how to design a SAT solver
part 2: traversals

Daniel Jackson
March 12, 2008
announcements

breakfast with your prof
  • see you tomorrow at 10am in 32-G725
  • will do this again...

reminders
  • one code handout
  • project 2, LNBs due on R

today
  • name that tune!
plan for today

**topics**

- reprise: basic recursive traversals
- patterns: Visitor, Factory, Iterator
- higher-order functions
- a real SAT algorithm: DPLL
basic recursive traversals
basic recursive traversal

how to build a recursive traversal

• write type declaration of function
  
  \texttt{size: List\langle E \rangle \rightarrow int}

• break function into cases, one per variant
  
  \texttt{List\langle E \rangle = Empty + Cons(first: E, rest: List\langle E \rangle)}
  
  \texttt{size (Empty) = 0}
  
  \texttt{size (Cons(first: e, rest: l)) = 1 + size(rest)}

• implement with one subclass method per case

  \begin{verbatim}
  public abstract class List\langle E \rangle {
    public abstract int size();
  }
  public class Empty\langle E \rangle extends List\langle E \rangle {
    public int size () {return 0;}
  }
  public class Cons\langle E \rangle extends List\langle E \rangle {
    private final E first;
    private final List\langle E \rangle rest;
    public int size () {return 1 + rest.size();}
  }
  \end{verbatim}
functions for SAT

generate and test strategy

• steps
  extract set of **literals** from formula
  generate all environments over those literals \(<--\) discuss later
  evaluate the formula for each

• functions needed
  literals: Formula \(\rightarrow\) Set<Literal>
  eval: Formula, Env \(\rightarrow\) Bool
computing literal set

applying strategy

• write type declaration of function

   literals: Formula \to Set\langle\text{Literal}\rangle

• break function into cases, one per variant

   \begin{align*}
   F &= \text{Literal(name: String)} + \text{Or(left: F, right: F)} + \text{And(left: F, right: F)} + \text{Not(formula: F)} \\
   \text{literals (Literal(name: n))} &= \{\text{Literal(name: n)}\} \\
   \text{literals (Or(left: fl, right: fr))} &= \text{literals(fl)} \cup \text{literals(fr)} \\
   \text{literals (And(left: fl, right: fr))} &= \text{literals(fl)} \cup \text{literals(fr)} \\
   \text{literals (Not(formula: f))} &= \text{literals(f)}
   \end{align*}

• implement with one subclass method per case, eg

   \begin{verbatim}
   public class AndFormula extends Formula {
       private final Formula left, right;
       public Set<Literal> literals () {
           return left.literals().addAll(right.literals());
       }
   }
   \end{verbatim}
public abstract class Formula {
    public abstract Set<Literal> literals();
}

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

public class OrFormula extends Formula {
    private final Formula left, right;
    public Set<Literal> literals() {
        return left.literals().addAll(right.literals());
    }
}

public class NotFormula extends Formula {
    private final Formula formula;
    public Set<Literal> literals() {
        return formula.literals();
    }
}

public class Literal extends Formula {
    public Set<Literal> literals() {
        return new ListSet<Literal>().add(this);
    }
}
in-class exercise

apply the strategy for eval

• write type declaration of function
  
  \[ \text{eval}: \text{Formula}, \text{Env} \rightarrow \text{Bool} \]

• break function into cases, one per variant
  
  \[ F = \text{Literal(name: String)} + \text{Or(left:F, right:F)} + \text{And(left:F, right:F)} + \text{Not(formula:F)} \]
  
  eval (Literal(name:n), e) = e.get(Literal(name:n))
  eval (Or(left:fl, right:fr), e) = eval(fl, e) || literals(fr, e)
  eval (And(left:fl, right:fr), e) = eval(fl, e) && literals(fr, e)
  literals (Not(formula: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) throws NoSuchLiteralException {
      return left.eval (e) && right.eval (e);
    }
  }
public abstract class Formula {
    public abstract boolean eval (Env e) throws NoSuchLiteralException;
}

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

public class OrFormula extends Formula {
    private final Formula left, right;
    public boolean eval (Env e) throws NoSuchLiteralException {
        return left.eval (e) || right.eval (e);
    }
}

public class NotFormula extends Formula {
    private final Formula formula;
    public boolean eval (Env e) throws NoSuchLiteralException {
        return !formula.eval (e);
    }
}

public class Literal extends Formula {
    public boolean eval (Env e) throws NoSuchLiteralException {
        return e.get (this);
    }
}
visitor pattern
localizing functions

look what we’re doing

• break function into cases, one per variant
  
  \[
  \begin{align*}
  \text{size (Empty)} &= 0 \\
  \text{size (Cons(first:e, rest: l))} &= 1 + \text{size(rest)}
  \end{align*}
  \]

• but then these cases get split across classes!

• wouldn’t it be nice to have them all in one place?

• in functional language can do exactly this: (in ML, eg)
  
  \[
  \text{fun size Empty = 0} \\
  \quad | \text{Cons(e, l) = 1 + size(rest)}
  \]

solution: localize function definition in “visitor”

• in my opinion, really ugly (but once familiar, you see through the mess)

• important to know, because it’s a common (and useful) idiom
basic visitor structure

\[\text{visitor}\]

```java
public interface ListIntVisitor<E> {
    int onEmpty (Empty<E> l);
    int onCons (Cons<E> l);
}

public class SizeVisitor<E> implements ListIntVisitor<E>{
    public Integer onEmpty(Empty<E> l) {return 0;}
    public Integer onCons(Cons<E> l) {return 1 + l.rest().accept(this);}
}

\[\text{datatype and variants}\]

```java
public abstract class List<E> {
    public abstract int accept(ListIntVisitor<E> visitor);
}

public class Empty<E> extends List<E> {
    public int accept(ListIntVisitor visitor) {return visitor.onEmpty(this);}
}

public class Cons<E> extends List<E> {
    public int accept(ListIntVisitor<E> visitor) {return visitor.onCons(this);}
}

\[\text{usage}\]

```java
int size = myList.accept(new SizeVisitor<E>());
```
the visitor carousel

note how

• control passes back and forth between visitor and datatype objects
• function is computed at visitor (steps 3 and 5)
going polymorphic

accept methods only work for visitor that returns integer

```java
public interface ListIntVisitor<E> {
    int onEmpty (Empty<E> l);
    int onCons (Cons<E> l);
}
```

so make the visitor polymorphic

```
new interface
```
```
public interface ListVisitor<E,T> {
    T onEmpty (Empty<E> l);
    T onCons (Cons<E> l);
}
```
```
new accept methods
```
```
public <T> T accept(ListVisitor<E,T> visitor) {return visitor.onEmptyList(this);}
```
```
new visitor
```
```
public class SizeVisitor<E> implements ListVisitor<E,Integer>{
    public Integer onEmpty(Empty<E> l) {return 0;}
    public Integer onCons(Cons<E> l) {return 1 + l.rest().accept(this);}
}
```

final refinement

accept method is almost boilerplate

```java
public class Cons<E> extends List<E> {
    public int accept(ListIntVisitor<E> visitor) { return visitor.onCons(this); }
}
```

can make identical by exploiting overloading

• new interface

```java
public interface ListVisitor<E,T> {
    T visit (Empty<E> l);
    T visit (Cons<E> l);
}
```

• new accept method: same in all variants

```java
public <T> T accept(ListVisitor<E,T> visitor) { return visitor.visit(this); }
```

• new visitor

```java
public class SizeVisitor<E> implements ListVisitor<E,Integer>{
    public Integer visit (Empty<E> l) {return 0;}
    public Integer visit (Cons<E> l) {return 1 + l.rest().accept(this);}
}
```
higher-order functions
generating all environments

this is not so easy

- don’t want a function to return entire set as one object (too big)
- hard (in Java) to return one at a time

one solution

- write a function that takes a predicate and checks it against each env
- then we can pass this function a predicate that evaluates a given formula
- express this with a polymorphic function declarations

\[
\text{find : Set<Literal>, (Env -> Bool) -> (Env + None)}
\]

- find is a higher-order function that takes a function as an arg
- in Java, implement function as an object

\[
\text{Pred.match: Env -> Bool}
\]

\[
\text{find : Set<Literal>, Pred -> (Env + None)}
\]
public interface Matcher <T> {
    // if match exists, return it, else return null
    T find (Predicate<T> p);
}
public interface Predicate <T>{
    boolean test (T t);
}
public class EnvSet implements Matcher<Env> {
    public Env find(Predicate<Env> pred) {...}
}

// sample usage
Formula f = ...;
Set<Literal> literals = f.literals();
EnvSet es = new EnvSet (literals);
Predicate pred = new Predicate<Env> () {
    public boolean test (Env e) {
        try {return f.eval (e);} catch (NoSuchLiteralException e1) {return false;}
    }
};
Env solution = es.find (pred);
System.out.println ("Solution is: " + solution);
public class EnvSet implements Matcher<Env> {
    Set<Literal> literals;

    public EnvSet(Set<Literal> literals) {
        this.literals = literals;
    }

    public Env find(Predicate<Env> pred) {
        return find (new Environment (), literals, pred);
    }

    private Env find(Env env, Set<Literal> literals, Predicate<Env> pred) {
        if (literals.isEmpty()) {
            if (pred.test(env)) return env;
            else return null;
        }
        Literal l = literals.choose();
        Set<Literal> literalsRest = literals.remove(l);
        Env newEnv = env.put(l, false);
        Env match = find(newEnv, literalsRest, pred);
        if (match != null) return match;
        newEnv = env.put(l, true);
        return find(newEnv, literalsRest, pred);
    }
}
the SAT algorithm
basic backtracking algorithm

clausal form
• recall that algorithm acts on formula represented as clause-set
• product of sums: need every clause true, some literal in each clause

elements of the algorithm
• backtracking search: pick a literal, try false then true
• if clause set is empty, success
• if clause set contains empty clause, failure

element
• want to prove Socrates⇒Mortal from Socrates⇒Human ∧ Human⇒Mortal
• so give solver: Socrates⇒Human ∧ Human⇒Mortal ∧ ¬ (Socrates⇒Mortal)
• in clausal form: {{¬Socrates, Human}, {¬Human, Mortal}, {Socrates}, {¬Mortal}}
• in shorthand: {SH}{HM}{S}{M}
backtracking execution

- stop when node contains {} (failure) or is empty (success)
- in this case, all paths fail, so theorem is valid
- in worst case, number of leaves is $2^{\#\text{literals}}$
DPLL

classic SAT algorithm
• Davis-Putnam-Logemann-Loveland, 1962

unit propagation
• on top of backtracking search
• if a clause contains one literal, set that literal to true

example (on right)
• in this case, no splitting needed
• propagate S, then H, then M

performance
• often much better, but worst case still exponential
clausifying

how to get formula in clausal form?

\[
\text{clausify: } \text{Formula} \rightarrow \text{Set<Clause>}
\]
\[
\text{Clause} = \text{Set<CLiteral>}
\]
\[
\text{CLiteral} = \text{Pos(name: String)} + \text{Neg(name: String)}
\]

applying our strategy

\[
F = \text{Literal(name: String)} + \text{Or(left:F, right:F)} + \text{And(left:F, right:F)} + \text{Not(formula:F)}
\]
\[
\text{clausify (Literal(name:n))} = \text{Pos(name: n)}
\]
\[
\text{clausify (And(left:fl, right:fr))} = \text{clausify (fl)} \cup \text{clausify (fr)}
\]
\[
\text{clausify (Or(left:fl, right:fr))} = \{\text{merge (cl, cr) | cl} \in \text{clausify (fl)} \land \text{cr} \in \text{clausify (fr)}\}
\]

how do you figure this out? just apply deMorgan

' problem in clausal form is \( C_1 \land C_2 \land \ldots \) for clauses \( C_1, C_2, \ldots \), so:

\[
(L_1 \land L_2 \land \ldots) \land (R_1 \land R_2 \land \ldots) = (L_1 \land L_2 \land \ldots \land R_1 \land R_2 \land \ldots)
\]
\[
(L_1 \land L_2 \land \ldots) \lor (R_1 \land R_2 \land \ldots) = (L_1 \lor R_1) \land (L_1 \lor R_2) \land \ldots
\]
factory method pattern
comparing literals

when merging clauses or setting literal

✓ need to compare literals

  eg, when S is true: \{SH\} reduces to \{H\}, and \{SH\} can be dropped

✓ a SAT solver will do this a lot, so must be efficient

equality of immutable types

✓ calling constructor twice on same args gives distinct objects

  Literal a = new Literal ("S");
  Literal b = new Literal ("S");
  System.out.println (a==b ? "same" : "not");  // prints not

two strategies

✓ use equals method, and code it to compare object values
  for literals, compare names char-by-char every time!

✓ intern the objects so there's at most one object with a given value
interning with a factory method

factory method pattern

- instead of constructor, client calls a static method

```java
public static T make () { return new T(); }
```

- factory method can call constructor, but can also recycle objects

```java
class Pos extends Literal {
    protected static Map<String,Pos> alloc = new ListMap<String,Pos>();
    private Pos (String name) {super(name);}
    public static Pos make (String name) {
        Pos l = alloc.get(name);
        if (l==null) {
            l = new Pos(name);
            Neg n = new Neg(name);
            l.negation = n; n.negation = l;
            alloc = alloc.put(name, l);
        }
        return l;
    }
}
```
another common use

factory method hides choice of class

‣ instead of constructor, client calls

    public static T make () { return new T1(); }

‣ decision to use subtype **T1** is encapsulated in factory method

factory pattern

‣ define a class whose sole purpose is generating objects

    public class T1Factory extends TFactory {
        public T make () { return new T1(); }
    }

    public X client (TFactory f) {
        T t = f.make();
        ...
    }
factory decoupling

factory decoupling client from subtype

- client does not depend on subtype
- but unlike our plugin example (lecture 3), client can create subtype objects
iterator pattern
how clausify works for OR

define a class to represent a formula in CNF

\- has methods to support clausifying
\- or method forms all pairwise merges of clauses

```java
public class Problem {
    List<Clause> clauses;

    public Problem or (Problem p) {
        List<Clause> result = new Empty<Clause>();
        for (Clause c1 : p.clauses)
            for (Clause c2 : clauses) {
                Clause c = c1.merge(c2);
                result = result.add(c);
            }
        return new Problem(result);
    }
}
```

how does this work?

\- hidden iterator at play
the iterator pattern

a Java shorthand

the statement

\[ \text{for (E e: c) {...}} \]

is short for

\[ \text{Iterator i = c.iterator();} \]
\[ \text{while (i.hasNext()) { } } \]
\[ \text{E e = i.next();} \]
\[ \text{...} \]

iterator interface

\[ \text{public interface Iterator<E> { } } \]
\[ \text{boolean hasNext () ;} \]
\[ \text{E next () ;} \]
\[ \text{void remove () ;} \]

list iterator

\[ \text{public class ListIterator<E> implements Iterator<E> { } } \]
\[ \text{List<E> remaining ;} \]
\[ \text{public ListIterator (List<E> list) { } } \]
\[ \text{remaining = list ;} \]
\[ \text{public boolean hasNext () { } } \]
\[ \text{return !remaining.isEmpty();} \]
\[ \text{public E next () { } } \]
\[ \text{E first = remaining.first ();} \]
\[ \text{remaining = remaining.rest();} \]
\[ \text{return first; } \]

iterator method

\[ \text{public abstract class List<E> implements Iterable<E> { } } \]
\[ \text{public Iterator<E> iterator () { } } \]
\[ \text{return new ListIterator<E>(this);} \]
\[ \text{}} \]
iterator state machine

why a stateful object in a side-effect free program?
  · the only convenient way to do iteration in Java
  · so long as iterator used only in for loop as shown, no mutability issues arise
putting it all together: demo
The Latin square problem involves filling a 10x10 square with numbers 0..9 in each row and column. There are 1000 boolean variables (literals) and the solution can be computed in about 20 seconds on a 2.4GHz Intel Mac with 4GB memory.

Solution:

```
| 7 | 8 | 9 | 5 | 1 | 6 | 4 | 2 | 0 | 3 |
| 6 | 9 | 7 | 8 | 3 | 4 | 2 | 5 | 1 | 0 |
| 9 | 7 | 3 | 4 | 8 | 5 | 0 | 6 | 2 | 1 |
| 3 | 5 | 8 | 6 | 9 | 1 | 7 | 0 | 4 | 2 |
| 8 | 6 | 2 | 9 | 0 | 7 | 1 | 4 | 3 | 5 |
| 1 | 4 | 0 | 7 | 2 | 9 | 3 | 8 | 5 | 6 |
| 0 | 3 | 1 | 2 | 4 | 8 | 5 | 9 | 6 | 7 |
| 4 | 2 | 5 | 3 | 6 | 0 | 9 | 1 | 7 | 8 |
| 2 | 1 | 4 | 0 | 5 | 3 | 6 | 7 | 8 | 9 |
| 5 | 0 | 6 | 1 | 7 | 2 | 8 | 3 | 9 | 4 |
```

Time: 18127ms
public class LatinSquare {
    private final Literal [][][][] occupies;
    private final int size;
    private Formula formula;
    public LatinSquare (int size) {
        this.size = size;
        occupies = new Literal [size][size][size];
        // occupies [i,j,k] means that symbol k occupies entry in row i, column j
        for (int i = 0; i < size; i++)
            for (int j = 0; j < size; j++)
                for (int k = 0; k < size; k++) {
                    Literal l = new Literal ("occupies(" + i + "," + j + "," + k + ")");
                    occupies[i][j][k] = l;
                }
        formula = Formula.TRUE;
        // each symbol appears exactly once in each row
        for (int k = 0; k < size; k++)
            for (int i = 0; i < size; i++) {
                Formula atMost = Formula.TRUE;
                Formula atLeast = Formula.FALSE;
                for (int j = 0; j < size; j++) {
                    atLeast = atLeast.or (occupies[i][j][k]);
                    for (int j2 = 0; j2 < size; j2++)
                        if (j != j2)
                            atMost =
                                atMost.and (occupies[i][j][k].implies(occupies[i][j2][k].not()));
                }
                formula = formula.and (atMost).and (atLeast);
            }
    }...
}
interparing the solution

can't just display environment

• need to translate to solution in the problem domain

• so provide a method to map back

• a tricky issue: clausify generates new literals, but because they're interned, the ones returned here by the factory method (make) are the ones used in the environment

```java
public String interpretSolution(Environment e) {
    String result = "";
    for (int i = 0; i < size; i++) {
        String row = "|";
        for (int j = 0; j < size; j++)
            for (int k = 0; k < size; k++) {
                Literal l = occupies[i][j][k];
                clausal.PosLiteral pl = clausal.PosLiteral.make(l.getName());
                if (l.eval(e) == Bool.TRUE || pl.eval(e) == Bool.TRUE)
                    row = row + k + "|";
            }
        result = result + row + "\n";
    }
    return result;
}
```
features of modern SAT solvers
modern SAT solvers

some great open-source SAT solvers

‣ Sat4J (all Java) http://www.sat4j.org/
‣ Chaff http://www.princeton.edu/~chaff
‣ Berkmin http://eigold.tripod.com/BerkMin.html
‣ MiniSat http://minisat.se/

what do they do beyond what I’ve explained?

‣ learning: if literal choices ABC ended in failure, add \{ABC\}
‣ splitting heuristics: pick the literal to split on carefully
‣ randomization: restart with new literal order
‣ clever representation invariants (explained later in course)

a less conventional SAT solver

summary
summary

traversals

• higher-order functions become objects in Java
• factory, iterator, visitor: patterns to study

where we are

• we’ve looked at how to represent datatypes and traverse them
• next week: how to package them, and decouple from client
lecture exercise

for Monday

[moderately hard] use of null in SAT code

• my use of null as a return value in Clause.merge is controversial

• why did I do it? what’s bad about it?

• what were the alternatives?