datatypes, part 1

Daniel Jackson
March 10, 2010
plan for today

topics

‣ SAT as a motivating example: why?
‣ 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

Note that CNF is just a format for a boolean formula -- but one that turns out to be very helpful, making it easier to write solvers. The notion of a literal is important, since it means you can only negate variables, and not clauses.
how to build a SAT solver, version one

\begin{itemize}
  \item just enumerate assignments, and check formula for each
  \item for \( k \) variables, \( 2^k \) assignments: surely can do better?
\end{itemize}

SAT is hard

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

how to be a pessimist

\begin{itemize}
  \item suppose you have a problem \( P \) (that is, a class of problems)
  \item show SAT reducible to \( P \) (ie, can translate any SAT-problem to a \( P \)-problem)
  \item then if \( P \) weren’t hard, SAT wouldn’t be either; so \( P \) is hard too
\end{itemize}
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

In the last 1980s, researchers were publishing papers on how to find hard SAT problems! It turned out that even though in the worst case SAT is really hard, in practice almost all the cases you get if you generate them randomly are easy. And the ones that arise in real problems are often easy too. The story’s actually more complicated than this though; it turns out that there’s what’s called a ‘phase transition’, a point at which problems get really hard, and this phase transition is roughly at the midpoint between the two extremes of the formula being so constrained it’s easy to solve because you can easily determine values for variables early on, and the formula being so underconstrained that it’s easy to solve just by guessing.
This is just a sample of SAT applications. There are many more. In my own research group, we developed a tool called Alloy that uses SAT to analyze software designs. It’s also been used to analyze state machines using a method called ‘bounded model checking’.
why are we teaching you this?

SAT is cool
- good for (geeky) cocktail parties
- many useful applications
- compilation-to-SAT idea is powerful
- 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

We’ll actually write pseudo code for functions over datatypes using a notation that is very similar to a programming language called ML. Then we’ll convert to Java using some standard patterns. Some of these patterns are taken from the book “A Little Java, A Few Patterns” by Matthias Felleisen and Daniel P. Friedman, which shows how to emulate the functional style in Java. When you’re comfortable with the idea of functions over values, you’ll be able to integrate it smoothly into your programming practice, so that you can write programs that mix these idioms.
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]

The idea of immutables is one of those restriction paradoxes: it’s not obvious at first why being so restrictive helps, but it’s actually liberating, because once you get rid of mutation lots of things becoming possible -- in particular sharing structures without worrying about whether you should be copying.
datatypes: describing structured values
The list notation \([a, b, c]\) means a list whose first element is \(a\), second is \(b\), third is \(c\). An empty list is written \([\]\). This exact notation can be used in many programming languages, but not in Java.

By \(\{0->\text{“csail”}, 1->\text{“mit”}, 2->\text{“edu”}\}\), I mean the function that maps 0 to the string “csail”, 1 to the string “mit” and 2 to the string “edu”. Representing things like this is called “model based” because everything is ‘modelled’ using standard mathematical constructs such as sets and functions, and it’s the view we’ll take in the third part of the course.
lists as recursive structures

first and rest

• view list as consisting of first element and rest of list
  eg, [“csail”, “mit”, “edu”] has first element “csail” and rest [“mit”, “edu”]
  [“mit”, “edu”] has first element “mit” and rest [“edu”]
  [“edu”] has first element “edu” and rest []

• this is recursive:
  a list is a first element (which is a thing), and the rest (which is a list)

writing definition as a datatype with constructors

• List = Empty + Cons(first: Object, rest: List)

• now can write list like this as a term or expression
  Cons (“csail”, Cons (“mit”, Cons (“edu”, Empty)))

• or draw a graph... (as drawn in class)

There’s a dilemma I face when declaring arguments. The syntax x:T to mean an argument x of type T is very standard in Algol–like languages (Pascal, Modula, Ada) and in typed functional languages (such as ML). But in Java, which follows C, we’d write the declaration as “T x”. That just seems wrong to me since these declarations have a more mathematical flavour; also, it’s quite natural to write “x, y: T”, but you can’t write “T x, y” in Java. Hmm.
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

Note that not all the datatypes are recursive. The notion is most useful for describing unbounded types (that is types, that can contain objects, such as trees, that can be arbitrarily large), but it works for simpler finite types too.

The term “constructor” doesn’t necessarily mean Java constructor, although we’ll see that it can often be implemented that way.
There’s only one empty list, because it’s a value: just as there’s only one zero. Answers: Empty, Cons(1, Empty), Cons(1, Cons(2, Empty)), Cons(Cons(1, Empty), Empty), Cons(Empty, Empty)
exercise: trees

represent this tree as a term
  • use a decl from previous slide

represent this tree as a term
  • write a datatype decl first

The second tree only has numbers at the leaves.
First tree: \[\text{Node}(1, \text{Node}(2, \text{Node}(3, \text{Leaf}(1), \text{Leaf}(2)), \text{Node}(4, \text{Leaf}(1), \text{Leaf}(2))), \text{Node}(5, \text{Leaf}(1), \text{Leaf}(2)))\]
Second tree:
Datatype decl: \[\text{Tree} = \text{Leaf}(\text{val}: \text{int}) + \text{Node}(\text{left}, \text{right}: \text{Tree})\]
Term: \[\text{Node}(\text{Node}(\text{Leaf}(1), \text{Leaf}(2)), \text{Leaf}(3))\]
note that this decl can’t express the tree you’d get if you removed the node marked 3; for that, you’d need to add a variant \text{Empty} to the decl.
functions on datatypes

why is this representation appealing?

• can handle unbounded structures
  (and actually infinite ones too)
• easy way to construct in programs
• nice form for defining functions

recipe: how to build a recursive traversal

• write type declaration of function
  \[ \text{size: List} \rightarrow \text{int} \]
• break function into cases, one per variant
  \[ \text{List} = \text{Empty} + \text{Cons(first: Object, rest: List)} \]
  \[ \text{size (Empty)} = 0 \]
  \[ \text{size (Cons(first: e, rest: l))} = 1 + \text{size(l)} \]

The notation ‘size: List \rightarrow int’ means that size is a function from lists to integers. Note the pattern matching in the function definition. The second clause says, for example, that the size of a list that is a cons of a first element e and a rest l is one more than the size of l. You can actually write this program just like that in ML, and the compiler will check that you’ve covered all cases. Cute, eh? We’ll be able to do something like this in Java, and use the compiler likewise to check that every case is covered.
can represent the execution by reduction steps:

\[
\text{size}(\text{Cons} \ (1, \text{Cons} \ (2, \text{Empty}))) \\
= 1 + \text{size}(\text{Cons} \ (2, \text{Empty})) \\
= 1 + (1 + \text{size}(\text{Empty})) \\
= 1 + (1 + 0) \\
= 2
\]

When a functional program executes, no structure is modified, so we can think of execution as a series of reductions -- just like doing arithmetic.
write a function that
• takes a list of integers and
• (1) computes the sum of the elements
• (2) adds one to each element
• (3) filters out negative elements

note
• we use operational language that suggests mutation
• but we really mean
  ‘returns a list each of whose elements is obtained by adding one to the corresponding element in the original list’ (phew!)

Functions 2 and 3 are trickier because they require constructing lists. Note also that both assume a list containing only integers. We’ll see soon how to specify that. For now we’ll just write List<Int> to say that the elements are integers.

sum: List -> int
sum (Empty) = 0
sum (Cons(first:e, rest: l)) = e + sum(l)

incr: List<Int> -> List<Int>
incr (Empty) = Empty
incr (Cons(first:e, rest: l)) = Cons(e+1, incr(l))

filter: List<Int> -> List<Int>
filter (Empty) = Empty
filter (Cons(first:e, rest: l)) = if e < 0 then filter(l) else Cons(e, filter(l))
how about arithmetic expressions such as this?

1 + (3 * 2)

define recursive datatype

\[ \text{Expr} = \text{Num}(i:\text{int}) + \text{PlusExpr} (\text{left}, \text{right}: \text{Expr}) + \text{TimesExpr} (\text{left}, \text{right}: \text{Expr}) \]

declare function

\[ \text{eval}: \text{Expr} \rightarrow \text{int} \]

break function into cases, one per variant

\[ \text{eval} (\text{Num}(i)) = i \]
\[ \text{eval} (\text{PlusExpr}(l,r)) = \text{eval}(l) + \text{eval}(r) \]
\[ \text{eval} (\text{TimesExpr}(l,r)) = \text{eval}(l) \times \text{eval}(r) \]
how about arithmetic expressions such as this?

\[ x + (3 \times y) \]

given environment \{x->1, y->2\}
evaluates to 7

define recursive datatype

\[ Expr = Num(i:int) + Var(v:String) \]
\[ + PlusExpr \ (left, \ right: \ Expr) + TimesExpr \ (left, \ right: \ Expr) \]

declare function

\[ eval: \ Expr, \ Env \rightarrow \ int \]

break function into cases, one per variant

\[ eval \ (Num(i), \ e) = i \]
\[ eval \ (Var(v), \ e) = lookup(e, v) \]
\[ eval \ (PlusExpr(l, r), \ e) = eval(l, e) + eval(r, e) \]
\[ eval \ (TimesExpr(l, r), \ e) = eval(l, e) \times eval(r, e) \]

This is getting pretty close to how an interpreter for a programming language works. An important issue is how to express changes to the environment, produced by variable declarations and assignments.
philosophical interlude

what do these productions mean?

definitional interpretation (used for designing class structure)
\[ \text{read left to right: an } X \text{ is either a } Y \text{ or a } Z \ldots \]
\[ \text{read } \text{List} = \text{Empty} + \text{Cons(first: Nat, rest: List)} \]
\[ \text{as } "\text{a List is either an Empty list or a Cons of a Nat and a List}" \]

inductive interpretation (used for designing functions)
\[ \text{read right to left: if } x \text{ is an } X, \text{ then } Y(x) \text{ is too } \ldots \]
\[ "\text{if } l \text{ is a List and } n \text{ is a Nat, then Cons(n, l) is a List too}" \]

aren't these equations circular?
\[ \text{yes, but OK so long as } \text{List} \text{ isn't a RHS option} \]
\[ \text{definitional view: means smallest set of objects satisfying equation} \]
\[ \text{otherwise, can make Banana a List; then Cons(1, Banana) is a list too, etc.} \]
polymorphic datatypes

suppose we want lists over any type

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

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

- another example

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

You can think of a polymorphic datatype decl as a ‘schema’; for each value of the parameter, it generates a particular concrete datatype declaration.
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
  
  \(\text{List}\langle\text{E}\rangle = \text{Empty} + \text{Cons} (\text{first}: \text{E}, \text{rest}: \text{List}\langle\text{E}\rangle)\)

• 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;}
}
```

27

Code in repository uses longer names for variants; subclass names should stand alone, hence “EmptyList”, not “Empty”. Should getters for Cons be declared in List? Tricky question. Makes it easier for clients, because no need to cast, but also makes errors more likely.
class structure for formulas

formula production

\[
\text{Formula} = \text{Var(name:\text{String})} + \text{Not(formula: Formula)} \\
+ \text{Or(left: Formula, right: Formula)} + \text{And(left: Formula, right: Formula)}
\]

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;}
}
```

Nothing interesting going on here, and all a bit verbose unfortunately. But this structure is easy to setup, and once you've set it up you can define functions on the datatype easily.
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

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

‣ 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();}
}
```

Note how the compiler will make sure that all variants are handled, since a subclass will be abstract (and therefore uninstantiable) unless the method size is implemented in it.
This is an interesting and important case: the type is immutable from the point of view of how it appears, but it uses a little bit of mutation in its implementation. Sometimes called a ‘beneficent side effect’.
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 ...

Size is a special case, because it can be easily precomputed. Other functions are different, as they might be costly to compute, or take additional arguments.
example: formulas
representing 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

solution

‣ represent using datatypes

‣ for now, allow arbitrary formulas; CNF later

You can think of datatype decls are being like little grammars -- just like the kind you’ve seen for specifying the form of programs, or commands to a shell, etc. But unlike grammars, which show the concrete form the objects take (ie, their physical appearance -- in this case, including the parens), datatype decls represent only the conceptual shape. Put another way, datatype decls don’t help you parse, but are good for representing the results of parsing.
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: \((P \lor Q) \land (\neg P \lor R)\)

\(\text{as a term:}\)

\[\text{And(Or(Var("P"), Var("Q")), (Not(Var("P")), Var("R")))}\]

sample formula: \(\text{Socrates} \Rightarrow \text{Human} \land \text{Human} \Rightarrow \text{Mortal} \land \neg (\text{Socrates} \Rightarrow \text{Mortal})\)

\(\text{as a term:}\)

\[\text{And(Or(Not(Var("Socrates")),Var("Human")),}
\text{And (Or(Not(Var("Human")),Var("Mortal")),}
\text{Not(Or(Not(Var("Socrates")),Var("Mortal"))))))}\]
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: functions on formulas, how to solve them