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
  - ~(~x) = x
  - ~(x /\ y) = ~x \/ ~y
  - ~(x \/ y) = ~x /\ ~ y
  - ~T = F
  - ~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 =
  case x of
    AbsurdP -> AbsurdP
    TruthP  -> TruthP
    (LetterP x) -> LetterP x
    (AndP x y) -> AndP (nnf x) (nnf y)
    (OrP x y) -> OrP (nnf x) (nnf y)
    (ImpliesP x y) -> nnf(OrP (NotP x) y)
    (NotP x) -> not1(nnf x)
```

- Note the recursive calls are “inside” the calls to the one-level transformer not1

Consider the equivalence

- $A \to B \cong \sim A \lor B$

```
implies1 x y = OrP (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

- $(\sim p1 \vee \sim p4 \vee p2) \wedge$
- $(\sim p1 \vee \sim p4 \vee p4) \wedge$
- $(\sim T \vee p2) \wedge$
- $(\sim T \vee p4)$

[[Literal]]

- We often represent terms in CNF as a list of list of literals. Writing this

(~p1 \/ ~p4 \/ p2) /\
 (~p1 \/ ~p4 \/ p4) /\
 (~T \/ p2) /\
(~T \/ p4)

- As [[~p1,~p4,p2],[~p1,~p4,p4],[~T,p2],[~T,p4]]
- 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. $A \vee (B \wedge C) \cong (A \vee B) \wedge (A \vee C)$
    2. $(B \wedge C) \vee A \cong (B \vee A) \wedge (C \vee A)$
    3. $(A \wedge B) \vee (C \wedge D) \cong (A \vee C) \wedge (A \vee D) \wedge (B \vee C) \wedge (B \vee D)$
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 [[Prop n]] (using flatten) in the definition of cnf3 for CNF formula.

$$A \vee (B \wedge C) \cong (A \vee B) \wedge (A \vee C)$$

$$(B \wedge C) \vee A \cong (B \vee A) \wedge (C \vee A)$$

$$(A \wedge B) \vee (C \wedge D) \cong (A \vee C) \wedge (A \vee D) \wedge (B \vee C) \wedge (B \vee D)$$

```
pushDisj x =   case x of
    OrP x y -> case (pushDisj x,pushDisj y) of
      (AndP a b,AndP c d) ->
         AndP (pushDisj (OrP a c))
              (AndP (pushDisj (OrP a d))
                    (AndP (pushDisj (OrP b c))
                    (pushDisj (OrP b d))))
      (a,AndP b c) ->
         AndP (pushDisj (OrP a b))
              (pushDisj (OrP a c))
      (AndP b c,a) ->
         AndP (pushDisj (OrP b a))
              (pushDisj (OrP c a))
      (x,y) -> OrP x y
    AbsurdP -> AbsurdP
    TruthP  -> TruthP
    (LetterP x) -> LetterP x
    (AndP x y) -> AndP (pushDisj x) (pushDisj y)
    (ImpliesP x y) -> pushDisj(OrP (NotP x) y)
    (NotP x) -> NotP (pushDisj x)
```

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
- [p1, p3 , ~p1] → remove
- [p1, T] → remove
- [p1,p2, p3] → remove if there is another disjunction that subsumes it like [p1,p3]

```haskell
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

$-(A \wedge B) \vee (C \wedge D) \cong$

$(A \vee C) \wedge (A \vee D) \wedge (B \vee C) \wedge (B \vee D)$

- Think of each disjunct as a list, then the result can be computed like this
- $[A,B] \bigvee [C,D] == \bigwedge [\, x \bigvee y \mid x <- [A,B],\, y <- [C,D]]$

- 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 nn 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?