SMURF

Serial MUsic Represented as Functions

Richard Townsend, Lianne Lairmore, Lindsay Neubauer, Van Bui, Kuangya Zhai

{rt2515, lel2143, lan2135, vb2363, kz2219}@columbia.edu

September 25, 2013

SMURF is a functional language that will allow a composer to create serialist music based on twelve-tone composition. In general, serialism is a musical composition technique where a set of values, chosen through some methodical process, generates a sequence of musical elements. Our languge will be based on the functional syntax and semantics set forth by Haskell, and we will compile our programs into C. The C program will then use OpenGL to generate images of sheet music corresponding to the user's initial program in SMURF.

1   Background: What is Serialism?

In general, serialism is a musical composition technique where a set of values, chosen through some methodical process, generates a sequence of musical elements. Its origins are often attributed to Arnold Schoenberg's twelve-tone technique, which he began to use in the 1920s. In this system, each note in the chromatic scale is assigned an integer value, giving us a set of twelve "pitch classes" (Figure 1 [1]). A composer utilizing this method then takes each of these integers, and orders them into a *twelve tone row*, where each number appears exactly once. We refer to this row as the *primeform* of a piece, and conventionally refer to it as P_0.

The composer can then generate other rows that are derived from P_0 through three types of transformations: transposition, inversion, and retrograde. In each of these transformations, we always use mod 12 arithmetic to preserve the numbering system of our pitch classes. Transposing a row consists of taking each pitch class in the row and adding the same number to each. If we transpose P_0 by four semitones, we add four mod twelve to each pitch class in P_0 and end up with a new row called P_4. In general, P_x is a transposition of P_0 by x semitones. To invert a row, we "flip" each interval between two pitch classes in that row.
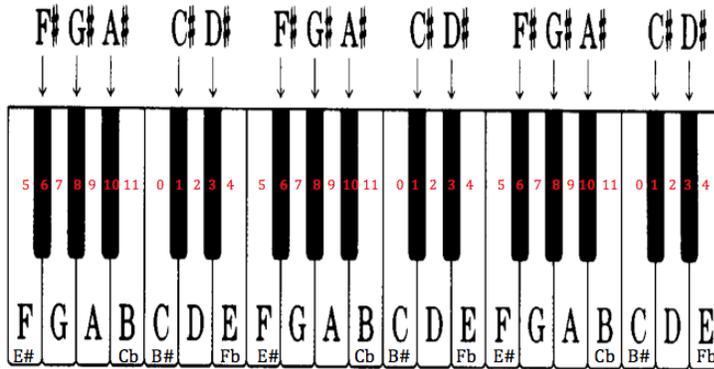
Figure 1: pitch classes

	I_0	I_{11}	I_7	I_8	I_3	I_1	I_2	I_{10}	I_5	I_5	I_4	I_9
P_0	0	11	7	8	3	1	2	10	6	5	4	9
P_1	1	0	8	9	4	2	3	11	7	6	5	10
P_5	5	4	0	1	8	6	7	3	11	10	9	2
P_4	4	3	11	0	7	5	6	2	10	9	8	1
P_9	9	8	4	5	0	10	11	7	3	2	1	6
P_{11}	11	10	6	7	2	0	1	9	5	4	3	8
P_{10}	10	9	5	6	1	11	0	8	4	3	2	7
P_2	2	1	9	10	5	3	4	0	8	7	6	11
P_6	6	5	1	2	9	7	8	4	0	11	10	3
P_7	7	6	2	3	10	8	9	5	1	0	11	4
P_8	8	7	3	4	11	9	10	6	2	1	0	5
P_3	3	2	10	11	6	4	5	1	9	8	7	0

Figure 2: twelve tone matrix

An interval is best thought of as the smallest "distance" between two pitch classes, using the proximity on the piano of the two pitch classes as the distance metric (refer to Figure 1 for reference). For example, pitch classes 0 and 11 have a distance of 1 from each other, since you can reach pitch class 0 from 11 by adding 1 to 11 (remember the mod 12 arithmetic) or reach 11 from 0 by subtracting 1 from 0. Thus an interval of +1 exists from 11 to 0, and an interval of -1 exists from 0 to 11. As a further example, if P_0 starts with pitch classes 0-11-7, then we have an interval of -1 between the first two pitches and -4 between the second two. Flipping an interval between two pitch classes is identical to negating its sign. Thus, in the inverse of P_0 (called I_0), the first interval would be +1 and the second would be +4, giving us 0-1-5 as our first three pitch classes. The subscript of I_x refers both to the number of transpositions required to arrive at I_x from I_0, and to the prime row P_x that would need to be inverted to generate I_x. The final row operation is a retrograde transformation, which merely consists of reading a row backwards. That is, R_x is generated by reading the pitch classes of P_x in their opposite order. One can also have a retrograde inversion; RI_x is generated by reading the pitch classes of I_x backwards.

Once a composer chooses a P_0, the three transformations outlined above can be applied to varying degrees to generate a *twelve tone matrix*, which will contain each P row as a row in the matrix and each I row as a column. Furthermore, all of the R and RI rows are found by reading the rows in the matrix from right to left or the columns from bottom to top, respectively. An example of a twelve tone matrix from one of Shoenberg's pieces can be found in Figure 2 [2]. Finally, using the twelve tone matrix as a guide, the composer picks various rows and columns to serve as melodic and harmonic elements in their composition, resulting in a piece of serial music.

2 Motivation

Twelve tone serialism is a mathematically intensive method of creating music which involves mapping notes to numbers. It is natural to work with twelve tone rows using a programming language since the method treats notes like numbers that can be added and subtracted. Our language will make twelve tone composition easier by using data types and functions that cater to the needs of a serial composer. By simplifying the method of inverting and transposing rows, composers can focus more on how to exploit new ways to make serial music and worry less about creating matrices.

We chose to implement a functional language because of the clear and succinct programs that functional languages produce. In addition, the well known ability of functional languages to work on lists is advantagous for twelve tone serialism, because most serial arithmetic operations use rows and columns from the twelve tone matrix as operands. As a group we are also interested on how a functional language compiler works.

Instead of compiling our language into byte code where notes are interpreted and played, we think it is more interesting to create a language that, once compiled, generates a score. Scores are made up of lines and dots, which are easy to create in a graphics library like OpenGL. We decided to compile our language into C since it is a significantly less abstract language that also has access to OpenGL library calls. Compiling into C (as opposed to compiling to a language that generates an audio file) is advantageous in that programs in our language can create functions that can be used in other programs to do the same transformation of rows. That way our language is not limited to functions defined in our libraries or what is in the current file.

Overall we hope to use the simplicity of a functional language to help composers write complex, new, and interesting music based on twelve tone serialism. These new compositions can then be printed and used by musicians for performances. This simplifies the composer's task of converting computer generated music to a form that musicians can easily read.

3 SMURF Pipeline

The proposed pipeline for composing music with SMURF is demonstrated in Figure 3. The composer creates serial music by first writing a SMURF program, which will be compiled by our SMURF compiler to generate a C program with OpenGL. Then the gcc compiler and assembler are used to generate a music score with which musicians can play.

4 Syntax

SMURF is a functional programming language loosely modeled off of Haskell. Functions are the main building blocks of SMURF. It has immutable memory, no global variables and no I/O. A typical program consists of constant definitions and functions, each separated by new line characters.
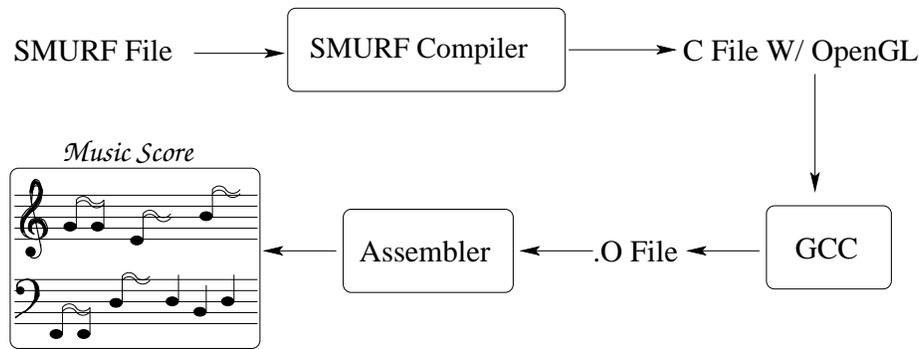
Figure 3: SMURF pipeline for composing serialist music

4.1 Types

4.1.1 Atomic Types

- Integer: int

- Boolean: bool

4.1.2 Structured Types

Structured types hold groups of elements

- Tuples: (a, ..., n), where items a - n are elements in the tuple

  - Elements can have different types

- Lists: [a, ..., n], where items a - n are elements in the list

  - Elements must have same type

4.1.3 Beat

A Beat represents a length of musical time. It has a Time tag and integer type.

- Must have the string "Time" followed by an integer that is a power of 2 and < 32 in declaration

  - whole note: Time 1
  - half note: Time 2
  - quarter note: Time 4
  - eighth note: Time 8
  - sixteenth note: Time 16
  - thirty-second note: Time 32

4

- Uses + operator to combine Time but only adds two operands that contain the same integer; recursively checks for Time operands that contain the same integers until only unequal Time integers are left

    - Time 4 + Time 16 + Time 16 + Time 16 + Time 16

    - Time 4 + Time 8 + Time 16 + Time 16

    - Time 4 + Time 8 + Time 8

    - Time 4 + Time 4 = Time 2 (quarter note + quarter note = half note)

4.1.4 Note

A Note is a tuple of three integers and is declared as (pc: int, beat: Beat, register: int)

- pc (pitch class): represented by integers 0-11

    - A Note with pc = -1 represents a rest. In this special case, the register for the Note only matters in relation to whether the rest lies on the treble or bass clef (i.e. whether the register is positive or negative)

- beat: has type Beat

- register: represented by integers -2 to 1

    - Treble Clef: notes middle C and higher represented by 0 and 1

        * middle C to the first B above middle C: 0
        * first C above middle C to next highest B: 1

    - Bass Clef: notes lower than middle C represented by -1 and -2

        * B directly below middle C to first C below middle C: -1
        * next lowest B to next lowest C: -2

4.1.5 Chord

A Chord is a list of notes and is declared as [Note]. The compiler will check that all notes in the list have the same beat count.

4.1.6 Measure

A Measure is a list of chords and is declared as [Chord]. The compiler will check that every measure is composed of 4 beats, where a quarter note constitutes one beat (we are assuming that musical scores will be in $\frac{4}{4}$ time). A list of measures is passed to the keyword genScore, which generates and outputs an image of the musical score.

4.2   Functions

Functions are the main building blocks of the language.

- Function declarations must declare type (can declare general type)

- Function declarations must be on own line

- All functions are first order

- No explicit return statement

- Pattern matching, guards, and if-then-else clauses used

  - Each pattern matching pattern must be on own line
  - Guards and if-then-else clauses do not have newline restrictions

4.3   Variables

Variables can be declared globally or locally. A global variable is declared with an assignment operator and acts as a constant e.g. x = 4. A local variable is declared with a let ... in ... expression. Multiple bindings may be declared in the let expression, and the bindings are mutually recursive. The bindings only exist in the expression following the let expression. In the example that follows, x is bound to 3 and y is bound to 4, and these bindings only exist in the expression following the let ... in ... expression.

```
let x = 3
    y = 4
in x + y
```

4.4   Operators

4.4.1   Comment Operators

SMURF allows nested, multiline comments in addition to single line comments.

Operator	Description	Example
/* */	Multiline comments, nesting allowed	/* This /* is all */ commented */
//	Single-line comment	// This is a comment

4.4.2   Arithmetic Operators

SMURF allows assignment and addition, subtraction, and modulus on expressions that evaluate to integers. These operators are all infix. The modulus operator ignores negatives e.g.

```
13 % 12 is equal to -13 % 12
```

.

Operator	Description	Example
=	Assignment operator	a = 4
+	Integer arithmetic: plus	a + 2
−	Integer arithmetic: minus	5 − a
%	Integer arithmetic: modulus, ignores negatives	14 mod 12

4.4.3 Comparison Operators

SMURF allows comparison operations between expressions that evaluate to integers.

Operator	Description	Example
<	Less than	if a < 5 then True else False
>	Greater than	if a > 5 then True else False
<=	Less than or equal to	if a <= 5 then True else False
>=	Greater than or equal to	if a >= 5 then True else False

4.4.4 Boolean Operators

SMURF allows logical negation, conjunction, and disjunction, in addition to structural comparison and boolean notation for use with guards.

Operator	Description	Example
==	Structural comparison	if a == 5 then a = True else a = False
not	Logical negation	if not a == 5 then True else False
&&	Logical conjunction	if b && c then True else False
||	Logical disjunction	if b || c then True else False

4.4.5 List Operators

SMURF allows concatenation and construction of lists.

Operator	Description	Example
++	Concatenation: concat	[1,2,3] ++ [4,5,6] (result is [1,2,3,4,5,6])
:	Construction: cons	1 : [2,3,4] (result is [1,2,3,4])

4.4.6 Function Operators

SMURF allows type, argument, and function return type specification in addition to concatenation and construction operations

Operator	Description	Example
::	Type specification	returnIntFunc :: Int
–>	Argument and function return type specification	isPositiveNum :: Int –> Bool
|	Boolean operator used with guards	isLowOrSeven num :: [Int] –> Bool | (num <5 || num == 7) = True | otherwise = False

4.5 Keywords

Keywords	
let	Specify variables and functions
in	Allow local variable binding in expression
if, then, else	Specify conditional expression, else compulsory
True, False	Specify boolean logic
otherwise	Specify conditional expression used with guards
genScore	Generate musical score given list of measures as argument

4.6 Library Functions

There are currently six library functions that can be used in SMURF: trans, inver, rev, head, tail, fillMeasure.

4.6.1 trans

Given an integer and a list (row), return a list that adds the integer to each element in the original list and computes mod 12 on the result of each addition. If not given an integer and then a list return an error.

4.6.2 inver

Given a list (row), return a list that has the original list elements inverted. If not given a list return error.

4.6.3 rev

Given a list (row), return a list that has the original list elements reversed. If not given a list return error.

4.6.4 head

Given a list of elements, return the first element in the list. If not given a list return error.

4.6.5 tail

Given a list of elements, return the list with the first element removed. If not given a list return error.

4.6.6 fillMeasure

Given a list of chords, return a boolean whether the beats of each chord in the list add up to a measure.

- Time 1 + Time 32 evaluates to False (cannot have more than a whole note in a measure)

- Time 4 + Time 16 evaluates to False (quarter note and sixteenth note do not contain enough beats for a measure)

- Time 4 + Time 4 + Time 4 + Time 4 evaluates to True (4 quarter notes in one measure)

5 Example SMURF Program

The SMURF program in Figure 4 shows how to use various language features in SMURF including function definitions, keywords, types, library functions, and operators. The code uses the `trans` and `inver` library functions to generate two tone rows that are then used to create chords and a list of measures that make up a music score.

The first step in composing serial music starts with defining the pitch classes for the prime row, which is defined in the example code in line 33. Using the prime row as a base, the program creates a score using the tone rows generated by transposing the prime row by three semitones (line 34) and inverting the resulting transposed row (line 35). The two tone rows are then passed to the `makeAltScore` function that creates a list of measures by first invoking `genAltChords` on the tone rows and then invoking `gen4cMeasures` on the chords (lines 24-30). The `genAltChords` function generates a list of single note chords that alternate between the 1 and -1 registers and applies pattern matching and recursion to accomplish this (lines 2-5). The `gen4cMeasures` function also applies pattern matching and recursion to generate a list of measures using the input chord list. `gen4cMeasures` uses the library routine `fillMeasure` to check if a list of chords fills a measure, which is four beats. `gen4cMeasures` also pads a measure with rest quarter notes if the size of the chord list is less than four in order to fill up the measure. The score is generated with the `genScore` keyword using the list of measures (line 37). The SMURF compiler translates the keyword **genScore** to C+OpenGL code to create a music score sheet.

```
1:  //Create a list of single note chords that alternate between registers 1/-1
2:  genAltChords::[int] -> [Chord]
3:  genAltChords [] = []
4:  genAltChords (x:y:ys) = [(x,Time 4,1)]:[(y,Time 4,-1)]:genAltChords ys
5:  genAltChords (x:xs) = [[(x,Time 4,1)]]
6:
7:  //Create a list of measures consisting of 4 chords per measure
8:  gen4cMeasures::[Chord] -> [Measure]
9:  gen4cMeasures [] = []
10: gen4cMeasures (w:x:y:z:zs) =
11:  if fillMeasure [w,x,y,z] then [w,x,y,z]:gen4cMeasures zs else gen4cMeasures zs
12: gen4cMeasures (w:ws) =
13:  let ms = [w,(-1,4,0),(-1,4,0),(-1,4,0)]
14:  in if fillMeasure ms then ms:gen4cMeasures ws else gen4cMeasures ws
15: gen4cMeasures (w:x:xs) =
16:  let ms = [w,x,(-1,4,0),(-1,4,0)]
17:  in if fillMeasure ms then ms:gen4cMeasures xs else gen4cMeasures xs
18: gen4cMeasures (w:x:y:ys) =
19:  let ms = [w,x,y,(-1,4,0)]
20:  in if fillMeasure ms then ms:gen4cMeasures ys else gen4cMeasures ys
21:
22: /*Create a list of measures by combining the resulting measure(s)
23:   produced from tone rows r1 and r2. */
24: makeAltScore::[int]->[int]->[Measure]
25: makeAltScore r1 r2 =
26:  let ch1 = genAltChords r1
27:      ch2 = genAltChords r2
28:      m1  = gen4cMeasures ch1
29:      m2  = gen4cMeasures ch2
30:  in m1 ++ m2
31:
32: //Create a serial score
33: let prime = [0,2,4,6,8,10,1,3,5,7,9,11]
34:     p3 = trans 3 prime
35:     i3 = inver p3
36:     score = makeAltScore p3 i3
37: in genScore score
```

Figure 4: Example SMURF program.

References

[1] A. Appleby, "Accidentals." `http://www.music-mind.com/Music/mpage4.HTM`, Sept. 2013.

[2] M. DeVoto, "Twelve-tone technique: A primer." `http://www.tufts.edu/~mdevoto/12TonePrimer.pdf`, Sept. 2013.