FAKULTÄT FÜR !NFORMATIK

Efficient problem solving based on datalog transformations

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Informatik

eingereicht von

Christoph Singewald
Matrikelnummer 9619059

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Univ.Prof. Dr. Reinhard Pichler
Mitwirkung: Univ.Ass. Dipl.-Ing. Michael Jakl

Wien, 03. 11. 2008

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Christoph Singewald
Pyrkergase 4a/14
1190 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 3. November 2008 _____

Kurzfassung

Viele interessante algorithmische Probleme in der Computerwissenschaft können im Allgmeinen nur mit hohem Aufwand auf einer Rechenmaschine gelöst werden. In diesem Zusammenhang sind in den letzten Jahren parametrisierbare Probleme immer wichtiger geworden. Es konnte gezeigt werden, dass viele harte Probleme effizient lösbar werden, wenn ein Parameter durch eine Konstante begrenzt ist. Besitzt ein Problem diese Eigenschaft, bezeichnet man es als fixed-parameter-lösbar (engl. fixed-parameter tractable). Bei Problemen aus dem Bereich der Graphentheorie ist die Baumweite (engl. treewidth) ein solcher Parameter. Aufgrund des Courcelle'schen Theorems wissen wir, dass sämtliche Grapheigenschaften, die sich mittels Monadic Second Order Logic (MSO) ausdrücken lassen, auf Graphen mit beschränkter Baumweite effizient entscheidbar sind. Gottlob et al. haben vor kurzem Monadisches Datalog (jedes Prädikat ist einstellig) über Strukturen mit beschränkter Baumweite als Alternative zu (MSO) vorgeschlagen. Dieser Ansatz sieht theoretisch vielversprechend aus. Eine praktische Evaluierung war aber bislang ausständig. Das Ziel dieser Diplomarbeit war es, den Monadischen Datalog Ansatz zu implementieren und mittels Tests zu evaluieren.

Im Rahmen dieser Diplomarbeit wurde der Prozess zur allgemeinen Verarbeitung - vom zugrunde liegenden Problem unabhängig - dieser Datalogregeln implementiert. Am Beispiel des Programms SAT wurde die automatische Transformation von Monadischem Datalog in "normales" Datalog für die verwendeten Datalog Interpreter (DLV, DLV Complex und DLV Complex mit externem Prädikat) durchgeführt und die Applikation mit unterschiedlichen Parametern ausführlich getestet. Als Ergebnis konnte gezeigt werden, dass dieser Ansatz ein großes Potential hat und mit Hilfe von allgemeinen Datalog Interpretern realisiert werden kann. Zudem hat sich herausgestellt, dass dieses Thema noch viel Potential zur Verbesserung bietet.

Abstract

Many interesting algorithmic problems in computer science are well known to be intractable and solvable only with high computational effort. In the recent years parameterized complexity as a new paradigm came up and is getting more and more important. It has turned out that many hard (intractable) problems become tractable if some problem parameter is fixed or bounded by a fixed constant. Problems having such a property are called fixed-parameter tractable (FPT). In case of graph problems the treewidth of a graph is such a constant parameter. By Courcelle's Theorem, we know that all graph properties expressible in Monadic Second Order Logic (MSO) can be efficiently decided on graphs with bounded treewidth. Gottlob et al. have recently proposed monadic datalog (i.e., all intensional predicates are unary) over structures with bounded treewidth as an alternative to Monadic Second Order Logic (MSO). This approach looks theoretically very promising. However, a practical evaluation has been missing so far. The goal of this diploma thesis was to implement the monadic datalog approach and to evaluate it by means of testing.

In this thesis we implement an evaluation process for datalog rules abstracted from the underlying problem. Using the example of the program SAT we implemented the automatic transformation from monadic datalog to "normal" datalog for the used datalog engine (DLV, DLV Complex and DLV Complex with external predicate) and tested our application with various parameters. As a result we could show that this approach is feasible, has (a lot of) potential and can be implemented using a generic datalog system. Further more we discovered that there is room for further improvements.

Dedication

I dedicate this master thesis to my daughter, Lena, who had to do without me a lot of time during my studies.

Acknowledgements

I would like to thank my supervisor, Prof. Reinhard Pichler, for providing an interesting, relevant and useful topic for this master thesis, for many constructive critical comments, for all the support and for showing so much patience reviewing this thesis. I would also like to thank Dr. Nysret Musliu, Dr. Fang Wei and Dipl.-Ing. Michael Jakl for their helpful comments and suggestions during the work on the practical part of this thesis. And finally, I am grateful to my mother and my family for their support throughout the years.
This master thesis was supported by the Austrian Science Fund (FWF), project P20704-N18.

Abbreviations

CNF	conjunctive normal form
DLV	disjunctive logic programming system
DLV-C	disjunctive logic programming system complex
EDB	extensional database
FOL	first order logic
IDB	intensional database
MSO	monadic second order logic
SAT	satisfiability problem
SQL	structured query language
tw	treewidth

List of Figures

List of Listings

Contents

Chapter 1

Introduction

Many problems (e.g. in AI and operations research) are well known to be intractable. In the recent years parameterized complexity as a new paradigm came up and is getting more and more important. It has turned out that hard (intractable) problems can be solved efficiently with this approach, if some problem parameter is fixed or bounded by a fixed constant. Problems having such a property are called fixed-parameter tractable (FPT). Considering the class of graph problems such a parameter can be the treewidth of a graph, which can be obtained from the tree decomposition of the graph. More generally the treewidth of arbitrary finite structures can be such a parameter. One of the famous and powerful methods to derive fixed-parameter tractable problems is Courcelles theorem [7], which says that any property of finite structures, which is expressible by an MSO sentence, can be decided in linear time (data complexity) if the structures have bounded treewidth.

In [15] was proposed to express the MSO with datalog over finite structures. This approach leads to monadic datalog. It was proven that if some property of a finite structure is expressible in MSO, then this property can also be expressed by means of a monadic datalog program over this structure plus the tree decomposition.

An interesting question, studied in this thesis, is how we may generalize the evaluation process of the monadic datalog rules to abstract it from the underlying problem. The fastest way to evaluate the datalog rules is to implement the evaluation process using an imperative programming language (e.g. C++), but this approach covers too many problem specific parameters and structures in the program code. The goal should be a separation of the underlying problem and the evaluation process, to allow replacing the underlying problem with another one.

In this thesis we examine a flexible implementation for the transformation

from the monadic datalog program as proposed in [15] to the datalog language for the Disjunctive Datalog System (also called DLV) without set semantics and for an extension of DLV called DLV Complex with set semantics (for DLV and DLV-C see [17]).

This whole translation process consists of the following parts:

1. Parsing the CNF formula and the monadic datalog program
2. Compute the tree decomposition
3. Normalize the tree decomposition
4. Generate and translate the internal predicates
5. Solve translated datalog program with DLV or DLV-C

1.1 Summary of results

Using the example of the program SAT we implemented the automatic transformation from monadic datalog to "normal" datalog for the used datalog engine (DLV, DLV Complex and DLV Complex with external predicate) and tested our application with various parameters. As a result we could show that this promising approach is feasible and can be implemented using a generic datalog system. Further more we discovered that there is room for further improvements.

1.2 Organization

The thesis is organized as follows. We start with Chapter 2 providing some basic notions. Chapter 3 describes the datalog algorithm and the transformation process. We use transformation and translation synonymously. In Chapter 4 we describe the implementation of the process and in Chapter 5 we present the experimental results. We conclude with Chapter 6.

Chapter 2

Preliminaries

In this chapter we give basic definitions and preliminary results. In general this work follows [15] and [14].

2.1 Finite structures and graphs

The finite model theory is an area of the mathematical logic. Basic concepts in this field are e.g. finite graphs, databases etc. Many of the problems of complexity theory and database theory can be formulated as problems of mathematical logic limited to finite structures.

Definition 2.1.1 *Let $\tau = \{R_1, .., R_K\}$ be a set of predicate symbols. A* finite structure *$\mathcal{A}$ over τ (short τ-structure) is given by a finite domain $A = dom(\mathcal{A})$ and relations $R_i^A \subseteq A^\alpha$, where α denotes the arity of $R_i \in \tau$*

Typical examples of structures are finite graphs where the binary symbol E denotes the edge-relation of the graph. We assume basic knowledge in graph theory, but we give a little overview of the terminology that is used in this thesis. Further information can be found e.g. in [16].

Definition 2.1.2 *An undirected graph G is a pair (V, E) where V is a set of vertices, and E is a set of edges. Each edge $e \in E$ is an unordered pair (v_1, v_2) with $v_1, v_2 \in V$. Two vertices are connected in a graph G if there is a path between them. A graph G is connected if every pair of vertices of G is connected. A graph with a finite number of nodes and edges is called finite graph.*

Definition 2.1.3 *A forest is an undirected cycle free graph. A connected forest is a tree. A rooted tree is a tree in which one of the vertices is distinguished from the others. This distinguished vertex is called the root of the tree.*

In this thesis we assume that all graphs are undirected. We also assume that the children of a node v in a tree T are in some fixed order, for example $child_1(v)$ denotes the first, $child_2(v)$ denotes the second and $child_n(v)$ denotes the child n of node v in tree T.

2.2 The SAT Problem

In this section we want to recall the widely (well) known satisfiability problem, also called SAT problem, which is one of the most famous NP-complete [6] problems. We provide a running example to which is referred to in further sections.

Definition 2.2.1 *Let $F = (x_1, x_2, ..., x_n)$ be a boolean formula built up from propositional variables $x_1, ..., x_n$ and the logical connectives $\vee$, $\wedge$ and $\neg$. An interpretation of F is simply a subset of $Var(F)$ with the intended meaning that variables in X evaluate to true, while all other variables evaluate to false. If F evaluates to true in X, then X is called a model of F, written as $X \models F$. Likewise, we write $F_1 \models F_2$ if every model of F_1 is also a model of F_2.*

Example 2.2.1 *An example formula in CNF notation*

$$F = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_4 \vee x_5) \wedge (x_3 \vee x_4 \vee x_6).$$

Example 2.2.2 *A possible model of Example 2.2.1*

$$T = \{x_1 = true, x_2 = false, x_3 = true, x_4 = false, x_5 = false, x_6 = true\}$$

A propositional formula in CNF can be represented as a finite structure over the alphabet $\tau = \{cl(.), var(.), pos(.,.), neg(.,.)\}$, where $cl(c)$ means that c is a clause, $var(x)$ means that x is a variable, $pos(x, c)$ means that x occurs positive in c and $neg(x, c)$ means that x occurs negative in c.

Example 2.2.3 *Ground atoms of the structure S representing formula F in Example 2.2.1*

$$\begin{aligned} S = \{&cl(c1), cl(c2), cl(c3),\\ &var(x1), var(x2), var(x3), var(x4), var(x5), var(x6),\\ &pos(c1, x1), pos(c1, x2), pos(c1, x3),\\ &neg(c2, x2), neg(c2, x4), pos(c2, x5),\\ &pos(c3, x3), pos(c3, x4), pos(c3, x6)\} \end{aligned}$$

2.3 Tree decomposition

In this section we give a brief overview on tree decomposition and treewidth. For further reading visit [2],[4],[18]. The concept of the tree decomposition has been first introduced by Robertson and Seymour in [19].

Definition 2.3.1 *A tree decomposition T of a τ-structure $\mathcal{A}$ is defined as a pair $\langle T, (A_t)_{t \in T} \rangle$ where T is a tree and each A_t is a subset of A with the following properties:*

1. *Every $a \in A$ is contained in some A_t.*
2. *For every $R_i \in \tau$ and every tuple $(a_1, ...a_\alpha) \in R_i^{\mathcal{A}}$, there exists some node $t \in T$ with $\{a_1, ..., a_\alpha\} \subseteq A_t$.*
3. *For every $a \in A$, the set $\{t | a \in A_t\}$ induces a subtree of T.*

The third condition is usually referred to as connectedness condition. The sets A_t are called bags of T. The width w of a tree decomposition $\langle T, (A_t)_{t \in T} \rangle$ is defined as $max\{|A_t| | t \in T\} - 1$. The treewidth of $\mathcal{A}$ is the minimal width of all tree decompositions of $\mathcal{A}$ and is denoted as $tw(\mathcal{A})$. Trees and forests are precisely the structures with treewidth 1.

An alternative definition of a tree decomposition based on graphs is provided in [18].

Definition 2.3.2 *A tree decomposition of a graph $G = (V, E)$ is a pair (T, χ), where $T = (I, F)$ is a tree with node set I and edge set F, and $\chi = \{\chi_i : i \in I\}$ is a family of subsets of V, one for each node of T, such that*

1. *$\bigcup_{i \in I} \chi_i = V$.*
2. *For every edge $(v, w) \in E$, there is an $i \in I$ with $v, w \in \chi_i$.*
3. *For all $i, j, k \in I$, if j is on the path from i to k in T, then $\chi_i \cap \chi_k \subseteq \chi_j$.*

According to Definition 2.3.2, the width w of a tree decomposition is $max_{i \in I} |\chi_i| - 1$. The treewidth of a graph G, denoted by $tw(G)$, is the minimum width over all possible tree decompositions of G.

The following proposition was shown in [12]: Let be G the primal graph of the τ-structure representing the CNF formula, then the treewidth of the τ-structure (Definition 2.3.1) is equal to the treewidth of graph G (Definition 2.3.2).

For a given $w \geq 1$, it can be decided in linear time if some structure (e.g. a graph) has a treewidth $\leq w$ and in case of a positive answer, a tree decomposition of width w can be computed in linear time [3].

If a set of finite structures have a tree decomposition with a bounded treewidth by some bounded constant k, then many hard (intractable) problems can be solved efficiently. This is the reason why the treewidth is such an important parameter. Let G be a graph containing a cycle, then the minimal treewidth is 2, otherwise it must be a tree or a forest with $tw = 1$. As we show in section 2.4 a given tree decomposition can easily be translated into a binary rooted tree.

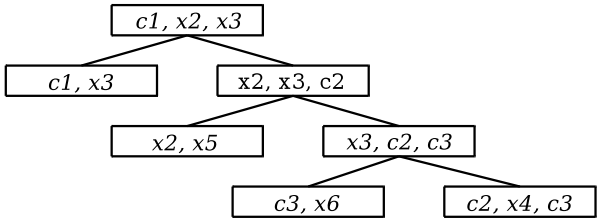

Figure 2.1: Tree decomposition of formula F

2.4 Normalization

In [15] following normal form of a tree decomposition is proposed and it was shown that any tree decomposition can be normalized in linear time.

Definition 2.4.1 *Normal form of a tree decomposition*

1. *The bags are considered as tuples of* $w+1$ *pairwise distinct values* $(a_0, ..., a_w)$ *rather then sets.*

2. *Every internal node* $t \in T$ *has either one or two children.*

3. *If a node* t *with a bag* $(a_0, ..., a_w)$ *has one child node, then the bag of the child is either obtained via a permutation of* $(a_0, ..., a_w)$ *or by replacing* a_0 *with another element* a'_0*. We call such a node* t *a permutation node or an element replacement node, respectively.*

4. *If a node* t *has two child nodes then these child nodes have identical bags as* t*, which is called branch node.*

The permutation node can be replaced with two other nodes called element introduction and removal node. In this case the internal bags can not consist of a constant number of values $(a_0, ..., a_k)$. This leads to following equivalent normal form:

Definition 2.4.2 *Normal form of a tree decomposition*

1. *The bags consist of pairwise distinct values* $(a_0, ..., a_k)$ *with* $k = w$ *or* $k = w - 1$.

2. *Every internal node* $t \in T$ *has either one or two children.*

3. *If an internal node* t *has one child node* t'*, then the bag* A_t *is obtained from the bag* $A_{t'}$ *by either removing one element or introducing a new element.*

4. *If a node* t *has two child nodes then these child nodes have identical bags as* t*, which is called branch node.*

Let $\langle T, (A_t)_{t \in T} \rangle$ an arbitrary tree decomposition then a normalized tree decomposition $\langle T', (A'_t)_{t \in T} \rangle$ can be obtained by processing following steps.

1. Padding: All bags can be padded to the size of $w + 1$ by adding elements from the neighbor nodes.

2. Binary split: If some node s has $k + 2$ children it is a standard technique to insert copies of s to turn the tree into a binary shape.

3. Insert copy-node: Suppose some node s having two children t_1 and t_2 where the bags of t_1, t_2 are not identical then insert a copy of s between s and t_1 and s and t_2. On every node s with two children there should be no difference between the children and s now.

4. Interpolate children: Between a parent node s and its child s' should only be a difference of one element. Let node s be the parent of node s' and let $k = A_s \setminus A_{s'}$ with $k > 1$, then interpolate such that there are new nodes $s_1, ..s_k - 1$ between s and s' and $s_k - 1$ is the new parent of s'.

2.5 Monadic Second Order Logic

Monadic Second Order Logic (also called MSO) extends the First Order Logic (also FOL) by the use of set variables, which range over subsets of domain elements. The identifiers of the set variables are usually denoted with upper case letters.

Syntactically new atomic formulas $x \in X$ are introduced, which means that x is a member of X. Uppercase letters will denote set variables and lowercase letters will denote first-order variables.

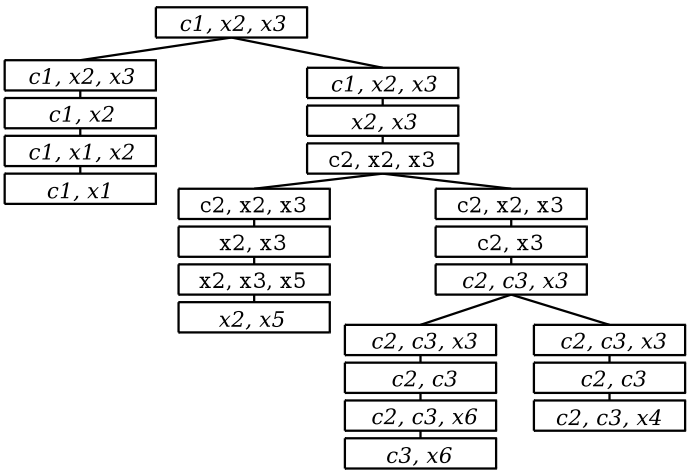

Figure 2.2: Normalized tree decomposition of formula F

Definition 2.5.1 *SAT Problem expressed as MSO formula (see [8], [14])*

$$(\exists X)(\forall c)(cl(c) \rightarrow (\exists z)[(pos(z,c) \vee z \in X) \wedge (neg(z,c) \vee z \notin X)])$$

Courcelle [7] has given a large class of graph properties, namely the class of properties that are definable in Monadic Second Order Logic or MSO for graphs. We define some of them used in this thesis.
MSO for graphs $G = (V, E)$ consists of a language in which predicates can be built with

1. the logic connectives $\vee, \wedge, \Rightarrow$ and $\Leftrightarrow$ (with their usual meanings),
2. quantifiers $\exists, \forall$,
3. the following binary relations:
   (a) the binary relation $x \in A$, which means that x is a member of A;
   (b) $e(x, y)$ which means $e \in E$ is an edge from x to y with $x, y \in V$;
   (c) equality for variables.

The importance of MSO is motivated by the following theorem taken from [7].

Theorem 2.5.1 *Let ϕ be a fixed* MSO sentence and let k be a fixed constant. Deciding whether ϕ holds for an input graph G (more generally, for an input structure A) can be done in linear time if the treewidth of the graphs (resp. of the structures) under consideration is bounded by k.

2.6 Datalog

This section covers a brief description of datalog, for more information see [20]. Datalog is a collection of rules, which are sets of Horn clauses with following form

$$Head \leftarrow Body.$$

The head is an atom, which is a predicate applied with arguments, and the body is a set of atoms.

$$pred(h_1, ..., h_n) \leftarrow pred_1(b_1, ..., b_n), ..., pred_r(b_1, ..., b_n).$$

The heads of rules having a body are called intensional database (IDB) and the heads of rules without a body (also facts) are called extensional database (EDB). Atoms occurring in the body of rules may be EDB or IDB predicates, they are also called subgoals. Informal meaning of a rule is, the head with its arguments is true whenever there exists a value for any local variable that makes the subgoals true.

Datalog programs are function-free logic programs. All evaluation algorithms of a datalog program P result in a fix-point calculation. We are interested in the minimal model semantics (see [20]) and in datalog over τ-structures (see [15]), especially graphs. Given a structure S and a datalog program P then the structure provides additional EDB predicates so that the evaluation is a fix-point calculation over $P \cup S$.

As mentioned in [9] combined complexity of datalog is EXPTIME complete, but some fragments can be evaluated much more efficiently.

1. Propositional datalog (no rule contains a variable that has to be grounded) can be evaluated in linear time.

2. The guarded fragment of datalog (all variables of rule r occur in an EDB atom in the body of r) can be evaluated in $\mathcal{O}(|P| \times |S|)$.

3. Monadic datalog (described in section 2.7) is NP-complete in combined complexity.

2.7 Monadic Datalog

Monadic datalog is a subset of datalog, where every intensional predicate is unary (see [13]). In this work we are primarily interested in monadic datalog over finite structures with bounded treewidth. We extend the set of predicate symbols τ_p of the monadic datalog program P with

$$\tau_{ext} = \tau_p \cup \{root, leaf, child_1, child_2, bag\}$$

The unary predicate $root(v)$ denotes the root node of the tree while the predicate $leaf(v)$ denotes the leafs. Because of the normalized tree every node has at most two child nodes thus we have the binary predicates $child_1$, $child_2$ to indicate the child parent relations. For instance $child_1(v_1, v)$ indicates that node v_1 is the first or the only child of node v and $child_2(v_2, v)$ denotes that v_2 is the second child of the node v. The predicate $bag(t, x_0, ..., x_n)$ has the arity of $n+2 < w+2$ where $w \geq 1$ denotes the treewidth of the decomposed tree.

Definition 2.7.1 *Let τ be a set of predicate symbols and let $w \geq 1$. A monadic datalog program over τ-structures with treewidth w is a set of datalog rules where all extensional predicates are from τ_{ext} and all intensional predicates are unary.*

As in Section 2.6 mentioned, the evaluation of monadic datalog is NP-complete (combined complexity). The evaluation of the restricted fragment of datalog called "quasi-guarded" is tractable, which is shown in [15].

Definition 2.7.2 *Let B be an atom and y a variable in some rule r. We call y "functionally dependent" on B if in every ground instantiation r' of r, the value of y is uniquely determined by the value of B. We call a datalog program P "quasi-guarded" if every rule r contains an extensional atom B, s.t. every variable occurring in r either occurs in B or is functionally dependent on B.*

An important theorem for this work is proved in [15].

Theorem 2.7.1 *Let τ an $w \geq 1$ be arbitrary but fixed. Every MSO-definable unary query over τ-structures of a treewidth w is also definable in the quasi-guarded fragment of monadic datalog over τ_{ext}.*

Depending on the implementation additional internal predicates for set operations are necessary. For this the set of predicate symbols τ_{ext} will be extended again with

$$\tau_{iext} = \tau_{ext} \cup \{union, intersect, diffel\}$$

where $union(X, A, B)$ and $intersect(X, A, B)$ representing the union respectively intersection of A and B. The predicate $diffel(A, B, el)$ with the sets A and B (with $B \subset A, el \notin B$) is true iff $A = B \cup \{el\}$.

Chapter 3

Evaluation and transformation

In this chapter we present the datalog program for the SAT problem as proposed in [15]. Furthermore we illustrate the datalog representation of the tree decomposition and we discuss some issues of the datalog translation into the language of DLV and DLV-C.

3.1 Monadic datalog program deciding SAT

Listing 3.1 shows the monadic datalog program which decides the SAT problem operating on the normalized decomposed tree. For a better readability the predicates for the set operations (*union*, *intersect*, *diffel*) are written as symbols. Usually in datalog variables are denoted with upper case letters and constants with lower case letters. We follow the pattern of [15] and denote set variables with upper case letters and variables (element variable furthermore) for a single bag, for a single clause or for a single propositional variable with lower case letters.

Because of the normalization (see section 2.4) of the tree decomposition, the SAT problem contains in its tree representation the following different types of nodes:

1. Leaf node
2. Removal node (variable or clause)
3. Introduction node (variable or clause)
4. Branch node
5. Result node

The leaf node denotes a leaf of the tree i.e. a node without children. The variable removal node differs from its child in one variable i.e. the bag of the

child node contains exactly one more variable. This is also the definition for the clause removal node. The introduction node is the opposite type of the removal node. A node of this type has exactly one variable or clause more than its child. There is one rule for a result node, which denotes the root of the tree and if the head (the predicate *success*) of this rule is true, then the formula is satisfiable. The program contains three intensional predicates: *solve*, *success*, *true*. The predicate $solve(v, P, N, C)$ has an arity of four with v is a node of the tree. P and N are subsets of the variables in the bag denoted by v containing the positive respectively the negative variables. C is a subset of the clauses in the bag identified by v which are true by the variables in P and N with $P \cap N$. The predicate *success* has the arity of 0 and will be derived when the formula is solvable. The third intensional predicate of the SAT program is $true(P, N, C_1, C)$. The informal meaning of this predicate is to calculate the maximal number of the clauses in C which are true with the interpretation given by P and N. In most cases, e.g. in the leaf node rule, P and N are disjoint sets. In Chapter 4 we will discuss this predicate again, as it is one of the critical parameters of the evaluation process. The following theorem is proved in [15]:

Theorem 3.1.1 *The datalog program in Listing 3.1 decides the SAT problem, i.e. the fact "success" is in the fix-point of this program iff the input τ_{ext}-structure encodes a satisfiable clause set C. Moreover for any clause set C with treewidth w, the computation of the τ_{ext}-structure and the evaluation of the datalog program can be done in time $\mathcal{O}(f(w) \times |C|)$ for some function f.*

3.2 Datalog tree representation

The datalog tree representation of the normalized tree decomposition (Figure 2.2) of the formula in Example 2.2.1 is shown in Listing 3.2. The tree structure provides additional extensional predicates and holds the knowledge about the tree decomposition. As in Section 2.7 mentioned, the predicates used in this representation are $\{root, leaf, child_1, child_2, bag\}$. The predicate $bag(v, X, C)$ represents the set of variables X and set of clauses C of a node v. Let tw be the treewidth, it is easy to see that $|X \cup C| \leq tw + 1$ holds.

Listing 3.1: Program SAT

```
/* leaf node. */
solve(v,P,N,C1) ← leaf(v), bag(v,X,C), P ∪ N=X, P ∩ N=∅,
true(P,N,C1,C).

/* variable removal node. */
solve(v,P,N,C1) ← bag(v,X,C), child1(v1,v), bag(v1,X ⊎ {x},C),
solve(v1,P⊎{x},N,C1).

solve(v,P,N,C1)← bag(v,X,C), child1(v1,v), bag(v1,X⊎{x},C),
solve(v1,P,N⊎{x},C1).

/* clause removal node. */
solve(v,P,N,C1) ← bag(v,X,C), child1(v1,v), bag(v1,X,C⊎{c}),
solve(v1,P,N,C1⊎{c}).
/* variable introduction node. */
solve(v,P⊎{x},N,C1∪C2) ← bag(v,X⊎{x},C), child1(v1,v),
bag(v1,X,C), solve(v1,P,N,C1), true({x},∅,C2,C).
solve(v,P,N⊎{x},C1∪C2) ← bag(v,X⊎{x},C), child1(v1,v),
bag(v1,X,C), solve(v1,P,N,C1), true(∅,{x},C2,C).

/* clause introduction node. */
solve(v,P,N,C1∪C2) ← bag(v,X,C⊎{c}), child1(v1,v),
bag(v1,X,C), solve(v1,P,N,C1), true(P,N,C2,{c}).

/* branch node. */
solve(v,P,N,C1∪C2) ← bag(v,X,C), child1(v1,v), bag(v1,X,C),
solve(v1,P,N,C1), child2(v2,v), bag(v2,X,C), solve(v2,P,N,C2).

/* result (at the root node). */
success ← root(v), bag(v,X,C), solve(v,P,N,C).
```

Listing 3.2: datalog tree representation of tree decomposition in Figure 2.2

```
% root
root(n1).
% leafs
leaf(n5).
leaf(n12).
leaf(n19).
leaf(n22).
% child1 and child2
child1(n2,n1). child1(n9,n8).    child1(n16,n15).
child1(n3,n2). child1(n10,n9).   child1(n17,n16).
child1(n4,n3). child1(n11,n10).  child1(n18,n17).
child1(n5,n4). child1(n12,n12).  child1(n19,n18).
%
child2(n6,n1). child2(n13,n8).   child2(n20,n15).
child1(n7,n6). child1(n14,n13).  child1(n21,n20).
child1(n8,n7). child1(n15,n14).  child1(n22,n21).
% bag predicates
bag(n1,{x2,x3},{c1}).    bag(n12,{x2,x3,x5},{}).
bag(n2,{x2,x3},{c1}).    bag(n13,{x2,x3},{c2}).
bag(n3,{x2},{c1}).       bag(n14,{x3},{c2}).
bag(n4,{x1,x2},{c1}).    bag(n15,{x3},{c2,c3}).
bag(n5,{x1},{c1}).       bag(n16,{x3},{c2,c3}).
bag(n6,{x2,x3},{c1}).    bag(n17,{},{c2,c3}).
bag(n7,{x2,x3},{}).      bag(n18,{x6},{c2,c3}).
bag(n8,{x2,x3},{c2}).    bag(n19,{x6},{c3}).
bag(n9,{x2,x3},{c2}).    bag(n20,{x3},{c2,c3}).
bag(n10,{x2,x3},{}).     bag(n21,{},{c2,c3}).
bag(n11,{x2,x3,x5},{}). bag(n22,{x4},{c2,c3}).
```

3.3 Evaluation

The evaluation of the datalog rules can be implemented in different ways. An efficient implementation using the programming language C++ has been proposed in [14]. The aim of this work is to provide an implementation which is independent of the underlying problem. This approach is useful for rapid development and prototyping, because for instance the SAT problem can be replaced by an other problem using the same system. For sure measured in absolute time an individual implementation in C++ may be faster regarding the large number of variables and clauses, but is less flexible and reusable.
Our proposed approach is to feed the disjunctive logic programming system DLV[1] and its recently published extension DLV Complex[2] (also DLV-C) with the merged and transformed datalog rules obtained from the SAT program plus the tree decomposition of the formula. The evaluation is processed and measured with three different methods:

1. DLV
2. DLV Complex
3. DLV Complex with an external "true" predicate

In this place we want to give a short introduction to DLV and DLV-C which follows the work of [17]. Please visit [17],[5] and the manual of DLV and DLV Complex for a detailed description.

3.3.1 Disjunctive logic programming system DLV

The disjunctive logic programming system (short DLV) is a common datalog engine which evaluates datalog rules under an answer set semantics. Disjunctive logic programs are function free datalog programs where disjunction is possible in the head. The language has been improved over the years since the DLV project started in 1996.

We only cite those features of DLV and DLV-C (see Section 3.3.2) which are relevant for this approach.

Definition 3.3.1 *A disjunctive rule r is a formula*

$$a_1 \vee ... \vee a_n \ :- \ b_1, ..., b_k, not\ b_{k+1}, ...not\ b_m$$

[1] http://www.dlvsystem.com
[2] http://www.mat.unical.it/dlv-complex

where $a_1, ..., a_n, b_1, ..., b_k$ *with* $n \geq 0$,$m \geq k \geq 0$ *are classical literals, where* $b_1, ..., b_k$ *are called positive and not* $b_{k+1}, ...not\ b_m$ *negative literals of the body of rule r. The disjunction* $a_1 \vee ... \vee a_n$ *is called the head of the rule and the conjunction* $b_1, ..., b_k, not\ b_{k+1}, ...not\ b_m$ *is called the body of the rule. The* $:-$ *sign is the right-hand implication following the notation of prolog. A rule with exactly one literal in the head is called normal rule while a rule without a literal in the head is treated as an integrity constraint. A rule with an empty body is also called a fact, where the* $:-$ *sign is omitted.*

Definition 3.3.2 *A disjunctive datalog program P is a finite set of disjunctive rules. A not-free disjunctive datalog program is called positive and a* $\vee$*-free program is called (normal) datalog program.*

In this thesis we are only interested in normal datalog programs. The disjunctive approach may play a major role in this topic in future.

Definition 3.3.3 *A rule r is save if every variable which occurs in rule r also appears in at least one positive literal of the body of the rule r. A logic program is save if all rules of the program are save.*

Example 3.3.1 *The following disjunctive logic program*

$$a \vee b.$$
$$b \vee c.$$
$$d \vee -d \ :- \ a, c.$$

has following solution $\{b\},\ \{a, c, -d\},\ \{a, c, d\}$.

DLV supports similar to SQL the use of aggregate functions (see [10]), where

1. $\#sum$ calculates the sum of a result
2. $\#max$,$\#min$ selects the maximum resp. minimum of a result
3. $\#count$ counts a result set

are the most important ones.

Definition 3.3.4 *A Symbolic Set has the form* $\{Vars : Conj\}$ *where* $Vars$ *is a list of local variables and* $Conj$ *is a conjunction of non-aggregate literals. An aggregate is used in combination with a Symbolic Set and has the form* $\#count\{Vars : Conj\}$,

Example 3.3.2 *Rule using the aggregate function #count*

$$countElements(X) :- \#count\{Set : elementOf(Set, \_)\} = X.$$

Example 3.3.3 *Rule using the aggregate function #max*

$$maxNumber(X) :- \#max\{I : numbers(I)\} = X.$$

DLV offers many command line options. In this thesis we only use few of them.

Listing 3.3: DLV command-line used in this work

```
$>dlv -N=99999 [filename [filename [...]]]
```

The option $N = x$ limits integers and their operations to the value of x, where 99999 is maximum value for this option. The upper bound is necessary because of the definition of save rules. For example the following program is valid and save:

Listing 3.4: Save program needing upper bounds

```
number(1).
x(N) :- number(N).
x(X) :- x(N),X=N+1.
```

If there are no upper bounds the recursion will be run infinite.

The version of DLV used in this thesis is based on

```
DLV [build BEN/Oct 11 2007 gcc 4.1.2 20061115 (Debian 4.1.1-21)]
```

3.3.2 Extension DLV Complex

DLV Complex is a very young project and extents DLV by (recursive) functions, sets, lists and external predicates definable as system library. The name of an external predicate has to start with a "#" sign. We only use sets and the provided built in set-functions. Because DLV Complex extends DLV it has of course the same language features of DLV plus the extensions.

The internal set functions used in this thesis are

1. $\#union(Set1, Set2, Set3)$ where $Set3$ is the result of the union of $Set1$ and $Set2$

2. $\#intersection(Set1, Set2, Set3)$ where $Set3$ is the result of the intersection of $Set1$ and $Set2$

3. $\#difference(Set1, Set2, Set3)$ where $Set3$ is the result of set difference of $Set1$ and $Set2$

4. $\#member(El, Set)$ is true if El is an element of Set

5. $\#subSet(Subset, Set)$ is true if $Subset$ is a subset of Set

6. $\#card(Set, Int)$ where Int is the cardinality of set Set

As mentioned above DLV Complex provides a facility to define external predicates in an external dynamic link library (so,dll,...). The path to this library has to be given via the command-line option $-libpath =< LibPath >$. Every programming language which is able to build a system library can be used to develop the external library. In this thesis we used C++ for our external library, because DLV Complex provides an SDK for C++(for details see Section 4.8).

As in DLV we have to limit the integer operations with the option $-N$ (see Listing 3.5). The libraries used in the datalog program must be included on the top of it via $\#include < LibraryName >$. So DLV knows which library has to be linked during execution.

Listing 3.5: DLV-C command-line used in this work

```
$>dlv-c -N=99999 -libpath=<LibPath> [filename [filename [...]]]
```

The version of DLV Complex used in this thesis is based on

```
DLV [build BEN/Jul 29 2008 gcc 4.1.2 20061115 (Debian 4.1.1-21)]
```

3.4 Transformation

In this section we provide an overview of the transformation process from monadic datalog with set semantics to the language of the destination system, which are DLV and DLV-C in this thesis. The whole process should run in polynomial time. We designed the application, we call it Dattrans, which implements the transformation process, as a preprocessor for the used datalog engine. It takes a command-line switch for generation of the datalog rules for the appropriate destination engine. The implementation details are discussed in Chapter 4. First we give an overview and afterwards we present the single steps more in detail.

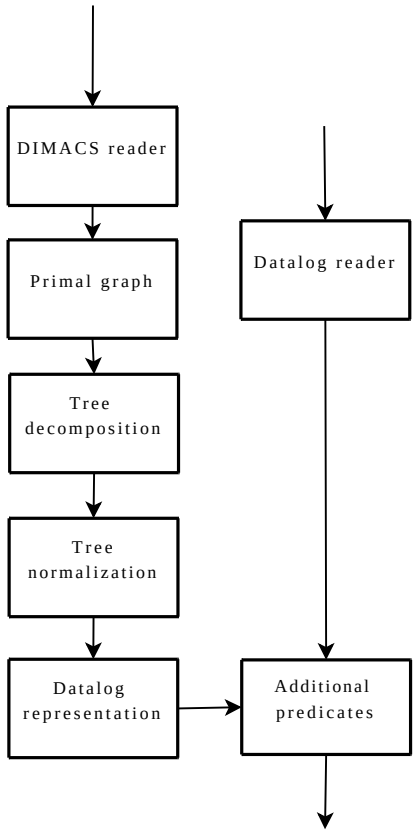

Figure 3.1: Transformation chain

3.4.1 Overview

In general the transformation process (Figure 3.1) is built up on the following chain of subgoals, which may differ depending on the destination datalog engine.

1. Reading the CNF formula
2. Reading the datalog program (i.e. the SAT program)
3. Building the primal graph of the τ-structure representing the CNF formula
4. Creating the tree decomposition
5. Normalization of the tree decomposition
6. Creating the datalog representation of the tree decomposition and the structure of the formula
7. Converting the names of the variables
8. Creating additional and converting existing predicates
9. Converting sets into datalog rules in case DLV is used

We first describe the reading of the input (formula and monadic datalog) in Section 3.4.2, then the computation of tree decomposition and the normalization of the tree are discussed in Section 3.4.3.

The generation of the datalog representation consists of two parts, the common part and a post processing depending on the underlying datalog engine. We present the common part in Section 3.4.4 and the post processing for DLV in Section 3.4.6, for DLV Complex in Section 3.4.7 and for DLV Complex with external predicate in Section 3.4.8. Afterwards we conclude this chapter with a summary in Section 3.4.9 .

3.4.2 Reading the input and creating the data structures

The input format for the CNF formula is the well known DIMACS [1] format (see Section 4.3). After reading the DIMACS graph format for satisfiability problems, a primal graph is built up on the formula (see Figure 3.2) and is stored in an appropriate data structure. The information about the negative variables is stored in the edges of the graph and is important representing the graph in datalog (see Example 2.2.3).

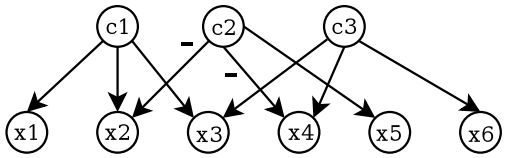

Figure 3.2: Primal graph of the formula F in Example 2.2.1

Independent from processing the CNF formula the file with the monadic datalog program is parsed and the rules are stored in an appropriate data structure (for implementation details see Chapter 4). The key to a fast and correct translation is a fast data structure.

We modified the monadic datalog program syntactically (see Listing 3.1) using the syntax of DLV (see Listing 3.6). First we replaced the symbols $\cap$ and $\cup$ with the predicate $union(X, P, N)$ where $X = P \cup N$ and the predicate $intersect(X, P, N)$ where $X = P \cap N$. Next we introduced a new predicate $diffel(X, Y, el)$ which represents the expression $X = Y \uplus \{el\}$. The informal meaning of this predicate is that X and Y differ in exactly one element. To avoid unsafe rules we added domain predicates for some variables (e.g. $pset(X)$ and $cset(C)$).

Listing 3.6: Program SAT

```
% leaf node
solve(v,P,N,C1) :- leaf(v),bag(v,X,C),true(v,P,N,C1,C),
                   union(X,P,N),intersect({},P,N).
% internal  node
% variable removal node
solve(v,P,N,C1) :-  bag(v,X,C), child1(v1,v),
                bag(v1,XX,C), diffel(XX,X,elx),
                solve(v1,X3,N,C1),diffel(X3,P,elx),pset(_,P).

solve(v,P,N,C1) :- bag(v,X,C),child1(v1,v),
                bag(v1,X1, C), diffel(X1,X,elx),
                solve(v1,P,X2,C1),union(X2,N,elx),pset(_,N).

% clause removal node
solve(v,P,N,C1) :- bag(v,X,C), child1(v1,v),
                bag(v1,X,CX1),diffel(CX1,C,elc),
                solve(v1,P,N,CX2),union(CX2,C1,elc).

% variable introduction node
solve(v,PX,N,C1C2) :- bag(v,XX,C),diffel(XX,X,elx),
                child1(v1,v),bag(v1,X,C),
                solve(v1,P,N,C1),true(v,x,{},C2,C),
                diffel(PX,P,elx),union(C1C2,C1,C2),pset(_,PX).

solve(v,P,NX,C1C2) :- bag(v,XX,C),diffel(XX,X,elx),
                child1(v1,v),bag(v1,X,C),
                solve(v1,P,N,C1),true(v,{},elx,C2,C),
                diffel(NX,N,elx),union(C1C2,C1,C2),pset(_,NX).

% clause introduction node
solve(v,P,N,C1C2) :- bag(v,X,CX),child1(v1,v),
                bag(v1,X,C),solve(v1,P,N,C1),
                true(v,P,N,C2,elc),diffel(CX,C,elc),
                union(C1C2,C1,C2),cset(_,CX).

% branch node
solve(v,P,N,C1C2) :- bag(v,X,C),
                child1(v1,v),bag(v1,X,C),
                solve(v1,P,N,C1),child2(v2,v),bag(v2,X,C),
                solve(v2,P,N,C2),union(C1C2,C1,C2).

%result (at root node)
success :-  root(v),bag(v,X,C),solve(v,P,N,C).
```

3.4.3 Tree decomposition and normalization

After the graph has been created the tree decomposition is processed on it. The negative edges in the graph have no relevance for computing the tree decomposition and its datalog representation, they are considered in the representation of the CNF formula. Afterwards the tree is normalized (see section 2.4) by processing the binary split, the insertion of a copy-node and the interpolation of nodes between parents and its children. In this section we want to present the interpolation algorithm called InterpolateChild. During the normalization of the tree, the number of clauses C plus the number of variables X in one node n must never exceed the treewidth plus 1, recall let tw be the treewidth then $|X \cup C| \leq tw + 1$ holds for every node. Thus the algorithm InterpolateChild1 (Algorithm 3.7) removes variables and clauses (further on called node elements) first and inserts node elements afterwards.

The algorithm InterpolateChild1 first generates a list Δ_P containing variables and clauses which are present in the node N_P but not in its child node N_C. Afterwards it generates a list Δ_C containing the node elements which are present in N_C but not in its parent N_P. After the initialization step, it repeats copying the last node (initially N_P) and removes the first element of Δ_P from the node and the list Δ_P. If Δ_P is empty, the algorithm repeats copying the last node and inserts the first element of Δ_C into the node and removes it from the list. The algorithm terminates if the list Δ_C is empty.

In case that the node elements of the node and its child node are disjoint then the algorithm produces one empty node. Thus we propose the algorithm InterpolateChild2 (Algorithm 3.8) which adds one node element of list Δ_C to the actual node before it removes the last element remaining in Δ_P from the node.

3.4.4 Creating the datalog representation

The most important purpose is to reduce the grounding effort for the underlying datalog system by avoiding the derivation of nonrelevant facts and keeping the domains of predicates small.

We want to consider the implementation of the predicate $diffel(X, Y, el)$ for DLV Complex. The first approach could be

```
diffel(Set1,Set2,EL) :- set(Set1),set(Set2),
                        #difference(Set1, Set2, EL),#card(EL)==1.
```

#difference(Set1, Set2, EL) and #card(EL) are internal predicates defined in an external library shipped with DLV Complex (see Section 3.3.2). To avoid an unsafe rule we have to define the domain of Set1 and Set2 by adding set(Set1)

Listing 3.7: The InterpolateChild1 algorithm

```
Algorithm InterpolateChild1
(* Variables and Clauses in a node are typed as node elements *)
Input N_P parent node, N_C child node of N_P, X,C nodes
var   Δ_C List of node elements,
      Δ_C List of node elements
begin
    Δ_P := node elements that are in N_P and not in N_C;
    Δ_C := node elements that are in N_C and not in N_P;

    X := N_P;
    repeat
        C := copy(X);
        El := pop(Δ_P);
        deleteElement(C,El);
        setChild(X,C);
        X:=C;
    until (Δ_P is empty);

    repeat
        C := copy(X);
        El := pop(Δ_C);
        insertElement(C,El);
        setChild(X,C);
        X:=C;
    until (Δ_C is empty or sizeOf Δ_C = 1);

    setChild(X,N_C);
end
```

Listing 3.8: The InterpolateChild2 algorithm

```
Algorithm InterpolateChild2
(* Variables and Clauses in a node are typed as node elements *)
Input N_P parent node, N_C child node of N_P, X,C nodes
var   Δ_C List of node elements,
      Δ_C List of node elements
begin
    Δ_P := node elements that are in N_P and not in N_C;
    Δ_C := node elements that are in N_C and not in N_P;

    X := N_P;
    repeat
    if |X|> 1 then
        removeElement(X,Δ_P);
    else
        insertElement(X,Δ_C);
    end;
    until (Δ_P is empty);

    repeat
    insertElement(X,Δ_C);
    until (Δ_C is empty or sizeOf Δ_C = 1);

    setChild(X,N_C);
end

Procedure removeElement(inout X:Node, inout δ : List)
begin
    C := copy(X);
    El := pop(Δ);
    deleteElement(C,El);
    setChild(X,C);
    X:=C;
end

Procedure insertElement(inout X:Node, inout δ : List)
begin
    C := copy(X);
    El := pop(Δ);
    insertElement(C,El);
    setChild(X,C);
   X:=C;
end
```

and set(Set2). This results in a calculation of the cross product of set Set1 and set Set2. During generating we replace it with the term #difference(Set1, Set2, EL),#card(EL)==1 and avoid this overhead.

The resulting datalog representation is obtained by merging three different parts:

1. The monadic datalog program
2. The normalized tree decomposition
3. The structure of the propositional formula

The datalog rules representing the normalized tree decomposition are generated out of the tree decomposition and are merged with the rules of the formula (see Section 3.4.2) into the base data structure of the monadic datalog program. Next we generate necessary internal predicates. Recall the rule for the leaf node of the SAT program in Listing 3.1.

Listing 3.9: SAT program's rule for the leaf node

```
/* leaf node. */
solve(v,P,N,C1) ← leaf(v),bag(v,X,C),P ∪ N=X,
                  P ∩ N=∅, true(P,N,C1,C).
```

The set N and the set P are disjoint subsets of X i.e. they form all possible partitions of X with two elements. It is easy to see that all possible values of P are all elements of the power set of X and the values of N are their corresponding complements with $N = X \setminus P$. Thus we generate these power sets and their complements for the variables and clauses in every bag, which is more efficient than creating them globally for all node elements. The result of this operation are new extensional predicates $pset(v, pvs)$ and $cset(v, pcs)$ where v is the bag, pvs and pcs are power sets of the variables X respectively the clauses C of the bag. For further statistics we generate an unary predicate $treewidth(tw)$ with tw as the treewidth of the tree decomposition.

3.4.5 Converting set variables

In Section 3.1 we defined that the element variables are denoted with lower case letters and set variables are denoted with upper case letters in the monadic datalog program. DLV and DLV-C are using upper case letters for all variables. Thus we have to loop through the symbol table containing the variables and convert their identifiers from lower case to upper case. If an element and a set variable

have the same name, a naming conflict can occur during the conversion. We resolve such a conflict by renaming the element variable. With an optimal data structure e.g. using a symbol table and a handle for every identifier the renaming is not a big effort.

3.4.6 Additional steps for DLV

DLV does not support set semantics, thus all sets have to be converted into datalog rules. We represent a set as datalog rule by introducing the predicate $elementOf(s, el)$ where s is a constant representing the name of a set and el is an element of the set s. We generate such a predicate for every set and for every element of a set, as shown in Example 3.4.1.

Example 3.4.1 *Let $s = \{a, b, c, d, e, f\}$ be a set, which can be expressed in normal datalog rules for DLV with the following rules*
elementOf(s,a). elementOf(s,b). elementOf(s,c).
elementOf(s,d). elementOf(s,e). elementOf(s,f).

The empty set is represented as the constant *empty*. In addition we provide set operations as datalog rules as follows:

Listing 3.10: Set operations for DLV

```
%
% definition of a set
%
set(A) :- element_of(A,_).
set(emptyset).
%
% union of two sets
%
union(S1,S2,E) :- set(S2),element_of(S1,E),S1!=S2.
union(S1,S2,E) :- set(S1),element_of(S2,E),S1!=S2.
%
% intersect: intersection of two sets
%
intersect(S1,S2,E) :- element_of(S1,E),element_of(S2,E),
                      S1!=S2.
```

The definition of diffel is

Listing 3.11: Definition of diffel for DLV

```
%
% diffel: difference of two sets
%
diffel(S1,S2,E) :- set(S1),set(S2), element_of(_,E),
                   not intersect(S1,S2,E),S1!=S2.
```

3.4.7 Additional steps for DLV Complex

In contrast to DLV, its extension DLV Complex supports set semantics. Therefore the predicates for the set operations have to be replaced by their equivalent of DLV Complex, as we already mentioned in Section 3.4.4. These external predicates are defined in the library *ListAndSet* shipped with DLV Complex. Thus, we have to insert the include directive #include<ListAndSet> for the external library. The complete datalog program is shown in Appendix A.2.

3.4.8 Additional steps for DLV Complex with external predicate

In the previous section we used the predicate *true* defined as intensional predicate in datalog. In this section we present an external predicate "true" as $\#true(P, N, C1, C, POS, NEG)$.

As mentioned in Section 3.3.2, the name of the external *true* predicate has to start with a "#" sign. Furthermore in the external library we have no access to the extensional predicates and their domains, so we extend the signature of *true* with two set variables i.e. POS and NEG representing set of sets. These sets model the structure of the propositional formula. More precisely, if a propositional variable x occurs positive in a clause c then POS will contain the set x, c. If x occurs negative in c then NEG will contain x, c. Example 3.4.2 illustrates the set POS and NEG for the formula of Example 2.2.1

Example 3.4.2 *Sets of sets for formula of Example 2.2.1*

$$POS = \{\{c0,x2\},\{c2,x6\},\{c2,x3\},\{c0,x3\},\{c0,x1\},\{c2,x4\},\{c1,x5\}\}$$
$$NEG = \{\{c1,x2\},\{c1,x4\}\}.$$

First, for the external libraries we have to insert the *include* directives which are #include<ListAndSet>, #include<exttrue>. Afterwards we replace all occurrences of $true(P, N, C1, C)$ with the notation of the external predicate $\#true(P, N, C1, C, POS, NEG)$. Furthermore the sets POS and NEG have to be created by looping over all instances of the predicates *pos* and *neg*.

Recall the rule for the leaf node of the SAT program in Listing 3.1, the translated datalog rule with the sets is presented in Listing 3.12, the complete datalog program is shown in Appendix A.3

Listing 3.12: Leave rule for DLV Complex with external predicate

```
solve(V,P,N,C1):-leaf(V),bag(V,X,C),
                 #true(P,N,C1,C,POSSET,NEGSET),
                  posset(POSSET),negset(NEGSET),
                 X=#union(P,N),
                 #intersection(N,P,{}).

posset({{c0,x2},{c2,x6},{c2,x3},{c0,x3},{c0,x1},{c2,x4},{c1,x5}}).
negset({{c1,x2},{c1,x4}}).
```

3.4.9 Summary

The focus lies on the representation of the sets, the set operations and the predicate $true(P, N, C1, C)$. The way of the generation of datalog rules participates in the success of a fast evaluation. DLV has no set semantics thus the sets have to be translated into normal datalog rules and the set operations $\cup \cap \uplus$ have to be implemented as intensional predicates. In DLV Complex it is possible to define the predicate *true* in an external library. Beside the automatically generated internal predicates (e.g. *root*, *leaf*, *child*1, *child*2, *bag*, *pos*, *neg*, *cset*, *pset*) there are some internal predicates, most defined in an additional file, which are independent from the structure of the CNF formula.

Chapter 4

Implementation

In this chapter we go into more detail regarding the implementation of the evaluation, the transformation process and the test environment. We start with an overview in Section 4.1, next we present the architecture of our application Dattrans in Section 4.2. Section 4.3 provides information how the input is generated and processed. Used data structures are presented in Section 4.4 and in Section 4.5 we discuss the implementation of the tree decomposition and the normalization of the tree. Section 4.7 gives an insight into the conversion of the datalog rules. The implementation of the external true predicate is shown in Section 4.8.

4.1 Overview

We have implemented a prototype implementation of the transformation process as a preprocessor for the underlying datalog system. Figure 4.1 gives an architectural overview over the system for our experiments. We call it Dattrans. The preprocessing approach allows us to generate datalog rules for different evaluation scenarios as we mentioned in Section 3.3.

Our implementation and testing environment consists of the following components:

1. Mkcnf [1]
2. Dattrans
3. Datalog engine (i.e. DLV and DLV Complex)
4. Evaluation script

[1] http://www.cs.ucsc.edu/~avg/software.html

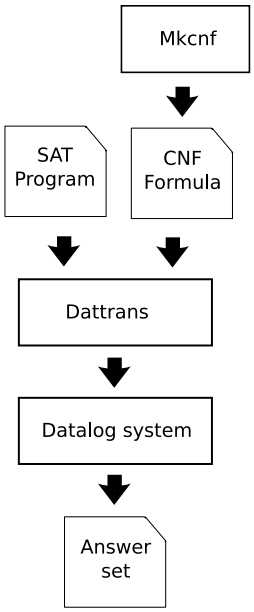

Figure 4.1: Implementation overview

5. Minisat [2]

At first we use *Mkcnf* to generate a file containing the CNF formula (see Section 4.3.1). Next our application Dattrans takes as input two files, the propositional formula and the SAT program, to generate the datalog rules for the destination datalog system. The whole process is controlled by a script which takes care about the calling order and the time measurement. After the datalog system has finished the evaluation the control script produces the necessary statistical records.

4.2 The preprocessor Dattrans

Within the scope of this thesis we developed the command line application *Dattrans*, which is a preprocessor for the requested datalog subsystem. The architecture of the application is shown in Figure 4.2. It takes two input files, the CNF formula and the monadic datalog program, and transforms the monadic datalog rules into the datalog rules for the destination system by processing the steps we described in Section 3.4. Dattrans takes the following command line options:

The command line option *-prog* specifies the name of the file containing the monadic datalog rules (i.e. the SAT Program of Listing 3.6). By default Dattrans reads the propositional formula via stdin, which is useful if shell redirection is used, e.g. in conjunction with *Mkcnf*. Alternatively the filename of the CNF file

[2] http://minisat.se/

```
$> dattrans [options] -prog <file> [-cnf <file>]
        -prog Datalog file
        -cnf DIMACS file (default stdin)
        options:
                        -c generate file for dlv-c
                        -e generate file for dlv-c using
                                external #true (overrules option c)
                        -h displays this message
```

can be specified using the command line argument *-cnf*. The output is written to stdout and if it is necessary it must be redirected via shell into a file. The destination datalog system can be specified with the options *-c*, *-e* or without an option. If this option is omit, the Dattrans will produce the output for DLV, if option *-c* is specified, it will create the output for DLV Complex and with option *-e* the output is generated for DLV Complex using the external true predicate. For a usage message the switch *-h* must by specified. For implementational details please visit the next sections.

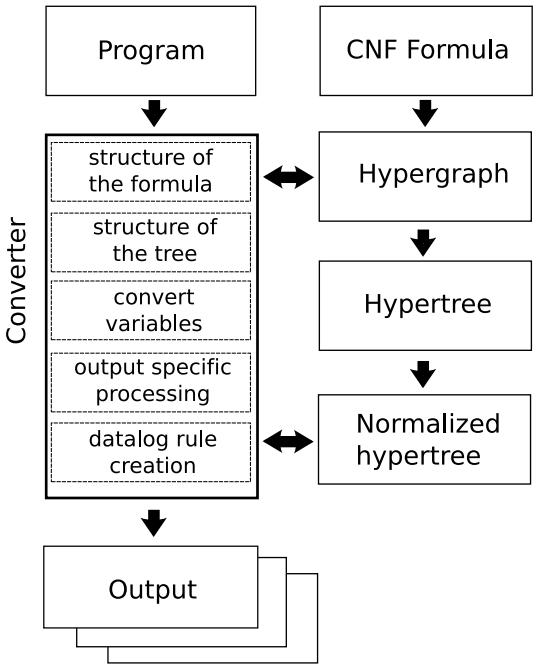

Figure 4.2: Architecture of the transformation module

4.3 Input

As mentioned in the previous section, our implementation takes two input files, the CNF formula using the DIMACS graph format (see Section 4.3.1) and the monadic datalog program shown in Listing 3.6.

4.3.1 DIMACS

The propositional formula is represented using the well known DIMACS graph format for satisfiability problems (see [1]). This graph coloring format is a simple ASCII text file containing several lines, which are terminated by an end-of-line character. Each line begins with a one-character identifier which specifies the line type. The fields in one line are separated by at least one blank space.

1. Comments. The comment line starts with the character c. It gives a human-readable information about the file and is ignored by programs. A comment line can appear anywhere in the file.

   ```
   c This is a comment
   ```

2. Problem line. The problem line starts with the character p and has following format

   ```
   p FORMAT VARIABLES CLAUSES
   ```

   where FORMAT must be cnf for satisfiability problems, $VARIABLES$ takes an integer n specifying the number of variables and CLAUSES specifies the number of clauses. One file contains only one problem line and it must appear before any clause line.

3. Clause line. The clause line appears immediately after the problem line. The variables are assumed to be numbers by an integer of $1...n$. Not every variable must appear in every clause. A variable that occurs positive (non-negated) in a clause is represented by i, the negative (negated) variable is represented by $-i$. Each clause is terminated by the value 0 to support multiple clauses per line.

Our DIMACS reader reads the text file (see Example 4.1) and provides the clauses for the next module, which builds the primal graph of the structure of the formula.

At the first step of our process we generate the CNF formula automatically using the random CNF formula generator for DIMACS format called *Mkcnf* [3]. The usage of Mkcnf shows the following command line options: where *vars* represents the number of variables, *clauses* the number clauses and *clauseLen* the length of one clause. The option seed is initially set to 50 by default, in our case

[3] http://www.cs.ucsc.edu/~avg/software.html

Listing 4.1: DIMACS graph format for satisfiability problems for Example 2.2.1

```
c
c SAT program
c
p cnf 6 3
 1   2   3   0
-2  -4   5   0
 3   4   6   0
0
```

```
$>mkcnf vars clauses clauseLen [seed] [-f | -u] [out_file]
```

we always use 1. The option -f forces the formula to be satisfiable, by default it is -u (unforced). In this thesis we use propositional formulas with the length of three for clauses and the number of variables are three times the number of clauses. Thus our command line for *Mkcnf* looks as follows

```
$>mkcnf 3*clauses clauses 3 1 -f [out_file]
```

4.3.2 Monadic datalog parser

The monadic datalog parser of Dattrans is developed using the scanner generator *flex (version 2.5.34)* [4] and the parser generator *GNU Bison (version 2.3)* [5]. Readers who are interested into the grammar files see Appendix B. The parser reads the input file, checks the semantical correctness and builds a parse tree which is converted into the appropriate data structure (see Section 4.4). The parse tree is also used to check for semantic errors. If the parser finds an error the application terminates and writes an appropriate error message to the output.

The parse tree is a binary tree consisting of nodes of the class `ParseTreeNode`. Every node has a link to its parent, a node type and a value, which stores the value of a constant (e.g. integer, a string,...) or the name of a variable. The node also stores the line number and the position of the first character of the token in the source file, to generate readable error messages. The parser maintains a global stack. The function `popErrorStack` takes as argument a pointer to a node of the parse tree, pops it on the top of the stack and returns the pointer. This stack acts as a garbage collector especially if for example an error occurs.

[4] http://www.gnu.org/software/flex/
[5] http://www.gnu.org/software/bison/

4.4 Data structures

A fast data structure is the key for an efficient and correct processing. In this section we present the essential part of the data structures (Figure 4.3). For a better readability, we omit special pointers like short cuts to parent and other classes.

- `Term` - a class representing constants, variables and numbers. The class holds a vector containing its parents for fast access to them.
- `Set` - subclass of the term class representing a set of terms.
- `Instance` - is an instance of a term for a variable.
- `SetInstance,MultisetInstance` - are subclasses of the class instance representing an instance for a set of terms respective set of sets.
- `InstanceSet` - a container holding a list of instances.
- `Predicate` - represents the predicates. Every predicate has a name and an arity and holds a set of instances, the domain of the predicate. Every predicate has a list of literals in which it appears.
- `Literal` - consists of a predicate and a number of arguments of terms. The number of arguments is equal to the arity of the predicate. The literal contains a list of rules in which it appears.
- `Rule` - represents a datalog rule and consists of the rule head, positive body and the negative body.
- `Program` - represents the datalog program. It consists of a set of rules. Every program has its own symbol tables containing the predicates, terms, sets and variables used in this program. This is useful if for example the name of a variable has to be changed.
- `Symbol table` - a class which stores the symbols fast accessible in a hash-map.

4.5 Hypertree library

To compute the tree decomposition of the primal graph of the τ-structure representing the formula we use the hypertree library [6] proposed in [11]. The hy-

[6] http://www.dbai.tuwien.ac.at/proj/hypertree/

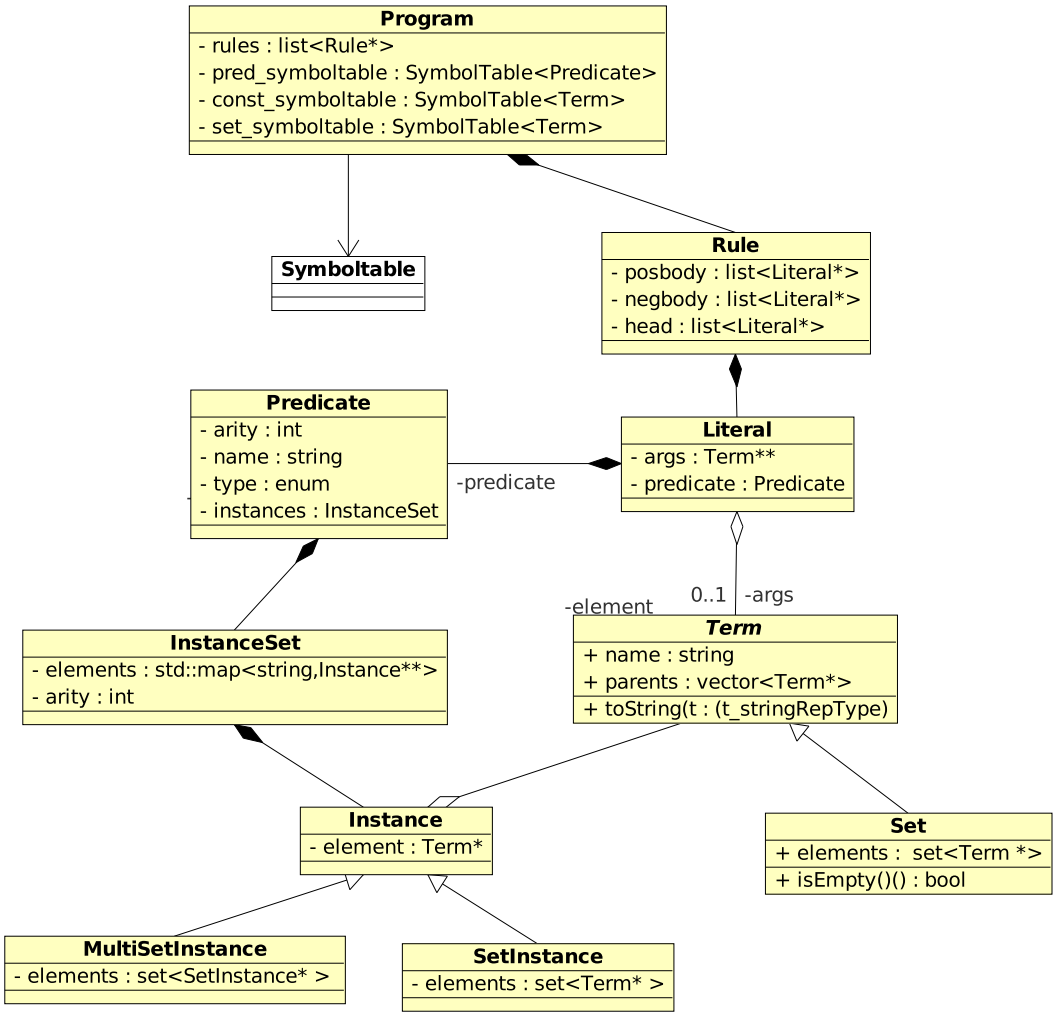

Figure 4.3: Data structures of the application Dattrans

pergraph for our running Example 2.2.1 is shown in Figure 4.4. We decided to integrate the library in our application to have direct access to the hypergraph and hypertree. It is also possible to create an input file containing the hypergraph for the application shipped with the hypertree library and parse the output to create the decomposed hypertree (see Figure 4.5).

The hypertree library uses serval heuristic methods to find a small treewidth (for further information see [11]). In our application the treewidth is calculated by applying bucket eliminations using following heuristics:

- Maximum cardinality search order heuristic
- Minimum induced width order heuristic
- Minimum fill-in order heuristic

The resulting treewidth is the minimum of all calculated treewidths by the heuristics above.

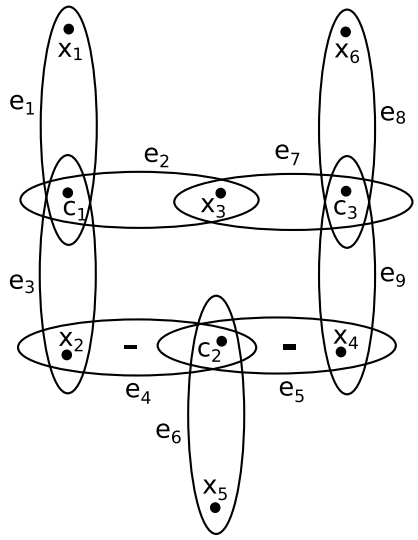

Figure 4.4: Hypergraph of Example 2.2.1

The hypertree library provides methods to check if it is a valid tree decomposition according to Definition 2.3.1. We use this methods to check if the decomposed tree is valid. In this thesis we are only interested in the tree decomposition of the primal graph, due to this we are ignoring the hyperedges (in Figure 4.4 e_1 ... e_9) and are using the nodes of the hypergraph (in Figure 4.4 x_1 ... x_6 and c_1 ... c_3). The hypertree library provides us the hypertree-width and the treewidth, we are using the treewidth.

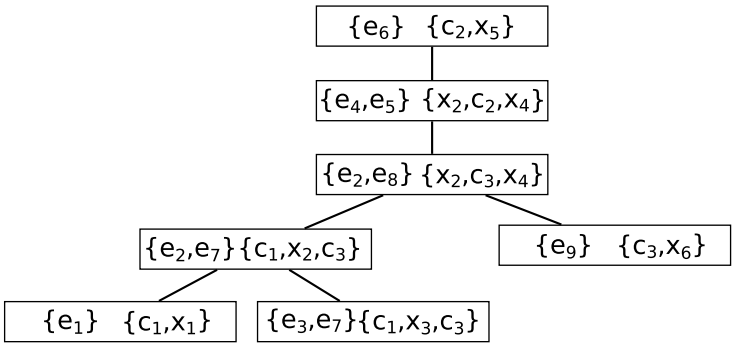

Figure 4.5: Tree decomposition of the hypergraph of Example 2.2.1

4.6 Normalization

After a valid tree decomposition is calculated, the tree is normalized as described in Section 3.4.3 and this is done by the class named `HypertreeNormalization`. Therefore the rooted hypertree is traversed in a preorder manner. The most important methods are shown in Figure 4.6.

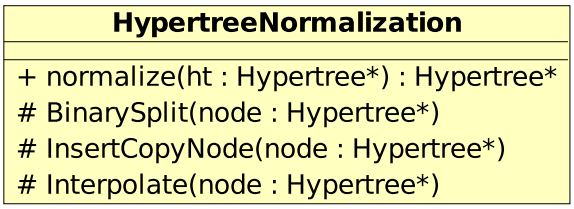

Figure 4.6: Class HypertreeNormalization

The only public method is `normalize`, which takes the root of the hypertree, normalizes it and returns the node of the tree. Each of the protected methods takes as argument a node of the hypertree and models one single step of the normalization algorithm:

- `BinarySplit` - carries out the binary split if necessary,
- `InsertCopyNode` - inserts a copy node if necessary,
- `Interpolate` - interpolation between the node and its children.

The normalized hypertree is shown in Figure 4.7. Once the hypertree is calculated, normalized and validated, we create the tree structure and merge it into the data structure created in Section 4.3.2.

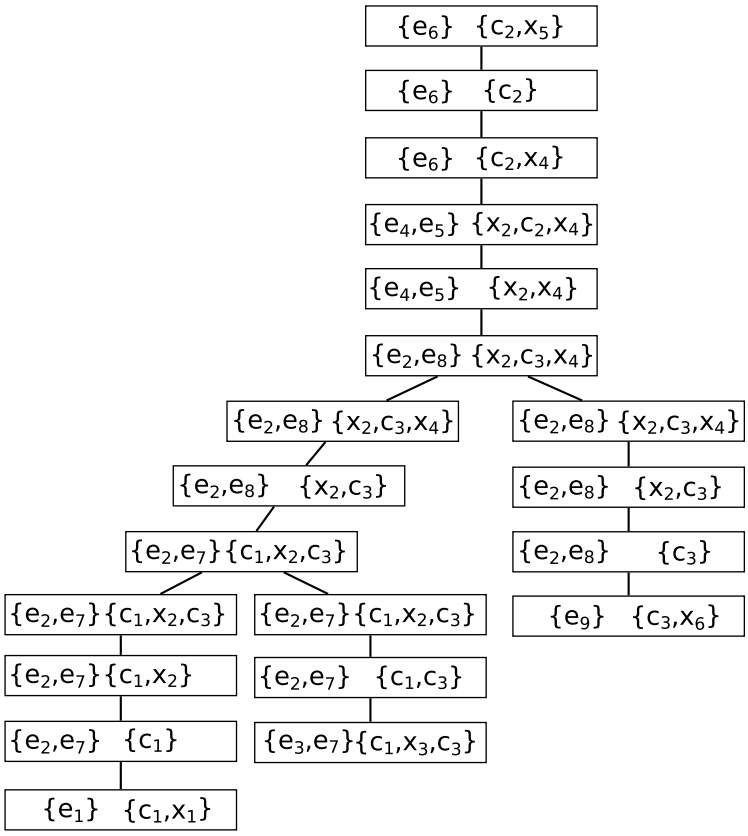

Figure 4.7: Normalized tree decomposition of the hypergraph of Example 2.2.1

4.7 Converting the datalog rules

The converter module converts the information held in the data structure (described in Section 4.4) into the requested output format and has the following tasks:

- Create datalog representation of the structure of the propositional formula,
- Create datalog representation of decomposed and normalized tree,
- Convert and create output dependent predicates and rules.

Generally converter module first creates the instances of the appropriate predicate and after all instances for all predicates are calculated, the rules are generated out of the instances. The objects for the predicates are created at the moment of the instantiation of the object program. If a predicate has no instances, no rules will be created and thus no output will be generated.

The Hypergraph (see Figure 4.4) consists of a list of nodes and a list of edges. It is used to create the instances of the predicates *c*l, *v*ar, *p*os and *n*eg, which are representing the structure of the propositional formula (recall Example 2.2.3).

Therefore the converter visits all nodes of the graph and creates an instance either for the predicate *c*l or for the predicate *v*ar depending on the type of the node. After that it visits all edges and creates the instances either for the predicate *p*os or *n*eg, depending on the sign of the edge. In case the DLV Complex with external predicate *t*rue is used as destination system the converter creates the instances for the predicates *p*osset and *n*egset (see Listing 3.12).

To create the instances of the predicates modeling the structure of the decomposed and normalized tree, the tree is traversed in a preorder manner. For every visited node the converter module creates an instance of the predicate *b*ag. If the node is a leaf node an instance of the predicate *l*eaf will be created and if the node is the root of the tree the instance for the predicate *r*oot will be created by the converter module. During the tree traversal for every parent child relation the converter module creates an instance of the predicate *c*hild1 or *c*hild2, depending whether the child is the only, first or second child. Finally it creates the instance for the predicate *t*reewidth. Next it loops through all instances of the predicate *b*ag and generates the instances for all predicates, which represent the partitions of the sets of variables and clauses.

In case the destination system is DLV the converter converts all instances which are sets into datalog rules next. Therefore it loops through all instances of all predicates, which have sets as arguments (e.g. *b*ag or *p*artition), creates the instances of *e*lementOf (see Example 3.4.1) and replaces the set instance of the predicate with the term of the name of the set.

If all instances are calculated the converter creates the rules out of the instances. This is done by looping through all instances of the predicates and creating a new literal and new rule for every instance. In our case all resulting rules are facts consisting of one literal in their head. The remaining conversions are done when the rules are converted to strings for the output. An example for such a conversion is the usage of an external true predicate. In the SAT program the notation of $true$ has the following form $true(v, P, X, C1, C)$ and the output $\#true(P, X, C1, C, POSSET, NEGSET)$ is needed for the external true predicate.

4.8 External predicate "true"

We have implemented the external "true" predicate in C++ using the SDK shipped with DLV Complex. The predicate is provided to DLV Complex as system library containing mainly two functions:

```
bool true_iiiiii(CONSTANT* argv, unsigned int argc);
```

```
bool true_iioiii(CONSTANT * argv, unsigned int argc);
```

The class constant defined in the SDK contains the value, the type of the argument and some useful operators (e.g. <,>,...). The type of an argument can be an integer, a symbol, a string or a null constant (we only use the types integer and symbol). As parameters the two functions take an argument vector and its size. The function `true_iiiiii` is called if all parameters are constants or variables, which are save in the rule. In this case all arguments are input parameters and the function decides if the given instance is valid or not. The function will return *true* if the instance is valid otherwise it will return $false$. If the third argument is a variable, which is not save, then DLV Complex will expect an output value for this variable, otherwise it terminates with an error.

Chapter 5

System test and experimental results

Primarily we are interested in the runtime performance of the evaluation of the datalog system. We measure the processing time, the number of bags in the normalized tree and the number of generated *true* predicates, which are ignored if the destination system is DLV Complex with external "true" predicate, because in this case no true predicates are generated. The size of the input is measured by the number of variables plus the number of clauses of the propositional formula.

Every input is evaluated three times by the whole process and the control script calculates the average over the measured records. As DLV and DLV Complex are closed source, we could not use the internal timing functions, thus our time measurement is done by the gnu version of Unix tool *time* using the call which tells us the time needed to execute a command.

```
$ >/usr/bin/time -o <output file> -a -f \%U <commmand> 2>\&1
```

The experiments were conducted on Linux kernel 2.6.24-21 with a 2.66GHz E6750 Intel®Core2Duo™CPU and 3 GB of memory.

5.1 Testing

If a formula is satisfiable, the predicate *s*uccess will be derived by the datalog system. In this section we present our testing environment. We mentioned in Section 4.2 that the input e.g. generated from *M*kcnf can be passed to our application via shell redirection. We do not use this feature yet, because we have to test if our result is correct. Instead we store the generated DIMACS output of

*M*kcnf in a separate file and use it as input for *M*inisat [1]. Afterwards we compare after every run the result of *M*inisat and the derivation of predicate *s*uccess. As *m*inisat is a widely used and tested application, we consider that *m*inisat provides correct results and we are able to use it as a reference to test our application.

5.2 Experimental Results

In this section we present the experimental results of our implemented system. In the tables and figures below, we show our results of the three evaluation methods, DLV, DLV Complex and DLV Complex with external predicate.

In this thesis we did not want to compete with an implementation in C++, rather we wanted to examine a practical evaluation of the approach proposed in [15], as a general-purpose method which allows to replace the underlying problem. Therefore we do not compare our results with an implantation in C++. Our purpose was comparing the results of the three evaluation methods. All tables have five columns representing the measured values, where *variables* stands for the amount of input variables, *clauses* for the amount of input clauses, *nodes* for the number of generated bags in the normalized decomposed tree, *true* for the number of generated *true* predicates by the datalog system and finally *time* stands for the measured time. To create our plots, we used Gnuplot [2]. Appendix C lists the scripts used to produce the graphs in this section. Following we describe our practical experiments and report some lessons learned.

As DLV does not support set semantics, the encoding of the sets and its operations was the biggest challenge. We had to keep an eye on avoiding cycles and minimizing the amount of derived predicates during grounding. As we can see from Table 5.1, Table 5.2 and Table 5.3 there is a considerable number of the predicate true. This is an effect of the encoding of the set semantics, thus DLV has much grounding effort.

DLV Complex in contrast to DLV has less grounding effort. As DLV and DLV Complex are closed source we could not determine how the set operations are implemented and how much grounding effort is caused by the sets. Obviously, the solution using DLV Complex is more performant than with DLV plain. In Table 5.4, Table 5.5 and Table 5.6 we show the results of our experiments using DLV Complex and treewidth of 3, 4, and 5. We got no surprise with DLV Complex using an external true predicate the best results, because the external true predicate is only evaluated if needed. In the previous approaches all possible

[1] http://minisat.se/

[2] Gnuplot is a standard plotting program available from http://gnuplot.sourceforge.net/.

instances of the predicate were derived, no matter if they were evaluated in some rule. In Table 5.7, Table 5.8 and Table 5.9 we present the results with treewidth 3, 4 and 5.

variables	clauses	nodes	true	time
6	2	9	552	0.12
15	5	20	2112	0.19
21	7	24	4105	0.38
24	8	26	8401	1.60
39	13	43	10184	3.91
42	14	50	15339	5.61
45	15	55	30683	7.98
51	17	61	32326	11.02
54	18	68	53678	14.5

Table 5.1: Measurement for DLV with treewidth 3

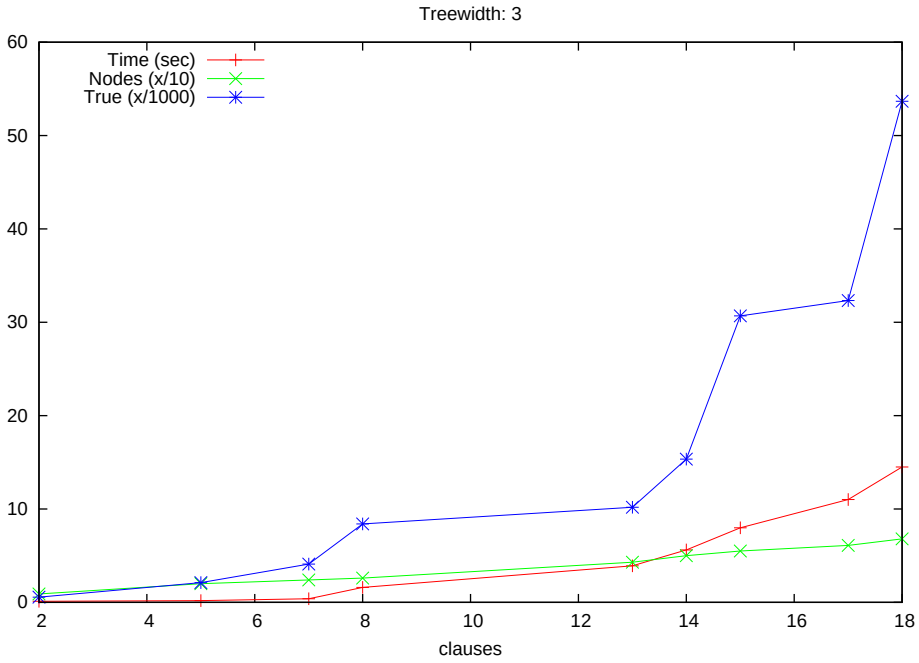

Figure 5.1: Results for DLV with treewidth 3

5.3 Summary and evaluation of results

Concluding this chapter we can say that we achieved best performance with DLV Complex using an external true predicate. As mentioned in Section 3.3.2 DLV Complex is a very young project and hopefully it will provide more useful features

9	3	18	2134	1.12
27	9	29	9114	2.33
30	10	39	10763	5.22
33	11	39	12652	7.27
36	12	46	17163	11.33
48	16	53	26891	16.14
57	19	64	29541	20.28
69	23	73	80185	35.17
84	28	97	99118	45.88
90	30	92	101693	50.91

Table 5.2: Measurement for DLV with treewidth 4

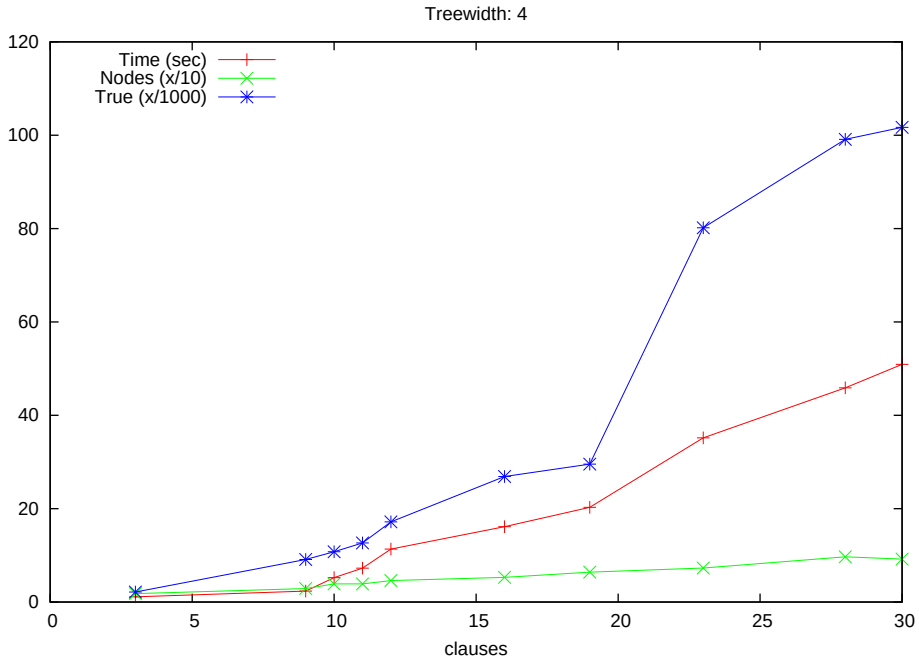

Figure 5.2: Results for DLV with treewidth 4

variables	clauses	nodes	true	time
60	20	72	82362	14.09
63	21	72	95032	19.23
66	22	70	112668	35.43
78	26	91	139836	55.22
87	29	92	173721	67.64
96	32	112	296183	73.23
105	35	115	314353	88.54

Table 5.3: Measurement for DLV with treewidth 5

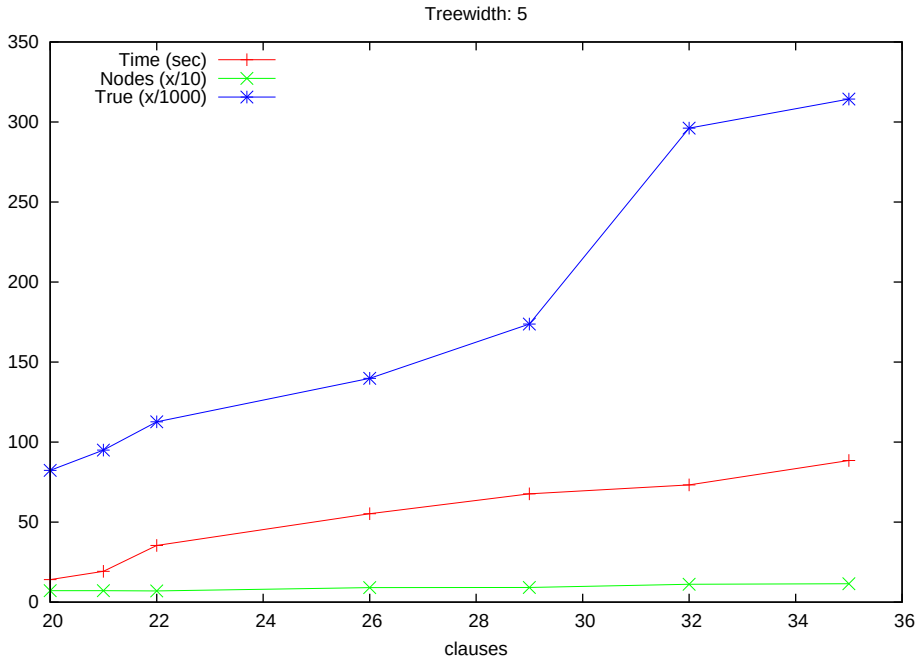

Figure 5.3: Results for DLV with treewidth 5

variables	clauses	nodes	true	time
6	2	9	252	0.04
15	5	20	1607	0.09
21	7	24	2005	0.14
24	8	26	2410	0.3
39	13	43	7184	0.83
42	14	50	9339	0.98
45	15	55	8683	1.58
51	17	61	12326	2.03
54	18	68	13678	4.17

Table 5.4: Measurement for DLV Complex with treewidth 3

variables	clauses	nodes	true	time
9	3	18	1454	0.08
27	9	29	8304	0.3
30	10	39	5864	0.42
33	11	39	6152	0.57
36	12	46	12563	1.73
48	16	53	14891	2.84
57	19	64	22345	3.25
69	23	73	25889	5.16
84	28	97	64548	23.28
90	30	92	42693	15.78

Table 5.5: Measurement for DLV Complex with treewidth 4

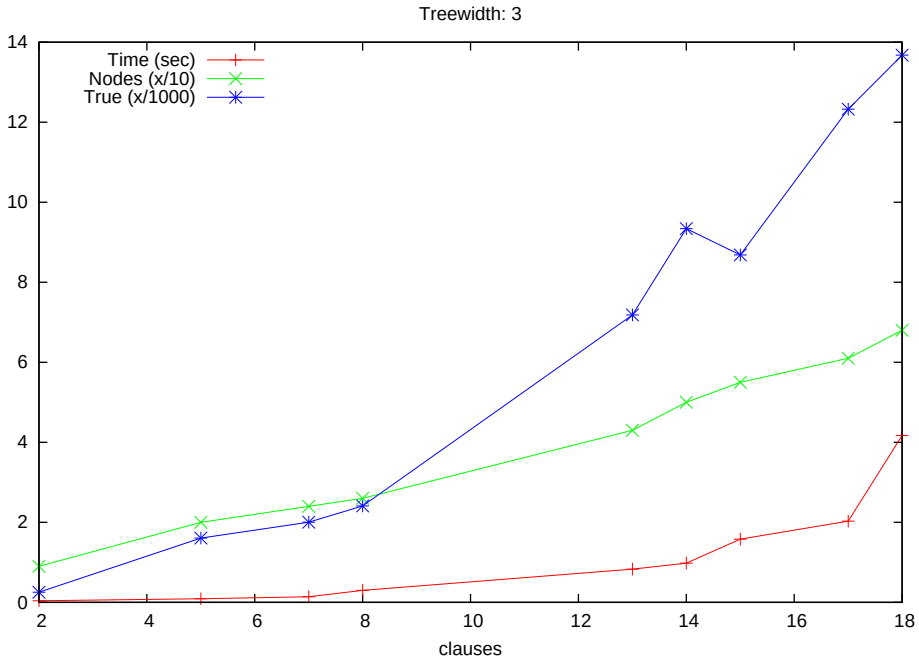

Figure 5.4: Results for DLV Complex with treewidth 3

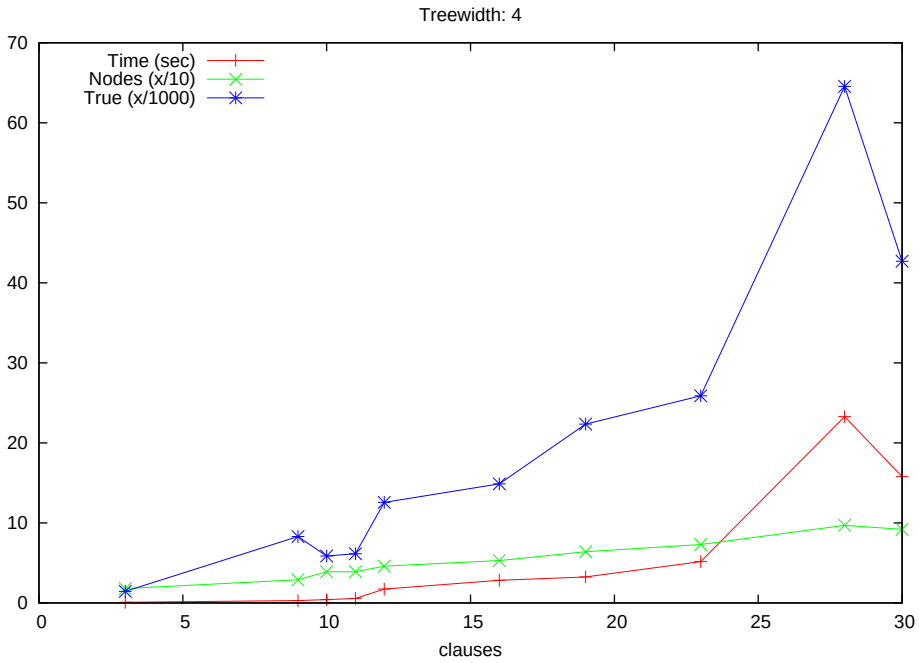

Figure 5.5: Results for DLV Complex with treewidth 4

variables	clauses	nodes	true	time
60	20	72	42261	14.26
63	21	72	43033	6.67
66	22	70	22658	4.67
78	26	91	99306	18.39
87	29	92	67021	18.31
96	32	112	136085	43.25
105	35	115	94529	50.3

Table 5.6: Measurement for DLV Complex with treewidth 5

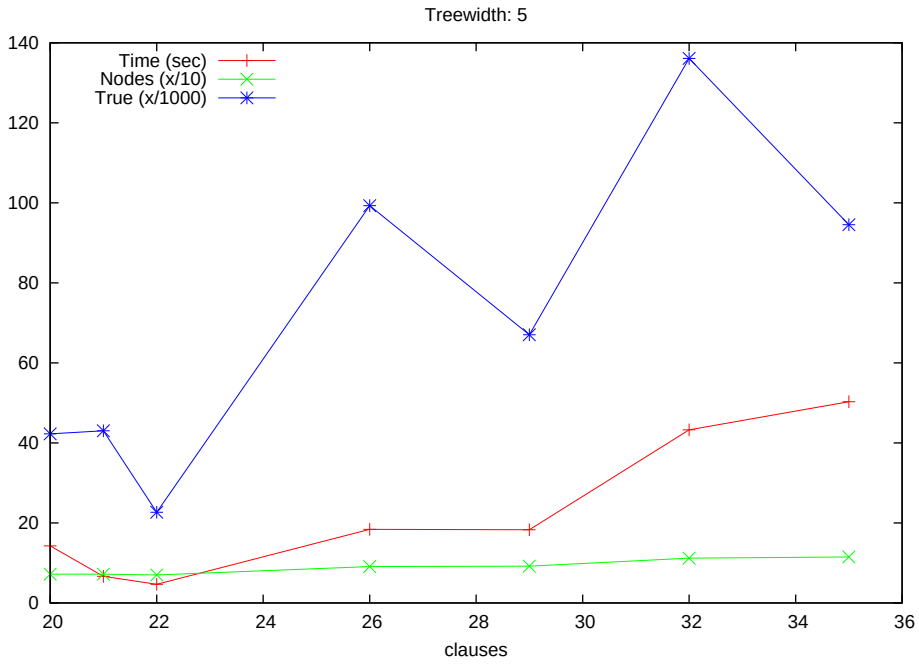

Figure 5.6: Results for DLV Complex with treewidth 5

variables	clauses	nodes	true	time
6	2	9	0	0.03
15	5	20	0	0.1
21	7	24	0	0.22
24	8	26	0	0.18
39	13	43	0	0.57
42	14	50	0	2.96
45	15	55	0	4.46
51	17	61	0	6.58
54	18	68	0	10.17

Table 5.7: Measurement for DLV Complex with ext. true and treewidth 3

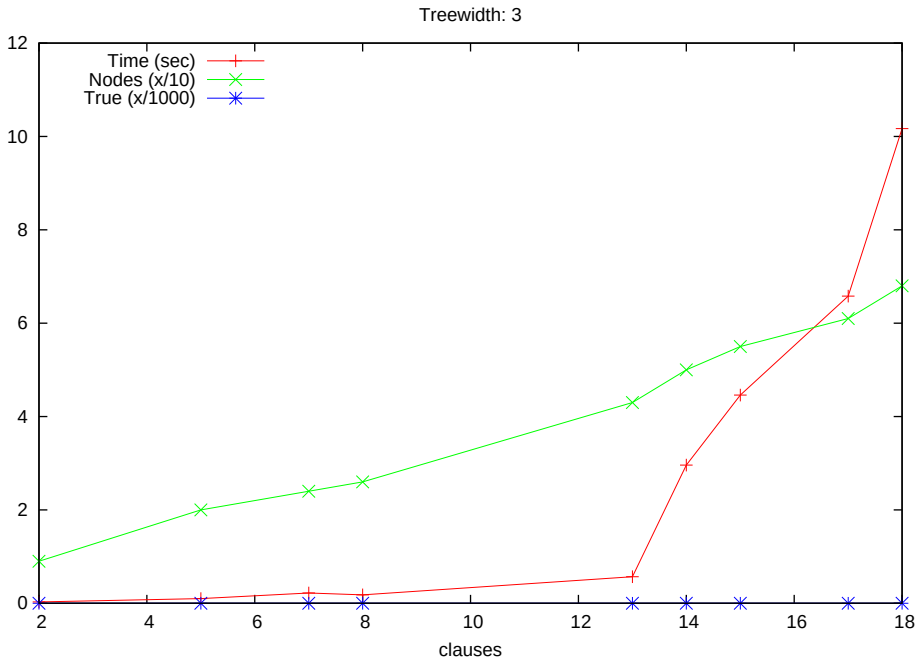

Figure 5.7: Results for DLV Complex with ext. true with treewidth 3

variables	clauses	nodes	true	time
9	3	18	0	0.04
27	9	29	0	0.14
30	10	39	0	1.1
33	11	39	0	1.31
36	12	46	0	1.49
48	16	53	0	0.68
57	19	64	0	1.5
69	23	73	0	2.2
84	28	97	0	5.03
90	30	92	0	4.68

Table 5.8: Measurement for DLV Complex with ext. true and treewidth 4

variables	clauses	nodes	true	time
60	20	72	0	2.89
63	21	72	0	2.05
66	22	70	0	2.0
78	26	91	0	4.29
87	29	92	0	5.04
96	32	112	0	8.08
105	35	115	0	9.06

Table 5.9: Measurement for DLV Complex with ext. true and treewidth 5

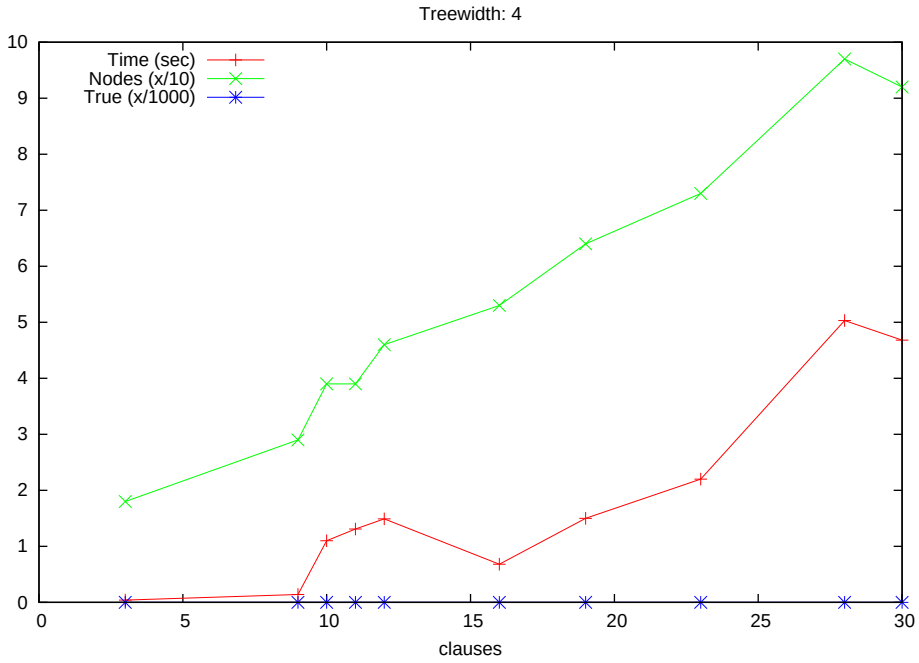

Figure 5.8: Results for DLV Complex with ext. true with treewidth 4

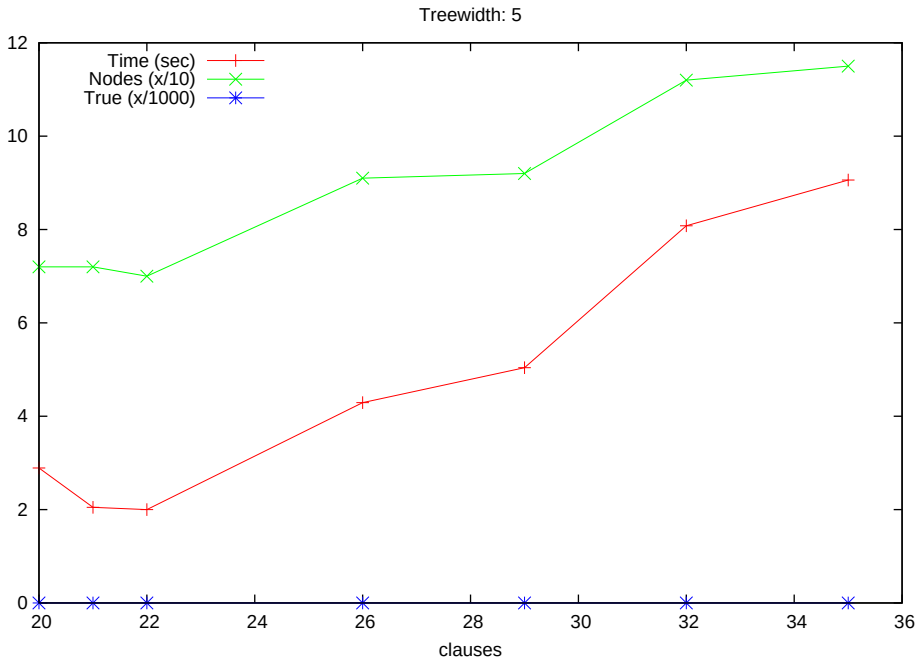

Figure 5.9: Results for DLV Complex with ext. true with treewidth 5

in this direction. The set semantics of DLV Complex is already convenient and raises the performance. Unfortunately the grounding effort is still high.

Chapter 6

Conclusion

In this chapter we summarize the results presented in this thesis and give some remarks and directions for further improvements. The goal of this thesis was not an efficient algorithm for the SAT problem, instead we were aiming to separate the underling problem and the evaluation process. This allows us to replace the underlying problem with others, e.g. the solvability problem, of abduction, circumscription problem, etc. whose FPT was established via Courcelle's Theorem [7]. In Chapter 2 we have revisited the approach to express the MSO with monadic datalog over finite structures as proposed in [15]. In Chapter 3 we have presented the basic idea of a preprocessor for a datalog system, which converts the monadic datalog program and a given CNF formula into a "normal" datalog program for the destination datalog engine. We have actually implemented a prototype of our preprocessor, which we have presented in Chapter 4. We have experimented with three approaches (DLV, DLV Complex, DLV Complex with an external predicates) to evaluate the transformed datalog rules and have presented the results. In particular, we have tested the feasibility to create a more generic system than an implementation using an imperative programming language. In fact we think this approach points into a promising direction and we have discovered that there is room for further improvements.

6.1 Future Work

A future improvement of the process would be to develop an automatic transformation from MSO to monadic datalog, as in [15] the proposed SAT program was built by hand.

In [17] it is mentioned that disjunctive logic programming is strictly more expressive than normal (disjunction-free) logic programming. Another possible

goal of future work is to evaluate if the use of disjunctive logic programming in the monadic datalog approach will enhance the performance of DLV. As we mentioned in the previous sections, the main goal is to reduce the grounding overhead for DLV and to use extensively external predicates in DLV Complex to improve the performance. This could be achieved by an efficient implementation of predicates as external predicates, as we have implemented the external "true" predicate.

Bibliography

[1] Home page of center for discrete mathematics and theoretical computer science (dimacs). `http://dimacs.rutgers.edu/Challenges/`.

[2] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybern.*, 11(1-2):1–22, 1993.

[3] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.

[4] Hans L. Bodlaender. Discovering treewidth. In *SOFSEM'05*, pages 1–16, 2005.

[5] Francesco Calimeri, Susanna Cozza, and Giovambattista Ianni. External sources of knowledge and value invention in logic programming. *Annals of Mathematics and Artificial Intelligence*, 50(3-4):333–361, 2007.

[6] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC'71*, pages 151–158, 1971.

[7] B. Courcelle. Graph rewriting: An algebraic and logic approach. In *Handbook of Theoretical Computer Science*, volume B, pages 193–242. Elsevier Science Publishers, 1990.

[8] B. Courcelle, J. A. Makowsky, and U. Rotics. On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic. *Discrete Appl. Math.*, 108(1):23–52, 2001.

[9] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. In *IEEE CCC'97*, pages 82–101, 1997.

[10] Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerald Pfeifer. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in dlv. In *IJCAI'03*, pages 847–852, 2003.

[11] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Benjamin J. McMahan, Nysret Musliu, and Marko Samer. Heuristic methods for hypertree decomposition. In *MICAI'08*, pages 1–11, 2008.

[12] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, New York, 1999.

[13] Georg Gottlob and Christoph Koch. Monadic queries over tree-structured data. In *LICS'02*, pages 189–202, 2002.

[14] Georg Gottlob, Reinhard Pichler, and Fang Wei. Bounded treewidth as a key to tractability of knowledge representation and reasoning. In *AAAI'06*, pages 250–256, 2006.

[15] Georg Gottlob, Reinhard Pichler, and Fang Wei. Monadic datalog over finite structures with bounded treewidth. In *PODS'07*, pages 165–174, 2007.

[16] Frank Harary. *Graph theory*. Addison-Wesley, 1969.

[17] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7(3):499–562, 2006.

[18] Nysret Musliu. *An Iterative Heuristic Algorithm for Tree Decomposition Studies in Computational Intelligence: Recent Advances in Evolutionary Computation for Combinatorial Optimization*, pages 133–150. Springer, 2008.

[19] Neil Robertson and Paul D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.

[20] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1988.

Appendix A

Datalog programs

A.1 DLV

```
solve(V,P,N,C1):- leaf(V),bag(V,X,C),
                  true(V,P,N,C1,C),union(X,P,N),
                  intersect(emptyset,P,N).

solve(V,P,N,C1):- bag(V,X,C),child1(V1,V),bag(V1,XX,C),
                  diffel(XX,X,ELX),solve(V1,X3,N,C1),
                  diffel(X3,P,ELX),pset(_,P).

solve(V,P,N,C1):- bag(V,X,C),child1(V1,V),bag(V1,X1,C),
                  diffel(X1,X,ELX),solve(V1,P,X2,C1),
                  union(X2,N,ELX),pset(_,N).

solve(V,P,N,C1):- bag(V,X,C),child1(V1,V),bag(V1,X,CX1),
                  diffel(CX1,C,ELC),solve(V1,P,N,CX2),
                  union(CX2,C1,ELC).

solve(V,PX,N,C1C2):- bag(V,XX,C),diffel(XX,X,ELX),
                  child1(V1,V),bag(V1,X,C),solve(V1,P,N,C1),
                  true(V,X,emptyset,C2,C),diffel(PX,P,ELX),
                  union(C1C2,C1,C2),
                  pset(_,PX).

solve(V,P,NX,C1C2):- bag(V,XX,C),diffel(XX,X,ELX),
                  child1(V1,V),bag(V1,X,C),solve(V1,P,N,C1),
                  true(V,emptyset,ELX,C2,C),diffel(NX,N,ELX),
                  union(C1C2,C1,C2),pset(_,NX).
solve(V,P,N,C1C2):- bag(V,X,CX),child1(V1,V),bag(V1,X,C),
                  solve(V1,P,N,C1),true(V,P,N,C2,ELC),
                  diffel(CX,C,ELC),union(C1C2,C1,C2),cset(_,CX).

solve(V,P,N,C1C2):- bag(V,X,C),child1(V1,V),bag(V1,X,C),
                    solve(V1,P,N,C1),child2(V2,V),bag(V2,X,C),
                    solve(V2,P,N,C2),union(C1C2,C1,C2).

success:-root(V),bag(V,X,C),solve(V,P,N,C).

cl(c1).
cl(c2).
```

```
cl(c0).
var(x4).
var(x3).
var(x2).
var(x1).
var(x6).
var(x5).
pos(x2,c0).
pos(x6,c2).
pos(x3,c2).
pos(x3,c0).
pos(x1,c0).
pos(x4,c2).
pos(x5,c1).
neg(x2,c1).
neg(x4,c1).
root(n1).
leaf(n12).
leaf(n8).
leaf(n9).
child1(n12,n11).
child1(n3,n2).
child1(n6,n5).
child1(n4,n3).
child1(n11,n10).
child1(n2,n1).
child1(n8,n7).
child1(n7,n6).
child1(n5,n4).
child2(n9,n7).
child2(n10,n6).
bag(n11,emptyset,svc2).
bag(n2,emptyset,svc1).
bag(n10,svx2,svc2).
bag(n9,svx3,svc0c2).
bag(n1,svx5,svc1).
bag(n8,svx1,svc0).
bag(n7,svx2,svc0c2).
bag(n6,svx2x4,svc2).
bag(n5,svx2x4,emptyset).
bag(n4,svx2x4,svc1).
bag(n12,svx6,svc2).
bag(n3,svx4,svc1).
treewidth(3).
```

A.2 DLV Complex

```
#include<ListAndSet>

solve(V,P,N,C1):- leaf(V),bag(V,X,C),
                  true(V,P,N,C1,C),X=#union(P,N),
                  aset(_,X), aset(_,P), aset(_,N),
                  #intersection(N,P,{}).

solve(V,P,N,C1):- bag(V,X,C),child1(V1,V),bag(V1,XX,C),
                  #difference(XX,X,ELX),#card(ELX)==1,
                  solve(V1,X3,N,C1),
                  #difference(X3,P,ELX),#card(ELX)==1,
                  pset(_,P).

solve(V,P,N,C1):- bag(V,X,C),child1(V1,V),bag(V1,X1,C),
                  #difference(X1,X,ELX),#card(ELX)==1,
                  solve(V1,P,X2,C1),X2=#union(N,ELX),
                  aset(_,X2), aset(_,N), aset(_,ELX),
                  pset(_,N).

solve(V,P,N,C1):- bag(V,X,C),child1(V1,V),bag(V1,X,CX1),
                  #difference(CX1,C,ELC),#card(ELC)==1,
                  solve(V1,P,N,CX2),
                  CX2=#union(C1,ELC),
                  aset(_,CX2), aset(_,C1), aset(_,ELC).

solve(V,PX,N,C1C2):- bag(V,XX,C),
                     #difference(XX,X,ELX),#card(ELX)==1,
                     child1(V1,V),bag(V1,X,C),
                     solve(V1,P,N,C1),true(V,X,{},C2,C),
                     #difference(PX,P,ELX),#card(ELX)==1,
                     C1C2=#union(C1,C2),
                     aset(_,C1C2), aset(_,C1), aset(_,C2),
                     pset(_,PX).

solve(V,P,NX,C1C2):- bag(V,XX,C),
                     #difference(XX,X,ELX),#card(ELX)==1,
                     child1(V1,V),bag(V1,X,C),solve(V1,P,N,C1),
                     true(V,{},ELX,C2,C),
                     #difference(NX,N,ELX),#card(ELX)==1,
                     C1C2=#union(C1,C2),
                     aset(_,C1C2), aset(_,C1), aset(_,C2),
                     pset(_,NX).

solve(V,P,N,C1C2):- bag(V,X,CX),child1(V1,V),bag(V1,X,C),
                    solve(V1,P,N,C1),true(V,P,N,C2,ELC),
                    #difference(CX,C,ELC),#card(ELC)==1,
                    C1C2=#union(C1,C2),
                    aset(_,C1C2), aset(_,C1), aset(_,C2),
                    cset(_,CX).

solve(V,P,N,C1C2):- bag(V,X,C),child1(V1,V),bag(V1,X,C),
                    solve(V1,P,N,C1),child2(V2,V),bag(V2,X,C),
                    solve(V2,P,N,C2),C1C2=#union(C1,C2),
                    aset(_,C1C2), aset(_,C1), aset(_,C2).

success:-root(V),bag(V,X,C),solve(V,P,N,C).
```

```
cl(c1).
cl(c2).
cl(c0).
var(x4).
var(x3).
var(x2).
var(x1).
var(x6).
var(x5).
pos(x2,c0).
pos(x6,c2).
pos(x3,c2).
pos(x3,c0).
pos(x1,c0).
pos(x4,c2).
pos(x5,c1).
neg(x2,c1).
neg(x4,c1).
root(n1).
leaf(n12).
leaf(n8).
leaf(n9).
child1(n12,n11).
child1(n3,n2).
child1(n6,n5).
child1(n4,n3).
child1(n11,n10).
child1(n2,n1).
child1(n8,n7).
child1(n7,n6).
child1(n5,n4).
child2(n9,n7).
child2(n10,n6).
bag(n11,{},{c2}).
bag(n2,{},{c1}).
bag(n10,{x2},{c2}).
bag(n9,{x3},{c0,c2}).
bag(n1,{x5},{c1}).
bag(n8,{x1},{c0}).
bag(n7,{x2},{c0,c2}).
bag(n6,{x2,x4},{c2}).
bag(n5,{x2,x4},{}).
bag(n4,{x2,x4},{c1}).
bag(n12,{x6},{c2}).
bag(n3,{x4},{c1}).
pset(n4,{x2}).
pset(n5,{x2}).
pset(n6,{x4}).
pset(n4,{x2,x4}).
pset(n5,{x2,x4}).
pset(n1,{x5}).
pset(n6,{x2}).
pset(n10,{x2}).
pset(n8,{x1}).
pset(n12,{x6}).
pset(n4,{x4}).
pset(n5,{x4}).
pset(n6,{x2,x4}).
```

```
pset(n9,{x3}).
pset(n7,{x2}).
pset(n3,{x4}).
cset(n1,{c1}).
cset(n4,{c1}).
cset(n9,{c2}).
cset(n2,{c1}).
cset(n7,{c2}).
cset(n10,{c2}).
cset(n8,{c0}).
cset(n12,{c2}).
cset(n9,{c0}).
cset(n6,{c2}).
cset(n7,{c0}).
cset(n3,{c1}).
cset(n11,{c2}).
cset(n9,{c0,c2}).
cset(n7,{c0,c2}).
treewidth(3).
```

A.3 DLV Complex with external predicate

```
#include<ListAndSet>
#include<exttrue>

solve(V,P,N,C1):- leaf(V),bag(V,X,C),
                  #true(P,N,C1,C,POSSET,NEGSET),
                  posset(POSSET),negset(NEGSET),
                  X=#union(P,N), aset(_,X), aset(_,P), aset(_,N),
                  #intersection(N,P,{}).

solve(V,P,N,C1):- bag(V,X,C),child1(V1,V),bag(V1,XX,C),
                  #difference(XX,X,ELX),#card(ELX)==1,
                  solve(V1,X3,N,C1),
                  #difference(X3,P,ELX),#card(ELX)==1,
                  pset(_,P).

solve(V,P,N,C1):- bag(V,X,C),child1(V1,V),bag(V1,X1,C),
                  #difference(X1,X,ELX),#card(ELX)==1,
                  solve(V1,P,X2,C1),X2=#union(N,ELX),
                  aset(_,X2), aset(_,N), aset(_,ELX),pset(_,N).

solve(V,P,N,C1):- bag(V,X,C),child1(V1,V),bag(V1,X,CX1),
                  #difference(CX1,C,ELC),#card(ELC)==1,
                  solve(V1,P,N,CX2),CX2=#union(C1,ELC),
                  aset(_,CX2), aset(_,C1), aset(_,ELC).

solve(V,PX,N,C1C2):- bag(V,XX,C),
                  #difference(XX,X,ELX),#card(ELX)==1,
                  child1(V1,V),bag(V1,X,C),solve(V1,P,N,C1),
                  #true(X,{},C2,C,POSSET,NEGSET),
                  posset(POSSET),negset(NEGSET),
                  #difference(PX,P,ELX),#card(ELX)==1,
                  C1C2=#union(C1,C2),
                  aset(_,C1C2), aset(_,C1), aset(_,C2),
                  pset(_,PX).

solve(V,P,NX,C1C2):- bag(V,XX,C),
                  #difference(XX,X,ELX),#card(ELX)==1,
                  child1(V1,V),bag(V1,X,C),solve(V1,P,N,C1),
                  #true({},ELX,C2,C,POSSET,NEGSET),
                  posset(POSSET),negset(NEGSET),
                  #difference(NX,N,ELX),#card(ELX)==1,
                  C1C2=#union(C1,C2),
                  aset(_,C1C2), aset(_,C1), aset(_,C2),
                  pset(_,NX).

solve(V,P,N,C1C2):- bag(V,X,CX),child1(V1,V),bag(V1,X,C),
                  solve(V1,P,N,C1),
                  #true(P,N,C2,ELC,POSSET,NEGSET),
                  posset(POSSET),negset(NEGSET),
                  #difference(CX,C,ELC),#card(ELC)==1,
                  C1C2=#union(C1,C2),
                  aset(_,C1C2), aset(_,C1), aset(_,C2),
                  cset(_,CX).

solve(V,P,N,C1C2):- bag(V,X,C),child1(V1,V),bag(V1,X,C),
                  solve(V1,P,N,C1),child2(V2,V),bag(V2,X,C),
```

```
                    solve(V2,P,N,C2),C1C2=#union(C1,C2),
                    aset(_,C1C2), aset(_,C1), aset(_,C2).

success:-root(V),bag(V,X,C),solve(V,P,N,C).

treewidth(3).
posset({{c0,x2},{c2,x6},{c2,x3},{c0,x3},{c0,x1},{c2,x4},{c1,x5}}).
negset({{c1,x2},{c1,x4}}).
```

Appendix B

Parsers

B.1 Flex - scanner.l

```
extern char *yytext;
long lineno = 1;
long curchar = 0;

void yyerror(char *st);

int YYLeng()
{
  return strlen(yytext);
}
char * Copyyytext()
{
      int len  = YYLeng()+1;
      char * buffer = new char[len];
      memset(buffer,0,len);
      strcpy(buffer, yytext);
      return buffer;
}

%}

COMMENT     %.*
SPACE       [\t\r\b ]*
NEWLINE     \n
DIGIT       [0-9]
NUMBER      {DIGIT}+
IDENT       [a-z_][a-zA-Z0-9_']*
SETVARIABLE [A-Z][a-zA-Z0-9'_']*
STRING      \"[^\"\n]*\"
IF          ":""-"
ASSIGN      "="
RULEEND     "."
OBRACKET    "("
CBRACKET    ")"
OACCOLADE    "{"
CACCOLADE    "}"
COMMA       ","
```

```
WEAK_NOT  not
UNION     union
INTERECT  intersect
PROPSUBSET  propsubset
SUBSET      subset
DIFFEL      diffel
CL        cl
VAR       var
POS       pos
NEG       neg
ROOT      root
LEAF      leaf
TRUE      true
CHILD1      child1
CHILD2      child2
BAG       bag
NULLVAL     NULL
EMPTYSET  "{""}"

%%

{SPACE}|{COMMENT} { curchar+=YYLeng();  }
{NEWLINE}          {   curchar=0; lineno++; }
{CL}        {   curchar+=YYLeng();return CL; }
{VAR}       {   curchar+=YYLeng();return VAR; }
{POS}       {   curchar+=YYLeng();return POS; }
{NEG}       {   curchar+=YYLeng();return NEG; }
{ROOT}        {   curchar+=YYLeng();return ROOT; }
{LEAF}        {   curchar+=YYLeng();return LEAF; }
{TRUE}        {   curchar+=YYLeng();return TRUE; }
{CHILD1}      {   curchar+=YYLeng();return CHILD1; }
{CHILD2}      {   curchar+=YYLeng();return CHILD2; }
{BAG}       {   curchar+=YYLeng();return BAG; }
{NULLVAL}     {   curchar+=YYLeng();return NULLVAL; }
{UNION}       {   curchar+=YYLeng();return UNION; }
{INTERECT}      {   curchar+=YYLeng();return INTERSECT; }
{SUBSET}      {   curchar+=YYLeng();return SUBSET; }
{PROPSUBSET}  {   curchar+=YYLeng();return PROPSUBSET; }
{DIFFEL}      { curchar+=YYLeng();return DIFFEL; }
{EMPTYSET}        {   curchar+=YYLeng();
            yylval.s = Copyyytext(); return EMPTYSET;
          }
{WEAK_NOT}      {   curchar+=YYLeng();return WEAK_NOT; }
{NUMBER}      {   curchar+=YYLeng();
            yylval.l = strtol(yytext,NULL,10); return  NUMBER;
          }
{IDENT}       {   curchar+=YYLeng(); yylval.s = Copyyytext();
            return IDENT;
          }
{STRING}      {   curchar+=YYLeng(); yylval.s = Copyyytext();
            return STRING;
          }
{SETVARIABLE} {   curchar+=YYLeng(); yylval.s = Copyyytext();
            return SETVARIABLE;
          }
{IF}        {   curchar+=YYLeng();return IF; }
{RULEEND}     {   curchar+=YYLeng();  return RULEEND; }
{OBRACKET}      {   curchar+=YYLeng();  return OBRACKET; }
```

```
{CBRACKET}      {    curchar+=YYLeng();   return CBRACKET; }
{OACCOLADE}     {    curchar+=YYLeng();   return OACCOLADE; }
{CACCOLADE}     {    curchar+=YYLeng();   return CACCOLADE; }
{COMMA}         {    curchar+=YYLeng(); return COMMA; }
{ASSIGN}      {    curchar+=YYLeng();return ASSIGN; }

.         {
            char msg[25];
            sprintf(msg,"%s <%s>","invalid character",yytext);
            yyerror(msg);
 }

<<EOF>>             {
yyterminate();
}

%%
```

B.2 Bison - parser.y

```
%start program

%token  <l> NUMBER
%token  <s> IDENT SETVARIABLE STRING NULLVAL EMPTYSET
%token  <s> IF WEAK_NOT OBRACKET CBRACKET OACCOLADE CACCOLADE COMMA
%token  <s> RULEEND CL VAR POS NEG ROOT LEAF CHILD1 CHILD2 BAG TRUE
%token  <s> ASSIGN NOT UNION INTERSECT PROPSUBSET SUBSET DIFFEL
%type <pn> constant expr term  expr_0 set
%type <pn> paramlist literal atom body_element constantlist
%type <pn> program body rule rules head buildin

%left ASSIGN NUMBER VARIABLE PIPE UNION INTERSECT
%left PROPSUBSET SUBSET DIFFEL
%left NOT
%right OBRACKET
%left  CBRACKET
%left COMMA
%right OACCOLADE
%left  CACCOLADE

%%

program : rules
    {  root_parse_node = $1; }
    |
    error
    { root_parse_node = NULL; }

 ;

rules : rule
    { $$ = popErrorStack(
        new ParseTreeNode(NT_RULES,NULL,$1,NULL,lineno));
    }
    |
    rule  rules
    { $$ = popErrorStack(
        new ParseTreeNode(NT_RULES,NULL,$1,$2,lineno));
    }
;

rule :   head RULEEND
    { $$ = popErrorStack(
        new ParseTreeNode(NT_RULE,NULL,$1,NULL,lineno));
    }
      | head IF body RULEEND
    { $$ = popErrorStack(
        new ParseTreeNode(NT_RULE,NULL,$1,$3,lineno));
    }
      | IF body RULEEND
    { $$ = popErrorStack(
        new ParseTreeNode(NT_RULE,NULL,NULL,$2,lineno));
    }
;

head : literal
```

```
      { $$ = popErrorStack(
      new ParseTreeNode(NT_HEAD,NULL,$1,NULL,lineno));
    }
      | head COMMA literal
    { $$ = popErrorStack(
      new ParseTreeNode(NT_HEAD,NULL,$3,$1,lineno));
    }
;

body : body_element
    {  $$ = popErrorStack(
       new ParseTreeNode(NT_BODY,NULL,$1,NULL,lineno));
    }
      | body COMMA body_element
    {  $$ = popErrorStack(
       new ParseTreeNode(NT_BODY,NULL,$3,$1,lineno));
    }
;

body_element :  literal
    {  $$=$1; }
      | expr
      {  $$ = popErrorStack(
       new ParseTreeNode(NT_EXPR,NULL,$1,NULL,lineno));
    }
;

literal : atom
    {  $$ = popErrorStack(
       new ParseTreeNode(NT_LITERAL,NULL,$1,NULL,lineno));
    }
    |
     WEAK_NOT atom
    {  $$ = popErrorStack(
       new ParseTreeNode(NT_LITERAL,NULL,$2,NULL,lineno));
      ((ParseTreeNode *)$$)->is_negative=true;
    }

;

atom :  IDENT
    {  $$ = popErrorStack(
       new ParseTreeNode(NT_ATOM,$1,NULL,NULL,lineno));
    }
    |
    IDENT OBRACKET paramlist CBRACKET
    {  $$ = popErrorStack(
       new ParseTreeNode(NT_ATOM,$1,NULL,$3,lineno));
    }
    |
    buildin
    {  $$ = $1; }

;

buildin:  CL OBRACKET paramlist CBRACKET
    {  $$ = popErrorStack(
       new ParseTreeNode(NT_BUILDIN,NULL,NULL,$3,lineno,BI_CL));
```

```
}
|
VAR OBRACKET paramlist CBRACKET
{   $$ = popErrorStack(
     new ParseTreeNode(NT_BUILDIN,NULL,NULL,$3,lineno,BI_VAR));
}
|
POS OBRACKET paramlist CBRACKET
{   $$ = popErrorStack(
     new ParseTreeNode(NT_BUILDIN,NULL,NULL,$3,lineno,BI_POS));
}
|
NEG OBRACKET paramlist CBRACKET
{   $$ = popErrorStack(
     new ParseTreeNode(NT_BUILDIN,NULL,NULL,$3,lineno,BI_NEG));
}
|
ROOT OBRACKET paramlist CBRACKET
{   $$ = popErrorStack(
     new ParseTreeNode(NT_BUILDIN,NULL,NULL,$3,lineno,BI_ROOT));
}
|
LEAF OBRACKET paramlist CBRACKET
{   $$ = popErrorStack(
     new ParseTreeNode(NT_BUILDIN,NULL,NULL,$3,lineno,BI_LEAF));
}
|
CHILD1 OBRACKET paramlist CBRACKET
{   $$ = popErrorStack(
     new ParseTreeNode(NT_BUILDIN,NULL,NULL,$3,lineno,
               BI_CHILD1));
}
|
CHILD2 OBRACKET paramlist CBRACKET
{   $$ = popErrorStack(
     new ParseTreeNode(NT_BUILDIN,NULL,NULL,$3,lineno,
                    BI_CHILD2));
}
|
BAG OBRACKET paramlist CBRACKET
{   $$ = popErrorStack(
     new ParseTreeNode(NT_BUILDIN,NULL,NULL,$3,lineno,BI_BAG));
}
|
TRUE OBRACKET paramlist CBRACKET
{   $$ = popErrorStack(
     new ParseTreeNode(NT_BUILDIN,NULL,NULL,$3,lineno,BI_TRUE));
}
|
UNION OBRACKET paramlist CBRACKET
{   $$ = popErrorStack(
     new ParseTreeNode(NT_BUILDIN,NULL,NULL,$3,lineno,BI_UNION));
}
|
INTERSECT OBRACKET  paramlist CBRACKET
{   $$ = popErrorStack(
     new ParseTreeNode(NT_BUILDIN,NULL,NULL,$3,lineno,
               BI_INTERSECT));
```

```
    }
    |
    DIFFEL OBRACKET  paramlist CBRACKET
    {  $$ = popErrorStack(
        new ParseTreeNode(NT_BUILDIN,NULL,NULL,$3,lineno,
                  BI_DIFFEL));
    }
;

paramlist : paramlist COMMA term
    {  $$ = popErrorStack(
        new ParseTreeNode(NT_PARAMLIST,NULL,$3,$1,lineno));
    }
    |
    term
    {  $$ = popErrorStack(
        new ParseTreeNode(NT_PARAMLIST,NULL,$1,NULL,lineno));
    }
    |
    paramlist COMMA expr
    {  $$ = popErrorStack(
        new ParseTreeNode(NT_PARAMLIST,NULL,$3,$1,lineno));
    }
    |
    expr
    {  $$ = popErrorStack(
        new ParseTreeNode(NT_PARAMLIST,NULL,$1,NULL,lineno));
    };


expr : expr_0
    {  $$ = $1; }
    |
    term ASSIGN expr_0
    {  $$ = popErrorStack(
      new ParseTreeNode(NT_EXPRNULL,$1,$3,lineno,EX_ASSIGN));
    }
;

expr_0 : term
    {  $$ = $1; }
    |
    SUBSET  OBRACKET  expr_0 COMMA expr_0  CBRACKET
    {  $$ = popErrorStack(
        new ParseTreeNode(NT_EXPR,NULL,$3,$5,lineno,EX_SUBSET));
    }
    |
    PROPSUBSET OBRACKET  expr_0 COMMA expr_0  CBRACKET
    {  $$ = popErrorStack(
        new ParseTreeNode(NT_EXPR,NULL,$3,$5,lineno,
                  EX_PROPSUBSET));
    }
;

term  : constant
    {  $$=$1; }
    |
    SETVARIABLE
```

```
    {  $$ = popErrorStack(
        new ParseTreeNode(NT_SETVARIABLE,$1,NULL,NULL,lineno));
    }
    |
    EMPTYSET
    {  $$ = popErrorStack(
        new ParseTreeNode(NT_EMPTYSET,$1,NULL,NULL,lineno));
    }
    |
    set
    {  $$=$1; }
;

set : OACCOLADE constantlist CACCOLADE
    {  $$ = popErrorStack(
        new ParseTreeNode(NT_SET,NULL,NULL,$2,lineno));
    }


constantlist : constantlist COMMA constant
    {  $$ = popErrorStack(
        new ParseTreeNode(NT_CONSTLIST,NULL,$3,$1,lineno));
    }
    |
    constant
    {  $$ = popErrorStack(
        new ParseTreeNode(NT_CONSTLIST,NULL,$1,NULL,lineno));
    }
;

constant: IDENT
    {  $$ = popErrorStack(
        new ParseTreeNode(NT_IDENT,$1,NULL,NULL,lineno));
    }
    |
    NUMBER
    {  $$ = popErrorStack(
        new ParseTreeNode($1,NULL,NULL,lineno));
    }
    |
    STRING
    {  $$ = popErrorStack(
        new ParseTreeNode(NT_STRING,$1,NULL,NULL,lineno));
    }
    |
    NULLVAL
    {  $$ = popErrorStack(
        new ParseTreeNode(NT_NULL,$1,NULL,NULL,lineno));
    }
    |
     OACCOLADE CACCOLADE
    {  $$ = popErrorStack(
        new ParseTreeNode(NT_EMPTYSET,$1,NULL,NULL,lineno));
    }

%%
```

Appendix C

Gnuplot

```
#set global options

set terminal png

set xlabel 'clauses'
set key left top
set data style linespoints
set pointsize 1.5


# create plots Treewidth: 3
set title 'Treewidth: 3'
set output "stats_complex/results_3.png"
plot 'stats_complex/results_3.dat' using 2:7 \
title "Time (sec)" with linespoints , \
'stats_complex/results_3.dat' using 2:($4/10) \
title "Nodes (x/10)" with linespoints , \
'stats_complex/results_3.dat' using 2:($6/1000) \
title "True (x/1000)" with linespoints

# create plots Treewidth: 4
set title 'Treewidth: 4'
set output "stats_complex/results_4.png"
plot 'stats_complex/results_4.dat' using 2:7 \
title "Time (sec)" with linespoints , \
'stats_complex/results_4.dat' using 2:($4/10) \
title "Nodes (x/10)" with linespoints , \
'stats_complex/results_4.dat' using 2:($6/1000) \
title "True (x/1000)" with linespoints


# create plots  Treewidth: 5
set title 'Treewidth: 5'
set output "stats_complex/results_5.png"
plot 'stats_complex/results_5.dat' using 2:7 \
title "Time (sec)" with linespoints , \
'stats_complex/results_5.dat' using 2:($4/10) \
title "Nodes (x/10)" with linespoints , \
'stats_complex/results_5.dat' using 2:($6/1000) \
title "True (x/1000)" with linespoints
```

```
# setting terminal to postscript
set terminal postscript color solid
set terminal postscript eps

# create plots Treewidth: 3
set title 'Treewidth: 3'
set output "stats_complex/results_3.eps"
plot 'stats_complex/results_3.dat' using 2:7 \
title "Time (sec)" with linespoints , \
'stats_complex/results_3.dat' using 2:($4/10) \
title "Nodes (x/10)" with linespoints , \
'stats_complex/results_3.dat' using 2:($6/1000) \
title "True (x/1000)" with linespoints

# create plots Treewidth: 4
set title 'Treewidth: 4'
set output "stats_complex/results_4.eps"
plot 'stats_complex/results_4.dat' using 2:7 \
title "Time (sec)" with linespoints , \
'stats_complex/results_4.dat' using 2:($4/10) \
title "Nodes (x/10)" with linespoints , \
'stats_complex/results_4.dat' using 2:($6/1000) \
title "True (x/1000)" with linespoints


# create plots  Treewidth: 5
set title 'Treewidth: 5'
set output "stats_complex/results_5.eps"
plot 'stats_complex/results_5.dat' using 2:7 \
title "Time (sec)" with linespoints , \
'stats_complex/results_5.dat' using 2:($4/10) \
title "Nodes (x/10)" with linespoints , \
'stats_complex/results_5.dat' using 2:($6/1000) \
title "True (x/1000)" with linespoints
```