Week 1: Some Testing Challenges
Languages Overview
Some Boolean Logic

Jan van Eijck

CWI

jve@cwi.nl

Sept 8, 2010

First Testing Challenge

You have 27 coins and a balance. 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?

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

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?

Aside: Balance Program in Haskell

```
module Balance where

data Coin = C Int

lighter, heavier :: Int
lighter = 3
heavier = 0

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
```

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?
2. How much information is required to pick out an odd coin 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?

[NOTE: Correction on what was said during the course: this takes n tests. There are $(m+1)^n$ functions of the same type as P and f, but to check whether P and f are the same, we only have to check for all the possible inputs.]

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.

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 $\{y, 1 - y\}$.
3. Program for Joe: output x, input F/S, output y.
4. Program for Jill: read x, next either pick a member of $\{x, 1 - x\}$ or read y and pick a member of of $\{y, 1 - y\}$.

5. Implement these programs as Joe and Jill. Run them against each other.

6. Next discuss: How can you test Joe and Jill?

Fifth Testing Challenge

Does the following Ruby program always terminate?

```
def run k
  raise "argument < 1" if k < 1
  return [k] if k == 1
  return [k] + run(k/2) if k.modulo(2) == 0
  return [k] + run(3*k + 1)
end

print  "Give positive whole number: "
n = gets.to_i
p (run n)
```

Sample Runs

```
lucht:~/courses/testing2007/notes jve$ ruby syracuse.rb
Give positive whole number: 7
[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8,
4, 2, 1]
lucht:~/courses/testing2007/notes jve$ ruby syracuse.rb
Give positive whole number: 8
[8, 4, 2, 1]
lucht:~/courses/testing2007/notes jve$ ruby syracuse.rb
Give positive whole number: 11
[11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

```
lucht:~/courses/testing2007/notes jve$ ruby syracuse.rb
Give positive whole number: 27
[27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322,
161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103,
310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395,
1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251,
754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958,
479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288,
3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232,
4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976,
488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160,
80, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Does this program always terminate? Can you test this?

What does the example tell us about the power of testing?

Aside: Syracuse Program in Haskell

```haskell
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)
```

Three New Programming Languages

- Ruby: `http://www.ruby-lang.org/en/`
- Haskell: `http://www.haskell.org/`
- Ocaml: `http://caml.inria.fr/`

Different Programming Styles for Quicksort

Quicksort sorts by employing a divide and conquer strategy to divide a list into two sub-lists:

1. Pick an element, called a pivot, from the list.
2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements. The base case of the recursion are lists of size zero or one, which never need to be sorted.

See Wikipedia: `http://en.wikipedia.org/wiki/Quicksort`.

Quicksort in C

```
// To sort array a[] of size n: qsort(a,0,n-1)
void qsort(int a[], int lo, int hi) {
{
  int h, l, p, t;

  if (lo < hi) {
    l = lo;
    h = hi;
    p = a[hi];

    do {
      while ((l < h) && (a[l] <= p))
          l = l+1;
      while ((h > l) && (a[h] >= p))
          h = h-1;
      if (l < h) {
          t = a[l];
          a[l] = a[h];
          a[h] = t;
      }
    } while (l < h);

    a[hi] = a[l];
    a[l] = p;

    qsort( a, lo, l-1 );
    qsort( a, l+1, hi );
  }
}
```

Quicksort in Haskell

```
qsort []     = []
qsort (x:xs) = let (a,b) = part xs in
   qsort a ++ x : qsort b
   where
     part [] = ([],[])
     part (y:ys) | y < x      = (y:a,b)
                 | otherwise = (a,y:b)
                    where (a,b) = part ys
```

Quicksort in Ruby

```
def qsort(list)
  return [] if list.size == 0
  x, *xs = *list
  less, more = xs.partition{|y| y < x}
  qsort(less) + [x] + qsort(more)
 end
```

Quicksort in Ocaml

```
let rec quicksort = function
    | (x::xs) ->
         (quicksort (List.filter (fun i -> i < x) xs))
         @ [x] @
         (quicksort (List.filter (fun i -> i >= x) xs))
    | [] -> []
        ;;
```

Propositional Logic: Language

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

This looks simple, but is quite 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.

Note: 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.

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 $\varphi_1 \vee \varphi_2$ */
begin function
case
    $\varphi_1$ is $\psi_{11} \wedge \psi_{12}$: return DIST ($\psi_{11}$,$\varphi_2$) $\wedge$ DIST ($\psi_{12}$,$\varphi_2$)
    $\varphi_2$ is $\psi_{21} \wedge \psi_{22}$: return DIST ($\varphi_1$,$\psi_{21}$) $\wedge$ DIST ($\varphi_1$,$\psi_{22}$)
    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 $(p \vee q) \wedge (p \vee r)$.

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.

LAI Chapter 5 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.