1 Introduction

This note will explain the connection between logic and computer programming using Horn Clauses and a special type of resolution, namely SLD resolution.

2 Horn Clauses

We will begin by introducing the language we will use.

Definition 1. (Horn Clause) A Horn clause is a first order formula on the following form

$$\forall \bar{x} (B_1 \land \cdots \land B_n \rightarrow H)$$

where $B_i$ and $H$ are positive terms (including $\top$ and $\bot$) containing only free variables from $\bar{x}$.

Throughout this note we will implicitly universally quantify all variables and write the implication in reverse, such that the above clause will be written

$$H \leftarrow B_1 \land \cdots \land B_n$$

Definition 2. (Facts and goals) A Horn Clause on the form $H \leftarrow \top$ is called a fact, and a Horn Clause on the form $\bot \leftarrow B_1 \land \cdots \land B_m$ is called a goal clause. A Horn clause that is not a goal clause is called a definite clause.

A fact $H \leftarrow \top$ is often abbreviated as just $H$, however, we will write the full implication as it reminds us of the allowed syntax.

Horn clauses are interesting in many applications due to their many nice properties. E.g. the resolvent of two Horn clauses is a new Horn clause, and the resolvent of a goal clause and a definite clause is a new goal clause. We will see why these are desired features for a programming language in the next section. Throughout this note, we will also see other nice features of sets of Horn clauses.

Definition 3. (Logic Program) A logic program $P$ is a set of definite Horn clauses.

In logic programs we will use lowercase letters for constant symbols, function symbols and relational symbols, and capital letters for variables.
Example 4. Below is an example program defining addition over the natural numbers:

\[
\begin{align*}
\text{nat}(z) & \leftarrow \top \\
\text{nat}(s(N)) & \leftarrow \text{nat}(N) \\
\text{pluss}(N, z, N) & \leftarrow \text{nat}(N) \\
\text{pluss}(N, s(M), s(L)) & \leftarrow \text{pluss}(N, M, L)
\end{align*}
\]

where \(z\) denotes 0, and \(s\) is the successor function, such that e.g. \(s(z)\) denotes 1, \(s(s(z))\) denotes 2 and so on.

For future reference we will call this program \(\text{Nat}\).

We have not yet said anything about the semantics of a logic program, or how a computation can be carried out in this system. We will devote the rest of this section to the semantics, and then show how we can execute a logic program in the next section. Before we can introduce the semantics, we first need some definitions.

Definition 5. (Herbrand interpretation) Assume \(P\) is a logic program. The Herbrand universe \(\mathcal{H}\) of \(P\) is the set of all well-formed expressions constructed from the constants and function symbols in the language of \(P\). The set of all ground atomic formulas on the form \(R(t_1, \ldots, t_n)\), where \(R\) is an \(n\)-ary relation symbol in the language of \(P\) and \(t_i\) are elements of the Herbrand universe, is called the Herbrand base \(\hat{\mathcal{H}}\) of \(P\).

A Herbrand interpretation of \(P\) is any subset \(I \subseteq \hat{\mathcal{H}}\).

Definition 6. (Truth in Herbrand interpretations) Assume that \(I\) is a Herbrand interpretation for a logic program \(P\). Truth in a Herbrand interpretation is defined as follows:

(i) a ground atomic sentence \(R(t_1, \ldots, t_n)\) is true in \(I\) iff \(R(t_1, \ldots, t_n) \in I\);

(ii) a ground atomic negative sentence \(\neg R(t_1, \ldots, t_n)\) is true in \(I\) iff \(R(t_1, \ldots, t_n) \notin I\);

(iii) a ground conjunction \(L_1 \land \cdots \land L_n\) for atomic \(L_i\) is true in \(I\) iff \(L_i \in I\) for all \(i \leq n\);

(iv) a ground Horn Clause \(H \leftarrow B_1 \land \cdots \land B_n\) is true in \(I\) iff \(B_1 \land \cdots \land B_n \in I \Rightarrow H \in I\).

(v) a general Horn Clause \(H \leftarrow B_1 \land \cdots \land B_n\) is true in \(I\) iff every ground instance \(H\sigma \leftarrow B_1\sigma \land \cdots \land B_n\sigma\) is true in \(I\) (where \(\sigma\) is a substitution from the set of variables of the clause to \(\mathcal{H}\)).

Definition 7. (Herbrand model) A Herbrand interpretation \(I\) of a logic program \(P\) is a Herbrand model of \(P\) if all clauses of \(P\) are true in \(I\). We will call \(P\) consistent if it has a Herbrand model.

Theorem 8. Assume \(P\) is a consistent logic program and that \(M(P)\) is the set of Herbrand interpretations of \(P\). Then \(M(P) := \bigcap_{I \in M(P)} I\) is a Herbrand model of \(P\), and more specifically, it is the least Herbrand model of \(P\).

This property is often called the model intersection property, and does not hold for general sets of clauses. This property also allows us to construct the least Herbrand model by a fix-point operator.
Definition 9. (Immediate consequence operator) Let $P$ be a logic program. Define the immediate consequence operator $T_P$ for $P$ over a Herbrand interpretation $I$ as follows:

$$T_P(I) := I \cup \{H \in \mathcal{H} \mid H \leftarrow B_1 \land \cdots \land B_n \text{ is a ground instance of a rule in } P \text{ and } B_1, \ldots, B_n \in I\}$$

Theorem 10. Assume $P$ is a logic program. The least fix-point of $T_P$ applied to $\emptyset$ is the least Herbrand model.

Example 11. We have that

$I_0 := T_{\text{Nat}}(\emptyset) = \{\text{nat}(z)\}$

$I_1 := T_{\text{Nat}}(I_0) = I_0 \cup \{\text{nat}(s(z)), \text{pluss}(z, z, z)\}$

$I_2 := T_{\text{Nat}}(I_1) = I_1 \cup \{\text{pluss}(s(z), z, s(z)), \text{pluss}(z, s(z), s(z))\}$

and that

$M(\text{Nat}) = \{\text{nat}(z), \text{nat}(s(z)), \text{nat}(s(s(z))), \ldots,$

$\text{pluss}(z, z, z), \text{pluss}(s(z), z, s(z)), \text{pluss}(s(s(z)), z, s(s(z))), \ldots,$

$\text{pluss}(z, s(z), s(z)), \text{pluss}(z, s(s(z)), s(s(z))), \text{pluss}(z, s(s(z)), s(s(s(z))), \ldots\}$

Up until now, we have not yet specified a way of extracting information from our programs, except for inspecting the entire least Herbrand model. In normal programming, information extraction is done by executing a function and inspecting its result. However, information extraction in logic is commonly done using queries. Note that

$$\exists \overline{x}(B_1 \land \cdots \land B_n) \iff \neg \forall \overline{x} \neg (B_1 \land \cdots \land B_n)$$

$$\iff \neg \forall \overline{x} \leftarrow B_1 \land \cdots \land B_n$$

So to find a solution to a conjunctive query $\exists \overline{x}(B_1 \land \cdots \land B_n)$ over $P$, we need to find out when the goal clause $\bot \leftarrow B_1 \land \cdots \land B_n$ does not hold in $M(P)$.

Definition 12. (Query) We will call a goal clause $g \equiv \bot \leftarrow B_1 \land \cdots \land B_n$ a query over a logic program $P$ if $g$ is formulated over the language of $P$ and does not share any variables with $P$.

Definition 13. (Execution) Assume $P$ is a logic program and $g$ is a query over $P$. We will call a pair $\langle P, g \rangle$ an execution of $P$.

As in normal programming, there is a separation between the definitions in the program, in our case the definite clauses, and the main function (or program call), in our case the goal clause.

Example 14. The goal clauses

$$\bot \leftarrow \text{nat}(s(z))$$

$$\bot \leftarrow \text{pluss}(s(s(z)), s(s(z)), X)$$

are both queries over the program Nat from the previous example.
Definition 15. (Answer) An answer to an execution $E := \langle P, g \rangle$, denoted $\text{ans}(E)$, is the set $S$ of substitutions of the variables of $g$ to elements of $H$ such that $\{g\sigma\} \cup \mathcal{M}(P)$ is inconsistent.

Assume $g$ is a ground query over $P$. Then note that if $g \cup \mathcal{M}(P)$ is inconsistent, then $\text{ans}(\langle P, g \rangle) = \{\emptyset\}$, but if $g \cup \mathcal{M}(P)$ is consistent, then $\text{ans}(\langle P, g \rangle) = \emptyset$. So we can let $\{\emptyset\}$ denote true and $\emptyset$ denote false.

Example 16. We have

$$\text{ans}(\langle \text{Nat}, \bot \leftarrow \text{nat}(\text{s}(\text{s}(\text{z}))) \rangle) = \{\emptyset\} = \text{true}$$

$$\text{ans}(\langle \text{Nat}, \bot \leftarrow \text{pluss}(\text{s}(\text{s}(\text{s}(\text{s}(\text{z})))), \text{s}(\text{s}(\text{s}(\text{s}(\text{z})))), \text{X}) \rangle) = \{\{\text{s}(\text{s}(\text{s}(\text{s}(\text{s}(\text{s}(\text{z}})))))/\text{X}\}\}$$

Note that in the program Nat, we did not specify which of the variables were input and which were output. In fact, there is no such distinction! We could equally well pose a query

$$\bot \leftarrow \text{pluss}(\text{X}, \text{Y}, \text{s}(\text{s}(\text{z})))$$

and the output would be all combinations of numbers that sum to 3. Similarly, we could query

$$\bot \leftarrow \text{pluss}(\text{s}(\text{z}), \text{X}, \text{s}(\text{s}(\text{s}(\text{s}(\text{z}))))))$$

which tells us what we have to add to 2 to get 5, or equivalently, what we need to subtract from 5 to get 2. Since addition implicitly defines subtraction, we can derive subtraction from addition:

$$\text{minus}(M, N, R) \leftarrow \text{pluss}(N, R, M)$$

This is the essence of logic programming, stating what a solution to a problem is instead of how to obtain it.

Until now, we have only defined the semantics of our logic programs, executions, and the answers thereof. However, to use logic programming on a computer, the model theoretic semantics is not enough. Our models might be infinite, so constructing $\mathcal{M}(P)$ is not necessarily possible. Also, in most cases we are not interested in the entire model, only the answer to a particular query. In the next section, we will introduce an actual execution mechanism for logic programs.

3 SLD resolution

In this section we will show how a type of resolution, namely SLD resolution, can be used to execute logic programs to get answers to queries. However, before we introduce the resolution rule, let us look at a more intuitive way of executing a logic program.

What we really want from an execution $\langle P, g \rangle$, is a set of substitutions of the variables in the goal clause $g$ with elements of the Herbrand universe of $P$. Let us assume that $g$ has the form $\bot \leftarrow B_1 \land \cdots \land B_n$, with $n$ free variables $\vec{X}$. Then we want to find out for which element vectors $\vec{C} \in \mathcal{H}^n$ we have that

$$\mathcal{M}(P) \cup \{(\bot \leftarrow B_1 \land \cdots \land B_n)[\vec{C}/\vec{X}]\}$$
is unsatisfiable, that is, inconsistent, or equivalently, when

\[ \mathcal{M}(P) \models (B_1 \land \cdots \land B_n)[C/X] \]

holds.

Assume we have a rule in \( P \) on the form \( H \leftarrow R_1 \land \cdots \land R_m \) where \( B_i \) unifies with \( H \) with unifier \( \sigma \). Then we know that \( B_i\sigma \leftarrow (R_1 \land \cdots \land R_m)\sigma \), so if we find out when \( (R_1 \land \cdots \land R_m)\sigma \) holds, then we know that \( B_i \) holds under the same substitution. Hence, if we just recursively apply implications in this way and apply the substitutions we get to the entire query, and this process ever terminates with an empty clause \( \top \), we have found a substitution \( \sigma' = [C/X] \) that makes the query true in \( \mathcal{M}(P) \).

**Example 17.** Assume we have the execution from above:

\[ \text{ans}((\text{Nat}, \bot \leftarrow \text{pluss}(s(s(z))), s(s(z)), X)) \]

then we want to find out for which \( c \) we have \( \text{Nat} \models \text{pluss}(s(s(z))), s(s(z)), c \).

We know that \( \text{pluss}(N, s(M), s(L)) \leftarrow \text{pluss}(N, M, L) \) and \( \text{pluss}(s(s(z))), s(s(z)), c \) unifies with \( \text{pluss}(N, s(M), s(L)) \) with unifier

\[ \sigma_0 := \{ s(W)/X, W/L, s(s(z))/N, s(z)/M \}. \]

If we apply this unifier to the body of the rule, we get \( \text{pluss}(N, M, L)\sigma_0 = \text{pluss}(s(s(z))), s(z), s(W)) \). So this is our new goal clause, that is, if we find a solution for \( W \), we have a solution for \( X \) and therefore a solution to our query. If we apply the same rule again, we get the unifier

\[ \sigma_1 := \{ s(s(z))/N, s(z)/M, s(V)/W \} \]

and a new goal clause \( \text{pluss}(s(s(z))), z, s(s(V))) \). This goal clause now unifies with the rule \( \text{pluss}(N, z, N) \leftarrow \text{natN} \) with unifier

\[ \sigma_2 := \{ s(s(z))/N, s(s(z))/V \}. \]

Applying this to the body gives us a new goal clause \( \text{nat}(s(s(z))) \), for which we can just iterate the second rule of \( \text{Nat} \) three times and the first rule once to get to the goal clause \( \top \). We then have the unifiers

\[
\begin{align*}
\sigma_0 & := \{ s(W)/X, W/L, s(s(z))/N, s(z)/M \} \\
\sigma_1 & := \{ s(s(z))/N, s(z)/M, s(V)/W \} \\
\sigma_2 & := \{ s(s(z))/N, s(s(z))/V \}
\end{align*}
\]

We can then apply the unifiers in sequence to \( X \), and obtain the result \( \{ s(s(s(s(z)))))/X \} \) which is an answer to the execution.

We can formulate this application and unification as a deduction rule as follows:

\[
\frac{\bot \leftarrow L_1 \land \cdots \land L_i \land \cdots \land L_m}{\bot \leftarrow L_1 \land \cdots \land K_1 \land \cdots \land K_m} \quad \frac{L \leftarrow K_i \land \cdots \land K_m}{(\bot \leftarrow L_1 \land \cdots \land K_1 \land \cdots \land K_m \land \cdots \land L_m)\sigma}
\]
where \( L \) unifies with \( L_i \) with most general unifier \( \sigma \).

It turns out that this intuitive execution algorithm actually is a special form of resolution, which we now will formulate. First, we need to translate our formulas from implications into disjunctions in the standard way. So rules on the form

\[ H \leftarrow B_1 \land \cdots \land B_n \]

becomes

\[ H \lor \neg B_1 \lor \cdots \lor \neg B_n \]

Note that a fact \( H \leftarrow \top \) is then \( H \lor \neg \top \) which is equivalent to just \( H \), and a query \( \bot \leftarrow B_1 \land \cdots \land B_n \) becomes \( \bot \lor \neg B_1 \lor \cdots \lor \neg B_n \), which is equivalent to \( \neg B_1 \lor \cdots \lor \neg B_n \). We are now ready to introduce the SLD resolution rule:

**Definition 18.** The SLD resolution rule is the following inference rule:

\[
\begin{align*}
&\neg L_1 \lor \cdots \lor \neg L_i \lor \cdots \lor \neg L_n \\
&L \lor \neg K_1 \lor \cdots \lor \neg K_m
\end{align*}
\]

\[
\frac{\neg L_1 \lor \cdots \lor \neg L_i \lor \cdots \lor \neg L_n}{L \lor \neg K_1 \lor \cdots \lor \neg K_m \lor \cdots \lor \neg L_n \sigma}
\]

where \( L \) unifies with \( L_i \) with most general unifier \( \sigma \).

Note that SLD resolution always unifies a goal clause (the left-most clause in the rule) with a definite clause (the right-most clause in the rule).

**Definition 19.** (SLD-derivation) Assume \( E \) is an execution. An SLD-derivation is a sequence of SLD resolution rule applications to \( E \).

**Definition 20.** (Successful branch) A successful branch is an SLD-derivation that ends with the goal clause \( \bot \).

**Definition 21.** (Computed answers) Assume \( E = (P, g) \) is an execution. We call the the restriction of the composition of the sequence of unifiers obtained in a successful branch in an SLD-derivation to the variables of \( g \), a computed answer of \( E \). The set of all computed answers of \( E \) is denoted \( \text{cans}(E) \).

**Theorem 22.** \( \text{cans}(E) = \text{ans}(E) \).

To find the answers to a query over a program, we simply apply the SLD resolution rule until it cannot be applied any more. If we have reached a contradiction, we have an answer. However, there might be more answers. Furthermore, it is not always the case that we get a contradiction. If we get to either of these leaf nodes in the proof tree, we need to backtrack and try a different choice of rule. While backtracking we apply the rule in reverse, hence we need to undo the unification and reintroduce removed goal literals. This process can be viewed as a search. When we have tried all possible choices and visited all possible leaf nodes, we are done.

**Example 23.** Assume we have the following logic program:

\[
\begin{align*}
r(a,b) & \leftarrow \top \\
r(a,a) & \leftarrow \top \\
r(c,d) & \leftarrow \top \\
r(X,Y) & \leftarrow r(X,Z) \land r(Z,Y)
\end{align*}
\]
and we have the query $g_0 := \bot \leftarrow r(a,W)$. Then we might start by unifying this with the last rule, obtain the new goal clause $g_1 := \bot \leftarrow r(a,V), r(V,Q)$, which we then can unify against the last fact to obtain $g_2 := \bot \leftarrow r(a,c), r(c,d)$.

However, the first of the two literals cannot be unified against any rule, and we need to backtrack. We would then get back to the goal clause $g_1$ and try a different unification, e.g. the first literal with the second fact.

**Theorem 24.** Executions of logic programs with SLD resolution is Turing complete.

This obviously means that the search procedure we have outlined will not always terminate, as it is with any Turing complete programming language.

Also note that even though the model of the program in 23 is finite, we still cannot guarantee termination of our search, since we could keep unifying the literals in our goal clause with the last rule, giving an infinite chain of increasing goal clauses.

For termination and efficiency problems, we normally have, in logic programming, the following relationship:

$$\text{Algorithm} = \text{Logic} + \text{Control}. $$

That is, an algorithm is composed of the logical axioms (read: rules and facts) and the strategy for searching for a solution (read: the choices made in the SLD-derivation). The reader might think I have been dishonest in saying that logic programming is all about describing what a solution to a problem is, rather than how to obtain it, and the reader is actually partially right. For practical application we need to pay some attention to the hows as well. This control is either something the implementation of the language could take full responsibility for, something the programmer can adapt for a particular application, or a combination of the two.

Prolog, the perhaps most used logic programming language, uses a combination. The implementation of Prolog has a lot of optimisations and advanced search strategies for the SLD-derivation, and in addition, the programmer is given many control features, e.g.:

- the order of the rules matter, such that the search procedure will try to unify the goal clause with rules in the order they appear. This makes, e.g. the program from example 23 guaranteed to terminate, if there is a solution;
- the order of the literals in the clauses matter, such that the user can put literals that are harder to satisfy first, pruning the search tree;
- a special purpose cut literal, telling the SLD-search to not backtrack past the introduction of the cut as a goal;
- a special assert predicate that destructively adds new facts.

All these features reduces the logical purity of Prolog, but is necessary for efficient execution of Prolog programs.

Another language, Mercury, lets the user restrict the usage of each relation, such as restricting which arguments of a head needs to be instantiated before a unification with the rule can take place. This allows one to e.g. specify that a rule should behave as a function, meaning it is one way and deterministic. This information can be used to compile the rules into very efficient code.
There are also logic programming languages that are used as query languages over databases, such as Datalog, although these do normally not employ SLD-resolution as execution mechanism. Datalog disallows function symbols, which forces every model to be finite. This makes every query terminate.

We will finish this note with a larger example of a logic program for binary trees:

**Example 25.** The below program defines the binary tree data structure:

\[
\begin{align*}
\text{bin}(\text{nil}) & \leftarrow \top \\
\text{bin}(\text{tree}(V, X, Y)) & \leftarrow \text{bin}(X) \land \text{bin}(Y) \\
\text{member}(V, \text{tree}(V, X, Y)) & \leftarrow \text{bin}(X) \land \text{bin}(Y) \\
\text{member}(V, \text{tree}(W, X, Y)) & \leftarrow \text{member}(V, X) \land \text{bin}(Y) \\
\text{member}(V, \text{tree}(W, X, Y)) & \leftarrow \text{member}(V, Y) \land \text{bin}(X) \\
\text{containedIn}(\text{nil}, Z) & \leftarrow \text{bin}(Z) \\
\text{containedIn}(\text{tree}(V, X, Y), Z) & \leftarrow \text{member}(V, Z) \land \text{containedIn}(X, Z) \land \text{containedIn}(Y, Z) \\
\text{sameMembers}(X, Y) & \leftarrow \text{containedIn}(X, Y) \land \text{containedIn}(Y, X) \\
\text{add}(V, X, Y) & \leftarrow \text{sameMembers}(\text{tree}(V, X, \text{nil}), Y) \\
\text{lessThanAll}(V, \text{nil}) & \leftarrow \top \\
\text{lessThanAll}(V, \text{tree}(W, X, Y)) & \leftarrow \text{lessThan}(V, W) \land \text{lessThanAll}(V, X) \land \text{lessThanAll}(V, Y) \\
\text{greaterThanAll}(V, \text{nil}) & \leftarrow \top \\
\text{greaterThanAll}(V, \text{tree}(W, X, Y)) & \leftarrow \text{greaterThan}(V, W) \land \text{greaterThanAll}(V, X) \land \text{greaterThanAll}(V, Y) \\
\text{sorted}(\text{nil}) & \leftarrow \top \\
\text{sorted}(\text{tree}(V, X, Y)) & \leftarrow \text{lessThanAll}(V, Y) \land \text{greaterThanAll}(V, X) \land \text{sorted}(X) \land \text{sorted}(Y) \\
\text{sort}(X, Y) & \leftarrow \text{sameMembers}(X, Y) \land \text{sorted}(Y)
\end{align*}
\]

We assume that the value used in the binary trees have the relations lessThan and greaterThan already implemented. We can now sort a tree by posing a
query, e.g.

\[ \bot \leftarrow \text{sort}(\text{tree}(1, \text{tree}(3, \text{nil}, \text{tree}(8, \text{nil}, \text{nil})), \text{tree}(5, \text{tree}(0, \text{nil}, \text{nil}), \text{nil})), X) \]

One possible answer to the query is

\[ \{ \text{tree}(0, \text{nil}, \text{tree}(1, \text{nil}, \text{tree}(3, \text{nil}, \text{tree}(5, \text{nil}, \text{tree}(8, \text{nil}, \text{nil})))), X \} \]}