



Universiteit Utrecht

[Faculty of Science  
Information and Computing Sciences]

# Functional Modelling of Musical Harmony

José Pedro Magalhães  
joint work with W. Bas de Haas

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

September 20, 2011

# Outline

Harmony

A harmony model in Haskell

From chord text to harmonic structure

Increasing model complexity

Harmonic similarity

Conclusion



# Outline

## Harmony

A harmony model in Haskell

From chord text to harmonic structure

Increasing model complexity

Harmonic similarity

Conclusion



# What is harmony?

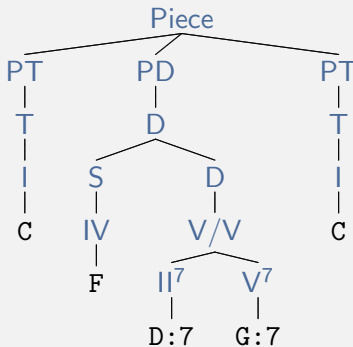
The diagram shows a sequence of five chords on a treble clef staff. Above the staff, the chords are labeled with Roman numerals and functional harmony terms. The first chord is C major (I), labeled 'Ton'. The second is F major (IV), labeled 'SDom'. The third is D7 (V/V), labeled 'Dom' with a bracket over the next two chords. The fourth is G7 (V), also labeled 'Dom'. The fifth is C major (I), labeled 'Ton'. Below the staff, the chords are labeled with their letter names: C, F, D<sup>7</sup>, G<sup>7</sup>, and C.

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



# An example harmonic analysis

*Ton*      *SDom*      *Dom*      *Ton*  
 I          IV          V/V      V          I  
 C          F          D<sup>7</sup>      G<sup>7</sup>      C



# Why are harmony models useful?

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

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

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



# Outline

Harmony

A harmony model in Haskell

From chord text to harmonic structure

Increasing model complexity

Harmonic similarity

Conclusion



# Representing harmony as a datatype, naively

**data** Piece = Piece [Phrase]

**data** Phrase = PT Ton | PD Dom

**data** Ton = T<sub>IMaj</sub> DegNum

**data** Dom = D<sub>VMaj</sub> DegNum | D<sub>S<sub>Dom</sub></sub> S<sub>Dom</sub> Dom

**data** S<sub>Dom</sub> = S<sub>IVMaj</sub> DegNum

**data** DegNum = I | II | III ...





# Representing harmony as a datatype, naively

```
data Piece    = Piece [Phrase]
data Phrase  = PT    Ton    | PD    Dom
data Ton     = TIMaj DegNum
data Dom     = DVMaj DegNum | DSDom SDom Dom
data SDom    = SIVMaj DegNum
data DegNum = I | II | III ...
```

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

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



# Representing harmony as a datatype I

**data** Piece = Piece [Phrase]

**data** Phrase = PT Ton | PD Dom

**data** Ton = T<sub>IMaj</sub> (Deg I)

**data** Dom = D<sub>VMaj</sub> (Deg V) | D<sub>SDom</sub> SDom Dom

**data** SDom = S<sub>IVMaj</sub> (Deg IV)

**data** I; **data** II; ... **data** VII;

**data** Deg  $\delta$  = Deg DegNum

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



# Representing harmony as a datatype II

```
deg :: ToDegree  $\delta$   $\Rightarrow$   $\delta$   $\rightarrow$  Deg  $\delta$   
deg d = Deg (toDegree d)
```

We need a mechanism to perform type-to-value conversions for degrees:

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



# Representing harmony as a datatype III

Now the term `Piece [PT (TIMaj (deg ( $\perp$  :: II)))]` does not typecheck, as we wanted:

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

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



# Representing harmony as a datatype IV

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

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

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



# Outline

Harmony

A harmony model in Haskell

From chord text to harmonic structure

Increasing model complexity

Harmonic similarity

Conclusion



# Parsing chord sequences I

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

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

We use uu-parsinglib, but we hide the details for this talk:

```
data ParserMusic  $\alpha$  -- abstract  
instance Alternative ParserMusic  
instance Applicative ParserMusic  
instance Functor ParserMusic
```



# Parsing chord sequences II

Most instances are trivial:

**instance** Parse Phrase **where**

parse = PT  $\langle \$ \rangle$  parse  
 $\langle | \rangle$  PD  $\langle \$ \rangle$  parse

**instance** Parse Dom **where**

parse = D<sub>VMaj</sub>  $\langle \$ \rangle$  parse  
 $\langle | \rangle$  D<sub>SDom</sub>  $\langle \$ \rangle$  parse  $\langle * \rangle$  parse

...

So trivial that we do not write them; we use a **generic parser**.  
We can only do this because we have type-level degrees!





# Outline

Harmony

A harmony model in Haskell

From chord text to harmonic structure

Increasing model complexity

Harmonic similarity

Conclusion



# 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  $\underline{Ab^7} G^7 C_{Maj}$

...



# Secondary dominants—datatype representation

Here the fun begins! Adapt the `Deg` type:

```
type DegSD  $\delta$  = BDegSD  $\delta$ 
```

```
data BDegSD  $\delta$  where
```

```
  Cons :: BDegSD (PerfV  $\delta$ )  $\rightarrow$  Deg  $\delta$   $\rightarrow$  BDegSD  $\delta$ 
```

```
  Base :: Deg  $\delta$   $\rightarrow$  BDegSD  $\delta$ 
```



# Secondary dominants—datatype representation

Here the fun begins! Adapt the `Deg` type:

```
type DegSD  $\delta$  = BDegSD  $\delta$ 
```

```
data BDegSD  $\delta$  where
```

```
  Cons :: BDegSD (PerfV  $\delta$ )  $\rightarrow$  Deg  $\delta$   $\rightarrow$  BDegSD  $\delta$ 
```

```
  Base :: Deg  $\delta$   $\rightarrow$  BDegSD  $\delta$ 
```

```
data Deg  $\delta$  = Deg ... -- as before
```

```
type family PerfV  $\delta$  -- computes the perfect fifth
```

```
type instance PerfV I = V
```

```
type instance PerfV V = II
```

```
...
```



# Secondary dominants—datatype representation

Here the fun begins! Adapt the `Deg` type:

```
type DegSD δ = BDegSD δ (Su (Su (Su Ze)))
```

```
data BDegSD δ η where
```

```
  Cons :: BDegSD (PerfV δ) η → Deg δ → BDegSD δ (Su η)
```

```
  Base :: Deg δ → BDegSD δ (Su η)
```

```
data Deg δ = Deg... -- as before
```

```
type family PerfV δ -- computes the perfect fifth
```

```
type instance PerfV I = V
```

```
type instance PerfV V = II
```

...

```
data Ze -- type level zero
```

```
data Su η -- type level successor
```



## Secondary dominants—instances

We also need an instance of `Parse` for `BDegSD`:

```
instance ( Parse (Deg δ)
           , Parse (BDegSD (PerfV δ) η) )
  => Parse (BDegSD δ (Su η)) where
  parse = Base <$> parse
        <|> Cons <$> parse <*> parse
```

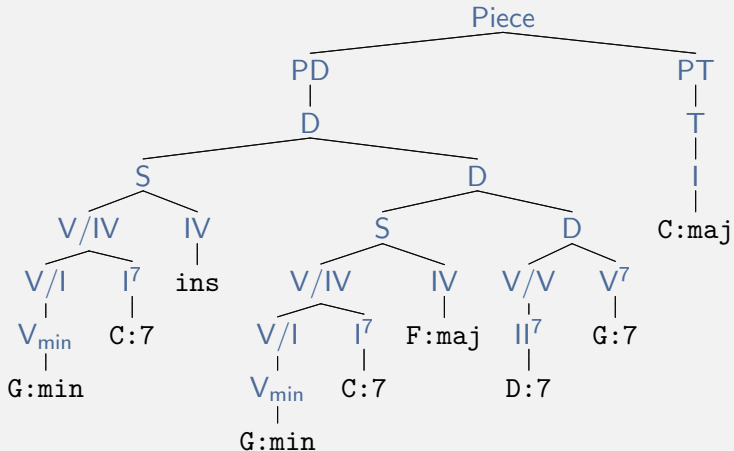
The code for `parse` is trivial, but the instance head is not. These instances are harder to handle generically!

This sparked work in “Generic Programming for Indexed Datatypes”, presented at WGP11.



# Example

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



# Outline

Harmony

A harmony model in Haskell

From chord text to harmonic structure

Increasing model complexity

Harmonic similarity

Conclusion





# Harmonic similarity

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



# Outline

Harmony

A harmony model in Haskell

From chord text to harmonic structure

Increasing model complexity

Harmonic similarity

Conclusion



# Comparison to a Java approach

When comparing the current solution to an earlier Java approach, we find that:

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



# Summary

What we have so far:

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



# Summary

What we have so far:

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

What we are working on:

- ▶ Recognizing harmony from audio sources



# Limitations

- ▶ Performance:
  - ▶ Type checking: exponential on  $\eta$
  - ▶ At runtime: grammar ambiguity
- ▶ Model:
  - ▶ Instances for complicated GADTs
  - ▶ Modulations



# Use it!

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

