how to design a SAT solver

Daniel Jackson
March 10, 2008
announcements

breakfast with your prof

- Thursday, 10am in 32-G704
- email me to say you're coming

reminders

- one handout
plan for today

topics

• what’s a SAT solver and why do you want one?
• new paradigm: functions over immutable values
• datatype productions
• deriving class structure
• recursive traversals
what's a SAT solver?
what is SAT?

the SAT problem

› given a formula made of boolean variables and operators

\[(P \lor Q) \land (¬P \lor R)\]

› find an assignment to the variables that makes it true

\[
\{Q = \text{true}, \ R = \text{true}\}
\]

conjunctive normal form (CNF) or “product of sums”

› set of clauses, each containing a set of literals

\[
\{\{P, Q\}, \{¬P, R\}\}
\]

› literal is just a variable, maybe negated

SAT solver

› program that takes a formula in CNF

› returns an assignment, or says none exists
SAT is hard

how to build a SAT solver, version one

• just enumerate assignments, and check formula for each
• for $k$ variables, $2^k$ assignments
• surely can do better?

SAT is hard

• in the worst case, no: you can’t do better
• Cook (1973): 3-SAT (3 literals/clause) is “NP-complete”
• the quintessential “hard problem” ever since

how to be a pessimist

• suppose you have a problem P (that is, a class of problems)
• show SAT reducible to P (ie, can translate a P-problem into a SAT-problem)
• then if P weren’t hard, SAT wouldn’t be either; so P is hard too
SAT is easy

remarkable discovery

- most SAT problems are easy
- can solve in much less than exponential time

how to be an optimist

- suppose you have a problem P
- reduce it to SAT, and solve with SAT solver

#boolean vars SAT solver can handle
(from Sharad Malik)
applications of SAT

configuration finding

\* solve \((\text{configuration rules})\) to obtain configuration
\* eg: generating network configurations from firewall rules
\* eg: course scheduling (http://andalus.csail.mit.edu:8180/scheduler/)

theorem proving

\* solve \((\text{axioms} \land \neg \text{theorem})\): valid if no assignment
\* hardware verification: solve \((\text{combinatorial logic design} \land \neg \text{specification})\)
\* model checking: solve \((\text{state machine design} \land \neg \text{invariant})\)
\* code verification: solve \((\text{method code} \land \neg \text{method spec})\)

more exotic application

\* solve \((\text{observations} \land \text{design structure})\) to obtain failure info
\* model-based diagnosis in deep space probes (http://mers.csail.mit.edu/)
why am we teaching you this?

SAT is cool

- good for (geeky) cocktail parties
- builds on your 6.042 knowledge

basic compiler techniques

- same ideas will work for any compiler or interpreter
the new paradigm
from machines to functions

6.005, part 1

• a program is a **state machine**

• computing is about taking state transitions on events

6.005, part 2

• a program is a **function**

• computing is about constructing and applying functions

an important paradigm

• functional or “side effect free” programming

• Haskell, ML, Scheme designed for this; Java not ideal, but it will do

• some apps are best viewed entirely functionally

• most apps have an aspect best viewed functionally
immutables

like mathematics, compute over values
- can reuse a variable to point to a new value
- but values themselves don’t change

why is this useful?
- easier reasoning: f(x) = f(x) is true
- safe concurrency: sharing does not cause races
- network objects: can send objects over the network
- performance: can exploit sharing

but not always what’s needed
- may need to copy more, and no cyclic structures
- mutability is sometimes natural (bank account that never changes?)
- [hence 6.005 part 3]
datatyp productions
recursive equations

idea: describe set of values with recursive equations

examples

• natural numbers
  
equation: Nat = 0 + Succ(Nat)
  values: 0, Succ(0), Succ(Succ(0)), ...

• lists of natural numbers
  
equation: List = Empty + Cons(first: Nat, rest: List)
  values: Empty, Cons(first:0, rest:Empty), Cons(first:0, rest:Cons(first:1, rest:Empty)), ...

structural form

• **datatype** name on left; **variants** separated by + on right
• each option is a **constructor** with zero or more named args
• equation is often called a **production**
interpretations

definitional interpretation (used for designing class structure)

\checkmark read left to right: an X is either a Y or a Z ...

\hspace{1cm} read $\text{List} = \text{Empty} + \text{Cons}(\text{first}: \text{Nat}, \text{rest}: \text{List})$

\hspace{1cm} as “a List is either an Empty list or a Cons of a Nat and a List”

inductive interpretation (used for designing functions)

\checkmark read right to left: if x is an X, then Y(x) is too ...

\hspace{1cm} “if l is a List and n is a Nat, then Cons(n, l) is a List too”

aren't these equations circular?

\checkmark yes: we are defining lists in terms of lists

\checkmark but well defined so long as \text{List} isn’t a RHS option

\checkmark definitional view: means smallest set of objects satisfying equation

\hspace{1cm} otherwise, can make Banana a List; then Cons(1, Banana) is a list too, etc.
formulas

productions

Formula = OrFormula + AndFormula + Not(Formula) + Literal(String)
OrFormula = Or(left:Formula, right:Formula)
AndFormula = And(left:Formula, right:Formula)

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

And(left:Or(left:Literal("P"), right:Literal("Q")),
right(left:Not(Literal("P")), right:Literal("R")))

sample formula: Socrates\(\Rightarrow\)Human \land Human\(\Rightarrow\)Mortal \land \neg (Socrates\Rightarrow\text{Mortal})
And(left: Or(left:Not(Literal("Socrates")), right:Literal("Human")),
right: And(left: Or(left:Not(Literal("Human")), right:Literal("Mortal")),
right: Not(Or(left: Not(Literal("Socrates")), right:Literal("Mortal")))))
drawing terms as trees

“abstract syntax tree” (AST) for Socrates formula
like language grammars?

programming language definitions

• define grammar with productions

just like our productions

• + often implicit (one option/line)

abstract syntax

• describes what the values are

• don’t have to worry about how to recognize them

• for example, Empty and Cons lists are tagged

concrete syntax

• how the values are represented

• need to parse to figure out which variant

from Java Language Spec
polymorphic datatypes

suppose we want lists over any type

- that is, allow list of naturals, list of formulas
- called “polymorphic” or “generic” lists

\[
\text{List}\langle E \rangle = \text{Empty} + \text{Cons}(\text{first: } E, \text{ rest: } \text{List}\langle E \rangle)
\]

- another example

\[
\text{Tree}\langle E \rangle = \text{Empty} + \text{Node}(\text{val: } E, \text{ left: Tree}\langle E \rangle, \text{ right: Tree}\langle E \rangle)
\]

application: environments

- in interpreter, environment holds values of variables
- assignment of boolean values to variables is an environment

\[
\text{Binding} = \text{Bind}(\text{key: Literal, val: Bool})
\]
\[
\text{Env} = \text{List}\langle \text{Binding} \rangle
\]

- what does equals sign mean in \( \text{Env} = \text{List}\langle \text{Binding} \rangle \) ?
  a tricky question to be discussed later (data abstraction)
deriving class structure
general strategy

exploit the definitional interpretation

› create an abstract class for the datatype
› and one subclass for each variant, with field for each arg

example

› production

  \[
  \text{List}<\text{E}> = \text{Empty} + \text{Cons} (\text{first}: \text{E}, \text{rest}: \text{List}<\text{E}>)
  \]

› code*

  ```java
  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;
    private Cons (E e, List<E> r) {first = e; rest = r;}
  }
  ```

*code handout uses longer names for variants; subclass names should stand alone
class structure for formulas

formula productions

Formula = OrFormula + AndFormula + Not(Formula) + Literal(String)
OrFormula = Or(left:Formula,right:Formula)
AndFormula = And(left:Formula,right:Formula)

code

public abstract class Formula {}
public class AndFormula extends Formula {
    private final Formula left, right;
    public AndFormula (Formula left, Formula right) {
        this.left = left; this.right = right;
    }
}
public class OrFormula extends Formula {
    private final Formula left, right;
    public OrFormula (Formula left, Formula right) {
        this.left = left; this.right = right;
    }
}
public class NotFormula extends Formula {
    private final Formula formula;
    public NotFormula (Formula f) {formula = f;}
}
public class Literal extends Formula {
    private final String name;
    public Literal (String name) {this.name = name;}
}
recursive traversals
kinds of functions

producers vs. observers

- observers: return values of another type
  - size: List<E> -> int
  - isEmpty: List<E> -> Bool
  - first: List<E> -> E
  - literals: Formula -> Set<Literal>
  - eval: Formula, Env -> Bool

- producers: return values of this datatype
  - rest: List<E> -> List<E>
  - normalize: Formula -> Formula
general strategy

exploit the inductive interpretation

- constructor without args: base case, function returns constant
- constructor with args: inductive case, make recursive call

example: size: List<E> -> int, where List<E> = Empty + Cons (first: E, rest: List<E>)

```java
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();}
}
```
caching results

look at this implementation

' representation is mutable, but abstractly object is still immutable!

```java
public abstract class List<E> {
    int size;
    boolean sizeSet;
    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 () {
        if (sizeSet) return size;
        int s = 1 + rest.size();
        size = s; sizeSet = true;
        return size;
    }
}
```
in this case, best just to set in constructor

' can determine size on creation -- and never changes* because immutable

```java
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}
}
```

*so why can't I mark it as final? ask the designers of Java ...
functions for SAT
naive SAT

simplest SAT solver: “generate-and-test”

- just generate all assignments
- test formula for each until get true, or run out

functions needed

- a function to get the set of literals in a formula
  \[ \text{literals: Formula} \rightarrow \text{Set<Literal>} \]
- a function to generate all environments from a given set of literals
  \[ \text{allEnvs: Set<Literal}> \rightarrow \text{Env}^* \]
- a function to evaluate the formula in a given environment
  \[ \text{eval: Formula, Env} \rightarrow \text{Bool} \]
example

• formula \( f = \)
  \[
  \text{Socrates} \Rightarrow \text{Human} \land \text{Human} \Rightarrow \text{Mortal} \land \neg (\text{Socrates} \Rightarrow \text{Mortal})
  \]

• literals(f) =
  \{ \text{Socrates, Human, Mortal} \}

• allEnvs(literals(f)) =
  \{ \{ \text{Socrates} \rightarrow \text{False}, \text{Human} \rightarrow \text{False}, \text{Mortal} \rightarrow \text{False} \},
  \{ \text{Socrates} \rightarrow \text{False}, \text{Human} \rightarrow \text{False}, \text{Mortal} \rightarrow \text{True} \},
  \{ \text{Socrates} \rightarrow \text{False}, \text{Human} \rightarrow \text{True}, \text{Mortal} \rightarrow \text{False} \},
  \{ \text{Socrates} \rightarrow \text{False}, \text{Human} \rightarrow \text{True}, \text{Mortal} \rightarrow \text{True} \},
  \{ \text{Socrates} \rightarrow \text{True}, \text{Human} \rightarrow \text{False}, \text{Mortal} \rightarrow \text{False} \},
  \{ \text{Socrates} \rightarrow \text{True}, \text{Human} \rightarrow \text{False}, \text{Mortal} \rightarrow \text{True} \},
  \{ \text{Socrates} \rightarrow \text{True}, \text{Human} \rightarrow \text{True}, \text{Mortal} \rightarrow \text{False} \},
  \{ \text{Socrates} \rightarrow \text{True}, \text{Human} \rightarrow \text{True}, \text{Mortal} \rightarrow \text{True} \} \}

• eval: false on all environments, so theorem is valid
in-class exercise

implement literals: Formula -> Set<Literal> given

```java
public interface Set<E> {
    Set<E> add (E e);
    Set<E> remove (E e);
    Set<E> addAll (Set<E> s);
    Set<E> removeAll (Set<E> s);
    boolean contains (E e);
    boolean isEmpty ();
    int size ();
}
```
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);
    }
}
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);
    }
}
public class Environment {
    Map <Literal, Boolean> bindings;
    public Env put(Literal l, boolean v) {
        return new Env (bindings.put (l, v));
    }
    public boolean get(Literal l) throws NoSuchLiteralException {
        Boolean b = bindings.get(l);
        if (b==null)
            throw new NoSuchLiteralException(l);
        else
            return b;
    }
}
generating all environments

this is not so easy

- don’t want a function to return entire set as one object (too big)
- hard to return one at a time, because Java’s control structures are too weak

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 polymorphic function declarations:

  \[
  \text{generate-and-test} : \text{Set}<\text{Literal}>, \text{Predicate} \to (\text{Env} + \text{None})
  \]

  \[
  \text{Predicate} = \text{Env} \to \text{Bool}
  \]
public interface Matcher <T> {
    // if match exists, return it, else return null
    T findMatch (Predicate<T> p);
}
public interface Predicate <T>{
    boolean test (T t);
}
public class EnvSet implements Matcher<Env> {
    public Env findMatch(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.findMatch (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 findMatch(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);
    }
}
barber puzzle

There is a barbers' club that obeys the following three conditions:
   (1) If any member A has shaved any other member B - whether
       himself or another - then all members have shaved A,
       though not necessarily at the same time.
   (2) Four of the members are named Guido, Lorenzo, Petrucio,
       and Cesare.
   (3) Guido has shaved Cesare.
Prove Petrucio has shaved Lorenzo
Puzzle PUZ003-1 from http://www.cs.miami.edu/~tptp

see solution in handout

• note how formula is constructed programmatically
• naive solver works fine in this case
• how many environments are considered?
summary
summary

big ideas

‣ SAT: an important problem, theoretically & practically
‣ datatype productions: a powerful way to think about sets of values
‣ polymorphic types and function decls: crucial structuring aid for code

where we are

‣ built a naive solver that works for small problems
‣ next time, a real SAT solver
lecture exercises

for Wednesday

the lion and the unicorn

• encode this as a SAT problem

  The Lion lies on Monday, Tuesday and Wednesday.
  The Unicorn lies on Thursday, Friday and Saturday.
  Both tell truth on other days.
  Both say yesterday was one of their lying days.
  Prove that today is Thursday.

• write Java code to construct the formula

• how elegantly can you do this?