Software Testing and Propositional Logic

Jan van Eijck

Specification and Testing, Week 2, 2012

Literate Programming, Again

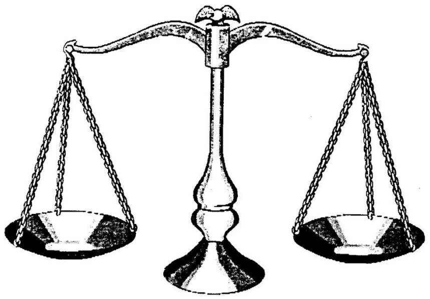
```
module Week2

where

import Data.List
import Data.Char
```

First Testing Challenge

You have 27 coins and a balance scale. All coins have the same weight, except for one coin, which is counterfeit: it is lighter than the other coins.

1. How many weighing tests are needed to find the light coin?
2. And how do you do it?
3. Can you show that a more efficient method (using fewer tests) is impossible?

Implementation of a balance scale



A balance scale gives three possible outcomes:

1. Left scale is lighter
2. Left scale is heavier
3. The two scales are in balance.

For this we can use the Haskell datatype `Ordering`, which takes the three values `LT`, `GT`, and `EQ`.

```haskell
data Coin = C Int

w :: Coin -> Float
w (C n) = if n == lighter then 1 - 0.01
          else if n == heavier then 1 + 0.01
          else 1

weight :: [Coin] -> Float
weight = sum . (map w)

balance :: [Coin] -> [Coin] -> Ordering
balance xs ys =
  if weight xs < weight ys then LT
  else if weight xs > weight ys then GT
  else EQ
```

Using the Balance

```
*Week2> balance [C 1] [C 2]
EQ
*Week2> balance [C 1, C 2] [C 3, C 4]
GT
```

Solution

```
*Week2> balance [C i | i <- [1..9]] [ C i | i <- [10..18]]
LT
*Week2> balance [C i | i <- [1..3]] [ C i | i <- [4..6]]
LT
*Week2> balance [C 1] [C 2]
EQ
```

So `C 3` is the counterfeit coin.

Second Testing Challenge

This time you have 3 coins and a balance. Two coins have the same weight, but there is one coin which has different weight from the other coins.

1. How many weighing tests are needed to find the odd coin?
2. And how do you do it?
3. Can you show that a more efficient method (using fewer tests) is impossible?

Third Testing Challenge

This time you have 12 coins and a balance. All coins have the same weight, except for one coin which has different weight from the other coins.

1. How many weighing tests are needed to find the odd coin?
2. And how do you do it?
3. Can you show that a more efficient method (using fewer tests) is impossible?

Some General Questions

1. Using a balance can be viewed as a test with three possible outcomes. If you perform n balance tests, how much information does that give you (at most)?

2. How much information is required to pick out an odd coin (lighter or heavier) from among n other coins? What can you conclude?

3. If you perform n tests, each of which has m possible outcomes, how much information does your testing activity give you?

Now About Programming

1. Suppose a program P takes user input (with n possible choices) and returns a number in the range $\{0, \ldots, m\}$. How many possibilities are there for how P could behave?

2. Suppose we know that p computes one particular function f. Then we can test the program by checking $p(x)$ against $f(x)$ for every possible input x. How many such tests are needed to check that the program always gives the right outcome?

3. Suppose p is a program of type `Int -> Int`, and we want to check that p implements f.

What does this mean?

To see how many tests are needed, say in the GHC implementation of Haskell, check this:

```
Prelude> minBound :: Int
-9223372036854775808
Prelude> maxBound :: Int
9223372036854775807
```

This indicates GHC uses 8 bytes for representing an `Int`.

So to exhaustively check whether a program of type `Int -> Int` is correct, there are 2^{64} possibilities to check:

```
Prelude> 2^64
18446744073709551616
```

Fourth Testing Challenge

Jill and Joe are dividing two cakes. The cakes are uniform, so only size matters. The rules are: Joe cuts, and Jill can choose once (either for the first cake, or for the second cake).

After the first cake is cut, Jill decides about first or second choice.

Example: the first cake is cut by Joe in two exact halves. Jill decides to choose on the second cake. Joe cuts the second cake is two exact halves. So Jill receives 1 cake (in fact: half of the first cake and half of the second cake).

1. Represent a cut as a floating point number between 0 and 1. Call the first cut x and the second cut y.
2. Represent the first choice as picking a member of $\{x, 1-x\}$ and the second choice as picking a member of $\{y, 1-y\}$.

Programs for Joe and Jill

- Program for Joe: output x, input from $\{F, S\}$, output y.
- Program for Jill: read x, next either pick a member of $\{x, 1-x\}$ and done, or read y and pick a member of $\{y, 1-y\}$.
- Implement these programs as Joe and Jill. Run them against each other.
- Next discuss: How can you test Joe and Jill?

These questions are part of your lab work for this week.

Representation in Haskell

First we ask ourselves what the type of Jill is. Jill is given a floating point number x and has to say ‘yes’ or ‘no’.

This gives:

```haskell
jill :: Float -> Bool
```

Joe produces x, plus a choice for y, dependent on Jill’s ‘yes’ or ‘no’. This gives the following type:

```haskell
joe :: (Float, Bool -> Float)
```

Computing the Outcome

We assume that Jill and Joe both want to get as much as possible. The algorithm for computing how much Jill gets is given by:

```
outcome :: (Float -> Bool) -> (Float, Bool -> Float)
            -> Float
outcome accept (x,decide) =
  if accept x then x + (decide True)
              else (1 - x) + (decide False)
```

This also fixes the outcome for Joe, of course, for Joe gets the remainder of the two cakes.

Once you have programs for `jill` and `joe`, the outcome is fixed. Suppose you define:

```
jill :: Float -> Bool
jill = \ x -> x > 1/2

joe :: (Float, Bool -> Float)
joe = (2/3, \p -> if p then 1/100000 else 1/2)
```

Then you get:

```
*Week2> outcome jill joe
0.6666766666666666
```

But this strategy for Jill is not optimal. And if Joe knows that Jill will play her optimal strategy, he knows what is his optimal strategy, too . . .

Fifth Testing Challenge

Does the following Haskell program terminate for any input?

```
run :: Integer -> [Integer]
run n | n < 1 = error "argument not positive"
      | n == 1 = [1]
      | even n = n: run (div n 2)
      | odd n  = n: run (3*n+1)
```

How many inputs do you have to check?

And if the input is of type `Int` instead of `Integer`?

Int vs Integer

Note the difference between `Int` and `Integer`:

```
*Week2> maxBound :: Int
9223372036854775807
*Week2> maxBound :: Integer

<interactive>:1:1:
    No instance for (Bounded Integer)
      arising from a use of ‘maxBound’
*Week2>
```

Pre- and Postconditions of Functions

Let f be a function of type $A \longrightarrow A$.

A precondition for f is a property of the input.

$\text{pre} : A \longrightarrow \{T, F\}$.

A postcondition for f is a property of the output:

$\text{post} : A \longrightarrow \{T, F\}$.

Notation:

$$\{\text{pre}\}\ f\ \{\text{post}\}.$$

Intended meaning: if the input x of f satisfies pre, then the output $f(x)$ of f satisfies post.

Pre- and Postcondition Composition Rule

The following rule applies:

$$\frac{\{\mathrm{pre}\}\ g\ \{\mathrm{post1}\} \qquad \mathrm{post1} \models \mathrm{pre1} \qquad \{\mathrm{pre1}\}\ f\ \{\mathrm{post}\}}{\{\mathrm{pre}\}\ f \cdot g\ \{\mathrm{post}\}}$$

In the rest of these slides, we will explain what

$$\mathrm{post1} \models \mathrm{pre1}$$

means (in the simplest possible case).

Also we will give an example of the use of pre- and post-conditions in the composition of functions.

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 $\varphi_1 \longrightarrow \varphi_2$.
- 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.
- $\varphi_1 \longrightarrow \varphi_2$ 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 ($\varphi$):
/* precondition: $\varphi$ is a formula. */
/* postcondition: ArrowFree ($\varphi$) returns arrow free version of $\varphi$ */
begin function
case
    $\varphi$ is a literal: return $\varphi$
    $\varphi$ is $\neg\psi$: return $\neg$ ArrowFree ($\psi$)
    $\varphi$ is $\psi_1 \wedge \psi_2$: return ArrowFree ($\psi_1$) $\wedge$ ArrowFree ($\psi_2$)
    $\varphi$ is $\psi_1 \vee \psi_2$: return ArrowFree ($\psi_1$) $\vee$ ArrowFree ($\psi_2$)
    $\varphi$ is $\psi_1 \rightarrow \psi_2$: return ArrowFree ($(\neg\psi_1) \vee \psi_2$)
    $\varphi$ 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 ($\varphi$):
/* precondition: $\varphi$ is arrow-free. */
/* postcondition: NNF ($\varphi$) returns NNF of $\varphi$ */
begin function
case
    $\varphi$ is a literal: return $\varphi$
    $\varphi$ is $\neg\neg\psi$: return NNF ($\psi$)
    $\varphi$ is $\psi_1 \wedge \psi_2$: return NNF ($\psi_1$) $\wedge$ NNF ($\psi_2$)
    $\varphi$ is $\psi_1 \vee \psi_2$: return NNF ($\psi_1$) $\vee$ NNF ($\psi_2$)
    $\varphi$ is $\neg(\psi_1 \wedge \psi_2)$: return NNF ($\neg\psi_1$) $\vee$ NNF ($\neg\psi_2$)
    $\varphi$ 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 ($\varphi$):
/* precondition: $\varphi$ is arrow-free and in NNF. */
/* postcondition: CNF ($\varphi$) returns CNF of $\varphi$ */
begin function
case
    $\varphi$ is a literal: return $\varphi$
    $\varphi$ is $\psi_1 \wedge \psi_2$: return CNF ($\psi_1$) $\wedge$ CNF ($\psi_2$)
    $\varphi$ is $\psi_1 \vee \psi_2$: return DIST (CNF ($\psi_1$), CNF ($\psi_2$))
end case
end function
```

Translating into CNF, auxiliary step

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

First case uses equivalence of $(p \land q) \lor r$ and $(p \lor r) \land (q \lor r)$.
Second case uses equivalence of $p \lor (q \land r)$ and $(p \lor q) \land (p \lor r)$.

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

```
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
```

Evaluation of Formulas

```haskell
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
```

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)
```

In the Lab exercises you are invited to write implementations of:

```
contradiction :: Form -> Bool

tautology :: Form -> Bool

-- logical entailment
entails :: Form -> Form -> Bool

-- logical equivalence
equiv :: Form -> Form -> Bool
```

and to test your definitions for correctness.

Assignments for this Week

1. Implement and test a triangle program
2. Implement and test Joe and Jill programs
3. Implement and test programs for (propositional) contradiction, tautology, logical entailment and logical equivalence.
4. Implement and test a program for CNF conversion of propositional formulas.

Some Help: Conversion to Arrow Free Form

```
-- no precondition: 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
```

What is the appropriate postcondition?

NNF for ArrowFree Formulas

```
-- precondition: input is arrowfree
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 is the appropriate postcondition?