

# Functional Modelling of Musical Harmony And Its Applications

José Pedro Magalhães

Department of Computer Science, University of Oxford

December 9, 2014  
72nd IFIP 2.1 Meeting  
Vermont, United States of America

# Introduction

- ▶ Quick introduction to musical harmony
- ▶ Modelling musical harmony using Haskell
- ▶ Applications of a model of harmony:
  - ▶ Musical analysis
  - ▶ Musical similarity
  - ▶ Generating chords and melodies
  - ▶ Correcting errors in chord extraction
  - ▶ Chordify—a web-based music player with chord recognition
  - ▶ Merging chord edits—another application?

# Table of Contents

## Harmony

A functional model of harmony

Harmony analysis

Harmonic similarity

Music generation

Chord recognition: Chordify

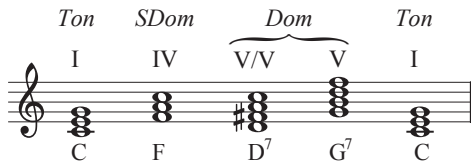
Crowd-sourcing chord edits

# What is harmony?

The image shows a musical staff with five chords. Above the staff, functional labels are placed: *Ton* above *I*, *SDom* above *IV*, *Dom* above *V/V* and *V*, and *Ton* above *I*. A bracket groups *V/V* and *V* under the label *Dom*. Below the staff, the chord names are written: C, F, D<sup>7</sup>, G<sup>7</sup>, 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
- ▶ It's the “vertical” aspect of music (with melody being the “horizontal” aspect)
- ▶ Harmony induces tension and release patterns, that can be described by music theory and music cognition
- ▶ The surrounding context also has a large influence
- ▶ It is generally highly structured, and obeys musical composition rules

# What is harmony?



- ▶ Harmony arises when at least two notes sound at the same time
- ▶ It's the “vertical” aspect of music (with melody being the “horizontal” aspect)
- ▶ Harmony induces tension and release patterns, that can be described by music theory and music cognition
- ▶ The surrounding context also has a large influence
- ▶ It is generally highly structured, and obeys musical composition rules

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.

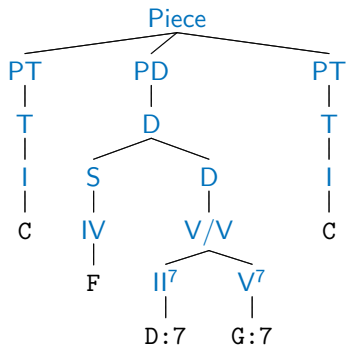
# Simplified harmony theory II

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,  $\text{II}^7$  is the secondary dominant of V, and  $\text{I}^7$  is the secondary dominant of IV.
- ▶ So you can start with I, add one note to get  $\text{I}^7$ , fall into IV, change two notes to get to  $\text{II}^7$ , fall into V, and then finally back to I.

# An example harmonic analysis

A musical staff in treble clef showing five chords. Above the staff, functional labels are placed: *Ton* above the first and fifth chords, *SDom* above the second, and *Dom* above the third and fourth, with a bracket spanning both. Below the staff, specific chord symbols are written: C, F, D<sup>7</sup>, G<sup>7</sup>, and C. The chords are represented by their constituent notes on the staff lines.





# 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 very useful, as we will see.

# Table of Contents

Harmony

A functional model of harmony

Harmony analysis

Harmonic similarity

Music generation

Chord recognition: Chordify

Crowd-sourcing chord edits

# 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

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

$\text{Phrase}_{\mathfrak{M}} \rightarrow \begin{array}{c} \text{Ton}_{\mathfrak{M}} \text{ Dom}_{\mathfrak{M}} \text{ Ton}_{\mathfrak{M}} \\ | \qquad \text{Dom}_{\mathfrak{M}} \text{ Ton}_{\mathfrak{M}} \end{array}$

# 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

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

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

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

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

$Dom_{\mathfrak{M}} \rightarrow V_{\mathfrak{M}}^7$

|  $V_{\mathfrak{M}}$

|  $VII_{\mathfrak{M}}^0$

|  $Sub_{\mathfrak{M}} \quad Dom_{\mathfrak{M}}$

|  $II_{\mathfrak{M}}^7 \quad V_{\mathfrak{M}}^7$

# A functional model of harmony

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

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

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

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

$Dom_{\mathfrak{M}} \rightarrow \begin{array}{c} V_{\mathfrak{M}}^7 \\ | \quad V_{\mathfrak{M}} \\ | \quad VII_{\mathfrak{M}}^0 \\ | \quad Sub_{\mathfrak{M}} \quad Dom_{\mathfrak{M}} \\ | \quad II_{\mathfrak{M}}^7 \quad V_{\mathfrak{M}}^7 \end{array}$

$Sub_{Maj} \rightarrow \begin{array}{c} II_{Maj}^m \\ | \quad IV_{Maj} \\ | \quad III_{Maj}^m \quad IV_{Maj} \end{array}$

$Sub_{Min} \rightarrow IV_{Min}^m$

# A functional model of harmony

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

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

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

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

$Dom_{\mathfrak{M}} \rightarrow \begin{array}{c} V_{\mathfrak{M}}^7 \\ | \quad V_{\mathfrak{M}} \\ | \quad VII_{\mathfrak{M}}^0 \\ | \quad Sub_{\mathfrak{M}} \quad Dom_{\mathfrak{M}} \\ | \quad II_{\mathfrak{M}}^7 \quad V_{\mathfrak{M}}^7 \end{array}$

$Sub_{Maj} \rightarrow \begin{array}{c} II_{Maj}^m \\ | \quad IV_{Maj} \\ | \quad III_{Maj}^m \quad IV_{Maj} \\ Sub_{Min} \rightarrow IV_{Min}^m \end{array}$

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—

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
```

```
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  $\mu$ 
```

Advanced functional programming begins here; we're using datatype promotion to constrain the shape of our indices.

## Now in Haskell—III

The rest of the model mimicks the context-free grammar shown before:

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

Phrase<sub>IV</sub> :: Ton  $\mu \rightarrow$  Dom  $\mu \rightarrow$  Ton  $\mu \rightarrow$  Phrase  $\mu$

Phrase<sub>VI</sub> :: Dom  $\mu \rightarrow$  Ton  $\mu \rightarrow$  Phrase  $\mu$

## Now in Haskell—III

The rest of the model mimicks the context-free grammar shown before:

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

$\text{Phrase}_{|V|} :: \text{Ton } \mu \rightarrow \text{Dom } \mu \rightarrow \text{Ton } \mu \rightarrow \text{Phrase } \mu$

$\text{Phrase}_{|V|} :: \text{Dom } \mu \rightarrow \text{Ton } \mu \rightarrow \text{Phrase } \mu$

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

$\text{Ton}_{\text{Maj}} :: \text{SD I Maj} \rightarrow \text{Ton Maj}_{\text{Mode}}$

$\text{Ton}_{\text{Min}} :: \text{SD I Min} \rightarrow \text{Ton Min}_{\text{Mode}}$

## Now in Haskell—III

The rest of the model mimicks the context-free grammar shown before:

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

$\text{Phrase}_{\text{VI}} :: \text{Ton } \mu \rightarrow \text{Dom } \mu \rightarrow \text{Ton } \mu \rightarrow \text{Phrase } \mu$

$\text{Phrase}_{\text{VI}} :: \text{Dom } \mu \rightarrow \text{Ton } \mu \rightarrow \text{Phrase } \mu$

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

$\text{Ton}_{\text{Maj}} :: \text{SD I Maj} \rightarrow \text{Ton Maj}_{\text{Mode}}$

$\text{Ton}_{\text{Min}} :: \text{SD I Min} \rightarrow \text{Ton Min}_{\text{Mode}}$

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

$\text{Dom}_{\text{V}^7} :: \text{SD V Dom}^7 \rightarrow \text{Dom } \mu$

$\text{Dom}_{\text{V}} :: \text{SD V Maj} \rightarrow \text{Dom } \mu$

$\text{Dom}_{\text{VII}^0} :: \text{SD VII Dim} \rightarrow \text{Dom } \mu$

$\text{Dom}_{\text{IV-V}} :: \text{SDom } \mu \rightarrow \text{Dom } \mu \rightarrow \text{Dom } \mu$

$\text{Dom}_{\text{II-V}} :: \text{SD II Dom}^7 \rightarrow \text{SD V Dom}^7 \rightarrow \text{Dom } \mu$



## Now in Haskell—IV

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—IV

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!

## Now in Haskell—V

We don't export `SurfaceChord`; these are built with a smart constructor:

```
deg :: (ToDegree  $\delta$ , ToQuality  $\gamma$ )  $\Rightarrow$  Proxy  $\delta$   $\rightarrow$  Proxy  $\gamma$   $\rightarrow$  SD  $\delta$   $\gamma$ 
deg d q = SurfaceChord (toDegree d) (toQuality q)
```

We need type-to-value conversions for degrees:

```
class ToDegree  $\delta$  where toDegree :: Proxy  $\delta$   $\rightarrow$  Degree
instance ToDegree I where toDegree _ = I
instance ToDegree II where toDegree _ = II
... -- instances for all degrees
```

(Type-to-value conversion actually handled automatically if we use the `singletons` package.)

## Now in Haskell—VI

Now terms like `TonMaj (deg (Proxy :: Proxy II) (Proxy :: Proxy Maj))` do not typecheck, as we wanted:

```
ghci> Ton_Maj (deg (Proxy :: Proxy II) (Proxy :: Proxy Maj))
      Couldn't match expected type 'I'
          with actual type 'II'
```

# Increasing complexity

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

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

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

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

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

Tritone substitutions  $\underline{A^b7} G^7 C_{Maj}$

...

# Handling secondary dominants

More fun: type families for computing perfect fifths:

```
data BDegSD ( $\delta$  :: DiatonicDegree) where  
  Cons :: BDegSD (PerfV  $\delta$ )  $\rightarrow$  Degree  $\delta$   $\rightarrow$  BDegSD  $\delta$   
  Base :: Degree  $\delta$   $\rightarrow$  BDegSD  $\delta$   
  
type family PerfV ( $\delta$  :: DiatonicDegree) :: DiatonicDegree  
type instance PerfV I = V  
type instance PerfV V = II  
...
```

# Handling secondary dominants

More fun: type families for computing perfect fifths:

```
data BDegSD ( $\delta$  :: DiatonicDegree) where  
  Cons :: BDegSD (PerfV  $\delta$ )  $\rightarrow$  Degree  $\delta$   $\rightarrow$  BDegSD  $\delta$   
  Base :: Degree  $\delta$   $\rightarrow$  BDegSD  $\delta$   
  
type family PerfV ( $\delta$  :: DiatonicDegree) :: DiatonicDegree  
type instance PerfV I = V  
type instance PerfV V = II  
...
```

I could go on for a while, but let's move on to actual applications of this model.

# Table of Contents

Harmony

A functional model of harmony

**Harmony analysis**

Harmonic similarity

Music generation

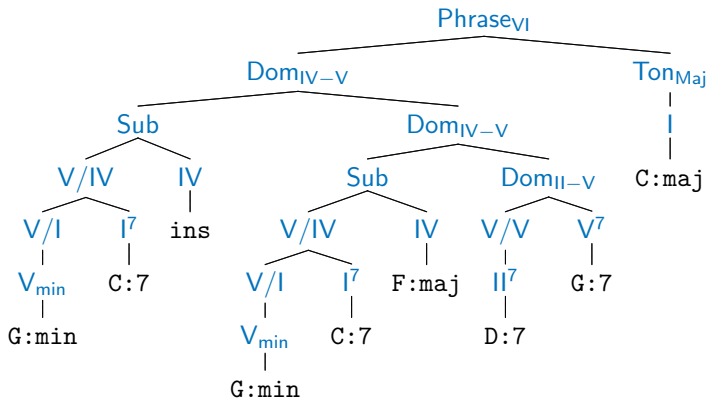
Chord recognition: Chordify

Crowd-sourcing chord edits



# Application: displaying harmony structurally

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



# Parsing chord sequences I

We now consider the problem of parsing (textual) chord sequences into our datatype representing musical harmony:

```
class Parse  $\alpha$  where  
  parse :: Parser  $\alpha$ 
```

# Parsing chord sequences I

We now consider the problem of parsing (textual) chord sequences into our datatype representing musical harmony:

```
class Parse  $\alpha$  where  
  parse :: Parser  $\alpha$ 
```

We use Doaitse's amazing error-correcting parser combinators, and we get harmony-compliant chord sequences for free!

# Parsing chord sequences II

Most instances are trivial:

**instance** Parse (Piece  $\mu$ ) **where**  
parse = Piece  $\langle \$ \rangle$  parse

**instance** Parse (Phrase  $\mu$ ) **where**  
parse = Phrase<sub>VI</sub>  $\langle \$ \rangle$  parse  
 $\langle | \rangle$  Phrase<sub>VI</sub>  $\langle \$ \rangle$  parse

**instance** Parse (Ton Maj<sub>Mode</sub>) **where**  
parse = Ton<sub>Maj</sub>  $\langle \$ \rangle$  parse

**instance** Parse (Ton Min<sub>Mode</sub>) **where**  
parse = Ton<sub>Min</sub>  $\langle \$ \rangle$  parse

...

# Parsing chord sequences II

Most instances are trivial:

**instance** Parse (Piece  $\mu$ ) **where**  
parse = Piece  $\langle \$ \rangle$  parse

**instance** Parse (Phrase  $\mu$ ) **where**  
parse = Phrase<sub>VI</sub>  $\langle \$ \rangle$  parse  
 $\langle | \rangle$  Phrase<sub>VI</sub>  $\langle \$ \rangle$  parse

**instance** Parse (Ton Maj<sub>Mode</sub>) **where**  
parse = Ton<sub>Maj</sub>  $\langle \$ \rangle$  parse

**instance** Parse (Ton Min<sub>Mode</sub>) **where**  
parse = Ton<sub>Min</sub>  $\langle \$ \rangle$  parse

...

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

# Parsing chord sequences III

Two challenges to triviality:

1. How to do generic programming with indexed datatypes

# Parsing chord sequences III

Two challenges to triviality:

1. How to do generic programming with indexed datatypes
  - ▶ Current libraries can't do it (Uniplate, SYB, `GHC.Generics`, etc.)
  - ▶ So we came up with one

# Parsing chord sequences III

Two challenges to triviality:

1. How to do generic programming with indexed datatypes
  - ▶ Current libraries can't do it (Uniplate, SYB, `GHC.Generics`, etc.)
  - ▶ So we came up with one
  - ▶ ... which kind of works.



# Parsing chord sequences III

Two challenges to triviality:

1. How to do generic programming with indexed datatypes
  - ▶ Current libraries can't do it (Uniplate, SYB, `GHC.Generics`, etc.)
  - ▶ So we came up with one
  - ▶ ... which kind of works.
2. How to instantiate (generic) functions to indexed datatypes
  - ▶ Consumers (like `show`) are easy
  - ▶ Producers (like `read`, or our `parse`) are trickier!

## Diversion 1: GP for indexed datatypes I

Indexed datatypes are actually “just” existential quantification together with equality constraints:

```
data Nat = Ze | Su Nat
data Vec ( $\eta :: \text{Nat}$ ) ( $\alpha :: \star$ ) where
  Nil  :: Vec Ze  $\alpha$ 
  Cons ::  $\alpha \rightarrow \text{Vec } \eta \alpha \rightarrow \text{Vec } (\text{Su } \eta) \alpha$ 
```

This can also be written as such:

```
data Vec ( $\eta :: \text{Nat}$ ) ( $\alpha :: \star$ ) =
   $\eta \sim \text{Ze} \Rightarrow \text{Nil}$ 
  |  $\forall \mu :: \text{Nat}. \eta \sim \text{Su } \mu \Rightarrow \text{Cons } \alpha (\text{Vec } \mu \alpha)$ 
```

## Diversion 1: GP for indexed datatypes II

Handling equality constraints generically is easy: we just add a type equality operator to the generic representation.

Existential quantification is much harder. My current solution uses skolemization; alternatives are welcome!

```
type instance Rep (Vec η α) = -- can't place a forall here!  
    (η ~: Ze)      :x: U  
    :+: (η ~: Su (X η)) :x: α :x: Vec (X η) α
```

```
type family X (σ :: κ)  
type instance X (Su σ) = σ
```

## Diversion 2: instantiating functions to indexed datatypes I

Indexed datatypes need one instance per type at which any of their constructors is indexed at, and possibly one more in case some constructors do not constrain the index. E.g.:

```
data Index = I1 | I2
```

```
data Data (ι :: Index) where
```

```
  D1 :: Data I1
```

```
  D2 :: Data I2
```

```
  D3 :: Data ι
```

```
instance Read (Data I1) where
```

```
  read "D1" = D1
```

```
instance Read (Data I2) where
```

```
  read "D2" = D2
```

```
instance Read (Data ι) where
```

```
  read "D3" = D3
```

## Diversion 2: instantiating functions to indexed datatypes I

Indexed datatypes need one instance per type at which any of their constructors is indexed at, and possibly one more in case some constructors do not constrain the index. E.g.:

```
data Index = I1 | I2
```

```
data Data (ι :: Index) where
```

```
  D1 :: Data I1
```

```
  D2 :: Data I2
```

```
  D3 :: Data ι
```

```
instance Read (Data I1) where
```

```
  read "D1" = D1
```

```
  read "D3" = D3  -- don't forget these!
```

```
instance Read (Data I2) where
```

```
  read "D2" = D2
```

```
  read "D3" = D3  -- don't forget these!
```

```
instance Read (Data ι) where
```

```
  read "D3" = D3
```

## Diversion 2: instantiating functions to indexed datatypes II

I think we should:

1. Add support for indexed datatypes in `GHC.Generics`;

## Diversion 2: instantiating functions to indexed datatypes II

I think we should:

1. Add support for indexed datatypes in `GHC.Generics`;
2. Let users use **deriving** for any class which has a generic implementation;

## Diversion 2: instantiating functions to indexed datatypes II

I think we should:

1. Add support for indexed datatypes in `GHC.Generics`;
2. Let users use **deriving** for any class which has a generic implementation;
3. Change the semantics of **deriving** for indexed datatypes (the current semantics is pretty much undefined or failure right now, anyway).



## Diversion 2: instantiating functions to indexed datatypes II

I think we should:

1. Add support for indexed datatypes in `GHC.Generics`;
2. Let users use **deriving** for any class which has a generic implementation;
3. Change the semantics of **deriving** for indexed datatypes (the current semantics is pretty much undefined or failure right now, anyway).

Then we'll finally be able to write:

```
data Vec ( $\eta :: \text{Nat}$ ) ( $\alpha :: \star$ ) where  
  Nil  :: Vec Ze  $\alpha$   
  Cons ::  $\alpha \rightarrow \text{Vec } \eta \alpha \rightarrow \text{Vec } (\text{Su } \eta) \alpha$   
      deriving (Show, Read)
```

(And I'll be able to do the same for all of my harmony model, too!)

# Table of Contents

Harmony

A functional model of harmony

Harmony analysis

**Harmonic similarity**

Music generation

Chord recognition: Chordify

Crowd-sourcing chord edits

## 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
- ▶ Once again, we're giving a new meaning to programming language techniques by applying them to musical harmony!

# Table of Contents

Harmony

A functional model of harmony

Harmony analysis

Harmonic similarity

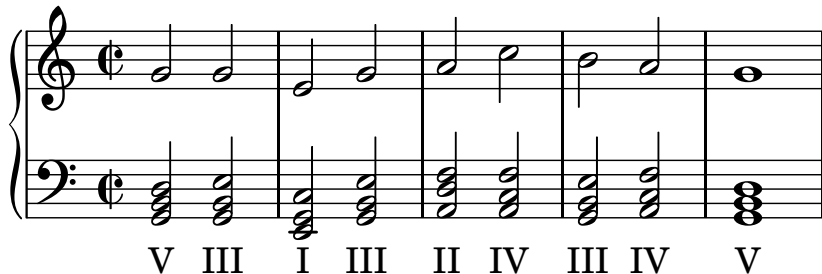
**Music generation**

Chord recognition: Chordify

Crowd-sourcing chord edits

# Application: automatic harmonisation of melodies

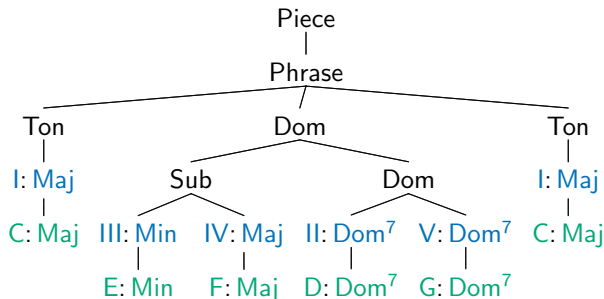
Another practical application of a harmony model is to help selecting good harmonisations (chord sequences) for a given melody:



The image displays a musical score for a single system. The upper staff is in the treble clef, showing a melody in C major. 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. The chord sequence is: V (C4-E4-G4), III (C4-E4-A4), I (C4-E4-G4), III (C4-E4-A4), II (D4-F4-A4), IV (F4-A4-C5), III (C4-E4-A4), IV (F4-A4-C5), and V (C4-E4-G4).

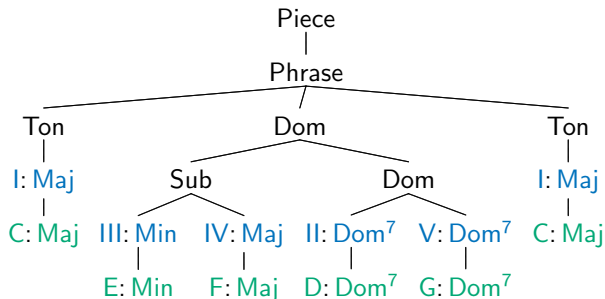
We generate candidate chord sequences, parse them with the harmony model, and select the one with the least errors.

# 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!

# 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 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 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. ... but generic programming saves us once again:

# 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 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... but generic programming saves us once again:

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

(You might recall my talk at the last IFIP meeting...)

# 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 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... but generic programming saves us once again:

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

(You might recall my talk at the last IFIP meeting...)

# 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

Harmonies without a melody are boring. I don't (yet) have a functional model of melody, so let's do something more mundane:

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

Harmonies without a melody are boring. I don't (yet) have a functional model of melody, so let's do something more mundane:

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.

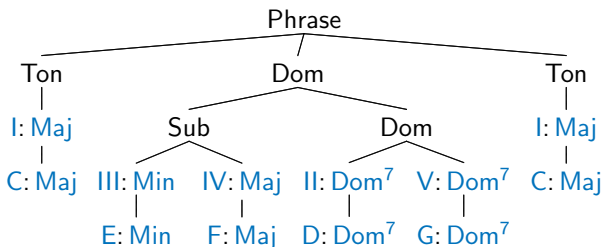
These four steps combine naturally using plain monadic bind:

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



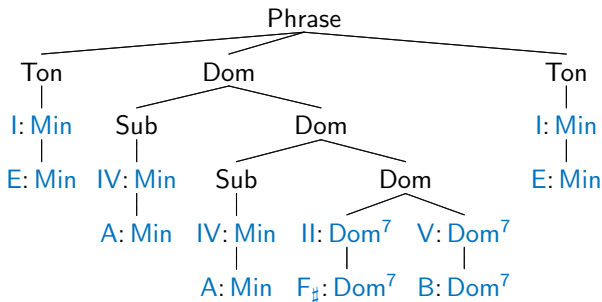
# Example I

A musical score consisting of two staves. The upper staff is a treble clef with a melodic line of eighth notes. The lower staff is a bass clef with a harmonic accompaniment of chords. The chords are: C major (C4, E4, G4), F major (F4, A4, C5), D minor (D4, F4, A4), G major (G4, B4, D5), and C major (C4, E4, G4).



# Example II

The musical score consists of two staves. The top staff is in treble clef with a key signature of one sharp (F#) and contains a melodic line of eighth notes: G4, A4, B4, C5, B4, A4, G4, F#4, E4, D4, C4. The bottom staff is in bass clef with a key signature of one sharp (F#) and contains a sequence of chords: G2-B2-E3 (G major), G2-B2-E3 (G major), G2-B2-E3 (G major), G2-B2-E3 (G major), G2-B2-E3 (G major), G2-B2-E3 (G major), G2-B2-E3 (G major), G2-B2-E3 (G major), G2-B2-E3 (G major), G2-B2-E3 (G major).



# Table of Contents

Harmony

A functional model of harmony

Harmony analysis

Harmonic similarity

Music generation

Chord recognition: Chordify

Crowd-sourcing chord edits

# Chord recognition: Chordify

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

Chord candidates	0.92 C	0.96 E	
	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.

# Demo: Chordify

Demo:

chordify<sup>®</sup>

<http://chordify.net>

# 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+



# Table of Contents

Harmony

A functional model of harmony

Harmony analysis

Harmonic similarity

Music generation

Chord recognition: Chordify

Crowd-sourcing chord edits

# Merging chord edits I

The next major feature planned for Chordify is to allow users to *edit* the automatically-recognised chord sequences.

# Merging chord edits I

The next major feature planned for Chordify is to allow users to *edit* the automatically-recognised chord sequences.

Two ways to make the edit process collaborative:

1. Wiki-style: your edits are visible to everyone;
2. You edit only your own version; the rest of the world still gets the automatically-recognised chords.

# Merging chord edits I

The next major feature planned for Chordify is to allow users to *edit* the automatically-recognised chord sequences.

Two ways to make the edit process collaborative:

1. Wiki-style: your edits are visible to everyone;
2. You edit only your own version; the rest of the world still gets the automatically-recognised chords.

We don't like (1); too prone to abuse. But (2) is not collaborative at all... unless we automatically improve the “automatically-recognised” default chords with user edits!

# Merging chord edits II

How will we do this? I don't know! But here are some ideas:

- ▶ Use metrics such as:
  - ▶ Regularity (e.g. do the chords change on strong metrical positions?)
  - ▶ Repetition (music always contains repetition)
  - ▶ How well it fits in the harmony model (e.g. number of parsing errors)
  - ▶ User reputation
- ▶ Detect concordance among edits by several users;
- ▶ ...and hopefully we'll get a PhD student to work a bit on this.

# Merging chord edits II

How will we do this? I don't know! But here are some ideas:

- ▶ Use metrics such as:
  - ▶ Regularity (e.g. do the chords change on strong metrical positions?)
  - ▶ Repetition (music always contains repetition)
  - ▶ How well it fits in the harmony model (e.g. number of parsing errors)
  - ▶ User reputation
- ▶ Detect concordance among edits by several users;
- ▶ ...and hopefully we'll get a PhD student to work a bit on this.

Ultimate goal: have a public database of high-quality human-annotated songs from public sources (Youtube/SoundCloud)—this is lacking, and could be quite helpful to improve automatic chord estimation.

# 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
- ▶ Analysing harmony—using error-correcting parsers
- ▶ Finding cover songs—with a generic diff
- ▶ Generating harmonies—with QuickCheck
- ▶ Recognising harmony from audio sources
- ▶ The synergy between two different CS fields, and the advantages of transporting academic research into industry

# 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
- ▶ Analysing harmony—using error-correcting parsers
- ▶ Finding cover songs—with a generic diff
- ▶ Generating harmonies—with QuickCheck
- ▶ Recognising harmony from audio sources
- ▶ The synergy between two different CS fields, and the advantages of transporting academic research into industry

Thank you for your time!