Hornify *

Pedro Barroso Mário Pereira António Ravara

July 2019

The Horn algorithm is a simple and easy solution to determine with polynomial complexity if a given propositional formula is satisfactory or contradictory. However, the algorithm works only for a particular set of formulae - the Horn Clauses. There are few presentation of this transformation algorithm and usually imperative, we could not find any functional presentation neither its implementation. This section presents the transformation algorithm from CNF to Horn Clause, including the implementation and verification of the algorithm.

1 Algorithm Definition

A basic Horn clause is a disjunction of literals, where at most one occurs positively. So, there are only three possibilities for a basic Horn clause:

1. Does not have any positive literal.
2. Does not have any negative literal, being only one positive literal.
3. Does have negative literals and only one positive.

Therefore, it is possible to present any basic Horn Clause as an implication:

1. $L \equiv \top \rightarrow L$
2. $\bigvee_{i=1}^{n} \neg L_i \equiv (\bigvee_{i=1}^{n} L_i) \rightarrow \bot$
3. $\bigvee_{i=1}^{n} \neg L_i \vee L \equiv (\bigvee_{i=1}^{n} L_i) \rightarrow L$

Where L and Li (for all i) are positive literals.

*This work is funded by Tezos Foundation through the FACTOR project.

A propositional formula is a Horn clause if it is a basic conjunction of Horn clauses:

$$\phi = \bigwedge_{i=1}^{n} (Ci \rightarrow L_i)$$

Where Li are positive literals and $C_i = \top$ *or* $C_i = \bigwedge_{j=1}^{k_i} L_{i,j}$

The following grammar defines a Horn formula:

$$\begin{aligned}
\psi &::= \mu \mid \psi \wedge \psi && \textit{(hornformula)} \\
\mu &::= \chi \rightarrow \omega && \textit{(basichornformula)} \\
\chi &::= \top \mid \alpha && \textit{(leftside)} \\
\alpha &::= p \mid \bot \mid \alpha \wedge \alpha && \textit{(positive)} \\
\omega &::= p \mid \bot \mid \top && \textit{(rightside)}
\end{aligned}$$

2 Functional Presentation of the Algorithm

The algorithm converts to a conjunction of basic Horn clauses, given a specific formula ϕ in conjunctive normal form that is defined by the following grammar:

$$\begin{aligned}
\phi &::= \phi \wedge \phi \mid \tau && \textit{(formula_cnf)} \\
\tau &::= l \mid \neg l \mid \tau \vee \tau && \textit{(disjunction)} \\
l &::= p \mid false && \textit{(literal)}
\end{aligned}$$

The main function is the `hornify` and has the following signature:

$$\text{hornify} : formula\_cnf \rightarrow hornformula$$

Precisely, the function goes through each sub-formula of the conjunctions and calls the `getBasicHorn` function:

$$\text{hornify}(\phi) \triangleq \begin{cases} \text{hornify}(\phi_1) \wedge \text{hornify}(\phi_2), & if\ \phi = \phi_1 \wedge \phi_2 \\ \text{getBasicHorn}(\phi), & if\ otherwise \end{cases}$$

The function `getBasicHorn` converts propositional formulae in CNF without conjunctions (τ) into basic Horn clauses:

$$\text{hornify} : formula\_cnf \rightarrow basichornformula$$

$$\text{getBasicHorn}(\phi) \triangleq \begin{cases} let\ (s,p) = \text{hornify\_aux}\ \phi\ \varnothing\ \varnothing\ in & \\ (\text{conjunction}\ s) \rightarrow (\text{getPositive}\ p), & if\ \phi = \phi_1 \vee \phi_2 \\ \phi_1 \rightarrow \bot, & if\ \phi = \neg\phi_1\ and\ \phi_1\ is\ a\ variable \\ \top \rightarrow \phi_1, & if\ \phi = \phi_1\ and\ \phi_1\ is\ a\ variable \\ \top \rightarrow \top, & if\ \phi = \neg false \\ \top \rightarrow \bot, & if\ \phi = false \end{cases}$$

This functions follows the properties presented in the Section 1. The main case of the function is when the formula is a disjunction. In this case, the function calls the `hornify_aux` functions to get all the positive and negative literals into a set, then uses the `conjunction` function to build the conjunction of the negative literals and the `getPositive` to get the positive literal. In the other cases the transformation is straightforward.

The function `hornify_aux` goes trought the formula and adds negative literals to the left set and positive literals to the right set:

$$\text{hornify\_aux} : disjunction * set * set \rightarrow set * set$$

$$\text{hornify\_aux}(\phi, s, p) \triangleq \begin{cases} let\ (s1, p1) = \text{hornify\_aux}\ \phi_1\ s\ p\ in & \\ \text{hornify\_aux}\ \phi_2\ s1\ p1, & if\ \phi = \phi_1 \vee \phi_2 \\ (s\ \cup\ \{\phi_1\},\ p), & if\ \phi\ =\ \neg\phi_1 \\ (s, \{\phi\}), & if\ \phi\ is\ a\ variable\ and\ p\ = \varnothing \end{cases}$$

The `conjunction` function returns a positive literal if the set has only one literal or constructs a conjunction with all the positive literals in the set:

$$\text{conjunction} : set \rightarrow leftside$$

$$conjunction(s) \triangleq \begin{cases} \phi, & if\ \phi \in s\ and\ |s| = 1 \\ \phi\ \wedge\ (\text{conjunction}\ (s \setminus \{\phi\})), & if\ \phi \in s\ and\ |s| > 1 \end{cases}$$

The `getPositive` functions is responsible to build the right side of the implication. Given a empty set or a set with only one positive literal, returns one positive literal ($\varnothing$ or itself):

$$\text{getPositive} : set \rightarrow rightside$$

$$getPositive(p) \triangleq \begin{cases} \bot, & if\ p = \varnothing \\ \phi, & if\ p = \{\phi\} \end{cases}$$

3 Implementation

Firstly, it is necessary to define the specific formulae types according to the grammar defined in Section 1:

- $\psi ::= \mu \mid \psi \wedge \psi$:

```
type hornclause =
  | HBasic basichornclause
  | HAnd hornclause hornclause
```

- $\mu ::= \chi \rightarrow \omega$:

```
type basichornclause =
  | BImpl leftside rightside
```

- $\chi ::= \top \mid \alpha$:

```
type leftside =
  | LTop
  | LPos positive
```

- $\alpha ::= p \mid \bot \mid \alpha \wedge \alpha$:

```
type positive =
  | PLCBottom
  | PLCVar ident
  | PLCAnd positive positive
```

- $\omega ::= p \mid \bot \mid \top$

```
type rightside =
  | RBottom
  | RTop
  | RVar ident
```

Functions. The implementation of the functions follows the structure of the mathematical definitions: The main function (`hornify`) goes through the formula and calls the `getBasicHorn` function when the formula is a disjunction `FClause_cnf`:

```
let rec hornify (phi: formula_cnf) : hornclause
  = match phi with
      | FClause_cnf phi1 → HBasic (getBasicHorn phi1)
      | FAnd_cnf phi1 phi2 → HAnd (hornify phi1) (hornify phi2)
  end
```

As mention before, the principal case of the function is when the formula is a disjunction. In this case, the `hornify_aux` functions is called in order to get all the positive and negative literals and uses the `conjunction` and `getPositive` functions to build the implication:

```
let getBasicHorn (phi: clause_cnf) : basichornclause
  = match phi with
      | DLiteral (LVar x) → BImpl (LTop) (RVar x)
      | DLiteral (LBottom) → BImpl (LTop) (RBottom)
      | DNeg_cnf (LVar x) → BImpl (LPos (PLCVar x)) (RBottom)
      | DNeg_cnf (LBottom) → BImpl (LTop) (RTop)
      | DOr_cnf _ _ → let (s,p) = hornify_aux phi (empty ()) None in
                              BImpl (conjunction s) (getPositive p)
    end
```

The `conjunction` function uses an built-in function `build` that returns a positive literal if the set has only one literal or constructs a conjunction with all the positive literals in the set:

```
let conjunction (s: set): leftside
  = let rec build (s: set)
     = if(is_empty s) then absurd else
            if((cardinal s) = 1) then (choose s) else
            PLCAnd (choose s) (build (remove (choose s) s)) in
       LPos (build s)
```

The `getPositive` functions returns one positive literal – $\bot$ if the `option` is `None` or itself if it is `Some x`:

```
 let getPositive (p: option rightside) : rightside
  = match p with
      | None → RBottom
      | Some x → x
    end
```

The `hornify_aux` function goes through the disjunction combinations, positively adds the negative literals to the input set and the positive literal to the `option` type. This function can also raise the `MoreThanOnePositive` exception if there is more than one positive literal in the formula:

```
let rec hornify_aux (phi: clause_cnf) (s: set) (p: option rightside) : (rs: set, rp: option rightside)
  raises{ MoreThanOnePositive }
  = match phi with
      | DOr_cnf (DLiteral _) (DLiteral _) → raise MoreThanOnePositive
      | DOr_cnf (DLiteral pl) (DNeg_cnf nl) | DOr_cnf (DNeg_cnf nl) (DLiteral pl) →
```

```
        processCombination pl nl s p
    | DOr_cnf (DNeg_cnf nl1) (DNeg_cnf nl2) →
        ((add (convertLiteralToPLC nl1) (add (convertLiteralToPLC nl2) s)), p)
    | DOr_cnf (DOr_cnf phi1 phi2) (DLiteral pl) | DOr_cnf (DLiteral pl) (DOr_cnf phi1 phi2) →
        match p with
            | None → hornify_aux (DOr_cnf phi1 phi2) s (Some (convertLiteralToR pl))
            | Some _ → raise MoreThanOnePositive
        end
    | DOr_cnf (DOr_cnf phi1 phi2) (DNeg_cnf nl) | DOr_cnf (DNeg_cnf nl) (DOr_cnf phi1 phi2) →
        hornify_aux (DOr_cnf phi1 phi2) (add (convertLiteralToPLC nl) s) p
    | DOr_cnf phi1 phi2 → let (s1,p1) = hornify_aux phi1 s p in
                          hornify_aux phi2 s1 p1
    | _ → absurd
  end
```

The `processCombination` function processes the disjunction formula when one is positive and the other is negative. If the `option` is `None` then the `addLiterals` function is called, otherwise the `MoreThanOnePositive` exception is raised:

```
let processCombination (pl: pliteral) (nl: pliteral) (s: set) (p: option rightside) : (r
    raises{ MoreThanOnePositive }
  = match p with
      | None → addLiterals pl nl s p
      | Some _ → raise MoreThanOnePositive
    end
```

The `addLiterals` function adds the negative literal to the set and sets the `option` with the positive literal:

```
let addLiterals (pl: pliteral) (nl: pliteral) (s: set) (p: option rightside) : (rs: set,
  = match pl with
      | LBottom → let rbottom = Some (RBottom) in
                  match nl with
                    | LBottom → ((add (PLCBottom) s), rbottom)
                    | LVar x → ((add (PLCVar x) s), rbottom)
                  end
      | LVar x → let rvar = Some (RVar x) in
                  match nl with
                    | LBottom → ((add (PLCBottom) s), rvar)
                    | LVar x → ((add (PLCVar x) s), rvar)

                  end
  end
```

The complete code is in Appendix A

4 Verification

Since the defined types represent exactly the grammar, the equivalence of the input and output formula is the only criterion needed to ensure the verification. The valuation functions for each type ensures this criterion (Apendix B).

Less trivial cases. In some cases, especially when the formula has more than one input and output, the equivalence of the formulae can be a little tricky. A combination of valuation functions is needed to resolve these cases. In the `hornify_aux` function it is need to ensure that:

1. the output set has at least one literal (is not empty).
2. phi $\vee$ $(\vee_{i=1}^{n} \neg E_i) \vee$ p = $(\vee_{i=1}^{n} \neg R_i) \vee$ rp
3. phi $\vee$ $(\vee_{i=1}^{n} \neg E_i) \vee$ p = $(\wedge i = 1^{n} R_i) \rightarrow$ rp

Where phi is a disjunction, E_i are the elements of the input set, p is the input `option`, R_i are the elements of the output set and rp the output `option`.

This function only receives disjunctions, the other cases are absurd. Given the need to prove the absurd cases and since the `clause_cnf` type contains non-disjunction constructors, it is a must to ensure that the input formula (`phi`) is effectively a disjunction. This obligation is ensured by adding a precondition in the specification of the function:

```
let rec hornify_aux (phi: clause_cnf) (s: set) (p: option rightside) : (rs: set, rp: option rightside)
  requires{ exists x y. phi = DOr_cnf x y }
  ensures{ (not is_empty rs) }
  ensures{ forall v. eval_domain v phi s p = eval_codomain v rs rp }
  ensures{ forall v. eval_domain v phi s p = implb (eval_conjunction_set v rs) (eval_optionrightside v rp) }
  variant{ phi }
  ...
  = match phi with
        ...
        | _ → absurd
    end
```

The `eval_domain` evaluates the domain of the function (left side of the equality of property 2 and 3) and the `eval_codomain` the codomain (right side of the equality of property 2):

```
function eval_domain (v: valuation) (phi: clause_cnf) (s: set) (p: option rightside) : bool
  = eval_disj_cnf v phi || eval_set v s || eval_optionrightside v p

function eval_codomain (v: valuation) (s: set) (p: option rightside) : bool
  = eval_set v s || eval_optionrightside v p
```

In the `processCombination` and `addLiterals` functions it is needed to ensure, once again, that the output set is not empty and that the domain is equivalent of the codomain. This obligation can be traduced into the following property:

$$pl \vee \neg nl \vee (\vee_{i=1}^{n} E_i) \vee p = (\vee_{i=1}^{n} R_i) \vee rp$$

Where pl and nl are the positive literal and negative literal respectively, E_i are the elements of the input set, p is the input `option`, R_i are the elements of the output set and rp the output `option`. The specification is the following one:

```
ensures{ (not is_empty rs) }
ensures { forall v. (eval_literal v pl || not (eval_literal v nl) || eval_set v s || eval_optionrightside v p)
          = (eval_set v rs || eval_optionrightside v rp) }
```

The full specification is in Appendix C.

Auxiliary Lemma and Axiom. Evaluation functions some times need to evaluate all the elements of a given set. Unfortunately, there are some properties that the type set of Why3 cannot assure. For example, the De Morgan's laws over elements of a set.

We defined the `deMorgan` lemma following the structural induction proof method. The objective with this lemma is to assure that the evaluation of a disjunction over all the elements of a set is equal to the negation of the evaluation of a conjunction over all the elements of a set, but this time with the elements negated:

```
let rec lemma deMorgan (v: valuation) (s: fset positive)
    requires{ not is_empty s }
    ensures{ eval_set v s = not eval_conjunction_set v s }
    variant{ FS.cardinal s }
  = if FS.cardinal s > 1 then
      let x = FS.pick s in
        deMorgan v (FS.remove x s)
```

Other property that the `set` type of Why3 cannot guarantee is that the evaluation of a certain set with an added element `x` is equal to the evaluation of the element `x` with the evaluation of the rest of the set. Since this property is crucial to the proof, we defined this theory. Starting with an empty set, if we add an element to the set, the evaluation is equal to evaluation of the element `x` itself.

```
lemma evalemptyset_aux:
    forall v s x. (is_empty s) → eval_set v (FS.add x s) = ((not eval_positive v x) ||
```

This lemma was naturally processed. However, when the set is not empty, it is not possible to prove. Several tries to prove this lemma were made but unfortunately with no progress. Since this lemma is not strong, we decided to put it as an axiom in order to finish the proof:

```
axiom evalset_aux:
    forall v s x. (not (is_empty s)) → eval_set v (FS.add x s) = ((not eval_positive v
```

5 Conclusions and Observations

There is few presentations of this algorithm and when presented it is usual imperative. We present herein a new formulation of the algorithm. This Horn clauses are then used in the Horn algorithm.

The algorithm is presented as recursive functions, being clear, readable, rigorous and ideal to undergraduate logic courses. However, even if the presentation is clear and fits on one page, the implementation and verification are a bit more complex. We present them in a functional language, showing that given their properties the implementation and specification is less complex and related to the presentation of the algorithm.

A Full Implementation

```
module Types

  type ident

  (* Conjunctive Normal Form *)

  type pliteral =
      | LBottom
      | LVar ident

  type clause_cnf =
      | DLiteral pliteral
      | DNeg_cnf pliteral
```

```
      | DOr_cnf clause_cnf clause_cnf

  type formula_cnf =
      | FClause_cnf clause_cnf
      | FAnd_cnf formula_cnf formula_cnf


  (* Horn Formula *)

  type rightside =
      | RBottom
      | RTop
      | RVar ident

  type positive =
      | PLCBottom
      | PLCVar ident
      | PLCAnd positive positive

  type leftside =
      | LTop
      | LPos positive

  type basichornclause =
      | BImpl leftside rightside

  type hornclause =
      | HBasic basichornclause
      | HAnd hornclause hornclause

end


module Hornify

  use option.Option, int.Int, bool.Bool, Types, Valuation, Lemmas

  exception MoreThanOnePositive

  let convertLiteralToR (pl: pliteral) : (rightside)
```

```
= match pl with
    | LBottom → RBottom
    | LVar x → RVar x
  end

let convertLiteralToPLC (pl: pliteral) : (positive)
= match pl with
    | LBottom → PLCBottom
    | LVar x → PLCVar x
  end

let addLiterals (pl: pliteral) (nl: pliteral) (s: set) (p: option rightside) : (rs: se
= match pl with
    | LBottom → let rbottom = Some (RBottom) in
                match nl with
                  | LBottom → ((add (PLCBottom) s), rbottom)
                  | LVar x → ((add (PLCVar x) s), rbottom)
                end
    | LVar x → let rvar = Some (RVar x) in
                match nl with
                  | LBottom → ((add (PLCBottom) s), rvar)
                  | LVar x → ((add (PLCVar x) s), rvar)

                end
 end

let processCombination (pl: pliteral) (nl: pliteral) (s: set) (p: option rightside) :
  raises{ MoreThanOnePositive }
= match p with
    | None → addLiterals pl nl s p
    | Some _ → raise MoreThanOnePositive
  end


let rec hornify_aux (phi: clause_cnf) (s: set) (p: option rightside) : (rs: set, rp: o
  raises{ MoreThanOnePositive }
variant{ phi }
= match phi with
    | DOr_cnf (DLiteral _) (DLiteral _) → raise MoreThanOnePositive
    | DOr_cnf (DLiteral pl) (DNeg_cnf nl) | DOr_cnf (DNeg_cnf nl) (DLiteral pl) → (*
```

```
                          processCombination pl nl s p
    | DOr_cnf (DNeg_cnf nl1) (DNeg_cnf nl2) → ((add (convertLiteralToPLC nl1) (add (c
    | DOr_cnf (DOr_cnf phi1 phi2) (DLiteral pl) | DOr_cnf (DLiteral pl) (DOr_cnf phi1
                          match p with
                            | None → hornify_aux (DOr_cnf phi1 phi2) s (Some (conve
                            | Some _ → raise MoreThanOnePositive
                          end
    | DOr_cnf (DOr_cnf phi1 phi2) (DNeg_cnf nl) | DOr_cnf (DNeg_cnf nl) (DOr_cnf phi1
                         hornify_aux (DOr_cnf phi1 phi2) (add (convertLiteralToPLC n
    | DOr_cnf phi1 phi2 → let (s1,p1) = hornify_aux phi1 s p in
                          hornify_aux phi2 s1 p1
    | _ → absurd
  end


let conjunction (s: set): leftside
= let rec build (s: set)
   = if(is_empty s) then absurd else
     if((cardinal s) = 1) then (choose s) else
     PLCAnd (choose s) (build (remove (choose s) s)) in
  LPos (build s)


let getPositive (p: option rightside) : rightside
= match p with
    | None → RBottom
    | Some x → x
  end

let getBasicHorn (phi: clause_cnf) : basichornclause
  raises{ MoreThanOnePositive }
= match phi with
    | DLiteral (LVar x) → BImpl (LTop) (RVar x)
    | DLiteral (LBottom) → BImpl (LTop) (RBottom)
    | DNeg_cnf (LVar x) → BImpl (LPos (PLCVar x)) (RBottom)
    | DNeg_cnf (LBottom) → BImpl (LTop) (RTop)
    | DOr_cnf _ _ → let (s,p) = hornify_aux phi (empty ()) None in
                          BImpl (conjunction s) (getPositive p)
  end
```

```
  let rec hornify (phi: formula_cnf) : hornclause
    raises{ MoreThanOnePositive }
  = match phi with
      | FClause_cnf phi1 → HBasic (getBasicHorn phi1)
      | FAnd_cnf phi1 phi2 → HAnd (hornify phi1) (hornify phi2)
  end

end
```

B Evaluation Functions

```
module Valuation

  use option.Option, int.Int, bool.Bool, Types

  clone export set.SetApp with type elt = positive

  use import set.Fset as FS

  type valuation = ident → bool


  (* CNF Valuation *)

  let function eval_literal (v: valuation) (l: pliteral) : bool
  = match l with
      | LBottom → false
      | LVar x → v x
    end

  let rec function eval_disj_cnf (v: valuation) (phi: clause_cnf) : bool
  = match phi with
      | DLiteral l → eval_literal v l
      | DNeg_cnf l → not eval_literal v l
      | DOr_cnf phi1 phi2 → eval_disj_cnf v phi1 || eval_disj_cnf v phi2
    end

  let rec function eval_formula_cnf (v: valuation) (phi: formula_cnf) : bool
  = match phi with
      | FClause_cnf phi1 → eval_disj_cnf v phi1
```

```
    | FAnd_cnf phi1 phi2 → eval_formula_cnf v phi1 && eval_formula_cnf v phi2
  end


(* Horn Valuation *)

let function eval_rightside (v: valuation) (r: rightside) : bool
= match r with
    | RBottom → false
    | RTop → true
    | RVar x → v x
  end

let rec function eval_positive (v: valuation) (plc: positive) : bool
= match plc with
    | PLCBottom → false
    | PLCVar x → v x
    | PLCAnd phi1 phi2 → eval_positive v phi1 && (eval_positive v phi2)
  end

let rec function eval_leftside (v: valuation) (l: leftside) : bool
= match l with
    | LTop → true
    | LPos phi1 → eval_positive v phi1
  end

let rec function eval_basichornclause (v: valuation) (b: basichornclause) : bool
= match b with
    | BImpl left right → implb (eval_leftside v left) (eval_rightside v right)
  end

let rec function eval_hornclause (v: valuation) (h: hornclause) : bool
= match h with
    | HBasic h1 → eval_basichornclause v h1
    | HAnd h1 h2 → eval_hornclause v h1 && eval_hornclause v h2
  end


(* Sets e Option Valuation *)
```

```
  function eval_optionrightside (v: valuation) (p: option rightside) : bool
  = match p with
      | None → false
      | Some x → eval_rightside v x
    end

  let rec ghost function eval_set (v: valuation) (s: fset positive) : bool
  variant{FS.cardinal s}
  = if FS.is_empty s then
      false
    else
      if FS.cardinal s = 1 then
        not (eval_positive v (FS.pick s))
      else
        let x = FS.pick s in
           not (eval_positive v x) || eval_set v (FS.remove x s)

  let rec ghost function eval_conjunction_set (v: valuation) (s: fset positive) : bool
  variant{FS.cardinal s}
  = if(FS.is_empty s) then
      false
    else
      if FS.cardinal s = 1 then
        eval_positive v (FS.pick s)
      else
        let x = FS.pick s in
          eval_positive v x && eval_conjunction_set v (FS.remove x s)

  (* Domain and Codomain Valuation *)

  function eval_domain (v: valuation) (phi: clause_cnf) (s: set) (p: option rightside) :
  = eval_disj_cnf v phi || eval_set v s || eval_optionrightside v p

  function eval_codomain (v: valuation) (s: set) (p: option rightside) : bool
  = eval_set v s || eval_optionrightside v p

end
```

C Hornify Proof

```
module Hornify

  use option.Option, int.Int, bool.Bool, Types, Valuation, Lemmas

  exception MoreThanOnePositive

  (* Functions *)

  let convertLiteralToR (pl: pliteral) : (rightside)
    ensures{ forall v. eval_literal v pl = eval_rightside v result }
  = ..

  let convertLiteralToPLC (pl: pliteral) : (positive)
    ensures{ forall v. eval_literal v pl = eval_positive v result }
  = ..

  let addLiterals (pl: pliteral) (nl: pliteral) (s: set) (p: option rightside) : (rs: se
    requires{ p = None }
    ensures{ (not is_empty rs) }
    ensures { forall v. (eval_literal v pl || (not (eval_literal v nl)) || eval_set v s
  = ...

  let processCombination (pl: pliteral) (nl: pliteral) (s: set) (p: option rightside) :
    raises{ MoreThanOnePositive }
    ensures{ (not is_empty rs) }
    ensures { forall v. (eval_literal v pl || not (eval_literal v nl) || eval_set v s ||
  = ...


  let rec hornify_aux (phi: clause_cnf) (s: set) (p: option rightside) : (rs: set, rp: c
    requires{ exists x y. phi = DOr_cnf x y }
    ensures{ (not is_empty rs) }
    ensures{ forall v. eval_domain v phi s p = eval_codomain v rs rp }
    ensures{ forall v. eval_domain v phi s p = implb (eval_conjunction_set v rs) (eval_c
    raises{ MoreThanOnePositive }
    variant{ phi }
  = ...


  let conjunction (s: set): leftside
```

```
    requires{not is_empty s}
    ensures{forall v. eval_conjunction_set v s = eval_leftside v result }
  = let rec build (s: set)
     requires{not is_empty s}
     ensures{forall v. eval_conjunction_set v s = eval_positive v result}
     variant{cardinal s}
     = ...


  let getPositive (p: option rightside) : rightside
    ensures{forall v. eval_optionrightside v p = eval_rightside v result}
  = ...

  let getBasicHorn (phi: clause_cnf) : basichornclause
    ensures{ forall v. eval_disj_cnf v phi = eval_basichornclause v result }
    raises{ MoreThanOnePositive }
  = ...

  let rec hornify (phi: formula_cnf) : hornclause
    raises{ MoreThanOnePositive }
    ensures{forall v. eval_formula_cnf v phi = eval_hornclause v result}
    variant{ phi }
  = ...


end
```