6.005 Elements of Software Construction
Fall 2008

6.005 elements of software construction

how to design a SAT solver, part 2

Daniel Jackson

plan for today

topics

- designing a naive solver
- more recursive functions over datatypes

today's patterns

- Interpreter: recursive traversals (again)
- Backtracking Search
- Facade for simpler use of API

where we are

datatype productions

last time we saw

- how to model formulas using datatype productions
- like a grammar, but abstract structure only

productions

```
Formula = OrFormula + AndFormula + Not(formula:Formula)+ Var(name:String)
OrFormula = OrVar(left:Formula,right:Formula)
AndFormula = And(left:Formula,right:Formula)
```

sample formula: $(P \lor Q) \land (\neg P \lor R)$

- as a term:

```
And(Or(Var("P"), Var("Q")), (Not(Var("P")), Var("R")))
```

Variant as Class pattern

last time we saw

- how to define a datatype to model a set of values
- how to build a class structure representing it
- how to implement recursive functions over the datatype

example

- production

  List<E> = Empty + Cons (first: E, rest: List<E>)

- code

```
public abstract class List<E> {}
public class Empty<E> extends List<E> {}
public class Cons<E> extends List<E> {
    private final E first;
    private final List<E> rest;
    public Cons (E e, List<E> r) {first = e;rest = r;}
    public E first () {return first;}
    public List<E> rest () {return rest;}
}
```

Interpreter pattern

how to build a recursive traversal

- write type declaration of function

  size: List<E> -> int

- break function into cases, one per variant

  List<E> = Empty + Cons(first:E, rest: List<E>)

  size (Empty) = 0
  size (Cons(first:e, rest: l)) = 1 + size(rest)

- implement with one subclass method per case

```
public abstract class List<E> {
    public abstract int size ();
}
public class Empty<E> extends List<E> {
    public int size () {return 0;}
}
public class Cons<E> extends List<E> {
    private final E first;
    private final List<E> rest;
    public int size () {return 1 + rest.size();}
}
```

SAT solver functions

functions for SAT

generate and test strategy

- steps
  - extract set of **variables** from formula
  - try all environments over those vars
  - **eval**uate the formula for each
- functions
  - vars: Formula -> Set<Var>
  - solve: Formula -> Option<Env>
  - eval: Formula, Env -> Bool

set and env

what are the Set and Env types?

- can define as datatypes too

  Set<T> = List<T>

  Env = List<Tuple<Var, Boolean>>

  Boolean = True + False

something new going on here

- what is the meaning of equals in Set<T> = List<T> ?
- representation (on right) is hidden from clients
- not all terms are acceptable: no duplicates, eg
- more on this later when we discuss abstract types

set and env specs

assume for now

- Set and Env implemented as classes, with list representations
- but offering special methods:

```
public class Set<E> {
    public Set () {...}
    public Set<E> add (E e) {...}
    public Set<E> remove (E e) {...}
    public Set<E> addAll (Set<E> s) {...}
    public boolean contains (E e) {...}
    public E choose () {...}
    public boolean isEmpty () {...}
    public int size () {...}
}

public class Env {
    public Env() {...}
    public Env put(Var v, boolean b) {...}
    public boolean get(Var v) {...} // requires: v is bound in this environment
}
```

computing var set

applying strategy

- write type declaration of function

  vars: Formula -> Set<Var>

- break function into cases, one per variant

  F = Var(name:String) + Or(left:F,right:F) + And(left:F,right:F) + Not(formula:F)

  vars (Var(n)) = {Var(n)}

  vars (Or(fl, fr)) = vars(fl) ∪ vars(fr)

  vars (And(fl, fr)) = vars(fl) ∪ vars(fr)

  vars (Not(f)) = vars(f)

- implement with one subclass method per case, eg

```
public class AndFormula extends Formula {
        private final Formula left, right;
        public Set<Var> vars () {
            return left.vars().addAll(right.vars());
        }
    }
```

vars in full

```
public abstract class Formula {
    public abstract Set<Var> vars ();
}
public class AndFormula extends Formula {
    private final Formula left, right;
    public Set<Var> vars () {
        return left.vars().addAll(right.vars());
    }
}
public class OrFormula extends Formula {
    private final Formula left, right;
    public Set<Var> vars () {
        return left.vars().addAll(right.vars());
    }
}
public class NotFormula extends Formula {
    private final Formula formula;
    public Set<Var> vars () {
        return formula.vars();
    }
}
public class Var extends Formula {
    public Set<Var> vars () {
        return new ListSet<Var>().add(this);
    }
}
```

in-class exercise

apply the strategy for eval

- write type declaration of function

  eval: Formula, Env -> Boolean

- break function into cases, one per variant

  F = Var(name:String) + Or(left:F,right:F) + And(left:F,right:F) + Not(formula:F)

  eval (Var(n), e) = e.get(Var(n))

  eval (Or(fl, fr), e) = eval(fl,e) || evals(fr,e)

  eval (And(fl, fr), e) = eval(fl,e) && eval(fr,e)

  eval (Not(f), e) = ! eval(f,e)

- implement with one subclass method per case, eg

```
public class AndFormula extends Formula {
    private final Formula left, right;
    public boolean eval (Env e) {
        return left.eval (e) && right.eval (e);
    }
}
```

eval in full

```
public abstract class Formula {
      public abstract boolean eval (Env e);
}
public class AndFormula extends Formula {
    private final Formula left, right;
    public boolean eval (Env e) {
        return left.eval (e) && right.eval (e);
    }
}
public class OrFormula extends Formula {
    private final Formula left, right;
    public boolean eval(Env e) {
        return left.eval(e) || right.eval(e);
    }
}
public class NotFormula extends Formula {
    private final Formula formula;
    public boolean eval (Env e) {
        return !formula.eval (e);
    }
public class Var extends Formula {
    public boolean eval (Env e) {
        return e.get(this);
    }
}
```

a naive solver

naive SAT

backtracking search

- pick a var, and try setting to false and then to true if that fails
- do this recursively, evaluating the formula when no vars left

implementation

```
public abstract class Formula {
    ...
    public Env solve () {
        return solve (new Env (), this.vars());
    }

    private Env solve(Env env, Set<Var> vars) {
        if (vars.isEmpty())
            return eval(env) ? env : null;
        Var v = vars.choose();
        Set<Var> restVars = vars.remove(v);
        Env e = solve (env.put(v, false), restVars);
        if (e != null) return e;
        return solve (env.put(v, true), restVars);
    }
}
```

example

- formula f =

  Socrates⇒Human ∧ Human⇒Mortal ∧ ¬ (Socrates⇒Mortal)

- vars(f) =

  {Socrates, Human, Mortal}

- possible environments

  {Socrates->False, Human->False, Mortal->False}

  {Socrates->False, Human->False, Mortal->True}

  {Socrates->False, Human->True, Mortal->False}

  {Socrates->False, Human->True, Mortal->True}

  {Socrates->True, Human->False, Mortal->False}

  {Socrates->True, Human->False, Mortal->True}

  {Socrates->True, Human->True, Mortal->False}

  {Socrates->True, Human->True, Mortal->True}

- formula evaluates to false on all, so theorem holds

class exercise

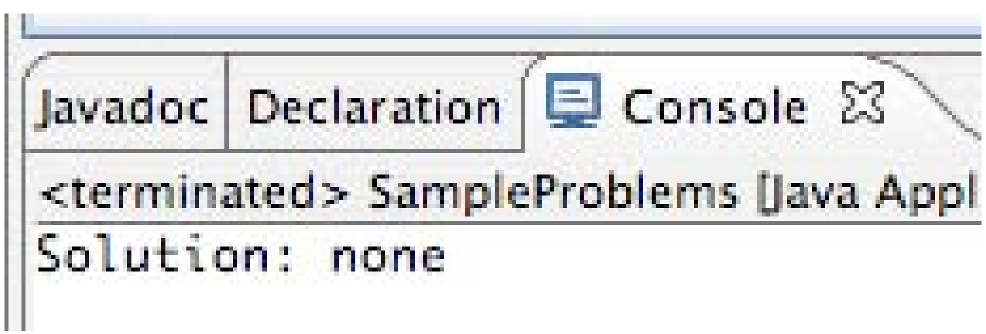
what order are environments checked in?

- depends on behaviour of Set.choose
- assume it returns vars in this order

    Socrates, Human, Mortal

running the example

```
    public static void main (String[] args) {
    Var s = new Var ("Socrates");
    Var h = new Var ("Human");
    Var m = new Var ("Mortal");
    Formula old_f =
        new AndFormula (new OrFormula (new NotFormula (s), h),
            new AndFormula (new OrFormula (new NotFormula (h), m),
                new NotFormula (new OrFormula (new NotFormula (s), m))));
    Environment e = f.solve();
    System.out.println ("Solution: " + (e == null ? "none" : e));
}
```



Courtesy of The Eclipse Foundation. Used with permission.

solving a Latin square

```
long started = System.nanoTime();
Sudoku s = new Sudoku (2);
System.out.println ("Creating SAT formula...");
Formula f = s.getFormula();
System.out.println ("Solving with naive method...");
Environment e = f.solve();
System.out.println ("Interpreting solution...");
String solution = s.interpretSolution(e);
System.out.println ("Solution is: \n" + solution);
long time = System.nanoTime();
long timeTaken = (time - started);
System.out.println ("Time:" + timeTaken/1000000 + "ms");
```

```
Creating SAT formula...
Solving with naive method...
Interpreting solution...
Solution is:
    |3|4|2|1|
    |1|2|4|3|
    |4|3|1|2|
    |2|1|3|4|
Time:797ms
```

design extras

an awkward API

look at how formula is created by client

- tedious to have to use constructors and multiple classes

```
Formula f =
    new AndFormula (new OrFormula (new NotFormula (s), h),
        new AndFormula (new OrFormula (new NotFormula (h), m),
            new NotFormula (new OrFormula (new NotFormula (s), m))));
```

define methods in Formula class to avoid this: example of Facade

```
public abstract class Formula {
    public Formula and (Formula f) {
    return new AndFormula (this, f);
    }
    public Formula or (Formula f) {
    return new OrFormula (this, f);
    }
    public Formula not () {
    return new NotFormula (this);
    }
}
```

- can now write

```
Formula f = s.not().or(h).and(h.not().or(m).and(s.not().or(m).not()));
```

module dependency diagram

handling unbound vars

how should get method handle unbound var?

- one approach: return an arbitrary value
- technically correct, but not very robust

```
public class Environment {
    Map <Var, Boolean> bindings;
    ...
    /**
     * requires that v is bound in this environment
     * @return the boolean value that v is bound to
     */
    public boolean get(Var v){
        Boolean b = bindings.get(v);
        if (b==null) return false;
        else return b;
    }
}
```

three-valued logic

an alternative: define 3 logical values

Boolean = True + False + Undefined

```
public enum Bool {
    TRUE, FALSE, UNDEFINED;

    public Bool and (Bool b) {
        if (this==FALSE || b==FALSE) return FALSE;
        if (this==TRUE && b==TRUE) return TRUE;
        return UNDEFINED;
    }
...}
```

now we can return undefined

```
/**
 * @return the boolean value that v is bound to, or
 * the special UNDEFINED value of it is not bound
 */
public Bool get(Var v){
    Bool b = bindings.get(v);
    if (b==null) return Bool.UNDEFINED;
    else return b;
}
```

using Bool

use methods of Bool instead of &&, ||, etc

```
public class AndFormula extends Formula {
    public Bool eval (Environment e) {
        return left.eval(e).and (right.eval (e));
        }
    }
```

and in solver, can evaluate before all vars are bound

```
public Environment solve () {
    return solve (new Environment (), this.vars());
}

private Environment solve(Environment env, Set<Var> vars) {
    if (eval(env) == Bool.TRUE) return env;
    if (eval(env) == Bool.FALSE) return null;
    Var v = vars.choose();
    Set<Var> restVars = vars.remove(v);
    Environment e = solve (env.put(v, Bool.FALSE), restVars);
    if (e != null) return e;
    return solve (env.put(v, Bool.TRUE), restVars);
}
```

puzzle

introduction of Bool

- produces dramatic performance improvement
- 4x4 Latin square actually doesn't terminate without it
- what's going on?

return type of solve

recall solve function

- prototype is

    solve: Formula -> Option<Env>

- recall option datatype

    Option<T> = Some(value:T) + None

how should this be implemented?

- we used nulls
- is there a better way?

look ma, no nulls!

```
public class Option<T> {}
public class None<T> extends Option<T>{}
public class Some<T> extends Option<T>{
   private T value;
   public Some (T v) {value = v;}
   public T getValue () {return value;}
}

public void displaySolution () {
   Option<Environment> o = solve (new Environment (), this.vars());
   if (o instanceof Some)
      System.out.println ((Some<Environment>) o).getValue();
   else System.out.println ("No solution");
}
private Option<Environment> solve (Environment env, Set<Literal> vars) {
   if (eval(env) == Bool.TRUE) return new Some<Environment>(env);
   if (eval(env) == Bool.FALSE) return new None<Environment>();
    Var v = vars.choose();
    Set<Var> restVars = vars.remove(v);
    Option<Environment> o = solve (env.put (c, Bool.FALSE), restVars);
    if (o instanceof Some) return o;
    return solve (env.put(v, Bool.TRUE), restVars);
}
```

comparing options

two options for `Option`

- have solve return an `Env` or a `null` value
- implement `Option<T>` directly

others?

- throw an exception if not successful
- have solve return a pair `(boolean, env)`

class discussion

- advantages and disadvantages of each

abstract classes vs. interfaces

what's an abstract class?

like a regular class

- but can't be instantiated

like an interface

- but can contain fields and method bodies
- methods not implemented are marked `abstract`

why useful?

- can collect fields and methods common across subclasses
  - eg: `Formula.solve`
- can use as Facade
  - eg: `Formula.and`, `Formula.or`, `Formula.not`

using interfaces instead

changes to List

- code is now

```
public interface List<E> {}
public class Empty<E> implements List<E> {}
public class Cons<E> implements List<E> {
    private final E first;
    private final List<E> rest;
    public Cons (E e, List<E> r) {first = e;rest = r;}
    public E first () {return first;}
    public List<E> rest () {return rest;}
}
```

fixing size

what becomes of this?

```
public abstract class List<E> {
    int size;
    public int size () {return size;}
}
public class Empty<E> extends List<E> {
    public EmptyList () {size = 0;}
}
public class Cons<E> extends List<E> {
    private final E first;
    private final List<E> rest;
    private Cons (E e, List<E> r) {first = e;rest = r;size = r.size()+1}
}
```

fixing facade

and what becomes of this?

```
public abstract class Formula {
    public Environment solve (Formula f) {
        return ...;
    }
    public Formula and (Formula f) {
        return new AndFormula (this, f);
    }
    public Formula or (Formula f) {
        return new OrFormula (this, f);
    }
    public Formula not () {
        return new NotFormula (this);
    }
}
```

formula facade

```
public class Formulas {
    public static Environment solve (Formula f) {
         return ...;
    }
    public static Formula and (Formula f, Formula g) {
         return new AndFormula (f, g);
    }
    public static Formula or (Formula f, Formula g) {
         return new OrFormula (f, g);
    }
    public static Formula not (Formula f) {
         return new NotFormula (f);
    }
}
```

interfaces vs. abstract classes

advantages of interfaces

- you know at compile time which method is executed
- enforces clean specification

disadvantages

- need extra (singleton) class for facade
- can't share code

what's wrong with our solver?

a missed opportunity

look at what happens

- after

  $Socrates \Rightarrow Human \land Human \Rightarrow Mortal \land \neg (Socrates \Rightarrow Mortal)$

- suppose order or evaluation is Socrates, Human, Mortal
- and suppose we set Socrates to true
- then clearly must set Human to true
- and then must set Mortal to true...
- but our solver ignores all this

next time

- a real SAT solver
- implements this scheme with unit propagation

summary

summary

big ideas

- backtracking search: easy with immutable types

patterns

- Variant as Class: abstract class for datatype, one subclass/variant
- Interpreter: recursive traversal over datatype with method in each subclass
- Facade: make client of API dependent on only a single class

where we are

- built a naive solver that works for small problems
- next time, a real SAT solver