Advanced Functional Programming in Industry

José Pedro Magalhães

chordfy®

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:

http://chordify.net
Table of Contents

Harmony

Haskell

Harmony analysis

Harmonic similarity

Music generation

Chord recognition: Chordify
What is harmony?

- 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?

- 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♭, 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 V7 I.
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
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.
\begin{verbatim}
data Root  = A \mid B \mid C \mid D \mid E \mid F \mid G
type Octave = Int
data Note  = Note Root Octave
\end{verbatim}
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

\[
cMajScale = [c_4, d_4, e_4, f_4, g_4, a_4, b_4]
\]
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 :: [α] → [α]
reverse [] = []
reverse (h : t) = reverse t ++ [h]

(++): [α] → [α] → [α]
(++)=...
Transposing a melody one octave higher:

\[
\text{octaveUp} :: \text{Octave} \rightarrow \text{Octave}
\]
\[
\text{octaveUp} \ n = n + 1
\]

\[
\text{noteOctaveUp} :: \text{Note} \rightarrow \text{Note}
\]
\[
\text{noteOctaveUp} \ (\text{Note} \ r \ o) = \text{Note} \ r \ (\text{octaveUp} \ o)
\]

\[
\text{melodyOctaveUp} :: \text{Melody} \rightarrow \text{Melody}
\]
\[
\text{melodyOctaveUp} \ m = \text{map} \ \text{noteOctaveUp} \ m
\]
Generation, analysis

Building a canon from a melody:

```haskell
    canon :: Melody → Melody
    canon m = m ++ canon m
```
Building a canon from a melody:

\[ \text{canon} :: \text{Melody} \rightarrow \text{Melody} \]
\[ \text{canon } m = m \oplus \text{canon } m \]

Is a given melody in C major?

\[ \text{root} :: \text{Note} \rightarrow \text{Root} \]
\[ \text{root (Note } r \circ \text{)} = r \]
\[ \text{isCMaj} :: \text{Melody} \rightarrow \text{Bool} \]
\[ \text{isCMaj } = \text{all } (\in \text{cMajScale}) \circ \text{map } \text{root} \]
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:

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

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

\[ \text{Piece}_M \rightarrow [\text{Phrase}_M] \quad (M \in \{\text{Maj, Min}\}) \]
A functional model of harmony

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

\[ \text{Phrase}_M \rightarrow \text{Ton}_M \text{ Dom}_M \text{ Ton}_M \]
\[ \quad \text{|} \quad \text{Dom}_M \text{ Ton}_M \]

Simple, but enough for now, and easy to extend.
A functional model of harmony

\[
\begin{align*}
\text{Piece}_M & \rightarrow [\text{Phrase}_M] \quad (M \in \{\text{Maj}, \text{Min}\}) \\
\text{Phrase}_M & \rightarrow \text{Ton}_M \text{ Dom}_M \text{ Ton}_M \\
& \quad | \\
& \quad \text{Dom}_M \text{ Ton}_M \\
\text{Ton}_{\text{Maj}} & \rightarrow I_{\text{Maj}} \\
\text{Ton}_{\text{Min}} & \rightarrow I_{\text{Min}}
\end{align*}
\]
A functional model of harmony

\[
\begin{align*}
\text{Piece}_M & \rightarrow [\text{Phrase}_M] \quad (M \in \{\text{Maj}, \text{Min}\}) \\
\text{Phrase}_M & \rightarrow \text{Ton}_M \ \text{Dom}_M \ \text{Ton}_M \\
& \quad | \quad \text{Dom}_M \ \text{Ton}_M \\
\text{Ton}_\text{Maj} & \rightarrow I_{\text{Maj}} \\
\text{Ton}_\text{Min} & \rightarrow I^m_{\text{Min}} \\
\text{Dom}_M & \rightarrow V^7_M \\
& \quad | \quad V_M \\
& \quad | \quad VII^0_M \\
& \quad | \quad \text{Sub}_M \ \text{Dom}_M \\
& \quad | \quad II^7_M \ V^7_M
\end{align*}
\]
A functional model of harmony

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

\[ \text{Phrase}_M \rightarrow \text{Ton}_M \text{ Dom}_M \text{ Ton}_M \]
\[ \quad | \quad \text{Dom}_M \text{ Ton}_M \]

\[ \text{Ton}_\text{Maj} \rightarrow I^\text{Maj}_M \]
\[ \text{Ton}_\text{Min} \rightarrow I^m_\text{Min}_M \]

\[ \text{Dom}_M \rightarrow V^7_M \]
\[ \quad | \quad V^m_M \]
\[ \quad | \quad VII^0_M \]
\[ \quad | \quad \text{Sub}_M \text{ Dom}_M \]
\[ \quad | \quad II^7_M \quad V^7_M \]

\[ \text{Sub}_\text{Maj} \rightarrow II^m_\text{Maj}_M \]
\[ \quad | \quad IV^m_\text{Maj}_M \]
\[ \quad | \quad III^m_\text{Maj}_M \quad IV^m_\text{Maj}_M \]

\[ \text{Sub}_\text{Min} \rightarrow IV^m_\text{Min}_M \]
A functional model of harmony

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

\[
\text{Phrase}_M \rightarrow \text{Ton}_M \text{ Dom}_M \text{ Ton}_M \\
| \text{ Dom}_M \text{ Ton}_M
\]

\[
\text{Ton}_{\text{Maj}} \rightarrow I_{\text{Maj}} \\
\text{Ton}_{\text{Min}} \rightarrow I_{m_{\text{Min}}}
\]

\[
\text{Dom}_M \rightarrow V_7^M \\
| V_3^M \\
| VI_7^0 \\
| \text{Sub}_M \text{ Dom}_M \\
| II_7^M \text{ V}_7^M
\]

\[
\text{Sub}_{\text{Maj}} \rightarrow II_{m_{\text{Maj}}} \\
| IV_{\text{Maj}} \\
| III_{m_{\text{Maj}}} IV_{\text{Maj}} \\
\text{Sub}_{\text{Min}} \rightarrow IV_{m_{\text{Min}}}
\]

Simple, but enough for now, and easy to extend.
Now in Haskell—I

A naive datatype encoding musical harmony:

```haskell
data Piece = Piece [Phrase]
data Phrase where
  PhraseIVI :: Ton → Dom → Ton → Phrase
  PhraseVI :: Dom → Ton → Phrase
```
Now in Haskell—I

A naive datatype encoding musical harmony:

```haskell
data Piece = Piece [Phrase]

data Phrase where
    PhraseIVI :: Ton → Dom → Ton → Phrase
    PhraseVI :: Dom → Ton → Phrase

data Ton where
    TonMaj :: Degree → Ton
    TonMin :: Degree → Ton
```
A naive datatype encoding musical harmony:

```haskell
data Piece = Piece [Phrase]

data Phrase where
    Phrase_IVI :: Ton -> Dom -> Ton -> Phrase
    Phrase_VI :: Dom -> Ton -> Phrase

data Ton where
    Ton_Maj :: Degree -> Ton
    Ton_Min :: Degree -> Ton

data Dom where
    Dom_V7 :: Degree -> Dom
    Dom_V :: Degree -> Dom
    Dom_VII0 :: Degree -> Dom
    Dom_IV_V :: SDom -> Dom -> Dom
    Dom_II_V :: Degree -> Degree -> Dom
```
A naive datatype encoding musical harmony:

```haskell
data Piece = Piece [Phrase]
data Phrase where
  Phrase_{IVI} :: Ton → Dom → Ton → Phrase
  Phrase_{VI} :: Dom → Ton → Phrase

data Ton where
  Ton_{Maj} :: Degree → Ton
  Ton_{Min} :: Degree → Ton

data Dom where
  Dom_{V^7} :: Degree → Dom
  Dom_{V} :: Degree → Dom
  Dom_{VII^0} :: Degree → Dom
  Dom_{IV−V} :: SDom → Dom → Dom
  Dom_{II−V} :: Degree → Degree → Dom

data Degree = I | II | III . . .
```
Now in Haskell—II

A GADT encoding musical harmony:

```haskell
data Mode = MajMode | MinMode

data Piece (\mu :: Mode) where
  Piece :: [Phrase \mu] \rightarrow Piece \mu
```
Now in Haskell—II

A GADT encoding musical harmony:

```haskell
data Mode = MajMode | MinMode

data Piece (µ :: Mode) where
  Piece :: [Phrase µ] → Piece µ

data Phrase (µ :: Mode) where
  PhraseIVI :: Ton µ → Dom µ → Ton µ → Phrase µ
  PhraseVI :: Dom µ → Ton µ → Phrase µ
```
A GADT encoding musical harmony:

```haskell
data Mode = Maj \text{Mode} \mid \text{Min} \text{Mode}

data Piece (\mu :: \text{Mode}) where
Piece :: [\text{Phrase} \mu] \rightarrow \text{Piece} \mu

data Phrase (\mu :: \text{Mode}) where
\text{Phrase}_{IVI} :: \text{Ton} \mu \rightarrow \text{Dom} \mu \rightarrow \text{Ton} \mu \rightarrow \text{Phrase} \mu
\text{Phrase}_{VI} :: \text{Dom} \mu \rightarrow \text{Ton} \mu \rightarrow \text{Phrase} \mu

data Ton (\mu :: \text{Mode}) where
\text{Ton}_{\text{Maj}} :: \text{SD I Maj} \rightarrow \text{Ton} \text{Maj} \text{Mode}
\text{Ton}_{\text{Min}} :: \text{SD I Min} \rightarrow \text{Ton} \text{Min} \text{Mode}
```
Now in Haskell—II

A GADT encoding musical harmony:

```haskell
data Mode = MajMode | MinMode

data Piece (\mu :: Mode) where
    Piece :: [Phrase \mu] \rightarrow Piece \mu

data Phrase (\mu :: Mode) where
    Phrase_{IVI} :: Ton \mu \rightarrow Dom \mu \rightarrow Ton \mu \rightarrow Phrase \mu
    Phrase_{VI} :: Dom \mu \rightarrow Ton \mu \rightarrow Phrase \mu

data Ton (\mu :: Mode) where
    Ton_{Maj} :: SD I Maj \rightarrow Ton MajMode
    Ton_{Min} :: SD I Min \rightarrow Ton MinMode

data Dom (\mu :: Mode) where
    Dom_{V^7} :: SD V Dom^7 \rightarrow Dom \mu
    Dom_{V} :: SD V Maj \rightarrow Dom \mu
    Dom_{VII^0} :: SD VII Dim \rightarrow Dom \mu
    Dom_{IV-V} :: SDom \mu \rightarrow Dom \mu \rightarrow Dom \mu
    Dom_{II-V} :: SD II Dom^7 \rightarrow SD V Dom^7 \rightarrow Dom \mu
```
Now in Haskell—III

Scale degrees are the leaves of our hierarchical structure:

```haskell
data DiatonicDegree = I | II | III | IV | V | VI | VII
data Quality = Maj | Min | Dom⁷ | Dim
data SD (δ :: DiatonicDegree) (γ :: Quality) where
  SurfaceChord :: ChordDegree → SD δ γ
```
Now in Haskell—III

Scale degrees are the leaves of our hierarchical structure:

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

data Quality = Maj | Min | Dom⁷ | Dim

data SD (δ :: DiatonicDegree) (γ :: Quality) where
  SurfaceChord :: ChordDegree → SD δ γ
```

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 \textit{Arbitrary} instances for each of the datatypes in our model.

\ldots 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\ldots so we use \textit{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:

\[
\text{gen} :: \forall \alpha. (\text{Representable } \alpha, \text{Generate} (\text{Rep } \alpha)) \Rightarrow \text{Gen } \alpha
\]
Generating harmony

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

QuickCheck! We give \textit{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 \textit{generic programming}:

\[
\text{gen} :: \forall \alpha. (\text{Representable } \alpha, \text{Generate } (\text{Rep } \alpha)) \\
\quad \Rightarrow [(\text{String,Int})] \rightarrow \text{Gen } \alpha
\]
Examples of harmony generation

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

testGen :: Gen (Phrase Maj\textsubscript{Mode})
testGen = gen [('Dom\textsubscript{IV}-V'', 3), ('Dom\textsubscript{II}-V'', 4)]

eexample :: IO ()
eexample = let k = Key (Note ♯ C) Maj\textsubscript{Mode}
in sample' testGen >>= mapM_ (printOnKey k)

> example
[ C: Maj, D: Dom\textsuperscript{7}, G: Dom\textsuperscript{7}, C: Maj]
[ C: Maj, G: Dom\textsuperscript{7}, C: Maj]
[ C: Maj, E: Min, F: Maj, G: Maj, C: Maj]
[ C: Maj, E: Min, F: Maj, D: Dom\textsuperscript{7}, G: Dom\textsuperscript{7}, C: Maj]
[ C: Maj, D: Min, E: Min, F: Maj, D: Dom\textsuperscript{7}, G: Dom\textsuperscript{7}, 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:

\[
\text{melody} :: \text{Key} \rightarrow \text{State MyState Song}
\]
\[
\begin{align*}
\text{melody} \ k &= \ \text{genCandidates} \gg \text{refine} \gg \text{pickOne} \gg \text{embellish} \\
&\gg \text{return} \circ \text{Song} \ k
\end{align*}
\]
Example I

Phrase

Ton

I: Maj

C: Maj

III: Min

IV: Maj

II: Dom\(^7\)

V: Dom\(^7\)

C: Maj

Dom

Sub

Dom

Ton

I: Maj

C: Maj

E: Min

F: Maj

D: Dom\(^7\)

G: Dom\(^7\)
Example II

\begin{music}
\begin{musicstaff}
\begin{musicnote}[\textbf{\#}]{\textbf{\#}}
\end{musicnote}
\begin{musicnote}[\textbf{\#}]\textbf{\#}\end{musicnote}
\end{musicstaff}
\end{music}

Phrase

\begin{music}
\begin{musicstaff}
\begin{musicnote}[\textbf{\#}]{\textbf{\#}}
\end{musicnote}
\begin{musicnote}[\textbf{\#}]\textbf{\#}\end{musicnote}
\end{musicstaff}
\end{music}

Ton

I: Min

E: Min

IV: Min

A: Min

Sub

Dom

Dom

IV: Min

II: Dom^7

A: Min

F\#: Dom^7

V: Dom^7

B: Dom^7

Ton

I: Min

E: Min

José Pedro Magalhães

Advanced Functional Programming in Industry, FP Days 2014
Table of Contents

Harmony

Haskell

Harmony analysis

Harmonic similarity

Music generation

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

<table>
<thead>
<tr>
<th>Chord candidates</th>
<th>0.92 C</th>
<th>0.96 Em</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.94 Gm</td>
<td>0.97 C</td>
</tr>
<tr>
<td></td>
<td>1.00 C</td>
<td>1.00 G</td>
</tr>
<tr>
<td>Beat number</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

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

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
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
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
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
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—Chordie

Karma Police

Radiohead

Simple tab for Karma Police. Just put the parts together and keep the same rhythm and it sounds deed on.

```
| E | 1-0-0-0-0-0-0-0-0-0 |
| G | 1-0-1-0-0-0-0-0-0-0 |
| D | 0-0-0-0-0-0-0-0-0-0 |
| A | 0-0-0-0-0-0-0-0-0-0 |
| E | 0-0-0-0-0-0-0-0-0-0 |
```

This is what you get...

```
| E | 1-0-1-0-0-0-0-0-0-0 |
| G | 1-0-0-0-0-0-0-0-0-0 |
| D | 0-0-0-0-0-0-0-0-0-0 |
| A | 0-0-0-0-0-0-0-0-0-0 |
| E | 0-0-0-0-0-0-0-0-0-0 |
```

When you mess with us...

```
| E | 0-0-0-0-0-0-0-0-0-0 |
| G | 0-0-0-0-0-0-0-0-0-0 |
| D | 0-0-0-0-0-0-0-0-0-0 |
| A | 0-0-0-0-0-0-0-0-0-0 |
| E | 0-0-0-0-0-0-0-0-0-0 |
```

Phew, for a minute...
The importance of the UI—Riffstation
The importance of the UI—Youtab.me
The importance of the UI—Chordify
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!

http://chordify.net
http://hackage.haskell.org/package/HarmTrace
http://hackage.haskell.org/package/FComp