# Reducing Propositional Theories in Equilibrium Logic to Logic Programs

Pedro Cabalar[1], David Pearce[2], and Agustín Valverde[3]

[1] Dept. of Computation, Univ. of Corunna, Spain.
cabalar@dc.fi.udc.es
[2] Dept. of Informatics, Statistics and Telematics,
Univ. Rey Juan Carlos, (Móstoles, Madrid), Spain.
d.pearce@escet.urjc.es
[3] Dept. of Applied Mathematics, Univ. of Málaga, Spain.
a_valverde@ctima.uma.es

**Abstract.** The paper studies reductions of propositional theories in equilibrium logic to logic programs under answer set semantics. Specifically we are concerned with the question of how to transform an arbitrary set of propositional formulas into an equivalent logic program and what are the complexity constraints on this process. We want the transformed program to be equivalent in a strong sense so that theory parts can be transformed independent of the wider context in which they might be embedded. It was only recently established [1] that propositional theories are indeed equivalent (in a strong sense) to logic programs. Here this result is extended with the following contributions. (i) We show how to effectively obtain an equivalent program starting from an arbitrary theory. (ii) We show that in general there is no polynomial time transformation if we require the resulting program to share precisely the vocabulary or signature of the initial theory. (iii) Extending previous work we show how polynomial transformations can be achieved if one allows the resulting program to contain new atoms. The program obtained is still in a strong sense equivalent to the original theory, and the answer sets of the theory can be retrieved from it.

## 1 Introduction

Answer set programming (ASP) is fast becoming a well-established environment for declarative programming and AI problem solving, with several implemented systems and advanced prototypes and applications. Though existing answer set solvers differ somewhat in their syntax and capabilities, the language of disjunctive logic programs with two negations, as exemplified in the `DLV` system [11] under essentially the semantics proposed in [7], provides a standard reference point. Many systems support different extensions of the language, either through direct implementation or through reductions to the basic language. For example weight constraints are included in `smodels` [24], while a system called `nlp` [22] for compiling nested logic programs is available as a front-end to `DLV`. Though differently motivated, these two kinds of extensions are actually closely related,

since as [6] shows, weight constraints can be represented equivalently by nested programs of a special kind.

Answer set semantics was already generalised and extended to arbitrary propositional theories with two negations in the system of *equilibrium logic*, defined in [17] and further studied in [18,19,20]. Equilibrium logic is based on a simple, minimal model construction in the nonclassical logic of here-and-there (with strong negation), and admits also a natural fixpoint characterisation in the style of nonmonotonic logics. In [20,13] it was shown that answer set semantics for nested programs [12] is also captured by equilibrium models.

While nested logic programs permit arbitrary boolean formulas to appear in the bodies and heads of rules, they do not support embedded implications; so for example one cannot write in `nlp` a rule with a conditional body, such as

$$p \leftarrow (q \leftarrow r).$$

In fact several authors have suggested the usefulness of embedded implications for knowledge representation (see eg [3,8,23]) but proposals for an adequate semantics have differed. Recently however Ferraris [5] has shown how, by modifying somewhat the definition of answer sets for nested programs, a natural extension for arbitrary propositional theories can be obtained. Though formulated using program reducts, in the style of [7,12], the new definition is also equivalent to that of equilibrium model. Consequently, to understand propositional theories, hence also embedded implications, in terms of answer sets one can apply equally well either equilibrium logic or the new reduct notion of [5]. Furthermore, [5] shows how the important concept of *aggregate* in ASP, understood according to the semantics of [4], can be represented by rules with embedded implications. This provides an important reason for handling arbitrary theories in equilibrium logic and motivates the topic of the present paper.

We are concerned here with the question how to transform a propositional theory in equilibrium logic into an equivalent logic program and what are the complexity constraints on this process. We want the transformed theory to be equivalent in a strong sense so that theory parts can be translated independent of the wider context in which they might be embedded. It was only recently established [1] that propositional theories are indeed equivalent (in a strong sense) to logic programs. The present paper extends this result with the following contributions. (i) We present an alternative reduction method which seems more interesting for computational purposes than the one presented in [1], as it extends the unfolding of nested expressions shown in [12] and generally leads to simpler logic programs. (ii) We show that in general there is no polynomial transformation if we require the resulting program to share precisely the vocabulary or signature of the initial theory. (iii) Extending the work of [15,16] we show how polynomial transformations can be achieved if one allows the resulting program to contain new atoms. The program obtained is still in a strong sense equivalent to the original theory, and the answer sets of the latter can be retrieved from the answer sets of the former.

## 2 Equilibrium Logic

We assume the reader is familiar with answer set semantics for disjunctive logic programs [7]. As a logical foundation for answer set programming we use the nonclassical logic of here-and-there, denoted here by **HT**, and its nonmonotonic extension, *equilibrium logic* [17], which generalises answer set semantics for logic programs to arbitrary propositional theories (see eg [13]). We give only a very brief overview here, for more details the reader is referred to [17,13,19] and the logic texts cited below.[4]

Given a propositional signature $V$ we define the corresponding propositional language $\mathcal{L}_V$ as the set of formulas built from atoms in $V$ with the usual connectives $\top, \bot, \neg, \wedge, \vee, \rightarrow$. A *literal* is any atom $p \in V$ or its negation $\neg p$. Given a formula $\varphi \in \mathcal{L}_V$, the function $subf(\varphi)$ represents the set of all subformulas of $\varphi$ (including $\varphi$ itself), whereas $vars(\varphi)$ is defined as $subf(\varphi) \cap V$, that is, the set of atoms occurring in $\varphi$. By $degree(\varphi)$ we understand the number of connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$ that occur in the formula $\varphi$. Note that $|subf(\varphi)|$ would be at most[5] $degree(\varphi) + |vars(\varphi)|$ plus the number of occurrences of $\top$ and $\bot$ in $\varphi$.

As usual, a set of formulas $\Pi \subseteq \mathcal{L}_V$ is called a *theory*. We extend the use of $subf$ and $vars$ to theories in the obvious way. The degree of a theory, $degree(\Pi)$, is defined as the degree of the conjunction of its formulas.

The axioms and rules of inference for **HT** are those of intuitionistic logic (see eg [2]) together with the axiom schema:

$$(\neg\alpha \rightarrow \beta) \rightarrow (((\beta \rightarrow \alpha) \rightarrow \beta) \rightarrow \beta)$$

The model theory of **HT** is based on the usual Kripke semantics for intuitionistic logic (see eg [2]), but **HT** is complete for Kripke frames $\langle W, \leq \rangle$ (where as usual $W$ is the set of points or worlds and $\leq$ is a partial-ordering on $W$) having exactly two worlds say $h$ ('here') and $t$ ('there') with $h \leq t$. As usual a *model* is a frame together with an assignment $i$ that associates to each element of $W$ a set of *atoms*, such that if $w \leq w'$ then $i(w) \subseteq i(w')$; an assignment is then extended inductively to all formulas via the usual rules for conjunction, disjunction, implication and negation in intuitionistic logic. It is convenient to represent an **HT** model as an ordered pair $\langle H, T \rangle$ of sets of atoms, where $H = i(h)$ and $T = i(t)$ under a suitable assignment $i$; by $h \leq t$, it follows that $H \subseteq T$.

A formula $\varphi$ is true in an **HT** model $\mathcal{M} = \langle H, T \rangle$, in symbols $\mathcal{M} \models \varphi$, if it is true at each world in $\mathcal{M}$. A formula $\varphi$ is said to be *valid* in **HT**, in symbols $\models \varphi$, if it is true in all **HT** models. Logical consequence for **HT** is understood as follows: $\varphi$ is said to be an **HT** consequence of a theory $\Pi$, written $\Pi \models \varphi$,

[4] As in some ASP systems the standard version of equilibrium logic has two kinds of negation, intuitionistic and strong negation. For simplicity we deal here with the restricted version containing just the first negation and based on the logic of here-and-there. So we do not consider here eg logic programs with strong or explicit negation.

[5] It would be strictly lower if we have repeated subformulas.

iff for all models $\mathcal{M}$ and any world $w \in \mathcal{M}$, $\mathcal{M}, w \models \Pi$ implies $\mathcal{M}, w \models \varphi$. Equivalently this can be expressed by saying that $\varphi$ is true in all models of $\Pi$.

Clearly **HT** is a 3-valued logic (usually known as Gödel's 3-valued logic) and we can also represent models via interpretations $I$ that assign to every atom $p$ a value in $\mathbf{3} = \{0, 1, 2\}$, where 2 is the designated value. Given a model $\langle H, T\rangle$, the corresponding 3-valued interpretation $I$ would assign, to each atom $p$, the value: $I(p) = 2$ iff $p \in H$; $I(p) = 1$ iff $p \in T \setminus H$ and $I(p) = 0$ iff $p \notin T$. An assignment $I$ is extended to all formulas using the interpretation of the connectives as operators in **3**. As a result, conjunction becomes the minimum value, disjunction the maximum, and implication and negation are evaluated by:

$$I(\varphi \to \psi) = \begin{cases} 2 & \text{if } I(\varphi) \leq I(\psi) \\ I(\psi) & \text{otherwise} \end{cases} \qquad I(\neg\varphi) = \begin{cases} 2 & \text{if } I(\varphi) = 0 \\ 0 & \text{otherwise} \end{cases}$$

For each connective $\bullet \in \{\wedge, \vee, \to, \neg\}$, we will denote the corresponding 3-valued operator as $f^{\bullet}$.

Equilibrium models are special kinds of minimal **HT**-models. Let $\Pi$ be a theory and $\langle H, T\rangle$ a model of $\Pi$. $\langle H, T\rangle$ is said to be *total* if $H = T$. $\langle H, T\rangle$ is said to be an *equilibrium model* if it is total and there is no model $\langle H', T\rangle$ of $\Pi$ with $H' \subset H$. The expression $Eq(V, \Pi)$ denotes the set of the equilibrium models of theory $\Pi$ on signature $V$. *Equilibrium logic* is the logic determined by the equilibrium models of a theory. It generalises answer set semantics in the following sense. For all the usual classes of logic programs, including normal, disjunctive and nested programs, equilibrium models correspond to answer sets. The 'translation' from the syntax of programs to **HT** propositional formulas is the trivial one, eg. a ground rule of a disjunctive program of the form

$$q_1 \vee \ldots \vee q_k \leftarrow p_1, \ldots, p_m, not\, p_{m+1}, \ldots, not\, p_n$$

where the $p_i$ and $q_j$ are atoms, corresponds to the **HT** sentence

$$p_1 \wedge \ldots \wedge p_m \wedge \neg p_{m+1} \wedge \ldots \wedge \neg p_n \to q_1 \vee \ldots \vee q_k$$

**Proposition 1 ([17,20,13]).** *For any logic program $\Pi$, an* **HT** *model $\langle T, T\rangle$ is an equilibrium model of $\Pi$ if and only if $T$ is an answer set of $\Pi$.*

Two theories, $\Pi_1$ and $\Pi_2$ are said to be *logically equivalent*, in symbols $\Pi_1 \equiv \Pi_2$, if they have the same **HT** models. They are said to be *strongly equivalent*, in symbols $\Pi_1 \equiv_s \Pi_2$, if and only if for any $\Pi$, $\Pi_1 \cup \Pi$ is equivalent to (has the same equilibrium models as) $\Pi_2 \cup \Pi$. The two notions are connected via:

**Proposition 2 ([13]).** *Any two theories, $\Pi_1$ and $\Pi_2$ are strongly equivalent iff they are logically equivalent, ie. $\Pi_1 \equiv_s \Pi_2$ iff $\Pi_1 \equiv \Pi_2$.*

Strong equivalence is important because it allows us to transform programs or theories to equivalent programs or theories independent of any larger context in which the theories concerned might be embedded. Implicitly, strong equivalence assumes that the theories involved share the same vocabulary, a restriction that has been removed in [21]. Here, in §4 below, we use a slight generalisation of strong equivalence, where we allow one language to include the other.

## 3 Vocabulary-preserving transformations

We first show how to transform an arbitrary theory into a strongly equivalent logic program in the same signature. [1] explores a similar aim but uses a transformation motivated by obtaining a simple proof of the existence of a translation, rather than the simplicity of the resulting programs or the final number of steps involved. To give an example, using the translation in [1], a simple program rule like $\neg a \rightarrow b \vee c$ would be first transformed to remove negations and disjunctions and then converted into a (nested) logic program via a bottom-up process (starting from subformulas) which eventually yields the program:

$$\neg a \rightarrow b \vee c \vee \neg b, \quad (b \vee \neg c) \wedge \neg a \rightarrow b, \quad \neg a \rightarrow b \vee c \vee \neg c, \quad (c \vee \neg b) \wedge \neg a \rightarrow c$$

The result would further require applying the unfolding rules of [12] to yield a non-nested program. Note that the original formula was already a non-nested program rule that did not need any transformation at all.

The transformation we present here adopts the opposite approach. It is a top-down process that relies on the successive application of several rewriting rules that operate on sets (conjunctions) of implications. A rewriting takes place whenever one of those implications does not yet have the form of a (non-nested) program rule.

Two sets of transformations are described next. A formula is said to be in *negation normal form* (NNF) when negation is only applied to literals. As a first step, we describe a set of rules that move negations inwards until a NNF is obtained:

$$\neg\top \Leftrightarrow \bot \quad (1) \qquad \neg(\varphi \wedge \psi) \Leftrightarrow \neg\varphi \vee \neg\psi \quad (4)$$
$$\neg\bot \Leftrightarrow \top \quad (2) \qquad \neg(\varphi \vee \psi) \Leftrightarrow \neg\varphi \wedge \neg\psi \quad (5)$$
$$\neg\neg\neg\varphi \Leftrightarrow \neg\varphi \quad (3) \qquad \neg(\varphi \rightarrow \psi) \Leftrightarrow \neg\neg\varphi \wedge \neg\psi \quad (6)$$

Transformations (1)-(5) were already used in [12] to obtain the NNF of so-called *nested expressions* (essentially formulas without implications). Thus, we have just included the treatment of a negated implication (6) to obtain the NNF in the general case.

In the second set of transformations, we deal with sets (conjunctions) of implications. Each step replaces one of the implications by new implications to be included in the set. If $\varphi$ is the original formula, the initial set of implications is the singleton $\{\top \rightarrow \varphi\}$. Without loss of generality, we assume that any implication $\alpha \rightarrow \beta$ to be replaced has been previously transformed into NNF. Furthermore, we always consider that $\alpha$ is a conjunction and $\beta$ a disjunction (if not, we just take $\alpha \wedge \top$ or $\beta \vee \bot$, respectively), and that we implicitly apply commutativity of conjunction and disjunction as needed.

Left side rules:

$$\top \wedge \alpha \rightarrow \beta \;\Leftrightarrow\; \{\; \alpha \rightarrow \beta \;\} \quad (L1)$$
$$\bot \wedge \alpha \rightarrow \beta \;\Leftrightarrow\; \varnothing \quad (L2)$$
$$\neg\neg\varphi \wedge \alpha \rightarrow \beta \;\Leftrightarrow\; \{\; \alpha \rightarrow \neg\varphi \vee \beta \;\} \quad (L3)$$

$$(\varphi \vee \psi) \wedge \alpha \rightarrow \beta \;\Leftrightarrow\; \left\{ \begin{array}{l} \varphi \wedge \alpha \rightarrow \beta \\ \psi \wedge \alpha \rightarrow \beta \end{array} \right\} \tag{L4}$$

$$(\varphi \rightarrow \psi) \wedge \alpha \rightarrow \beta \;\Leftrightarrow\; \left\{ \begin{array}{l} \neg\varphi \wedge \alpha \rightarrow \beta \\ \psi \wedge \alpha \rightarrow \beta \\ \alpha \rightarrow \varphi \vee \neg\psi \vee \beta \end{array} \right\} \tag{L5}$$

Right side rules

$$\alpha \rightarrow \bot \vee \beta \;\Leftrightarrow\; \{\, \alpha \rightarrow \beta \,\} \tag{R1}$$

$$\alpha \rightarrow \top \vee \beta \;\Leftrightarrow\; \varnothing \tag{R2}$$

$$\alpha \rightarrow \neg\neg\varphi \vee \beta \;\Leftrightarrow\; \{\, \neg\varphi \wedge \alpha \rightarrow \beta \,\} \tag{R3}$$

$$\alpha \rightarrow (\varphi \wedge \psi) \vee \beta \;\Leftrightarrow\; \left\{ \begin{array}{l} \alpha \rightarrow \varphi \vee \beta \\ \alpha \rightarrow \psi \vee \beta \end{array} \right\} \tag{R4}$$

$$\alpha \rightarrow (\varphi \rightarrow \psi) \vee \beta \;\Leftrightarrow\; \left\{ \begin{array}{l} \varphi \wedge \alpha \rightarrow \psi \vee \beta \\ \neg\psi \wedge \alpha \rightarrow \neg\varphi \vee \beta \end{array} \right\} \tag{R5}$$

As with NNF transformations, the rules (L1)-(L4), (R1)-(R4) were already used in [12] for unfolding nested expressions into disjunctive program rules. The additions in this case are transformations (L5) and (R5) that deal with an implication respectively in the antecedent or the consequent of another implication. In fact, an instance of rule (L5) where we take $\alpha = \top$ was the main tool used in [1] to provide the first transformation of propositional theories into logic programs. Note that rules (L5) and (R5) are the only ones to introduce new negations and that they both result in $\neg\varphi$ and $\neg\psi$ for the inner implication $\varphi \rightarrow \psi$. Thus, if the original propositional formula was in NNF, the computation of NNF in each intermediate step is needed for these newly generated $\neg\varphi$ and $\neg\psi$.

**Proposition 3.** *The set of transformation rules* (1)-(6), (L1)-(L5), (R1)-(R5) *is sound with respect to* **HT**, *that is,* $\models \varphi \leftrightarrow \psi$ *for each transformation rule* $\varphi \Leftrightarrow \psi$.

Of course, these transformations do not guarantee the absence of redundant formulas. As an example, when we have $\beta = \bot$ in (R5), we would obtain the pair of rules $\varphi \wedge \alpha \rightarrow \psi$ and $\neg\psi \wedge \alpha \rightarrow \neg\varphi$, but it can be easily checked that the latter follows from the former. A specialised version is also possible:

$$\alpha \rightarrow (\varphi \rightarrow \psi) \;\Leftrightarrow\; \{\, \varphi \wedge \alpha \rightarrow \psi \,\} \tag{R5'}$$

*Example 1.* Let $\varphi$ be the formula $(\neg p \rightarrow q) \rightarrow \neg(p \rightarrow r)$. Figure 1 shows a possible application of rules (L1)-(L5),(R1)-(R5). Each horizontal line represents a new step. The reference on the right shows the transformation rule that will be applied next to the corresponding formula on the left. From the final result we can remove[6] trivial tautologies and subsumed rules to obtain: $\{q \wedge \neg p \rightarrow \bot, q \rightarrow \neg r, \neg r \vee \neg p\}$ □

[6] In fact, the rule $q \rightarrow \neg r$ is redundant and could be further removed, although perhaps not in a directly automated way.

$$
\begin{array}{rll}
(\neg p \to q) \to \neg(p \to r) & & (\text{NNF}) \\
\hline
(\neg p \to q) \to \neg\neg p \wedge \neg r & & (L5) \\
\hline
q \to \neg\neg p \wedge \neg r & & \\
\neg\neg p \to \neg\neg p \wedge \neg r & & (L3) \\
\neg p \vee \neg q \vee \neg\neg p \wedge \neg r & & \\
\hline
q \to \neg\neg p \wedge \neg r & & (R4) \\
\neg\neg p \wedge \neg r \vee \neg p & & \\
\neg p \vee \neg q \vee \neg\neg p \wedge \neg r & & \\
\hline
q \to \neg\neg p & & (R3) \\
q \to \neg r & & \\
\neg\neg p \wedge \neg r \vee \neg p & & \\
\neg p \vee \neg q \vee \neg\neg p \wedge \neg r & & \\
\hline
q \wedge \neg p \to \bot & & \\
q \to \neg r & &
\end{array}
\qquad
\begin{array}{rll}
\neg\neg p \wedge \neg r \vee \neg p & & (R4) \\
\neg p \vee \neg q \vee \neg\neg p \wedge \neg r & & (R4) \\
\hline
q \wedge \neg p \to \bot & & \\
q \to \neg r & & \\
\neg\neg p \vee \neg p & & (R3) \\
\neg r \vee \neg p & & \\
\neg p \vee \neg q \vee \neg\neg p & & (R3) \\
\neg p \vee \neg q \vee \neg r & & \\
\hline
q \wedge \neg p \to \bot & & \\
q \to \neg r & & \\
\neg p \to \neg p & & \\
\neg r \vee \neg p & & \\
\neg p \to \neg p \vee \neg q & & \\
\neg p \vee \neg q \vee \neg r & &
\end{array}
$$

**Fig. 1.** Application of transformation rules in example 1

### 3.1 Complexity

Perhaps the main drawback of the method presented above is the exponential size of the generated program and the number of steps to obtain it. In fact, it was already observed in [16] that the set of rules (R1)-(R4) and (L1)-(L5) originally introduced in [12] (that is, when we do not consider nested implications) also leads to an exponential blow-up. The main reason is the presence of the "distributivity" laws (R4) and (L4). To give an example, just note that the successive application of (R4) to a formula like $(p_1 \wedge q_1) \vee (p_2 \wedge q_2) \vee \cdots \vee (p_n \wedge q_n)$ eventually yields $2^n$ disjunctions of $n$ atoms. In this section we show, however, that this drawback is actually inherent to *any* transformation that preserves the vocabulary of the original theory.

A recent result, presented in Theorem 2 in [1], shows that it is possible to build a strongly equivalent logic program starting from the set of countermodels of any arbitrary theory. The method for obtaining such a program is quite straightforward: it consists in adding, per each countermodel, a rule that refers to all the atoms in the propositional signature in a way that depends on their truth assignment in the countermodel. This result seems to point out that the complexity of obtaining a strongly equivalent program mostly depends on the generation of the countermodels of the original theory. Now, it is well-known that in classical logic this generation cannot be done in polynomial time because the validity problem is coNP-complete. This also holds for several finite-valued logics, and in particular for the family of Gödel logics (which includes **HT**) as shown in:

**Proposition 4 (Theorem 5 in [9]).** *The validity problem for Gödel logics for an arbitrary theory has a coNP-complete time complexity.* □

The keypoint for our complexity result comes from the observation that validity of a logic program rule can be checked in polynomial time. To see why, let

us consider for instance an arbitrary program rule like:

$$a_1 \wedge \cdots \wedge a_m \wedge \neg b_1 \wedge \cdots \wedge \neg b_n \rightarrow c_1 \vee \cdots \vee c_s \vee \neg d_1 \vee \cdots \vee \neg d_t \qquad (7)$$

with $m, n, s, t \geq 0$ and let us define the sets[7] of atoms $A = \{a_1, \ldots, a_m\}$, $B = \{b_1, \ldots, b_n\}$, $C = \{c_1, \ldots, c_s\}$ and $D = \{d_1, \ldots, d_t\}$.

**Lemma 1.** *An arbitrary rule like* (7) *(with $A, B, C, D$ defined as above) is valid in* **HT** *iff $A \cap B \neq \varnothing$ or $A \cap C \neq \varnothing$ or $B \cap D \neq \varnothing$.*

*Proof.* For the left to right direction, assume that (7) is valid but $A \cap B = \varnothing$, $A \cap C = \varnothing$ and $B \cap D = \varnothing$. In this situation, it is possible to build a 3-valued assignment $I$ where $I(a) = 2$, $I(b) = 0$, $I(c) \neq 2$ and $I(d) \neq 0$ for each $a, b, c, d$ in $A, B, C, D$ respectively. Note that for any atom in any of the remaining three possible intersections $A \cap D$, $B \cap C$ or $C \cap D$, assignment $I$ is still feasible. Now, it is easy to see that $I$ assigns 2 to the antecedent of (7) whereas it assigns a value strictly lower than 2 to its consequent. Therefore, $I$ is a countermodel for the rule, which contradicts the hypothesis of validity.

For the right to left direction, when $A \cap B \neq \varnothing$, it suffices to note that $p \wedge \neg p \wedge \alpha \rightarrow \beta$ is an **HT** tautology. Similarly, the other two cases are consequences of the **HT** tautology $\alpha \wedge \beta \rightarrow \alpha \vee \gamma$. □

Since checking whether one of the intersections $A \cap B$, $A \cap C$ or $B \cap D$ is not empty can be done by a simple sorting of the rule literals, we immediately get that validity for an arbitrary program rule has a polynomial time complexity.

**Theorem 1.** *There is no polynomial time algorithm to translate an arbitrary propositional theory in equilibrium logic into a strongly equivalent logic program in the same vocabulary (provided that $coNP \neq P$).*

*Proof.* Assume we had a polynomial algorithm that translates the theory into a strongly equivalent logic program. Checking whether this program is valid can be done by checking the validity of all its rules one by one. As validity of each rule can be done in polynomial time (by Lemma 1), the validity of the whole program is polynomial too. But then, the translation first plus the validity checking of the resulting program afterwards becomes altogether a polynomial time method for validity checking of an arbitrary theory. This contradicts Proposition 4, if we assume that $coNP \neq P$. □

## 4 Polynomial transformations

Let $I$ be an interpretation for a signature $U$ and let $V \subset U$. The expression $I \cap V$ denotes the interpretation $I$ restricted to signature $V$, that is, $(I \cap V)(p) = I(p)$ for any atom $p \in V$. For any theory $\Pi$, $subf(\Pi)$ denotes the set of all subformulas of $\Pi$.

[7] We can use sets instead of multisets without loss of generality, since **HT** satisfies the idempotency laws for conjunction and disjunction.

**Definition 1.** *We say that the translation $\sigma(\Pi) \subseteq \mathcal{L}_U$ of some theory $\Pi \subseteq \mathcal{L}_V$ with $V \subseteq U$ is* strongly faithful *if, for any theory $\Pi' \subseteq \mathcal{L}_V$:*

$$Eq(V, \Pi \cup \Pi') = \{J \cap V \mid J \in Eq(U, \sigma(\Pi) \cup \Pi')\}$$

The translations we will consider use a signature $V_{\mathbf{L}}$ that contains an atom (a label) for each non-constant formula in the original language $\mathcal{L}_V$, that is:

$$V_{\mathbf{L}} = \{\mathbf{L}_\varphi \mid \varphi \in \mathcal{L}_V \setminus \{\bot, \top\}\}$$

For convenience, we use $\mathbf{L}_\varphi \stackrel{\text{def}}{=} \varphi$ when $\varphi$ is $\top$, $\bot$ or an atom $p \in V$. This allows us to consider $V_{\mathbf{L}}$ as a superset of $V$. For any non-atomic formula $\varphi \bullet \psi$ built with a binary connective $\bullet$, we call its *definition*, $df(\varphi \bullet \psi)$, the formula:

$$\mathbf{L}_{\varphi \bullet \psi} \leftrightarrow \mathbf{L}_\varphi \bullet \mathbf{L}_\psi$$

Similarly $df(\neg\varphi)$ represents the formula $\mathbf{L}_{\neg\varphi} \leftrightarrow \neg\mathbf{L}_\varphi$.

**Definition 2.** *For any theory $\Pi$ in $\mathcal{L}_V$, we define the translation $\sigma(\Pi)$ as:*

$$\sigma(\Pi) \stackrel{\text{def}}{=} \{\mathbf{L}_\varphi \mid \varphi \in \Pi\} \cup \bigcup_{\gamma \in subf(\Pi)} df(\gamma)$$

That is, $\sigma(\Pi)$ collects the labels for all the formulas in $\Pi$ plus the definitions for all the subformulas in $\Pi$.

**Theorem 2.** *For any theory $\Pi$ in $\mathcal{L}_V$:* $\{I \mid I \models \Pi\} = \{J \cap V \mid J \models \sigma(\Pi)\}$.

*Proof.* Firstly note that $I \models \varphi$ iff $I(\varphi) = 2$ and $I \models \varphi \leftrightarrow \psi$ iff $I(\varphi) = I(\psi)$.
'$\subseteq$' direction: Let $I$ be a model of $\Pi$ and $J$ the assignment defined as $J(\mathbf{L}_\varphi) = I(\varphi)$ for any formula $\varphi \in \mathcal{L}_V$. Note that as $J(\mathbf{L}_p) = J(p) = I(p)$ for any atom $p \in V$, $J \cap V = I$. Furthermore, $J \models \mathbf{L}_\varphi$ for each formula $\varphi \in \Pi$ too, since $I \models \Pi$. Thus, it remains to show that $J \models df(\gamma)$ for any $\gamma \in subf(\Pi)$. For any connective $\bullet$ we have $J \models \mathbf{L}_{\varphi\bullet\psi} \leftrightarrow \mathbf{L}_\varphi \bullet \mathbf{L}_\psi$ because:

$$J(\mathbf{L}_{\varphi\bullet\psi}) = I(\varphi \bullet \psi) = f^\bullet(I(\varphi), I(\psi)) = f^\bullet(J(\mathbf{L}_\varphi), J(\mathbf{L}_\psi)) = J(\mathbf{L}_\varphi \bullet \mathbf{L}_\psi)$$

This same reasoning can be applied to prove that $J \models df(\neg\varphi)$.
'$\supseteq$' direction: We must show that $J \models \sigma(\Pi)$ implies $J \cap V \models \Pi$, that is, $J \models \Pi$. First, by structural induction we show that for any subformula $\gamma$ of $\Pi$, $J(\mathbf{L}_\gamma) = J(\gamma)$. When the subformula $\gamma$ has the shape $\top$, $\bot$ or an atom $p$ this is trivial, since $\mathbf{L}_\gamma = \gamma$ by definition. When $\gamma = \varphi \bullet \psi$ for any connective $\bullet$ then:

$$J(\mathbf{L}_{\varphi\bullet\psi}) \stackrel{*}{=} J(\mathbf{L}_\varphi \bullet \mathbf{L}_\psi) = f^\bullet(J(\mathbf{L}_\varphi), J(\mathbf{L}_\psi)) \stackrel{**}{=} f^\bullet(J(\varphi), J(\psi)) = J(\varphi \bullet \psi)$$

In $(*)$ we have used that $J \models df(\varphi \bullet \psi)$ and in $(**)$ we apply the induction hypothesis. The same reasoning holds for the unary connective $\neg$. Finally, as $J$ is a model of $\sigma(\Pi)$, in particular, we have that $J \models \mathbf{L}_\varphi$ for each $\varphi \in \Pi$. But, as we have seen, $J(\mathbf{L}_\varphi) = J(\varphi)$ and so $J \models \varphi$. □

| $\gamma$ | $df(\gamma)$ | $\pi(\gamma)$ | $\gamma$ | $df(\gamma)$ | $\pi(\gamma)$ |
|---|---|---|---|---|---|
| $\varphi \wedge \psi$ | $\mathbf{L}_{\varphi\wedge\psi} \leftrightarrow \mathbf{L}_\varphi \wedge \mathbf{L}_\psi$ | $\mathbf{L}_{\varphi\wedge\psi} \rightarrow \mathbf{L}_\varphi$<br>$\mathbf{L}_{\varphi\wedge\psi} \rightarrow \mathbf{L}_\psi$<br>$\mathbf{L}_\varphi \wedge \mathbf{L}_\psi \rightarrow \mathbf{L}_{\varphi\wedge\psi}$ | $\neg\varphi$ | $\mathbf{L}_{\neg\varphi} \leftrightarrow \neg\mathbf{L}_\varphi$ | $\neg\mathbf{L}_\varphi \rightarrow \mathbf{L}_{\neg\varphi}$<br>$\mathbf{L}_{\neg\varphi} \rightarrow \neg\mathbf{L}_\varphi$ |
| $\varphi \vee \psi$ | $\mathbf{L}_{\varphi\vee\psi} \leftrightarrow \mathbf{L}_\varphi \vee \mathbf{L}_\psi$ | $\mathbf{L}_\varphi \rightarrow \mathbf{L}_{\varphi\vee\psi}$<br>$\mathbf{L}_\psi \rightarrow \mathbf{L}_{\varphi\vee\psi}$<br>$\mathbf{L}_{\varphi\vee\psi} \rightarrow \mathbf{L}_\varphi \vee \mathbf{L}_\psi$ | $\varphi \rightarrow \psi$ | $\mathbf{L}_{\varphi\rightarrow\psi} \leftrightarrow (\mathbf{L}_\varphi \rightarrow \mathbf{L}_\psi)$ | $\mathbf{L}_{\varphi\rightarrow\psi} \wedge \mathbf{L}_\varphi \rightarrow \mathbf{L}_\psi$<br>$\neg\mathbf{L}_\varphi \rightarrow \mathbf{L}_{\varphi\rightarrow\psi}$<br>$\mathbf{L}_\psi \rightarrow \mathbf{L}_{\varphi\rightarrow\psi}$<br>$\mathbf{L}_\varphi \vee \neg\mathbf{L}_\psi \vee \mathbf{L}_{\varphi\rightarrow\psi}$ |

**Fig. 2.** Transformation $\pi(\gamma)$ generating a generalised disjunctive logic program.

Clearly, including an arbitrary theory $\Pi' \subseteq \mathcal{L}_V$ in Theorem 2 as follows:

$$\{I \mid I \models \Pi \cup \Pi'\} = \{J \cap V \mid J \models \sigma(\Pi) \cup \Pi'\}$$

and then taking the minimal models on both sides trivially preserves the equality. Therefore, the following is straightforward.

**Corollary 1.** *Translation $\sigma(\Pi)$ is strongly faithful.*

Modularity of $\sigma(\Pi)$ is quite obvious, and the polynomial complexity of its computation can also be easily deduced. However, $\sigma(\Pi)$ does not have the shape of a logic program: it contains double implications where the implication symbol may occur nested. Fortunately, we can unfold these double implications in linear time without changing the signature $V_{\mathbf{L}}$ (in fact, we can use transformations in Section 3 for this purpose). For each definition $df(\gamma)$, we define the strongly equivalent set (understood as the conjunction) of logic program rules $\pi(\gamma)$ as shown in Figure 2. The fact $df(\gamma) \equiv_s \pi(\gamma)$ can be easily checked in here-and-there. The main difference with respect to [16] is of course the treatment of the implication. In fact, the set of rules $\pi(\varphi \rightarrow \psi)$ was already used in [15] to unfold nested implications in an arbitrary theory, with the exception that, in that work, labelling was exclusively limited to implications. The explanation for this set of rules can be easily outlined using transformations in Section 3. For the left to right direction in $df(\varphi \rightarrow \psi)$, that is, the implication $\mathbf{L}_{\varphi\rightarrow\psi} \rightarrow (\mathbf{L}_\varphi \rightarrow \mathbf{L}_\psi)$, we can apply (R5') to obtain the first rule shown in Figure 2 for $\pi(\varphi \rightarrow \psi)$. The remaining three rules are the direct application of (L5) (being $\alpha = \top$) for the right to left direction $(\mathbf{L}_\varphi \rightarrow \mathbf{L}_\psi) \rightarrow \mathbf{L}_{\varphi\rightarrow\psi}$.

The program $\pi(\Pi)$ is obtained by replacing in $\sigma(\Pi)$ each subformula definition $df(\varphi)$ by the corresponding set of rules $\pi(\varphi)$. As $\pi(\Pi)$ is strongly equivalent to $\sigma(\Pi)$ (under the same vocabulary) it preserves strong faithfulness with respect to $\Pi$. Furthermore, if we consider the complexity of the direct translation from $\Pi$ to $\pi(\Pi)$ we obtain the following result.

**Theorem 3.** *Translation $\pi(\Pi)$ is linear and its size can be bounded as follows:* $|vars(\pi(\Pi)| \leq |vars(\Pi)| + degree(\Pi), \quad degree(\pi(\Pi)) \leq |\Pi| + 12\ degree(\Pi).$

| $df(\neg\varphi)$ | $\pi'(\neg\varphi)$ |
|---|---|
| $\mathbf{L}_{\neg\varphi} \leftrightarrow \neg\mathbf{L}_{\varphi}$ | $\neg\mathbf{L}_{\varphi} \rightarrow \mathbf{L}_{\neg\varphi}$ |
| | $\mathbf{L}_{\neg\varphi} \wedge \mathbf{L}_{\varphi} \rightarrow \bot$ |

| $df(\varphi \rightarrow \psi)$ | $\pi'(\varphi \rightarrow \psi)$ |
|---|---|
| $\mathbf{L}_{\varphi\rightarrow\psi} \leftrightarrow (\mathbf{L}_{\varphi} \rightarrow \mathbf{L}_{\psi})$ | $\mathbf{L}_{\varphi\rightarrow\psi} \wedge \mathbf{L}_{\varphi} \rightarrow \mathbf{L}_{\psi}$ |
| | $\neg\mathbf{L}_{\varphi} \rightarrow \mathbf{L}_{\varphi\rightarrow\psi}$ |
| | $\mathbf{L}_{\psi} \rightarrow \mathbf{L}_{\varphi\rightarrow\psi}$ |
| | $\mathbf{L}_{\varphi} \vee \mathbf{L}_{\neg\psi} \vee \mathbf{L}_{\varphi\rightarrow\psi}$ |
| | $\neg\mathbf{L}_{\psi} \rightarrow \mathbf{L}_{\neg\psi}$ |
| | $\mathbf{L}_{\neg\psi} \wedge \mathbf{L}_{\psi} \rightarrow \bot$ |

**Fig. 3.** Transformation $\pi'(\gamma)$ generating a disjunctive logic program.

*Proof.* As we explained before, we use a label in $\pi(\Pi)$ per each non-constant subformula in $\Pi$ (including atoms). We can count the subformulas as the number of connectives[8], $\leq degree(\Pi)$, plus the number of atoms in $\Pi$, $|vars(\Pi)|$.

As for the second bound, note that $\pi(\Pi)$ consists of two subtheories. The first one contains a label $\mathbf{L}_{\varphi}$ per each formula $\varphi$ in $\Pi$. The amount $|\Pi|$ counts implicit conjunction used to connect each label to the rest of the theory. The second part of $\pi(\Pi)$ collects a set of rules $\pi(\gamma)$ per each subformula $\gamma$ of $\Pi$. The worst case, corresponding to the translation of implication, uses eight connectives plus four implicit conjunctions to connect the four rules to the rest of the theory. □

A possible objection to $\pi(\Pi)$ is that it makes use of negation in the rule heads, something not usual in the current tools for answer sets programming. Although there exists a general translation [10] for removing negation in the head, it is possible to use a slight modification of $\pi(\Pi)$ to yield a disjunctive program in a direct way. To this end, we define a new $\pi'(\gamma)$ for each subformula $\gamma$ of $T$ that coincides with $\pi(\gamma)$ except for implication and negation, which are treated as shown in Figure 3. As we can see, the use of negation in the head for $\pi(\neg\varphi)$ can be easily removed by just using a constraint $\mathbf{L}_{\neg\varphi} \wedge \mathbf{L}_{\varphi} \rightarrow \bot$. In the case of implication, we have replaced negated label $\neg\mathbf{L}_{\psi}$ by the labeled negation $\mathbf{L}_{\neg\psi}$ in the resulting disjunctive rule.The only problem with this technique is that $\neg\psi$ need not occur as subformula in the original theory $\Pi$. Therefore, we must include the definition for the newly introduced label $\mathbf{L}_{\neg\psi}$, that leads to the last two additional rules, and we need a larger signature, $V_{\mathbf{L}} = \{\mathbf{L}_{\varphi}, \mathbf{L}_{\neg\varphi} \mid \varphi \in \mathcal{L}_V \setminus \{\bot, \top\}\}$. If we consider now the translation $\pi'(\Pi)$ it is not difficult to see that modularity and strong faithfulness are still preserved, while its computation can be shown to be polynomial, although using slightly greater bounds $|vars(\pi(\Pi))| \leq 2\,|vars(\Pi)|+2\,degree(\Pi)$ and $degree(\pi(\Pi)) \leq |\Pi| + 22\ degree(\Pi)$.

## 5 Concluding remarks

Equilibrium logic provides a natural generalisation of answer set semantics to propositional logic. It allows one to handle embedded implications, in particular

[8] Note that $degree(\Pi)$ also counts the implicit conjunction of all formulas in $T$

to write programs containing rules with conditional heads or bodies. As [5] has recently shown, such rules can be used to represent aggregates in ASP under the semantics of [4].

In this paper we have explored different ways in which arbitrary propositional theories in equilibrium logic can be reduced to logic programs and thus implemented in an ASP solver. First, we presented rules for transforming any theory into a strongly equivalent program in the same vocabulary. Second, we showed that there is no polynomial algorithm for such a transformation. Third, we showed that if we allow new atoms or 'labels' to be added to the language of a theory, it can be reduced to a logic program in polynomial time. The program is still in a strong sense equivalent to the original theory and the theory's equilibrium models or answer sets can be be retrieved from the program.

We have extended several previous works in the following way. In [12] several of the reduction rules of §3 were already proposed in order to show that nested programs can be reduced to generalised disjunctive programs. In [1] it is shown for the first time that arbitrary propositional theories have equivalent programs in the same vocabulary; but complexity issues are not discussed. In [16] a reduction is proposed for nested programs into disjunctive programs containing new atoms. The reduction is shown to be polynomial and has been implemented as a front-end to `DLV` called `nlp`. Following the tradition of structure-preserving normal form translations for nonclassical logics, as illustrated in [14], the reduction procedure of [16] uses the idea of adding labels as described here. Our main contribution has been to simplify the translation and the proof of faithfulness as well as extend it to the full propositional language including embedded implications. The formulas we have added for eliminating implications were previously mentioned in [15]. However that work does not provide any details on the complexity bounds of the resulting translation, nor does it describe the labelling method in full detail.

Many issues remain open for future study. Concerning the transformation described in §3, for example, several questions of efficiency remain open. In particular the logic program obtained by this method is not necessarily 'optimal' for computational purposes. In the future we hope to study additional transformations that lead to a minimal set of program rules. Concerning the reduction procedure of §4, since, as mentioned, it extends a system `nlp` [22] already available for nested programs, it should be relatively straightforward to implement and test. One area for investigation here is to see if such a system might provide a prototype for implementing aggregates in ASP, an issue that is currently under study elsewhere.

Another area of research concerns the language of equilibrium logic with an additional, strong negation operator for expressing explicit falsity. The relation between intermediate logics and their least strong negation extensions has been well studied in the literature. From this body of work one can deduce that most of the results of this paper carry over intact to the case of strong negation. However, the reductions are not as simple as the methods currently used for eliminating strong negation in ASP. In particular, for the polynomial translations

of propositional theories additional defining formulas are needed. We postpone the details for a future work.

## References


1. P. Cabalar & P. Ferraris. Propositional Theories are Strongly Equivalent to Logic Programs. Unpublished draft, 2005, available at `http://www.dc.fi.udc.es/~cabalar/pt2lp.pdf`.
2. D. van Dalen. Intuitionistic logic. In *Handbook of Philosophical Logic, Volume III: Alternatives in Classical Logic*, Kluwer, Dordrecht, 1986.
3. P. M. Dung. Declarative Semantics of Hypothetical Logic Programing with Negation as Failure. in *Proceedings ELP 92*, 1992, 99. 45-58.
4. W. Faber, N. Leone & G. Pfeifer. Recursive Aggregates in Disjunctive Logic Programs: semantics and Complexity. in J.J. Alferes & J. Leite (eds), *Logics In Artificial Intelligence. Proceedings JELIA'04*, Springer LNAI 3229, 2004, pp. 200-212.
5. P. Ferraris. Answer Sets for Propositional Theories. In *Eighth Intl. Conf. on Logic Programming and Nonmonotonic Reasoning* (LPNMR'05), 2005 (to appear).
6. P. Ferraris & V. Lifschitz. Weight Constraints as Nested Expressions. *Theory and Practice of Logic Programming* (to appear).
7. M. Gelfond & V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
8. L. Giordano & N. Olivetti. Combining Negation-as-Failure and Embedded Implications in Logic Programs. *Journal of Logic Programming* 36 (1998), 91-147.
9. R. Hähnle. Complexity of Many-Valued Logics. In Proc. 31st International Symposium on Multiple-Valued Logics, IEEE CS Press, Los Alamitos (2001) 137–146.
10. T. Janhunen, I. Niemelä, P. Simons & J.-H. You. Unfolding Partiality and Disjunctions in Stable Model Semantics. In A. G. Cohn, F. Giunchiglia & B. Selman (eds), *Principles of Knowledge Representation and Reasoning (KR-00)*, pages 411424. Morgan Kaufmann, 2000.
11. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri & F. Scarcello. The dlv System for Knowledge Representation and Reasoning. CoRR: cs.AI/0211004, September 2003.
12. V. Lifschitz, L. R. Tang & H. Turner. Nested Expressions in Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 25 (1999), 369–389.
13. V. Lifschitz, D. Pearce & A. Valverde. Strongly Equivalent Logic Programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.
14. G. Mints. Resolution Strategies for the Intuitionistic Logic. In B. Mayoh, E. Tyugu & J. Penjaam (eds), *Constraint Programming* NATO ASI Series, Springer, 1994, pp.282-304.
15. M. Osorio, J. A. Navarro Pérez & J. Arrazola Safe Beliefs for Propositional Theories *Ann. Pure & Applied Logic* (in press).
16. D. Pearce, V. Sarsakov, T. Schaub, H. Tompits & S. Woltran. Polynomial Translations of Nested Logic Programs into Disjunctive Logic Programs. In Proc. of the 19th Int. Conf. on Logic Programming (ICLP'02), 405–420, 2002.
17. D. Pearce. A New Logical Characterisation of Stable Models and Answer Sets. In *Non-Monotonic Extensions of Logic Programming, NMELP 96*, LNCS 1216, pages 57–70. Springer, 1997.
18. D. Pearce. From Here to There: stable negation in logic programming. In D. Gabbay & H. Wansing, eds., *What is Negation?*, pp. 161–181. Kluwer Academic Pub., 1999.

19. D. Pearce, I. P. de Guzmán & A. Valverde. A Tableau Calculus for Equilibrium Entailment. In *Automated Reasoning with Analytic Tableaux and Related Methods, TABLEAUX 2000*, LNAI 1847, pages 352–367. Springer, 2000.
20. D. Pearce, I.P. de Guzmán & A. Valverde. Computing Equilibrium Models using Signed Formulas. In *Proc. of CL2000*, LNCS 1861, pp. 688–703. Springer, 2000.
21. D. Pearce & A. Valverde. Synonymous Theories in Answer Set Programming and Equilibrium Logic. in R. López de Mántaras & L. Saitta (eds), *Proceedings ECAI 04*, IOS Press, 2004, pp. 388-392.
22. V. Sarsakov, T. Schaub, H. Tompits & S. Woltran. nlp: A Compiler for Nested Logic Programming. in *Proceedings of LPNMR 2004*, pp. 361-364. Springer LNAI 2923, 2004.
23. D. Seipel. Using Clausal Deductive Databases for Defining Semantics in Disjunctive Deductive Databases. *Annals of Mathematics and Artificial Intelligence* 33 (2001), pp. 347-378.
24. P. Simons, I. Niemelä & T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.