From Modular Horn Programs to Flat Ones: a Formal Proof for the Propositional Case. ⋆

M. Navarro

Dpto de L.S.I., Facultad de Informática, Paseo Manuel de Lardizabal, 1, Apdo 649, 20080-San Sebastián, SPAIN. Tel: +34 (9)43 015072, Fax: +34 (9)43 219306, e-mail: marisa@si.ehu.es.

Abstract. $Horn^{\supset}$ is a logic programming language, defined on the underlying logic $\mathcal{FO}^{\supset}$ (an extension of $\mathcal{FO}$ with intuitionistic implication), which permits a form of inner modularity in terms of *open blocks of local clauses* [8, 7, 3, 1, 9]. A translation from these logic programs with embedded implications to Horn clause programs is an interesting approach not only for giving logical foundation to this kind of extended logic programs but also for making it useful for implementation issues. In this paper we present a suitable translation algorithm from $Horn^{\supset}$ programs to Horn clause programs, in the propositional setting, and we formally prove that this translation preserves the original operational semantics in $Horn^{\supset}$ by means of SLD-resolution on the translation result. We also give an implementation of the translation algorithm written in Haskell and we show execution examples.

1 Introduction

Many approaches are concerned with extending Horn clauses with some features for program structuring that can be seen as a form of modularity in logic programming (see for instance [3] for a survey). Some of them consider the extension of Horn clauses with implication goals of the form $D \supset G$, called *blocks*, where D can be seen as a *set of local clauses* for proving the goal G. This approach yields to different extensions of Horn clause programming depending on the given semantics to such blocks.

A first basic distinction is between *closed blocks*: G can be proved only using local clauses from D, and *open blocks*: G can be proved using D and also the external environment. In general, by dealing with open blocks, a module can extend the definition of a predicate already defined in the environment. That is, different definitions of the same predicate could have to be considered, depending on the collection of modules corresponding to different goals. Therefore, open blocks require *scope rules* to fix the interplay between the predicate definitions inside a module D and those in the environment. There are mainly two scoped rules, named *static* and *dynamic*, allowing this kind of extension of predicate definitions.

In the dynamic case the set of modules used for proving a goal G can only be determined from the sequence of goals generated until G whereas in the static case this set of modules can be determined (for each goal) statically from the program block structure. Different proposals of logic programming languages for open blocks with dynamic scope have been presented and studied in several papers (e.g.[4–6, 12–15]). In [12] Miller proves that the proof-theoretic semantics for its dynamic scope programming language is based on intuitionistic logic. The static scope approach was introduced

⋆ This work has been partially supported by CICYT-project TIC2001-2476-C03-03

in [8] and formally studied in [8, 7, 1, 9]. In [3, 7] both different approaches are compared. The formal study developed in [1] on the logical foundations of the static scope programming language introduces the complete logic $\mathcal{FO}^{\supset}$, which is an extension of $\mathcal{FO}$ with intuitionistic implication (see [9] for details of this logic) and it gives a new characterization of the semantics for the static scope programming language $Horn^{\supset}$ as a *strong logic programming language* on the underlying logic $\mathcal{FO}^{\supset}$. The notion of strong logic programming language is formalized in [10, 11] and it basically means to set which subclasses of formulas correspond to the classes of *programs* and *queries or goals* and to prove that the underlying logic, for these subclasses, must satisfy three desirable properties: mathematical semantics, goal completeness and operational semantics.

A different approach to give logical foundations to this kind of logic programming languages (or in general to Horn clause extensions) is the transformational one that consists of translating programs into the language of some well-known logic. For instance, in [7] a transformational logical foundation to both static and dynamic languages (those defined in [8] and [12] respectively) is given, translating them to $S4$-modal logic. Inside this setting, a translation method from a modal programming language (with modalities and embedded implication) to Horn clause programs is introduced in [2]. This method consists of two steps: the first one eliminates embedded implications by introducing new modalities, and the second one eliminates modalities (by adding an argument to all predicates). The transformational approach is also taken in [16] where (in some more restricted sense) logic programs with embedded implications are translated to Horn clause programs by introducing new predicates. This is a direct mapping because the definition of a predicate in a new module overrides its definition in previous modules. However, as it is pointed in [3, 16], when predicate extension is allowed, the translation of each predicate definition (inside a module) raises different predicate definitions, each one depending on a collection of modules to be used. In the dynamic case such collection can only be determined in run-time, but in the case of the static programming language $Horn^{\supset}$ there is a lexical way to determine such collection of modules (for each goal) making this approach useful for implementation issues. However, this translation is not direct because of the multiple transformation of each original predicate.

Our aim is to study and implement a translation method for the static scope programming language $Horn^{\supset}$ into Horn clause language. We propose a one-step method where embedded implications are eliminated by introducing new predicates, so both source and target languages have the same underlying logic $\mathcal{FO}^{\supset}$. This translation must preserve the original operational semantics in $Horn^{\supset}$ by means of SLD-resolution on the translation result. Therefore a proof of the correctness of the algorithm is necessary.

In this paper, as starting point in our study, we restrict it to the propositional case and we present a suitable translation algorithm from $Horn^{\supset}$ programs to Horn clause programs, a detailed proof of soundness and completeness of the translation, and a concrete implementation of the algorithm. Some clues on how to proceed in the first order case are also given in the last section. In concrete, the paper is organized as follows: In Section 2 the static programming language $Horn^{\supset}$ is presented by giving its syntax and (operational) semantics (by means of the $\vdash_{\supset}$ deduction). Also some examples of goal deductions from $Horn^{\supset}$ programs are shown. In Section 3 we introduce the (abstract) translation algorithm by means of two mutually recursive functions for translating program clauses and goals. Because of the operational semantics of the language, this translation has to be extended to sequences of $Horn^{\supset}$ programs. In Section 4 we show within two examples how the translation proceeds. Section 5 is the core of the paper where the soundness and completeness results are proved. Concretely, the translation of a (program, goal) pair in $Horn^{\supset}$ to a new (program, goal) pair in $Horn$ preserves the original operational semantics of the extended language, which is now simulated by SLD-resolution. In Section 6 a (concrete) translation algorithm written in Haskell is shown together an example of execution. We conclude, in Section 7, by summarizing our results and by showing further work we plan to do.

2 Preliminaries

In this section we introduce the static scope programming language $Horn^{\supset}$ by showing its syntax and operational semantics. The syntax is an extension of the Horn clause language, by adding the intuitionistic implication $\supset$ in goals and clause bodies. A $Horn^{\supset}$ program is a finite set of closed D-clauses. The program clauses, named D-clauses, and the goals, named G-clauses, are recursively defined as follows (where A stands for an atomic formula):

$$G := A \mid G_1 \wedge G_2 \mid D \supset G \mid \exists x G \qquad\qquad D := A \mid G \rightarrow A \mid D_1 \wedge D_2 \mid \forall x D$$

The main difference with respect to Horn clauses is the use of a local clauses set D in goals of the kind $D \supset G$ (and therefore also in program clause bodies). Moreover, in the first order case, the extended language needs to use quantifiers explicitly. In the propositional case, the previous definitions are simplified to:

$$G := A \mid G_1 \wedge G_2 \mid D \supset G \qquad\qquad D := A \mid G \rightarrow A \mid D_1 \wedge D_2$$

The way of proving a goal $D \supset G$ from a program P is to "add" D to P, named $P \mid D$, and to prove G from $P \mid D$. Since G can itself be (or contain) a goal $D' \supset G'$ the length of the sequence $P \mid D \mid D' \ldots$ can be arbitrarily large.

Following [8, 1], we use a simple definition of the operational semantics of $Horn^{\supset}$, given by a non-deterministic set of rules which define when a goal G is operationally derivable from a program sequence $\Delta = D_0|D_1|\ldots|D_n$, in symbols $\Delta \vdash_{\supset} G$. For the sake of simplicity we only present here the set of rules simplified to the propositional case. These rules are given in Figure 1 where $A \in \Delta$ means that A belongs to some of the programs in the sequence Δ.

(1) $\Delta \vdash_{\supset} A$ if A is atomic and $A \in \Delta$

(2) $\dfrac{D_0|\ldots|D_i \vdash_{\supset} G}{D_0|\ldots|D_i|\ldots|D_n \vdash_{\supset} A}$ if $G \rightarrow A \in D_i$ and $0 \leq i \leq n$

(3) $\dfrac{\Delta \vdash_{\supset} G_1 \quad \Delta \vdash_{\supset} G_2}{\Delta \vdash_{\supset} G_1 \wedge G_2}$

(4) $\dfrac{\Delta|\{D\} \vdash_{\supset} G}{\Delta \vdash_{\supset} D \supset G}$

Fig. 1. Operational Semantics for Propositional $Horn^{\supset}$.

A sequence $\Delta = D_0|D_1|\dots|D_n$ can be viewed as a stack with top element D_n. Therefore rule (2) says that if the clause $G \rightarrow A$ selected for proving A from Δ belongs to D_i then the body G must be proved from clauses only in $D_0|\dots|D_i$ (that is, $D_{i+1}, \dots D_n$ cannot be used). On the other hand, the stack is enlarged by means of rule (4).
The following example illustrates the operational behaviour of this language.

Example 1. Let the program with two clauses $P = \{((b \rightarrow c) \supset c) \rightarrow a,\ b\}$ and let the goal $G1 = a$. A proof of $P \vdash_{\supset} G1$ is given by the following steps (applying rules in Figure 1):

$P \vdash_{\supset} a$	holds by Rule (2) for the clause $((b \rightarrow c) \supset c) \rightarrow a$ in P if
$P \vdash_{\supset} (b \rightarrow c) \supset c$	which holds by Rule (4) if
$P \mid \{b \rightarrow c\} \vdash_{\supset} c$	which holds by Rule (2) for the clause $b \rightarrow c$ if
$P \mid \{b \rightarrow c\} \vdash_{\supset} b$	which holds by Rule (1) since $b \in P \mid \{b \rightarrow c\}$

That is, in this case, for proving $P \vdash_{\supset} a$ it is necessary to prove c from the "extended program" $P \mid \{b \rightarrow c\}$ which finally holds.
Now let the program with an unique clause $Q = \{((b \rightarrow c) \supset c) \rightarrow a\}$ and let the goal $G2 = b \supset a$. The only way to look for a proof of $Q \vdash_{\supset} G2$ is by giving the following steps:

$Q \vdash_{\supset} b \supset a$	by Rule (4) if
$Q \mid \{b\} \vdash_{\supset} a$	by Rule (2) for the clause in Q if
$Q \vdash_{\supset} (b \rightarrow c) \supset c$	by Rule (4) if
$Q \mid \{b \rightarrow c\} \vdash_{\supset} c$	by Rule (2) for the clause $b \rightarrow c$ if
$Q \mid \{b \rightarrow c\} \vdash_{\supset} b$	which does not hold

Therefore $Q \not\vdash_{\supset} G2$. It must be noted that at the second step the "extension" $\{b\}$ disappears since the chosen clause belongs to Q. ■

This example also shows the "static scope rule" meaning: the set of clauses which can be used to solve a goal depends on the program block structure. Whereas $G1 = a$ can be proved from the program P because b was defined in P, in the case of $G2 = b \supset a$ and the program Q the "external" definition of b in the sequence $Q|\{b\}$ is not permitted for proving the body of a clause in Q. This is a major difference with the "dynamic scope rule" used in [12].

3 The Abstract Translation Algorithm

In this section we introduce the translation algorithm, where "renamings" for locally defined predicates are abstracted to be "new". Given a renaming σ for a set Σ of predicates, $ext(\sigma)$ will denote the set of D-clauses $\{A \rightarrow A\sigma \ / \ A \in \Sigma\}$
The algorithm consists of defining the functions $tradP$ for D-clauses and $tradG$ for G-clauses as follows:

$tradP(A) = \{A\}$
$tradP(G \rightarrow A) = \{G' \rightarrow A\} \cup P'$ where $(G', P') = tradG(G)$
$tradP(D_1 \wedge D_2) = tradP(D_1) \cup tradP(D_2)$
$tradG(A) = (A, \emptyset)$
$tradG(G_1 \wedge G_2) = (G'_1 \wedge G'_2, P_1 \cup P_2)$ where
$(G'_1, P_1) = tradG(G_1)$ and $(G'_2, P_2) = tradG(G_2)$

$tradG(D \supset G) = (G', P_1 \cup P_2 \cup P_3)$ where σ is a new renaming for predicates defined in D, $(G', P_1) = tradG(G\sigma)$, $P_2 = tradP(D\sigma)$ and $P_3 = ext(\sigma)$

Note 1. We shall use $1tradG(G)$ and $2tradG(G)$ for the first and second components (respectively) of $tradG(G)$

Since the language $Horn^{\supset}$ works with sequences of programs, it is necessary to extend $tradP$ to a new function $trad\Delta$ which translates a sequence of $Horn^{\supset}$ programs to a single $Horn$ program.

Definition 2. Given a sequence of $Horn^{\supset}$ programs $\Delta_n = D_0|D_1|\ldots|D_n$, the set of Horn clauses $trad\Delta(\Delta_n)$ is recursively defined in the following way:

- For $n = 0$ (i.e. $\Delta_n = D_0$): $trad\Delta(\Delta_n) = tradP(D_0)$
- For $n = i + 1$ (i.e. $\Delta_n = \Delta_i|D_{i+1}$): $trad\Delta(\Delta_n) = trad\Delta(\Delta_i) \cup tradP(D_{i+1}\sigma_1 \ldots \sigma_{i+1}) \cup ext(\sigma_{i+1})$ where σ_{i+1} is a new renaming for predicates defined in $D_{i+1}\sigma_1 \ldots \sigma_i$

Again in the previous definition each renaming σ_i is defined as "new" in an abstract way. To concrete this fact, the following remark explains how the algorithm has to proceed to assure that the predicates are extended in a correct manner.

Remark 3. Let $\Delta_n = D_0|D_1|\ldots|D_n$ be a sequence of $Horn^{\supset}$ programs and let Σ be the set of predicate names in Δ_n. The algorithm proceeds by translating first D_0 and obtaining $tradP(D_0)$ whose predicates belong to the signature $\Sigma \cup \Sigma'_0$, with Σ'_0 containing the new added predicates for the internal blocks of D_0 such that $\Sigma \cap \Sigma'_0 = \emptyset$. Now it looks for the predicates defined in D_1 (that is, those in the head of clauses in D_1) and it selects a "new" renaming σ_1 for them. This means that σ_1 can be viewed as a function $\sigma_1 : DefPred(D_1) \rightarrow \Sigma_{\sigma_1}$, such that $\Sigma_{\sigma_1} \cap (\Sigma \cup \Sigma'_0) = \emptyset$. Then the algorithm proceeds by translating $D_1\sigma_1$ and obtaining $tradP(D_1\sigma_1)$ whose predicates belong to the signature $(\Sigma \cup \Sigma_{\sigma_1}) \cup \Sigma'_1$. Σ'_1 contains the new added predicates for the internal blocks of $D_1\sigma_1$ so that, as before, $(\Sigma \cup \Sigma_{\sigma_1}) \cap \Sigma'_1 = \emptyset$. But now it must also verify that $\Sigma'_1 \cap \Sigma'_0 = \emptyset$. With the rest of the sequence the algorithm follows in the same way. ■

From this fact the following particular observation can be deduced:

Remark 4. If $G \rightarrow A$ is a Horn clause in $tradP(D_i\sigma_1 \ldots \sigma_i) \cup ext(\sigma_i)$ then G can neither include predicate names defined in $tradP(D_j\sigma_1 \ldots \sigma_j)$ nor those defined in $ext(\sigma_j)$ for $j > i$.
Since $trad\Delta(D_0|D_1|\ldots|D_n)$=$tradP(D_0) \cup \ldots \cup tradP(D_i\sigma_1 \ldots \sigma_i) \cup ext(\sigma_i) \cup \ldots \cup tradP(D_n\sigma_1 \ldots \sigma_n) \cup ext(\sigma_n)$, if such G is deduced (by SLD-resolution) from the set of Horn clauses $trad\Delta(D_0|D_1|\ldots|D_n)$ it means that G is indeed deduced from $trad\Delta(D_0|D_1|\ldots|D_i)$. ■

4 Examples

In this section we shall show how a concrete implementation (in the sense of selecting some renaming) of the translation algorithm proceeds within two examples. In these examples, the translated clauses will also be written as it is usual in Logic Programming (i.e., a clause $G \rightarrow A$ will be denoted $A : -G.$) for the sake of readability.

Example 5. Let $\Delta_2 = D_0|D_1|D_2$ be a sequence of $Horn^{\supset}$ programs where $D_0 = s \wedge (r \rightarrow p)$, $D_1 = p \wedge (s \rightarrow q)$ and $D_2 = (p \rightarrow t) \wedge (q \rightarrow p)$.
The translation of Δ_2 can follow these steps:

1. $tradP(D_0) = \{s., \ p : -r.\}$
2. Since $DefPred(D_1) = \{p, \ q\}$ the first renaming is $\sigma_1 = \{p1/p, \ q1/q\}$. Then $tradP(D_1\sigma_1) = \{p1., \ q1 : -s.\}$ and $ext(\sigma_1) = \{p1 : -p., \ q1 : -q.\}$.
3. Now we consider $D_2\sigma_1 = (p1 \rightarrow t) \wedge (q1 \rightarrow p1)$. Since $DefPred(D_2\sigma_1) = \{t, \ p1\}$ the second renaming is $\sigma_2 = \{p2/p1, \ t1/t\}$. Then $tradP(D_2\sigma_1\sigma_2) = \{t1 : -p2., \ p2 : -q1.\}$ and $ext(\sigma_2) = \{t1 : -t., \ p2 : -p1\}$.

Therefore the set of Horn clauses obtained by the translation of Δ_2 is $trad\Delta(\Delta_2) = tradP(D_0) \cup tradP(D_1\sigma_1) \cup ext(\sigma_1) \cup tradP(D_2\sigma_1\sigma_2) \cup ext(\sigma_2) = \{s., \ p : -r., \ p1., \ q1 : -s., \ p1 : -p., \ q1 : -q., \ t1 : -p2., \ p2 : -q1., \ t1 : -t., \ p2 : -p1\}$.

Example 6. Let $\Delta_1 = D_0|D_1$ be a sequence of $Horn^{\supset}$ programs where $D_0 = r \wedge ((D_0' \supset q) \rightarrow p) \wedge ((D_0'' \supset q) \rightarrow h)$, with $D_0' = r \rightarrow q$ and $D_0'' = s \rightarrow q$ and where $D_1 = r \rightarrow q$.
The translation of Δ_1 follows these steps:

1. The translation algorithm starts by translating D_0 as before, but in this case D_0 contains two sets of local clauses D_0' and D_0'' which are independent of each other. By definition of $tradP$: $tradP(D_0) = tradP(r) \cup tradP((D_0' \supset q) \rightarrow p) \cup tradP((D_0'' \supset q) \rightarrow h)$.
Now by taking the renaming $\sigma_0' = \{q_0'/q\}$ for the local clauses in D_0', it is obtained that $tradP((D_0' \supset q) \rightarrow p) = \{p : -q_0'., \ q_0' : -r., \ q_0' : -q.\}$. Similarly, by taking σ_0'' $= \{q_0''/q\}$ as (new) renaming for the local clauses in D_0'', $tradP((D_0'' \supset q) \rightarrow h) = \{h : -q_0''., \ q_0'' : -s., \ q_0'' : -q.\}$.
That is, $tradP(D_0) = \{r., \ p : -q_0'., \ q_0' : -r., \ q_0' : -q., \ h : -q_0''., \ q_0'' : -s., \ q_0'' : -q.\}$.
2. Since $DefPred(D_1) = \{q\}$, the renaming σ_1 can be $\{q1/q\}$, which must be disjoint from the previous local renamings σ_0' and σ_0''. Then $tradP(D_1\sigma_1) = \{q1 : -r.\}$ and $ext(\sigma_1) = \{q1 : -q.\}$.

Therefore the set of Horn clauses obtained by the translation of Δ_1 is $trad\Delta(\Delta_1) = tradP(D_0) \cup tradP(D_1\sigma_1) \cup ext(\sigma_1) = \{r., \ p : -q_0'., \ q_0' : -r., \ q_0' : -q., \ h : -q_0''., \ q_0'' : -s., \ q_0'' : -q., \ q1 : -r., \ q1 : -q.\}$.

5 Soundness and Completeness

The aim of this section is to prove that the given translation algorithm is suitable for simulating the $\vdash_{\supset}$ deduction by means of SLD-deduction. More concretely, given a $Horn^{\supset}$ program P and given an atom A, $P \vdash_{\supset} A$ if and only if $tradP(P) \vdash_{SLD} A$.
To prove this result it is necessary to prove a more general result which is established in the following lemmas 7 and 10.

Lemma 7. *For every sequence of $Horn^{\supset}$ programs $\Delta_n = D_0|D_1|\ldots|D_n$ with $n \geq 0$, and every goal G, it holds:*

$$\Delta_n \vdash_{\supset} G \Longrightarrow trad\Delta(\Delta_n) \cup 2tradG(G\sigma_1 \ldots \sigma_n) \vdash_{SLD} 1tradG(G\sigma_1 \ldots \sigma_n)$$

Proof. We proceed by induction on the number of steps (m) in the deduction proof of $\Delta_n \vdash_{\supset} G$.

Case m=1 In this case, G is necessarily an atom B belonging to D_i for some $i \in \{0 \dots n\}$. Therefore $1tradG(G\sigma_1 \dots \sigma_n) = B\sigma_1 \dots \sigma_n$ and $2tradG(G\sigma_1 \dots \sigma_n) = \emptyset$. Then we have to prove that $trad\Delta(\Delta_n) \vdash_{SLD} B\sigma_1 \dots \sigma_n$.
Since $B \in D_i$ it holds $B\sigma_1 \dots \sigma_i \in tradP(D_i\sigma_1 \dots \sigma_i)$ and then $trad\Delta(\Delta_n) \vdash_{SLD} B\sigma_1 \dots \sigma_i$ in one step. But for each $j \in \{i+1 \dots n\}$, either the program clause $B\sigma_1 \dots \sigma_{j-1} \rightarrow B\sigma_1 \dots \sigma_j$ belongs to $ext(\sigma_j)$ or $B\sigma_1 \dots \sigma_{j-1}$ is the same than $B\sigma_1 \dots \sigma_j$. Therefore $trad\Delta(\Delta_n) \vdash_{SLD} B\sigma_1 \dots \sigma_n$ by applying several deduction steps in $\vdash_{SLD}$.

Induction hypothesis The lemma holds whenever the number of steps in the deduction proof of $\Delta_n \vdash_{\supset} G$ is less or equal than m, for every Δ_n and G.

Case m+1 It is supposed that $\Delta_n \vdash_{\supset} G$ in m+1 steps with $m+1 > 1$. We proceed by case analysis on G:

- If $G = B$ then there exist $i \in \{0 \dots n\}$ and G_1 such that $G_1 \rightarrow B \in D_i$ and $\Delta_i \vdash_{\supset} G_1$ in a number of steps less or equal than m. Since G is an atom, again we have to prove that $trad\Delta(\Delta_n) \vdash_{SLD} B\sigma_1 \dots \sigma_n$. By induction hypothesis, $trad\Delta(\Delta_i) \cup 2tradG(G_1\sigma_1 \dots \sigma_i) \vdash_{SLD} 1tradG(G_1\sigma_1 \dots \sigma_i)$. Now $G_1 \rightarrow B \in D_i$ implies that the $Horn$ clause $1tradG(G_1\sigma_1 \dots \sigma_i) \rightarrow B\sigma_1 \dots \sigma_i$ and the $Horn$ clauses in the set $2tradG(G_1\sigma_1 \dots \sigma_i)$ belong to $tradP(D_i\sigma_1 \dots \sigma_i) \subseteq trad\Delta(\Delta_i)$.
Then $trad\Delta(\Delta_n) \vdash_{SLD} B\sigma_1 \dots \sigma_i$ and therefore (as in the case $m = 1$) also $trad\Delta(\Delta_n) \vdash_{SLD} B\sigma_1 \dots \sigma_n$
- If $G = G_1 \wedge G_2$ then both $\Delta_n \vdash_{\supset} G_1$ and $\Delta_n \vdash_{\supset} G_2$, each deduction in a number of steps $\leq m$. By induction hypothesis,
$trad\Delta(\Delta_n) \cup 2tradG(G_k\sigma_1 \dots \sigma_n) \vdash_{SLD} 1tradG(G_k\sigma_1 \dots \sigma_n)$ for $k = 1, 2$. Then it holds $trad\Delta(\Delta_n) \cup 2tradG(G_1\sigma_1 \dots \sigma_n) \cup 2tradG(G_2\sigma_1 \dots \sigma_n) \vdash_{SLD} 1tradG(G_1\sigma_1 \dots \sigma_n) \wedge 1tradG(G_2\sigma_1 \dots \sigma_n)$ which is equivalent to $trad\Delta(\Delta_n) \cup 2tradG((G_1 \wedge G_2)\sigma_1 \dots \sigma_n) \vdash_{SLD} 1tradG((G_1 \wedge G_2)\sigma_1 \dots \sigma_n)$
- If $G = D_{n+1} \supset G'$ then for $\Delta_{n+1} = \Delta_n | D_{n+1}$ it holds $\Delta_{n+1} \vdash_{\supset} G'$ in a number of steps less or equal than m and by induction hypothesis
$trad\Delta(\Delta_{n+1}) \cup 2tradG(G'\sigma_1 \dots \sigma_{n+1}) \vdash_{SLD} 1tradG(G'\sigma_1 \dots \sigma_{n+1})$.
But from definition of $tradG$ it holds $trad\Delta(\Delta_{n+1}) \cup 2tradG(G'\sigma_1 \dots \sigma_{n+1})$ $= trad\Delta(\Delta_n) \cup tradP(D_{n+1}\sigma_1 \dots \sigma_{n+1}) \cup ext(\sigma_{n+1}) \cup 2tradG(G'\sigma_1 \dots \sigma_{n+1})$ $= trad\Delta(\Delta_n) \cup 2tradG((D_{n+1} \supset G')\sigma_1 \dots \sigma_n)$ and $1tradG(G'\sigma_1 \dots \sigma_{n+1}) = 1tradG((D_{n+1} \supset G')\sigma_1 \dots \sigma_n)$. Then, we have obtained that $trad\Delta(\Delta_n) \cup 2tradG((D_{n+1} \supset G')\sigma_1 \dots \sigma_n) \vdash_{SLD} 1tradG((D_{n+1} \supset G')\sigma_1 \dots \sigma_n)$.
Case $m+1$ has finished and therefore the proof of this lemma. ■

In the particular case when the sequence of programs is a single program P and the goal is an atom A the following corollary is obtained:

Corollary 8. *For every $Horn^{\supset}$ program P and every atom A, it holds:*

$$P \vdash_{\supset} A \Longrightarrow tradP(P) \vdash_{SLD} A$$

This corollary is the "completeness" of the translation algorithm in this sense: Given a $Horn^{\supset}$ program P, every atom A that can be deduced from P (within $\vdash_{\supset}$) can also be deduced by SLD-resolution from the translated program $tradP(P)$.
With respect to more general goals, this result is directly generalised to conjunctions of atoms, but in the case of implication goals the translation has to be made on the program and the goal in the following way:

Corollary 9. *For every Horn$^{\supset}$ program P and every Horn$^{\supset}$ goal G, it holds:*

$$P \vdash_{\supset} G \Longrightarrow P' \vdash_{SLD} G' \text{ where } (G', P') \text{ is the result of } tradG(P \supset G)$$

Proof. It is obvious from $P \vdash_{\supset} G \Longrightarrow \emptyset \vdash_{\supset} (P \supset G)$ and Lemma 7. ∎

In order to prove the "soundness" of the translation algorithm, we need again a general result, namely, the converse of Lemma 7.

Lemma 10. *For every sequence of Horn$^{\supset}$ programs $\Delta_n = D_0|D_1|\ldots|D_n$ with $n \geq 0$, and every goal G, it holds:*

$$trad\Delta(\Delta_n) \cup 2tradG(G\sigma_1 \ldots \sigma_n) \vdash_{SLD} 1tradG(G\sigma_1 \ldots \sigma_n) \Longrightarrow \Delta_n \vdash_{\supset} G$$

Proof. We proceed by induction on the number m of steps in the $\vdash_{SLD}$ deduction proof and, inside each case of m, by structural induction on the goal.

Case m=1 In this case, $1tradG(G\sigma_1 \ldots \sigma_n)$ is necessarily an atom $B1$ belonging to $trad\Delta(\Delta_n) \cup 2tradG(G\sigma_1 \ldots \sigma_n)$. Since $B1$ is an atom then either
(1) $G = B$ for some atom B (with $B1 = B\sigma_1 \ldots \sigma_n$) or
(2) $G = D_{n+1} \supset G'$ for some goal G' (with $B1 = 1tradG(G'\sigma_1 \ldots \sigma_{n+1})$).
In the subcase (1) we have that $trad\Delta(\Delta_n) \vdash_{SLD} B\sigma_1 \ldots \sigma_n$ in one step; that is, $B\sigma_1 \ldots \sigma_n \in trad\Delta(\Delta_n)$. Then there exists $i \in \{0 \ldots n\}$ such that $B\sigma_1 \ldots \sigma_n = B\sigma_1 \ldots \sigma_i \in tradP(D_i\sigma_1 \ldots \sigma_i)$ and therefore $B \in D_i$. Then $\Delta_n \vdash_{\supset} B$.
In the subcase (2) it holds $B1 \in trad\Delta(\Delta_n) \cup 2tradG(G\sigma_1 \ldots \sigma_n) = trad\Delta(\Delta_n) \cup tradP(D_{n+1}\sigma_1 \ldots \sigma_{n+1}) \cup ext(\sigma_{n+1}) \cup 2tradG(G'\sigma_1 \ldots \sigma_{n+1}) = trad\Delta(\Delta_{n+1}) \cup 2tradG(G'\sigma_1 \ldots \sigma_{n+1})$, for $\Delta_{n+1} = \Delta_n|D_{n+1}$. Then we have obtained that $trad\Delta(\Delta_{n+1}) \cup 2tradG(G'\sigma_1 \ldots \sigma_{n+1}) \vdash_{SLD} 1tradG(G'\sigma_1 \ldots \sigma_{n+1})$ in one step and, by applying the lemma to G' (subterm of G), it is obtained $\Delta_{n+1} \vdash_{\supset} G'$ and therefore $\Delta_n \vdash_{\supset} D_{n+1} \supset G'$.

Induction Hypothesis (I.H) The lemma holds for a $\vdash_{SLD}$-deduction of m steps.

Case m+1 We proceed by structural induction on G:

– If $G = B$ then by hypothesis $trad\Delta(\Delta_n) \vdash_{SLD} B\sigma_1 \ldots \sigma_n$ in $m+1$ steps. Then there exists a program clause $G' \to B\sigma_1 \ldots \sigma_n \in trad\Delta(\Delta_n)$ such that $trad\Delta(\Delta_n) \vdash_{SLD} G'$ in m steps. By definition of $trad\Delta(\Delta_n)$, it must occur either
(1.1) there exists $i \in \{0 \ldots n\}$ such that $G' \to B\sigma_1 \ldots \sigma_n \in tradP(D_i\sigma_1 \ldots \sigma_i)$ or
(1.2) there exists $i \in \{1 \ldots n\}$ such that $G' \to B\sigma_1 \ldots \sigma_n \in ext(\sigma_i)$.
In both subcases $B\sigma_1 \ldots \sigma_n = B\sigma_1 \ldots \sigma_i$ since the new renamings $\sigma_{i+1}, \ldots, \sigma_n$ do not affect $tradP(D_i\sigma_1 \ldots \sigma_i) \cup ext(\sigma_i)$. On the other hand, as it is said in Remark 4, $trad\Delta(\Delta_n) \vdash_{SLD} G'$ in m steps implies that also $trad\Delta(\Delta_i) \vdash_{SLD} G'$ in m steps.
In the subcase (1.1), there exists a goal $G1$ such that $G1 \to B \in D_i$ (i.e., $G1\sigma_1 \ldots \sigma_i \to B\sigma_1 \ldots \sigma_i \in D_i\sigma_1 \ldots \sigma_i$) with $1tradG(G1\sigma_1 \ldots \sigma_i) = G'$ and $2tradG(G1\sigma_1 \ldots \sigma_i) \subseteq tradP(D_i\sigma_1 \ldots \sigma_i) \subseteq trad\Delta(\Delta_i)$.
Since $trad\Delta(\Delta_i) \vdash_{SLD} G'$ in m steps then $trad\Delta(\Delta_i) \cup 2tradG(G1\sigma_1 \ldots \sigma_i) \vdash_{SLD} 1tradG(G1\sigma_1 \ldots \sigma_i)$ in m steps and then, by applying (I.H), $\Delta_i \vdash_{\supset} G1$. But $G1 \to B \in D_i$ and therefore $\Delta_n \vdash_{\supset} B$.
In the subcase (1.2), by definition of $ext(\sigma_i)$, it holds that $G'{=}B\sigma_1 \ldots \sigma_{i-1}$. Then $trad\Delta(\Delta_n) \vdash_{SLD} B\sigma_1 \ldots \sigma_{i-1}$ in m steps which implies (see Remark 4) $trad\Delta(\Delta_{i-1}) \vdash_{SLD} B\sigma_1 \ldots \sigma_{i-1}$ in m steps. Now by applying (I.H) $\Delta_{i-1} \vdash_{\supset} B$ and therefore also $\Delta_n \vdash_{\supset} B$.

– If $G = G_1 \wedge G_2$ then by the hypothesis $trad\Delta(\Delta_n) \cup 2tradG(G_1\sigma_1 \ldots \sigma_n) \cup 2tradG(G_2\sigma_1 \ldots \sigma_n) \vdash_{SLD} 1tradG(G_1\sigma_1 \ldots \sigma_n) \wedge 1tradG(G_2\sigma_1 \ldots \sigma_n)$ in $m+1$ steps. Let Q be $trad\Delta(\Delta_n) \cup 2tradG(G_1\sigma_1 \ldots \sigma_n) \cup 2tradG(G_2\sigma_1 \ldots \sigma_n)$. Then both $Q \vdash_{SLD} 1tradG(G_1\sigma_1 \ldots \sigma_n)$ and $Q \vdash_{SLD} 1tradG(G_2\sigma_1 \ldots \sigma_n)$ in a number of steps less or equal than m. But in order to apply the (I.H) we need to have $Q_k \vdash_{SLD} 1tradG(G_k\sigma_1 \ldots \sigma_n)$ with Q_k being $trad\Delta(\Delta_n) \cup 2tradG(G_k\sigma_1 \ldots \sigma_n)$ for $k = 1, 2$. This fact is satisfied because, as it was explained in Remark 3, the renamings are parewise disjoint. The clauses in (for instance) $2tradG(G_2\sigma_1 \ldots \sigma_n)$ define new predicates for the inner blocks of $G_2\sigma_1 \ldots \sigma_n$ which do not appear in $1tradG(G_1\sigma_1 \ldots \sigma_n)$. Thus such set of clauses is not used to prove such goal (and similarly for the other case). Then the (I.H) can be applied to obtain $\Delta_n \vdash_{\supset} G_k$ for $k = 1, 2$ and therefore $\Delta_n \vdash_{\supset} G_1 \wedge G_2$.
– If $G = D_{n+1} \supset G'$ then $trad\Delta(\Delta_n) \cup tradP(D_{n+1}\sigma_1 \ldots \sigma_{n+1}) \cup ext(\sigma_{n+1}) \cup 2tradG(G'\sigma_1 \ldots \sigma_{n+1}) \vdash_{SLD} 1tradG(G'\sigma_1 \ldots \sigma_{n+1})$ in $m+1$ steps. That is, $trad\Delta(\Delta_{n+1}) \cup 2tradG(G'\sigma_1 \ldots \sigma_{n+1}) \vdash_{SLD} 1tradG(G'\sigma_1 \ldots \sigma_{n+1})$ in $m+1$ steps, with $\Delta_{n+1} = \Delta_n | D_{n+1}$. As we did before (in subcase (2) of Case $m = 1$), the lemma can now be applied to G' (subterm of G) and $m+1$ steps to obtain $\Delta_{n+1} \vdash_{\supset} G'$ and therefore $\Delta_n \vdash_{\supset} D_{n+1} \supset G'$.
Case $m+1$ has finished and therefore the proof of this lemma. ■

The soundness and completeness result is given in the following theorem:

Theorem 11. *For every Horn$^{\supset}$ program P and every atom A, it holds:*

$$P \vdash_{\supset} A \Longleftrightarrow tradP(P) \vdash_{SLD} A$$

Proof. Obvious from Corollary 8 and the previous Lemma (when the sequence of programs is a single program P and the goal is an atom A). ■

6 A Concrete Translation Algorithm in Haskell

In this section we implement the translation algorithm in the functional programming language Haskell. We have chosen a functional language since it is straightforward for implementing the two mutually recursive functions *tradP* and *tradG*. Basic types for representing programs and goals are given, where a program is represented by a list of program clauses, program clauses and goals are represented within two algebraic types and a substitution is given by a list of atom pairs.

```
-- Types --
type TProgram = [TClause]
data TClause = Flecha TAtom TGoal | And TClause TClause deriving (Eq,Show)
data TGoal = TRUE | At TAtom | AndG TGoal TGoal | Implica T TGoal
             deriving (Eq,Show)
type TAtom = String

type Sust = [(TAtom,TAtom)]
```

The functions $tradP$ and $tradG$ are defined in Haskell, according to the representation types, following the previously given abstract translation algorithm. The only difference is given by "concreting" the "renaming" for locally defined predicates. The translation functions are the following:

```
-- Translating a program --
tradP :: TProgram -> TProgram
tradP p = fst(tradLisC (p,[]))

-- Translating a list of program clauses --
tradLisC :: (TProgram,Sust) -> (TProgram,Sust)
tradLisC ([],s) =([],s)
tradLisC (d:res,s) = (p1++p2,s2)
                     where
                     (p1,s1) = tradC(d,s)
                     (p2,s2) = tradLisC(res,s1)

-- Translating a program clause --
tradC :: (TClause,Sust) -> (TProgram,Sust)
tradC (Flecha a g,s)
          | (g == TRUE) = ([Flecha a g],s)
          | otherwise   = ((Flecha a gnew):prognew,s1)
             where  (gnew,prognew,s1) = tradG(g,s)
tradC (And d1 d2,s) = (p1++p2,s2)
                      where
                      (p1,s1) = tradC(d1,s)
                      (p2,s2) = tradC(d2,s1)

-- Translating a goal --
tradG :: (TGoal,Sust) -> (TGoal, TProgram, Sust)
tradG (TRUE,s) = (TRUE,[],s)
tradG (At a,s) = (At a,[],s)
tradG (AndG g1 g2,s) = (AndG g1new g2new,p1++p2,s2)
                       where
                       (g1new,p1,s1) = tradG (g1,s)
                       (g2new,p2,s2) = tradG (g2,s1)
tradG (Implica d g,s)
    = (gnew,prog,s3)
            where
            lpr_loc = lPredClaus d
            sust = [(q, nuevo(q,s)) | q <- lpr_loc]
            clausExt = [Flecha newq (At oldq) | (oldq,newq) <- sust]
            s1 = union (s,sust)
            (p,s2) = tradC (aplicarC sust d,s1)
            (gnew,pnew,s3) = tradG (aplicarG sust g,s2)
            prog = p ++ clausExt ++ pnew

 -- Defined predicate list in a clause --
lPredClaus :: TClause-> [TAtom]
```

```
lPredClaus (Flecha a g) = [a]
lPredClaus (And d1 d2)  = quitarRep (lPredClaus d1 ++ lPredClaus d2)

-- Applying a substitution to a clause --
aplicarC :: Sust -> TClause-> TClause
aplicarC sust (Flecha a g) = Flecha (aplicarA sust a) (aplicarG sust g)
aplicarC sust (And d1 d2)  = And (aplicarC sust d1) (aplicarC sust d2)

-- Applying a substitution to a goal --
aplicarG :: Sust -> TGoal -> TGoal
aplicarG sust TRUE = TRUE
aplicarG sust (At a)        = At (aplicarA sust a)
aplicarG sust (AndG g1 g2)  = AndG (aplicarG sust g1) (aplicarG sust g2)
aplicarG sust (Implica d g) = Implica (aplicarC sust d) (aplicarG sust g)

-- Applying a substitution to an atom --
aplicarA :: Sust -> TAtom -> TAtom
aplicarA sust np  = if (lis == []) then np else head lis
                    where lis = [q | (p,q) <- sust, p == np]

-- Auxiliar functions --
union :: (Sust,Sust) -> Sust
union (s1,s2) = refine (s1 ++ s2)
                 where
                 refine [] = []
                 refine ((p,q):res)
                     | (filter ((== p).fst) res) /= [] = refine res
                     | otherwise  = (p,q): refine res

nuevo :: (TAtom,Sust) -> TAtom
nuevo (p,s)
          | (lis == []) = p++"_1"
          | otherwise = actualp
                where
                lis = filter ((== p).fst) s
                np = snd(head lis)
                (n,r) = break (=='_') (reverse np)
                actualp = reverse r ++ suma1 (reverse n)
                suma1 st = show (aNum st + 1)

aNum :: String -> Int
aNum s = foldr1 f [ord c - ord '0'| c <- s] where f n m = n*10 + m

quitarRep :: Eq a => [a] -> [a]
quitarRep []     = []
quitarRep (x:xs) = if x `elem` xs then quitarRep xs else x:quitarRep xs
```

To illustrate this concrete algorithm, let us take the following $Horn^{\supset}$ program, with four clauses (already written in Haskell):

```
prog   = [Flecha "p" TRUE,
          Flecha "t" TRUE,
          Flecha "r" (AndG g1 g2),
          Flecha "s" g3]
            where
            g1 = Implica d1 (At "q")
            g2 = Implica d2 (At "q")
            d1 = Flecha "q" (At "p")
            d2 = Flecha "q" (At "t")
            goal = AndG (At "q") goal2
            goal2 = Implica d3 (AndG (At "q") (At "p"))
            d3 = And d1 (Flecha "p" (At "r"))
            g3 = Implica (Flecha "q" (At "r")) goal
```

The translation algorithm gives the following result:

```
tradP prog = [Flecha "p" TRUE,
              Flecha "t" TRUE,
              Flecha "r" (AndG (At "q_1") (At "q_2")),
              Flecha "q_1" (At "p"),
              Flecha "q_1" (At "q"),
              Flecha "q_2" (At "t"),
              Flecha "q_2" (At "q"),
              Flecha "s" (AndG (At "q_3") (AndG (At "q_3_1") (At "p_1"))),
              Flecha "q_3" (At "r"),
              Flecha "q_3" (At "q"),
              Flecha "q_3_1" (At "p_1"),
              Flecha "p_1" (At "r"),
              Flecha "q_3_1" (At "q_3"),
              Flecha "p_1" (At "p")]
```

For the sake of legibility, a Haskell function "pr" has been added to the algorithm. This function permits to show clauses in a Prolog style, with the added connective "imp" for $\supset$. Here is an example of execution for the previous program "prog" and its translation:

```
Main> pr prog
p .
t .
r :- ( {q :- p .} imp (q) ) , ( {q :- t .} imp (q) ) .
s :- ( {q :- r .} imp (q , ( {q :- p .,   p :- r .} imp (q , p) )) ) .

Main> pr (tradP prog)
p .
t .
r :- q_1 , q_2 .
q_1 :- p .
q_1 :- q .
```

```
q_2 :- t .
q_2 :- q .
s :- q_3 , q_3_1 , p_1 .
q_3 :- r .
q_3 :- q .
q_3_1 :- p_1 .
p_1 :- r .
q_3_1 :- q_3 .
p_1 :- p .
```

7 Conclusions and Further Work

In this paper we have introduced a translation from $Horn^{\supset}$ programs to Horn clause programs, in the propositional case. Our main aim has been to prove that this translation is sound and complete with respect to the semantics of both programming languages. In concrete, the original operational semantics in the extended language $Horn^{\supset}$ can be now simulated by SLD-resolution in $Horn$.
The translation has been given first by an algorithm that can be considered "abstract" in two senses. On the one hand, we have forgotten "some details" in order to make more legible the proof of soundness and completeness and, on the other hand, it has been presented free of implementation language details. Later the forgotten details have been concreted, by selecting a particular way of renaming the new predicates, and a particular language (Haskell) has been used to implement the algorithm. Some examples have also been given in both levels of abstraction.
We think that the proposed translation algorithm is simple and that it is easy to see the relation between a $Horn^{\supset}$ program P and its translation $tradP(P)$. However, to formally establish the correspondence between the $\vdash_{\supset}$ deduction of an atom A from P and the corresponding $\vdash_{SLD}$ deduction of A from $tradP(P)$, it has been necessary to define the function $trad\Delta$, which translates a sequence of $Horn^{\supset}$ programs to a single $Horn$ program, and to establish a more general result (given in lemmas 7 and 10) for an arbitrary sequence of programs (or blocks) and a goal.

With respect to further work, we are actually working on lifting this translation to the first order case and we plan to prove it formally. The idea is to extend the algorithm in such a way that, for instance, the $Horn^{\supset}$ program $P = \{\forall X(p(X) \rightarrow q(X)).,\ p(a).,\ \forall Y((D \supset r(c,Y)) \rightarrow q(Y)).\}$ with $D = \{\forall X(p(X) \rightarrow r(X,X)).,\ p(c).\}$ would be translated to the $Horn$ program $tradP(P) = \{q(X) : -p(X).,\ p(a).,\ q(Y) : -r_1(c,Y).,\ r_1(Z,Z) : -p_1(Z).,\ p_1(c).,\ r_1(U,V) : -r(U,V).,\ p_1(T) : -p(T).\}$
As we can see in this example, besides introducing two new predicates (r_1 and p_1), the algorithm needs to distinguish between variables and constants and it has to extend each new predicate accordingly to its number of arguments.
We think that for $Horn^{\supset}$ programs with only "closed" sets of local clauses (that is, each D being itself also a program) an extended translation can easily be defined to conserve, as in the propositional case, the original semantics. However, for the whole class of $Horn^{\supset}$ programs where in general a local block D can have free variables which are seen as global variables for D (see [8, 1]) such translation may have more difficulties since it seems to yield to parameterised $Horn$ programs.

References

1. Arruabarrena, R., Lucio P. and Navarro, M. A Strong Logic Programming View for Static Embedded Implications. In: *Proc. of FOSSACS'99*,Springer-Verlag Lect. Notes in Comput. Sciences 1578:56-72 (1999).
2. Baldoni, M., Giordano, L., and Martelli, A. Translating a Modal Language with Embedded Implication into Horn Clause Logic In: *Proc. of 5 Int. Worhshop of Extensions of Logic Programming, ELP'96*, Springer-Verlag Lect. Notes in Comput. Sciences 1050(1996).
3. Bugliesi, M., Lamma, E. and Mello, P. Modularity in Logic Programming. *Journal of Logic Programming*, (19-20): 443-502, (1994).
4. Bonner, A. J., McCarty, L. T., and Vadaparty, K. Expresing Database Queries with Intuitionistic Logic. In: *Proc. of the North American Conf. on Logic Programming*, MIT Press, 831-850, (1989).
5. Gabbay, D. M. N-Prolog: An Extension of Prolog with Hypothetical Implications. II. Logical Foundations and Negation as Failure. *Journal of Logic Programming* 2(4):251-283 (1985).
6. Gabbay, D. M. and Reyle, U. N-Prolog: An Extension of Prolog with Hypothetical Implications. I. *Journal of Logic Programming* 1(4):319-355 (1984).
7. Giordano, L., and Martelli, A. Structuring Logic Programs: A Modal Approach. *Journal of Logic Programming* 21:59-94 (1994).
8. Giordano, L., Martelli, A., and Rossi, G. Extending Horn Clause Logic with Implication Goals. *Theoretical Computer Science* 95:43-74, (1992).
9. Lucio, P. Structured Sequent Calculi for Combining Intuitionistic and Classical First-Order Logic. In: *Proc. of FroCoSS'2000*,Springer-Verlag Lect. Notes in Artificial Intelligence 1794:88-104 (2000).
10. Meseguer, J. General Logics. In: Ebbinghaus H.-D. et al. (eds), *Logic Colloquium'87*, North-Holland, 275-329, (1989).
11. Meseguer, J. Multiparadigm Logic Programming. In: *Proc. of ALP'92*, Springer-Verlag Lect. Notes in Comput. Sciences 632:158-200, (1992).
12. Miller, D. A Logical Analysis of Modules in Logic Programming. In: *Journal of Logic Programming* 6:79-108, (1989).
13. Miller, D. Abstraction in Logic Programs. In: Odifreddi, P. (ed), *Logic and Computer Science*, Academic Press, 329-359, (1990).
14. Miller, D., Nadathur, G., Pfenning, F. and Scedrov, A. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and App. Logic* 51:125-157, (1991).
15. Monteiro, L., Porto, A. Contextual Logic Programming. In: *Proc. 6th International Conf. on Logic Programming* 284-299, (1989).
16. Moscowitz, Y., and Shapiro, E. Lexical logic programs. In: *Proc. 8th International Conf. on Logic Programming* 349-363, (1991).