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
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

```java
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, counterpoints
- 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...

Recall our MIDI piano from early lectures

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

© Robert Miller 2009
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)

➢ $T = C_1(...,T) +...+ C_n(...,T) + P_1(...) +...+ P_m(...)$

Music and Formula are composite data types.
Simple Rounds

We need one more operation:

delay : Music × double → Music

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

And now we can express Row Row Row Your Boat


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

round : Music × double × int → Music

round(m, dur, n) = m if n == 1

together(m, round(delay(m, dur), dur, n-1)) if n > 1

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

© Robert Miller 2009
Distinguishing Voices

We want each voice in the round 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 round()
- Fortunately operations implemented as visitors already are objects

**canon() applies a visitor to the repeated melody**

```plaintext
canon : Music x double x Visitor<Music> x int → Music
e.g. canon(rrryb, 4, new TransposeVisitor(OCTAVE), 4)
```

produces 4 voices, each one octave higher than the last

**canon() is a higher-order function**

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

Not all operations are visitors

Let’s generalize the idea of a music transformer function

interface UnaryFunction<T,U> {
    U apply(T t);
}

An instance of UnaryFunction is a **functional object**, representing some function \( f : T \rightarrow U \)

For example:

```java
new UnaryFunction<Music,Music>() {
    public Music apply(Music m) { return delay(m, 4); }
}
```

In general, we might want a `delayer()` method that produces a delay transformer with an arbitrary delay (not just 4 beats):

```java
delay : int \rightarrow \text{UnaryFunction<Music,Music>}
```

let’s write it this way, the abstract type that `UnaryFunction` represents

note that `delayer` is a higher-order function too

`this` anonymous class is essentially a lambda expression producing a functional object

© Robert Miller 2009
Counterpoint

A canon is a special case of a more general pattern

-> **Counterpoint** is \( n \) voices singing related music, not necessarily delayed
counterpoint : Music \( \times (\text{Music} \rightarrow \text{Music}) \times \text{int} \rightarrow \text{Music}

-> Expressed as counterpoint, a canon applies two functions to the music: delay and transform
canon(m, delay, f, n) = counterpoint(m, f \circ \text{delay}(\text{delay}), n)

Another general pattern

\[
\text{function composition } \circ : (U \rightarrow V) \times (T \rightarrow U) \rightarrow (T \rightarrow V)
\]

```java
public static <T,U,V> UnaryFunction<T,V> compose(final UnaryFunction<U,V> g, final UnaryFunction<T,U> f) {
    return new UnaryFunction<T,V>() {
        public V apply(T t) {  return g.apply(f.apply(t));  }
    };
}
```

© Robert Miller 2009
Repeating

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

\[
\text{repeat} : \text{Music} \times (\text{Music} \to \text{Music}) \times \text{int} \to \text{Music}
\]

e.g. \(\text{repeat}(rrryb, \text{octaveHigher}, 2) = \text{concat}(rryb, \text{octaveHigher}(rryb))\)

- Note the similarity to counterpoint():
  - counterpoint: \(m \text{ together } f(m) \text{ together } ... \text{ together } f^{n-1}(m)\)
  - repetition: \(m \text{ concat } f(m) \text{ concat } ... \text{ concat } f^{n-1}(m)\)

- And in other domains as well:
  - sum: \(x + f(x) + ... + f^{n-1}(m)\)
  - product: \(x \cdot f(x) \cdot ... \cdot f^{n-1}(m)\)

- There’s a general pattern here, too; let’s capture it
Binary Functionals

We need first-class representation for binary operations like together, concat, plus, times

```java
interface BinaryFunction<T,U,V> {
    V apply(T t, U u);
}
```

An instance of BinaryFunction represents some \( f : T \times U \rightarrow V \)

- together: Music \times Music \rightarrow Music
- concat: Music \times Music \rightarrow Music

Now we can capture the pattern

```java
series : T \times (T \times T \rightarrow T) \times (T \rightarrow T) \times int \rightarrow T
```

- initial value
- binary op
- \( f \)
- \( n \)

\[
\text{counterpoint}(m, f, n) = \text{series}(m, \text{together}, f, n)
\]

\[
\text{repeat}(m, f, n) = \text{series}(m, \text{concat}, f, n)
\]
Music that repeats forever is useful for canons

\[ \text{forever: Music} \rightarrow \text{Music} \]

\[ \text{play(forever(m))} \text{ plays m repeatedly, forever} \]

\[ \text{duration(forever(m))} = +\infty \]

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, octaveHigher, 4)
Accompaniment

accompany: Music \times\ Music \rightarrow\ Music

repeats second piece until its length matches the first piece

\begin{align*}
\text{melody line} & \quad \text{bass line or drum line,} \\
\text{repeated to match melody’s length} & \end{align*}

\begin{align*}
\text{accompany}(m, b) &= \begin{cases} 
\text{together}(m, \text{repeat}(b, \text{identity}, \text{duration}(m)/\text{duration}(b))) & \text{if duration}(m) \text{ finite} \\
\text{together}(m, \text{forever}(b)) & \text{if duration}(m) \text{ infinite}
\end{cases}
\end{align*}

© Robert Miller 2009
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),
16,
identity,
3)

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

© Robert Miller 2009
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

© Robert Miller 2009
### Embedded Languages

**Useful languages have three critical elements**

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Formula language</th>
<th>Music language</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primitives</strong></td>
<td>3, false</td>
<td>Var, Bool</td>
<td>notes, rest</td>
</tr>
<tr>
<td><strong>Means of Combination</strong></td>
<td>+, *, ==, &amp;&amp;,</td>
<td></td>
<td>, ...</td>
</tr>
<tr>
<td><strong>Means of Abstraction</strong></td>
<td>variables, methods, classes</td>
<td>Java mechanisms</td>
<td>functional objects + Java mechanisms</td>
</tr>
</tbody>
</table>

> 6.01 calls this PCAP (the primitive-combination-abstraction pattern)
Summary

**Composite pattern**
- Composite data types allow a group of objects to be treated the same as a single object

**Functionals**
- UnaryFunction and BinaryFunction represent functions as Java objects
- So do Runnable and Visitor, in fact

**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
- Interpreter pattern, visitor pattern, and higher-order functions are useful for implementing powerful languages
- But in fact any well-designed abstract data type is like a new language

© Robert Miller 2009