

Advanced Functional Programming in Industry

José Pedro Magalhães



November 21, 2014
London, United Kingdom

Introduction

- ▶ Haskell: a statically typed, lazy, purely functional language
- ▶ Modelling musical harmony using Haskell
- ▶ Applications of a model of harmony:
 - ▶ Musical analysis
 - ▶ Finding cover songs
 - ▶ Generating chords and melodies
 - ▶ Correcting errors in chord extraction from audio sources
 - ▶ Chordify—a web-based music player with chord recognition

Demo: Chordify

Demo:

chordify[®]

<http://chordify.net>

Table of Contents

Harmony

Haskell

Harmony analysis

Harmonic similarity

Music generation

Chord recognition: Chordify

What is harmony?

The diagram illustrates a harmonic progression on a treble clef staff. The chords and their functional labels are as follows:

| Chord | Functional Label |
|----------------|------------------|
| C | Ton I |
| F | SDom IV |
| D ⁷ | Dom V/V |
| G ⁷ | Dom V |
| C | Ton I |

- ▶ 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

What is harmony?

The image shows a musical staff with five chords. Above the staff, the functional categories are labeled: *Ton*, *SDom*, *Dom*, and *Ton*. Below these, the Roman numerals are: I, IV, V/V, V, and I. A bracket groups the V/V and V chords under the *Dom* label. Below the staff, the chord names are: C, F, D⁷, G⁷, and C. The notes for each chord are shown as black dots on the staff lines.

- ▶ 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

Demo: how harmony affects melody

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 \flat , II \sharp . . . VI \sharp , 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.

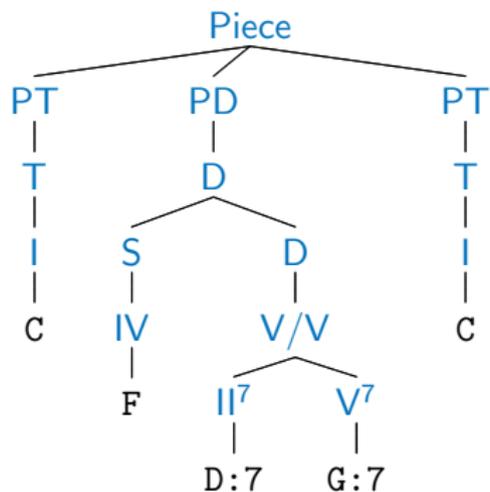
Simplified harmony theory II

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

Harmonic analysis of a chord progression on a treble clef staff. The notes are C, F, D⁷, G⁷, and C. Above the staff, the functional categories are labeled: *Ton* (I), *SDom* (IV), *Dom* (V/V, V), and *Ton* (I). The *Dom* label is bracketed over the V/V and V chords.



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)

Table of Contents

Harmony

Haskell

Harmony analysis

Harmonic similarity

Music generation

Chord recognition: Chordify

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
```

Melody

```
type Melody = [Note]
```

```
cMajScale :: Melody
```

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

```
cMajScaleRev :: Melody
```

```
cMajScaleRev = reverse cMajScale
```

```
reverse :: [ $\alpha$ ] → [ $\alpha$ ]
```

```
reverse [] = []
```

```
reverse (h : t) = reverse t ++ [h]
```

```
(++) :: [ $\alpha$ ] → [ $\alpha$ ] → [ $\alpha$ ]
```

```
(++) = ...
```

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:

canon :: Melody \rightarrow Melody

canon m = m $\#$ canon m

Generation, analysis

Building a canon from a melody:

`canon :: Melody → Melody`

`canon m = m ++ canon m`

Is a given melody in C major?

`root :: Note → Root`

`root (Note r o) = r`

`isCMaj :: Melody → Bool`

`isCMaj = all (∈ cMajScale) ∘ 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>

Table of Contents

Harmony

Haskell

Harmony analysis

Harmonic similarity

Music generation

Chord recognition: Chordify

Application: harmony analysis

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

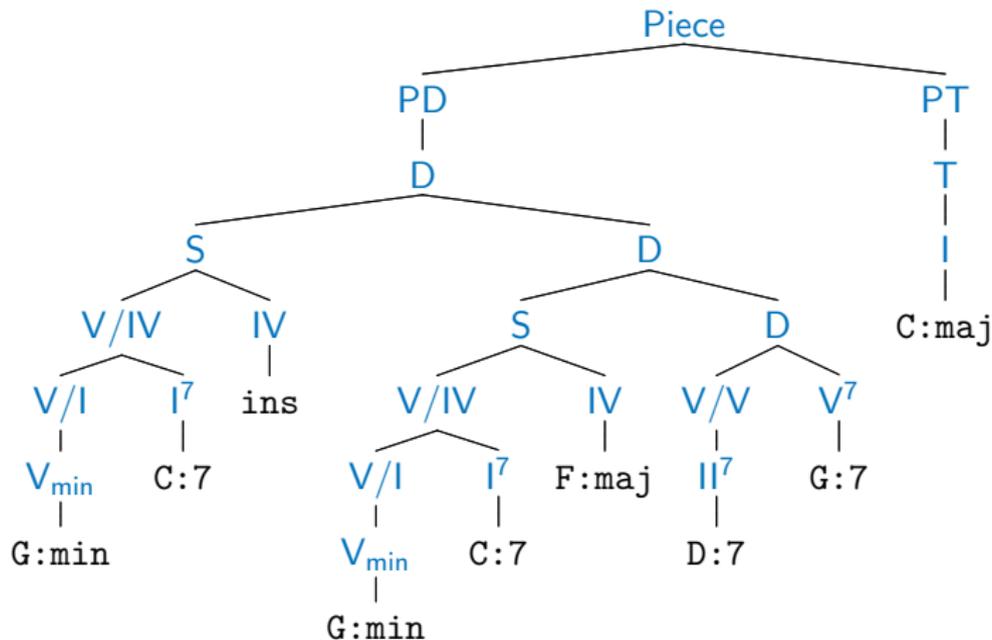


Table of Contents

Harmony

Haskell

Harmony analysis

Harmonic similarity

Music generation

Chord recognition: Chordify

Application: 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, part of a student's MSc work at Utrecht University
- ▶ Generic, thus working for any model, and independent of changes to the model

Table of Contents

Harmony

Haskell

Harmony analysis

Harmonic similarity

Music generation

Chord recognition: Chordify

Application: automatic harmonisation of melodies

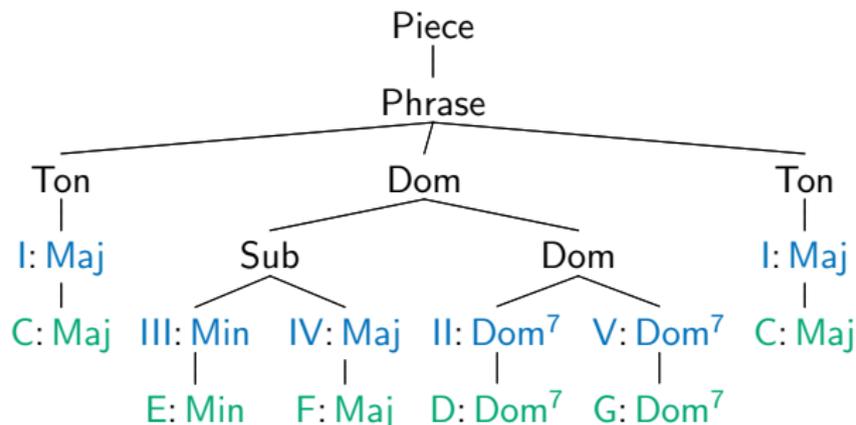
Another practical application of a harmony model is to help selecting good harmonisations (chord sequences) for a given melody:



The image displays a musical score for a single system. The upper staff is in the treble clef, showing a melody in C major with a common time signature. The lower staff is in the bass clef, showing a sequence of chords. The chords are labeled with Roman numerals: V, III, I, III, II, IV, III, IV, V. The melody consists of the following notes: C4, D4, E4, F4, G4, A4, B4, C5, B4, A4, G4, F4, E4, D4, C4.

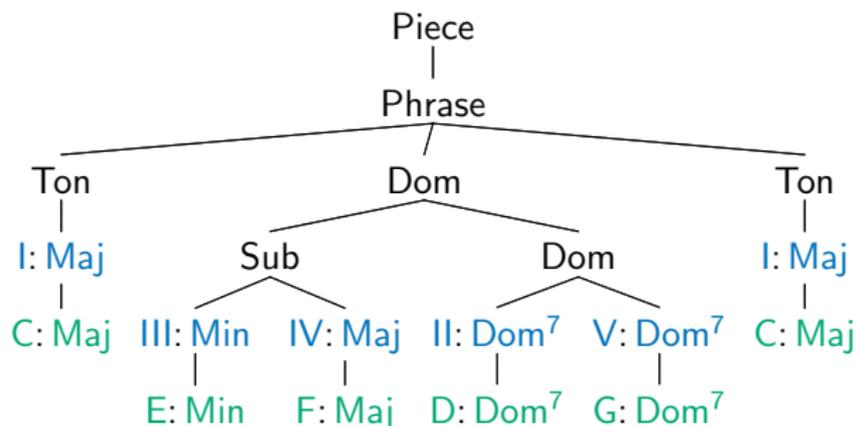
We generate candidate chord sequences, parse them with the harmony model, and select the one with the least errors.

Visualising harmonic structure



You can see this tree as having been produced by taking the chords in green as input...

Generating harmonic structure



You can see this tree as having been produced by taking the chords in green as input... or the chords might have been dictated by the structure!

A functional model of harmony

$\text{Piece}_{\mathfrak{M}} \rightarrow [\text{Phrase}_{\mathfrak{M}}]$ ($\mathfrak{M} \in \{\text{Maj}, \text{Min}\}$)

A functional model of harmony

$\text{Piece}_{\mathfrak{M}} \rightarrow [\text{Phrase}_{\mathfrak{M}}] \quad (\mathfrak{M} \in \{\text{Maj}, \text{Min}\})$

$\text{Phrase}_{\mathfrak{M}} \rightarrow \text{Ton}_{\mathfrak{M}} \text{ Dom}_{\mathfrak{M}} \text{ Ton}_{\mathfrak{M}}$
| $\text{Dom}_{\mathfrak{M}} \text{ Ton}_{\mathfrak{M}}$

A functional model of harmony

$Piece_{\mathfrak{M}} \rightarrow [Phrase_{\mathfrak{M}}]$ ($\mathfrak{M} \in \{Maj, Min\}$)

$Phrase_{\mathfrak{M}} \rightarrow \begin{array}{c} Ton_{\mathfrak{M}} \quad Dom_{\mathfrak{M}} \quad Ton_{\mathfrak{M}} \\ | \quad \quad \quad Dom_{\mathfrak{M}} \quad Ton_{\mathfrak{M}} \end{array}$

$Ton_{Maj} \rightarrow I_{Maj}$

$Ton_{Min} \rightarrow I_{Min}^m$

A functional model of harmony

$\text{Piece}_{\mathfrak{M}} \rightarrow [\text{Phrase}_{\mathfrak{M}}]$ ($\mathfrak{M} \in \{\text{Maj}, \text{Min}\}$)

$\text{Phrase}_{\mathfrak{M}} \rightarrow \text{Ton}_{\mathfrak{M}} \text{ Dom}_{\mathfrak{M}} \text{ Ton}_{\mathfrak{M}}$
 | $\text{Dom}_{\mathfrak{M}} \text{ Ton}_{\mathfrak{M}}$

$\text{Ton}_{\text{Maj}} \rightarrow I_{\text{Maj}}$

$\text{Ton}_{\text{Min}} \rightarrow I_{\text{Min}}^m$

$\text{Dom}_{\mathfrak{M}} \rightarrow V_{\mathfrak{M}}^7$
 | $V_{\mathfrak{M}}$
 | $VII_{\mathfrak{M}}^0$
 | $\text{Sub}_{\mathfrak{M}} \text{ Dom}_{\mathfrak{M}}$
 | $II_{\mathfrak{M}}^7 V_{\mathfrak{M}}^7$

A functional model of harmony

$\text{Piece}_{\mathfrak{M}} \rightarrow [\text{Phrase}_{\mathfrak{M}}]$ ($\mathfrak{M} \in \{\text{Maj}, \text{Min}\}$)

$\text{Phrase}_{\mathfrak{M}} \rightarrow \text{Ton}_{\mathfrak{M}} \text{ Dom}_{\mathfrak{M}} \text{ Ton}_{\mathfrak{M}}$
 | $\text{Dom}_{\mathfrak{M}} \text{ Ton}_{\mathfrak{M}}$

$\text{Ton}_{\text{Maj}} \rightarrow \text{I}_{\text{Maj}}$

$\text{Ton}_{\text{Min}} \rightarrow \text{I}_{\text{Min}}^m$

$\text{Dom}_{\mathfrak{M}} \rightarrow \text{V}_{\mathfrak{M}}^7$
 | $\text{V}_{\mathfrak{M}}$
 | $\text{VII}_{\mathfrak{M}}^0$
 | $\text{Sub}_{\mathfrak{M}} \text{ Dom}_{\mathfrak{M}}$
 | $\text{II}_{\mathfrak{M}}^7 \text{ V}_{\mathfrak{M}}^7$

$\text{Sub}_{\text{Maj}} \rightarrow \text{II}_{\text{Maj}}^m$
 | IV_{Maj}
 | $\text{III}_{\text{Maj}}^m \text{ IV}_{\text{Maj}}$
 $\text{Sub}_{\text{Min}} \rightarrow \text{IV}_{\text{Min}}^m$

A functional model of harmony

$\text{Piece}_{\mathfrak{M}} \rightarrow [\text{Phrase}_{\mathfrak{M}}]$ ($\mathfrak{M} \in \{\text{Maj}, \text{Min}\}$)

$\text{Phrase}_{\mathfrak{M}} \rightarrow \text{Ton}_{\mathfrak{M}} \text{ Dom}_{\mathfrak{M}} \text{ Ton}_{\mathfrak{M}}$
 | $\text{Dom}_{\mathfrak{M}} \text{ Ton}_{\mathfrak{M}}$

$\text{Ton}_{\text{Maj}} \rightarrow \text{I}_{\text{Maj}}$

$\text{Ton}_{\text{Min}} \rightarrow \text{I}_{\text{Min}}^m$

$\text{Dom}_{\mathfrak{M}} \rightarrow \text{V}_{\mathfrak{M}}^7$
 | $\text{V}_{\mathfrak{M}}$
 | $\text{VII}_{\mathfrak{M}}^0$
 | $\text{Sub}_{\mathfrak{M}} \text{ Dom}_{\mathfrak{M}}$
 | $\text{II}_{\mathfrak{M}}^7 \text{ V}_{\mathfrak{M}}^7$

$\text{Sub}_{\text{Maj}} \rightarrow \text{II}_{\text{Maj}}^m$
 | IV_{Maj}
 | $\text{III}_{\text{Maj}}^m \text{ IV}_{\text{Maj}}$
 $\text{Sub}_{\text{Min}} \rightarrow \text{IV}_{\text{Min}}^m$

Simple, but enough for now, *and easy to extend*.

Now in Haskell—I

A naive datatype encoding musical harmony:

```
data Piece = Piece [Phrase]
```

```
data Phrase where
```

```
PhraseVI :: Ton → Dom → Ton → Phrase
```

```
PhraseVI :: Dom → Ton → Phrase
```

Now in Haskell—I

A naive datatype encoding musical harmony:

```
data Piece = Piece [Phrase]
```

```
data Phrase where
```

```
PhraseVI :: Ton → Dom → Ton → Phrase
```

```
PhraseVI :: Dom → Ton → Phrase
```

```
data Ton where
```

```
TonMaj :: Degree → Ton
```

```
TonMin :: Degree → Ton
```

Now in Haskell—I

A naive datatype encoding musical harmony:

```
data Piece = Piece [Phrase]
```

```
data Phrase where
```

```
PhraseVI :: Ton → Dom → Ton → Phrase
```

```
PhraseV  :: Dom → Ton → Phrase
```

```
data Ton where
```

```
TonMaj :: Degree → Ton
```

```
TonMin :: Degree → Ton
```

```
data Dom where
```

```
DomV7  :: Degree → Dom
```

```
DomV   :: Degree → Dom
```

```
DomVII0 :: Degree → Dom
```

```
DomIV-V :: SDom → Dom → Dom
```

```
DomII-V  :: Degree → Degree → Dom
```

Now in Haskell—I

A naive datatype encoding musical harmony:

```
data Piece = Piece [ Phrase ]
```

```
data Phrase where
```

```
PhraseVI :: Ton → Dom → Ton → Phrase
```

```
PhraseV  ::      Dom → Ton → Phrase
```

```
data Ton where
```

```
TonMaj :: Degree → Ton
```

```
TonMin :: Degree → Ton
```

```
data Dom where
```

```
DomV7  :: Degree → Dom
```

```
DomV   :: Degree → Dom
```

```
DomVII0 :: Degree → Dom
```

```
DomIV-V :: SDom → Dom → Dom
```

```
DomII-V  :: Degree → Degree → Dom
```

```
data Degree = I | II | III ...
```

Now in Haskell—II

A GADT encoding musical harmony:

```
data Mode = MajMode | MinMode
```

```
data Piece ( $\mu$  :: Mode) where
```

```
  Piece :: [Phrase  $\mu$ ]  $\rightarrow$  Piece  $\mu$ 
```

Now in Haskell—II

A GADT encoding musical harmony:

```
data Mode = MajMode | MinMode
```

```
data Piece ( $\mu :: \text{Mode}$ ) where
```

```
  Piece :: [Phrase  $\mu$ ]  $\rightarrow$  Piece  $\mu$ 
```

```
data Phrase ( $\mu :: \text{Mode}$ ) where
```

```
  PhraseIV :: Ton  $\mu \rightarrow$  Dom  $\mu \rightarrow$  Ton  $\mu \rightarrow$  Phrase  $\mu$ 
```

```
  PhraseVI :: Dom  $\mu \rightarrow$  Ton  $\mu \rightarrow$  Phrase  $\mu$ 
```

Now in Haskell—II

A GADT encoding musical harmony:

```
data Mode = MajMode | MinMode
```

```
data Piece ( $\mu :: \text{Mode}$ ) where
```

```
  Piece :: [Phrase  $\mu$ ]  $\rightarrow$  Piece  $\mu$ 
```

```
data Phrase ( $\mu :: \text{Mode}$ ) where
```

```
  PhraseIV :: Ton  $\mu$   $\rightarrow$  Dom  $\mu$   $\rightarrow$  Ton  $\mu$   $\rightarrow$  Phrase  $\mu$ 
```

```
  PhraseVI ::           Dom  $\mu$   $\rightarrow$  Ton  $\mu$   $\rightarrow$  Phrase  $\mu$ 
```

```
data Ton ( $\mu :: \text{Mode}$ ) where
```

```
  TonMaj :: SD I Maj  $\rightarrow$  Ton MajMode
```

```
  TonMin :: SD I Min  $\rightarrow$  Ton MinMode
```

Now in Haskell—II

A GADT encoding musical harmony:

```
data Mode = MajMode | MinMode
```

```
data Piece ( $\mu :: \text{Mode}$ ) where
```

```
  Piece :: [Phrase  $\mu$ ]  $\rightarrow$  Piece  $\mu$ 
```

```
data Phrase ( $\mu :: \text{Mode}$ ) where
```

```
  PhraseVI :: Ton  $\mu \rightarrow$  Dom  $\mu \rightarrow$  Ton  $\mu \rightarrow$  Phrase  $\mu$ 
```

```
  PhraseVI :: Dom  $\mu \rightarrow$  Ton  $\mu \rightarrow$  Phrase  $\mu$ 
```

```
data Ton ( $\mu :: \text{Mode}$ ) where
```

```
  TonMaj :: SD I Maj  $\rightarrow$  Ton MajMode
```

```
  TonMin :: SD I Min  $\rightarrow$  Ton MinMode
```

```
data Dom ( $\mu :: \text{Mode}$ ) where
```

```
  DomV7 :: SD V Dom7  $\rightarrow$  Dom  $\mu$ 
```

```
  DomV :: SD V Maj  $\rightarrow$  Dom  $\mu$ 
```

```
  DomVII0 :: SD VII Dim  $\rightarrow$  Dom  $\mu$ 
```

```
  DomIV-V :: SDom  $\mu \rightarrow$  Dom  $\mu \rightarrow$  Dom  $\mu$ 
```

```
  DomII-V :: SD II Dom7  $\rightarrow$  SD V Dom7  $\rightarrow$  Dom  $\mu$ 
```

Now in Haskell—III

Scale degrees are the leaves of our hierarchical structure:

```
data DiatonicDegree = I | II | III | IV | V | VI | VII
```

```
data Quality       = Maj | Min | Dom7 | Dim
```

```
data SD ( $\delta$  :: DiatonicDegree) ( $\gamma$  :: Quality) where  
  SurfaceChord :: ChordDegree  $\rightarrow$  SD  $\delta$   $\gamma$ 
```

Now in Haskell—III

Scale degrees are the leaves of our hierarchical structure:

```
data DiatonicDegree = I | II | III | IV | V | VI | VII
```

```
data Quality       = Maj | Min | Dom7 | Dim
```

```
data SD ( $\delta$  :: DiatonicDegree) ( $\gamma$  :: Quality) where  
  SurfaceChord :: ChordDegree  $\rightarrow$  SD  $\delta$   $\gamma$ 
```

Now everything is properly indexed, and our GADT is effectively constrained to allow only “harmonically valid” sequences!

Generating harmony

Now that we have a datatype representing harmony sequences, how do we generate a sequence of chords?

Generating harmony

Now that we have a datatype representing harmony sequences, how do we generate a sequence of chords?

QuickCheck! We give **Arbitrary** instances for each of the datatypes in our model.

Generating harmony

Now that we have a datatype representing harmony sequences, how do we generate a sequence of chords?

QuickCheck! We give **Arbitrary** instances for each of the datatypes in our model.

... but we don't want to do this by hand, for every datatype, and to have to adapt the instances every time we change the model... so we use *generic programming*:

Generating harmony

Now that we have a datatype representing harmony sequences, how do we generate a sequence of chords?

QuickCheck! We give **Arbitrary** instances for each of the datatypes in our model.

... but we don't want to do this by hand, for every datatype, and to have to adapt the instances every time we change the model... so we use *generic programming*:

$$\begin{aligned} \text{gen} &:: \forall \alpha. (\text{Representable } \alpha, \text{Generate } (\text{Rep } \alpha)) \\ &\Rightarrow \text{Gen } \alpha \end{aligned}$$

Generating harmony

Now that we have a datatype representing harmony sequences, how do we generate a sequence of chords?

QuickCheck! We give **Arbitrary** instances for each of the datatypes in our model.

... but we don't want to do this by hand, for every datatype, and to have to adapt the instances every time we change the model... so we use *generic programming*:

$$\begin{aligned} \text{gen} &:: \forall \alpha. (\text{Representable } \alpha, \text{Generate } (\text{Rep } \alpha)) \\ &\Rightarrow [(\text{String}, \text{Int})] \rightarrow \text{Gen } \alpha \end{aligned}$$

Examples of harmony generation

```
testGen :: Gen (Phrase MajMode)
testGen = gen [("Dom_IV-V", 3), ("Dom_II-V", 4)]
example :: IO ()
example = let k = Key (Note ♯ C) MajMode
          in sample' testGen >>= mapM_ (printOnKey k)
```

Examples of harmony generation

```
testGen :: Gen (Phrase MajMode)
testGen = gen [("Dom_IV-V", 3), ("Dom_II-V", 4)]
example :: IO ()
example = let k = Key (Note ♯ C) MajMode
          in sample' testGen >>= mapM_ (printOnKey k)
```

> example

```
[C: Maj, D: Dom7, G: Dom7, C: Maj]
[C: Maj, G: Dom7, C: Maj]
[C: Maj, E: Min, F: Maj, G: Maj, C: Maj]
[C: Maj, E: Min, F: Maj, D: Dom7, G: Dom7, C: Maj]
[C: Maj, D: Min, E: Min, F: Maj, D: Dom7, G: Dom7, C: Maj]
```

Generating a melody for a given harmony

We then generate a melody in 4 steps:

1. Generate a list of candidate melody notes per chord;
2. Refine the candidates by filtering out obviously bad candidates;
3. Pick one focal candidate melody note per chord;
4. Embellish the candidate notes to produce a final melody.

Generating a melody for a given harmony

We then generate a melody in 4 steps:

1. Generate a list of candidate melody notes per chord;
2. Refine the candidates by filtering out obviously bad candidates;
3. Pick one focal candidate melody note per chord;
4. Embellish the candidate notes to produce a final melody.

These four steps combine naturally using plain monadic bind:

```
melody :: Key → State MyState Song
melody k = genCandidates >>= refine >>= pickOne >>= embellish
         >>= return ◦ Song k
```


Table of Contents

Harmony

Haskell

Harmony analysis

Harmonic similarity

Music generation

Chord recognition: Chordify

Back to Chordify: chord recognition

Yet another practical application of a harmony model is to improve chord recognition from audio sources.

| | | | |
|------------------|---------|---------|---------|
| Chord candidates | 0.92 C | 0.96 Em | |
| | 0.94 Gm | 0.97 C | |
| | 1.00 C | 1.00 G | 1.00 Em |
| Beat number | 1 | 2 | 3 |

How to pick the right chord from the chord candidate list? Ask the harmony model which one fits best.

Chordify: architecture

- ▶ Frontend

- ▶ Reads user input, such as YouTube/Soundcloud/Deezer links, or files
- ▶ Extracts audio
- ▶ Calls the backend to obtain the chords for the audio
- ▶ Displays the result to the user
- ▶ Implements a queueing system, and library functionality
- ▶ Uses PHP, JavaScript, MongoDB

Chordify: architecture

- ▶ Frontend
 - ▶ Reads user input, such as YouTube/Soundcloud/Deezer links, or files
 - ▶ Extracts audio
 - ▶ Calls the backend to obtain the chords for the audio
 - ▶ Displays the result to the user
 - ▶ Implements a queueing system, and library functionality
 - ▶ Uses PHP, JavaScript, MongoDB
- ▶ Backend
 - ▶ Takes an audio file as input, analyses it, extracts the chords
 - ▶ The chord extraction code uses GADTs, type families, generic programming (see the HarmTrace package on Hackage)
 - ▶ Performs PDF and MIDI export (using LilyPond)
 - ▶ Uses Haskell, SoX, sonic annotator, and is mostly open source

Chordify: numbers

- ▶ Online since January 2013
- ▶ Top countries: US, UK, Germany, Indonesia, Canada
- ▶ Views: 3M+ (monthly)
- ▶ Chordified songs: 1.5M+
- ▶ Registered users: 200K

How do we handle these visitors?

- ▶ Single VPS, 6 Intel Xeon cores, 24GB RAM, 500GB SSD, 2TB hard drive

How do we handle these visitors?

- ▶ Single VPS, 6 Intel Xeon cores, 24GB RAM, 500GB SSD, 2TB hard drive
- ▶ Single server hosts both the web and database servers

How do we handle these visitors?

- ▶ Single VPS, 6 Intel Xeon cores, 24GB RAM, 500GB SSD, 2TB hard drive
- ▶ Single server hosts both the web and database servers
- ▶ Can easily handle peaks of (at least) 700 visitors at a time

How do we handle these visitors?

- ▶ Single VPS, 6 Intel Xeon cores, 24GB RAM, 500GB SSD, 2TB hard drive
- ▶ Single server hosts both the web and database servers
- ▶ Can easily handle peaks of (at least) 700 visitors at a time
- ▶ Chordifying new songs takes some computing power, but most songs are in the database already

How do we handle these visitors?

- ▶ Single VPS, 6 Intel Xeon cores, 24GB RAM, 500GB SSD, 2TB hard drive
- ▶ Single server hosts both the web and database servers
- ▶ Can easily handle peaks of (at least) 700 visitors at a time
- ▶ Chordifying new songs takes some computing power, but most songs are in the database already
- ▶ Queueing system for busy periods

How do we handle these visitors?

- ▶ Single VPS, 6 Intel Xeon cores, 24GB RAM, 500GB SSD, 2TB hard drive
- ▶ Single server hosts both the web and database servers
- ▶ Can easily handle peaks of (at least) 700 visitors at a time
- ▶ Chordifying new songs takes some computing power, but most songs are in the database already
- ▶ Queueing system for busy periods
- ▶ Infrastructure costs are minimal

Frontend (PHP/JS) and backend (Haskell) interaction

- ▶ Frontend receives a music file, calls backend with it

Frontend (PHP/JS) and backend (Haskell) interaction

- ▶ Frontend receives a music file, calls backend with it
- ▶ Backend computes the chords, writes them to a file:

Frontend (PHP/JS) and backend (Haskell) interaction

- ▶ Frontend receives a music file, calls backend with it
- ▶ Backend computes the chords, writes them to a file:
 - ▶ `1;D:min;0.232199546;0.615328798`
 - ▶ `2;D:min;0.615328798;0.998458049`
 - ▶ ...

Frontend (PHP/JS) and backend (Haskell) interaction

- ▶ Frontend receives a music file, calls backend with it
- ▶ Backend computes the chords, writes them to a file:
 - ▶ `1;D:min;0.232199546;0.615328798`
 - ▶ `2;D:min;0.615328798;0.998458049`
 - ▶ ...
- ▶ Frontend reads this file, updates the database if necessary, and renders the result

Frontend (PHP/JS) and backend (Haskell) interaction

- ▶ Frontend receives a music file, calls backend with it
- ▶ Backend computes the chords, writes them to a file:
 - ▶ `1;D:min;0.232199546;0.615328798`
 - ▶ `2;D:min;0.615328798;0.998458049`
 - ▶ ...
- ▶ Frontend reads this file, updates the database if necessary, and renders the result
- ▶ Backend is open-source (and GPL3); only option is to run it as a standalone executable

The importance of the UI

Let's have a look at four different online services giving you the chords for a song (Radiohead's Karma Police).

The importance of the UI—Riffstation

The screenshot displays the Riffstation application interface. At the top, there is a search bar with the text "Search Song or Artist..." and a "Search" button. To the right is a "Share" button with a share icon. Below the search bar is a video player showing a music video for "Radiohead - Karma Police" with the Vevo logo. The video player includes a progress bar and playback controls. To the right of the video player is a "Capo" button and a guitar fretboard diagram. The fretboard diagram shows a capo on the first fret and three yellow dots indicating finger positions on the second, third, and first strings. Below the fretboard is a row of buttons for guitar chords: Am, F#m, D, E, G, and an empty button. To the right of these is a button for the E chord. At the bottom, there are three buttons: "Play", "Step Back", and "Zoom out". On the far right, there are three circular icons: a guitar, a bass, and a keyboard.

The importance of the UI—Youtab.me

The screenshot displays the Youtab.me interface for the song "Karma Police" by Radiohead. The page features a search bar at the top with "karma police" entered. Below the search bar, there are navigation links for "Discover", "Add a song", "Login", and "Sign up". The main content area shows the song's title and artist, followed by a series of guitar chords: Am, D9/F#, Em, G, Am, Fadd9, Em, G, Am, D, G, D/F#, C/E, Em7/D, Am, Bm, D, Am, D9/F#, Em, G, Am, F, Em, G, Am, D. The lyrics are displayed below the chords, with a red vertical line indicating the current playback position. The lyrics include: "Arrest this man", "He talks in maths", "He buzzes like a fridge", and "He's like a deflated ro". There are buttons for "Has Lyrics", "Has Chords", and "Verified". A "Tips & Tricks" sidebar on the right provides instructions on how to use the player, including a "Fix" button. The bottom of the page shows a video player with the song title "Karma Police - Radiohead" and a progress bar.

The importance of the UI—Chordify

The screenshot displays the Chordify web application interface. At the top left is a video player showing a night scene of a road. To its right is a dark green panel with the text "Share your favourites via social media" and icons for a refresh symbol, a share icon, and a menu icon. Further right is a white panel with the Chordify logo and a description: "Chordify is a free online music service - made for and by music enthusiasts - that transforms any song into chords. New! Print chords as seen on screen." Below these panels is a control bar with buttons for "song", "chords", "transpose", "speed", "download", and "diagrams". The main area features a chord chart for "Radiohead - Karma Police - emfmusic". The chart is a grid with five rows of chords and six columns of measures. The chords are: Row 1: A, D, G, D, Em, Bm; Row 2: Am, Bm, D; Row 3: Am, F#m, G; Row 4: Am, F, G; Row 5: Am, D, G, C, Bm. A vertical sidebar on the left contains social media icons for Facebook, Twitter, Google+, LinkedIn, and Email.

Chordify

Chordify is a free online music service - made for and by music enthusiasts - that transforms any song into chords.

New! Print chords as seen on screen.

Share your favourites via social media

song | chords | transpose | speed | download | diagrams

Radiohead - Karma Police - emfmusic

| | | | | | |
|----|-----|----|---|----|----|
| A | D | G | D | Em | Bm |
| Am | | Bm | | D | |
| Am | F#m | Em | | G | |
| Am | F | E | | G | |
| Am | D | G | | C | Bm |

Logistics of an internet start-up

- ▶ Chordify is created and funded by 5 people

Logistics of an internet start-up

- ▶ Chordify is created and funded by 5 people
- ▶ If you can do without venture capital, do it!

Logistics of an internet start-up

- ▶ Chordify is created and funded by 5 people
- ▶ If you can do without venture capital, do it!
- ▶ You might end up doing more than just functional programming, though:
 - ▶ Deciding on what features to implement next
 - ▶ Recruiting, interviewing, dealing with legal issues related to employment
 - ▶ Taxation (complicated by the fact that we sell worldwide and support multiple currencies)
 - ▶ User support
 - ▶ Outreach (pitching events, media, this talk, etc.)

Logistics of an internet start-up

- ▶ Chordify is created and funded by 5 people
- ▶ If you can do without venture capital, do it!
- ▶ You might end up doing more than just functional programming, though:
 - ▶ Deciding on what features to implement next
 - ▶ Recruiting, interviewing, dealing with legal issues related to employment
 - ▶ Taxation (complicated by the fact that we sell worldwide and support multiple currencies)
 - ▶ User support
 - ▶ Outreach (pitching events, media, this talk, etc.)
- ▶ But it's fun, and you learn a lot!

Summary

Musical modelling with Haskell:

- ▶ A model for musical harmony as a Haskell datatype
- ▶ Makes use of several advanced functional programming techniques, such as generic programming, GADTs, and type families
- ▶ When chords do not fit the model: error correction
- ▶ Harmonising melodies
- ▶ Generating harmonies
- ▶ Recognising harmony from audio sources
- ▶ Transporting academic research into industry

Play with it!

chordify[®]

`http://chordify.net`

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

`http://hackage.haskell.org/package/FComp`