6.005 Elements of Software Construction
Fall 2008

6.005 elements of software construction

designing a SAT solver, part 3

Daniel Jackson

plan for today

topics

- datatypes and structure
- the idea of data abstraction
- types and operations for DPLL
- example abstract types & design challenges
- designing an equals operation

patterns

- Factory Method (in Literal)

a datatype revisited

using sets

recall computing set of vars appearing in a formula

- declare function

  vars: F -> Set<Var>

- declare datatype

  F = Var(name:String) + Or(left:F,right:F) + And(left:F,right:F) + Not(formula:F)

- define function over variants

  vars (Var(n)) = {Var(n)}
  vars (Or(fl, fr)) = vars(fl) ∪ vars(fr)
  vars (And(fl, fr)) = vars(fl) ∪ vars(fr)
  vars (Not(f)) = vars(f)

where do sets come from?

- defined structurally like this

  Set<T> = List<T>

- but should be defined by <u>operations</u> instead: {}, ∪

a set interface

```
public interface Set<E> {
    public Set<E> add (E e);
    public Set<E> remove (E e);
    public Set<E> addAll (Set<E> s);
    public boolean contains (E e);
    public E choose ();
    public boolean isEmpty ();
    public int size ();
}
```

a set implementation

```
public class ListSet<E> implements Set<E> {
private List<E> elements;

public ListSet () {elements = new EmptyList<E> ();}

public Set<E> add (E e) {
    if (elements.contains (e)) return this;
    return new ListSet<E> (elements.add (e));
}

public Set<E> remove (E e) {
    if (isEmpty()) return this;
    E first = elements.first();
    ListSet<E> rest = new ListSet<E> (elements.rest());
    if (first.equals(e))
        return rest;
    else
        return rest.remove(e).add(first);
}

public boolean contains (E e) {
    return elements.contains(e);
}

...}
```

a new viewpoint

datatype productions

- datatypes defined by their structure or representation

abstract datatypes

- datatypes defined by their operations or behavior

extending the type repertoire

- used to thinking of basic types behaviourally:

    integers: +, *, <, =

    array: get(a,i), store(a,i,e)

- abstract datatypes: user-defined types

    string: concat(s,t), charAt(s,i)

    set: {}, ∪, ∈

what makes an abstract type?

defined by operations

- an integer <u>is</u> something you can add, multiply, etc
- a set is something you can test membership in, union, etc

representation is hidden or "encapsulated"

- client can't see how the type is represented in memory
- is integer twos-complement? big or little endian?
- is set a list? a binary tree? an array?

language support for data abstraction

- packaging operations with representations
- hiding representation from clients

encapsulation

two reasons for encapsulation of representations

rep independence

- if client can't see choice of rep, implementor can change it
- eg: integers: your program can run on a different platform
- eg: sets: programmer can switch rep from list to array

rep invariants

- not all values of the rep make legal abstract values
- prevent client from accessing rep so code of ADT can preserve invariants
- eg: sets: make sure element does not appear twice

classic types

domain specific and generic types

- some types are specific to a domain (clause, literal)
- some have wide application (list, set)
- widely applicable types are usually polymorphic
- these are the "classic ADTs"

in Java

- found in the standard package `java.util`
- often called "Java collection framework"

a zoo of types

type	overview	producers	observers	common reps
list	sequence for concatenation and front-append	add, append	first, rest, ith	array, linked list
queue	FIFO: first in, first out	enq, deq	first	array, list, circular buffer
stack	LIFO: last in, first out	push, pop	top	array, list
map	associates keys and values	put	get	association list, hash table, tree
set	unordered collection	insert, remove	contains	map, list, array, bitvector, tree
bag	like set, but element can appear more than once	insert, remove	count	map, array, association list

note

- producers and observers: just examples
- common reps: some (eg, hash table, bitvector) just for mutable versions

the DPLL algorithm

what types do you need?

a square root procedure needs

- floating point numbers

a SAT solver needs

- booleans, literals, clauses, environments

characteristic of complex programs

- computations defined over set of datatypes
- most of the datatypes are not built-in, but **user-defined**
- so design datatypes before other program components

let's examine the DPLL algorithm

- and see what types it needs

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

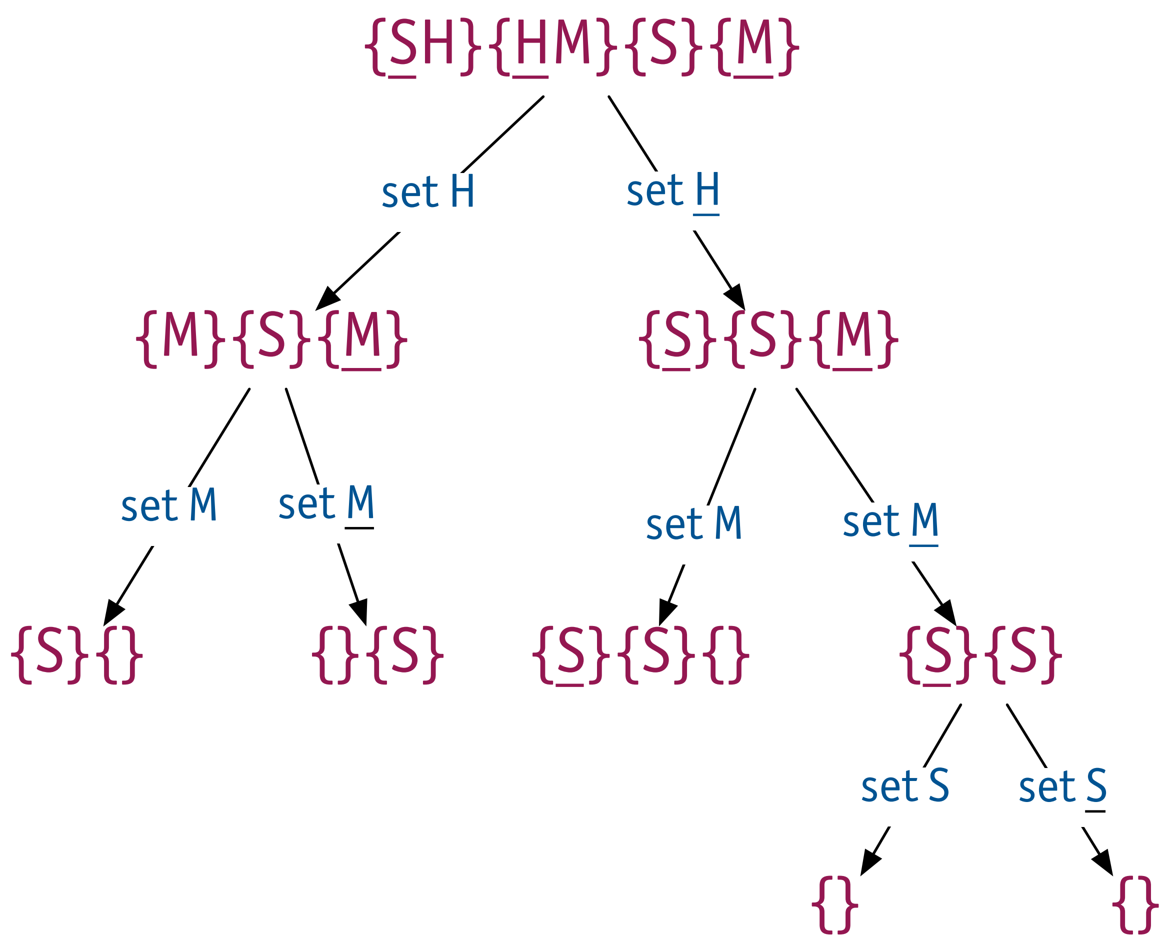
elements of the algorithm

- backtracking search: pick a literal, try false then true
- if clause set is empty, success
- if clause set contains empty clause, failure

example

- 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: {<u>S</u>H}{<u>H</u>M}{S}{<u>M</u>}

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^#literals

DPLL

classic SAT algorithm

- Davis-Putnam-Logemann-Loveland, 1962

unit propagation

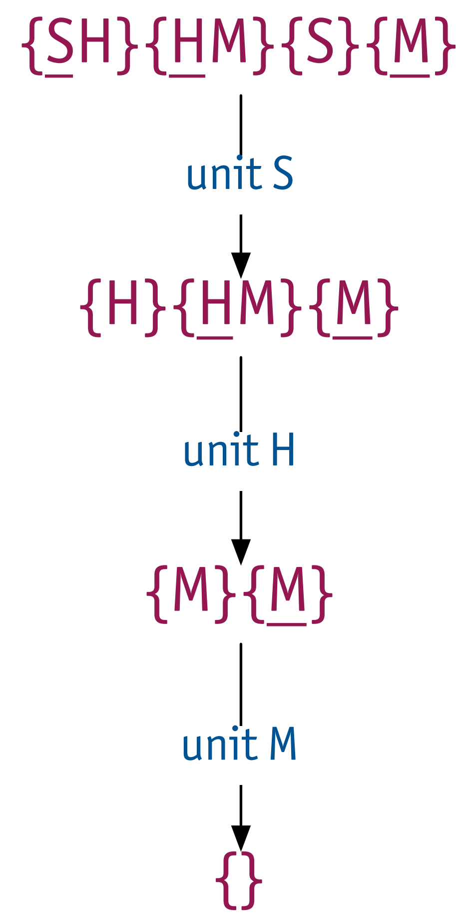
- on top of backtracking search
- if a clause contains one literal, set that literal to true

example (on right)

- in this case, no splitting needed
- propagate S, then H, then M

performance

- often much better, but worst case still exponential



an implementation

```
 public static Environment solve(List<Clause> clauses) {
    return solve (clauses, new Environment());}

private static Environment solve(List<Clause> clauses, Environment env) {
    if (clauses.isEmpty()) return env; // if no clauses, trivially solvable
    Clause min = null;
    for (Clause c : clauses) {
        if (c.isEmpty()) return null; // if empty clause found, then unsat
        if (min == null || c.size() < min.size()) min = c;
        }
    Literal l = min.chooseLiteral();
    bool.Variable v = l.getVariable();
    if (min.isUnit()) { // a unit clause was found, so propagate
        env = env.put(v, l instanceof PosLiteral ? Bool.TRUE : Bool.FALSE);
        return solve(reduceClauses (clauses,l), env);
    } // else split
    if (l instanceof NegLiteral) l = l.getNegation();
    Environment solvePos = solve (reduceClauses (clauses,l), env.put(v, Bool.TRUE));
    if (solvePos == null)
        return solve (reduceClauses (clauses,l.getNegation()), env.put(v, Bool.FALSE));
    else return solvePos;
}

private static List<Clause> reduceClauses(List<Clause> clauses, Literal l) {
    List<Clause> reducedClauses = new EmptyList<Clause>();
    for (Clause c : clauses) {
        Clause r = c.reduce(l);
        if (r != null)
            reducedClauses = reducedClauses.add(r);
    }
    return reducedClauses;
}
```

basic types for SAT

types and operations

```
public static Environment solve(List<Clause> clauses) {
    return solve (clauses, new Environment());}

private static Environment solve(List<Clause> clauses, Environment env) {
    if (clauses.isEmpty()) return env; // if no clauses, trivially solvable
    Clause min = null;
    for (Clause c : clauses) {
        if (c.isEmpty()) return null; // if empty clause found, then unsat
        if (min == null || c.size() < min.size()) min = c;
        }
    Literal l = min.chooseLiteral();
    bool.Variable v = l.getVariable();
    if (min.isUnit()) { // a unit clause was found, so propagate
        env = env.put(v, l instanceof PosLiteral ? Bool.TRUE : Bool.FALSE);
        return solve(reduceClauses (clauses,l), env);
    } // else split
    if (l instanceof NegLiteral) l = l.getNegation();
    Environment solvePos = solve (reduceClauses (clauses,l), env.put(v, Bool.TRUE));
    if (solvePos == null)
        return solve (reduceClauses (clauses,l.getNegation()), env.put(v, Bool.FALSE));
    else return solvePos;
}

private static List<Clause> reduceClauses(List<Clause> clauses, Literal l) {
    List<Clause> reducedClauses = new EmptyList<Clause>();
    for (Clause c : clauses) {
        Clause r = c.reduce(l);
        if (r != null)
            reducedClauses = reducedClauses.add(r);
    }
    return reducedClauses;
}
```

bool type

introduced my own boolean ADT

- has three boolean values: TRUE, FALSE and UNDEFINED
- why did I do this?

```
public enum Bool {
    TRUE, FALSE, UNDEFINED;
    public Bool and (Bool b) {
        if (this==FALSE || b==FALSE) return FALSE;
        if (this==TRUE && b==TRUE) return TRUE;
        return UNDEFINED;
    }
    public Bool or (Bool b) {
        if (this==FALSE && b==FALSE) return FALSE;
        if (this==TRUE || b==TRUE) return TRUE;
        return UNDEFINED;
    }
    public Bool not () {
        if (this==FALSE) return TRUE;
        if (this==TRUE) return FALSE;
        return UNDEFINED;
    }
}
```

environment type

should Environment be an ADT at all?

- just a mapping from literals to booleans
- decided yes, in case I wanted to add functionality later
- sure enough, I did: return Bool.UNDEFINED if no mapping

```
public class Environment {
    private Map <Variable, Bool> bindings;

    public Environment put(Variable v, Bool b) {
       return new Environment (bindings.put (v, b));
    }

    public Bool get(Variable v){
        Bool b = bindings.get(v);
        if (b==null) return Bool.UNDEFINED;
        else return b;
    }
   ...
}
```

clause type

what's a clause?

- clause is disjunction of set of literals; empty means FALSE, no rep of TRUE

```
public class Clause {
  public Clause() {...}
  public Clause(Literal literal) {...}
  public Clause add(Literal l) {...}
  public Clause reduce(Literal literal) {...}
  public Literal chooseLiteral() {...}
  public boolean isUnit() {...}
  public boolean isEmpty() {...}
  public int size() {...}
  }
```

notes

- order not exposed in observers: chooseLiteral is non-deterministic
- isUnit, isEmpty are for convenience of clients, not strictly necessary
- add, reduce are the key 'producers':

  add (l): return clause obtained by adding l as a disjunct
  reduce (l): return clause obtained by setting l to TRUE

designing operations

issue

- what should add, reduce return when result is TRUE? eg, add S to {S}

design options

- create clause for special value TRUE
- throw an exception
- return null

considerations

- clause set should not contain vacuous TRUE clauses
- exceptions are awkward; in Java, best used only when not expected
- compiler doesn't ensure that null return value is checked

representation independence

choice of rep

an abstract type can be implemented with different reps

- example: two versions of Environment

```
public class Environment {
    private Map <Variable, Bool> bindings;
    ...
    public Bool get(Variable v){
        Bool b = bindings.get(v);
        if (b==null) return Bool.UNDEFINED;
        else return b;
    }
}

public class Environment {
   private Set <Variable> trues, falses;
   ...
   public Bool get(Variable v){
       if (trues.contains (v)) return Bool.TRUE;
       if (falses.contains (v)) return Bool.FALSE;
       return Bool.UNDEFINED;
   }
}
```

achieving rep independence

rep independence

- want to be able to change rep without changing client

what does this require?

- if client can access fields directly
  - rep is fully "exposed": heavy modification of client code required
- if client calls methods that return fields directly
  - can fix by modifying ADT methods, but will be ugly
- if client can't access fields even indirectly (as in previous slide)
  - ADT is easily modified locally

so independence is achieved by

- combination of language support and programmer discipline

designing equality

comparing literals

need to compare literals

- eg, in Clause.reduce

  eg, when S is true: {S̲H} reduces to {H}, and {SH} reduces to TRUE

- 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 ‘factory’ method

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

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

```
public abstract class Literal {
    protected Literal negation;
    protected Variable var;
    public Literal (Variable name) {this.var = new bool.Variable(name);}
}
public 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;
    }
```

putting it all together: demo

allocating variables

Sudoku abstract type contains

- 2D array of known values (square)
- 3D array of boolean variables (occupies)

```
public class Sudoku {
    private final int dim;
    private final int size;
    private int [][] square;
    private Formula [][][] occupies;

    public Sudoku (int dim) {
        this.dim = dim;
        size = dim * dim;
        square = new int [size][size];
        occupies = new Formula [size][size][size];
        for (int i = 0; i < size; i++)
            for (int j = 0; j < size; j++)
                for (int k = 0; k < size; k++) {
                    Formula l = Formula.makeVariable ("occupies(" + i + ","+ j + ","+ k + ")");
                    occupies[i][j][k] = l;
                }
    }

    public static Sudoku fromFile (String filename, int dim) {...}
```

creating formula

to create formula

- create at-most and at-least formulas per row, column, block
- my solver converts to CNF

```
public Formula getFormula () {
    Formula 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);
        }
    ...
    return formula;
}
```

interpreting the solution

to interpret solution

- just iterate over puzzle, and look up each variable in environment

```
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++) {
                Formula l = occupies[i][j][k];
                if (l.eval(e) == Bool.TRUE)
                    row = row + (k+1) + "|";
            }
        result = result + row + "\n";
    }
    return result;
}
```

executing the solver

steps

- create Sudoku object from file
- extract formula, solve and interpret

```
public static void solveStandardPuzzle (String filename) throws IOException {
    long started = System.nanoTime();
    System.out.println ("Parsing...");
    Sudoku s = Sudoku.fromFile (filename, 3);
    System.out.println ("Creating SAT formula...");
    Formula f = s.getFormula();
    System.out.println ("Solving...");
    Environment e = f.solve();
    System.out.println ("Interpreting solution...");
    String solution = s.interpretSolution(e);
    System.out.println ("Solution is: \n" + solution);
    long time = System.nanoTime();
    long timeTaken = (time - started);
    System.out.println ("Time:" + timeTaken/1000000 + "ms");
}
```

sample run

solving a sample Sudoku puzzle

- 1,000 variables and 24,000 clauses
- about 10 seconds (on 2.4GHz Intel Mac with 2GB memory)

```
Parsing...
Creating SAT formula...
Solving...
Interpreting solution...
Solution is:
|9|1|6|8|4|3|5|2|7|
|8|4|2|7|5|6|9|3|1|
|7|5|3|2|9|1|8|6|4|
|3|6|4|9|2|7|1|8|5|
|2|8|1|5|6|4|7|9|3|
|5|9|7|1|3|8|2|4|6|
|6|7|8|4|1|9|3|5|2|
|4|2|9|3|7|5|6|1|8|
|1|3|5|6|8|2|4|7|9|

Time:9211ms
```

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 {$\overline{ABC}$}
- 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

- "In Classic Math Riddle, DNA Gives a Satisfying Answer", George Johnson, New York Times, March 19, 2002

summary

summary

principles

- define an abstract type by its operations
- hide the representation from clients

patterns

- Factory Method