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

```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

© Robert Miller 2011
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:

  ```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?
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

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
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

© Robert Miller 2011
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

\[
\text{Music} = \text{Note}(\text{duration:double, pitch:Pitch, instr:Instrument}) + \text{Rest}(\text{duration:double}) + \text{Concat}(m1:Music, m2:Music) + \text{Together}(m1:Music, m2:Music)
\]

Here’s where our decision to make Concat() tree-like becomes very useful

- Suppose we instead had:
  \[
  \text{Concat} = \text{List<Note + Rest>}
  \]
  \[
  \text{Together} = \text{List<Concat>}
  \]
- 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)

\[
T = C_1(\ldots, T) + \ldots + C_n(\ldots, T) + P_1(\ldots) + \ldots + P_m(\ldots)
\]

Music and Formula are composite data types.
Simple Rounds

We need one more operation:

\[
\text{delay} : \text{Music} \times \text{double} \rightarrow \text{Music}
\]

\[
\text{delay}(m, \text{dur}) = \text{concat} (\text{rest} (\text{dur}), m)
\]

And now we can express Row Row Row Your Boat

\[
\text{rrryb} = \text{notes} ("C C C^{3/4} D^{3/4} E | E^{3/4} D^{3/4} E^{3/4} F^{3/4} G2 | \ldots", \text{PIANO})
\]

\[
\text{together}(\text{rrryb}, \text{delay}(\text{rrryb}, 4))
\]

- 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

\[
\text{canon} : \text{Music} \times \text{double} \times \text{int} \rightarrow \text{Music}
\]

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

> The ability to capture a general pattern like \text{canon}() is one of the advantages of music as a first-class object rather than merely a sequence of \text{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

\[
\text{canon} : \text{Music} \times \text{int} \times \text{double} \times (\text{Music} \rightarrow \text{Music}) \rightarrow \text{Music}
\]

- e.g. \(\text{canon}(\text{rrryb}, 4, 4, \text{transposer}(\text{OCTAVE}))\)
  produces 4 voices, each one octave higher than the last

transposer: \text{int} \rightarrow (\text{Music} \rightarrow \text{Music})

\[
\text{transposer}(\text{semitones}) = \lambda m: \text{transpose}(m, \text{semitones})
\]

\text{canon}() \text{ is a higher-order function}

- A higher-order function takes a function as an argument or returns a function as its result
A canon is a special case of a more general pattern

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

\[
\text{counterpoint} : \text{Music} \times (\text{Music} \rightarrow \text{Music}) \times \text{int} \rightarrow \text{Music}
\]

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

\[
\text{canon}(m, d, f, n) = \text{counterpoint}(m, f \circ \text{delay}(d), n)
\]

\[
\text{delay} : \text{int} \rightarrow (\text{Music} \rightarrow \text{Music})
\]

\[
\text{delay}(d) = \lambda m: \text{delay}(m, d)
\]

Another general pattern

- function composition \( \circ : (U \rightarrow V) \times (T \rightarrow U) \rightarrow (T \rightarrow V) \)
A line of music can also be repeated by the same voice

repeat : Music x int x (Music \rightarrow Music) \rightarrow 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 \cdot f(x) \cdot ... \cdot f^{n-1}(m)

- There’s a general pattern here, too; let’s capture it
  series : T x (T x T \rightarrow T) x (T \rightarrow T) x int \rightarrow 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)) = +∞

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 → 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) =

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

<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>naming + methods in Java</td>
<td>naming + functions in Python</td>
</tr>
</tbody>
</table>

> 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
6.005 Elements of Software Construction
Fall 2011

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.