Equivalences and Normal Forms

Logic and Programming Languages
Lecture #2
Equivalences

- Equivalences play a large role in building efficient algorithms for logical systems.
- How do we write programs that
  - Test equivalence?
  - Construct transformations where the output is equivalent to the input?

It is often easy to make mistakes, so how do we test such programs?
Writing a program

• Take an equivalence as a rule.
• Now apply it to every sub-term in a logical formula.
• At least two possibilities
  – Top Down
  – Bottom Up

• For example take the equivalences that
  – \( \sim(\sim x) = x \)
  – \( \sim(x \land y) = \sim x \lor \sim y \)
  – \( \sim(x \lor y) = \sim x \land \sim y \)
  – \( \sim T = F \)
  – \( \sim F = T \)
First a one-level program

not1 TruthP = AbsurdP
not1 AbsurdP = TruthP
not1 (NotP x) = x
not1 (AndP x y) = 
  OrP (not1 x) (not1 y)
not1 (OrP x y) = 
  AndP (not1 x) (not1 y)
not1 (ImpliesP x y) = 
  AndP x (not1 y)
not1 x = NotP x
Apply it bottom up

\[
nnf\ x = \\
\text{case } x \text{ of} \\
\text{AbsurdP } \rightarrow \text{AbsurdP} \\
\text{TruthP } \rightarrow \text{TruthP} \\
(\text{LetterP } x) \rightarrow \text{LetterP } x \\
(\text{AndP } x \ y) \rightarrow \text{AndP } (\text{nnf } x) (\text{nnf } y) \\
(\text{OrP } x \ y) \rightarrow \text{OrP } (\text{nnf } x) (\text{nnf } y) \\
(\text{ImpliesP } x \ y) \rightarrow \text{nnf}(\text{OrP } (\text{NotP } x) \ y) \\
(\text{NotP } x) \rightarrow \text{not1}(\text{nnf } x) \\
\]

- Note the recursive calls are “inside” the calls to the one-level transformer \text{not1}
Consider the equivalence

• \( A \rightarrow B \cong \sim A \lor B \)

implies \( \text{implies1 } x \ y = \text{OrP} \ (\text{not1 } x) \ y \)

• Now lets apply it top down
Top down

elimImplies x =
   case x of
      AbsurdP -> AbsurdP
      TruthP  -> TruthP
      (LetterP x) -> LetterP x
      (AndP x y) ->
         AndP (elimImplies x) (elimImplies y)
      (OrP x y) ->
         OrP (elimImplies x) (elimImplies y)
      (ImpliesP x y) -> elimImplies(implies1 x y)
      (NotP x) -> NotP (elimImplies x)

• Note the one-level call implies1 inside the recursive calls
Normal Forms

• Normal forms play a large role in many algorithms

• Some things to consider
  – What structural properties does a normal form have
  – Are their efficient data structures to capture normal forms
  – Are their efficient algorithms to compute them
CNF

• Conjunctive Normal Form plays a role in many algorithms
  – Tautology checking
  – SAT solving

• A term in CNF has all its conjunctions (AndP) at the top level. Each conjunct is a second level disjunct (OrP) and every disjunct is a literal

• A literal is TruthP, AbsurdP, (LetterP x), or NotP(LetterP x)
Example

- \((\lnot p_1 \lor \lnot p_4 \lor p_2) \land\)
- \((\lnot p_1 \lor \lnot p_4 \lor p_4) \land\)
- \((\lnot T \lor p_2) \land\)
- \((\lnot T \lor p_4)\)
We often represent terms in CNF as a list of list of literals. Writing this:

\[(\neg p_1 \lor \neg p_4 \lor p_2) \land (\neg p_1 \lor \neg p_4 \lor p_4) \land (\neg T \lor p_2) \land (\neg T \lor p_4)\]

As \[[[\neg p_1, \neg p_4, p_2], [[\neg p_1, \neg p_4, p_4], [\neg T, p_2], [\neg T, p_4]]\]

How do we represent \(T\) or \(F\)?
An algorithm

Coble gives an algorithm in 4 steps (or passes)

1. Eliminate implication
2. Push negations inside so they are only on literals
3. Apply the distributive laws
   1. \( A \lor (B \land C) \equiv (A \lor B) \land (A \lor C) \)
   2. \( (B \land C) \lor A \equiv (B \lor A) \land (C \lor A) \)
   3. \( (A \land B) \lor (C \land D) \equiv (A \lor C) \land (A \lor D) \land (B \lor C) \land (B \lor D) \)
4. Simplify the results

1. We will write a Haskell Program
Several passes

cnf3 :: Eq n => Prop n -> [[Prop n]]
cnf3 x = (simple .
    flatten .
    pushDisj .
    nnf . elimImplies) x

cnf4 :: Prop t -> Prop t
cnf4 = pushDisj . nnf . elimImplies

• Note the use of a function for each pass and the change in representation \([[[\text{Prop n}]])\) (using flatten) in the definition of cnf3 for CNF formula.
\[
\text{pushDisj } x = \begin{案件}
\text{OfOrP } x y \rightarrow \text{case } (\text{pushDisj } x, \text{pushDisj } y) \text{ of } \\
(\text{AndP } a b, \text{AndP } c d) \rightarrow \\
\text{AndP } (\text{pushDisj } (\text{OrP } a c)) \\
(\text{AndP } (\text{pushDisj } (\text{OrP } a d)) \\
(\text{AndP } (\text{pushDisj } (\text{OrP } b c)) \\
(\text{pushDisj } (\text{OrP } b d))) \\
(a, \text{AndP } b c) \rightarrow \\
\text{AndP } (\text{pushDisj } (\text{OrP } a b)) \\
(\text{pushDisj } (\text{OrP } a c)) \\
(\text{AndP } b c, a) \rightarrow \\
\text{AndP } (\text{pushDisj } (\text{OrP } b a)) \\
(\text{pushDisj } (\text{OrP } c a)) \\
(x, y) \rightarrow \text{OrP } x y \\
\text{AbsurdP } \rightarrow \text{AbsurdP} \\
\text{TruthP } \rightarrow \text{TruthP} \\
(\text{LetterP } x) \rightarrow \text{LetterP } x \\
(\text{AndP } x y) \rightarrow \text{AndP } (\text{pushDisj } x) (\text{pushDisj } y) \\
(\text{ImpliesP } x y) \rightarrow \text{pushDisj}(\text{OrP } (\text{NotP } x) y) \\
(\text{NotP } x) \rightarrow \text{NotP } (\text{pushDisj } x)
\end{案件}
\]
Change representation

-- assumes all disj’s are pushed inside
-- so only literals appear inside OrP
flatten:: Prop n -> [[Prop n]]
flatten (AndP x y) = flatten x ++ flatten y
flatten (OrP x y) = [collect [x,y]]
    where collect [] = []
        collect (OrP x y : zs) =
            collect (x:y:zs)
        collect (z:zs) = z : collect zs
flatten x = [[x]]
Simplify

• Simplify (or remove disjunctions) that are always true
• \([p_1, p_3, \sim p_1]\) \rightarrow \text{remove}
• \([p_1, T]\) \rightarrow \text{remove}
• \([p_1, p_2, p_3]\) \rightarrow \text{remove if there is another disjunction that subsumes it like } [p_1, p_3]\)
simple :: Eq n => [[Prop n]] -> [[Prop n]]
simple [] = []
simple (x:xs)
    | elem TruthP x = simple xs
    | conjugatePair x = simple xs
    | subsumes xs x = simple xs
    | otherwise = x : simple xs
A principled approach

• Study the equivalence
  
  \(- (A \land B) \lor (C \land D) \equiv (A \lor C) \land (A \lor D) \land (B \lor C) \land (B \lor D)\)

• Think of each disjunct as a list, then the result can be computed like this

  \([A,B] \lor [C,D] == \land [ x \lor y | x \leftarrow [A,B], y \leftarrow [C,D]]\)

• Take as input 2 lists of disjunctions, and apply the cross product rule
Representation

• process :: [Prop a] -> [[Prop a]]

• Think of the input as a list of disjunctions, so we want to take the cross product of all these disjunctions.

• If there are n-disjunctions then we’ll have \( n^n \) literals in each resulting inner disjunction

• We’ll also have the product of the size of each disjunction as the number of conjunctions.
Applying Equivalences

• As we process the list of disjunctions we apply equivalences as we go.
Positive cases

process [] = [[]]

process (p:ps) =

  case p of
    (AbsurdP) -> map (AbsurdP:) (process ps)
    (TruthP)  -> map (TruthP:) (process ps)
    (LetterP _) -> map (p:) (process ps)
    (AndP x y) -> process (x:ps) ++
                 process (y:ps)
    (OrP x y)  -> process (x : y : ps)
    (ImpliesP x y) -> process(NotP x : y : ps)
Negative cases

process [] = [[]]
process (p:ps) =
  case p of
    . . .
    (NotP z) ->
      case z of
        (AbsurdP) -> map (TruthP:) (process ps)
        (TruthP)  -> map (AbsurdP:) (process ps)
        (LetterP _) -> map (p:) (process ps)
        (AndP x y) -> process (NotP x : NotP y : ps)
        (OrP x y) -> process (NotP x:ps) ++
                     process (NotP y:ps)
        (ImpliesP x y) -> process (x:ps) ++
                          process (NotP y:ps)
        (NotP p2)      -> process (p2:ps)
Observations

• How big can a answer get? What is the complexity?
• Many of the cases are very similar.
• What are the three cases?
• Can we exploit this to write a shorter program?