Propositional Logic and SAT Solving

Jan van Eijck

Specification and Testing, Week 3, 2014

Overview

- Propositional Logic: Language and Semantics
- Testing Ground for Parsing Techniques
- Random Testing Techniques: Random Datatype Generation
- Transformation to Normal Form
- Pre- and Postconditions of Stages in the Transformation
- Clausal form — Connection With SAT Solving

Propositional Logic: Language

$$\varphi ::= p \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid (\varphi \leftrightarrow \varphi)$$

This looks simple, but it is very expressive.

Many problems in computer science can be phrased as satisfiability problems for Boolean formulas.

Semantics

- What do the Boolean formulas mean?
- Better question (better, because more precise): when do Boolean formulas have the same meanings?
- Valuations: functions from proposition letters to boolean values.
- Definition of set of proposition letters of a formula.
- A valuation V satisfies a formula φ if giving the proposition letters the values specified by V yields 'true' for the whole formula.
- The 'truth table method' is a method for finding the satisfying valuations.
- Two formulas have the same meaning if they have the same satisfying valuations.

Tautologies, Contradictions, Satisfiable Formulas

- **tautology**: a formula that is satisfied by **every** valuation.
- **contradiction**: a formula that is satisfied by **no** valuation.
- **satisfiable formula** a formula that is satisfied by **some** valuation.
- If φ is a tautology, then $\neg\varphi$ is a contradiction.
- If φ is a contradiction, then $\neg\varphi$ is a tautology.
- If φ is not satisfiable, then φ is a contradiction.
- If φ is not a contradiction, then φ is satisfiable.

Implication versus 'if then else'

- An implication is a formula of the form $\varphi_1 \to \varphi_2$.
- This is not the same as an 'if then else' in Java or Ruby.
- The 'if then else' syntax is:

  `if <expression> then <statement> else <statement>`,

  where `expression` is indeed a Boolean expression, but `statement` is a program instruction, which in general is not a Boolean expression.
- $\varphi_1 \to \varphi_2$ is a Boolean expression, and it is equivalent to $\neg\varphi_1 \vee \varphi_2$, and also to $\neg(\varphi_1 \wedge \neg\varphi_2)$.

Logical Consequence

- From φ_1 it follows logically that φ_2. Notation $\varphi_1 \models \varphi_2$.
- Defined as: every valuation that satisfies φ_1 also satisfies φ_2.
- Note: φ_1 has φ_2 as logical consequence if and only if $\varphi_1 \to \varphi_2$ is a tautology.
- Useful abbreviation for 'if and only if': iff.

Complexity

- It is unknown how complex the satisfiability problem for Boolean formulas is. The best known solution methods are in nondeterministic polynomial time (NP). This is widely believed to be more complex than polynomial time (P), but the proof of P $\neq$ NP is an open problem.
- Nobody believes it is possible to check the satisfiability of a propositional formula (Boolean formula) in polynomial time.
- If φ has n proposition letters, there are 2^n relevant valuations, so the truth table for φ will have 2^n lines. No general method is known that works faster than checking possibilities one by one.
- Given a candidate valuation for a Boolean formula, it can be checked in polynomial time whether that valuation satisfies the formula.

Normal Forms

CNF or conjunctive normal forms are conjunctions of clauses, where a clause is a disjunction of literals, where a literal is a proposition letter or its negation.

Syntactic definition:

$$\begin{aligned} L &::= p \mid \neg p \\ D &::= L \mid L \vee D \\ C &::= D \mid D \wedge C \end{aligned}$$

DNF or disjunctive normal form: similar definition, left to you.

Why is CNF useful? CNF formulas can easily be tested for validity. How?

Translating into CNF, first step

First step: translate formulas into equivalent formulas that are arrow-free: formulas without $\leftrightarrow$ and $\rightarrow$ operators.

- Use the equivalence between $p \rightarrow q$ and $\neg p \vee q$ to get rid of $\rightarrow$ symbols.
- Use the equivalence of $p \leftrightarrow q$ and $(\neg p \vee q) \wedge (p \vee \neg q)$, to get rid of $\leftrightarrow$ symbols.

Pseudo-code on next page:

Translating into CNF, first step in pseudocode

function ArrowFree (φ):
/* precondition: φ is a formula. */
/* postcondition: ArrowFree (φ) returns arrow free version of φ */
begin function
case
    φ is a literal: **return** φ
    φ is $\neg\psi$: **return** $\neg$ ArrowFree (ψ)
    φ is $\psi_1 \wedge \psi_2$: **return** ArrowFree (ψ_1) $\wedge$ ArrowFree (ψ_2)
    φ is $\psi_1 \vee \psi_2$: **return** ArrowFree (ψ_1) $\vee$ ArrowFree (ψ_2)
    φ is $\psi_1 \rightarrow \psi_2$: **return** ArrowFree ($(\neg\psi_1) \vee \psi_2$)
    φ is $\psi_1 \leftrightarrow \psi_2$: **return** ArrowFree ($((\neg\psi_1) \vee \psi_2) \wedge (\psi_1 \vee (\neg\psi_2))$)
end case
end function

Translating into CNF, second step

function NNF (φ):
/* precondition: φ is arrow-free. */
/* postcondition: NNF (φ) returns NNF of φ */
begin function
case
    φ is a literal: **return** φ
    φ is $\neg\neg\psi$: **return** NNF (ψ)
    φ is $\psi_1 \wedge \psi_2$: **return** NNF (ψ_1) $\wedge$ NNF (ψ_2)
    φ is $\psi_1 \vee \psi_2$: **return** NNF (ψ_1) $\vee$ NNF (ψ_2)
    φ is $\neg(\psi_1 \wedge \psi_2)$: **return** NNF ($\neg\psi_1$) $\vee$ NNF ($\neg\psi_2$)
    φ is $\neg(\psi_1 \vee \psi_2)$: **return** NNF ($\neg\psi_1$) $\wedge$ NNF ($\neg\psi_2$)
end case
end function

Translating into CNF, third step

function CNF (φ):
/* precondition: φ is arrow-free and in NNF. */
/* postcondition: CNF (φ) returns CNF of φ */
begin function
case
    φ is a literal: **return** φ
    φ is $\psi_1 \wedge \psi_2$: **return** CNF (ψ_1) $\wedge$ CNF (ψ_2)
    φ is $\psi_1 \vee \psi_2$: **return** DIST (CNF (ψ_1), CNF (ψ_2))
end case
end function

Translating into CNF, auxiliary step

function DIST (φ_1, φ_2):
/* precondition: φ_1, φ_2 are in CNF. */
/* postcondition: DIST (φ_1, φ_2) returns CNF of $\varphi_1 \vee \varphi_2$ */
begin function
case
φ_1 is $\psi_{11} \wedge \psi_{12}$: **return** DIST $(\psi_{11},\varphi_2) \wedge$ DIST (ψ_{12},φ_2)
φ_2 is $\psi_{21} \wedge \psi_{22}$: **return** DIST $(\varphi_1,\psi_{21}) \wedge$ DIST (φ_1,ψ_{22})
otherwise: **return** $\varphi_1 \vee \varphi_2$
end case
end function

First case uses equivalence of $(p \wedge q) \vee r$ and $(p \vee r) \wedge (q \vee r)$.
Second case uses equivalence of $p \vee (q \wedge r)$ and $(p \vee q) \wedge (p \vee r)$.

Note the Pre- and Postconditions

- φ is a formula.
- ArrowFree (φ) returns arrow free version of φ.
- φ is arrow-free.
- NNF (φ) returns NNF of φ.
- φ is arrow-free and in NNF.
- CNF (φ) returns CNF of φ.

Importance for Testing

- Conditions in programming languages use Boolean formulas.
- Simplifying these conditions transforms programs into equivalent programs that are easier to understand.
- Propositional logic is at the core of more expressive logics that are used for specification.
- "The Haskell Road" Chapter 2 has all the details. Also, see workshop for real life examples.
- SAT solvers use Boolean formulas in CNF. SAT solvers are important for the implementation of test program for formal specifications. Much more about this in the rest of the course.
- If you need to read up on propositional logic, please consult http://www.logicinaction.org/docs/ch2.pdf.

Representing Propositional Logic in Haskell

```
module Week3 where

import Data.List
import Data.Char
import System.Random

type Name = Int

data Form = Prop Name
          | Neg  Form
          | Cnj [Form]
          | Dsj [Form]
          | Impl Form Form
          | Equiv Form Form
          deriving Eq
```

Displaying Formulas

```
instance Show Form where
  show (Prop x)    = show x
  show (Neg f)     = '-' : show f
  show (Cnj fs)      = "*(" ++ showLst fs ++ ")"
  show (Dsj fs)      = "+(" ++ showLst fs ++ ")"
  show (Impl f1 f2)  = "(" ++ show f1 ++ "==>"
                             ++ show f2 ++ ")"
  show (Equiv f1 f2)  = "(" ++ show f1 ++ "<=>"
                             ++ show f2 ++ ")"

showLst,showRest :: [Form] -> String
showLst [] = ""
showLst (f:fs) = show f ++ showRest fs
showRest [] = ""
showRest (f:fs) = ' ': show f ++ showRest fs
```

Example Formulas

```
p = Prop 1
q = Prop 2
r = Prop 3

form1 = Equiv (Impl p q) (Impl (Neg q) (Neg p))
form2 = Equiv (Impl p q) (Impl (Neg p) (Neg q))
form3 = Impl (Cnj [Impl p q, Impl q r]) (Impl p r)
```

Proposition Letters Occurring in a Formula

```
propNames :: Form -> [Name]
propNames = sort.nub.pnames where
  pnames (Prop name) = [name]
  pnames (Neg f)  = pnames f
  pnames (Cnj fs) = concat (map pnames fs)
  pnames (Dsj fs) = concat (map pnames fs)
  pnames (Impl f1 f2) = concat (map pnames [f1,f2])
  pnames (Equiv f1 f2) =
          concat (map pnames [f1,f2])
```

Valuations

```
type Valuation = [(Name,Bool)]

-- all possible valuations for list of prop letters
genVals :: [Name] -> [Valuation]
genVals [] = [[]]
genVals (name:names) =
  map ((name,True) :) (genVals names)
  ++ map ((name,False):) (genVals names)

-- generate all possible valuations for a formula
allVals :: Form -> [Valuation]
allVals = genVals . propNames
```

Note that an exponential blowup takes place here. If a propositional formula has n variables, there are 2^n different valuations for that formula. To see what that means, just look at this:

```
*PL> map (2^) [1..20]
[2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,
 16384,32768,65536,131072,262144,524288,1048576]
```

The algorithm used in the definition of `genVals` is not feasible. The exponential blow-up reveals itself in the fact that there are two recursive calls to `genVals` in the definition.

Evaluation of formulas.

```
eval :: Valuation -> Form -> Bool
eval [] (Prop c)    = error ("no info: " ++ show c)
eval ((i,b):xs) (Prop c)
     | c == i    = b
     | otherwise = eval xs (Prop c)
eval xs (Neg f)  = not (eval xs f)
eval xs (Cnj fs) = all (eval xs) fs
eval xs (Dsj fs) = any (eval xs) fs
eval xs (Impl f1 f2) =
     not (eval xs f1) || eval xs f2
eval xs (Equiv f1 f2) = eval xs f1 == eval xs f2
```

Note that the evaluation algorithm is feasible: `eval` is called only once for each subformula. Finding a satisfying valuation for a formula is hard, but checking whether a valuation satisfies a formula is easy.

Satisfiability, Logical Entailment, Equivalence

A formula is satisfiable if some valuation makes it true. We know what the valuations of a formula `f` are. These are given by `allVals f`. We also know how to express that a valuation `v` makes a formula `f` true: `eval v f`. This gives:

```
satisfiable :: Form -> Bool
satisfiable f = any (\ v -> eval v f) (allVals f)
```

Lab Exercise for this week:

Write implementations of contradiction, tautology, logical entailment, logical equivalence, and test them.

Parsing Propositional Formulas

The process of converting an input string to a list of tokens is called lexical scanning. The following is a relevant list of tokens for propositional formulas.

```
data Token
      = TokenNeg
      | TokenCnj
      | TokenDsj
      | TokenImpl
      | TokenEquiv
      | TokenInt Int
      | TokenOP
      | TokenCP
 deriving (Show,Eq)
```

The lexer converts a string to a list of tokens.

```
lexer :: String -> [Token]
lexer [] = []
lexer (c:cs) | isSpace c = lexer cs
             | isDigit c = lexNum (c:cs)
lexer ('(':cs) = TokenOP : lexer cs
lexer (')':cs) = TokenCP : lexer cs
lexer ('*':cs) = TokenCnj : lexer cs
lexer ('+':cs) = TokenDsj : lexer cs
lexer ('-':cs) = TokenNeg : lexer cs
lexer ('=':'=':'>':cs) = TokenImpl : lexer cs
lexer ('<':'=':'>':cs) = TokenEquiv : lexer cs
lexer (x:_) = error ("unknown token: " ++ [x])
```

Read an integer and convert it into a structured token of the form `TokenInt i`.

```
lexNum cs = TokenInt (read num) : lexer rest
     where (num,rest) = span isDigit cs
```

Example use:

```
*PS> lexer "*(2 3 -4 +("
[TokenCnj,TokenOP,TokenInt 2,TokenInt 3,TokenNeg,
 TokenInt 4,TokenDsj,TokenOP]
```

A parser for token type `a` that constructs a datatype `b` has the following type:

```
type Parser a b = [a] -> [(b,[a])]
```

The parser constructs a list of tuples `(b,[a])` from an initial segment of a token string `[a]`. The remainder list in the second element of the

result is the list of tokens that were not used in the construction of the datatype.

If the output list is empty, the parse has not succeeded. If the output list more than one element, the token list was ambiguous.

The simplest possible parser is the parser that succeeds immediately, while consuming no input:

```
succeed :: b -> Parser a b
succeed x xs = [(x,xs)]
```

Parsing a formula.

```
parseForm :: Parser Token Form
parseForm (TokenInt x: tokens) = [(Prop x,tokens)]
parseForm (TokenNeg : tokens) =
  [ (Neg f, rest) | (f,rest) <- parseForm tokens ]
parseForm (TokenCnj : TokenOP : tokens) =
  [ (Cnj fs, rest) | (fs,rest) <- parseForms tokens ]
parseForm (TokenDsj : TokenOP : tokens) =
  [ (Dsj fs, rest) | (fs,rest) <- parseForms tokens ]
parseForm (TokenOP : tokens) =
  [ (Impl f1 f2, rest) | (f1,ys) <- parseForm tokens,
                         (f2,rest) <- parseImpl ys ]
   ++
  [ (Equiv f1 f2, rest) | (f1,ys) <- parseForm tokens,
                          (f2,rest) <- parseEquiv ys ]
parseForm tokens = []
```

Parsing a list of formulas: success if a closing parenthesis is encountered. This uses the `succeed` parser above.

```
parseForms :: Parser Token [Form]
parseForms (TokenCP : tokens) = succeed [] tokens
parseForms tokens =
   [(f:fs, rest) | (f,ys) <- parseForm tokens,
                   (fs,rest) <- parseForms ys ]
```

Parsing implications and equivalences uses separate functions, for these constructions have infix operators.

```
parseImpl :: Parser Token Form
parseImpl (TokenImpl : tokens) =
  [ (f,ys) | (f,y:ys) <- parseForm tokens, y == TokenCP ]
parseImpl tokens = []

parseEquiv :: Parser Token Form
parseEquiv (TokenEquiv : tokens) =
  [ (f,ys) | (f,y:ys) <- parseForm tokens, y == TokenCP ]
parseEquiv tokens = []
```

The parse function.

```
parse :: String -> [Form]
parse s = [ f | (f,_) <- parseForm (lexer s) ]
```

This gives:

```
*PL> parse "*(1 +(2 -3))"
[*(1 +(2 -3))]
*PL> parse "*(1 +(2 -3)"
[]
*PL> parse "*(1 +(2 -3))))"
[*(1 +(2 -3))]
*PL> parseForm (lexer "*(1 +(2 -3))))")
[(*(1 +(2 -3)),[TokenCP,TokenCP])]
```

Exercise 1 Write an appropriate test for the parse function. Hint: use the `show` function for formulas.

Random Formula Generation for Testing

This section gives code for random formula generation, to be used for automated testing of formula properties.

Getting a random non-negative integer in the range $\{0, \ldots, n\}$.

```
getRandomInt :: Int -> IO Int
getRandomInt n = getStdRandom (randomR (0,n))
```

Getting a random formula. There is a parameter for the average complexity of the formula that can be adjusted. Here we set the maximum average complexity to 4.

```
getRandomF :: IO Form
getRandomF = do d <- getRandomInt 4
                getRndF d
```

Formulas of complexity 0 are proposition letters. We generate positive indices for them in the range $\{1, \ldots, 21\}$ but this can be adjusted.

```
getRndF :: Int -> IO Form
getRndF 0 = do m <- getRandomInt 20
               return (Prop (m+1))
```

Formulas of complexity > 0 are equally likely to be of one of the six possible syntactic forms. If they consist of a combinator with formula argument(s), the embedded formulas have lower complexity.

```
getRndF d = do n <- getRandomInt 5
               case n of
                  0 -> do m <- getRandomInt 20
                          return (Prop (m+1))
                  1 -> do f <- getRndF (d-1)
                          return (Neg f)
                  2 -> do m  <- getRandomInt 5
                          fs <- getRndFs (d-1) m
                          return (Cnj fs)
                  3 -> do m  <- getRandomInt 5
                          fs <- getRndFs (d-1) m
                          return (Dsj fs)
                  4 -> do f <- getRndF (d-1)
                          g <- getRndF (d-1)
                          return (Impl f g)
                  5 -> do f <- getRndF (d-1)
                          g <- getRndF (d-1)
                          return (Equiv f g)
```

Get a list of n random formulas:

```
getRandomFs :: Int ->  IO [Form]
getRandomFs n = do d <- getRandomInt 3
                   getRndFs d n
```

Get a list of n random formulas of complexity d:

```
getRndFs :: Int -> Int -> IO [Form]
getRndFs _ 0 = return []
getRndFs d n = do f <- getRndF d
                  fs <- getRndFs d (n-1)
                  return (f:fs)
```

Try this out

```
getRndF 0
getRndF 3
getRndF 10
getRndF 20
```

Test Automation

Here is how to test with this. The integer argument n stores the total number of tests.

```
test :: Int -> (Form -> Bool) -> [Form] -> IO ()
test n _ [] = print (show n ++ " tests passed")
test n p (f:fs) =
  if p f
  then do print ("pass on:" ++ show f)
          test n p fs
  else error ("failed test on:" ++ show f)
```

Test n formulas, using property p.

```
testForms :: Int -> (Form -> Bool) -> IO ()
testForms n p = do
  fs <- getRandomFs n
  test n p fs
```

Code for a sequence of random tests of the parser:

```
testParser = testForms 100
   (\ f -> let [g] = parse (show f) in
           show f == show g)
```

Conjunctive Normal Form

CNF and DNF are important for automated theorem proving: CNF formulas can easily be tested for validity, by checking that each clause contains some letter p and its negation $\neg p$. Automated theorem provers often start out from formulas in CNF.

First Step

The first step for converting to CNF is to translate into an equivalent formula that is arrow-free: a formula without $\leftrightarrow$ and $\rightarrow$ operators. Here is the recipe:

- Use the equivalence between $p \rightarrow q$ and $\neg p \vee q$ to get rid of $\rightarrow$ symbols.
- Use the equivalence of $p \leftrightarrow q$ and $(\neg p \vee q) \wedge (p \vee \neg q)$, to get rid of $\leftrightarrow$ symbols.

This conversion has no precondition: it should work for any formula.

```
arrowfree :: Form -> Form
arrowfree (Prop x) = Prop x
arrowfree (Neg f) = Neg (arrowfree f)
arrowfree (Cnj fs) = Cnj (map arrowfree fs)
arrowfree (Dsj fs) = Dsj (map arrowfree fs)
arrowfree (Impl f1 f2) =
  Dsj [Neg (arrowfree f1), arrowfree f2]
arrowfree (Equiv f1 f2) =
  Dsj [Cnj [f1', f2'], Cnj [Neg f1', Neg f2']]
  where f1' = arrowfree f1
        f2' = arrowfree f2
```

The postconditions are: (i) the result should not have no occurrences of `Impl` and `Equiv`, and (ii) the result should be logically equivalent to the original.

Second Step

The second step of the translation into CNF is conversion to negation normal form. Here is the syntactic definition:

$$\varphi ::= p \mid \neg p \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi)$$

No $\rightarrow$, no $\leftrightarrow$, and negation signs are allowed only in front of proposition letters. The precondition of the following transformation is that the input formula is arrowfree. The transformation uses the equivalences between $\neg\neg\varphi$ and φ, between $\neg(\varphi \wedge \psi)$ and $\neg\varphi \vee \neg\psi$, and between $\neg(\varphi \vee \psi)$ and $\neg\varphi \wedge \neg\psi$.

```
nnf :: Form -> Form
nnf (Prop x) = Prop x
nnf (Neg (Prop x)) = Neg (Prop x)
nnf (Neg (Neg f)) = nnf f
nnf (Cnj fs) = Cnj (map nnf fs)
nnf (Dsj fs) = Dsj (map nnf fs)
nnf (Neg (Cnj fs)) = Dsj (map (nnf.Neg) fs)
nnf (Neg (Dsj fs)) = Cnj (map (nnf.Neg) fs)
```

What are the pre- and postconditions?

Lab Work

This CNF conversion process has to be completed with further steps, and tested. This is your lab homework for this week.

See the beginning of the slides. Bear in mind that in the slides above, the conversion gets explained for a version of the language where conjunction and disjunction are binary.

Your task this week will be to modify this for a version of the language with list conjunction and list disjunction.

Next, you should test your program by means of randomly generated input, using appropriate postcondition properties.

The workshop of today will help you to get more familiar with the language and semantics of propositional logic.