Computational Complexity of a Knowledge Compilation Approach to the Static Semantics for Disjunctive Logic Programs

Stefania Costantini
Dip. di Matematica Pura e Appl. , Univ. di L'Aquila
via Vetoio loc. Coppito, I-67100 L'Aquila
phone +39(0862)433130
fax +39(0862)433180
stefcost@univaq. it
Tiziana Morbidoni
Hewlett Packard, Corporate Offices
675 E. Middlefield Rd, 94043 Mountain View, California
phone +1(650)3144127
fax +1(650)3144092
tiziana_morbidoni@hp. com

Keywords: disjunctive logic programs, knowledge compilation, computational complexity

Abstract

Aim of the paper is to assess the computational complexity of Static Semantics for disjunctive logic programs, according to the definition proposed by Costantini and Lanzarone, based on the notion of Knowledge Compilation. The computation is here divided in two phases: a pre-processing Program Transformation (PT) and a constructive phase IFC, which is a suitable variation of the well-founded model computation. The results show a complexity for the PT which is 2-EXPTIME for the resource time and 2-EXPSPACE for the resource space. However, these results are really pessimistic since we always consider the WORST CASE, that sometimes is not even a possible scenario. The actual cost of the PT highly depends on the program at hand: the experiments show that in many practical cases the PT has a linear cost. In fact, in knowledge representation disjunctions express incomplete or uncertain knowledge. Therefore, their number and length should be kept as limited as possible, for the representation to be practically usable (otherwise, not only the complexity increases, but most queries would have disjunctions as answer). The IFC has a quadratic cost for the resource time, and a linear cost in the length of the transformed program for the resource space. Also in this case the analysis is for the worst case scenario. Since the IFC is performed for each query answering request, the resource time is crucial. For this reason we propose to introduce a step of query pre-processing with the aim of reducing the part of the program considered in the IFC. This is feasible because the Static Semantics verifies the Relevance Principle.

1. Introduction

As pointed out in [EG92], [EG92b], [Sch92], [CS93], in general the inference in Non-Monotonic Reasoning (NMR) and hence in disjunctive logic programming is more complex than the one for monotonic logic. Therefore we expect very high complexities whenever we deal with NMR. The key point is that problems of AI are computationally very difficult in the worst case. An important goal of todays' research is to be able to use the power of NMR, without paying all the cost of the computation. Formally, the question becomes:

Given a knowledge base K, and query Q, is it possible to find a new one K' such that
K(unknown char)—$_{NMR}$ Q if and only if K' |= Q
and, if so, what is the space allocation needed by K' ?

Knowledge compilation [CD98] is an AI technique for addressing computational intractability of reasoning problems. It is aimed at fastening the on-line procedure of query answering by introducing a pre-processing phase, that can be seen (as the name suggests) as a sort of compiling. In general, the output of the compiling phase is computationally heavy and requires a lot of space. If we use this approach of non monotonic logic, in many cases it could be possible to reduce the complexity of Query Answering from a complexity or -complete to a complexity NP or co-NP.

The very high complexity of disjunctive programs [EG92] is due to two interacting factors: the disjunction in the head of the clauses and the conditions of minimality that the answer sets must respect. Among the different semantics which have been proposed for this class of programs, in this paper we consider the Static Semantics, which is a natural generalization of the well-founded model semantics for general logic programs. The declarative original version of Przymusinski, based on an Autoepistemic Logic of Knowledge and Belief, has complexity in [CM99]. This means that this is the most complex semantics for disjunctive logic programs, as all the others proposed semantics are at most -complete (this is the case for checking the existence of a perfect or of a partial stable model for disjunctive programs). Correspondingly, it is also much more powerful than any other semantics defined for this class of programs.

Related to the notion of Knowledge Compilation, another definition of Static Semantics has been proposed by Costantini and Lanzarone [CL95]. The computation is here divided into two phases: a pre-processing Program Transformation (PT) and a constructive phase (IFC). This new definition of the Static Semantics can be seen as a first try of compiling for logic, disjunctive programs. The Iterated Fixpoint Computation uses the result of the Program Transformation for solving the Query Answering problem.

In this paper, we assess the computational complexity of the Static Semantics defined as PT + IFC. The general result is that the PT has complexity 2-EXPTIME for the resource time and 2-EXPSPACE for the resource space, and the IFC has a quadratic cost for the resource time, and a linear cost in the length of the transformed program for the resource space. However, we try to identify the classes of programs for which the complexity (according to our algorithms) is tractable. In particular, this is the case when: the program is linearly expanded by the PT (which in our experiments seem to happen on many real programs), and/or the subset of the program used for query answering can be reduced. About this point, we specifically refer to the principle of

Relevance [Dix94], which is verified by the Static Semantics. Consequently, the IFC can be performed not on the whole program P' which results from PT, but on the subset of rules which are relevant to the given query. This subset can be determined in polynomial, quadratic time. Then, the application of this principle results in a significant decrease of the cost of IFC, which correspond to a significant increase of the efficiency of on-line query-answering. Yet, a formal comparison with single-step optimized algorithms is still to be made.
The paper is organized as follows. In Section 2 we define the Static Semantics according to [Prz94], and provide the reader with some necessary definitions. In Section 3 we define the Static Semantics according to [CL94], as Program Transformation (PT) and Iterated Fixpoint Computation (IFC). In Section 4 we assess the complexity of PT, in Section 5 the complexity of IFC. In Section 6 we summarize the results obtained, and suggest possible improvements. In Section 7 are the concluding remarks. All the proofs of the given results, as well as some of the algorithms that for the sake of space we have omitted, can be found in the extended version of this paper [CM99].

1 2. Preliminary Definitions

In this paper we discuss logical, disjunctive, ground propositional programs, and we will always refer to Herbrand interpretations. We restrict our attention to the Well-Founded Semantics for general logic programs [VGRS91] [Prz89], and of Static Semantics for general disjunctive logic programs [Prz94]. Following the approach of [CL95], a constructive definition of Static Semantics can be obtained as an extension of the constructive definition of the Well-Founded Semantics proposed in [Prz89]. As it is customary in the literature, we assume that the program is already instantiated, and therefore all the clauses are propositional, defined on a language *L*. The negation indicated in the clauses is non monotonic, hence it has to be read as

not C is true if and only if C is believed to be false

As propositional logics only support the classic negation, it becomes necessary to widen the language L with a modal operator called Belief. The atoms that do not contain the modal operator B are called objective atoms; the atoms of the form BF, where F is a generic formula in L, are called Belief atoms. From this language, it is possible to build an Autoepistemic Theory or Belief, defined as follows.
An **Affirmative Belief Theory** T in the language ***L*** consists of a (possibly infinite) set of clauses of the form:
B ∧... ∧ B ∧ ***B***G ∧... ∧ ***B***G ∧ ¬ ***B***F ∧... ∧ ¬***B***F ⊃ A ∨... ∨ A with $l > 0, k, m, n \geq 0$, Ai's and Bi's objective atoms and Fi's , Gi's generic formulae
The following axioms describe Belief Atoms' properties :

- Consistency Axiom (D) ¬ ***B***⊥
- Normality Axiom (K) ***B*** (F ⊃G) ⊃ (***B***F ⊃ ***B***G) F and G generic formula
- Necessitation Rule (N) for any formula F
- We also assume the distributivity of belief conjunction ***B***(F ∧ G) ≡ ***B***F ∧ ***B*** G

Every disjunctive logic program can be transformed into a semantically equivalent Affirmative Belief Theory T.

Translation Logic Program - Belief Theory : given logic program P, every clause of the form

A ∨... ∨ A ← B ∧... ∧ B ∧ not C ∧... ∧ not C l $\geq$ 1, m,n $\geq$0

where A , B , C are objective atoms, is replaced with the following formula in ***L*** *:*

B ∧... ∧ B ∧ ***B***¬ ***C*** ∧... ∧ ***B***¬ C ⊃ A ∨... ∨ A

Derivable Formulas. A formula F is derivable from a Belief Theory if it belongs to the smallest set C_{n*}(T) of formulas in $\boldsymbol{L}_B$ which contains the theory T and all the instances of axioms (K) and (D) and is closed with respect to (N). We refer to this set as T $\wp\otimes_*$ F. Then, C_{n*} (T) = {F : T $\wp\otimes_*$ F}

A **Minimal Model** of a Belief Theory T is a model M of T such that it does not exist any smaller model N of T which coincides with M on belief atoms.

The **Belief Closure Operator** Ψ_T of T defined in [Prz94], is based on the idea of building extensions of a belief theory T based on another belief theory S, obtained by augmenting T with precisely those belief atoms ***B***F which satisfy the condition that F is true in all minimal models of S. The belief closure operator is monotonic, and its fixpoint are called **Static Expansions** of T. The least fixed point of Ψ_T is the **Static Completion** of T. For an Affirmative Belief Theory, the Static Completion is always consistent. The **Static Semantics** of a belief theory T is the set of all formulae which belongs to the Static Completion of T.

In the rest of this paper we refer to the following version of the constructive procedure for obtaining the well-founded model WFM_P of a general logic program P.

We assume that B_P, for every atom A in P, will contain both A and not A.

Enhanced Immediate Consequence Operator J $\subseteq B_P$ is the set of literals believed true such that

$T_J \uparrow 0$ = {not A $\in B_P$: A $\notin$ J}

$T_J \uparrow$ n+1 = {A $\in B_P$: A ← B_1,... , B_k is a clause in P, {B_1,... ,B_k } $\subseteq T_J \uparrow$n }

S(J) = $T_J \uparrow\omega$

Iterated Fixpoint Computation. Let Σ = S^2 = S(S), and

Σ ↑0 = ϕ Σ ↑n+1 = Σ(Σ ↑n)) = S(S(Σ ↑n))

Σ computes as least fixed point a set WFC (Well-Founded Completion) of literals, such that if T = {A : A ∈ WFC} and F = {A : not A ∈ WFC}, we have WFM_P = < T;F >.

The following notation will be used in the following sections:

- conjunctions and disjunctions of literals are treated as sets
- ***D*** D is a disjunction such as $D_1 \vee D_2 \vee ... \vee D_n$ (positive disjunction)
- ﬃ ¬ ***D*** D is a disjunction such as $\neg D_1 \vee \neg D_2 \vee ... \vee \neg D_n$ (negative disjunction)
- ˆ ***D*** D is any disjunction (positive and/or negative disjuncts)
- ***C*** B is a conjunction such as $B_1 \wedge B_2 \wedge ... \wedge B_n$ (positive conjunction)
- ¬ ***C*** K is a conjunction such as $\neg K_1 \wedge \neg K_2 \wedge ... \wedge \neg K_n$ (negative conjunction)
- ˆ***C*** B is a generic conjunction (positive and/or negative conjuncts)
- ∼ is negation , for example ¬***D*** D = ∼ (***D*** D) = not A ∨ not B (according to De Morgan rules)

3 - The Static Semantics as Program Transformation and Well-Founded Computation

We are now ready to introduce an alternative definition of the Static Semantics. This

new definition is divided into two parts: first a semplification of the program, and then a variation of the well-founded model construction.
We will define in fact: first, a transformation procedure for the program, which formalizes the impact of disjunctions in the derivation process and the correlation between implicit and explicit disjunctions; then, a modification in the construction of the well-founded model which links disjunction and negation. A procedural interpretation of the static semantics might accordingly consist in: an inefficient pre-processing phase, to be executed only once, and a second, efficient phase, to be executed on-line for each query, based on the procedural semantics for well-founded model.

The objective of the Program Transformation is to add to the original program P, a set of clauses that make the disjunctions in P and all their logical consequences explicit. In the resulting program P' the disjunctions like $(A \vee B \vee C)$ are treated as single, new atoms which differ from the atoms A, B and C. Also the atoms $\neg$ A are treated as new atoms, distinct from A. P' corresponds to an Affirmative Belief Theory simply by interpreting atoms $\neg$ A as $\boldsymbol{B}\neg\, \boldsymbol{A}$, and by interpreting the connective ":-" as classical implication.

Given Q, P, R, W objective atoms , D a disjunction of objective atoms and C a generic conjunction. The transformation from P to P', as explained in [CL95], consists of the following steps :

- initial step - links the explicit and implicit disjunctions, by applying the axiom:
  (1) $((C \wedge W \supset Q) \wedge (C \wedge R \supset S) \supset (C \wedge (W \vee R)) \supset (Q \vee S)$
- intermediate step - exploits the disjunctions that appear as heads in P, by applying the axiom: (2) $((C \wedge W \supset Q) \supset (C \wedge (W \vee S)) \supset (Q \vee S))$
- final step - deletes redundant clauses, by applying the axiom:
  (3) $((C \supset D) \supset (C \wedge W \supset (D \vee S))$

P' is equivalent to P in order to compute the Static Semantics, provided that the Herbrand base is redefined in the following way :

Definition 1. 1. Expanded Herbrand base B .
B = B$_P$ $\cup$ $\{$A : A is in P $\}$ $\cup$ $\{$not A : not A is in P $\}$ $\cup$
$\{$ ***D*** D : ***D*** D is in P $\}$ $\cup$ $\{\sim$ ***D*** D : ***D*** D is in P$\}$

By using **B ,** it is possible to treat P' as a Horn program.

Lemma 3. 1. If P is a generic disjunctive logic program and (unknown char)P(unknown char) = n, then (unknown char)**B** (unknown char) $\leq$ 2n.

The static completion of P coincides with the WFC (Well-Founded Completion) of P', that can be obtained by means of the Iterated Fixpoint Computation of Section 2, provided that the initial step be modified as follows.

Let A,not A,D D,$\neg$ D D $\in$ B , J $\subseteq$ **B .**

$T_J \uparrow 0$ = **(1)** $\{$not A : A $\notin$ J and $\forall$ ***D*** D is true (A $\in$ ***D*** D implies ***D*** D $\notin$ J) and $\forall$ $\neg$***D*** D is true (not A $\in$ $\neg$***D*** D implies $\neg$***D*** D $\notin$ J)$\}$ $\cup$
(2) $\{$not A : A $\notin$ J e $\forall$ ***D*** D is true (A $\in$ ***D*** D implies ***D*** D' = ***D*** D - $\{$A$\}$ $\in$ J)$\}$ $\cup$
(3) $\{\neg$***D*** D = $\sim$***D*** D : $\exists$ A $\in$ ***D*** D, A $\notin$ J$\}$

Lemma 3. 2. Given J $\subseteq$ **B , J** consistent set. Given F negative literal F or negative disjunction,
$T_J \uparrow 0$ equals the set $\{$***B***F : J $\models_{min}$ F$\}$.

4. Computational Complexity of Program Transformation

Aim of this paper is to study the computational complexity in the worst case of the approach of computing the static completion via program transformation and well-founded computation.

For the Program Transformation, the resource Time it is not crucial because this operation will be executed only once. On the contrary, the resource Space is very important because the modified program P' will b

Given a program P, let n be the length of P, M the number of clauses of P, L the average clause length and L_{max} the max clause length. The following relation n = M * L is true. Moreover, if L if the length of a clause, there are no more than L/2 atoms in this clause , since between any couple of atoms there is always a logic connector. Let us introduce the following :

Definition 4. 1. Base. Let P be a disjunctive logic program.

Base(P) = $\{$A | A is a propositional symbol in P$\}$ $\cup$ $\{$A | not A is in P $\}$ $\cup$
$\{$A | A is a propositional symbol contained in a disjunction in P $\}$

Base(P) is therefore the alphabet of the propositional symbols which the program P and the transformed program P' are based on. In this context, a *literal* is a generic propositional symbol in P or a disjunction of propositional symbols.

Lemma 4. 1. Let P be a disjunctive logic program P and N_a = (unknown char)Base(P)(unknown char). The maximum number of distinct literals that we can build from P is : .

It is easy to see that, if the program has length n, it will contain up to n/2 distinct propositional symbols. Together with the above Lemma , this implies the following:

Theorem 4. 1. Max Extension of P. A program P of length n can be extended until the number of distinct literals is

Note 1. *The Theorem of Max Extension provides a theoretical limit that can hardly be reached for the following reasons.*

1. The presence of disjunctions in a logic program P is due to some kind of incomplete knowledge. Therefore, the use of disjunctions should be limited to few clauses.

2. If N_a is about n/2, this means that the intersections between head and body of the clauses are minimal or empty, and so the resolution process will not apply; this is the typical case of data bases, and programs where relations are defined mostly extensionally.

Lemma 4. 2. Given a clause in P such as $\{$***D*** T $\leftarrow$ ˆ***D*** D,***C*** H, ¬ ***C*** K$\}$, the literals in ***D*** T,ˆ***D*** D, ***C*** H and ¬***C*** K are all different.

Lemma 4. 3. Given a program P where N = (unknown char)Base(P)(unknown char), T = (unknown char)Atoms(P)(unknown char), the maximum length L_{max} of a clause is 4N * 3^N.

Example 4. 1 *Let's consider the following program P*

a b,g　　　a c,h　　　b $\vee$ c e,f

and let's add the clause a $\vee$ d b,g,e

The last clause is evidently a logical consequence of the first one. If we don't remove this clause right away, it will be considered in the initial step and the clause

a $\vee$ d (b $\vee$ c),g,h,e (*)　　*will be added.*

Please note that (*) *is a logical consequence of*

a (b $\vee$ c),g,h

that is derived in the initial step from the first three clauses.
As we introduce in P new clauses, it becomes more likely to encounter redundant clauses. In particular, the greater the number of atoms in a clause and of literals in the head, the higher is the probability that this clause is redundant.
Then, it is necessary to define an algorithm for redundancy check, *Redundancy(C1,P)*. It is easy to see that the algorithm is deterministic and the time required is polynomial in the length of program P.
We are now ready to design the algorithm *Transform_1* that applies the initial step of the program transformation. Let P be the initial program and P' the final program. Before proceeding, please be reminded that the test $\exists$ $A_i \in \boldsymbol{C}$ B_I or the similar $\exists$ not $A_i \in \boldsymbol{C}$ C_i means $\exists$ unique A_i (not A_i).
Note2. *The initial step can be used also for reducing the program, for instance by removing the clauses that cannot be true because of the disjunctions contained.*

Transform_1(P)
input : logic program P; output : transformed logic program P'
begin
for each clause CL = $\{\boldsymbol{D}\ T \leftarrow \boldsymbol{C}\ H, \neg\ \boldsymbol{C}\ K\} \in P$
do { $\boldsymbol{C}$ B' : = ϕ $\boldsymbol{D}$ D' : = ϕ $\boldsymbol{D}$ A : = ϕ $\neg$ $\boldsymbol{C}$ C' : = ϕ
$\boldsymbol{C}$ P : = ϕ $\neg$ $\boldsymbol{C}$ Q : = ϕ
for each clause CLB = $\{\boldsymbol{D}\ D_i \leftarrow \boldsymbol{C}\ B_i, \neg\ \boldsymbol{C}\ C_i\} \in P$
CL $\neq$ CLB
do { if $\exists$ $A_i \in \boldsymbol{C}$ B_i s.t. $A_i \in \boldsymbol{D}$ T
$\boldsymbol{C}$ B' : = $\boldsymbol{C}$ B' $\cup$ { $\boldsymbol{C}$ B_i - A_i}
$\neg$ $\boldsymbol{C}$ C' : = $\neg$ $\boldsymbol{C}$ C' $\cup$ { $\neg$ $\boldsymbol{C}$ C_i}
$\boldsymbol{D}$ D' : = $\boldsymbol{D}$ D' $\cup$ $\boldsymbol{D}$ D_i $\boldsymbol{D}$ A
: = $\boldsymbol{D}$ A $\cup$ {A_i} }

{ if $\exists$ not $A_i \in \boldsymbol{C}$ C_i s.t.. not $A_i \in \sim \boldsymbol{D}$ T
$\neg$ $\boldsymbol{C}$ C' : = $\neg$ $\boldsymbol{C}$ C' $\cup$ {$\neg\boldsymbol{C}$ C_i - not A_i}
$\boldsymbol{C}$ P : = $\boldsymbol{C}$ P $\cup$ $\boldsymbol{C}$ B_i $\neg$ $\boldsymbol{C}$ Q : = $\neg$ $\boldsymbol{C}$ Q $\cup$ $\neg\boldsymbol{C}$ C_i
$\boldsymbol{D}$ D' : = $\boldsymbol{D}$ D' $\cup$ $\boldsymbol{D}$ D_i $\neg$ $\boldsymbol{D}$
D : = $\boldsymbol{D}$ D $\cup$ {not A_i} }
C1 : = {$\boldsymbol{D}$ D' $\boldsymbol{D}$ A , $\boldsymbol{C}$ B', $\neg$ $\boldsymbol{C}$ C'}

C2 : = { $\boldsymbol{D}$ D'
$\neg$ $\boldsymbol{D}$ D , $\boldsymbol{C}$ P, $\neg$ $\boldsymbol{C}$Q}
C3 : = redundancy(C1,P) P' : = P' $\cup$ {C1}
- {C3}
P : = P $\cup$ {C1} - {C2} C4 : = redundancy(C2,P)

P' : = P' $\cup$ {C2} - {C4} P : = P $\cup$ {C2} - {C4}
} P' : = P; return P'
end

Note that every iteration of 'for' potentially adds a new clause that will be compared with all the other clauses, and that the maximum length of the clauses increases step by step. We do have a basic question: does the procedure end after a finite number of

steps ?
The answer is yes, for the following reasons:

- Theorem of Max Extension of P limits the number of distinct literals that can be built from P;
- Lemma 4. 3 relates the above limit with the maximum length of a clause;
- The number of clauses that can be created from Base(P) is limited.

We will now introduce several Lemmas, useful for determining the cost of *Transform_1*.

Lemma 4. 4. Let P(i) be the program obtained after i iterations of 'for' cycle in *Transform_1*. If M(0) is the number of initial clauses, then the max number of clauses after i steps is: $M(i) \leq M(0) + i$

In order to determine the maximum number of steps, we need to determine the maximum number of clauses that can be formed in ***L*** = $\{$Base(P),$\vee$,$\wedge$, ,$\neg\}$.
The following scheme of reasoning can be applied:

- Lemma 4. 1 states the maximum number of atoms that can be created from the program P; we will name this set Atoms(P);
- Lemma 4. 3 states that the max length of a clause is $2 * 3^N$;
- we will consider all the subsets of Atoms(P) with a size $\leq 2 * 3^N$;
- given a subset of Atoms(P), we can determine all the possible clauses by varying the head of the clause among the elements in Atoms(P).

Summarized, this becomes :

$\forall$ Atom $\in$ Powerset(Atoms(P)) s. t. D = (unknown char)Atom(unknown char) $\leq 2 * 3^N$ there are D possible formulae.

Example 4. 2 *Given the subset* $\{$a,b,(c $\vee$ d)$\}$ $\subseteq$ Powerset(Atoms(P)) *we can form the clauses:*

a b,(c $\vee$ d) b a,(c $\vee$ d) (c $\vee$ d) a,b

Formally, we have:

Lemma 4. 5. Let P be a disjunctive logic program such as N = (unknown char)Base(P)(unknown char) and T = (unknown char)Atoms(P)(unknown char); then the maximum number of clauses that can be created from P is $M_{max} \leq T/2 * 2^T$

Note 3. *The result actually overestimates the number of the generable clauses because it does not consider contradictory and redundant clauses. To do so, assumptions would be needed, about the number of disjunctions the given program is supposed to contain, an about the mumber of atoms these disjunctions are composed of.*

In Lemma 4. 5, we consider only the clauses that have a disjunction in the head, so we need to determine how many clauses with a disjunction in the head we can create.

Lemma 4. 6. Given a disjunctive logic program P where N = (unknown char)Base(P)(unknown char), there are up to 3^{2N} clauses with a disjunction in the head.

Let us now assess the cost of *Transform_1* for time and space.

The algorithm *Transform_1* contains two nested "for" cycles:

- the outermost cycle considers all clauses of P, and all clauses generated in previous steps.
- the innermost cycle considers all clauses of P, except the one selected by the external cycle.

From Lemma 4. 4 and Lemma 4. 6 we conclude that the maximum number i of steps of *Transform_1* is $i \leq 3^{2N}$ - M(0)

Let C be an estimate of the cost C of the inner block of instructions, and let M_{max} be the maximum number of clauses with a disjunctive conclusion that can be possibly generated. Then, the cost in time, depending on the number M of clauses, is:

$$T(M) = C * M(i-1))^2 = C * M(0) + i - 1)^2 \leq C * M(0) + i)^2 \cong C * (M_{max})^3$$

Since $M_{max} = 3^{2N}$ (from Lemma 4. 6) and N = n/2, we get $T(n) = C * (3^n)^3 \cong C * 3^{3n}$

Then, we have a cost in time which is exponential in the length n of the original program. In particular, since EXPTIME is the set of all decision problems solvable in time bounded by $2^{p(n)}$ and since we have $3^{2n} \leq 2^{4n}$, we can conclude that the time complexity of the PT is in EXPTIME.

An upper bound for space can be obtained by considering the global length of all clauses that can be generated, i. e. at most one for each processing step. From Lemma 4. 3, we have that the maximum length of a single clause is $4N * 3^N$, and then we have:

$$S(N) = (\text{Num. steps}) * L_{max} = 3^{2N} * 4N * 3^N \cong 4N\, 3^{3N}$$

The length of P after the base step is: $S(n) = n + 4\,(n/2) * 3^{3(n/2)} \cong n + 2n * 3^{2n}$

which is again an exponential value with respect to the original length n. In particular, since EXPSPACE is the set of all decision problems solvable in space bounded by $2^{p(n)}$ and since we have $3^{2n} \leq 2^{4n}$, we can conclude that the space complexity of the PT is in EXPSPACE.

In summary, given disjunctive logic program P, the base step can be applied by a deterministic Turing Machine in exponential time and space.

Observation *Notice that this is an overestimation of complexity, since time and space costs are actually not reachable. A reasonable upper bound can be found if making hypotheses on the class of programs under consideration.*

Below is the algorithm that implements the intermediate step of the program transformation. Like for the base step, it is sufficient to compare couples of clauses, of which at least one is disjunctive. Also in this case a redundancy check is provided.

```
    Transform_2(P) input : disjunctive logic program P
output : disjunctive logic program P' modified by intermediate step
begin
            P' : = Apply_intermediate(P)
            while (P' ≠ P)
                        P : = P'
    P' : = apply_intermediate(P)
    end
```

Transform_2 iterates the intermediate step until the fixpoint is reached. It relies on the oracle *Apply_intermediate*, that scans the program so as to compare every couple of clauses, and possibly add a new one, according to the definitions in Section 3. The new clauses are not immediately added to P, rather they are temporarily stored in P', so as redundant ones can be discarded. Similarly to the base step, only disjunctive clauses can produce new clauses to add. Therefore, the starting point is the maximum number of disjunctive clauses.

Lemma 4. 7. Given disjunctive logic program P, with N = (unknown char)Base(P)(unknown

char) and T = (unknown char)Atoms(P)(unknown char), the maximum number of disjunctive clauses is (T - 2N) * 2^{T-1}.
Apply_intermediate will compare disjunctive clauses with the others, and then

Comparisons (N,T) = (disjunctive clauses) * (clauses) =

= (by Lemma 4. 5 and Lemma 4. 7) = (T - 2N) 2^{T-1} * T 2^{T-1} = 2^{2T} (T^2 - 2N) / 4

Since *Transform_2* calls *Apply_intermediate* up to (T/2)*2^T times, the total number of comparisons is

Comparisons(N,T) ≅ (T^3 - 2NT) * 2^{3T} / 8

If C is the cost of a single comparison , by taking T = 3^N and by making the pessimistic assumption N=n/2, we get T(n) = C ($3^{3n/2}$ - n $3^{n/2}$) * 2exp($3^{n/2}$) / 8
which is exponential in the length of the original program P. As far as space is concerned, the result is the same as for the base step, i. e.

S(n) ≅ n + 2n * 3^{2n}

Observation *As before, the result applies to the worst case scenario because :*

- *the length of program P increases dynamically during the program transformation and it is not maximum at every iterations of Apply_intermediate, as assumed in the above discussion;*
- *each time Apply_intermediate is called, all the clauses added in step i are considered only from step (i+1) on*
- *the redundancy check introduced in Apply_intermediate reduces the length of P with an immediate effect in the current step*

The formal conclusion however is that the Program Transformation can be executed by a deterministic Turing machine working in exponential time/space with respect to the original length of P. Again, the results presented are very pessimistic because:

1. the maximization of N with n/2 means that we are working with programs which are not very meaningful;
2. all the transformations on the program apply to clauses with a disjunction in the head; usually a program should contain only few such clauses, because disjunction signifies lack of information;
3. the redundancy check removes a lot of clauses, but the actual number cannot be easily quantified because it highly depends on P.

5. Computational Complexity of the Iterated Fixpoint Computation (IFC)

After the Program Transformation has been applied to P, the Static Semantics of P can be computed by applying the Iterated Fixpoint Computation with a Modified Base Step. Based on Lemma 3. 2, a negative disjunction or atom is not believed true because of the lack of information, instead not A $\in T_J \uparrow 0$ if and only if J $|=_{min}$ not A and $\sim$ ***D*** D $\in T_J \uparrow 0$ if and only if J $|=_{min}$ $\sim$ ***D*** D.
Therefore, given J $\subseteq$ **B ,** $T_J \uparrow 0$ contains all the Atoms *not A* or the negative disjunctions $\sim$***D*** D = not $D_1 \vee ... \vee$ not D_n that are minimally implied by J. In $T_J \uparrow 1$ we can add all the facts in P and all the heads of clauses of P such that the body is in $T_J \uparrow 0$. Thus, at

each step up to M elements are added in P. In general in $T_J \uparrow i$ we can add :

- all the facts in P
- all the heads of clauses of P s. t. the body $\in T_J \uparrow (i - 1)$

Each iteration adds at least one element, because otherwise it means that we have already reached the fixed point. Therefore we have:

Lemma 5. 1. Given a disjunctive logic program P with M clauses and J $\subseteq$ **B** , the procedure for computing S(J) has to be iterated up to M+1 times.

Each iteration implies the scan of M clauses and of the set J, which has maximum dimension equal to **B .**

We will now consider an algorithm for computing S(J), *ComputeS(J,P)*, where P is the program and J the current belief set. Its main part is the algorithm *Apply_basic_step(J0,P)*, which applies the modified base step before the iteration. In fact, the complexity of *ComputeS* is heavily based upon the complexity of *Apply_basic step*. Let us remind the reader that S(J) = $T_J \uparrow \omega$.

Apply_basic_step(J0,P)
input : J0 = current belief set; P = disjunctive logic program
output : J1 = new belief set
begin
{ J1 : = ϕ
determine **B** = Extended Herbrand base of P
J^C : = **B** - J0
for each atom A in J^C
do { D1 : = {ˆ***D*** D : ˆ***D*** D contains A}
if ∀ ˆ***D*** D ∈ D1 , ˆ***D*** D ∈ J^C is verified
{ D2 : = {¬***D*** D : ¬***D*** D contains not A}
if ∀ ¬***D*** D ∈ D2 , ¬***D*** D ∈ J^C is verified
{ J1 : = J1 ∪ {not A} } }
D3 : = {ˆ***D*** D : ˆ***D*** D does not non contain A}
if ∀ ˆ***D*** D ∈ D3 , ˆ***D*** D ∈ J is verified
{ J1 : = J1 ∪ {not A} }
D4 : = {***D*** D : ***D*** D contains A}
∀ ***D*** D ∈ D4
{ J1 : = J1 ∪ {∼ ***D*** D} }
} return J1
}end

We may notice that:

- **B** can be determined by a sequential scan of P;
- D1, D2, D3, D4 can be determined by a sequential scan of **B ;**
- the test ***D*** D ∈ Set, where Set is either J or J^C , requires up to $\log_2$ ((unknown char)Set(unknown char)) comparisons.

Since J and J^C constitute a partition of **B ,** we'll test either ***D*** D ∈J0 or ***D*** D ∈ J^C depending on the truth of (unknown char)J0(unknown char) < (unknown char)J^C(unknown char).

This implies that the search time is no more than (1/2)*$\log_2$((unknown char)**B** (unknown char) = (1/2) + (1/2)$\log_2$ n

The cost of the algorithm *Apply_basic_step* is composed by:

- the cost of determining **B** , which is linear in the length n of P
- the cost of determining J^C, which is linear in the length 2n of **B** • for each of the max 2n atoms of J^C, the cost of identifying the sets D1, D2, D3, D4 is 2n, since one scan of **B** is sufficient. Given that (unknown char)D1(unknown char) + (unknown char)D2(unknown char) + (unknown char)D3(unknown char) + (unknown char)D4(unknown char) $\leq$ (unknown char)**B** (unknown char), we obtain

$2n * (1/2+1/2\log_2 n) = n + n \log_2 n \leq n^2$

Globally we have that Time(*Apply_basic_step*) = n + 2n (n^2 + 2n) = 7n + $2n^3$ = o($2n^3$)

Let's focus now on the cost of the algorithm *ComputeS*. We can identify the following components:

- the cost of the base step is $2n^3$;
- the identification of all the facts in P has linear cost in n because we only need to scan P once;
- the while loop, executed for each step of P,but the facts, checks if each atom is in J1. P' has up to n clauses, therefore

$$n * (1/2+1/2\log_2 n) = n/2 + n/2 (\log_2 n) \approx n^2.$$

How many times will the while cycle be executed ? The sets J0 and J1 can have for 1 to 2n elements. However, the following Lemma holds.

Lemma 5. 2. If J0 contains all the Atoms that are in the head of the clauses of P, then J1 = T_{J0} = J0 and we have reached the fixed point.

The Lemma implies that the while cycle in the algorithm *ComputeS* won't be executed more than M times, where M is the number of clauses in P. Again, we can maximize M with n and so

$$\text{Time}(\textit{ComputeS}) = 2n^3 + n + M * n^2 \leq 2n^3 + n + n^3 \cong 3n^3$$

The algorithm *Compute*Σ(J,P) determines $\Sigma=S^2$ and is very simple because, basically, it just invokes the algorithm *ComputeS* twice. It won't be reported here for its simplicity. Obviously

Time(*Compute*Σ) = $6n^3$

We are now ready to evaluate the algorithm *Compute_WFC* for the Iterated Fixpoint Computation (IFC) of the Well-Founded Completion of P.

Since it can be proved that the maximum number of iterations of Σ for reaching the fixpoint is M+1, Time(*Compute_WFC*) = (M + 1) * Time(*Compute*Σ) = (M + 1) * $6n^3 \leq 6n^4$

Let us summarize what we found so far:

The Iterated Fixpoint Computation for a disjunctive logic program P modified by the Program Transformation can be computed in polynomial time, so its complexity is in P.

As concerns the resource space, the IFC requires polynomial space and so it is in PSPACE. However, if we consider the length of P before the Program Transformation, the space required is exponential.

6. Summary of the results obtained and possible improvements

So far we have proved that PT is exponential in time and can expand the length of P exponentially, while IFC is polynomial quadratic in time and linear in space (based on the length of P after PT). Because of the expansion of the program after PT, the IFC can become not feasible, even if it has a quadratic cost itself. The problem applies to

both time and space resources.
However, we have experimented this approach on many examples of different sizes and different kinds. We have considered all the examples presented in [LMR92] and [Prz94]). We have also considered randomly generated examples, up to a lenght of twenty clauses (since we do not have a good implementation yet), varying the number and the length of the disjunctions. The results of the experiments tell us the following. In most cases, the PT increases linearly the length of P. In some cases, the PT actually reduces the length of P by eliminating redundancies. This process is very similar to the Davis-Putnam procedure. More specifically, the PT will remove: (i) redundant clauses (the shortest clauses are kept); (ii) clauses that contain at least two atoms in their body that appear in a disjunction and that therefore cannot be verified for the exclusive OR assumption; (iii) the atoms in the body that are also present as facts in the program.

In general though, the computational feasibility can be guaranteed in two ways.

1. Working only with programs that do not expand exponentially after PT. Typically, those programs satisfy at least one of these conditions : (a) contain a lot of facts (b) contain only few disjunctions; (c) contain few distinct propositional symbols; (d) many atoms are either only in the head or only in the body of clauses.
2. Given a program P and a query A, applying IFC only on a subset of the program P, called P'_A that is the "relevant set" of P with respect to the query A.

The basic idea that we want to support is that, given a program P and a query Q, only a subset of P is used during the deduction process.
Should this idea be valid, it is possible to proceed in the following way :

1. Execute *off-line* PT and store P'=PT(P) on external memory.
2. Given query Q, determine subset P"=Relevant_Subset(P',Q).
3. Apply IFC on P" and then verify Q.

On (2), we specifically refer to the principle of Relevance, detailed in [Dix94] and [AB94]. In particular, a semantics verifies the Relevance principleif the following holds:

given a program P, and a literal L, the truth value assigned to L depends only on the subset rel_rul(P,L) of rules "relevant" to L, so that

SEM(P)(L) = SEM (rel_rule(P,L)) (L)

The STATIC Semantics verifies the Principle of Relevance. Therefore, the IFC will be applied to P"= *rel_rul*(P' = PT(P))

All the definitions of dependency graph, *dependencies_of* and *rel_rul* introduced in [Dix94] and [AB94] do not need to be changed because of the disjunction. In fact, if the query Q is constituted of conjunctions of literals, Q= $C_1 \cup C_2 \cup \ldots C_n$ rel_rul (P, Q) =)
The set *rel_rule* can be determined in polynomial, quadratic time. Based on that, we can redefine

Time(query_answering_on-line) =
Time(program_reduction from P' to P") +
Time(Iterated_Fixpoint_Construction) = $(n')^2 + (6\, n")^4$ $(*)$

Where n'= (unknown char)P'(unknown char) e n"= (unknown char)P"(unknown char)
Please be reminded that originally it was
Time(query_answering_on-line) = Time(Iterated_Fixpoint_Construction) = $6(n')^4$ $(**)$
With a reduction of just 1/10 from P' to P", we have a significant decrease in the time

and the reduction becomes bigger as n' grows.

7. Concluding remarks

We have assessed the computational complexity of Static Semantics, according to the definition proposed by Costantini and Lanzarone, based on the notion of Knowledge Compilation. The computation is here divided in two phases: a pre-processing Program Transformation (PT) and a building phase IFC.

The results presented in previous sections show a complexity for the PT which is **2-EXPTIME** for the resource time and **2-EXPSPACE** for the resource space, so the PT is intractable (according to our algorithm). However, these results are really pessimistic since we overestimate many factors (in order not to restrict ourselves to some particular class of programs, on which more precise hypotheses could be made). Moreover, the redundancy check and the transformation itself might in many cases reduce the program length. The actual cost of the PT highly depends on the program at hand, and in particular on the number of distinct propositional symbols, the number of disjunctions, the occurrence of the atoms in the head and body of clauses. In fact, the experiments show that in most practical cases the PT has a linear cost.
The IFC has a quadratic cost for the resource time and a linear cost in the length of the transformed program P for the resource space. Also in this case the analysis is for the worst case scenario. Since the IFC is performed for each query answering request, the resource time is crucial. For this reason we propose to introduce a step of query pre-processing with the aim of reducing the part of the program considered in the IFC. This is feasible because the Static Semantics verifies the Relevance Principle (see [Dix94]); the reduction in the size of the program considered for IFC is particularly important for programs with a lot of facts.

Throughout the paper we have produced a basic sequence of operations required for defining a prototype of interpreter for both PT and IFC. Among the possible future developments, we see the following: characterization of computationally feasible programs, also in relation to known classes of programs, such as disjunctive stratified programs; implementation of a prototype of interpreter.

References

[AB94] K. R. Apt, R. Bol, Logic Programming and Negation: A Survey*, Journal of Logic Programming* 39/40 , 1994.

[CL95] S. Costantini, G. A. Lanzarone, Static semantics as program transformation and well-founded computation. In: J. Dix, L. M. Pereira, T. C. Przymusinski (eds.), *Non-Monotonic Extensions of Logic Programming,* LNAI 927, 1995.

[CM99] S. Costantini, T. Morbidoni, On the computational Complexity of Static Semantics for Disjunctive Logic Programs, Extended Version. *Technical Report, Univ. of L'Aquila*, 1999.

[CD98] M. Cadoli, F. Donini, A Survey on Knowledge Compilation, *AI Communication – The European Journal for Artificial Intelligence* 10, 1998.

[CS93] M. Cadoli, M. Schaerf, A Survey of Complexity Results for Non-Monotonic Logics, *Journal of Logic Programming*, 17,1993

[Dix94] J. Dix, Semantics of Logic Programs : Their Intuitions and Formal Properties. An Overview, in: Andre Fuhrmann and Hans Rott (eds.), *Logic, Action and Information*, Proc. of the Konstanz Colloquium in Logic and Information (LogIn '92), 1992.

[EG92] T. Eiter, G. Gottlob, Complexity Results for Disjunctive Logic Programming and Application to Nonmonotonic Logics, in: Proc. of the International Logic Programming Symposium, 1993.

[EG92b] T. Eiter, G. Gottlob, Reasoning with Parsimonious and Moderately Grounded Expansions, *Fundamenta Informaticae* 17, 1992.

[LMR92] J. Lobo, J. Minker, A. Rajasekar, *Foundations of Disjunctive Logic Programming*, The MIT Press, Cambridge, Massachusetts, 1992.

[Prz89] T. C. Przymusinski, Every logic program has a natural stratification and an iterated fixpoint model. In: *Proc. of the Eight ACM SIGACT-SIGMOD-SIGART Symposium onPrinciples of Database Systems*, ACM Press, Philadelphia, Pennsylvania, 1989.

[Prz94] T. C. Przymusinski, Static semantics for normal and disjunctive logic programs. *The Annals of Mathematics and Artificial Intelligence*, Special Issue on Disjunctive Programs, 1994.

[Sch92] J. S. Schlipf, A Survey of Complexity and Undecidability Results in Logic Programming,. *The Annals of Mathematics and Artificial Intelligence*, vol. 15, 1995.

[VGRS91] A. Van Gelder, K. A. Ross, J. S. Schlipf. The well-founded semantics for general logic programs. *JACM* 38, 1991.