Logic roundup: SAT CNF Tseytin DPLL

In the first part of this tutorial you will combine the code from CL tutorial 6 and FP tutorial 6, and implement and test the Tseytin transformation.

The code in CL7.hs includes material from both earlier tutorials. In particular, we have the following type declarations for our two languages – `Wff`s and `Form`s.

```
data Wff a = V a
           | T
           | F
           | Not (Wff a)
           | Wff a :|: Wff a
           | Wff a :&: Wff a
           | Wff a :->: Wff a
           | Wff a :<->: Wff a
```

```
data   Literal a = N a | P a
                 deriving (Ord, Eq, Show)

newtype Clause a = Or  [Literal a]
                 deriving (Ord, Eq, Show)

newtype   Form a = And [Clause a]
                 deriving (Eq, Show)
```

The code also includes a version of our dpll algorithm:

```
(<<) :: Ord a => [Clause a] -> Literal a -> [Clause a]
cs << x = [ Or (delete (neg x) ys) | Or ys <- cs, not $ x `elem` ys ]

dpll :: Eq a => Form a -> [[Literal a]]
dpll (And css) =
  let models [] = [[]]
      models cs = case prioritise cs of
        Or []          -> []   -- empty clause: no models
        Or [lit]       -> [    lit : m | m <- models (cs << lit)] -- unit clause
        Or (lit : _)   -> [    lit : m | m <- models (cs << lit)]
                          ++
                          [neg lit : m | m <- models (cs << neg lit)]
  in models css
  where prioritise = minimumBy (comparing (\(Or lits) -> length lits))
```

1. Your first task is to provide a function

   `wffToForm :: Eq a => Wff a -> Form a`

   fiable (use `satisfiable`) iff the corresponding form, produced by `wffToForm`, has a model (use `dpll`).

2. Then write a test to check that a Wff is satis-

   `prop_form_equiv :: Wff Atom -> Bool`

3. The Tseytin procedure introduces a variable for each subformula – or implementation of the Tseytin transform has the following type `tseytinToForm :: Ord a => Wff a -> Form (Wff a)`

   For each compound subformula (one that includes a connective, call the connective $*$) the transform `tt` includes a CNF for the expression $r \leftrightarrow a(*)b$.
   The code given shows the pattern for `:&:`.
   $r \leftrightarrow a \wedge b \equiv (r \vee \neg a \vee \neg b) \wedge (\neg r \vee a) \wedge (\neg r \vee b)$
   We include this and then add `tt a ++ tt b`, the transforms of its subexpressions. .

   ```
   tt r@(a :&: b)   =
     [ Or[P r, N a, N b], Or[N r, P a]
     , Or[N r, P b]] ++ tt a ++ tt b

   tt r@(a :|: b)   = undefined
   tt r@(a :->:  b) = undefined
   tt r@(a :<->: b) = undefined
   ```

   Follow this pattern, and use CNFs you have derived earlier to replace each instance of undefined.

4. Now write a test `prop_tseytin_equiv :: Wff Atom -> Bool` to check that a Wff is satisfiable (using the function `satisfiable`) iff its Tseytin transform has a model (found using `dpll`).

5. To complete this part of the exercise, define a reasonable function `size :: Form a -> Int` and use `quickCheck` to find a Wff for which `size (wffToForm wff) > 100 * size (tseytinToForm wff)`.

1 Counting models

If we want to go beyond yes/no questions, it is natural to ask, *How many ...?* We are interested in sets, so we will ask how many elements there are in a set of states defined by a logial formula. So, the question is, *How many states satisfy a given formula?*

This exercise need not involve any Haskell. However, you may well find Haskell is a useful tool for checking your answers. Not that our Haskell type Atom is defined to be `data Atom = A|B|C|D|W|X|Y|Z` – to avoid clashes with other uses of `TFPN`, for example. You an use `String` or `Char` as your atoms.

6. This question concerns the 256 possible truth valuations of the following eight propositional letters A, B, C, D, E, F, G, H. For each of the following expressions, say how many of the 256 valuations satisfy the expression, and briefly explain your reasoning. For example, the expression D is satisfied by 128 of the 256 valuations, Since for each valuation that makes D true there is a matching valuation that make D false.

   (a) $A \vee B$ (b) $A \to B$ (b) $(A \to B) \wedge (C \to D)$ (d) $A \oplus B$

7. For the following questions, use the arrow rule. This is explained in the first video of these four. For more details see overleaf. If you still have problems understanding this let me know and I will post more video examples.

   (a) $(A \to B) \wedge (B \to C)$

   (b) $(A \to B) \wedge (B \to C) \wedge (C \to D)$

   (c) $(A \to B) \wedge (C \to B)$

   (d) $(A \to B) \wedge (B \to C) \wedge (D \to B)$

   (e) $(A \to B) \wedge (A \to C)$

   (f) $(A \to B) \wedge (B \to C) \wedge (D \to C)$

   (g) $(A \to B) \wedge (B \to C) \wedge (C \to D) \wedge (A \to E) \wedge (E \to D)$

   (h) $(\neg A \to B) \wedge (B \to \neg C) \wedge (C \to D) \wedge (A \to \neg E) \wedge (E \to D)$

   (i) $(A \to B) \wedge (B \to C) \wedge (C \to D) \wedge (A \to E) \wedge (E \to D) \wedge (D \to F)$

   (j) $(A \to B) \wedge (B \to C) \wedge (C \to D) \wedge (A \to \neg B) \wedge (\neg B \to D) \wedge (F \to A)$

   (k) $(A \to B) \wedge (\neg B \to C) \wedge (C \to D) \wedge (\neg A \to E) \wedge (E \to D) \wedge (F \to A)$

   (l) $(A \to B) \wedge (B \to A) \wedge (C \to D) \wedge (D \to E) \wedge (E \to F) \wedge (F \to G) \wedge (G \to H)$

   (m) $(A \to B) \wedge (B \to A) \wedge (C \to D) \wedge (D \to C) \wedge (E \to F) \wedge (F \to G) \wedge (G \to H)$

   (n) $(H \to A) \wedge (A \to B \wedge C) \wedge (B \vee C \to D) \wedge (A \to E) \wedge (E \to F) \wedge (F \to G) \wedge (G \to H)$

   You will find more examples in past papers for the past few years.

The arrow rule

The arrow rule applies to 2-SAT problems. It lets us identify and count the satisfying valuations for a 2-SAT problem. We begin by converting each clause with two literals to two implications, both equivalent to the clause – each implication is the contrapositive of the other.

In mathematical terms, the arrows generate a preorder: if there is an arrow $A \to B$ then $A \leq B$.

To apply the arrow rule, make a diagram with a directed graph whose nodes are all the literals, together with $\top$ and $\bot$, and whose edges are arrows representing the implications we have identified, with additional arrows from $\bot$ to each minimal node and from each maximal node to $\top$.

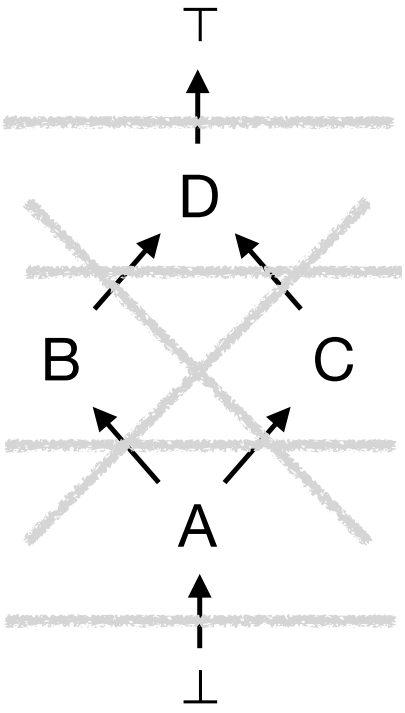
A *legal cut* of this diagram is a line that must separate $\top$ from $\bot$, and each literal from its negation, and such that each arrow that crosses the line goes from the side of $\bot$ (below) to the side of $\top$ (above).

The idea is that a valuation $V :: \text{Atom} -> \text{Bool}$ satisfies every one of the implications iff it preserves the ordering: if $A \to B$ then $VA \leq VB$.

For example,

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

gives the diagram shown here. Any legal cut corresponds to a valuation, making the literals above the line (on the same side as $\top$) true, and those below the line (on the same side as $\bot$) false. These are exactly the valuations that satisfy all of the original clauses.



We could also draw the opposite graph, with negated atoms and reversed arrows. We would arrive a t the same valuations of the atoms. In this example, the positive and negative occurrences of each atom lie in separate components of the graph, and it suffices to consider one component. In some examples the graph does not split in this way and we have to consider the entire graph, with each arrow and its contrapositive.

Here, there are six satisfying valuations corresponding to the six grey lines shown; these are the legal cuts for this diagram.

This tutorial exercise sheet was written by Michael Fourman. please send comments to `michael@ed.ac.uk`