6.005 elements of software construction

Little Languages

Rob Miller

Fall 2011

Today's Topics

Functionals

- Objects representing executable code

Higher-order functions

- Functions that accept functions as arguments or return them as results

Domain-specific languages

- PCAP: primitives, combination, abstraction pattern

Representing Code with Data

Consider a datatype representing language syntax

- Formula is the language of propositional logic formulas
- a Formula value *represents* program code in a data structure; i.e.

  new And(new Var("x"), new Var("y"))

  has the same semantic meaning as the *Java code*

  x && y
- but a Formula value is a **first-class** object
  - first-class: a value that can be passed, returned, stored, manipulated
  - the Java expression "x && y" is *not* first-class

Representing Code as Data

Recall the visitor pattern

- A visitor represents a function over a datatype
  - e.g. new SizeVisitor() represents size : List → int

```
public class SizeVisitor<E> implements ListVisitor<E,Integer> {
    public Integer visit(Empty<E> l) { return 0; }
    public Integer visit(Cons<E> l) { return 1 +
l.rest().accept(this); }
}
```

A visitor represents code as a first-class object, too

- A visitor is an **object** that can be passed around, returned, and stored
- But it's also a **function** that can be invoked

Today's lecture will see more examples of code as data

Today's Problem: Music

Interesting music tends to have a lot of repetition

- Let's look at rounds, canons, fugues
- A familiar simple **round** is "Row Row Row Your Boat": one voice starts, other voices enter after a delay

  Row row row your boat, gently down the stream, merrily merrily ...

  Row row row your boat, gently down the stream...
- Bach was a master of this kind of music
  - Recommended reading: *Godel Escher Bach,* by Douglas Hofstadter

Recall our MIDI piano from early lectures

- A song could be represented by Java code doing a sequence of calls on a state machine:

  machine.play(E); machine.play(D); machine.play(C); ...
- We want to capture the code that operates this kind of machine as first-class **data objects** that we can manipulate, transform, and repeat easily

Music Data Type

Let's start by representing simple tunes

Music = Note(duration:double, pitch:Pitch, instr:Instrument)
            + Rest(duration:double)
            + Concat(m1:Music, m2:Music)

- duration is measured in *beats*
- Pitch represents note frequency (e.g. C, D, E, F, G; essentially the keys on the piano keyboard)
- Instrument represents the instruments available on a MIDI synthesizer

Design questions

- is this a tree or a list? what would it look like defined the other way?
- what is the "empty" Music object?
  - it's usually good for a data type to be able to represent *nothing*
  - avoid null
- what are the rep invariants for Note, Rest, Concat?

A Few of Music's Operations

notes : String x Instrument → Music

   requires string is in a subset of abc music notation

      e.g. notes("E D C D | E E E2 |", PIANO)

      1 beat note   2-beat note

> abc notation can also encode sharps & flats, higher/lower octaves

duration : Music → double

   returns total duration of music in beats

      e.g. duration(Concat(m1, m2)) = duration(m1) + duration(m2)

transpose : Music x int → Music

   returns music with all notes shifted up or down in pitch by the given number of semitones (i.e., steps on a piano keyboard)

play : Music → void

   effects plays the music

> all these operations also have precondition that parameters are non-null

Implementation Choices

Creators can be constructors or factory methods

- Java constructors are limited: interfaces can't have them, and constructor can't choose which runtime type to return
  - new C() must always be an object of type C,
  - so we can't have a constructor Music(String, Instrument), whether Music is an interface or an abstract class

Observers & producers can be methods or visitors

- Methods break up function into many files; visitor is all in one place
- Adding a method requires changing source of classes (not always possible)
- Visitor keeps dependencies out of data type itself (e.g. MIDI dependence)
- Method has direct access to private rep; visitor needs to use observers

Producers can also be new subclasses of the datatype

- e.g. Music = ... + Transpose(m:Music, semitones:int)
- Defers the actual evaluation of the function
- Enables more sharing between values
- Adding a new subclass requires changing all visitors

Duality Between Interpreter and Visitor

Operation using interpreter pattern

- Adding new operation is hard (must add a method to every existing class)
- Adding new class is easy (changes only one place: the new class)

Operation using visitor pattern

- Adding new operation is easy (changes only one place: the new visitor)
- Adding new class is hard (must add a method to every existing visitor)

Multiple Voices

For a round, the parts need to be sung simultaneously

Music = Note(duration:double, pitch:Pitch, instr:Instrument)
+ Rest(duration:double)
+ Concat(m1:Music, m2:Music)
+ Together(m1:Music, m2:Music)

- Here's where our decision to make Concat() tree-like becomes very useful
  - Suppose we instead had:

    Concat = List<Note + Rest>

    Together = List<Concat>
  - What kinds of music would we be unable to express?

Composite pattern

- The composite pattern means that groups of objects (*composites*) can be treated the same way as single objects (*primitives*)
- $T = C_1(\ldots, T) + \ldots + C_n(\ldots, T) + P_1(\ldots) + \ldots + P_m(\ldots)$

composites

primitives

Music and Formula are composite data types.

Simple Rounds

We need one more operation:

delay : Music x double → Music

delay(m, dur) = concat(rest(dur), m)

And now we can express Row Row Row Your Boat

rrryb = notes(“C C C3/4 D/4 E | E3/4 D/4 E3/4 F/4 G2 | ...”, PIANO)

together(rrryb, delay(rrryb, 4))

- Two voices playing together, with the second voice delayed by 4 beats

➢ This pattern is found in all rounds, not just Row Row Row Your Boat

➢ Abstract out the common pattern

canon : Music x double x int → Music

$$\text{canon(m, dur, n)} = \begin{cases} \text{m if n == 1} \\ \text{together(m, canon(delay(m, dur), dur, n-1))} \quad \text{if n > 1} \end{cases}$$

➢ The ability to capture a general pattern like canon() is one of the advantages of music as a first-class object rather than merely a sequence of play() calls

Distinguishing Voices

We want each voice in the canon to be distinguishable

- e.g. an octave higher, or lower, or using a different instrument
- So these **operations** over Music also need to be first-class objects that can be passed to canon()

Extend canon() to apply a function to the repeated melody

canon : Music x int x double x (Music->Music) → Music

e.g. canon(rrryb, 4, 4, transposer(OCTAVE))

produces 4 voices, each one octave higher than the last

transposer: int -> (Music->Music)

transposer(semitones) = lambda m: transpose(m, semitones)

canon() is a higher-order function

- A higher-order function takes a function as an argument or returns a function as its result

Counterpoint

A canon is a special case of a more general pattern

- **Counterpoint** is *n* voices singing related music, not necessarily delayed

  counterpoint : Music x (Music → Music) x int → Music

- Expressed as counterpoint, a canon applies two functions to the music: delay and transform

  canon(m, d, f, n) = counterpoint(m, f ○ delayer(d), n)

  delayer : int → (Music->Music)

  delayer(d) = lambda m: delay(m, d)

Another general pattern

function composition ○ : (U → V) x (T → U) → (T → V)

Repeating

A line of music can also be repeated by the same voice

repeat : Music x int x (Music → Music) → Music

e.g. repeat(rrryb, 2, octaveHigher) = concat(rryb, octaveHigher(rryb))

- Note the similarity to counterpoint():

  counterpoint: m together f(m) together ... together f^{n-1}(m)

  repetition: m concat f(m) concat ... concat f^{n-1}(m)

- And in other domains as well:

  sum: x + f(x) + ... + f^{n-1}(m)

  product: x · f(x) · ... · f^{n-1}(m)

- There's a general pattern here, too; let's capture it

  series : T x (T x T → T) x (T → T) x int → T

  initial value binary op f n

  counterpoint(m, f, n) = series(m, together, f, n)

  repeat(m, f, n) = series(m, concat, f, n)

Repeating Forever

Music that repeats forever is useful for canons

forever: Music → Music

play(forever(m)) plays m repeatedly, forever

duration(forever(m)) = $+\infty$

double actually has a value for this: Double.POSITIVE_INFINITY

Music = Note(duration:double, pitch:Pitch, instr:Instrument)
+ Rest(duration:double)
+ Concat(m1:Music, m2:Music)
+ Together(m1:Music, m2:Music)
+ Forever(m:Music)

why can't we implement forever() using repeat(), or any of the existing Music subtypes?

- Here's the Row Row Row Your Boat round, forever:

  canon (forever(rrryb), 4, 4, octaveHigher)

Accompaniment

accompany: Music x Music $\rightarrow$ Music

repeats second piece until its length matches the first piece

melody line

bass line or drum line,
repeated to match melody's length

accompany(m, b) =

together(m, repeat(b, identity, duration(m)/duration(b))) if duration(m) finite

together(m, forever(b)) if duration(m) infinite

Pachelbel’s Canon

(well, the first part of it, anyway...)

pachelbelBass = notes(“D,2 A,,2 | B,,2 ^F,, | ... | “, CELLO)

pachelbelMelody = notes(“^F’2 E’2 | D’2 ^C’2 | ... | ... | ... | ... | ... |“, VIOLIN)

pachelbelCanon = canon(forever(pachelbelMelody), 3, 16)

pachelbel = concat(pachelbelBass, accompany(pachelbelCanon, pachelbelBass))

Little Languages

We’ve built a new language embedded in Java

- Music data type and its operations constitute a **language** for describing music generation
- Instead of just solving one problem (like playing Row Row Row Your Boat), build a language or toolbox that can solve a range of related problems (e.g. Pachelbel’s canon)
- This approach gives you more flexibility if your original problem turns out to be the wrong one to solve (which is not uncommon in practice!)
- Capture common patterns as reusable abstractions

Formula was an embedded language too

- Formula combined with SAT solver is a powerful tool that solves a wide range of problems

Embedded Languages

Useful languages have three critical elements

	Java	Formula language	Music language
Primitives	`3, false`	`Var, Bool`	`notes, rest`
Means of Combination	`+, *, ==, &&, ||, ...`	`and, or, not`	`together, concat, transpose, delay, …`
Means of Abstraction	variables, methods, classes	naming + methods in Java	naming + functions in Python

- 6.01 calls this PCAP (the Primitive-Combination-Abstraction pattern)

Summary

Review of many concepts we've seen in 6.005

- Abstract data types, recursive data types, interpreter/visitor, composite, immutability

Code as data

- Recursive datatypes, visitors, and functional objects are all ways to express *behavior* as data that can be manipulated and changed programmatically

Higher-order functions

- Operations that take or return functional objects

Building languages to solve problems

- A language has greater flexibility than a mere program, because it can solve large classes of related problems instead of a single problem
- Composite, interpreter, visitor, and higher-order functions are useful for implementing powerful languages
- But in fact any well-designed abstract data type is like a new language

6 (OHPHQWV RI 6RIWZDUH &RQVWUXFWLRQ
Fall 2011