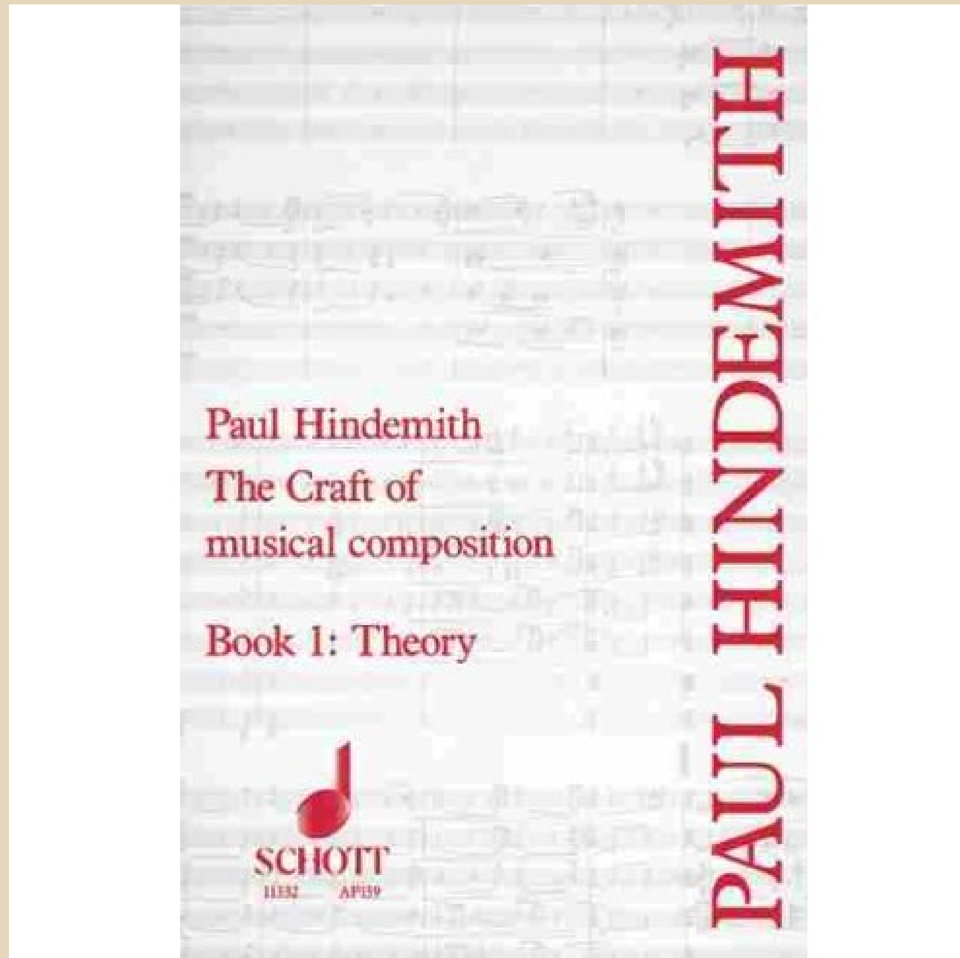
Hindemith

in Haskell


Paul Hindemith
The Craft of
musical composition
Book 1: Theory
SCHOTT
PAUL HINDEMITH

Hindemith's Problem

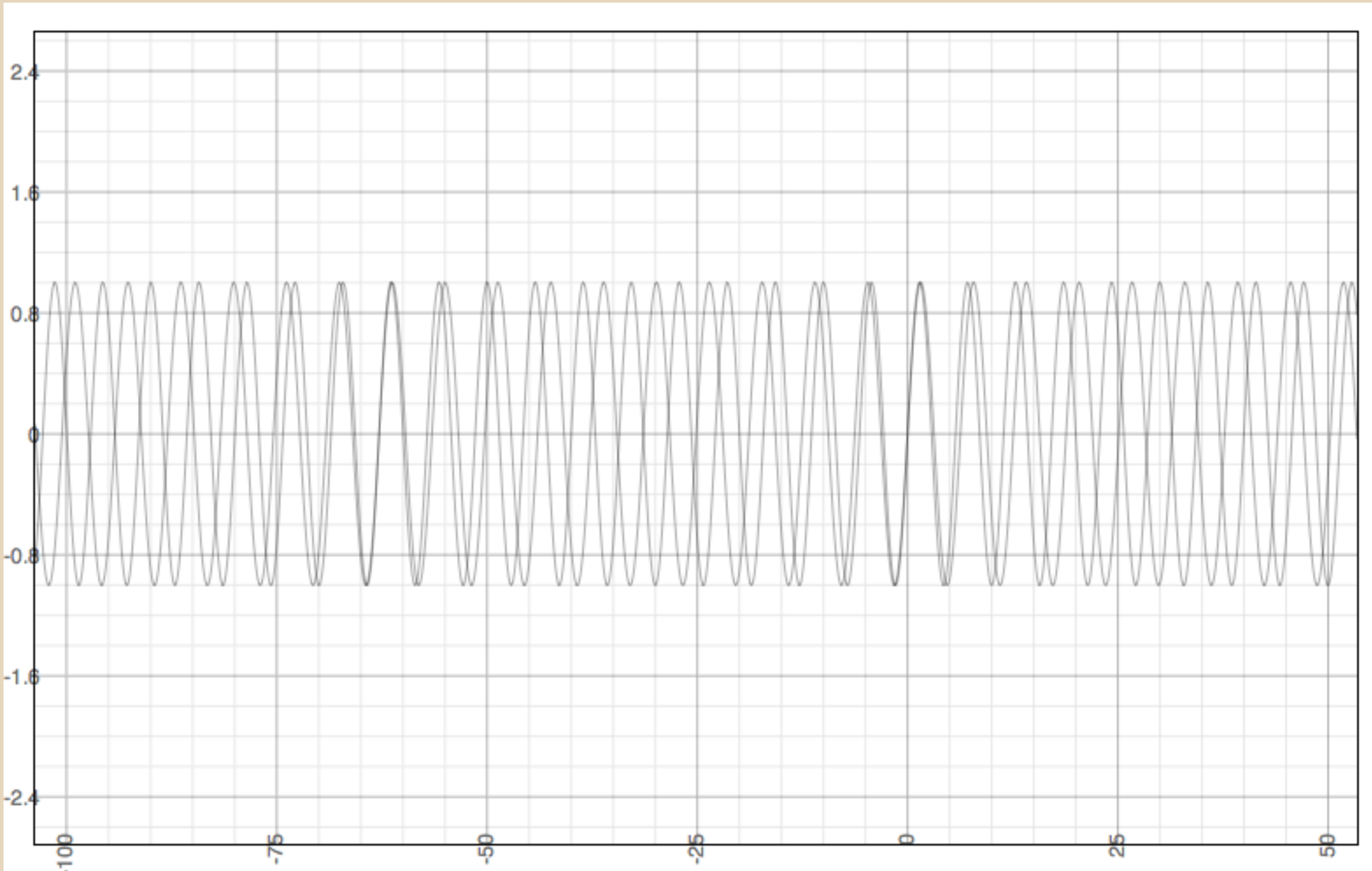
- Discarding the old rules
- Replacing them with what?

Fundamentals of Music

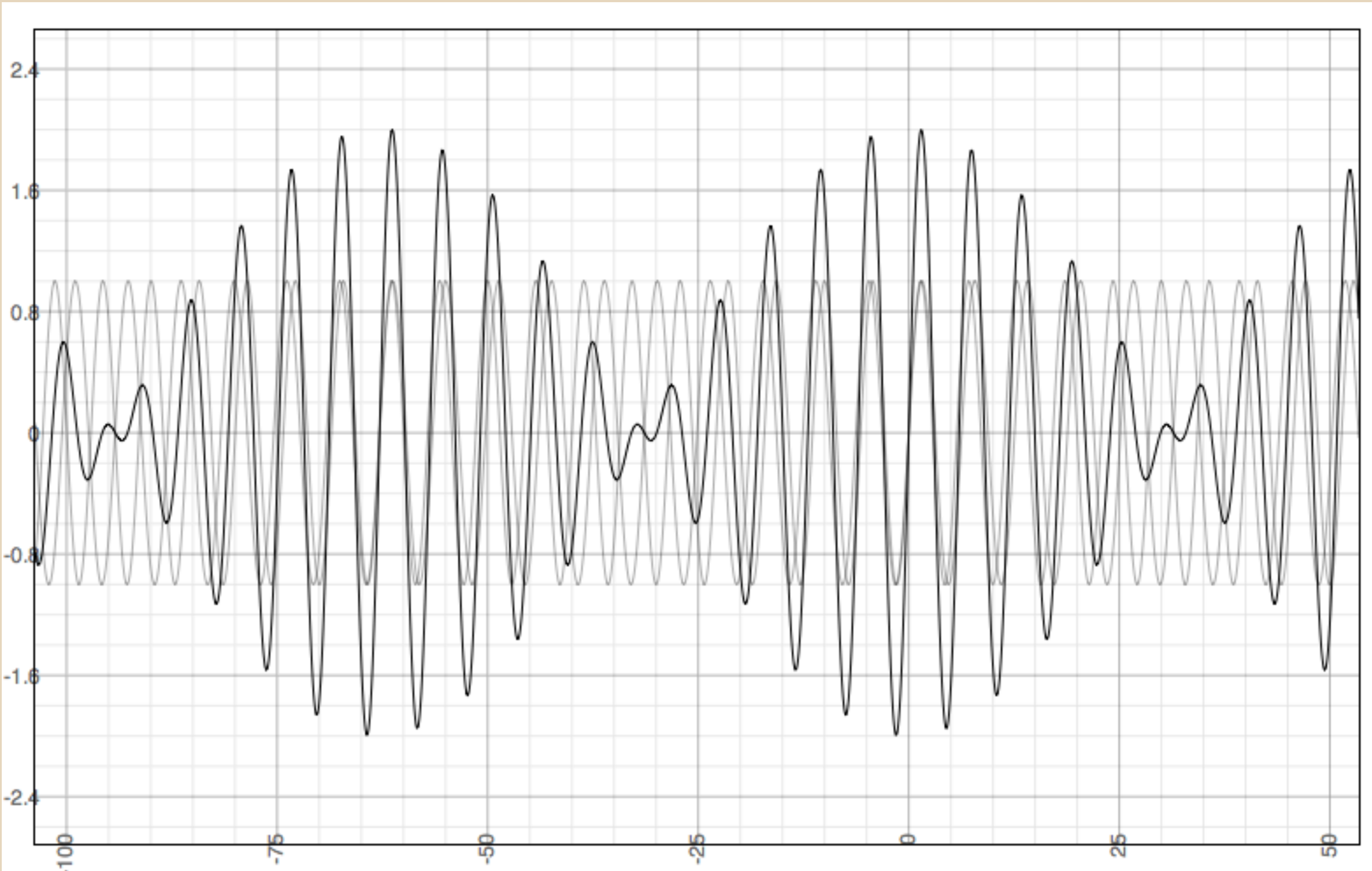
- Notes
  - Melodies
- Intervals
  - Harmonies
- Chords
  - Progressions

Fundamentals of Music

- Sound is waves in air
  - notes have characteristic frequencies
- Frequency doubling is special
  - the "octave"
- Notes playing together generate interference
- Musical instruments aren't perfect
  - each note has "overtones"



2.4
1.6
0.8
0
-0.8
-1.6
-2.4
-100
-75
-50
-25
0
25
50



2.4
1.6
0.8
0
-0.8
-1.6
-2.4
-100
-75
-50
-25
0
25
50

Overtones

- If f is the root frequency of the note, there will be overtones at nf for integer n

Scales

- 12 notes

- C .. (c♯/d♭) .. d .. (d♯/e♭) .. e .. f .. (f♯/g♭) .. g .. (g♯/a♭) .. a .. (a♯/b♭) .. b .. C

Our Data Structure

```
type Pitch = Double

data DerivedTone a = O (DerivedTone a) Int
                   | R (DerivedTone a) Int
                   | Base a
                     deriving (Show)

c :: DerivedTone Pitch
c = Base 64
```

Our Abstract Interface

```
class Note a where
  pitch :: a -> Pitch
  overtone :: a -> Int -> a
  undertone :: a -> Int -> a
```

Implementations

```
instance Note Pitch where
    pitch = id
    overtone p n = p * fromIntegral n
    undertone p n = p / fromIntegral n


instance Note (DerivedTone Pitch) where
    pitch (O p n) = fromIntegral n * pitch p
    pitch (R p n) = pitch p / fromIntegral n
    pitch (Base p) = p
    overtone (R p n) m | n == m = p
    overtone p n = O p n
    undertone (O p n) m | n == m = p
    undertone p n = R p n
```

Convenience Methods

```
instance Eq (DerivedTone Pitch) where
    a == b = pitch a == pitch b

instance Ord (DerivedTone Pitch) where
    a < b = pitch a < pitch b
         a > b = pitch a > pitch b

octave x = overtone x 2

overtoneRatio over root =
     flip undertone root . flip overtone over
(//) = overtoneRatio
```

Pythagorean Tuning

C .. G .. D .. A .. E .. B .. F♯

C .. F .. B♭ .. E♭ .. A♭ .. D♭ .. G♭

Pythagorean Tuning

```
nextTone tone = (3 // 2) tone
prevTone tone = (2 // 3) tone
normalise base tone = if tone > octave base
                      then normalise base (tone `undertone` 2)
                      else (
                        if pitch tone < pitch base
                        then normalise base (octave tone)
                        else tone)

ptones = map (normalise c) $ (take 7 $ iterate nextTone c) ++
                             (take 6 . drop 1 $ iterate prevTone c)

(pc:pg:pd:pa:pe:pb:pfs:pf:pbb:peb:pab:pdb:pgb :[]) = ptones

pscale = pc:pdb:pd:peb:pe:pf:pfs:pgb:pg:pab:pa:pbb:pb:[]
```

Pythagorean Tuning

G♭	D♭	A♭	E♭	B♭	F	C	G	D	A	E	B	F♯
1024 729	256 243	128 81	32 27	16 9	4 3	1 1	3 2	9 8	64 27	81 64	243 128	729 512

Five-Limit Tuning

- Thirds are important

- A good major third requires a ratio of 5/4
  - the strongest generated note is 1/4 the root tone

- A good minor third requires a ratio of 6/5
  - the strongest generated note is 1/5 the root tone, which is two octaves below 4/5, which is a major third below the root

- Use factors of 2, 3 and 5

Five-Limit Tuning

```
factorRows = [(1, 9), (1, 3), (1, 1), (3, 1), (9, 1)]
factorCols = [(5, 1), (1, 1), (1, 5)]

fltones = map (normalise c)
            [R (O c (a*a')) (b*b') | (a, b) <- factorRows,
                                      (a', b') <- factorCols]

(fld1:flbb1:flgb:fla:flf:fldb:fle:flc:flab:flb:flg:fleb:
flfs:fld2:flbb2:[]) = fltones

flscale = flc:fldb:fld2:fleb:fle:flf:flfs:flg:flab:fla:
flbb2:flb:[]
```

Five-Limit Tuning

	1/9	1/3	1	3	9
5	D (10/9)	A (5/3)	E (5/4)	B (15/8)	F♯ (45/32)
1	B♭ (16/9)	F (4/3)	C (1/1)	G (3/2)	D (9/8)
1/5	G♭(64/45)	D♭(16/15)	A♭(8/5)	E♭(6/5)	B♭(9/5)

Equal Temperament

```
ratio = 2 ** (1/12)
etscale = map Base . take 12 $ iterate (* ratio) 64
```

C	C♯	D	E♭	E	F	F♯	G	A♭	A	B♭	B
64	67.81	71.84	76.11	80.63	85.43	90.51	95.89	101.59	107.63	114.04	120.82
			76.8	80			96				

Hindemith's Method

```
firstRatios base =
            [result | over <- [1 .. 6], root <- [1 .. 6],
             let result = (over // root) base,
             result > base, result < octave base]
firstResults =  nub . firstRatios
```

G (3/2), F (4/3), A (5/3), E (5/4), E $\flat$ $(6/5)$

Hindemith's Method

```
secondRatios base =
  [result | over <- [1 .. 6], root <- [1 .. 6], root > over,
   let result = octave $ (over // root) base,
   result > base, result < octave base]
secondResults base = nub (secondRatios base)
                  \\ firstRatios base
```

A ♭ (8/5)

Hindemith's Method

```
thirdRatios base =
   [result | tone <- take 4 $ firstResults base,
    over <- [3 .. 6], root <- [2 .. 6],
    let result = (over // root) tone,
    tone `overtone` over < (base `overtone` 6),
    result > base, result < octave base]
thirdResults base = (nub (thirdRatios base)
                 \\ firstRatios base)
                 \\ secondRatios base
```

D (9/8), B $\flat$ (16/9), D $\flat$ (16/15), B (15/8)

Hindemith's Method

```
tritones base = [
   overtoneRatio 4 5 (thirdResults base!!1),
   overtoneRatio 4 3 (thirdResults base!!2),
   overtoneRatio 5 4 (thirdResults base!!0),
   overtoneRatio 3 4 (thirdResults base!!3)
   ]
```

```
tones base = firstResults base ++
             secondResults base ++
             thirdResults base ++
  [tritones base !! 1, tritones base !! 2]

(g:f:a:e:eb:ab:d:bb:db:b:gb:fs:[]) = tones c

scale = c:db:d:eb:e:f:fs:g:ab:a:bb:b:[]
```

Generating Melodies

- Use the relatedness of notes as a measure of how strong or resolved a progression from one to the next sounds

- Start with strong progressions, introduce tension by weakening the progressions, then bring strong progressions back at the end of each 'phrase' of the melody.

More fun

Comparing other scales to Hindemith's

Next Time...

- Intervals
- Chords
- Chord progressions

Any Questions?