Y),c(Y,Z),d(Z)), f(Z)
6	(a(X),b(X,Y),c(Y,Z),e(Z)), g(Z)

Example 1		
a(1).	c(1,1).	e(2).
a(2).	c(1,2).	f(2).
b(2,1).	d(2).	g(2).

Example 2			
a(1).	c(1,1).	d(3).	f(15).
b(1,1).	c(1,2).	d(15).	g(15).
	⋮	e(3).	
	c(1,15).	e(15).	

Figure 6.6: Example query trace.

Example	Experiment	Query								
		1	**2**	**1+2**	**3**	**4**	**3+4**	**5**	**6**	**5+6**
1	No Optimization	4	1	5	8	8	16	9	9	18
	Prefix Tabling	4	1	5	8	4	12	9	9	18
	Query Tabling	4	1	5	4	4	8	1	1	2
	Query Packs			5			10			12
2	No Optimization	2	1	3	8	8	16	34	34	68
	Prefix Tabling	2	1	3	8	6	14	34	34	68
	Query Tabling	2	1	3	6	6	12	26	26	52
	Query Packs			3			11			51

Table 6.3: Number of calls (including backtracking) for different execution mechanisms. For every query from Figure 6.6, its column indicates the total number of calls and redos for that query. The combined total of calls and redos for all queries in an iteration is given in the last column of each block.

memory, and the only calls remaining are the call and redo to `c` and both calls to `e`, resulting in a total of 3 calls and one redo for the second query (instead of 6 calls and 2 redos without reusing the answers from the prefix). Additionally, remembering the answers to the queries in the first iteration avoids execution of the prefixes in the second iteration altogether, resulting in a total of 6 calls and 2 redos for the second iteration (instead of 9 calls and 3 redos when only the answers for the prefix are remembered).

The third iteration from Figure 6.6 consists of refinements of both queries from the previous iteration. Consequently, the prefixes of the queries are not identical, and prefix tabling will therefore not be able to reuse previously computed solutions. Remembering the answers to the full queries from the previous iteration remedies this, and saves execution of every prefixes.

Transforming the queries from the second iteration of Figure 6.6 in a query pack results in the following pack:

```
?- a(X),b(X,Y),c(Y,Z),
        (   d(Z)
        ;p e(Z)).
```

When executing this pack on the first example, the prefix of the queries is executed only once over the whole iteration, as is the case with prefix tabling. In the third iteration, the following query pack is executed:

```
?- a(X),b(X,Y),c(Y,Z),
        (   d(Z),f(Z)
        ;p e(Z),g(Z)).
```

In this case, query pack execution outperforms prefix tabling: because both prefixes are not identical, prefix tabling is unable to reuse any computed answer, and both prefixes are executed separately for every query; however, the execution of `a`, `b` and `c` is shared in the query pack, leading to fewer calls for the third iteration.

Although query tabling outperforms query packs on the first example, this is not always the case. This is illustrated by the results of running the queries on the second example from Figure 6.6. In the second iteration, query pack execution has the advantage that the execution of `c` in the refinement is shared between both queries, whereas it is executed separately with the query tabling approach. The same goes for the prefix of the queries in the third iteration, where the execution of `a`, `b`, and `c` is shared by query pack execution, whereas with query tabling only the answers to (`a(X),b(X,Y)`) are reused during the execution of both queries.

One can easily see that query pack execution always performs at least as good as prefix tabling when it comes to the total number of calls to goals: both approaches execute identical prefixes only once, but query packs can additionally

exploit the similarity of non-identical prefixes. For query tabling and query packs, none of the approaches always outperforms the other. Remembering the answers of queries across iterations can avoid executing prefixes, whereas query packs always need to execute the prefix at least once. However, by exploiting similarity in refinements and prefixes, query packs can also provide an advantage over query tabling in the case where queries are refined with more than one literal at a time (i.e. with lookahead enabled).

6.4.3 Prefix and Query Tabling

In this section, we describe the approach taken to perform both prefix tabling and query tabling. This approach conforms to the approach taken in (Rocha, Fonseca, and Costa 2005), only without tabling the calls to separate goals themselves (which is described and analyzed separately in Section 6.3.4). The tabling of separate goals is left out because it is independent of the actual execution mechanism of queries (i.e. query tabling or query packs).

Queries which are to be evaluated are of the following form:

$$?- \ \mathit{Prefix}(\overline{x}), \mathit{Refinement}(\overline{y}) \tag{6.1}$$

where *Prefix* and *Refinement* are both conjunctions, and $\overline{x}$ and $\overline{y}$ are the sets of variables occurring in them respectively. In the first iteration of the ILP algorithm, *Prefix* is always empty (`true`); in the next iterations, *Prefix* is a query from the previous iteration, and *Refinement* is either a goal or a conjunction of goals, added to the query in the current iteration.

For every query of the form (6.1) to be evaluated, do the following:

1. If no such predicate has been created yet, create a predicate

$$\mathtt{P}_i(\overline{x}) \ :- \ \mathit{Prefix}(\overline{x}). \tag{6.2}$$

   (where i is a unique identifier for *Prefix*), and table the answers for $\mathtt{P}_i(\overline{x})$.

2. Transform the query into

$$?- \ \mathtt{P}_i(\overline{x}), \mathit{Refinement}(\overline{y}) \tag{6.3}$$

3. Evaluate the transformed query.

For example, suppose the query

$$?- \ \mathtt{a(X,Y)}, \mathtt{b(Y,Z)}$$

is refined into the following set of queries:

$$?- \ (\mathtt{a(X,Y)}, \mathtt{b(Y,Z)}), \mathtt{c(Z,U)}.$$
$$?- \ (\mathtt{a(X,Y)}, \mathtt{b(Y,Z)}), \mathtt{d(Z,U)}, \mathtt{e(U)}.$$

(the prefix is put between brackets for notational purposes). The tabling mechanism transforms these queries into

$$?- \ \mathtt{P_{a\_b}(X,Y,Z),c(Z,U)} \tag{6.4}$$
$$?- \ \mathtt{P_{a\_b}(X,Y,Z),d(Z,U),e(U)} \tag{6.5}$$

and creates a new tabled predicate

$$\mathtt{P_{a\_b}(X,Y,Z)} \ :- \ \mathtt{a(X,Y),b(Y,Z)}.$$

Executing the transformed version of these queries re-uses previously computed answers of the prefix, thus avoiding redundancy in execution. After having evaluated the queries, the newly created predicates and their tables can be cleared, thus keeping the extra memory usage local to the iteration. We call this approach *prefix tabling*.

Prefix tabling can be extended further to yield full *query tabling*. This is achieved by transforming the query (6.3) further into

$$?- \ \mathtt{P}_j(\overline{x} \cup \overline{y})$$

where $\mathtt{P}_j(\overline{x} \cup \overline{y})$ is a new tabled predicate, defined as follows:

$$\mathtt{P}_j(\overline{x} \cup \overline{y}) \ :- \ \mathtt{P}_i(\overline{x}), \mathit{Refinement}(\overline{y}) \tag{6.6}$$

Due to the prefix transformation in the next iteration, one of the tabled full queries is re-used when executing the prefix of the refined queries. For example, the transformed queries (6.4) and (6.5) are transformed further into

$$?- \ \mathtt{P_{a\_b\_c}(X,Y,Z,U)}$$
$$?- \ \mathtt{P_{a\_b\_d\_e}(X,Y,Z,U)}$$

with the new tabled predicates

$$\mathtt{P_{a\_b\_c}(X,Y,Z,U)} \ :- \ \mathtt{P_{a\_b}(X,Y,Z),c(Z,U)}.$$
$$\mathtt{P_{a\_b\_d\_e}(X,Y,Z,U)} \ :- \ \mathtt{P_{a\_b}(X,Y,Z),d(Z,U),e(U)}.$$

Suppose the first query is refined in the next iteration into

$$?- \ (\mathtt{P_{a\_b}(X,Y,Z),c(Z,U)}),\mathtt{d(U,V)}$$

The prefix transformation transforms this query into

$$?- \ \mathtt{P_{a\_b\_c}(X,Y,Z,U),d(U,V)}$$

which is exactly the tabled predicate created for the full query in the previous iteration.

Iteration 1	
1	a(X), b(X,Y)

Iteration 2	
2	(a(X),b(X,Y)), c(Y).

Example 1		
a(1).	b(2,1).	c(1).
a(2).	b(2,2).	c(2).

Example 2		
a(1).	b(2,2).	c(3).
a(2).	b(2,3).	

Figure 6.7: Example query trace.

6.4.4 Once Tabling

We observed that during the execution of queries, many times only the first solution for a query is used when executing the refinement of the query in a later iteration. It is therefore worth investigating the performance of a weaker alternative for the query tabling approach from Section 6.4.3, where the weaker version only stores one answer for every succeeded query. We call this approach *once tabling*.

We start by illustrating the intuition behind once tabling using the example from Figure 6.7. After the first iteration of Figure 6.7 finished, the query succeeded with answers {X=2,Y=1} and {X=2,Y=2} for the two examples respectively. Instead of executing the query from the second iteration on the first example, we first transform the query to

$$?- \ (\mathtt{X} = 2, \mathtt{Y} = 1 \ ; \ \mathtt{a(X,Y)}, \mathtt{b(X,Y)}), \mathtt{c(Y)}.$$

This transformed query reuses the previously computed answer to its prefix by immediately binding the variables to their solution, and using the original prefix if this solution makes the refinement fail. Executing this transformed query on the second example indeed avoids the execution of the prefix, as the query succeeds immediately when calling d after binding Y to 1. For the second example, the corresponding transformed query is:

$$?- \ (\mathtt{X} = 2, \mathtt{Y} = 2 \ ; \ \mathtt{a(X,Y)}, \mathtt{b(X,Y)}), \mathtt{c(Y)}.$$

In this case, however, the previously computed solution {X=2,Y=2} of the prefix does not lead to a solution of the refinement. The prefix therefore has to be executed as normal.

Once-tabling can be implemented as a simple query transformation. Suppose again that a query is of the form (6.1):

$$?- \ \mathit{Prefix}(\overline{x}), \mathit{Refinement}(\overline{y}).$$

We transform this query into

$$?- \ (\mathtt{load\_solution}(i, \overline{x}) \ ; \mathit{Prefix}(\overline{x})), \ \mathit{Refinement}(\overline{y}), \mathtt{save\_solution}(j, \overline{x} \cup \overline{y}).$$

where i is a unique identifier for *Prefix*, j a unique identifier for the whole query. The predicates `load_solution` and `save_solution` store for a given key the variable bindings of their second argument. The solutions of queries are stored separately for each example. Executing this transformed query first retrieves the previously computed solution for the prefix, and then executes the refinement. If this solution fails to satisfy the refinement, the original prefix is executed.

A first advantage of the once tabling approach is that the extra memory required for storing answers to queries is limited to at most one solution per query. The second advantage is that it is relatively easy to implement this approach in any system, without necessarily having to resort to tabled execution. The disadvantage of this approach is that the prefix sometimes needs to be re-executed, although this extra overhead is compensated for if the majority of the queries succeed using the first solution of their prefix.

To be able to make a fair comparison between once tabling and the query tabling approach from Section 6.4.3, we implemented once tabling similarly to the implementation of query tabling, i.e. by using tabled execution. For every query of the form (6.1), we create the following predicate:

$$\mathtt{P}_j(\overline{x} \cup \overline{y}) \ :- \ \mathtt{once}(((\mathtt{P}_i(\overline{x}); \mathit{Prefix}(\overline{x})), \mathit{Refinement}(\overline{y})))$$

where i and j are unique identifiers for *Prefix* and *(Prefix,Refinement)* respectively. Similar to the predicate (6.6) created for query tabling, this predicate is also tabled. The query itself is then transformed into

$$?- \ \mathtt{P}_j(\overline{x} \cup \overline{y}).$$

With this transformation, only the first answer to every query will be stored in memory.

For example, a query from the first iteration (without prefix)

$$?- \ \mathtt{a(X,Y)}, \mathtt{b(Y,Z)}$$

is transformed into

$$?- \ \mathtt{P_{a\_b}(X,Y,Z)}.$$

with predicate $\mathtt{P_{a\_b}}$ tabled, and defined as

$$\mathtt{P_{a\_b}(X,Y,Z)} \ :- \ \mathtt{once((a(X,Y), b(Y,Z)))}.$$

A refinement of this query

$$?- \ \mathtt{(a(X,Y), b(Y,Z)), c(Z,U)}$$

is transformed into

$$?- \ \mathtt{P_{a\_b\_c}(X,Y,Z,U)}$$

with predicate $\mathtt{P_{a\_b\_c}}$ also tabled, and defined as

$$\mathtt{P_{a\_b\_c}(X,Y,Z,U)} \ :- \ \mathtt{once(((P_{a\_b}(X,Y,Z); a(X,Y), b(Y,Z)), c(Z,U)))}.$$

Experiment	Lookahead		
	0	1	2
No Optimization	390715	1347047	169003392
Prefix Tabling (Local)	296134	937022	9287876
Prefix Tabling (Batched)	296283	937120	9288178
Query Tabling (Local)	105560	542372	4013514
Query Tabling (Batched)	52115	443086	3667033
Once Tabling	103237	777236	69336452
Query Packs	184267	444453	2407452

Table 6.4: Total number of goal calls for running TILDE on Mutagenesis.

6.4.5 Evaluation

The main goal of our evaluation is to see whether the tabling approaches described in Sections 6.4.3 and 6.4.4 provide an advantage over the query packs approach (at the cost of extra memory for storing the answers). To measure this, we built a prototype for the tabling approaches, based on the implementation of (Rocha, Fonseca, and Costa 2005). Because the tabling approaches require support for predicate tabling in the engine, and since this is not available in the hipP system, the prototype was implemented in YAP. On the other hand, YAP does not support query pack execution, making it impossible to compare both approaches based on timing measurements. We therefore estimate the potential benefit of these techniques by measuring the total number of goals that are actually called, and comparing these to each other. We do this by tracing and simulating execution as described in Chapter 5, using trace simulators for hipP and YAP.

In our experiments, query traces from the TILDE and the WARMR algorithms with different lookahead settings on both the Mutagenesis and Carcinogenesis data sets were recorded. These traces were adorned with calls to predicates recording the number of calls, and the resulting traces were fed to three different trace simulators: one that executes the queries from a trace in their original form, one that first applies the tabling transformations on the queries before executing the queries, and one that executes query packs. YAP provides two different modes for tabling predicates: *local* tabling computes the complete table of a called predicate before continuing execution, while in *batched* tabling the table is constructed by need. Batched tabling therefore in theory performs less calls to predicates than local tabling. However, due to the way YAP handles the combination of tabling with the cut at the end of each query, answers of a query sometimes need to be recomputed when the table needs to be completed further.

The total number of goal calls for the different TILDE runs are shown in Tables 6.4 and 6.6. As expected, query packs outperform prefix tabling in

Experiment	Lookahead		
	0	1	2
Prefix Tabling (Local)	37 kB	37 kB	36 kB
Prefix Tabling (Batched)	37 kB	37 kB	36 kB
Query Tabling (Local)	10 MB	15 MB	92 MB
Query Tabling (Batched)	3 MB	6 MB	73 MB
Once Tabling	3 MB	6 MB	73 MB

Table 6.5: Maximum table size for running TILDE on Mutagenesis.

Experiment	Lookahead	
	0	1
No Optimization	62094851	629636203
Prefix Tabling (Local)	17826104	122158703
Prefix Tabling (Batched)	17826104	122157269
Query Tabling (Local)	12878382	52005387
Query Tabling (Batched)	10436092	50486078
Once Tabling	33541719	287747657
Query Packs	17659174	30388904

Table 6.6: Total number of goal calls for running TILDE on Carcinogenesis.

Experiment	Lookahead	
	0	1
Prefix Tabling (Local)	1.5 MB	241 kB
Prefix Tabling (Batched)	1.5 MB	241 kB
Query Tabling (Local)	283 MB	365 MB
Query Tabling (Batched)	25 MB	257 MB
Once Tabling	20 MB	255 MB

Table 6.7: Maximum table size for running TILDE on Carcinogenesis.

Experiment	Lookahead		
	0	1	2
No Optimization	5007638	149930892	2625159941
Prefix Tabling (Local)	1699134	72049883	1180152456
Prefix Tabling (Batched)	1642281	72249388	1181515793
Query Tabling (Local)	6410317	-	-
Query Tabling (Batched)	1284685	70777264	-
Query Packs	1367278	36991076	418259748

Table 6.8: Total number of goal calls for running WARMR on Mutagenesis.

Experiment	Lookahead		
	0	1	2
Prefix Tabling (Local)	2 MB	2 MB	57 MB
Prefix Tabling (Batched)	2 MB	2 MB	57 MB
Query Tabling (Local)	776 MB	-	-
Query Tabling (Batched)	130 MB	882 MB	-

Table 6.9: Maximum table size for running WARMR on Mutagenesis.

all experiments, and query tabling performs better than query packs in the settings without lookahead. However, with lookahead enabled, query packs start compensating the prefix recomputation cost by taking advantage of similarity in the prefix and refinements, and as such outperform query tabling with a higher lookahead setting. The once tabling approach always performs at least twice as good as the non-transformed queries. However, in most cases, it is still outperformed by the other optimizations. Because the tables are cleared in every iteration, the prefix tabling approach has very limited memory overhead, as can be seen in Tables 6.5 and 6.7. The total size of the tables used during once tabling does not differ much from the table sizes of query tabling. This confirms that execution seldom computes more than one solution for the tabled prefixes.

The results are even more pronounced when using WARMR, as can be seen in Tables 6.8 and 6.9. Unfortunately, the machine on which the experiments were conducted (a Pentium 4 with 2 Gb RAM) ran out of memory for some experiments due to the size of the tables of the queries. This illustrates that the tabling approach can become a problem in practice. Tables 6.8 and 6.9 do not include results of the once table experiments, because these could not be performed because of a bug in the YAP system at the time of this writing.

6.4.6 Conclusions

The tabling approaches described in this section where aimed at reducing redundancy in execution of prefixes, at the cost of extra memory usage. While the extra memory cost these approaches induce is still low enough for small data sets and simple queries, the size of the query tables grows to an unmanageable amount for more complex experiments. Moreover, a qualitative comparison shows that query packs outperform the tabling-based approaches for the more complex experiments, without requiring the extra overhead.

6.5 Remembering Query Coverage

6.5.1 Introduction

We concluded in Section 6.4 that remembering the answers to all queries in order to avoid executing them can be effective to reduce the total run time of an ILP algorithm, yet scales very badly with larger problems. In this section, we look at two specific ILP algorithms, and briefly describe algorithm-specific optimizations that also store answers in memory, yet at a much lower cost than the general tabling scheme from Section 6.4.

6.5.2 Tilde

Consider the evaluation step from TILDE as described in Section 2.2.2. After having selected the best query from the set of evaluated queries, TILDE partitions the current set of active examples into two sets: the set of examples on which the selected query succeeds, and the set on which it fails. The algorithm is then recursively called on both sets, thus creating two subtrees of the current node. In the left subtree (the one with the examples on which the selected query succeeds), TILDE refines the selected query further by adding literals to the query. In the right subtree, TILDE has to find a different best query that describes the remaining set of examples (i.e. the set of examples that are not covered by the selected query). However, this means that the same set of queries (minus the query selected in the previous node) has to be evaluated on all the examples. Therefore, if the algorithm stores for every query which examples it covers, it avoids having to execute the execution of the queries in the right subtree.

We implemented this optimization in query packs implementation of the hipP engine. For every query pack that is loaded, an extra data structure is allocated to record the success of the separate queries in the pack. This data structure stores a bitmap for each example in the data set, where every bit in a bitmap of an example represents the coverage of a query from the pack on the example. When a query pack is executed for the first time on a certain example, it is executed as normal. When a leaf of the pack is reached, the success for the corresponding query is recorded in the bitmap of the corresponding example. When the query pack is executed on the same example again, the success of every query in the bitmap is directly reported to the caller, and the pack is not executed anymore.

The results of running this implementation on the Mutagenesis, Carcinogenesis and HIV data sets can bee seen in Table 6.10. The first experiment consists of running TILDE on the data set without any optimizations. The second experiment enables the existing right sub-tree (RST) optimization, where TILDE explores the right branch of the sub-tree first, and reuses the compiled version of the pack. The final experiment enables the RST optimization and reuses the answers to the queries as well. The results show that remembering the answers

Experiment	Dataset							
	Mutagenesis			Carcinogenesis			HIV	
	LA0	LA1	LA2	LA0	LA1	LA2	LA0	LA1
No Opt.	0.37	1.38	5.78	0.75	1.48	6.80	35.82	169.52
RST Opt.	0.21	0.70	3.45	0.67	1.25	6.20	20.71	97.99
Reuse Opt.	0.19	0.68	3.32	0.45	0.89	5.63	11.01	41.23

Table 6.10: Total execution time for running TILDE with different optimizations on different data sets with different lookahead settings (in seconds).

Dataset	Examples[a]	LA[b]	Q[c]	Size[d]
Mutagenesis	230	0	43	1.3 kB
		1	278	7.9 kB
		2	1376	38.7 kB
Carcinogenesis	330	0	97	4.2 kB
		1	1525	61.6 kB
		2	7613	306.8 kB
HIV	4149	0	385	198.5 kB
		1	31573	15.6 MB

[a]Total number of examples in the data set
[b]Lookahead setting
[c]Maximum number of queries per pack
[d]Maximum size of the reuse data structure ($=Examples \cdot \lceil Q/8 \rceil$ bytes)

Table 6.11: Memory overhead of the success reuse data structure.

of queries can reduce the total execution time of the ILP algorithm up to 4 times over the normal execution, and up to 2.5 times over the RST optimization. The total memory introduced by the extra data structure is shown for each experiment in Table 6.11. Since the bitmap needs to contain one bit per query, and every example has one corresponding bitmap, the maximum size is $E \cdot \lceil Q/8 \rceil$ bytes, where E is the number of examples in the data set, and Q the maximum number of queries that occur in a pack during the whole run. For the most complex experiment on the largest data set, this size is 15.6 MB.

By implementing the success reuse optimization at the engine level, the optimization is independent of the ILP algorithm evaluating the query pack. However, this approach might as well be implemented at the algorithm level, as only algorithms running the same query pack on the same examples benefit from this optimization.

6.5.3 Warmr

Consider the example query trace from Figure 6.8. The first query of the first iteration covers the first example, but does not cover the second example, while

Iteration 1
a(X).
b(X).

Iteration 2
(a(X)),c(X).
(b(X)),d(X).

Example 1
a(1).
c(2).

Example 2
b(1).
d(2).

Figure 6.8: An example WARMR run.

```
function ADPACK-DISABLE(𝒟) :
    current_query := 0
    for each b in BranchTable, is_leaf(b) :
        if current_query ∈ 𝒟 :
            BranchTable[b].success := true
            alttable := BranchTable[b].adpackAltTable
            alttable.tosucceed := alttable.tosucceed - 1
            while alttable.tosucceed = 0 and alttable.parent_branch :
                BranchTable[alttable.parent_branch].success = true
                alttable := BranchTable[alttable.parent_branch].adpackAltTable
                alttable.tosucceed := alttable.tosucceed - 1
        current_query := current_query + 1
```

Figure 6.9: ADPACK-DISABLE: Disable the queries with indexes $\mathcal{D}$ in the current adpack.

the second query only covers the second example. Suppose that WARMR selects both queries for further refinement, and refines them into the queries from the second iteration. When these queries are run on both examples, we already know that the first query fails on the second example (since it is a refinement of a failing query from the first iteration), and likewise for the second query on the first example. This can be avoided by remembering for each query on which examples its parent query succeeded, and by not executing a query on an example if its parent query failed on that example. This optimization is trivial to implement on normal query execution, but requires more work when queries are executed using query pack execution. Compiling a pack with different queries for each example introduces significant overhead. We therefore extend the pack execution implementation to allow disabling certain queries in the pack.

Because the data structures used for adpack execution in Chapter 3 were designed to allow the (temporary) disabling of branches, it makes sense to reuse these data structures to implement the dynamic disabling of branches. Using the adpack execution mechanism in the absence of activate/deactivates is equivalent to query pack execution. We extend the ADPackAltTalbe from Figure 3.7

Experiment	Dataset				
	Mutagenesis			Carcinogenesis	
	LA0	LA1	LA2	LA0	LA1
No Optimization	0.718	15.5	107.1	4.1	149.3
Disable Optimization	0.676	9.0	63.0	2.2	71.0

Table 6.12: Query execution time for running WARMR traces with and without the dynamic query disabling optimization (in seconds).

(page 38) with one extra field: a reference to the parent branch of an adpack-or in the BranchTable. Using these data structures, it is now possible to write a function ADPACK-DISABLE that disables queries in an adpack, shown in Figure 6.9. ADPACK-DISABLE assumes that all the adpack data structures have been initialized. The parameter $\mathcal{D}$ is a list of indexes of queries that have to be disabled. ADPACK-DISABLE iterates over all leaves of the adpack, and checks against $\mathcal{D}$ if they need to be disabled. If so, the branch is marked as successful, and we iteratively disable all parent branches until there is another sibling branch that was not yet disabled. At the end of this process, all branches belonging only to queries that are disabled are marked as successful, and will therefore not be executed anymore.

Storing the success of queries on examples can be done similarly as in Section 6.5.2. For every example, a bitmap is stored, where a bit represents the success of a query on the example. As shown in Section 6.5.2, this data structure does not introduce a lot of extra memory usage.

The optimization described above cannot be implemented exclusively at the engine level: it requires modifying the WARMR algorithm to keep track of the parents of the generated queries, and make it disable queries in the adpack if necessary. To estimate the impact of this optimization without needing to modify WARMR itself, we record query traces of a WARMR run, and implement a prototype of this optimization in a simple trace simulator (see Chapter 5). This trace simulator keeps track of the success or failure of queries during every iteration. Before executing an adpack on an example, it calls ADPACK-DISABLE with a list of queries whose parent query failed in the previous iteration on the same example. Traversing the adpack while closing branches in ADPACK-DISABLE causes overhead in evaluation time, but this should be compensated for by not executing these queries. Running the optimizing trace simulator on various traces of WARMR runs on the Mutagenesis, Carcinogenesis and HIV data sets and measuring the query evaluation time results in Table 6.12. Disabling queries improves the evaluation time for all experiments, reaching a speedup of factor 2. However, these timings only cover the query evaluation time. The WARMR algorithm has a complex query generation phase, and so the impact of this optimization will be smaller on the total run time of the WARMR algorithm.

Level	Technique	Pro	Contra
Background	**Precomputation**	Best performance	Ad-hoc
		Memory	No cuts
	Specialization	Supports cuts	Bad performance
			Does not scale
	Tabling	Supports cuts	Dynamic overhead
		Good performance	
Query	**Prefix tabling**	Limited tables	Does not scale
	Query tabling	Best performance	Does not scale
	Once tabling	Limited tables	Does not scale
		Portable	Bad performance
Algorithm	**Tilde**	Memory	
		Performance	
	Warmr	Memory	
		Performance	

Table 6.13: Overview of the techniques described. Techniques applied on different levels are independent of each other.

6.5.4 Conclusions

In this section, we presented two optimizations at the ILP algorithm level. By only remembering whether or not a query succeeds on a given example, both the Tilde the Warmr algorithm can be optimized by avoiding the execution of queries that are known to succeed or fail. Contrary to remembering the full answers to queries as in Section 6.4, storing query coverage information does not introduce much memory overhead.

6.6 Conclusions

In this chapter, we discussed different techniques for storing execution results in memory, and reusing them to improve the efficiency of ILP algorithms. A brief overview of the approaches described can be seen in Table 6.13.

Storing the results for complex background relations as described in Section 6.3 is crucial to tackle ILP problems in acceptable time. This is currently done in an ad-hoc way by manually identifying the complex predicates in the background knowledge, precomputing their answers, and adding the resulting facts to the knowledge base. A more general approach is to apply specialization techniques on the background knowledge beforehand, or by tabling the execution of these predicates during the execution of the ILP algorithms. Preliminary experiments show that partial evaluation suffers from control problems: when the size of an example increases, the total time needed to specialize a background predicate for an example increases rapidly with increasing complexity of the example. Bottom-up (query-independent) specialization suffers even more from control problems, because the number of specialized versions of predicates

explodes with increasing size of examples. Tabled execution of background predicates reduces the total execution time significantly. The advantage over manually tabulated background knowledge is that it only stores the answers needed during the execution of an algorithm, and that it is safe in the presence of cuts in predicates (as far as the tabled execution supports cuts). However, this approach introduces a run-time overhead, and moreover does not measure up to the total speed of executing learning algorithms on the manually tabulated version of the data set.

Storing results of queries and sub-queries, and reusing these answers later when executing refined versions of the queries avoids recomputation during query execution. Section 6.4 reported that prefix and query tabling indeed avoid executing large parts of queries, but scale poorly with respect to memory usage when the data and problem complexity increases. The query packs approach also aims to avoid similar redundancy, yet does this by defining an alternative execution scheme instead of storing information. We compared both approaches by measuring the number of goal calls called when using each approach. From this experiment, we concluded that query packs perform only slightly worse on smaller problems, yet perform better on larger problems, without the heavy memory requirement of having to store information for every example in the database.

Finally, instead of storing the full answers to queries, more refined memorization schemes can be used for specific algorithms. By only storing the success of queries on the examples, algorithms like Tilde and Warmr can avoid executing queries (or parts of queries) as well. While these techniques also store information for every example in the data set, the amount of storage needed is very small, and therefore usable in real-life data sets such as the HIV data set.

Chapter 7

Conclusions

The main goal of this work was to develop techniques for optimizing the query evaluation step of ILP algorithms. The most crucial part for query evaluation is the ILP engine itself, which has to execute a large number of queries a large number of times. This work was therefore focused on developing techniques to yield more efficient ILP engines.

7.1 Contributions

This section discusses the various contributions presented in this work.

ADPacks. In Chapter 3, we focused on combining two successful (independent) query evaluation techniques, namely query packs and the once transformation. To achieve this, the notion of an *adpack* was introduced, which is an extension for query packs to deal with once transformations. We defined an execution mechanism for adpacks, and implemented this execution mechanism in terms of new WAM instructions. Additionally, two techniques for transforming a set of queries into an adpack were discussed. Evaluation of the implementation of this new execution mechanism showed that query pack execution indeed benefits from combining it with the once transformation, as execution time reached up to a two time decrease compared to the fastest approach, query packs. However, the time needed to transform a set of queries into an adpack is larger than the time needed to compile and execute the query, making the new approach less performant than query packs in some experiments.

Embedded meta-call. Based on the observation that compilation time in the classical compile-and-run approach for query evaluation dominates the query evaluation time, we investigated the meta-call in Chapter 4 as an alternative for query execution that does not require a compilation step. By embedding a version of the meta-call specialized for query execution in the internals of the

system, the execution time was improved five-fold over the traditional meta-call. Although the query execution using the embedded meta-call reaches the same speeds as that of executing compiled queries, the meta-call suffers from extendibility problems: combining the embedded meta-call with advanced execution mechanisms such as query packs or adpacks requires hard-coding these execution mechanisms in the meta-call itself, and moreover introduces the need for a preprocessing step, undoing the major advantage of the meta-call.

(Lazy) control flow compilation. Control flow compilation was developed in Chapter 4 as another alternative to compile-and-run query execution, combining the meta-call with a simple compilation step. Not only does this approach improve compilation times with an order of magnitude, it also lends itself well to extension, and allows reusing built-in instructions such as those used for query pack and adpack execution. This approach was further extended to yield a lazy compilation variant, compiling parts of queries only when they are needed. This approach improved the total query evaluation time even more.

Debugging and Analysis techniques. Chapter 5 discussed trace-based techniques for analyzing and debugging the query execution of ILP algorithms. By using static traces recorded during the run of an ILP algorithm, analysis and debugging can be performed outside of the ILP system, and independent of the ILP algorithm executing the queries. This eases the development of query analyses, and speeds up the execution of the analyses. Locating bugs becomes easier because less code needs to be traced, and other (potentially costly) steps from the ILP algorithm are skipped. Moreover, the time needed to expose a bug can be reduced drastically by trimming down the trace, which can be done automatically by applying the delta debugging algorithm.

Study of space/time tradeoffs. In Chapter 6, we studied the advantages and disadvantages of storing computed results in memory, such that these results can be reused in future execution steps. This answer reuse can be done on different levels. For calls to complex background knowledge predicates, it is necessary to remember computations for the ILP algorithm to finish in reasonable time. The current ad-hoc approach, consisting of precomputing all answers to complex background knowledge predicates in advance, outperforms the more general approaches where specialization techniques are applied on the data set, or calls to background predicates are tabled. Tabling the answers to queries and prefixes only results in less goal calls than query packs for simple experiments. Moreover, the size of the query tables becomes a problem for the larger experiments. Moreover, query packs save more calls for more complex experiments, without introducing such a heavy memory requirement. The once tabling approach aims at providing a tradeoff between storing answers and recomputing them. However, the memory usage of this approach in practice does not differ much from the other tabling approaches, yet performs worse. Finally, we showed that by remembering only the success of queries, TILDE receives a performance

improvement of up to 2.5 times, whereas the query execution time of WARMR improves up to a factor of 2. The extra memory requirement of this optimization is very small.

7.2 Discussion

Throughout this work, different techniques were proposed that optimize ILP query execution. These approaches have mostly been presented isolated from each other. In this section, we give a global discussion of the benefit of these approaches, and when to apply which technique.

A first dimension in this discussion is the tradeoff between compilation and no compilation. The embedded meta-call developed in Chapter 4 is a strict improvement over the classical meta-call, and should therefore be used whenever meta-call was previously used for query evaluation. Meta-calling queries is interesting whenever an ILP algorithm needs to evaluate a query on only a few examples, as it does not require the initial cost paired with compiling the query. Moreover, query evaluation using the embedded meta-call can be combined with the once transformation. The decision of using meta-call instead of compiling a query first needs to be made by the ILP algorithm itself, based on an estimate of the cost it takes to meta-call or compile a query.

Control flow compilation provides a strict improvement over the classical compile-and-run approach: control flow compiling is 10 times faster than classical compilation, and the code it generates executes at least as fast. Whenever compilation is involved, it should therefore be replaced with control flow compilation. However, replacing compile-and-run with control flow compilation in every compilation-based execution mechanism requires some extra work. Although the instructions generated by the compile-and-run approach can be reused for control flow compiled code without any modification, a separate compiler is necessary. The control flow compiler is a very simple compiler that only needs to generate code regarding the control flow of queries, as illustrated in Appendix C. Hence, porting compilation schemes for execution mechanisms from the original compiler to the control flow compiler does not require much effort. The lazy variant of control flow compilation improves execution even more, but requires a larger effort to integrate with existing execution mechanisms. Execution mechanisms such as query packs and adpacks can no longer assume that the entire code is loaded at once, which makes the administration for the data structures used in these approaches harder. Because the data structures of adpacks are more complex than those of query packs, we have chosen query pack execution as the initial target for lazy control flow compilation.

A second discussion is whether to use query packs or adpacks. Although the query execution part of the adpacks is always better than query pack execution, the high cost of the transformation step makes the adpack approach slower

	Run time				Ranking	
	Gen.	Eval.			Many	Few
		Trans.	Comp.	Exec.		
Meta-call				□□□□□	7	2
Emb. Meta-call				□□□□	6	1
Tabling				□□□	4	2
ADPacks (C&R)	⋮	□	□□□	□	5	5
Query Packs (C&R)			□□□	□□	3	5
Query Packs (CF)			□□	□□	2	4
Query Packs (LCF)			□	□□	1	3

Table 7.1: Overview of different query evaluation mechanisms, in terms of the various components of the ILP algorithm run time: the time needed to generate, evaluate, transform, compile, and execute all queries. Higher numbers represent more time (and less performance). A global ranking is given for the case where few or many queries are evaluated.

in some cases. The evaluation of adpacks has been done using the classical compile-and-run approach. However, combining adpacks with lazy control flow compilation will make the impact of the transformation time on the total query evaluation process even more pronounced: the share of the compilation step will reduce significantly in both approaches, leaving the transformation time with a bigger share of the total time needed to preprocess queries. Without a better transformation, adpacks will therefore be outperformed by query packs in some experiments.

Of the new space/time tradeoffs introduced in Chapter 6, the most interesting optimizations are the algorithm-specific ones. Remembering query coverage in Tilde at the algorithm level is independent of the execution mechanism used. However, we implemented this query coverage at the engine level, which required small changes to the query pack execution mechanism. The second optimization was the dynamic disabling of queries for Warmr. This approach relied on the design of the adpack data structures to be able to disable parts of a pack. The classical data structures for query packs do not cater for this, and it is therefore not possible to integrate this approach with the standard query pack implementation.

To conclude this discussion, we present an informal summary of the performance of the most important query evaluation techniques in Table 7.1. This table illustrates for each component of the ILP algorithm run time how much time the different techniques relatively consume. For example, the embedded meta-call executes slower than query packs, but is faster than the regular meta-call. The rightmost part of the table shows how well the different approaches perform when either few or many queries need to be evaluated. The optimal approach is to use query packs in combination with lazy control flow compila-

tion for evaluating many queries, and to use the embedded meta-call if only a few queries need to be evaluated.

7.3 Further Work

In this section, we present possible directions for future work regarding the techniques described throughout this text.

ADPacks. The weak point of the adpack approach is the transformation. To be able to outperform query packs in every experiment, the time needed to transform a set of queries to an adpack needs to be reduced significantly. The major bottleneck of the pack based transformation we presented was the repeated transformation of a set of queries into a query pack. A possible way to overcome this expensive step is to make the transformation modify the pack in place. Such a destructive transformation requires a low level implementation, which is not trivial.

Another interesting direction for further research is investigating the application of more 'advanced' techniques of avoiding backtracking (such as intelligent backtracking), and to combine this with query packs.

(Lazy) Control Flow Compilation. We mentioned in Section 7.2 that replacing classical compilation with lazy control flow compilation requires some work. This has been done for query packs, but not yet for the adpacks. The most difficult part of this conversion is the modification of the data structures to provide support for incremental loading of goals. We expect lazy control flow compilation to yield the same speedups for adpacks as it did for query packs.

Incremental Compilation. Lazy control flow compilation makes it possible to exploit the incremental nature of queries generated by an ILP system, as was illustrated at the end of Section 4.4 (page 71). Queries can be compiled incrementally, such that parts of queries compiled in previous iterations are reused, and that only the new parts of queries are compiled. This has not been investigated further, because the total share of query compilation after introducing control flow compilation was no longer big enough in our experiments for incremental compilation to yield a potential benefit at this time. This approach would be interesting in situations where the complexity of queries is much bigger.

Appendix A

Datasets

A.1 Mutagenesis

The Mutagenesis data set (Srinivasan, Muggleton, Sternberg, and King 1996) is one of the most frequently used data sets throughout the ILP community. This database contains descriptions of 230 molecules, for which the the mutagenicity of the molecules is to be predicted (mutagenicity is the ability to cause DNA to mutate, which is a possible cause for cancer).

An example of a molecule description from this data set can be seen in Figure A.1. An `atom/4` fact states the name of the atom, the element (e.g. `c` for carbon), the type of the atom, and its partial charge. For example, the fact `atom(d1_1,c,22,-0.117)` means that atom `d1_1` is an aromatic carbon atom with negative charge -0.117. A `bond/3` fact states that there is a bond of a certain type between 2 atoms. For example, the fact `bond(d1_1,d1_2,7)` means that there is an aromatic bond between the atoms `d1_1` and `d1_2`.

The background knowledge of the Mutagenesis data set contains definitions of higher level sub-molecular structures such as benzene rings, phenanthrene structures, ... For example, the definitions of `phenanthrene/1` and `benzene/1` is shown in Figure A.2. These definitions in turn depend on the definition of `ring6/3` and other auxiliary predicates (`interjoin/3` and `members_bonded/2`). Note that, for efficiency reasons, the most complex predicates in the background knowledge are usually replaced by their precomputed answers. Except for Chapter 6, we always use this specialized version of the Mutagenesis data set.

The Mutagenesis data set is a relatively small one compared to the other data sets used throughout this text.

A.2 Carcinogenesis

The Carcinogenesis data set (Srinivasan, King, and Bristol 1999) contains descriptions of 330 molecules, stored in the same format as the Mutagenesis data set. The data set contains for each molecule information about the carcino-

```
pos.

atom(d1_1,c,22,-0.117).
atom(d1_2,c,22,-0.117).
atom(d1_3,c,22,-0.117).
atom(d1_4,c,195,-0.087).
atom(d1_5,c,195,0.013).
atom(d1_6,c,22,-0.117).
atom(d1_7,h,3,0.142).
atom(d1_8,h,3,0.143).
 ⋮

bond(d1_1,d1_2,7).
bond(d1_2,d1_3,7).
bond(d1_3,d1_4,7).
bond(d1_4,d1_5,7).
bond(d1_5,d1_6,7).
bond(d1_6,d1_1,7).
bond(d1_1,d1_7,1).
bond(d1_2,d1_8,1).
bond(d1_3,d1_9,1).
bond(d1_6,d1_10,1).
 ⋮
```

Figure A.1: Example of a molecule in the Mutagenesis data set.

```
phenanthrene([Ring1,Ring2,Ring3]) :-
    benzene(Ring1),
    benzene(Ring2),
    Ring1 @> Ring2,
    interjoin(Ring1,Ring2,Join1),
    benzene(Ring3),
    Ring1 @> Ring3,
    Ring2 @> Ring3,
    interjoin(Ring2,Ring3,Join2),
    \+ interjoin(Join1,Join2,_),
    members_bonded(Join1,Join2).

benzene(Ring_list) :-
    atoms(6,Atom_list,[c,c,c,c,c,c]),
    ring6(Atom_list,Ring_list,[7,7,7,7,7,7]).

atoms(1,[Atom],[T]) :-
    atom(Atom,T,_,_),
    T \== h.
atoms(N1,[Atom1|[Atom2|List_a]],[T1|[T2|List_t]]) :-
    N1 > 1,
    N2 is N1 - 1,
    atoms(N2,[Atom2|List_a],[T2|List_t]),
    atom(Atom1,T1,_,_),
    Atom1 @> Atom2,
    T1 \== h.
```

Figure A.2: Excerpt from the background knowledge of the Mutagenesis data set.

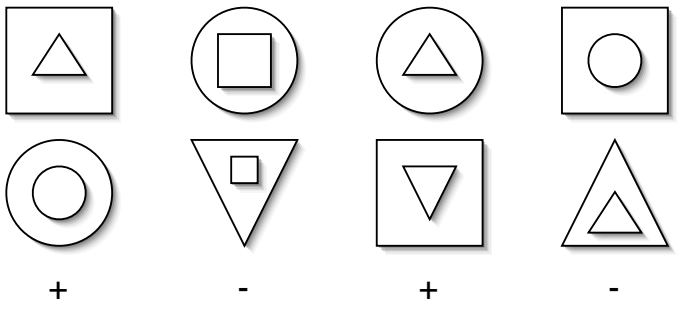


Figure A.3: A Bongard problem.

```
triangle(1).
orientation(1,up).
bounding_box(1,1,1,2,2).

square(2).
bounding_box(2,0,0,3,3).
```

Figure A.4: Example from the Bongard data set.

genicity (i.e. whether or not it causes cancer) of the molecule, obtained from experiments by the National Institute of Environmental Health Sciences. The goal of an ILP algorithm is to be able to predict the carcinogenicity of unseen molecules.

A.3 Bongard

The Bongard data set (De Raedt and Van Laer 1995) is an artificial data set, based on pattern recognition problems from (Bongard 1970). Every example in this data set corresponds to a positive or negative drawing, and the goal is to discover what characterizes a positive drawing. Figure A.3 shows a Bongard problem with 4 examples, where a positive example is characterized by the presence of a triangle contained inside a square. Each example in the data set consists of `square/1`, `triangle/1`, `circle/1` facts, together with the geometries of their bounded boxes as `bounding_box/2` facts, and optionally with information about orientation in the form of `orientation/2` facts. For example, the square and triangle from the leftmost illustration of Figure A.3 is depicted in Figure A.4. The background knowledge of the Bongard data set consists of predicates such as `inside/2`, `north_of/2`, ... representing the relative positions of the figures, described in terms of their bounding boxes.

A.4 HIV

The HIV data set (Kramer, Raedt, and Helma 2001) originates from the Developmental Therapeutics Program (NIH/NCI 2001) of the U.S. Department of Health and Human Services. The original data set contains a structural description of 43576 compounds, for which was measured whether or not they were able to protect human cells from HIV-1 infection. The structural information about the molecules are stored in a similar format as that of the Mutagenesis and Carcinogenesis data sets. Throughout this text, we use a trimmed down version of this data set, consisting of 4150 compounds.

Appendix B

ADPack Meta-interpreter

This appendix contains a Prolog meta-interpreter for the ADPacks execution mechanism, as defined in Section 3.3 (page 26). Note that the interpreter uses the hipP-specific built-in predicate `'_$savecp'/1` to retrieve the most recent choice point, and `'_$cutto'/1` to cut away all choice points up to a given choice point. To execute the adpack from Figure 3.2(c) (page 29) and list its successful branches, use the following query:

```
?- adpack_execute(
     [a(X),activate(1), activate(2), b(X,Y),
      adpack_or([
          branch(br1,[once(c(Y)), d(Y)]),
          branch(br2,[e(Y,Z), adpack_or([
              branch(br3,[deactivate(2), f(X)]),
              branch(br4,[g(Y,Z), deactivate(1), d(X)])])
          ])
      ])]),
   write('Success: '), (success(S), write(S), write(' '), fail ; nl).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ADPack Meta-interpreter
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Execute an adpack
adpack_execute(Pack) :-
    init_datastructures(Pack),
    '_$savecp'(B),
    adpack_execute(Pack,br0,[],B).
adpack_execute(_).

% Main predicate for executing the adpack.
%
% Argument 1: the adpack datastructure
% Argument 2: the identifier of the branch currently executing
% Argument 3: a list of activate identifiers, paired with the
%             current choicepoint at the time of the activate
% Argument 4: the choicepoint of the parent adpack-or of the
%             current branch
adpack_execute([activate(Id)|Gs],CurBranch,Activates,ParentCutp) :- !,
    deactivate_branch(Id,DeactBranch),
    (\+ success(DeactBranch) ->
open_path(DeactBranch,CurBranch),
        '_$savecp'(B),
        NActivates = [act(Id,B)|Activates]
    ;
        NActivates = Activates
    ),
    adpack_execute(Gs,CurBranch,NActivates,ParentCutp).
adpack_execute([deactivate(_)|Gs],CurBranch,Activates,ParentCutp) :- !,
    set_closed(CurBranch),
    '_$cutto'(ParentCutp),
    adpack_execute(Gs,CurBranch,Activates,ParentCutp).
adpack_execute([],CurBranch,_Activates,ParentCutp) :- !,
    set_success(CurBranch),
    '_$cutto'(ParentCutp),
    fail.
adpack_execute([adpack_or(Branches)],CurBranch,Activates,ParentCutp) :- !,
    (
        %% Forward execution %%
        member(branch(Branch,Goals), Branches),
        \+ success(Branch), open(Branch),
        '_$savecp'(B),
        adpack_execute(Goals,Branch,[],B)
    ;
        %% Backtracking %%
        ( all_success(Branches) ->
            set_success(CurBranch),
            '_$cutto'(ParentCutp)
        ;
```

```
            \+ contains_open_branch(Branches),
            determine_cutpoint(Activates,ParentCutp,CP),
            (CP == ParentCutp ->
                set_closed(CurBranch)
            ;
                '_$cutto'(CP)
            )
        ),
        fail
    ).
adpack_execute([G|Gs],CurBranch,Activates,ParentCutp) :-
    call(G),
    adpack_execute(Gs,CurBranch,Activates,ParentCutp).

% Opens all branches in a path until an already open branch
% is found or the begin branch is reached
open_path(BeginBranch,BeginBranch) :- !.
open_path(CurBranch,BeginBranch) :-
    ( open(CurBranch) ->
        true
    ;
        set_open(CurBranch),
        parent(CurBranch,ParentBranch),
        open_path(ParentBranch,BeginBranch)
    ).

% Succeeds if all branches in the list are successful
all_success([]).
all_success([branch(Branch,_)|Branches]) :-
    success(Branch),
    all_success(Branches).

% Succeeds if the list of branches contains an open,
% unsuccessful branch
contains_open_branch(Branches) :-
    member(branch(Branch,_),Branches),
    \+ success(Branch),
    open(Branch).

% Computes the most recent useful cutpoint to backtrack to
determine_cutpoint([],ParentCutp,ParentCutp).
determine_cutpoint([act(Id,B)|Acts],ParentCutp,Cutp) :-
    deactivate_branch(Id,E),
    (\+ success(E) ->
        Cutp = B
    ;
        determine_cutpoint(Acts,ParentCutp,Cutp)
    ).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Datastructures
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- dynamic open/1, success/1, deactivate_branch/2, parent/2.

% Initializes the used datastructures
init_datastructures(Pack) :-
    init_datastructures(Pack,br0).

init_datastructures([],_) :- !.
init_datastructures([deactivate(Id)|Gs],CurrentBranch) :- !,
    assert(deactivate_branch(Id,CurrentBranch)),
    init_datastructures(Gs,CurrentBranch).
init_datastructures([adpack_or(Branches)],CurrentBranch) :- !,
    (
        member(branch(E,G),Branches),
        set_open(E),
        retractall(success(E)),
        assert(parent(E,CurrentBranch)),
        init_datastructures(G,E),
        fail
    ;
       true
    ).
init_datastructures([_|Gs],E) :-
    init_datastructures(Gs,E).

% Checks and sets different flags of branches
set_open(E) :- once((open(E) ; assert(open(E)))).
set_closed(E) :- retractall(open(E)).
set_success(E) :- once((success(E) ; assert(success(E)))).

% Auxiliary predicate
member(X,[X|_]).
member(X,[_|Ys]) :- member(X,Ys).
```

Appendix C

Control Flow Compiler

This appendix contains a control flow compiler, written in Prolog. The compiler is used for compiling queries containing conjunctions and disjunctions to WAM instructions. For goals with an arity smaller than 4, a specialized instruction is emitted. The goals `true/0` and `'<'/2` are treated as special built-ins.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Control Flow Compiler
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Compile a predicate
cf_compile((Head :- Body),Code) :-
    Code = [allocate(2)|Code1],
    ( atom(Head) ->
        Code1 = Code2
    ;
        Code1 = [cf_unifyhead(Head)|Code2]
    ),
    cf_compile(Body,0,Code2,_,[]).


% Compile a conjunction
cf_compile(','(A,B),Label,Code,NLabel,NCode) :- !,
    cf_compile_goal(A,Code,Code1),
    cf_compile(B,Label,Code1,NLabel,NCode).
cf_compile(';'(A,B),Label,Code,NLabel,NCode) :- !,
    Code = [trymeorelse(Label)|Code1],
    Label1 is Label + 1,
    cf_compile(A,Label1,Code1,Label2,Code2),
    Code2 = [label(Label)|Code3],
    cf_compiledisj(B,Label2,Code3,NLabel,NCode).
```

```
cf_compile(B,Label,Code,Label,NCode) :-
    ( cf_builtin(B) ->
        cf_compile_goal(B,Code,Code1),
        Code1 = [deallocate,proceed|NCode]
    ;
        functor(B,_,Arity),
        ( Arity =< 4 ->
            Code = [cf_deallex_4(B)|NCode]
        ;
            Code = [cf_deallex(B)|NCode]
        )
    ).

% Compile a disjunction
cf_compiledisj(';'(A,B),Label,Code,NLabel,NCode) :- !,
    Code = [retrymeorelse(Label)|Code1],
    Label1 is Label + 1,
    cf_compile(A,Label1,Code1,Label2,Code2),
    Code2 = [label(Label)|Code3],
    cf_compiledisj(B,Label2,Code3,NLabel,NCode).
cf_compiledisj(G,Label,Code,NLabel,NCode) :-
    Code = [trustmeorelsefail|Code1],
    cf_compile(G,Label,Code1,NLabel,NCode).

% Compile a single (non-final) goal
cf_compile_goal(true,Code,Code) :- !.
cf_compile_goal('<'(A,B),Code,NCode) :- !,
    Code = [cf_test_smaller(A,B)|NCode].
cf_compile_goal(A,Code,NCode) :-
    functor(A,_,Arity),
    ( Arity =< 4 ->
        Code = [cf_call_4(A)|NCode]
    ;
        Code = [cf_call(A)|NCode]
    ).

% Goals treated as special builtins
cf_builtin('<'(_,_)).
cf_builtin(true).
```

References

ACE (2006). The ACE data mining system. http://www.cs.kuleuven.be/~dtai/ACE/.

Agrawal, R., H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo (1996). Fast discovery of association rules. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy (Eds.), *Advances in Knowledge Discovery and Data Mining*, pp. 307–328. MIT.

Aït-Kaci, H. (1991). *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press. http://www.vanx.org/archive/wam/wam.html.

Andersen, L. O. (1993). Binding-time analysis and the taming of C pointers. In *PEPM93*, pp. 47–58. ACM.

Aycock, J. (2003). A brief history of just-in-time. *ACM Computing Surveys 35*(2), 97–113.

Blockeel, H. and L. De Raedt (1998, June). Top-down induction of first order logical decision trees. *Artificial Intelligence 101*(1-2), 285–297.

Blockeel, H., L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele (2002). Improving the efficiency of Inductive Logic Programming through the use of query packs. *Journal of Artificial Intelligence Research 16*, 135–166. http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=36467.

Blockeel, H., S. Dzeroski, B. Kompare, S. Kramer, B. Pfahringer, and W. Van Laer (2004). Experiments in predicting biodegradability. *Applied Artificial Intelligence 18*(2), 157–181. http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=40890.

Bongard, M. (1970). *Pattern Recognition*. Spartan Books.

Bratko, I. (2001). *Prolog Programming for Artificial Intelligence* (3rd ed.). Addison-Wesley.

Bratko, I. and M. Grobelnik (1993). Inductive learning applied to program construction and verification. In *Proc. Third International Workshop on Inductive Logic Programming*, Bled, Slovenia, pp. 279–292.

Breiman, L., J. Friedman, R. Olshen, and C. Stone (1984). *Classification and Regression Trees*. Wadsworth.

Bruynooghe, M. and L.-M. Pereira (1984). Deduction revision by intelligent backtracking. In J. Campbell (Ed.), *Implementation of Prolog*, pp. 194–215. Ellis Horwood.

Byrd, L. (1980). Understanding the control flow of Prolog programs. In S.-A. Tarnlund (Ed.), *Proceedings of the Workshop on Logic Programming*.

Chan, T. W. and A. Lakhotia (1998). Debugging program failure exhibited by voluminous data. *Journal of Software Maintenance 10*(2), 111–150.

Clocksin, W. F. and C. S. Melish (2003). *Programming in Prolog* (5th ed.). Springer-Verlag.

Costa, V. S., A. Srinivasan, R. Camacho, H. Blockeel, B. Demoen, G. Janssens, J. Struyf, H. Vandecasteele, and W. Van Laer (2002). Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research*. http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=38848.

Damas, L. and V. S. Costa (2003). YAP user's manual. http://yap.sourceforge.net.

Davis, J., E. Burnside, I. Dutra, D. Page, R. Ramakrishnan, V. S. Costa, and J. Shavlik (2005). View learning for statistical relational learning: With an application to mammography. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-05)*.

De Raedt, L. (1998). Attribute-value learning versus Inductive Logic Programming: The missing links. In D. Page (Ed.), *Proceedings of 8th International Conference on Inductive Logic Programming*, Volume 1446 of *Lecture Notes in Artificial Intelligence*, pp. 1–8. Springer-Verlag.

De Raedt, L. and W. Van Laer (1995). Inductive constraint logic. In K. P. Jantke, T. Shinohara, and T. Zeugmann (Eds.), *Proceedings of the Sixth International Workshop on Algorithmic Learning Theory*, Volume 997 of *Lecture Notes in Artificial Intelligence*, pp. 80–94. Springer-Verlag.

Dean, J., C. Chambers, and D. Grove (1995). Selective specialization for object-oriented languages. *SIGPLAN Not. 30*(6), 93–102.

Debray, S. and P. Mishra (1988). Denotational and operational semantics for Prolog. *Journal of Logic Programming 5*(1), 61–91.

Dehaspe, L. and H. Toivonen (1999). Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery 3*(1), 7–36.

Demoen, B., G. Janssens, and H. Vandecasteele (1999, November). Executing query flocks for ILP. In S. Etalle (Ed.), *Proceedings of the Eleventh Benelux Workshop on Logic Programming*, Maastricht, The Netherlands, pp. 1–14. http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=18794.

Dolsak, B. and S. Muggleton (1992). The application of Inductive Logic Programming to finite element mesh design. In S. Muggleton (Ed.), *Inductive Logic Programming*. Academic Press.

DTP (2001). The developmental therapeutics program. U.S. Departement of Health and Human Services NIH, National Cancer Institute NCI. http://dtp.nci.nih.gov.

Ducassé, M. (1999a). Coca: an automated debugger for C. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pp. 504–513. IEEE Computer Society Press.

Ducassé, M. (1999b). Opium: An extendable trace analyser for Prolog. *The Journal of Logic programming*. Special issue on Synthesis, Transformation and Analysis of Logic Programs, A. Bossi and Y. Deville (eds), Also Rapport de recherche INRIA RR-3257 and Publication Interne IRISA PI-1127.

Džeroski, S. and N. Lavrač (2001). *Relational Data Mining*. Springer-Verlag.

Fayyad, U., G. Piatetsky-Shapiro, and P. Smyth (1996). From data mining to knowledge discovery: An overview. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy (Eds.), *Advances in Knowledge Discovery and Data Mining*, pp. 495–515. MIT.

Feng, C. (1992). Inducing temporal fault diagnostic rules from a qualitative model. In S.Muggleton (Ed.), *Inductive Logic Programming*. Academic Press.

Gallagher, J. (1993). Specialisation of logic programs: A tutorial. In *PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 88–98. ACM Press.

Gaschnig, J. (1979). Performance measurement and analysis of certain search algorith ms. Technical Report CMU-CS-79-124, Carnegie Mellon University, Pittsburgh, PA.

hipP (2006). hipP: A high performance Prolog system. http://www.cs.kuleuven.be/~dtai/hipp/.

Jahier, E. and M. Ducassé (2002). Generic program monitoring by trace analysis. *Theory and Practice of Logic Programming 2*(4-5), 611–643.

Jones, N., C. Gomard, and P. Sestoft (1994). *Partial Evaluation and Automatic Program Generation*. Prentice Hall.

King, R., S. Muggleton, R. Lewis, and M. Sternberg (1992). Drug design by machine learning: The use of inductive logic programming to model the structure-activity relationship of trimephoprim analogues binding to dihydrofolate reductase. *Proceedings of the National Academy of Sciences 89*(23).

Kowalski, R. (1974). Predicate logic as programming language. In J. L. Rosenfeld (Ed.), *Proceedings of the Sixth IFIP Congress (Information Processing 74)*, Stockholm, Sweden, pp. 569–574.

Kramer, S., L. D. Raedt, and C. Helma (2001). Molecular feature mining in HIV data. In *KDD '01: Proceedings of the Seventh ACM SIGKDD*

International Conference on Knowledge Discovery and Data Mining, New York, NY, USA, pp. 136–143. ACM Press.

Lavrač, N., S. Džeroski, V. Pirnat, and V. Krizman (1993). The use of background knowledge in learning medical diagnostic rules. *Applied Artificial Intelligence 7*, 273–293.

Leuschel, M. and M. Bruynooghe (2002). Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming 2*(4&5), 461–515. http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=37399.

Leuschel, M. and D. D. Schreye (1996). Logic program specialisation: How to be more specific. In H. Kuchen and S. Swierstra (Eds.), *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, Volume 1140 of *Lecture Notes in Computer Science*, pp. 137–151. Springer-Verlag.

Lloyd, J. (1987). *Foundations of Logic Programming* (2nd ed.). Springer-Verlag.

Lloyd, J. and J. Shepherdson (1991). Partial evaluation in logic programming. *Journal of Logic Programming 11*(3&4), 217–242.

Muggleton, S. and L. De Raedt (1994). Inductive Logic Programming: Theory and methods. *Journal of Logic Programming 19/20*, 629–679.

Muggleton, S., R. D. King, and M. Sternberg (1992). Protein secondary structure prediction using logic-based machine learning. *Protein Engineering 5*(7), 647–657.

O'Keefe, R. A. (1990). *The Craft of Prolog*. MIT Press.

Page, D. and M. Craven (2003). Biological applications of multi-relational data mining. *SIGKDD Explor. Newsl. 5*(1), 69–79.

Proietti, M. and A. Pettorossi (1991). Semantics preserving transformation rules for Prolog. In *Proceedings of PEPM'91*, Volume 26 of *SIGPLAN Notices*, pp. 274–284.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc.

Ramon, J. and J. Struyf (2004). Efficient theta-subsumption of sets of patterns. In *Proceedings of the 13th Belgian-Dutch Conference on Machine Learning*, pp. 95–102.

Rocha, R., N. A. Fonseca, and V. S. Costa (2005). On Applying Tabling to ILP. In *Proceedings of the 16th European Conference on Machine Learning, ECML-05*, Lecture Notes in Artificial Intelligence. Springer-Verlag.

Rocha, R., F. Silva, and V. Costa (2000, September). Yaptab: A tabling engine designed to support parallelism. In *Proceedings of the 2nd Conference on Tabulation in Parsing and Deduction, TAPD'2000*, Vigo, Spain, pp. 77–87.

Sahlin, D. (1993). Mixtus: An automatic partial avaluator for full Prolog. *New Generation Comput. 12*(1), 7–51.

Spoto, F. (2000). Operational and goal-independent denotational semantics for Prolog with cut. *Journal of Logic Programming 42*(1), 1–46.

Srinivasan, A. (1999). A study of two sampling methods for analysing large datasets wi th ILP. *Data Mining and Knowledge Discovery 3*(1), 95–123.

Srinivasan, A. (2005). The Aleph manual. http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/.

Srinivasan, A., R. King, and D. Bristol (1999). An assessment of ILP-assisted models for toxicology and the PTE-3 experiment. In *Proceedings of the Ninth International Workshop on Inductive Logic Programming*, Volume 1634 of *Lecture Notes in Artificial Intelligence*, pp. 291–302. Springer-Verlag.

Srinivasan, A., S. Muggleton, M. Sternberg, and R. King (1996). Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence 85*(1,2), 277–299.

Sterling, L. and E. Shapiro (1994). *The Art of Prolog: Advanced Programming Techniques* (2nd ed.). MIT Press.

Struyf, J. (2004). *Techniques for improving the efficiency of inductive logic programming in the context of data mining*. Ph. D. thesis, K.U.Leuven, Leuven, Belgium. http://www.cs.kuleuven.be/publicaties/doctoraten/cw/CW2004_12.abs.html.

Tarau, P. (1992). Program transformations and WAM-support for the compilation of definite metaprograms. In A. Voronkov (Ed.), *Russian Conference on Logic Programming*, Number 592 in Lecture Notes in Artificial Intelligence, Berlin, Heidelberg, pp. 462–473. Springer-Verlag.

Tronçon, R., G. Janssens, and B. Demoen (2003). Alternatives for compile & run in the WAM. In *Proceedings of CICLOPS 2003: Colloquium on Implementation of Constraint and LOgic Programming Systems*, pp. 45–58. University of Porto. Technical Report DCC-2003-05, DCC - FC & LIACC, University of Porto, December 2003. http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=41065.

Tronçon, R., H. Vandecasteele, J. Struyf, B. Demoen, and G. Janssens (2003). Query optimization: Combining query packs and the once-tranformation. In *Inductive Logic Programming, 13th International Conference, ILP 2003, Szeged, Hungary, Short Presentations*, pp. 105–115. http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=40938.

Vandecasteele, H., B. Demoen, and G. Janssens (2000). Compiling large disjunctions. In I. de Castro Dutra, E. Pontelli, and V. S. Costa (Eds.), *First International Conference on Computational Logic: Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, pp. 103–121. Imperial College. http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=32065.

Vanhoof, W., D. De Schreye, and B. Martens (1999). Bottom-up partial deduction of logic programs. *The Journal of Functional and Logic Programming 1999*, 1–33.

Vanhoof, W., R. Tronçon, and M. Bruynooghe (2003). A fixed point semantics for logic programs extended with cuts. In *Logic Based Program Synthesis and Transformation, LOPSTR 2002, Revised Selected Papers*, Volume 2664 of *LNCS*, pp. 238–257. Springer-Verlag. http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=40744.

Warren, D. H. D. (1983). An abstract Prolog instruction set. Technical Report 309, SRI.

Warren, D. S. et al. (2005, January). The XSB Programmer's Manual: version 2.7, vols. 1 and 2. http://xsb.sourceforge.net.

Weise, D., R. Conybeare, E. Ruf, and S. Seligman (1991). Automatic online partial evaluation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*.

Zeller, A. and R. Hildebrandt (2002). Simplifying and isolating failure-inducing input. *Software Engineering 28*(2), 183–200. http://www.st.cs.uni-sb.de/papers/tse2002/.

Index

Publication List

Articles in international reviewed journals

- R. Tronçon, G. Janssens, B. Demoen, and H. Vandecasteele, *Fast frequent quering with lazy control flow compilation*, Theory and Practice of Logic Programming, 2006, accepted.
  http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=41995

Contributions at international conferences, published in proceedings

- R. Tronçon, B. Demoen, and G. Janssens *When tabling does not work*, Proceedings of CICLOPS 2006: Colloquium on Implementation of Constraint and LOgic Programming Systems (Guo, H.-F. and Pontelli, E., eds.), pp. 18-31, 2006.

- R. Tronçon, and G. Janssens, *A delta debugger for ILP query execution*, Proceedings of The 16th Workshop on Logic-based methods in Programming Environments (Vanhoof, W. and Hernández, S., eds.), pp. 1-12, 2006.
  http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=42268

- R. Tronçon, and G. Janssens, *Analyzing and debugging ILP data mining query execution*, Proceedings of the Sixth International Workshop on Automated Debugging, AADEBUG 2005 (Jeffery, C. and Choi, J., Lencevicius, R., eds.), pp. 105-109, 2005.
  http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=41742

- R. Tronçon, G. Janssens, and H. Vandecasteele, *Fast query evaluation with (lazy) control flow compilation*, Logic Programming, 20th International Conference, ICLP 2004, Proceedings (Demoen, B. and Lifschitz, V., eds.), vol 3132, Lecture Notes in Computer Science, pp. 240-253, 2004.
  http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=41198

- W. Vanhoof, R. Tronçon, and M. Bruynooghe, *A fixed point semantics for logic programs extended with cuts*, Logic Based Program Synthesis and

Transformation, LOPSTR 2002, Revised Selected Papers (Leuschel, M., ed.), vol 2664, Lecture Notes in Computer Science, pp. 238-257, 2003.
http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=40744

- R. Tronçon, G. Janssens, and B. Demoen, *Alternatives for compile & run in the WAM*, Proceedings of CICLOPS 2003: Colloquium on Implementation of Constraint and LOgic Programming Systems (Lopes, R. and Ferreira, M., eds.), pp. 45-58, 2003.
http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=41065

- R. Tronçon, H. Vandecasteele, J. Struyf, B. Demoen, and G. Janssens, *Query optimization: Combining query packs and the once-tranformation*, Inductive Logic Programming, 13th International Conference, ILP 2003, Szeged, Hungary, Short Presentations (Horvath, T. and Yamamoto, A., eds.), pp. 105-115, 2003.
http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=40938

- R. Tronçon, M. Bruynooghe, G. Janssens, and F. Catthoor, *Storage size reduction by in-place mapping of arrays*, Verification, Model Checking and Abstract Interpretation, Third Int. Workshop, VMCAI 2002, Revised Papers (Cortesi, A., ed.), vol 2294, Lecture Notes in Computer Science, pp. 167-181, 2002.
http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=38268

- R. Tronçon, M. Bruynooghe, G. Janssens, and F. Catthoor, *Storage size reduction by in-place mapping of arrays*, VMCAI 2002, Third International Workshop on Verification, Model Checking and Abstract Interpretation, Pre-Proceedings (Cortesi, A., ed.), pp. 1-12, 2002.
http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=37393

Contributions at international conferences, not published or only as abstract

- R. Tronçon, *Speeding up inductive logic programming queries*, ESSES PhD Student Meeting at PLI2003, ESSES, Uppsala, Sweden, August 26-27, 2003.
http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=40887

- R. Tronçon, *Towards more performant ILP data mining engines*, ESSES PhD Symposium, ESSES, Madrid, Spain, September 15-21, 2002.
http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=40877

Contributions at other conferences, not published or only as abstract

- R. Tronçon, *Storage Size Reduction in VLSI Design*, PACT-Symposium, Oostende, Belgium, September 3-4, 2001, PACT-WOG.
  http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=36197

Technical reports

- R. Tronçon, H. Vandecasteele, J. Struyf, B. Demoen, and G. Janssens, *An execution mechanism for combining query packs and once-transformations*, Department of Computer Science, K.U.Leuven, Report CW 362, Leuven, Belgium, October, 2003.
  http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW362.abs.html

Books

- B. Demoen, P. Nguyen, T. Schrijvers, and R. Tronçon, *The first 10 Prolog programming contests*, ISBN 9090197826, 2005.
  http://www.cs.kuleuven.be/~dtai/ppcbook

Biography

Remko Tronçon was born January 21, 1980 in Braine l'Alleud (Belgium). After finishing high school at the Sint-Jorisinstituut in Brussels in 1997, he started studying computer science at the Katholieke Universiteit Leuven. In 2001, he graduated Magna Cum Laude and received his Master's degree in Informatics. His Master's thesis, titled 'Storage Size Reduction in VLSI Design' was supervised by Maurice Bruynooghe.

In 2001, he joined the Declarative Languages and Artificial Intelligence group of the Katholieke Universiteit Leuven, and started a Ph.D. funded by a research grant from the university. In 2002, he received a personal Ph.D. grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (I.W.T.), on the basis of a proposal titled 'Towards more performant ILP data mining engines'. His Ph.D. research was supervised by Bart Demoen and Gerda Janssens.

Nederlandse Samenvatting

Technieken voor efficiëntere ILP Data Mining systemen

1 Inleiding

Data Mining[11] is een recente wetenschap die elementen van statistiek, databanken en artificiële intelligentie combineert. Data mining tracht 'regels' (ook *hypothesen* of *queries* genoemd) te vinden die niet-triviale verbanden, patronen of eigenschappen van grote hoeveelheden data omschrijven. Dergelijke hypothesen moeten bijdragen tot een beter begrip van de gegevens, en vormen aldus nieuwe kennis. Verschillende technieken kunnen gebruikt worden om aan data mining (of, meer algemeen, *knowledge discovery*) te doen. *Inductief logisch programmeren*, of kortweg ILP [16], is een data mining techniek die gebaseerd is op eerste-orde logica en *logisch programmeren*. Omwille van zijn logische basis is ILP een zeer natuurlijke maar krachtige techniek, in vergelijking met de andere gangbare data mining aanpakken.

Niettegenstaande de kracht van het formalisme en de reeds behaalde successen, kampt de techniek met een aantal problemen. Aangezien de af te leiden hypothesen willekeurige eerste-orde logische formules kunnen zijn, is de zoekruimte naar een geschikte hypothese in principe oneindig groot. Dit legt zware eisen op aan de efficiëntie van de het systeem onderliggend aan de ILP algoritmen.

In dit werk ontwikkelen we verschillende technieken die de efficiëntie van ILP systemen verbeteren:

- In het verleden werden verschillende query-transformaties ontwikkeld die de uitvoeringstijd van ILP algoritmen verbeteren. Één van de meest doeltreffende transformaties is de *once-transformatie*. Onafhankelijk van deze transformaties werd het *query-pack* uitvoeringsmechanisme ontwikkeld om de uitvoering van verzamelingen gelijkaardige queries te optimaliseren. Beide onafhankelijke technieken zorgen voor significante verbeteringen in efficiëntie. Het doel van onze eerste bijdrage is om een nieuw uitvoeringsmechanisme te ontwikkelen dat de once-transformatie en query-packs integreert, met als doel de voordelen van beide aanpakken te verkrijgen. Hiertoe voeren we de notie van een *adpack* in.

- Omdat queries in ILP moeten meerdere malen moeten uitgevoerd wor-

den, is de klassieke aanpak om een query eerst naar een efficiëntere versie te compileren, en deze versie vervolgens uit te voeren. Desondanks het feit dat deze aanpak significante verbeteringen oplevert, blijkt uit experimenten dat de compilatiestap het grootste aandeel heeft in query-evaluatietijd. Als tweede bijdrage ontwikkelen we een geoptimaliseerd query-interpretatieschema, hetwelk geen compilatiestap vereist om queries efficiënt uit te voeren.

- Als derde bijdrage ontwikkelen *control-flow compilatie*, een flexibeler alternatief voor de compileren-en-uitvoeren aanpak. Deze techniek combineert een eenvoudige compilatiestap met een efficiënte uitvoering. Naast een betere performantie laat deze techniek eveneens toe om query-pack uitvoering te ondersteunen. De flexibiliteit van dit nieuwe compilatieschema laat toe om een luie variant te ontwikkelen, die delen van queries slechts compileert wanneer deze gebruikt worden, wat de totale evaluatietijd nog meer verbetert.

Optimalisaties zoals diegenen die hierboven beschreven zijn gebeuren typisch op een laag niveau, wat het moeilijk maakt om fouten in de implementatie van deze technieken op te sporen. Het debuggen van deze uitvoeringsmechanismen worden daarenboven bemoeilijkt door verschillende andere factoren, zoals niet-determinisme van ILP algoritmen. Anderzijds beïnvloeden deze factoren ook de haalbaarheid van het uitvoeren van analyses van ILP uitvoering. In onze vierde bijdrage beschrijven we een algoritme-onafhankelijke manier om automatisch te debuggen en analyses van ILP uitvoering te doen.

Onze laatste bijdrage bestaat uit het onderzoeken van het inruilen van geheugen voor uitvoeringssnelheid. We bespreken bestaande technieken om dit te verwezenlijken op verschillende niveaus van een ILP algoritme, en maken een qualitatieve vergelijking met andere technieken die geen geheugen opofferen voor uitvoering.

Deze tekst is als volgt gestructureerd: in Sectie 2 geven we een korte beschrijving van de begrippen en achtergrond waarop in deze tekst gesteund worden; in Sectie 3 bespreken we *adpacks*, een combinatie van query-packs en de once-transformatie; Sectie 4 beschrijft de ingebedde meta-call en control-flow compilatie met zijn uitbreidingen als alternatief voor de klassieke compileren-en-uitvoeren aanpak; Sectie 5 handelt over een algoritme- en systeemonafhankelijke aanpak voor het analyseren en debuggen van ILP algoritmen; Sectie 6 bespreekt technieken die geheugen inruilen voor betere uitvoeringstijden; tenslotte concluderen we in Sectie 7.

2 Inleidende begrippen en achtergrond

Logisch programmeren is een programmeerparadigma, gebaseerd op eerste-orde logica. Een logisch programma bestaat uit een aantal relaties (of predikaten) en feiten, en wordt uitgevoerd door queries uit te voeren tegen het programma.

De bekendste logische programmeertaal is Prolog, wat de taal zal zijn die we doorheen dit werk zullen gebruiken. Implementaties van Prolog zijn typisch gebaseerd op de *Warren Abstract Machine* (kortweg WAM), een ontwerp van een virtuele machine met instructies die gericht zijn naar het uitvoeringsmechanisme van logische queries. Queries worden gecompileerd naar deze efficiënte instructies, dewelke vervolgens uitgevoerd worden in de virtuele machine.

Inductief logisch programmeren (kortweg ILP) is een data mining techniek die gebaseerd is op logisch programmeren. Het doel van een ILP algoritme is om een hypothese (of verzameling hypothesen) te vinden die de gegevens uit een gegeven dataset best omschrijft. De hypothese die de gegevens beschrijft is een eerste-orde logische formule. De gegevens bestaan uit een groot aantal *voorbeelden*, waarbij elk voorbeeld voorgesteld wordt door een logisch programma. De *achtergrondkennis* beschrijft kennis over het probleemdomein door middel van predikaatdefinities, en vormt zo een verzameling predikaten die gelden voor elk voorbeeld uit de dataset. ILP algoritmen gebruiken een genereer-en-test aanpak om te zoeken naar geschikte hypothesen: tijdens elke iteratie van het algoritme worden een aantal kandidaatshypothesen gegenereerd, dewelke geëvalueerd worden op de dataset door ze als queries uit te voeren. Op basis van deze evaluatie selecteert het algoritme de beste hypothese(n), breidt deze in de volgende iteratie uit (door delen toe te voegen aan de hypothese(n)), en evalueert de uitbreidingen. Dit proces herhaalt zich tot een bevredigende hypothese gevonden is. Welke hypothesen geselecteerd worden, en wanneer een hypothese bevredigend is om te eindigen, hangt af van algoritme tot algoritme.

Concrete voorbeelden van ILP algoritmen zijn Tilde en Warmr. Tilde leert eerste-orde beslissingsbomen, terwijl Warmr frequente patronen uit gegevens afleidt. Een voorbeeld van een ILP systeem dat deze (en andere) algoritmen implementeert is ACE [1], hetwelk gebouwd is bovenop hipP, een efficiënte Prolog implementatie met speciale voorzieningen voor ILP. Voorbeelden van bekende datasets zijn Mutagenesis [19], Carcinogenesis [18], en Bongard [8]. Naast deze datasets gebruiken we ook nog de grotere HIV [15] dataset.

Omdat tijdens elke iteratie van een ILP algoritme een groot aantal queries moet uitgevoerd worden op een groot aantal voorbeelden uit de dataset, is de evaluatiestap van het algoritme typisch de flessenhals van ILP uitvoering. Daarom werd er in het verleden reeds aandacht geschonken aan het optimaliseren van query-uitvoering binnen ILP. We bespreken hier twee belangrijke optimalisatietechnieken.

Een query kan meestal opgedeeld worden in verschillende onafhankelijke stukken die elkaars uitvoering niet beïnvloeden. Dit betekent dat, wanneer een bepaald deel van de query faalt, er mogelijk tevergeefs gebacktracked wordt over alternatieven van onafhankelijke delen. Om deze redundantie te vermijden werd de *once-transformatie* [6] ontwikkeld. Deze transformatie snijdt alternatieven van queries weg van zodra er geweten is dat deze geen invloed meer zullen hebben op de rest van de uitvoering van de query. Dit wegsnijden wordt gedaan door de onafhankelijke delen in te bedden in een `once/1` predikaat.

Aangezien de queries die uitgevoerd moeten worden verfijningen zijn van

een andere query, zal het begin (prefix) van de uit te voeren queries gemeenschappelijk zijn. Wanneer men de queries als groep beschouwt, kan men de uitvoering versnellen door er enerzijds voor te zorgen dat het gemeenschappelijke prefix slechts één keer uitgevoerd wordt, en dat er anderzijds niet meer over een query gebacktracked wordt wanneer die slaagt. Zulke groepen van queries noemt men *query-packs* [3]. Query-packs worden geïmplementeerd door speciale WAM-instructies, die gebruik maken van extra datastructuren die bijhouden welke delen van query-packs nog uitgevoerd dienen te worden.

3 Combineren van query-packs met de once-transformatie

3.1 Inleiding

Query-packs en de once-transformatie werden onafhankelijk van elkaar bestudeerd, en zorgen elk voor significante verbeteringen in uitvoeringstijd. Daarom lijkt het interessant om beiden te combineren. Het is echter niet triviaal om beide aanpakken in hun oorspronkelijke vorm te combineren: door de herordening die de once-transformatie op elke query apart uitvoert kan het zijn dat de verzameling queries minder gemeenschappelijke delen zullen hebben, en dat het effect van uitvoering in query-packs voor een stuk ongedaan gemaakt wordt. In deze sectie combineren we de once-transformatie met query-packs, en voeren hiertoe *adpacks* in, een nieuwe variant van query-packs.

3.2 ADPacks

ADPacks hebben dezelfde vorm van klassieke query-packs, maar bevatten speciale controlestructuren om de `once` te integreren, namelijk `activate`/`deactivate` paren. Zulk een paar duidt het begin en het einde van een `once` in de oorspronkelijke query aan.

Intuïtief betekent het voorkomen van een `deactivate/1` dat alle resterende alternatieven van doelen die tussen de `deactivate` en zijn overeenkomstige `activate` liggen niet meer nuttig zijn voor het al dan niet succesvol uitvoeren van de doelen die na de `deactivate` komen. Een `deactivate` zal daarom de tak waar hij op ligt *tijdelijk* uit de pack verwijderen, tot op het ogenblik dat de uitvoering alternatieven probeert die voor de overeenkomstige `activate` liggen.

3.3 Uitvoering

De uitvoering van een adpack (net zoals de uitvoering van een gewone query of een query-pack) kan opgesplitst worden in twee fazen: de voorwaartse uitvoering, en het backtracken (wanneer een bepaald doel faalt, en er een nieuw alternatief voor een eerder doel geprobeerd moet worden). Tijdens de uitvoering houden we van elke vertakking van een adpack-or bij of die open of gesloten is, en of die succesvol of nog niet succesvol is. Intuïtief betekent ‘succes’ dat

een tak nooit meer geprobeerd moet worden, en 'gesloten' dat een tak tijdelijk niet geprobeerd moet worden. Het belangrijkste deel van de uitvoering is het backtracken, waarbij er moet bepaald worden welke delen van de query irrelevant zijn om naar te backtracken, en geen nieuwe oplossingen zullen bieden. Dit wordt bepaald aan de hand van gegevens over de structuur van de adpack, en extra informatie bijgehouden tijdens de voorwaartse fase.

Eerder onderzoek [3] wijst uit dat, om een gespecialiseerd uitvoeringsmechanisme zoals dat van query-packs ten volle te kunnen benutten, het geïmplementeerd moet worden in de kern van het systeem zelf. Net zoals bij query-packs compileren we daarom adpacks naar gespecialiseerde WAM instructies. Deze instructies maken gebruik van nieuwe datastructuren die ontworpen zijn voor efficiënte uitvoering van adpacks.

3.4 Transformatie

Alvorens de queries gegenereerd door een ILP algoritme uitgevoerd kunnen worden met het adpack uitvoeringsmechanisme, dient de verzameling queries omgezet te worden in een adpack structuur. We beschouwen twee alternatieven om dit te bekomen:

- De *query-gebaseerde adpack transformatie* itereert over de verzameling van queries, voert de once-transformatie op elke query uit, en integreert de resulterende query in een accumulerende adpack.

- De *pack-gebaseerd adpack transformatie* vertrekt van een query-pack, en voert de once-transformatie rechtstreeks op de pack structuur uit. De motivatie achter deze aanpak is dat het uitvoeren van de once-transformatie op elke query afzonderlijk voor redundantie zorgt (omwille van de gelijkaardigheid van de queries).

3.5 Evaluatie

Als eerste experiment vergelijken we de query-gebaseerde adpack-transformatie met de pack-gebaseerde transformatie. Beide transformaties werden in het ACE systeem geïmplementeerd, en werden toegepast in Tilde. Metingen op de Mutagenesis, Carcinogenesis en Bongard datasets tonen aan dat, ondanks het herhaaldelijk transformeren van dezelfde delen van queries, de query-gebaseerde transformatie beter presteert dan de pack-gebaseerde transformatie. De zwaarste factor in de pack-gebaseerde adpack transformatie is het herhaaldelijk transformeren van een verzameling queries naar een query-pack.

Om het adpack-uitvoeringsmechanisme te evalueren voeren we Tilde uit op de drie bovenvermelde datasets, en laten hierbij de complexiteit van de queries variëren. De adpack uitvoeringstijd is effectief kleiner dan query-pack uitvoering en uitvoering van once-getransformeerde queries, maar ligt tevens dicht bij de uitvoeringstijd van query-packs. De grootste winst werd behaald bij de Carcinogenesis dataset ($\pm$10 keer sneller dan query packs en 40 keer sneller dan de

once-transformatie). De totale tijd nodig om een verzameling queries te transformeren naar een adpack is echter zo hoog dat in een aantal gevallen de totale uitvoeringstijd van TILDE trager is dan deze van query-packs.

3.6 Conclusies

Experimenten tonen aan dat de evaluatie van queries voordeel heeft bij het nieuwe adpacks uitvoeringsmechanisme, in vergelijking met enkel query-packs of enkel de once-transformatie toe te passen. De uitvoering van adpacks was in de experimenten tot een factor 10 sneller dan query-packs, en tot 50 keer sneller dan de once-transformatie. De complexiteit van de adpack-transformatie weegt echter zwaar door op de totale uitvoeringstijd. Een potentiële aanpak om dit te voorkomen is de pack-gebaseerde transformatie rechtstreeks in de kern van het systeem te implementeren.

Andere technieken die overtollig backtracken proberen te vermijden (zoals *backjumping* [12] en *intelligent backtracking* [4]) maken gebruik van dynamische tests om delen van de zoekruimte te vermijden. Hoewel de extra kost van dergelijke tests niet geschikt zijn voor de ILP context, zou het interessant zijn om statische versies van deze uitvoeringsmechanismen te ontwikkelen in de context van query-uitvoering.

4 Alternatieven voor compileren-en-uitvoeren

4.1 Motivatie

Wanneer een verzameling queries dynamisch uitgevoerd dient te worden op een verzameling gegevens, zijn er minstens twee aanpakken om dit te doen: ofwel worden de queries worden onmiddellijk uitgevoerd d.m.v. de meta-call, die de query interpreteert, ofwel worden ze eerst gecompileerd naar efficiënte WAM instructies. Het grote nadeel van de eerste aanpak is dat deze dynamische tests moet doen bij elke uitvoering van de query, terwijl de tweede aanpak een groot deel van deze tests op voorhand doet tijdens de compilatiestap. Aangezien queries in ILP een groot aantal keer uitgevoerd dienen te worden, werd de laatste aanpak als de beste bevonden in het verleden [3]. Niet alleen wordt de initiële compilatiestap van deze aanpak meer dan gecompenseerd door de uitvoering van de snellere code, speciale uitvoeringsmechanismen zoals query-packs en adpacks hebben deze compilatiestap nodig om efficiënt te kunnen uitvoeren. Als we echter de totale evaluatietijd van query-packs bekijken, zien we dat de compilatietijd relatief veel tijd in beslag neemt, en soms zelfs even veel als de uitvoering van de query zelf. Hierdoor kan de vraag gesteld worden of de hoeveelheid code die gecompileerd moet worden kan verminderd worden, of de compilatiestap kan vereenvoudigd worden, of zelfs gewoon vermeden worden.

In dit deel bestuderen we alternatieven voor compileren-en-uitvoeren, en trachten we het aandeel van compilatie in het evaluatieproces te verlagen.

4.2 (Ingebedde) meta-call

Het grote nadeel van de meta-call is de dynamische overhead die deze introduceert door tests te doen tijdens elke uitvoering van een query. We kunnen echter een experiment opstellen waarbij gecompileerde code veel trager uitvoert dan de meta-call, wat verklaard kan worden door het feit dat gecompileerde code steeds de argumenten van zijn op te roepen doelen moet construeren alvorens de oproep te doen, terwijl bij meta-call de argumenten reeds op voorhand geconstrueerd zijn. Dit voordeel motiveert het onderzoek naar het optimaliseren van de meta-call.

Gebruik makend van de veronderstelling dat queries rechts-lineair zijn kunnen we een kleine meta-vertolker schrijven die gespecialiseerd is voor het uitvoeren van conjunctieve queries. Deze meta-vertolker voert essentieel de gewone meta-call uit, behalve dat deze extra tests vermijdt door veronderstellingen te maken over de vorm van de query die uitgevoerd wordt. Experimenten tonen aan dat deze vertolker inderdaad een versnelling oplevert t.o.v. de klassieke meta-call, maar toch nog steeds veel trager is dan gecompileerde code. We kunnen deze meta-vertolker echter rechtstreeks in het systeem implementeren, wat ons een *ingebedde meta-call* oplevert.

Wanneer we compileren-en-uitvoeren en de meta-call vergelijken met de ingebedde meta-call, constateren we dat de ingebedde meta-call de uitvoeringstijd van gecompileerde code benadert. Gecompileerde code is echter in een aantal gevallen nog steeds sneller dan de ingebedde meta-call. De ingebouwde meta-call aanpak heeft bovendien nog een aantal andere nadelen. gecompileerde code kan gebruik maken van ingebouwde operaties voor arithmetiek; de ingebouwde meta-call uitbreiden om deze ingebouwde operaties uit te voeren vereist het uitbreiden van de ingebedde implementatie, wat niet alleen hinderlijk is, maar de uitvoering ook zal vertragen. Een ander probleem is dat het zeer moeilijk is om speciale uitvoeringsmechanismen zoals query-packs en adpacks te implementeren met behulp van meta-call alleen. Om ze efficiënt te kunnen uitvoeren, moet de structuur van packs moet op voorhand gekend zijn, wat betekent dat het niet mogelijk is om query-packs te implementeren met een meta-call die geen voorverwerkingsstap heeft. Bovendien vereist het integreren van query-packs met de ingebedde meta-call dat heel het uitvoeringsmechanisme volledig terug geïmplementeerd moet worden, rechtstreeks in het systeem. Deze eigenschappen maakt de ingebedde meta-call minder aantrekkelijk voor verdere uitwerking in de context van ILP.

4.3 Control-flow compilatie

Het grote voordeel van de meta-call methode t.o.v. compilatie en uitvoering is het feit dat meta-call geen extra (dure) emulatorcycli nodig heeft om de oproep op te bouwen alvorens hem uit te voeren; dit gebeurt allemaal in dezelfde emulatorcyclus. Daarom lijkt het interessant om dit voordeel te combineren met een simpele vorm van compilatie die geen complexe taken als registerallocatie moet uitvoeren.

We introduceren *control-flow compilatie*, een hybride vorm van compilatie en meta-call. Het grote verschil met gewone compilatie zijn de gegenereerde instructies voor het opzetten en oproepen van doelen. Terwijl de gewone compilatie `put` en `call` instructies hiervoor genereert, wordt in control-flow compilatie een nieuwe `cf_call` instructie gegenereerd. Deze instructie heeft als argument een referentie naar een term in het geheugen die opgeroepen moet worden. Dit betekent dat de enige instructies die overblijven instructies zijn die gerelateerd zijn aan de 'control-flow' (proberen van alternatieven) van de query, en de nieuwe `cf_call` instructies. Deze nieuwe instructies zorgen ervoor dat de emulatorcycli die vroeger nodig waren om de argumenten bij oproepen in gecompileerde code op te zetten niet langer nodig zijn. Bovendien is de compilatie naar control-flow gecompileerde code zeer eenvoudig (aangezien er niet langer aan registerallocatie gedaan moet worden), en wordt het eenvoudig om achteraf code toe te voegen aan bestaande gecompileerde code, wat interessant is voor luie compilatie. Net zoals bij klassiek gecompileerde code kunnen (dezelfde) ingebouwde instructies gebruikt worden voor veel voorkomende operaties.

In tegenstelling tot klassiek gecompileerde code kan control-flow gecompileerde code niet op zichzelf bestaan, aangezien het directe referenties naar het geheugen bevat. De garbage collector van het systeem moet hiermee rekening houden door alle termen van een query levend te houden zolang de overeenkomstige gecompileerde code levend is, en door de nodige referenties in de code aan te passen wanneer termen van plaats veranderen in het geheugen. Een ander feit dat in rekening gehouden moet worden is dat control-flow gecompileerde code termen in het geheugen aan waarden kan binden, en er bijgevolg geen recursieve oproepen kunnen gebeuren. Aangezien recursieve oproepen niet voorkomen in queries, vormt dit echter geen probleem in praktijk.

Het control-flow compilatieschema werd eerst en vooral geëvalueerd door artificieel gegenereerde queries te compileren en uit te voeren, waarbij queries volgens verschillende parameters (het aantal doelen per query, het aantal vertakkingen per disjunctie, ...) gegenereerd werden. Binnen deze artificiële context werd de compilatietijd telkens met minstens één grootteorde gereduceerd. De uitvoeringstijd van de control-flow gecompileerde code was bovendien ook steeds iets beter dan de uitvoering van de klassiek gecompileerde tijd (tot dubbel zo snel).

Vervolgens werd control-flow compilatie getest binnen een meer praktische ILP toepassing. Het Tilde algoritme werd toegepast op de Mutagenesis, Carcinogenesis, Bongard, en HIV datasets. Hier was control-flow compilatie 5 tot 8 keer sneller dan klassieke compilatie, en bleef de uitvoeringstijd van de gegenereerde code in beide aanpakken ongeveer even groot. Globaal gezien is control-flow compilatie dus de beste aanpak.

4.4 Luie control-flow compilatie

Zoals reeds eerder aangehaald is compilatie een complexe taak die relatief veel tijd kost. Control-flow compilatie reduceert deze compilatietijd al sterk, maar desondanks blijft het nog steeds een belangrijk aandeel hebben in de totale tijd

nodig voor het evalueren van queries: afhankelijk van de dataset kan control-flow compilatie in praktijk tot een kwart van de totale tijd in beslag nemen. Wanneer we de uitvoering van query-packs bekijken, zien we dat er regelmatig bepaalde doelen van query-packs falen op elk voorbeeld. Dit betekent dat het deel dat op zo een doel volgt nooit uitgevoerd wordt, en dus tevergeefs gecompileerd werd. In dit deel reduceren we de totale compilatietijd nog meer door enkel stukken van de query te compileren die effectief uitgevoerd worden. Hiervoor gebruiken we *luie compilatie* [2], een vorm van *just-in-time* (JIT) compilatie. Deze luie compilatie zorgt er voor dat de compiler pas opgeroepen wordt wanneer er een stuk van de query bereikt wordt dat nog niet gecompileerd geweest is. Omwille van zijn flexibiliteit en snelheid gebruiken we hiervoor de control-flow compilatie. We beperken ons tot luie compilatie van conjunctieve queries. Disjuncties worden in Sectie 4.5 behandeld als query packs.

Luie compilatie zoals we die beschreven hebben compileert doel per doel. Hierdoor wordt na de uitvoering van elk doel steeds terug naar de compiler overgeschakeld. We hebben ook andere granulariteiten geïmplementeerd en geëvalueerd. Experimenten tonen aan dat de aanpakken met grotere granulariteit inderdaad veel beter presteren dan per doel te compileren. Onderling verschillen de grofkorrelige varianten echter niet zo veel.

4.5 Luie control-flow compilatie voor query-packs

Tot nu toe hebben we ons beperkt tot luie compilatie van queries met disjuncties. In praktijk zijn het echter de query-packs die interessant zijn. Het essentiële verschil tussen gewone disjuncties en query-packs zijn de instructies die gegenereerd worden voor het uitvoeren van de disjunctie. Daarom is het uitbreiden van de control-flow compiler om query-packs te ondersteunen geen al te zware taak. Het zwaarste werk is het aanpassen van de datastructuren die door de uitvoering gebruikt worden, aangezien die er van uit gaan dat de totale structuur op voorhand gekend is (wat niet langer het geval is).

Ter evaluatie vergelijken we eerst luie compilatie met gewone control-flow compilatie op een reeks artificieel gegenereerde queries van dewelke de doelen steeds slagen. Aangezien alle doelen slagen, meet dit experiment enkel de extra kost die geïntroduceerd wordt door de luiheid. De effectieve metingen tonen aan dat de uitvoeringstijd ongeveer gelijk is, en dat de extra overhead van luie compilatie voor gewone queries dus verwaarloosbaar is. Na de artificiële experimenten werd de aanpak ook geëvalueerd op concrete datasets. Het belangrijkste experiment hiervan is het uitvoeren van Tilde op de Mutagenesis, Carcinogenesis, en Bongard datasets. Wanneer we control-flow compilatie met klassieke compilatie vergelijken zien we dat de control-flow compilatie en de uitvoering van zijn gegenereerde code tot een factor 5 sneller is dan klassieke compilatie Hoewel luie compilatie sneller is dan control-flow compilatie, is het verschil tussen deze twee aanpakken minder uitgesproken.

4.6 Conclusies

We concluderen dat control-flow compilatie voor query-packs een aanzienlijke verbetering is tegenover klassieke compilatie van query-packs. De luie variant van control-flow compilatie is in de meeste gevallen sneller dan gewone control-flow compilatie, maar het verschil is niet altijd zo groot. We verwachten dat dit verschil groter wordt wanneer de huidige garbage collector uitgebreid wordt om control-flow gecompileerde code te ondersteunen.

5 Query uitvoering analyseren en debuggen

Het ontwikkelen van nieuwe uitvoeringsmechanismen in ILP speelt zich vooral af in de motor gebruikt door het ILP algoritme. Deze geoptimaliseerde uitvoeringsstrategieën vereisen typisch een laag-niveau implementatie om significante verbeteringen voort te brengen. Bijvoorbeeld, het adpack uitvoeringsmechanisme uit Sectie 3 vereist nieuwe WAM instructies, gepaard met bijhorende datastructuren. De ingebedde meta-call en control-flow compilatieschemas uit Sectie 4 maken eveneens gebruik van nieuwe WAM instructies, geïmplementeerd in de kern van het systeem. Het lage niveau van deze implementaties maakt het ontdekken van bugs in deze uitvoeringsmechanismen zeer moeilijk. Hoewel debuggen nog haalbaar is voor kleinere testprogramma's, komen vele bugs pas aan de oppervlakte wanneer een ILP algoritme op een grote dataset uitgevoerd wordt. Verschillende factoren maken debuggen in dit geval moeilijk: de omvang van de broncode van het ILP systeem maakt het moeilijk om standaard traceertechnieken te gebruiken; de complexiteit van het ILP probleem, zorgt ervoor dat het lang kan duren eer een fout zich manifesteert; de complexiteit van de hypothese-generatiefase bij algoritmen als Warmr vertragen het geheel nog meer, desondanks het feit dat er weinig tijd in de uitvoering zelf gebeuren; bij niet-deterministische ILP algoritmen kan het ogenblik wanneer een fout zich manifesteert veranderen van uitvoering tot uitvoering.

Deze eigenschappen verhinderen niet enkel het opsporen van bugs, ze scheppen ook problemen voor het analyseren van query uitvoering. Bij het ontwikkelen van nieuwe uitvoeringsmechanismen kunnen dergelijke analysen nuttige informatie verschaffen om verdere ontwikkelingen te sturen, om flessenhalzen te detecteren, en om uitvoeringsmechanismen te vergelijken. Hierbij doen zich echter gelijkaardige problemen voor als bij het debuggen. Een bijkomend probleem is dat het onmogelijk is om uitvoeringsmechanismen van verschillende ILP motoren te vergelijken.

In het verleden werden *uitvoeringssporen* gebruikt om fouten in programma's te detecteren [9, 10] en om performantie van programma's te meten [13]. In dit deel werken we een spoorgebaseerde aanpak uit voor het debuggen en analyseren van ILP algoritmen, zonder de eigenlijke implementatie van de algoritmen te moeten aanpassen. Deze aanpak is algoritme- en motoronafhankelijk, en laat toe om snel en gemakkelijk te debuggen. We leggen bovendien uit hoe het debuggen kan vereenvoudigd worden door de uitvoering automatisch te beperken tot het

deel dat de fout veroorzaakt.

5.1 Verzamelen van uitvoeringsinformatie

Het doel van ILP motoroptimalisaties zoals beschreven in dit werk is de evaluatiestap van het ILP algoritme. De andere stappen die het algoritme karakteriseren (zoals het vinden van goede verfijningen) zijn niet belangrijk uit het oogpunt van een motor-ontwikkelaar. Desondanks zijn deze laatsten toch de stappen die het meeste code in beslag nemen in de implementatie van een ILP algoritme. We extraheren genoeg informatie van een ILP uitvoering om de evaluatiestap te kunnen reproduceren, zonder daarbij het ILP algoritme volledig te moeten uitvoeren. We hebben meer bepaald enkel de queries nodig die het algoritme uitvoert, en op welke voorbeelden deze uitgevoerd worden; hoe deze queries bekomen werden is onbelangrijk voor de uitvoering.

Om de nodige informatie te extraheren passen we de evaluatiestap aan zodat die al de geëvalueerde queries in een bestand opslaat, hetwelk we het *spoor* van het algoritme noemen. Dit spoor is onafhankelijk van het algoritme zelf, aangezien het enkel een lijst van queries bevat. Het spoor kan nu in een eenvoudige spoor-simulator ingeladen worden, die de evaluatiestap volledig kan reproduceren. We bespreken in de volgende secties verschillende soorten spoorsimulatoren, die gebruikt worden voor het debuggen en analyseren van de evaluatiestap.

5.2 Debuggen van query uitvoering

Tijdens het ontwikkelen van optimalisaties voor query evaluatie worden verschillende uitvoeringsmechanismen onderzocht. In het geval dat een nieuw uitvoeringsmechanisme de uiteindelijke resultaten van een ILP algoritme hoort te bewaren, kunnen inconsistenties in het nieuwe mechanisme ontdekt worden door het ILP algoritme uit te voeren met en zonder de optimalisatie, en de resultaten te vergelijken. Deze vergelijking kan echter enkel zeggen *of* er een fout zit in het algoritme, maar niet *waar*. Om de exacte plaats te vinden van de fout moet de debugger van de taal van het ILP algoritme (bv. Prolog) on de debugger van de taal van de motor (bv. C) gebruikt worden, wat bemoeilijkt wordt door de grootte van het ILP systeem. Een ander probleem is dat deze aanpak enkel werkt als het algoritme volledig deterministisch is. Het gebruik van uitvoeringssporen te gebruiken om te debuggen lost veel van deze problemen op: de uitvoering van een spoor is deterministisch, en de broncode van een simulator is zo klein dat het debugging proces kan toegespitst worden op de optimalisaties van de motor zelf.

Door een spoor te laten uitvoeren door een spoorsimulator zowel met als zonder een bepaalde optimalisatie, kan gedetecteerd worden welk stuk van de uitvoering een inconsistent resultaat oplevert: door het al dan niet slagen of falen van elke query in het spoor te onthouden, en het resultaat te vergelijken met een uitvoering van het spoor zonder de optimalisatie kan gezien worden welke query een fout veroorzaakt. Door de grootte van het spoor kan het echter

nog zijn dat een groot deel van de uitvoering moet doorlopen worden eer de fout zich manifesteert. Aangezien het spoor alle informatie bevat om de uitvoering te simuleren, kunnen we het lokaliseren van een fout in het spoor in een *data-slicing* [5] probleem omvormen. Het doel van data-slicing is om een verzameling gegevens te reduceren tot de kleinste deelverzameling die een fout doet voorkomen. Hoewel de standaard techniek voor data-slicing inhoudt om de verzameling gegevens steeds te halveren, en enkel de helft te houden die de fout veroorzaakt, kampt deze aanpak met het probleem dat binair zoeken soms te grofkorrelig is om een spoor te reduceren. *Delta-debuggen* [22] is een automatische data-slicing techniek, die dit probleem vermijdt. Dit algoritme past de granulariteit aan tijdens zijn uitvoering, en maakt combinaties van deelverzamelingen van gegevens, teneinde een zo minimaal mogelijk spoor te bekomen.

De delta-debug aanpak werd gebruikt tijdens het ontwikkelen van nieuwe uitvoeringsmechanismen in hipP. Hiertoe werden twee soorten delta-debuggers gemaakt, die beiden een spoor gegenereerd door een ILP algoritme reduceren tot een kleiner spoor dat een fout doet manifesteren. De twee delta debuggers verschillen in de test die ze uitvoeren om te bepalen of er al dan niet een fout in het spoor zit. De meest eenvoudige debugger controleert of de uitvoering van het spoor succesvol eindigt. Dit is nuttig voor het opsporen van bugs die de motor doen falen (zoals segmentatie-fouten). De tweede delta-debugger vergelijkt de uitvoering van het spoor tussen twee motoren (één mét, en één zonder de optimalisatie), en controleert of de queries uit beide sporen overeenkomstige resultaten bekomen. Beide delta-debuggers kunnen ingesteld worden om fijn- of grofkorrelig naar fouten te zoeken.

5.3 Query analyse

Om een inzicht te krijgen in het gedrag van ILP algoritmen in termen van query uitvoering, is het interessant om de karakteristieken van queries en hun uitvoering te kennen. Een ILP systeem aanpassen om alle interessante informatie te verzamelen is een zware taak. In dit deel bespreken we het analyseren van query uitvoering door middel van sporen. Structurele query-analyse biedt een inzicht in ILP algoritmen door de structurele eigenschappen van gegenereerde queries te onderzoeken. Dit kan onder andere helpen om de performantie van uitvoeringsmechanismen te verklaren. Dynamisch analyseren van queries biedt informatie over het run-time gedrag van query uitvoeringsmechanismen.

Een structurele query-analyse moet in principe door elke query van het spoor lopen, en tijdens dit proces verschillende statistieken bijhouden. Dit kan bekomen worden door een eenvoudige spoorsimulator uit te breiden, zodat deze de nodige informatie bijhoudt. Wanneer het aantal statistieken oploopt wordt dergelijke simulator echter moeilijk te onderhouden. Dit kan vermeden worden door een gebeurtenis-gebaseerde analyse te ontwikkelen. We definiëren een verzameling gebeurtenissen in een spoor, zoals bv. het begin van een query, het einde van een query, en het begin van een iteratie. Statistieken kunnen dan gedefinieerd worden in termen van de acties die genomen moeten worden bij het optreden van een gebeurtenis. Deze acties doen bewerkingen op een lijst

van tellers, die onderling doorgegeven wordt. De spoorsimulator moet dan enkel het spoor doorlopen, en bij elke gebeurtenis alle acties ondernemen die opgelegd worden door de statistieken. Hierdoor kan de spoorsimulator erg eenvoudig gehouden worden.

Informatie over de uitvoering van queries kan gebruikt worden voor allerlei doelen: het gedrag van uitvoeringsmechanismen voorspellen of verklaren, vergelijken van uitvoeringsmechanismen, ontdekken van flessenhalzen tijdens query uitvoering, ...Spoorgebaseerde dynamische analyse van uitvoering voert alle queries van een spoor uit, nadat elke query getransformeerd werd zodat die dynamische informatie opslaat. Interessante informatie is bijvoorbeeld het aantal keer dat een bepaald doel uitgevoerd werd.

Zowel de structurele als de dynamische analyse werd geïmplementeerd en gebruikt tijdens de ontwikkeling van nieuwe uitvoeringsmechanismen in hipP. Informatie uit beide analysen wordt samengebracht in één HTML rapport dat een overzicht geeft over de uitvoering van het ILP algoritme op de specifieke dataset.

5.4 Conclusies

In dit deel hebben we spoorgebaseerde aanpakken beschreven voor het debuggen van query-uitvoeringsmechanismen in ILP. Het gebruik van een spoorgebaseerde aanpak heeft verschillende voordelen in deze context:

- De werking van specifieke ILP algoritmen moet niet gekend zijn, aangezien de sporen algoritme-onafhankelijk zijn.
- Door de sporen aan te passen kan de uitvoering gereduceerd worden tot het deel van de uitvoering dat een fout veroorzaakt, zodat fouten heel snel opgespoord kunnen worden.
- Bij spoorgebaseerde uitvoering wordt er uitsluitend tijd gespendeerd aan de uitvoering van queries. Complexe generatiefasen van ILP algoritmen hebben dus geen invloed op de totale uitvoeringstijd van een spoor, zodat debuggen sneller kan gebeuren.
- Het is niet nodig om volledige kennis te hebben van de broncode van het ILP systeem.

Een spoorgebaseerde aanpak is ook nuttig in de context van het analyseren van query-uitvoering. Deze analyse biedt nuttige informatie over het aantal en de structuur van de queries. Dynamische analyse gebruikt transformeert queries alvorens ze uit te voeren, zodat er informatie over de uitvoering geregistreerd wordt tijdens de uitvoering zelf.

Deze aanpakken werden gebruikt tijdens de ontwikkeling van de verschillende technieken beschreven in dit werk. De delta-debugtechniek maakte het mogelijk om snel fouten op te sporen in de laag-niveau implementatie van adpacks (Sectie 3) en control-flow compilatie (Sectie 4). De analyse-technieken maakten het ook mogelijk om de tabelleringstechnieken uit Sectie 6 te evalueren.

Een toepassing van sporen die niet verkend werd in dit deel is visualisatie. Visualisaties van query-uitvoering kan helpen begrijpen wat er gebeurt in de evaluatiestap van ILP algoritmen. Deze aanpak kan bovendien uitgebreid worden om volledige ILP algoritmen zelf te visualiseren, zonder het ILP systeem drastisch aan te moeten passen.

6 Afwegen van tijd en ruimte

6.1 Inleiding

Omdat de kandidaatshypothesen gegenereerd door een ILP algoritme verfijningen zijn van queries uit vorige iteraties, is de uitvoering van queries gelijkaardig. Technieken zoals query-packs en adpacks vermijden redundantie in de uitvoering door de manier waarop queries uitgevoerd worden aan te passen. Een andere aanpak om redundantie te vermijden is door resultaten van de uitvoering op te slagen en te hergebruiken in latere stappen. In dit deel bespreken we het memoriseren en hergebruiken van berekende antwoorden op verschillende niveaus van ILP uitvoering.

Voor de technieken ontwikkeld in dit deel gebruiken we twee bestaande technieken: *specialisatie* en *tabelleren*.

Specialisatie is een transformatietechniek die, gegeven een programma en een deel van zijn invoer (de statische invoer), een nieuw programma genereert dat dezelfde uitvoer geeft als het oorspronkelijke programma, maar waar berekeningen afhankelijk van de statische invoer voorberekend zijn. Hierdoor voert het resulterende programma sneller uit dan het oorspronkelijke programma. Automatische specialisatie is een gekende optimalisatietechniek [14], waar in het verleden een variëteit aan aanpakken voor ontwikkeld werden. In de context van logisch programmeren zijn de meeste technieken gebaseerd op het top-down principe, waar informatie van een partieel geïnstantieerde query gepropageerd wordt door het programma. Een andere, query-onafhankelijke techniek is bottom-up specialisatie, waar informatie uit de feiten van een programma naar boven gepropageerd wordt.

Tabelleren is een uitvoeringstechniek waar de antwoorden van oproepen naar doelen onthouden worden, en bij volgende oproepen hergebruikt worden.

6.2 Memoriseren van achtergrondkennis

Wanneer de achtergrondkennis van een dataset complexe predikaten bevat, kan het uitvoeren van queries veel tijd in beslag nemen, aangezien deze predikaten herhaaldelijk uitgevoerd zullen worden. Daarom worden alle antwoorden voor deze predikaten op voorhand berekend voor elk voorbeeld. Deze ad-hoc manier van voorverwerking van een dataset is simpel en effectief, maar kan echter foute resultaten opleveren in de aanwezigheid van cuts in de predikaten.

Als alternatief voor het ad-hoc voorberekenen van antwoorden passen we specialisatie-technieken toe op de achtergrondkennis en de dataset. Aangezien de queries die gebruikt zullen worden niet op voorhand gekend zijn, is bottom-up specialisatie de interessantste aanpak. Elk voorbeeld uit de dataset wordt samen met de achtergrondkennis als logisch programma gespecialiseerd, en de nieuwe voorbeelden vormen de gespecialiseerde dataset. Het voordeel van deze aanpak is dat deze robuust is tegen de aanwezigheid van cuts. Om deze aanpak te evalueren hebben we een prototype van de semantiek uit [21] geïntegreerd met het raamwerk voor bottom-up specialisatie van [20]. Experimenten tonen echter aan dat een bottom-up gespecialiseerde dataset slechter presteert dan de originele dataset. De verklaring hiervoor is dat het aantal logische zinnen drastisch stijgt wanneer het aantal predikaten in de achtergrondkennis groter wordt. Een top-down gespecialiseerde versie van een dataset presteert beter, maar blijft trager dan een manueel gespecialiseerde dataset. Bovendien heeft het specialisatieproces snel te kampen met geheugenproblemen.

Als tweede alternatief onderzoeken we getabelleerde uitvoering van de achtergrondpredikaten. In plaats van een dataset voor te verwerken, worden de antwoorden van de achtergrondpredikaten getabelleerd tijdens hun uitvoering, waardoor ze slechts éénmalig berekend moeten worden. Aangezien er tabelleringsmethoden bestaan die de cut in rekening brengen, is deze techniek robuust voor onpure constructies in de predikaten. Het voordeel van deze aanpak is dat de oorspronkelijke dataset onveranderd kan blijven, en dat enkel de antwoorden onthouden worden van predikaten die effectief opgeroepen worden. Het nadeel is dat de antwoorden van deze predikaten bij elke uitvoering van een ILP algoritme één keer moeten berekend worden. Experimenten tonen aan dat getabelleerde uitvoering inderdaad significante verbeteringen oplevert. Manueel voorverwerkte datasets zijn echter nog steeds tot twee ordegrootten sneller dan getabelleerde uitvoering.

6.3 Conjuncties tabelleren

Queries gegenereerd door een ILP algoritme bestaan uit een prefix en een verfijning, waarbij het prefix een query uit de vorige iteratie is. Wanneer verfijningen van éénzelfde query uitgevoerd worden, zullen voor elke query dezelfde antwoorden herberekend worden voor hun gedeeld prefix. Om dit te vermijden kunnen de antwoorden voor de verschillende prefixen onthouden worden, zodat deze bij volgende queries hergebruikt kunnen worden. Deze aanpak noemen we *prefix-tabellering*. Een ander gevolg van de incrementele aard van queries is dat, aangezien het prefix van een query een query uit de vorige iteratie is, de antwoorden voor het prefix reeds in de vorige iteratie berekend zijn. Wanneer de antwoorden van volledige queries onthouden worden, hoeft het prefix dus nooit uitgevoerd worden. We noemen deze aanpak *query-tabellering*. Zowel prefix- als query-tabellering werden beschreven in [17].

Experimenten tonen aan dat in veel gevallen enkel het eerste antwoord van een query gebruikt wordt bij het uitvoeren van een verfijning. We voeren daarom *once-tabellering* in. Deze aanpak bewaart enkel de eerste oplossing van een

query, met als doel de totale opslagruimte benodigd voor het onthouden van oplossingen beperkt blijft.

Merk op dat query-packs ook de incrementaliteit van queries uitbuiten. Het voordeel van deze aanpak is dat deze geen extra geheugeneisen stelt voor het opslaan van oplossingen. Het nadeel is dat sommige oplossingen van prefixen nog steeds herberekend moeten worden.

We voeren een vergelijkend experiment uit om de verschillende aanpakken tegen elkaar af te wijken. De aanpakken vergelijken op basis van hun tijdsperformantie is echter niet mogelijk: hipP ondersteunt wel query-packs maar geen tabellering, terwijl YAP [7] wel tabellering ondersteunt maar geen query-packs. Als experiment meten we daarom het aantal opgeroepen doelen, en maken hiervoor gebruik van een spoor van Tilde uitvoering op de Mutagenesis en Carcinogenesis dataset. De gemeten resultaten tonen aan dat prefix-tabellering over de ganse lijn minder goed is dan query-packs. Query-tabellering presteert beter in het geval van eenvoudige queries, maar verliest van query-packs wanneer de queries complexer worden. Een bijkomend probleem is dat, bij grotere experimenten, de query-tabellering niet succesvol tot een einde komt omwille van een tekort aan geheugen. Once-tabellering is altijd beter dan niet-geoptimaliseerde uitvoering, maar is in de meeste gevallen minder goed dan de andere aanpakken. De totale geheugenwinst die deze aanpak boekt ten opzichte van query-tabellering is bovendien niet groot.

Uit de uitgevoerde experimenten concluderen we dat, hoewel query-tabellering beter presteert dan query-packs voor kleinere experimenten, deze aanpak snel te kampen krijgt met een geheugenprobleem, en dat bovendien query-packs voor grotere experimenten minder doelen moeten oproepen.

6.4 Memoriseren van query dekking

We concludeerden eerder dat het onthouden van antwoorden van queries tot betere performantie kan leiden, maar dat deze aanpak zeer slecht scaleert bij grotere problemen. In dit deel bekijken we twee algoritme-specifieke optimalisaties die eveneens antwoorden in het geheugen opslaan, maar tegen een veel lagere kost dan de aanpakken uit Sectie 6.3.

Na het selecteren van de beste query uit de verzameling geëvalueerde queries, splitst Tilde de voorbeelden op in voorbeelden die slagen en de voorbeelden die falen op de geselecteerde query. Het algoritme wordt dan recursief opgeroepen op beide verzamelingen. Voor de negatieve voorbeelden moet Tilde een andere beste query vinden, wat betekent dat dezelfde queries zullen geëvalueerd worden op alle voorbeelden. Als het algoritme bijhoudt voor welke voorbeelden welke query slaagt, kan het oproepen van de queries voor de negatieve voorbeelden vermeden worden. Wanneer we deze optimalisatie evalueren op de Mutagenesis, Carcinogenesis en HIV datasets, merken we dat de uitvoering tot 2.5 keer sneller is dan zonder de optimalisatie. De geheugenvereisten voor deze aanpak blijven daarenboven zeer beperkt.

Warmr selecteert in elke iteratie verschillende queries voor uitbreiding.

Wanneer echter een geselecteerde query op een bepaald voorbeeld faalde tijdens een iteratie, zal zijn verfijning ook falen in de volgende iteratie. Wanneer onthouden wordt tijdens elke iteratie welke queries op welke voorbeelden falen, kan de uitvoering van uitgebreide queries vermeden worden wanneer op voorhand geweten is dat deze zullen falen. Hoewel het verhinderen van uitvoering van een query eenvoudig is wanneer geen query-optimalisaties gebruikt worden, is dit minder triviaal in aanwezigheid van query-packs. We breiden daarom het query-pack mechanisme uit met de mogelijkheid om bepaalde queries uit packs uit te schakelen. Aangezien de datastructuren ontwikkeld voor adpacks zich goed tot het uitschakelen van queries lenen, gebruiken we hiervoor adpacks. Wanneer we deze optimalisatie uitvoeren op de Mutagenesis, Carcinogenesis en HIV datasets, merken we dat deze optimalisatie de query uitvoering tot meer dan 2 keer versnelt. Deze meting geldt echter enkel voor de query uitvoeringstijd; aangezien WARMR een complexe query-generatiefase heeft deze optimalisatie een kleinere impact.

6.5 Conclusies

In dit deel bestudeerden we verschillende technieken waarbij uitvoerresultaten onthouden en hergebruikt werden, teneinde de efficiëntie van ILP algoritmen te verhogen. Het opslaan van resultaten van complexe predikaten uit de achtergrondkennis is noodzakelijk om het ILP algoritme binnen redelijke tijd te laten uitvoeren. De standaard aanpak hiervoor is om de dataset op voorhand te verwerken, en manueel voor elk voorbeeld alle oplossingen te berekenen. Een alternatieve aanpak is om specialisatie toe te passen op de achtergrondkennis, wat als voordeel heeft om overweg te kunnen met cuts, maar helaas de omvang van de dataset sterk doet stijgen, en daardoor zeer slecht presteert. Getabelleerd uitvoeren van achtergrondpredikaten presteert beter, maar is nog steeds twee ordegrootten trager dan het uitvoeren op een manueel voorverwerkte dataset. Het opslaan van antwoorden van queries en hun prefixen kan herberekeningen bij het uitvoeren van queries vermijden. Deze aanpakken hebben echter het nadeel dat ze hoge geheugeneisen stellen, waardoor ze niet bruikbaar zijn bij grotere ILP problemen. Net zoals deze technieken buiten query-packs de gelijkaardigheid van queries uit, maar doen dit zonder de hoge geheugenkost. Metingen tonen aan dat, voor grotere problemen, query-packs zelfs beter presteren dan de tabelleringstechnieken. In plaats van volledige antwoorden van queries op te slaan, kan informatie over het al dan niet slagen of falen van een query gebruikt worden om uitvoering te versnellen. Zowel voor TILDE als voor WARMR zorgt een dergelijke optimalisatie voor aanzienlijke verbetering in query-evaluatietijd.

7 Conclusies

Het doel van dit werk was het ontwikkelen van technieken om de query-evaluatiestap van ILP algoritmen te optimaliseren. Het meest cruciale onderdeel van query-evaluatie is de ILP motor, die een hoog aantal queries veelvuldig moet

uitvoeren. In dit werk hebben we ons daarom toegespitst op technieken voor meer performantere ILP motoren.

We bespreken kort de mogelijke richtingen voor toekomstig werk in verband met de technieken beschreven in dit werk.

ADPacks. Het grote knelpunt van de adpacks aanpak is de transformatiestap. Om globaal beter te kunnen presteren dan query-packs moet de transformatietijd nodig om queries om te zetten in een adpack drastisch verminderd worden. Een mogelijke aanpak hiervoor is om de transformatie volledig in de kern van het systeem te implementeren. Een ander interessant spoor om te verkennen is het integreren van geavanceerde backtracking-technieken met query-packs.

(Luie) control-flow compilatie. Het integreren van luiheid in het compilatieproces vereist aanpassingen aan de datastructuren gebruikt voor de uitvoering. We hebben deze aanpassingen gedaan voor query-packs, maar dit dient ook nog gedaan te worden voor adpacks. We verwachten dat de versnelling die luie control-flow compilatie teweegbrengt bij adpacks vergelijkbaar zal zijn als die van query-packs.

Incrementele compilatie. Luie control-flow compilatie maakt het mogelijk om de incrementele aard van queries nog meer uit te buiten. Gecompileerde query-packs kunnen rechtstreeks uitgebreid worden, zodat tijdens de uitvoering van de volgende iteratie enkel de nieuwe delen van de query-pack gecompileerd moeten worden. Dit werd nog niet uitgewerkt, omdat de totale compilatietijd van control-flow compilatie momenteel te laag ligt om significante verbetering te kunnen brengen. Dit kan wel interessant zijn in situaties waar compilatietijd de query-uitvoeringstijd overheerst.

Bibliografie

[1] ACE. The ACE data mining system, 2006. http://www.cs.kuleuven.be/~dtai/ACE/.

[2] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.

[3] H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of Inductive Logic Programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166, 2002. http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=36467.

[4] M. Bruynooghe and L.-M. Pereira. Deduction revision by intelligent backtracking. In J. Campbell, editor, *Implementation of Prolog*, pages 194–215. Ellis Horwood, 1984.

[5] T. W. Chan and A. Lakhotia. Debugging program failure exhibited by voluminous data. *Journal of Software Maintenance*, 10(2):111–150, 1998.

[6] V. S. Costa, A. Srinivasan, R. Camacho, H. Blockeel, B. Demoen, G. Janssens, J. Struyf, H. Vandecasteele, and W. Van Laer. Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research*, 2002. http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=38848.

[7] L. Damas and V. S. Costa. YAP user's manual, 2003. http://yap.sourceforge.net.

[8] L. De Raedt and W. Van Laer. Inductive constraint logic. In K. P. Jantke, T. Shinohara, and T. Zeugmann, editors, *Proceedings of the Sixth International Workshop on Algorithmic Learning Theory*, volume 997 of *Lecture Notes in Artificial Intelligence*, pages 80–94. Springer-Verlag, 1995.

[9] M. Ducassé. Coca: an automated debugger for C. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 504–513. IEEE Computer Society Press, 1999.

[10] M. Ducassé. Opium: An extendable trace analyser for Prolog. *The Journal of Logic programming*, 1999. Special issue on Synthesis, Transformation and

Analysis of Logic Programs, A. Bossi and Y. Deville (eds), Also Rapport de recherche INRIA RR-3257 and Publication Interne IRISA PI-1127.

[11] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery: An overview. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 495–515. MIT, 1996.

[12] J. Gaschnig. Performance measurement and analysis of certain search algorith ms. Technical Report CMU-CS-79-124, Carnegie Mellon University, Pittsburgh, PA, 1979.

[13] E. Jahier and M. Ducassé. Generic program monitoring by trace analysis. *Theory and Practice of Logic Programming*, 2(4-5):611–643, 2002.

[14] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1994.

[15] S. Kramer, L. D. Raedt, and C. Helma. Molecular feature mining in HIV data. In *KDD '01: Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 136–143, New York, NY, USA, 2001. ACM Press.

[16] S. Muggleton and L. De Raedt. Inductive Logic Programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.

[17] R. Rocha, N. A. Fonseca, and V. S. Costa. On Applying Tabling to ILP. In *Proceedings of the 16th European Conference on Machine Learning, ECML-05*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2005.

[18] A. Srinivasan, R. King, and D. Bristol. An assessment of ILP-assisted models for toxicology and the PTE-3 experiment. In *Proceedings of the Ninth International Workshop on Inductive Logic Programming*, volume 1634 of *Lecture Notes in Artificial Intelligence*, pages 291–302. Springer-Verlag, 1999.

[19] A. Srinivasan, S. Muggleton, M. Sternberg, and R. King. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, 85(1,2):277–299, 1996.

[20] W. Vanhoof, D. De Schreye, and B. Martens. Bottom-up partial deduction of logic programs. *The Journal of Functional and Logic Programming*, 1999:1–33, 1999.

[21] W. Vanhoof, R. Tronçon, and M. Bruynooghe. A fixed point semantics for logic programs extended with cuts. In *Logic Based Program Synthesis and Transformation, LOPSTR 2002, Revised Selected Papers*, volume 2664 of *LNCS*, pages 238–257. Springer-Verlag, 2003. http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=40744.

[22] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering*, 28(2):183–200, 2002. http://www.st.cs.uni-sb.de/papers/tse2002/.