datatypes

Daniel Jackson
October 15, 2009
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 = \text{false}, Q = \text{false}, R = \text{false}\}
  \{P = \text{false}, Q = \text{false}, R = \text{true}\}
  \{P = \text{false}, Q = \text{true}, R = \text{false}\}
  \{P = \text{false}, Q = \text{true}, R = \text{true}\}
  \{P = \text{true}, Q = \text{false}, R = \text{false}\}
  \{P = \text{true}, Q = \text{false}, R = \text{true}\}
  \{P = \text{true}, Q = \text{true}, R = \text{false}\}
  \{P = \text{true}, Q = \text{true}, R = \text{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

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

planning
\[- solve \ (initial\ state \land\ goal\land\ rules)\ to\ obtain\ plan\]
\[- eg,\ ZYpp\ package\ manager\ for\ Linux\]

verification
\[- solve\ (code\land\ \neg\ spec)\ to\ obtain\ counterexample\]
\[- industrial\ application\ to\ hardware;\ software\ applications\ coming\]
\[- eg,\ Cadence\ Incisive\ hardware\ verifier\]

design
\[- solve\ (design\ rules\land\ constraints\land\ requirements)\ to\ obtain\ design\]

for more info
\[- see\ http://www.satlive.org\]
why are we teaching you this?

SAT is cool
- good for (geeky) cocktail parties
- many useful applications
- 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 
  \((P \lor Q) \land (\neg P \lor R)\)

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

sample formula: \( (P \lor Q) \land (\neg P \lor R) \)
\* as a term:
\[
\text{And(Or(Var(“P”), Var(“Q”)), (Not(Var(“P”)), Var(“R”)))}
\]

sample formula: Socrates \( \rightarrow \) Human \land Human \( \rightarrow \) Mortal \land \neg (Socrates \rightarrow \text{Mortal})
\* as a term:
\[
\text{And(Or(Not(Var(“Socrates”)),Var(“Human”)), And (Or(Not(Var(“Human”)),Var(“Mortal”)), Not(Or(Not(Var(“Socrates”)),Var(“Mortal”)))))}
\]
drawing terms as trees

“abstract syntax tree” (AST) for Socrates formula

```
And
  Or
    Not Lit(H)
    Lit(S)
  Lit(H)
And
  Not
    Lit(M)
    Lit(H)
  Or
    Not
      Lit(M)
      Lit(S)
```
other data structures

many data structures can be described in this way

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

structural form of production

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

what kind of data structure is \text{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

• the empty list, not an empty list
• we’re talking values here, not objects
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 $\text{List} = \text{Empty} + \text{Cons(first: Nat, rest: List)}$
      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 $\text{List}$ 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

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

another example

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

```java
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

  \[\text{size: List}\langle E \rangle \rightarrow \text{int}\]

• break function into cases, one per variant

  \[\text{List}\langle E \rangle = \text{Empty} + \text{Cons(first: E, rest: List}\langle E \rangle)\]
  \[\text{size (Empty)} = 0\]
  \[\text{size (Cons(first: e, rest: l))} = 1 + \text{size(rest)}\]

• implement with one subclass method per case

  ```java
  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();}
  }
  ```
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;
    }
}
```
size, finally

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 ...
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