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 ⊃ is a logic programming language, defined on the underlying logic FO ⊃ (an extension of 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 ⊃ programs to Horn clause programs, in the propositional setting, and we formally prove that this translation preserves the original operational semantics in Horn ⊃ 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 $\text{Horn}^\supset$ 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 $S_4$-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 $\text{Horn}^\supset$ 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 $\text{Horn}^\supset$ 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 $\text{Horn}^\supset$ 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 $\text{Horn}^\supset$ 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 $\text{Horn}^\supset$ is presented by giving its syntax and (operational) semantics (by means of the $\vdash_{\supset}$ deduction). Also some examples of goal deductions from $\text{Horn}^\supset$ 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^> 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^> 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^> 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^> 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):

\[
G := A \mid G_1 \land G_2 \mid D \supset G \mid \exists x G
\]
\[
D := A \mid G \rightarrow A \mid D_1 \land D_2 \mid \forall x D
\]

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:

\[
G := A \mid G_1 \land G_2 \mid D \supset G
\]
\[
D := A \mid G \rightarrow A \mid D_1 \land D_2
\]

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

Following [8, 1], we use a simple definition of the operational semantics of Horn^>, given by a non-deterministic set of rules which define when a goal \( G \) is operationally derivable from a program sequence \( \Delta = D_0 \upharpoonright D_1 \upharpoonright \ldots \upharpoonright D_n \), 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 \( \Delta \).

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

(2) \[
\frac{D_0 \upharpoonright \ldots \upharpoonright D_i \vdash_{\supset} G}{D_0 \upharpoonright \ldots \upharpoonright D_i \upharpoonright \ldots \upharpoonright D_n \vdash_{\supset} A}
\]
if \( G \rightarrow A \in D_i \) and \( 0 \leq i \leq n \)

(3) \[
\frac{\Delta \vdash_{\supset} G_1 \quad \Delta \vdash_{\supset} G_2}{\Delta \vdash_{\supset} G_1 \land G_2}
\]

(4) \[
\frac{\Delta \vdash_{\supset} G}{\Delta \vdash_{\supset} D \supset G}
\]

Fig. 1. Operational Semantics for Propositional Horn^>.
A sequence $\Delta = D_0|D_1|\ldots|D_n$ 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 $\Delta$ belongs to $D_i$, then the body $G$ must be proved from clauses only in $D_0|\ldots|D_i$ (that is, $D_{i+1},\ldots,D_n$ 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 $G_1 = a$. A proof of $P \vdash_G G_1$ is given by the following steps (applying rules in Figure 1):

$$
P \vdash_G a \quad \text{holds by Rule (2) for the clause} \quad (b \rightarrow c) \supset c \rightarrow a \quad \text{in} \quad P \text{ if}
$$

$$
P \vdash_G (b \rightarrow c) \supset c \quad \text{which holds by Rule (4) if}
$$

$$
P |\{b \rightarrow c\} \vdash_G c \quad \text{which holds by Rule (2) for the clause} \quad b \rightarrow c \quad \text{if}
$$

$$
P |\{b \rightarrow c\} \vdash_G b \quad \text{which holds by Rule (1) since} \quad b \in P |\{b \rightarrow c\}
$$

That is, in this case, for proving $P \vdash_G a$ it is necessary to prove $c$ from the "extended program" $P |\{b \rightarrow c\}$ which finally holds.

Now let the program with an unique clause $Q = \{(b \rightarrow c) \supset c \rightarrow a\}$ and let the goal $G_2 = b \supset a$. The only way to look for a proof of $Q \vdash_G G_2$ is by giving the following steps:

$$
Q \vdash_G b \supset a \quad \text{by Rule (4) if}
$$

$$
Q |\{b\} \vdash_G a \quad \text{by Rule (2) for the clause in} \quad Q \quad \text{if}
$$

$$
Q \vdash_G (b \rightarrow c) \supset c \quad \text{by Rule (4) if}
$$

$$
Q |\{b \rightarrow c\} \vdash_G c \quad \text{by Rule (2) for the clause} \quad b \rightarrow c \quad \text{if}
$$

$$
Q |\{b \rightarrow c\} \vdash_G b \quad \text{which does not hold}
$$

Therefore $Q \not\vdash_G G_2$. 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 $G_1 = a$ can be proved from the program $P$ because $b$ was defined in $P$, in the case of $G_2 = b \supset a$ 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 $\sigma$ for a set $\Sigma$ 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 $\text{tradP}$ for $D$-clauses and $\text{tradG}$ for $G$-clauses as follows:

$$
\text{tradP}(A) = \{A\}
$$

$$
\text{tradP}(G \rightarrow A) = \{G' \rightarrow A\} \cup P' \quad \text{where} \quad (G', P') = \text{tradG}(G)
$$

$$
\text{tradP}(D_1 \land D_2) = \text{tradP}(D_1) \cup \text{tradP}(D_2)
$$

$$
\text{tradG}(A) = \{A, \emptyset\}
$$

$$
\text{tradG}(G_1 \land G_2) = \{G'_1 \land G'_2, P_1 \cup P_2\} \quad \text{where}
$$

$$
(G'_1, P_1) = \text{tradG}(G_1) \quad \text{and} \quad (G'_2, P_2) = \text{tradG}(G_2)
$$
Definition 2. Given a sequence of Horn clauses \( \Delta_n = \Delta_0|\Delta_1|\ldots|\Delta_n \) where \( \sigma \) is a new renaming for predicates defined in \( \Delta \), \( (G', P_i) = \text{trad}G(G') \), \( P_2 = \text{trad}P(D\sigma) \) and \( P_3 = \text{ext}(\sigma) \)

**Note 1.** We shall use 1\( \text{trad}G(G) \) and 2\( \text{trad}P(G) \) for the first and second components (respectively) of \( \text{trad}G(G) \)

Since the language \( \text{Horn}^< \) works with sequences of programs, it is necessary to extend \( \text{trad}P \) to a new function \( \text{trad}\Delta \) which translates a sequence of \( \text{Horn}^< \) programs to a single \( \text{Horn} \) program.

**Definition 2.** Given a sequence of \( \text{Horn}^< \) programs \( \Delta_n = \Delta_0|\Delta_1|\ldots|\Delta_n \), the set of Horn clauses \( \text{trad}\Delta(\Delta_n) \) is recursively defined in the following way:

- For \( n = 0 \) (i.e. \( \Delta_n = \Delta_0 \)): \( \text{trad}\Delta(\Delta_n) = \text{trad}P(D_0) \)
- For \( n = i + 1 \) (i.e. \( \Delta_n = \Delta_i|\Delta_{i+1} \)):
  \[
  \text{trad}\Delta(\Delta_n) = \text{trad}\Delta(\Delta_i) \cup \text{trad}P(D_{i+1}\sigma_1\ldots\sigma_{i+1}) \cup \text{ext}(\sigma_{i+1}) \quad \text{where} \quad \sigma_{i+1} \quad \text{is a new renaming for predicates defined in} \quad D_{i+1}\sigma_1\ldots\sigma_i.
  \]

Again in the previous definition each renaming \( \sigma_i \) 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 \( \Delta_n = \Delta_0|\Delta_1|\ldots|\Delta_n \) be a sequence of \( \text{Horn}^< \) programs and let \( \Sigma \) be the set of predicate names in \( \Delta_n \). The algorithm proceeds by translating first \( \Delta_0 \) and obtaining \( \text{trad}P(D_0) \) whose predicates belong to the signature \( \Sigma \cup \Sigma'_0 \), with \( \Sigma'_0 \) 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 \( \sigma_1 \) for them. This means that \( \sigma_1 \) can be viewed as a function \( \sigma_1 : \text{DefPred}(D_1) \rightarrow \Sigma_{\sigma_1} \), such that \( \Sigma_{\sigma_1} \cap (\Sigma \cup \Sigma'_0) = \emptyset \). Then the algorithm proceeds by translating \( D_1\sigma_1 \) and obtaining \( \text{trad}P(D_1\sigma_1) \) whose predicates belong to the signature \( (\Sigma \cup \Sigma'_{\sigma_1}) \cup \Sigma'_1 \). \( \Sigma'_1 \) 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 \( \Sigma'_1 \cap \Sigma'_{\sigma_1} = \emptyset \). 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 \( \text{trad}P(D_1\sigma_1\ldots\sigma_j) \cup \text{ext}(\sigma_i) \) then \( G \) can neither include predicate names defined in \( \text{trad}P(D_1\sigma_1\ldots\sigma_j) \) nor those defined in \( \text{ext}(\sigma_j) \) for \( j > i \).

Since \( \text{trad}\Delta(D_0|\Delta_1|\ldots|\Delta_n) = \text{trad}P(D_0) \) \( \cup \ldots \cup \text{trad}P(D_1\sigma_1\ldots\sigma_i) \cup \text{ext}(\sigma_i) \cup \ldots \cup \text{trad}P(D_n\sigma_1\ldots\sigma_n) \cup \text{ext}(\sigma_n) \), if such \( G \) is deduced (by SLD-resolution) from the set of Horn clauses \( \text{trad}\Delta(D_0|\Delta_1|\ldots|\Delta_n) \) it means that \( G \) is indeed deduced from \( \text{trad}\Delta(D_0|\Delta_1|\ldots|\Delta_i) \).

### 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.
Lemma 7. For every sequence of Horn\supseteq programs $\Delta_n = D_0\mid D_1\mid \ldots \mid D_n$ with $n \geq 0$, and every goal $G$, it holds:

$$\Delta_n \vdash G \iff \text{trad}\Delta(\Delta_n) \cup 2\text{tradG}(G\sigma_1 \ldots \sigma_n) \vdash_{SLD} 1\text{tradG}(G\sigma_1 \ldots \sigma_n)$$

**Proof.** We proceed by induction on the number of steps (m) in the deduction proof of $\Delta_n \vdash G$. 

---

5 Soundness and Completeness

The aim of this section is to prove that the given translation algorithm is suitable for simulating a Horn\supseteq deduction by means of SLD-deduction. More concretely, given a Horn\supseteq program $P$ and given an atom $A$, $P \vdash A$ if and only if $\text{tradP}(P) \vdash_{SLD} A$.

To prove this result it is necessary to prove a more general result which is established in the following lemmas 7 and 10.
Case \( m = 1 \) In this case, \( G \) is necessarily an atom \( B \) belonging to \( D_i \) for some \( i \in \{0 \ldots n\} \). Therefore \( \text{ltrad}(G\sigma_1 \ldots \sigma_n) = B\sigma_1 \ldots \sigma_n \) and \( \text{2trad}(G\sigma_1 \ldots \sigma_n) = \emptyset \). Then we have to prove that \( \text{trad}_{\Delta_n} \vdash_{\text{SLD}} B\sigma_1 \ldots \sigma_n \).

Since \( B \in D_i \) it holds \( B\sigma_1 \ldots \sigma_n \in \text{trad}(D_i\sigma_1 \ldots \sigma_i) \) and then \( \text{trad}_{\Delta_n} \vdash_{\text{SLD}} B\sigma_1 \ldots \sigma_n \) in one step. But for each \( j \in \{i + 1 \ldots n\} \), either the program clause \( B\sigma_1 \ldots \sigma_j \rightarrow B\sigma_1 \ldots \sigma_j \) belongs to \( \text{ext}(\sigma_j) \) or \( B\sigma_1 \ldots \sigma_j \) is the same than \( B\sigma_1 \ldots \sigma_j \). Therefore \( \text{trad}_{\Delta_n} \vdash_{\text{SLD}} B\sigma_1 \ldots \sigma_n \) by applying several deduction steps in \( \vdash_{\text{SLD}} \).

**Induction hypothesis** The lemma holds whenever the number of steps in the deduction proof of \( \Delta_n \vdash G \) is less or equal than \( m \), for every \( \Delta_n \) and \( G \).

Case \( m + 1 \) It is supposed that \( \Delta_n \vdash G \) in \( m + 1 \) steps with \( m + 1 > 1 \). We proceed by case analysis on \( G \):

- If \( G = B \) then there exist \( i \in \{0 \ldots n\} \) and \( G_1 \) such that \( G_1 \rightarrow B \in D_i \) and \( \Delta_n \vdash G_1 \) in a number of steps less or equal than \( m \). Since \( G \) is an atom, again we have to prove that \( \text{trad}_{\Delta_n} \vdash_{\text{SLD}} B\sigma_1 \ldots \sigma_n \).
- By induction hypothesis, \( \text{trad}_{\Delta_n} \cup \text{2trad}(G_1\sigma_1 \ldots \sigma_i) \vdash_{\text{SLD}} \text{ltrad}(G_1\sigma_1 \ldots \sigma_i) \). Now \( G_1 \rightarrow B \in D_i \) implies that the Horn clause \( \text{ltrad}(G_1\sigma_1 \ldots \sigma_i) \rightarrow B\sigma_1 \ldots \sigma_i \) and the Horn clauses in the set \( \text{2trad}(G_1\sigma_1 \ldots \sigma_i) \) belong to \( \text{trad}(D_1\sigma_1 \ldots \sigma_i) \).

Then \( \text{trad}_{\Delta_n} \vdash_{\text{SLD}} B\sigma_1 \ldots \sigma_i \) and therefore (as in the case \( m = 1 \)) also \( \text{trad}_{\Delta_n} \vdash_{\text{SLD}} B\sigma_1 \ldots \sigma_n \).

- If \( G = G_1 \lor G_2 \) then both \( \Delta_n \vdash G_1 \) and \( \Delta_n \vdash G_2 \), each deduction in a number of steps \( \leq m \). By induction hypothesis, \( \text{trad}_{\Delta_n} \cup \text{2trad}(G_k\sigma_1 \ldots \sigma_n) \vdash_{\text{SLD}} \text{ltrad}(G_k\sigma_1 \ldots \sigma_n) \) for \( k = 1, 2 \).

Then it holds \( \text{trad}_{\Delta_n} \cup \text{2trad}(G_1\sigma_1 \ldots \sigma_n) \cup \text{2trad}(G_2\sigma_1 \ldots \sigma_n) \vdash_{\text{SLD}} \text{ltrad}(G_1\sigma_1 \ldots \sigma_n) \cup \text{ltrad}(G_2\sigma_1 \ldots \sigma_n) \) which is equivalent to \( \text{trad}_{\Delta_n} \cup \text{2trad}(G_1\sigma_1 \ldots \sigma_n) \vdash_{\text{SLD}} \text{ltrad}(G_1\sigma_1 \ldots \sigma_n) \cup \text{ltrad}(G_2\sigma_1 \ldots \sigma_n) \).

- If \( G = D_{n+1} \lor G' \) then for \( \Delta_{n+1} = \Delta_n|D_{n+1} \) it holds \( \Delta_{n+1} \vdash G' \) in a number of steps less or equal than \( m \) and by induction hypothesis \( \text{trad}_{\Delta_{n+1}} \cup \text{2trad}(G'\sigma_1 \ldots \sigma_{n+1}) \vdash_{\text{SLD}} \text{ltrad}(G'\sigma_1 \ldots \sigma_{n+1}) \).

But from definition of \( \text{ltrad} \) it holds \( \text{trad}_{\Delta_{n+1}} \cup \text{2trad}(G'\sigma_1 \ldots \sigma_{n+1}) = \text{trad}_{\Delta_n} \cup \text{2trad}(D_{n+1}\sigma_1 \ldots \sigma_{n+1}) \cup \text{ext}(\sigma_{n+1}) \cup \text{2trad}(G'\sigma_1 \ldots \sigma_{n+1}) \).

Then, we have obtained that \( \text{trad}_{\Delta_n} \cup \text{2trad}(D_{n+1} \lor G') \vdash_{\text{SLD}} \text{ltrad}(D_{n+1} \lor G') \).

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:

\[
P \vdash A \implies \text{trad}(P) \vdash_{\text{SLD}} A
\]

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 \( \vdash \)) can also be deduced by SLD-resolution from the translated program \( \text{trad}(P) \).

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 program $P$ and every Horn goal $G$, it holds:

$$P \vdash G \implies P' \vdash_{SLD} G'$$

where $(G', P')$ is the result of $\text{trad}(P \supset G)$

Proof. It is obvious from $P \vdash G \implies \emptyset \vdash (P \supset G)$ 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 programs $\Delta_n = D_0|D_1|\ldots|D_n$ with $n \geq 0$, and every goal $G$, it holds:

$$\text{trad}\Delta(\Delta_n) \cup 2\text{trad}G(\sigma_1\ldots\sigma_n) \vdash_{SLD} 1\text{trad}G(\sigma_1\ldots\sigma_n) \implies \Delta_n \vdash G$$

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, $1\text{trad}G(\sigma_1\ldots\sigma_n)$ is necessarily an atom $B_1$ belonging to $\text{trad}\Delta(\Delta_n) \cup 2\text{trad}G(\sigma_1\ldots\sigma_n)$. Since $B_1$ is an atom then either (1) $G = B$ for some atom $B$ (with $B_1 = B\sigma_1\ldots\sigma_n$) or (2) $G = D_{n+1} \supset G'$ for some goal $G'$ (with $B_1 = 1\text{trad}G(G'\sigma_1\ldots\sigma_{n+1})$).

In the subcase (1) we have that $\text{trad}\Delta(\Delta_n) \vdash_{SLD} B\sigma_1\ldots\sigma_n$ in one step; that is, $B_1 \sigma_1\ldots\sigma_n \in \text{trad}\Delta(\Delta_n)$. Then there exists $i \in \{0\ldots n\}$ such that $B_1 \sigma_1\ldots\sigma_n = B_1 \sigma_1\ldots\sigma_i \in \text{trad}P(D_i \sigma_1\ldots\sigma_i)$ and therefore $B \vdash \Delta_n$.

In the subcase (2) it holds $B_1 \in \text{trad}\Delta(\Delta_n) \cup 2\text{trad}G(G'\sigma_1\ldots\sigma_n) = \text{trad}\Delta(\Delta_n) \cup \text{trad}P(D_{n+1} \sigma_1\ldots\sigma_{n+1}) \cup \text{ext}(\sigma_{n+1}) \cup 2\text{trad}G(G'\sigma_1\ldots\sigma_{n+1}) = \text{trad}\Delta(\Delta_{n+1}) \cup 2\text{trad}G(G'\sigma_1\ldots\sigma_{n+1})$, for $\Delta_{n+1} = \Delta_n|D_{n+1}$. Then we have obtained that $\text{trad}\Delta(\Delta_{n+1}) \cup 2\text{trad}G(G'\sigma_1\ldots\sigma_{n+1}) \vdash_{SLD} 1\text{trad}G(G'\sigma_1\ldots\sigma_{n+1})$ in one step and, by applying the lemma to $G'$ (subterm of $G$), it is obtained $\Delta_{n+1} \vdash G'$ and therefore $\Delta_n \vdash D_{n+1} \supset G'$.

Induction Hypothesis (IH) 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 $\text{trad}\Delta(\Delta_n) \vdash_{SLD} B\sigma_1\ldots\sigma_n$ in $m+1$ steps.

Then there exists a program clause $G' \rightarrow B\sigma_1\ldots\sigma_n \in \text{trad}\Delta(\Delta_n)$ such that $\text{trad}\Delta(\Delta_n) \vdash_{SLD} G'$ in $m$ steps. By definition of $\text{trad}\Delta(\Delta_n)$, it must occur either (1.1) there exists $i \in \{0\ldots n\}$ such that $G' = B_1 \sigma_1\ldots\sigma_n \in \text{trad}P(D_i \sigma_1\ldots\sigma_i)$ or (1.2) there exists $i \in \{1\ldots n\}$ such that $G' = B_1 \sigma_1\ldots\sigma_n \in \text{ext}(\sigma_i)$.

In both subcases $B_1 \sigma_1\ldots\sigma_n = B_1 \sigma_1\ldots\sigma_i$ since the new renamings $\sigma_{i+1}\ldots\sigma_n$ do not affect $\text{trad}P(D_i \sigma_1\ldots\sigma_i) \cup \text{ext}(\sigma_i)$. On the other hand, as it is said in Remark 4, $\text{trad}\Delta(\Delta_n) \vdash_{SLD} G'$ in $m$ steps implies that also $\text{trad}\Delta(\Delta_i) \vdash_{SLD} G'$ in $m$ steps.

In the subcase (1.1), there exists a goal $G_1$ such that $G_1 \rightarrow B \in D_i$ (i.e., $G_1 \sigma_1\ldots\sigma_i \rightarrow B\sigma_1\ldots\sigma_i \in D_i \sigma_1\ldots\sigma_i$) with $1\text{trad}G(G_1 \sigma_1\ldots\sigma_i) = G'$ and $2\text{trad}G(G_1 \sigma_1\ldots\sigma_i) \subseteq \text{trad}P(D_i \sigma_1\ldots\sigma_i)$.

Since $\text{trad}\Delta(\Delta_i) \vdash_{SLD} G'$ in $m$ steps then $\text{trad}\Delta(\Delta_i) \cup 2\text{trad}G(G_1 \sigma_1\ldots\sigma_i) \vdash_{SLD} 1\text{trad}G(G_1 \sigma_1\ldots\sigma_i)$ in $m$ steps and then, by applying (IH), $\Delta_i \vdash G_1$.

But $G_1 \rightarrow B \in D_i$ and therefore $\Delta_i \vdash B$.

In the subcase (1.2), by definition of $\text{ext}(\sigma_i)$, it holds that $G' = B_1 \sigma_1\ldots\sigma_{i-1}$. Then $\text{trad}\Delta(\Delta_i) \vdash_{SLD} B_1 \sigma_1\ldots\sigma_{i-1}$ in $m$ steps which implies (see Remark 4) $\text{trad}\Delta(\Delta_{i-1}) \vdash_{SLD} B_1 \sigma_1\ldots\sigma_{i-1}$ in $m$ steps. Now by applying (IH) $\Delta_{i-1} \vdash B$ and therefore also $\Delta_i \vdash B$. 


-- 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 \texttt{tradP} and \texttt{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:

\begin{verbatim}
-- 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, novo(q,s)) | q <- lpr_loc]
        clauseExt = [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 ++ clauseExt ++ pnew

-- Defined predicate list in a clause --
lPredClaus :: TClause -> [TAtom]
\end{verbatim}
Predicate Claus (Flecha a g) = [a]
Predicate Claus (And d1 d2) = quitarRep (Predicate Claus d1 ++ Predicate Claus 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 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 .
7 Conclusions and Further Work

In this paper we have introduced a translation from $\text{Horn}^\supset$ 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 $\text{Horn}^\supset$ can be now simulated by SLD-resolution in $\text{Horn}$.

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 $\text{Horn}^\supset$ program $P$ and its translation $\text{trad}P(P)$. However, to formally establish the correspondence between the $\vdash\supset$ deduction of an atom $A$ from $P$ and the corresponding $\vdash_{\text{SLD}}$ deduction of $A$ from $\text{trad}P(P)$, it has been necessary to define the function $\text{trad}\Delta$, which translates a sequence of $\text{Horn}^\supset$ programs to a single $\text{Horn}$ 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 $\text{Horn}^\supset$ program $P = \{ \forall X (p(X) \implies q(X)), p(a), \forall Y ((D \supset r(c,Y)) \implies q(Y)) \}$ with $D = \{ \forall X (p(X) \implies r(X,X)), p(c) \}$ would be translated to the $\text{Horn}$ program $\text{trad}P(P) = \{ q(X) : -p(X), p(a), q(Y) : -r_1(c,Y), r_1(Z,Z) : -p_1(Z), p_1(c), r_1(U,V) : -r(U,V), p_1(T) : -p(T) \}$.

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 $\text{Horn}^\supset$ 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 $\text{Horn}^\supset$ 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 $\text{Horn}$ programs.
References


