Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Functional Modelling of Musical Harmony

José Pedro Magalhães

Dept. of Information and Computing Sciences, Utrecht University
http://dreixel.net

February 29, 2012

This talk

- Modelling musical harmony using Haskell

This talk

- Modelling musical harmony using Haskell
- Modelling musica~~l harmony~~ using Haskell

This talk

- Modelling musical harmony using Haskell
- Modelling music~~al harmony~~ using Haskell
- From textual chord labels to analysis trees

This talk

- Modelling musical harmony using Haskell
- Modelling musical ~~al harmony~~ using Haskell
- From textual chord labels to analysis trees
- Avoiding programming

This talk

- Modelling musical harmony using Haskell
- Modelling music~~al harmony~~ using Haskell
- From textual chord labels to analysis trees
- Avoiding programming
- Why Haskell is a better fit than Java for harmony modelling

Outline

Harmony

Basic musical notions in Haskell

A harmony model in Haskell

From chord text to harmonic structure

Increasing model complexity

Conclusion

Outline

Harmony

Basic musical notions in Haskell

A harmony model in Haskell

From chord text to harmonic structure

Increasing model complexity

Conclusion

What is harmony?

Ton *SDom* *Dom* *Ton*

I IV V/V V I

C F D^7 G^7 C

- Harmony arises when at least two notes sound at the same time
- Harmony induces tension and release patterns, that can be described by music theory and music cognition
- The internal structure of the chord has a large influence on the consonance or dissonance of a chord
- The surrounding context also has a large influence

Simplified harmony theory I

- A chord is a group of tones separated by intervals of roughly the same size.
- All music is made out of chords (whether explicitly or not).
- There are 12 different notes. Instead of naming them, we number them relative to the first and most important one, the tonic. So we get I, II♭, II . . . VI♯, VII.
- A chord is built on a root note. So I also stands for the chord built on the first degree, V for the chord built on the fifth degree, etc.
- So the following is a chord sequence: I IV II^7 V^7 I.

Simplified harmony theory II

Models for musical harmony explain the harmonic progression in music:

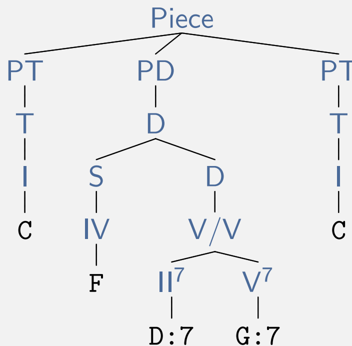
- Everything works around the tonic (I).
- The dominant (V) leads to the tonic.
- The subdominant (IV) tends to lead to the dominant.
- Therefore, the I IV V I progression is very common.
- There are also secondary dominants, which lead to a relative tonic. For instance, II^7 is the secondary dominant of V, and I^7 is the secondary dominant of IV.
- So you can start with I, add one note to get I^7, fall into IV, change two notes to get to II^7, fall into V, and then finally back to I.

An example harmonic analysis

Ton *SDom* *Dom* *Ton*

I IV V/V V I

C F D^7 G^7 C



Why are harmony models useful?

Having a model for musical harmony allows us to automatically determine the functional meaning of chords in the tonal context.

The model determines which chords "fit" on a particular moment in a song.

Why are harmony models useful?

Having a model for musical harmony allows us to automatically determine the functional meaning of chords in the tonal context.

The model determines which chords “fit” on a particular moment in a song. This is useful for:

- Musical information retrieval (find songs similar to a given song)
- Audio and score recognition (improving recognition by knowing which chords are more likely to appear)
- Music generation (create sequences of chords that conform to the model)

Outline

Harmony

Basic musical notions in Haskell

A harmony model in Haskell

From chord text to harmonic structure

Increasing model complexity

Conclusion

Why Haskell?

Haskell is a strongly-typed pure functional programming language:

Why Haskell?

Haskell is a strongly-typed pure functional programming language:

Strongly-typed All values are classified by their type, and types are known at compile time (statically). This gives us strong guarantees about our code, avoiding many common mistakes.

Why Haskell?

Haskell is a strongly-typed pure functional programming language:

Strongly-typed
: All values are classified by their type, and types are known at compile time (statically). This gives us strong guarantees about our code, avoiding many common mistakes.

Pure
: There are no side-effects, so Haskell functions are like mathematical functions.

Why Haskell?

Haskell is a strongly-typed pure functional programming language:

Strongly-typed All values are classified by their type, and types are known at compile time (statically). This gives us strong guarantees about our code, avoiding many common mistakes.

Pure There are no side-effects, so Haskell functions are like mathematical functions.

Functional A Haskell program is an expression, not a sequence of statements. Functions are first class citizens, and explicit state is avoided.

Notes

```
data Root    = A | B | C | D | E | F | G
type Octave = Int
data Note    = Note Root Octave
```

Notes

```
data Root    = A | B | C | D | E | F | G
type Octave = Int
data Note    = Note Root Octave

a4, b4, c4, d4, e4, f4, g4 :: Note
a4 = Note A 4
b4 = Note B 4
c4 = Note C 4
d4 = Note D 4
e4 = Note E 4
f4 = Note F 4
g4 = Note G 4
```

Melody

```
type Melody = [Note]

cMajScale :: Melody
cMajScale = [c4, d4, e4, f4, g4, a4, b4]
```

Melody

```
type Melody = [Note]

cMajScale :: Melody
cMajScale = [c4, d4, e4, f4, g4, a4, b4]

cMajScaleRev :: Melody
cMajScaleRev = reverse cMajScale
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Melody

```
type Melody = [Note]

cMajScale :: Melody
cMajScale = [c4, d4, e4, f4, g4, a4, b4]

cMajScaleRev :: Melody
cMajScaleRev = reverse cMajScale

reverse :: [α] → [α]
reverse []      = []
reverse (h : t) = reverse t ++ [h]
(++) :: [α] → [α] → [α]
(++) = ...
```

Transposition

Transposing a melody one octave higher:

```
octaveUp :: Octave → Octave
octaveUp n = n + 1

noteOctaveUp :: Note → Note
noteOctaveUp (Note r o) = Note r (octaveUp o)

melodyOctaveUp :: Melody → Melody
melodyOctaveUp m = map noteOctaveUp m
```

Generation, analysis

Building a canon from a melody:

$$\text{canon} :: \text{Melody} \rightarrow \text{Melody}$$
$$\text{canon}\ m = m \mathbin{+\!\!+} \text{canon}\ m$$

Generation, analysis

Building a canon from a melody:

$$\text{canon} :: \text{Melody} \to \text{Melody}$$
$$\text{canon } m = m \mathbin{+\!\!+} \text{canon } m$$

Is a given melody in C major?

$$\text{root} :: \text{Note} \to \text{Root}$$
$$\text{root } (\text{Note } r \; o) = r$$

$$\text{isCMaj} :: \text{Melody} \to \text{Bool}$$
$$\text{isCMaj} = (\equiv \text{map root cMajScaleRev}) \circ \text{sort} \circ \text{nub} \circ \text{map root}$$

“Details” left out

We have seen only a glimpse of music representation in Haskell.

- Rhythm
- Accidentals
- Intervals
- Voicing
- ...

A good pedagogical reference on using Haskell to represent music: `http://di.uminho.pt/~jno/html/ipm-1011.html`

A serious library for music manipulation: `http://www.haskell.org/haskellwiki/Haskore`

Outline

Harmony

Basic musical notions in Haskell

A harmony model in Haskell

From chord text to harmonic structure

Increasing model complexity

Conclusion

Representing harmony as a datatype, naively

$$
\begin{array}{llllll}
\textbf{data}\ \text{Piece} & = \text{Piece} & [\text{Phrase}] & & & \\
\textbf{data}\ \text{Phrase} & = \text{PT} & \text{Ton} & \mid \text{PD} & \text{Dom} & \\
\textbf{data}\ \text{Ton} & = \text{T}_{\text{IMaj}} & \text{DegNum} & & & \\
\textbf{data}\ \text{Dom} & = \text{D}_{\text{VMaj}} & \text{DegNum} & \mid \text{D}_{\text{SDom}} & \text{SDom} & \text{Dom} \\
\textbf{data}\ \text{SDom} & = \text{S}_{\text{IVMaj}} & \text{DegNum} & & & \\
\textbf{data}\ \text{DegNum} & = \text{I} \mid \text{II} \mid \text{III} \ldots & & & &
\end{array}
$$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Representing harmony as a datatype, naively

```
data Piece  = Piece [Phrase]
data Phrase = PT Ton | PD Dom
data Ton    = T_IMaj DegNum
data Dom    = D_VMaj DegNum | D_SDom SDom Dom
data SDom   = S_IVMaj DegNum
data DegNum = I | II | III ...
```

Problem: the term Piece [PT (T_{IMaj} II)] typechecks, but represents an invalid harmonic structure. T_{IMaj} should only take I as argument.

How can we enforce this? (And why do we want it?)

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Representing harmony as a datatype I

```
data Piece  = Piece    [Phrase]
data Phrase = PT       Ton        | PD Dom
data Ton    = T_IMaj   (Deg I)
data Dom    = D_VMaj   (Deg V)    | D_SDom SDom Dom
data SDom   = S_IVMaj  (Deg IV)
```

```
data Deg δ = Deg DegNum
```

We do not export the Deg constructor. So how do we build Degs?

Representing harmony as a datatype II

$$\mathsf{deg} :: \mathsf{ToDegree}\ \delta \Rightarrow \delta \rightarrow \mathsf{Deg}\ \delta$$
$$\mathsf{deg}\ \mathsf{d} = \mathsf{Deg}\ (\mathsf{toDegree}\ \mathsf{d})$$

We need degrees at the type level:

```
data I; data II; ... data VII;
```

And type-to-value conversions for degrees:

```
class    ToDegree δ  where toDegree :: δ → DegNum
instance ToDegree I  where toDegree _ = I
instance ToDegree II where toDegree _ = II
...   -- instances for all degrees
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Representing harmony as a datatype III

Now the term Piece [PT (T_{IMaj} (deg ($\bot$ :: II)))] does not typecheck, as we wanted:

```
*ghci> Piece [PT (TIMaj (deg (undefined :: II)))]

    Couldn't match expected type ‘I’
      with actual type ‘II’
```

Representing harmony as a datatype IV

We have a well-typed harmony model in Haskell that accepts only "correct" harmonic sequences.

Here we have only seen a very basic subset of our model. In reality, it consists of 14 datatypes with a total of 46 constructors.

Also, while developing the model we have to change it often, trying to find a balance between complexity and expressiveness.

Outline

Harmony

Basic musical notions in Haskell

A harmony model in Haskell

From chord text to harmonic structure

Increasing model complexity

Conclusion

Parsing chord sequences I

We now want to parse (textual) chord sequences into our datatype representing musical harmony:

```
class Parse α where
  parse :: ParserMusic α
```

Parsing chord sequences II

Most instances are trivial:

$$
\begin{aligned}
&\textbf{instance } \mathsf{Parse}\ \mathsf{Phrase}\ \textbf{where}\\
&\quad \mathsf{parse} = \mathsf{PT} <\$> \mathsf{parse}\\
&\qquad\quad <|> \mathsf{PD} <\$> \mathsf{parse}\\
&\textbf{instance } \mathsf{Parse}\ \mathsf{Ton}\ \textbf{where}\\
&\quad \mathsf{parse} = \mathsf{T}_{\mathsf{IMaj}} <\$> \mathsf{parse}\\
&\textbf{instance } \mathsf{Parse}\ \mathsf{Dom}\ \textbf{where}\\
&\quad \mathsf{parse} = \mathsf{D}_{\mathsf{VMaj}} <\$> \mathsf{parse}\\
&\qquad\quad <|> \mathsf{D}_{\mathsf{SDom}} <\$> \mathsf{parse} <*> \mathsf{parse}\\
&\ldots
\end{aligned}
$$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Parsing chord sequences II

Most instances are trivial:

```
instance Parse Phrase where
  parse = PT <$> parse
      <|> PD <$> parse
instance Parse Ton where
  parse = T_IMaj <$> parse
instance Parse Dom where
  parse = D_VMaj <$> parse
      <|> D_SDom <$> parse <*> parse
...
```

So trivial that we do not write them; we use a generic parser.

Parsing degrees

For degrees we need an adhoc parser:

$$\textbf{instance}\ (\text{ToDegree}\ \delta) \Rightarrow \text{Parse}\ (\text{Deg}\ \delta)\ \textbf{where}$$
$$\quad \text{parse} = \text{pChord}\ (\text{toDegree}\ (\bot :: \delta))$$

$$\text{pChord} :: \text{DegNum} \rightarrow \text{ParserMusic}\ (\text{Deg}\ \delta)$$
$$\text{pChord} = \ldots$$

Again we use the type-to-value conversion class ToDegree.

Outline

Harmony

Basic musical notions in Haskell

A harmony model in Haskell

From chord text to harmonic structure

Increasing model complexity

Conclusion

Increasing complexity

The model we presented so far is very simple and cannot account for:

Repeated chords $\underline{G_{Maj}\ G_{Maj}}\ C_{Maj}$

Diatonic secondary dominants $\underline{E_{min}\ A_{min}\ D_{min}}\ G^7\ C_{Maj}$

Chromatic secondary dominants $\underline{A^7\ D^7}\ G^7\ C_{Maj}$

Minor key dominant borrowing $\underline{G_{min}}\ C_{Maj}$

Tritone substitutions $\underline{A\flat^7}\ G^7\ C_{Maj}$

...

Handling repeated chords

Easy: adapt the Deg type and pChord:

```
data Deg δ = Deg {chordRole :: DegNum
                 , numReps   :: Int}

pChord :: DegNum → ParserMusic (Deg δ)
pChord = . . .    -- adapted accordingly
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Secondary dominants—datatype representation

Here the fun begins! We adapt the Deg type again:

$$\textbf{type}\ \mathsf{Deg}_{\mathsf{SD}}\ \delta = \mathsf{BDeg}_{\mathsf{SD}}\ \delta$$

$$\begin{aligned}&\textbf{data}\ \mathsf{BDeg}_{\mathsf{SD}}\ \delta\ \textbf{where}\\ &\quad \mathsf{Cons} :: \mathsf{BDeg}_{\mathsf{SD}}\ (\mathsf{Perf}_{\mathsf{V}}\ \delta) \to \mathsf{Deg}\ \delta \to \mathsf{BDeg}_{\mathsf{SD}}\ \delta\\ &\quad \mathsf{Base} :: \mathsf{Deg}\ \delta \to \mathsf{BDeg}_{\mathsf{SD}}\ \delta\end{aligned}$$

Secondary dominants—datatype representation

Here the fun begins! We adapt the Deg type again:

$$\textbf{type}\ \mathsf{Deg}_{\mathsf{SD}}\ \delta = \mathsf{BDeg}_{\mathsf{SD}}\ \delta$$

$$\textbf{data}\ \mathsf{BDeg}_{\mathsf{SD}}\ \delta\ \textbf{where}$$

$$\quad \mathsf{Cons} :: \mathsf{BDeg}_{\mathsf{SD}}\ (\mathsf{Perf}_{\mathsf{V}}\ \delta) \rightarrow \mathsf{Deg}\ \delta \rightarrow \mathsf{BDeg}_{\mathsf{SD}}\ \delta$$

$$\quad \mathsf{Base} :: \mathsf{Deg}\ \delta \rightarrow \mathsf{BDeg}_{\mathsf{SD}}\ \delta$$

$$\textbf{data}\ \mathsf{Deg}\ \delta = \mathsf{Deg} \ldots \quad \text{-- as before}$$

$$\textbf{type family}\ \mathsf{Perf}_{\mathsf{V}}\ \delta \quad \text{-- computes the perfect fifth}$$

$$\textbf{type instance}\ \mathsf{Perf}_{\mathsf{V}}\ \mathsf{I} = \mathsf{V}$$

$$\textbf{type instance}\ \mathsf{Perf}_{\mathsf{V}}\ \mathsf{V} = \mathsf{II}$$

$$\ldots$$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Example

Parsing the sequence G_{min} C^7 G_{min} C^7 F_{Maj} D^7 G^7 C_{Maj}:

- Piece
  - PD
    - D
      - S
        - V/IV
          - V/I
            - V_{min}
              - `G:min`
          - I^7
            - `C:7`
        - IV
          - `ins`
      - D
        - S
          - V/IV
            - V/I
              - V_{min}
                - `G:min`
            - I^7
              - `C:7`
          - IV
            - `F:maj`
        - D
          - V/V
            - II^7
              - `D:7`
          - V^7
            - `G:7`
  - PT
    - T
      - I
        - `C:maj`

Harmonic similarity

- A practical application of a harmony model is to estimate harmonic similarity between songs
- The more similar the trees, the more similar the harmony
- We don't want to write a diff algorithm for our complicated model; we get it automatically by using a generic diff
- The generic diff is a type-safe tree-diff algorithm. It is part of Eelco Lempsink's MSc work here in Utrecht
- Generic, thus working for any model, and independent of changes to the model

Outline

Harmony

Basic musical notions in Haskell

A harmony model in Haskell

From chord text to harmonic structure

Increasing model complexity

Conclusion

Comparison to a Java approach

Comparing the current solution to an earlier Java approach we find that:

- Error-correction makes the grammar simpler, and ensures parsing never fails
- GADTs allow us to control grammar ambiguity
- GADTs help simplifying the grammar
- The Haskell parser is faster and more robust
- The generic diff performs well
- The Haskell approach has fewer source lines of code

Summary

What we have so far:

- A model for musical harmony as a Haskell GADT
- The datatypes can change, the code does not have to:
  - Generic parser
  - Generic pretty-printer
  - Generic diff
- Multiple models, lots of code reuse
- When chords do not fit the model: error correction
- Recognizing harmony from audio sources

Play with it!

http://hackage.haskell.org/package/HarmTrace