6.005 Elements of Software Construction
Fall 2008

6.005 elements of software construction

how to design a SAT solver, part 1

Daniel Jackson

plan for today

topics

- demo: solving Sudoku
- what's a SAT solver and why do you want one?
- new paradigm: functions over immutable values
- big idea: using datatypes to represent formulas

today's patterns

- Variant as Class: deriving class structure
- Interpreter: recursive traversals

what's a SAT solver?

what is SAT?

the SAT problem

- given a formula made of boolean variables and operators

  $(P \lor Q) \land (\neg P \lor R)$

- find an assignment to the variables that makes it true
- possible assignments, with solutions in green, are:

  {P = false, Q = false, R = false}

  {P = false, Q = false, R = true}

  {P = false, Q = true, R = false}

  {P = false, Q = true, R = true}

  {P = true, Q = false, R = false}

  {P = true, Q = false, R = true}

  {P = true, Q = true, R = false}

  {P = true, Q = true, R = true}

what real SAT solvers do

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

- set of clauses, each containing a set of literals

  $\{\{P, Q\}, \{\neg P, R\}\}$

- 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?

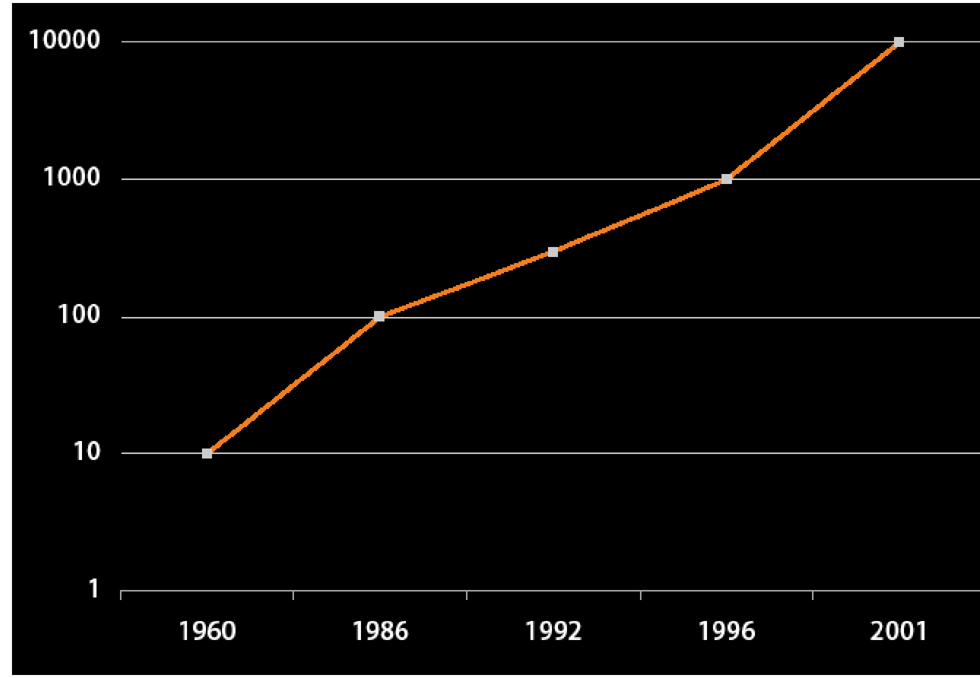
SAT is hard

- 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 any SAT-problem to a P-problem)
- then if P weren't hard, SAT wouldn't be either; so P is hard too

SAT is easy



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

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

applications of SAT

configuration finding

- solve (configuration rules $\land$ partial solution) to obtain configuration
- eg: generating network configurations from firewall rules
- eg: course scheduling (http://andalus.csail.mit.edu:8180/scheduler/)

theorem proving

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

more exotic application

- solve (observations $\land$ design structure) to obtain failure info
-

why are we teaching you this?

SAT is cool

- good for (geeky) cocktail parties
- you'll build a Sudoku solver for Exploration 2
- builds on your 6.042 knowledge

fundamental techniques

- you'll learn about datatypes and functions
- 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]

datatypes: describing structured values

modeling formulas

problem

- want to represent and manipulate formulas such as

  $(P \vee Q) \wedge (\neg P \vee R)$

- concerned about programmatic representation
- not interested in lexical representation for parsing

how do we represent the set of all such formulas?

- can use a grammar, but abstract not concrete syntax

datatype productions

- recursive equations like grammar productions
- expressions only from abstract constructors and choice
- productions define terms, not sentences

example: formulas

productions

Formula = OrFormula + AndFormula + Not(formula:Formula)+ Var(name:String)

OrFormula = OrVar(left:Formula,right:Formula)

AndFormula = And(left:Formula,right:Formula)

sample formula: $(P \vee Q) \wedge (\neg P \vee R)$

- as a term:

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

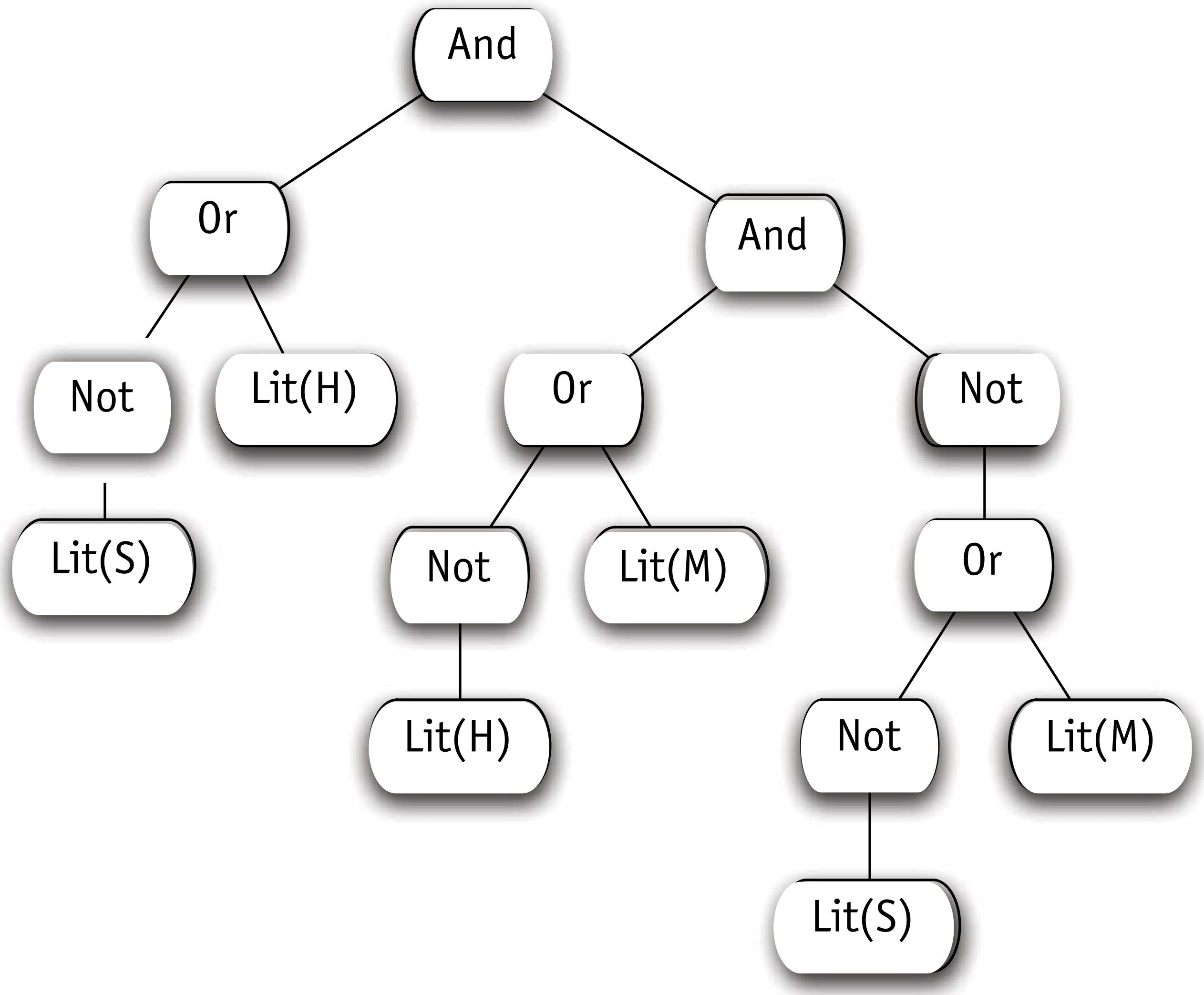
sample formula: Socrates⇒Human ∧ Human⇒Mortal ∧ ¬ (Socrates⇒Mortal)

- as a term:

And(Or(Not(Var("Socrates")),Var("Human")),
And (Or(Not(Var("Human")),Var("Mortal")),
Not(Or(Not(Var("Socrates")),Var("Mortal")))))

drawing terms as trees

"abstract syntax tree" (AST) for Socrates formula



other data structures

many data structures can be described in this way

- tuples: Tuple = Tup (fst: Object, snd: Object)
- options: Option = Some(value: Object) + None
- lists: List = Empty + Cons(first: Object, rest: List)
- trees: Tree = Empty + Node(val: Object, left: Tree, right: Tree)
- even natural numbers: Nat = 0 + Succ(Nat)

structural form of production

- **datatype** name on left; **variants** separated by + on right
- each option is a **constructor** with zero or more named args

what kind of data structure is Formula?

exercise: representing lists

writing terms

- write these concrete lists as terms
  - [] -- the empty list
  - [1] -- the list whose first element is 1
  - [1, 2] -- the list whose elements are 1 and 2
  - [[1]] -- the list whose first element is the list [1]
  - [[]] -- the list whose first element is the empty list

note

- <u>the</u> empty list, not <u>an</u> empty list
- we're talking <u>values</u> here, not <u>objects</u>

philosophical interlude

what do these productions mean?

definitional interpretation (used for designing class structure)

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

  read List = Empty + Cons(first: Nat, rest: List)

  as "a List is either an Empty list or a Cons of a Nat and a List"

inductive interpretation (used for designing functions)

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

  "if l is a List and n is a Nat, then Cons(n, l) is a List too"

aren't these equations circular?

- yes, but OK so long as List isn't a RHS option
- definitional view: means smallest set of objects satisfying equation

  otherwise, can make Banana a List; then Cons(1,Banana) is a list too, etc.

polymorphic datatypes

suppose we want lists over any type

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

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

- another example

```
Tree<E> = Empty + Node(val: E, left: Tree<E>, right: Tree<E>)
```

classes from datatypes

Variant as Class pattern

exploit the definitional interpretation

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

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;}
}
```

class structure for formulas

formula production

Formula = Var(name:String) + Not(formula: Formula)
 + Or(left: Formula,right: Formula) + 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 Var extends Formula {
    private final String name;
    public Var (String name) {this.name = name;}
    }
```

functions over datatypes

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();}
}
```

caching results

look at this implementation

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

```
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;
        }
}
```

size, finally

in this case, best just to set in constructor

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

```
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 ...

summary

summary

big ideas

- SAT: an important problem, theoretically & practically
- datatype productions: a powerful way to think about immutable types

patterns

- Variant as Class: abstract class for datatype, one subclass/variant
- Interpreter: recursive traversal over datatype with method in each subclass

where we are

- now we know how to represent formulas
- next time: how to solve them