6.005 Elements of Software Construction
Fall 2008

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.
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 = \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”

\[ \{\{P, Q\}, \{\neg P, R\}\} \]

\[ \text{set of clauses, each containing a set of literals} \]

\[ \text{literal is just a variable, maybe negated} \]

SAT solver

\[ \text{program that takes a formula in CNF} \]

\[ \text{returns an assignment, or says none exists} \]
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)
Courtesy of Sharad Malik. Used with permission.

© Daniel Jackson 2008
applications of SAT

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

theorem proving
\* solve \((\text{axioms} \land \neg \text{theorem})\): 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
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

\cdot a program is a **state machine**
\cdot computing is about taking state transitions on events

6.005, part 2

\cdot a program is a **function**
\cdot computing is about constructing and applying functions

an important paradigm

\cdot functional or “side effect free” programming
\cdot Haskell, ML, Scheme designed for this; Java not ideal, but it will do
\cdot some apps are best viewed entirely functionally
\cdot 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

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

\- 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 = OrVar(left:Formula,right:Formula)
AndFormula = And(left:Formula,right:Formula)

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

sample formula: Socrates⇒Human \land Human⇒Mortal \land \neg (Socrates⇒Mortal)
\(\text{as a term:} \) 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
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

- **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

\[ \text{[]} -- \text{the empty list} \\
\[1\] -- \text{the list whose first element is 1} \\
\[1, 2\] -- \text{the list whose elements are 1 and 2} \\
\[[1]\] -- \text{the list whose first element is the list [1]} \\
\[[\text{ }\] -- \text{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)
\begin{itemize}
\item read left to right: an X is either a Y or a Z ...
\begin{itemize}
\item read \texttt{List = Empty + Cons(first: Nat, rest: List)}
\item as “a List is either an Empty list or a Cons of a Nat and a List”
\end{itemize}
\end{itemize}

inductive interpretation (used for designing functions)
\begin{itemize}
\item read right to left: if x is an X, then Y(x) is too ...
\begin{itemize}
\item “if l is a List and n is a Nat, then Cons(n, l) is a List too”
\end{itemize}
\end{itemize}

aren’t these equations circular?
\begin{itemize}
\item yes, but OK so long as \texttt{List} isn’t a RHS option
\item definitional view: means smallest set of objects satisfying equation
\end{itemize}

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

suppose we want lists over any type
  \[ \text{that is, allow list of naturals, list of formulas} \]
  \[ \text{called “polymorphic” or “generic” lists} \]
  \[ \text{List}<E> = \text{Empty} + \text{Cons(} \text{first: E, rest: List}<E> \text{)} \]
  \[ \text{another example} \]
  \[ \text{Tree}<E> = \text{Empty} + \text{Node(} \text{val: E, left: Tree}<E>, \text{ right: Tree}<E> \text{)} \]
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

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

© Daniel Jackson 2008
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;}
}
```

© Daniel Jackson 2008
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
  ```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();}
  }
  ```

© Daniel Jackson 2008
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}
}
```
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