Little Languages

Rob Miller
Fall 2007
Review: Code as Data

Interpreter pattern
- A data type representing the syntax of a language
  - e.g. Expr is the language of algebraic expressions over rationals
- Another way to say it is that an Expr value like
  
  ```java
  new Plus(new Var("x"), new Var("y"))
  ```
  has the same meaning as the Java code
  
  ```java
  x.plus(y)
  ```
  but an Expr value is a first-class object

Visitor pattern
- A visitor represents a function over a recursive or option type
  - e.g. new DerivVisitor(new Var("x")) represents \( \frac{d}{dx} : \text{Expr} \rightarrow \text{Expr} \)
- So a visitor object also represents code in a first-class object

Today: higher-order functions
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 (Hofstadter’s *Gödel Escher Bach* is strongly recommended)

Recall our music machine from early lectures

- The core machine had a **play** operation for sounding a note
- So a song could be represented by Java code that makes a sequence of **play()** calls
  
  ```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

© Robert Miller 2007
Music Data Type

Let’s start by representing simple tunes

Music = Note(double duration x Pitch x Instrument)
∪ Rest(double duration)
∪ Concat(Music x 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 is the rep invariant?
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

duration : Music → double

returns total duration of music in beats

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

transpose : Music x int semitones → 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

© Robert Miller 2007
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(Music × int)
- Defers the actual evaluation of the function
- Enables more sharing between values
- Adding a new subclass requires changing all visitors

© Robert Miller 2007
Duality Between Method and Visitor

**Operation as method**
- 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 as visitor**
- 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(double duration x Pitch x Instrument)
   ∪ Rest(double duration)
   ∪ Concat(Music x Music)
   ∪ Together(Music x 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 = C₁(...T) ∪ ... ∪ Cₙ(...T) ∪ P₁(...) ∪ ... ∪ Pₘ(...)
Simple Rounds

We need one more operation:

\[
\text{delay} : \text{Music} \times \text{double} \rightarrow \text{Music}
\]
\[
\text{delay}(m, \text{dur}) = \text{concat}(\text{rest}(	ext{dur}), m)
\]

And now we can express Row Row Row Your Boat

\[
\text{rrryb} = \text{notes}(\text{“C C C3/4 D/4 E \mid E3/4 D/4 E3/4 F/4 G2 \mid ...”}, \text{PIANO})
\]
\[
\text{together}(\text{rrryb}, \text{delay}(\text{rrryb}, 4))
\]

- 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

\[
\text{round} : \text{Music} \times \text{double} \times \text{int} \rightarrow \text{Music}
\]
\[
\text{round}(m, \text{dur}, n) = \begin{cases} 
m & \text{if } n == 1 \\
\text{together}(m, \text{round}(\text{delay}(m, \text{dur}), \text{dur}, n-1)) & \text{if } n > 1
\end{cases}
\]

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

© Robert Miller 2007
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 $f : T \rightarrow U$

- 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
delayer : int \rightarrow \text{UnaryFunction<Music,Music>}
```

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

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

© Robert Miller 2007
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, delay, f, n) = counterpoint(m, f ○ delayer(delay), n)

Another general pattern
function composition ○ : (U → V) x (T → U) → (T → V)

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)); }
    };
}
Repeating

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

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

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

- 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

© Robert Miller 2007
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

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

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

counterpoint\( (m, f, n) = \) series\( (m, \text{together}, f, n) \)
repeat\( (m, f, n) = \) series\( (m, \text{concat}, f, n) \)
Repeating Forever

Music that repeats forever is useful for canons

\text{forever: Music \rightarrow Music}

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

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

\text{double \ actually \ has \ a \ value \ for \ this:}
\text{Double.POSITIVE_INFINITY}

\text{Music = Note(double duration \ x \ Pitch \ x \ Instrument)}
\cup \text{Rest(double duration)}
\cup \text{Concat(Music \ x \ Music)}
\cup \text{Together(Music \ x \ Music)}
\cup \text{Forever(Music)}

\text{Here’s the Row Row Row Your Boat round, forever:}
\text{canon \ (forever(rrryb), 4, octaveHigher, 4)}

© Robert Miller 2007
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) =
together(m, repeat(b, identity, duration(m)/duration(b))) if duration(m) finite
together(m, forever(b)) if duration(m) infinite

© Robert Miller 2007
Pachelbel’s Canon

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

pachelbelBass = notes("D,2 A,,2 | B,,2 ^F,, | ... |", CONTRABASS)

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

pachelbelCanon = canon(FOREVER(pachelbelMelody),
  16,
  identity,
  3)

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

Expr was an embedded language too, though not as powerful
# Embedded Languages

Languages have three critical elements

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Expr language</th>
<th>Music language</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primitives</strong></td>
<td>3, false</td>
<td>const</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>var + Java mechanisms</td>
<td>functional objects + Java mechanisms</td>
</tr>
</tbody>
</table>
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 2007