Little Languages

Rob Miller and Saman Amarasinghe
Spring 2008
Code and Data

Before 1945

- Code and data were separate

1945: John von Neumann

- EDVAC machine

Today: we will treat code as data

- higher-order functions
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 \(d/dx : \text{Expr} \rightarrow \text{Expr}\)
- So a visitor object also represents code in a first-class object
Today’s Problem: Music

Interesting music tends to have a lot of repetition
- Let’s look at rounds and canons
- 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 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
  - 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 \textit{duration} \times \textit{Pitch} \times \textit{Instrument})
\quad \cup \text{Rest(double duration)}
\quad \cup \text{Concat(Music \times Music)}

- duration is measured in \textit{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 \textit{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

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 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

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

© 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 x int)
- Defers the actual evaluation of the function
- Enables more sharing between values
- Adding a new subclass requires changing all visitors
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)
**Visitor Revisited**

```java
interface Music {
    <T> T accept(Visitor<T> v);
}

class Note implements Music {
    public <T> T accept(Visitor<T> v) { return v.onNote(this); }
}

public class Concat implements Music {
    private final Music first;
    private final Music second;
    public <T> T accept(Visitor<T> v) { return v.onConcat(this); }
}

interface Visitor<T> {
    T onNote(Note m);
    T onConcat(Concat m);
}

class MusicPlayer {
    private final Music m;
    public MusicPlayer(Music m) { this.m = m; }
    public void play() throws MidiUnavailableException {
        m.accept(new Visitor<Void>() {
            public Void onConcat(Concat m) {
                m.first().accept(this);
                m.second().accept(this);
                return null;
            }
            public Void onNote(Note m) {
                midi.play(...)
                return null;
            }
        });
    }
}

MusicPlayer player = new MusicPlayer(myMusic);
Player.play();
```

© Robert Miller 2007
public class RowYourBoat {

    public static void main(String[] args) throws MidiUnavailableException {
        Music rowYourBoat =
            notes("C C C3/4 D/4 E |
            +"E3/4 D/4 E3/4 F/4 G2 |
            +"C'/3 C'/3 C'/3 G/3 G/3 G/3 E/3 E/3 E/3 C/3 C/3 C/3 |
            +"G3/4 F/4 E3/4 D/4 C2",
            PIANO);
        new MusicPlayer(rowYourBoat).play();
    }
}

DEMO
Multiple Voices

For a round, the parts need to be sung simultaneously
Music = Note(double $duration \times$ Pitch $\times$ Instrument)
  $\cup$ Rest(double $duration$)
  $\cup$ Concat(Music $\times$ Music)
  $\cup$ Together(Music $\times$ Music)

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

- Suppose we instead had:
  Concat = List<Note $\cup$ 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)
- $T = C_1(...T) \cup ... \cup C_n(...T) \cup P_1(...) \cup ... \cup P_m(\ldots)$

Music and Expr are composite data types.
So is Shape (from Proj 3)
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

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

- 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

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

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

© Robert Miller 2007
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 \( f : T \rightarrow U \)

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\) UnaryFunction<Music, Music>
```

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

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

```
let's write it this way, the abstract type that UnaryFunction represents
```
public interface UnaryFunction<T,U> {
    U apply(T t);
}

public class IdentityVisitor implements UnaryFunction<Music,Music>, Visitor<Music> {
    public Music apply(Music m) {
        return m.accept(this);
    }
    public Music onConcat(Concat m) {
        Music m1 = m.first();
        Music m2 = m.second();
        Music newM1 = m1.accept(this);
        Music newM2 = m2.accept(this);
        return new Concat(newM1, newM2);
    }
    public Music onNote(Note m) {
        return m;
    }
}

public class MusicLanguage {
    public static UnaryFunction<Music,Music> transposer(final int semitonesUp) {
        return new IdentityVisitor() {
            @Override
            public Music onNote(Note m) {
                return new Note(m.duration(), m.pitch().transpose(semitonesUp), m.instrument());
            }
        };
    }
    public static Music transpose(Music m, int semitonesUp) {
        return transposer(semitonesUp).apply(m);
    }
    MusicLanguage.transpose(MyMusic, 2);
}
public class RowYourBoatSimple {

    public static void main(String[] args) throws MidiUnavailableException {
        Music rowYourBoat =
            notes("C C C3/4 D/4 E |
                 +"E3/4 D/4 E3/4 F/4 G2 |
                 +"C'/3 C'/3 C'/3 G/3 G/3 G/3 E/3 E/3 E/3 C/3 C/3 C/3 |
                 +"G3/4 F/4 E3/4 D/4 C2",
                 PIANO);
        Music rowTwice =
            concat(rowYourBoat, transpose(rowYourBoat, Pitch.OCTAVE));
        new MusicPlayer(rowTwice).play();
    }
}

DEMO
Counterpoint

A canon is a special case of a more general pattern

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

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

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

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

Another general pattern

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

```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 2007
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
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 \( f : T \times U \to V \)
- `together`: Music x Music \( \to \) Music
- `concat`: Music x Music \( \to \) Music

Now we can capture the pattern

```java
series : T x (T x T \to T) x (T \to T) x int \to T
```

`counterpoint(m, f, n) = series(m, together, f, n)`
`repeat(m, f, n) = series(m, concat, f, n)`

© Robert Miller 2007
public interface UnaryFunction<T,U> {
    U apply(T t);
}

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

class Functions {
    static <T> UnaryFunction<T,T> identity() {
        return new UnaryFunction<T,T>() {
            public T apply(T t) { return t; }
        };
    }
    static <T,U,V> UnaryFunction<T,V> compose(UnaryFunction<U,V> g, UnaryFunction<T,U> f) {
        return new UnaryFunction<T,V>() {
            public V apply(T t) { return g.apply(f.apply(t)); }
        };
    }
    static <T> T series(T t, BinaryFunction<T,T,T> binop, UnaryFunction<T,T> f, int n) {
        if (n == 1) return t;
        else return binop.apply(t, series(f.apply(t), binop, f, n-1));
    }
}

class MusicLanguage {
    static UnaryFunction<Music,Music> IDENTITY = functions.Functions.<Music>identity();
    static BinaryFunction<Music,Music,Music> TOGETHER = new BinaryFunction<Music,Music,Music>() {
        Music apply(Music m1, Music m2) { return new Together(m1, m2); }
    };
    static Music counterpoint(Music m, UnaryFunction<Music,Music> f, int n) {
        return series(m, TOGETHER, f, n);
    }
    static UnaryFunction<Music,Music> delayer(final double delay) {
        new UnaryFunction<Music,Music>() {
            Music apply(Music m) { return concat(new Rest(delay), m); }
        };
    }
    static Music canon(Music m, double delay, UnaryFunction<Music,Music> f, int n) {
        return counterpoint(m, compose(f, delayer(delay)), n);
    }
}

MusicLanguage.cannon(Melody, 16, IDENTITY, 2);
public interface UnaryFunction<T, U> {
    U apply(T t);
}

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

class Functions {
    static <T> UnaryFunction<T, T> identity() {
        return new UnaryFunction<T, T>() {
            public T apply(T t) { return t; }
        };
    }
    static <T, U, V> UnaryFunction<T, V> compose(UnaryFunction<U, V> g, UnaryFunction<T, U> f) {
        return new UnaryFunction<T, V>() {
            public V apply(T t) { return g.apply(f.apply(t)); }
        };
    }
    static <T> T series(T t, BinaryFunction<T, T, T> binop, UnaryFunction<T, T> f, int n) {
        if (n == 1) return t;
        else return binop.apply(t, series(f.apply(t), binop, f, n-1));
    }
}

class MusicLanguage {
    static UnaryFunction<Music, Music> IDENTITY = Functions.<Music>identity();
    static BinaryFunction<Music, Music, Music> TOGETHER = new BinaryFunction<Music, Music, Music>() {
        Music apply(Music m1, Music m2) { return new Together(m1, m2); }
    };
    static Music counterpoint(Music m, UnaryFunction<Music, Music> f, int n) {
        return series(m, TOGETHER, f, n);
    }
    static UnaryFunction<Music, Music> delayer(final double delay) {
        new UnaryFunction<Music, Music>() {
            Music apply(Music m) { return concat(new Rest(delay), m); }
        };
    }
    static Music canon(Music m, double delay, UnaryFunction<Music, Music> f, int n) {
        return counterpoint(m, compose(f, delayer(delay)), n);
    }
}

MusicLanguage.cannon(Melody, 16, IDENTITY, 2);

© Robert Miller 2007
public class RowYourBoatOctaves {

    public static void main(String[] args) throws MidiUnavailableException {
        Music rowYourBoat =
            notes("C C C3/4 D/4 E |
                +"E3/4 D/4 E3/4 F/4 G2 |
                +"C'/3 C'/3 C'/3 G/3 G/3 G/3 E/3 E/3 E/3 C/3 C/3 C/3 |
                +"G3/4 F/4 E3/4 D/4 C2",
                PIANO);
        Music rowRound =
            canon(rowYourBoat, 4 /*beats of delay*/, transposer(Pitch.OCTAVE), 4 /*voices*/);
        new MusicPlayer(rowRound).play();
    }
}

DEMO
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(double duration x Pitch x Instrument)
∪ Rest(double duration)
∪ Concat(Music x Music)
∪ Together(Music x Music)
∪ Forever(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)

© Robert Miller 2007
Accompaniment

accompany: Music $\times$ Music $\rightarrow$ Music

repeats second piece until its length matches the first piece

\[
\text{melody line}
\]

\[
\text{bass line or drum line, repeated to match melody’s length}
\]

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

© Robert Miller 2007
public class PachelbelCanon {
    public static void main(String[] args) throws MidiUnavailableException {
        Music pachelbelBass =
            notes("D,2 A,,2 | B,,2 ^F,,2 | G,,2 D,,2 | G,,2 A,,2",
                  CONTRABASS);
        Music pachelbelMelody =
                  VIOLIN);
        Music pachelbelCanon =
            canon(forever(pachelbelMelody),
                16, // each new voice enters after four 4-beat measures
                IDENTITY,
                3); // 3 voices
        Music pachelbel =
            concat(pachelbelBass, // bass line starts by itself
                accompany(pachelbelCanon, // then joined by melody
                    pachelbelBass));
        new MusicPlayer(pachelbel).play();
    }
}
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><em>3, false</em></td>
<td>const</td>
<td><em>notes, rest</em></td>
</tr>
<tr>
<td><strong>Means of</strong></td>
<td>+, *, ==, &amp;&amp;,</td>
<td>plus, minus,</td>
<td>together,</td>
</tr>
<tr>
<td><strong>Combination</strong></td>
<td></td>
<td></td>
<td>, ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td>transpose,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>delay, ...</td>
</tr>
<tr>
<td><strong>Means of</strong></td>
<td></td>
<td></td>
<td>functional</td>
</tr>
<tr>
<td><strong>Abstraction</strong></td>
<td></td>
<td></td>
<td>objects +</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Java mechanisms</td>
</tr>
</tbody>
</table>

© Robert Miller 2007
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
Higher Order Functions in the SAT Solver

Generating all environments

- this is not so easy
  - don’t want a function to return entire set as one object (too big)
  - hard (in Java) to return one at a time

one solution

- write a function that takes a predicate and checks it against each env
- then we can pass this function a predicate that evaluates a given formula
- express this with a polymorphic function declarations

  find : Set<Literal>, (Env -> Bool) -> (Env + None)

- find is a higher-order function that takes a function as an arg
in Java, implement function as an object

  Pred.match: Env -> Bool

  find : Set<Literal>, Pred -> (Env + None)

Go back and take a look at the code

© Robert Miller 2007
Lecture Exercises

Reduction is a common operation on a list of elements. For example, finding the max or sum of a list of integers is a common operation.

Create a static function `reduce` in the class `Functions` that returns a `UnaryFunction`. This `UnaryFunction` should take a list as the argument and return the reduced value.

What is the type signature of `reduce`?

Show how to find the min and sum of a list of integers using `reduce`.

Given a list of strings, use `reduce` to count the number of strings that begins with “a”.

© Robert Miller 2007