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

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 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 Your Boat": one voice starts, other voices enter after a delay
  
  Row row your boat, gently down the stream, merrily merrily ...
  
  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:
  
  ```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

---

A Few of Music's Operations

```java
notes : String × Instrument → Music
  
  requires string is a subset of abc music notation
  
  e.g. notes("E D C D | E E E G | 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 × 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
```

---

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?

---

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(m:Music, m2:Music)
- Together(m: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)
- $$T = C_1(...) + ... + C_n(...) + P_1(...) + ... + P_m(...)$$

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**
- 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 x double x 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

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**
- 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
  ```java
  interface UnaryFunction<T,U> {
    U apply(T t);
  }
  ```
- An instance of UnaryFunction is a **functional object**, representing some function \( f : T \to U \)

**Enrich the idea of a music transformation**

- For example:
  ```java
  new UnaryFunction<Music,Music>() {
    public Music apply(Music m) { return delay(m, 4); }
  }
  ```
- In general, we might want a delay() method that produces a delay transformer with an arbitrary delay (not just 4 beats):
  ```java
delay : int \to UnaryFunction<Music,Music>
```

**Let's write it this way, the abstract type that UnaryFunction represents**

**Counterpoint**

**A canon is a special case of a more general pattern**

- **Counterpoint** is \( n \) voices singing related music, not necessarily delayed
delay time: \( f : T \to U \)
- Expressed as counterpoint, a canon applies two functions to the music:
delay and transform
canon(m, delay, f, n) = counterpoint(m, f o delay, delay, n)

**Another general pattern**

- **function composition** \( \circ : (U \to V) \times (T \to U) \to (T \to V) \)

**Binary Functionals**

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

- **Interface BinaryFunction<T,U,V>**
  ```java
  V apply(T t, U u);
  ```
- An instance of BinaryFunction represents some \( f : T \times U \to V \)
together:
  ```java
  public V apply(T t) { return g.apply(f.apply(c)); }
  ```

**Now we can capture the pattern**

- **series** : \( T \times (T \times T) \times (T \to T) \times int \to T \)
  ```java
  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)) = +∞

\[ \text{Music} = \text{Note}(\text{duration}: \text{double}, \ \text{pitch}: \text{Pitch}, \ \text{instr}: \text{Instrument}) \]
\[ + \text{Rest}(\text{duration}: \text{double}) \]
\[ + \text{Concat}(m_1: \text{Music}, m_2: \text{Music}) \]
\[ + \text{Together}(m_1: \text{Music}, m_2: \text{Music}) \]
\[ + \text{Forever}(m: \text{Music}) \]

Here’s the Row Row Row Your Boat round, forever:
canon (forever(rrry), 4, octaveHigher, 4)

© Robert Miller 2008

Accompaniment

accompany: Music x Music → Music
  repeats second piece until its length matches the first piece

\[ \text{accompany}(m, b) = \begin{cases} \\text{concat}(\text{m}, \text{b}) & \text{if } \text{duration}(m) \text{ finite} \\
\text{together}(\text{m}, \text{forever}(\text{b})) & \text{if } \text{duration}(m) \text{ infinite} \\
\end{cases} \]

© Robert Miller 2008

Pachelbel’s Canon

(well, the first part of it, anyway…)

pachelbelBass = notes("D,2 A,2 | B,2 ^E, | ... |", 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 2008

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 2008
**Embedded Languages**

**Useful languages have three critical elements**

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