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`

August 26, 2011

This talk

- A non-trivial application of several advanced functional programming techniques

This talk

- A non-trivial application of several advanced functional programming techniques
- Modeling musical harmony using Haskell datatypes

This talk

- A non-trivial application of several advanced functional programming techniques
- Modeling musical harmony using Haskell datatypes
- From textual chord labels to analysis trees

This talk

- A non-trivial application of several advanced functional programming techniques
- Modeling musical harmony using Haskell datatypes
- From textual chord labels to analysis trees
- Why Haskell is a better fit than Java for harmony modeling

This talk

- A non-trivial application of several advanced functional programming techniques
- Modeling musical harmony using Haskell datatypes
- From textual chord labels to analysis trees
- Why Haskell is a better fit than Java for harmony modeling
- To be presented at the 16th International Conference on Functional Programming (ICFP'11) in Tokyo, Japan

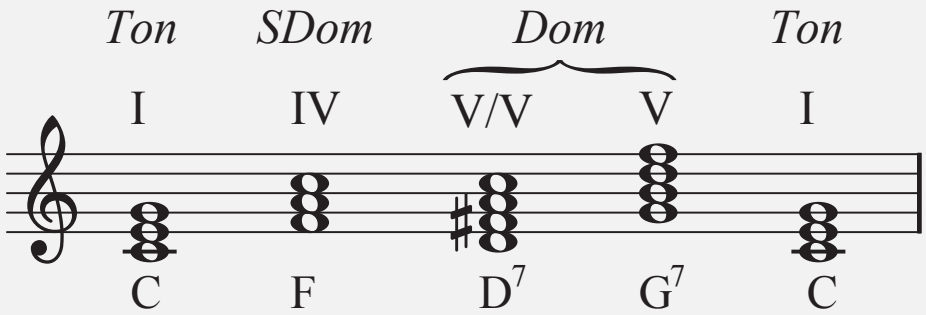
This talk

- A non-trivial application of several advanced functional programming techniques
- Modeling musical harmony using Haskell datatypes
- From textual chord labels to analysis trees
- Why Haskell is a better fit than Java for harmony modeling
- To be presented at the 16th International Conference on Functional Programming (ICFP'11) in Tokyo, Japan
- Work in progress

Outline

Outline

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

Simplified harmony theory I

- A chord is as a group of tones separated by intervals of roughly the same size.

Simplified harmony theory I

- A chord is as a group of tones separated by intervals of roughly the same size.
- All music is made out of chords (whether explicitly or not).

Simplified harmony theory I

- A chord is as 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.

Simplified harmony theory I

- A chord is as 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.

Simplified harmony theory I

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

Simplified harmony theory II

Models for musical harmony explain the harmonic progression in music:

- Everything works around the tonic (I).

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.

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.

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.

Simplified harmony theory II

Models for musical harmony explain the harmonic progression in music:

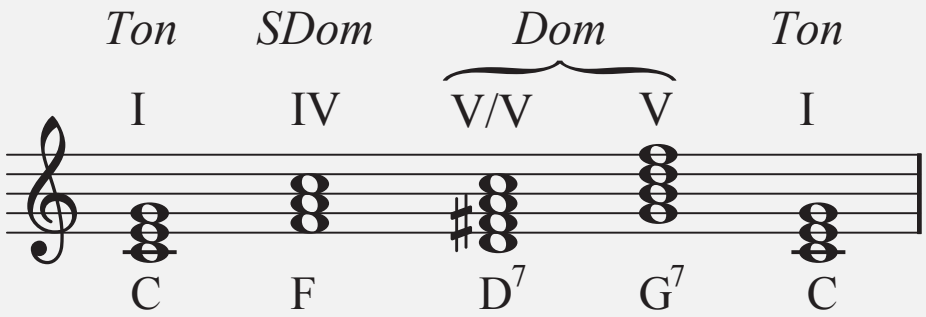
- Everything works around the tonic (I).
- The dominant (V) leads to the tonic.
- The subdominant (IV) tends to lead to the dominant.
- Therefore, the I IV V I progression is very common.
- There are also secondary dominants, which lead to a relative tonic. For instance, II^7 is the secondary dominant of V, and I^7 is the secondary dominant of IV.

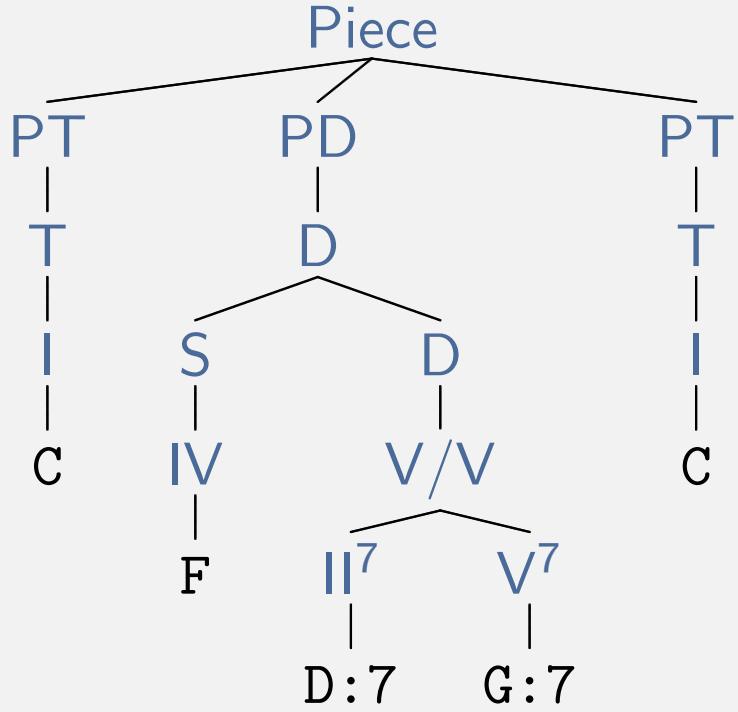
Simplified harmony theory II

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





Universiteit Utrecht

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

Why are harmony models useful?

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

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

Why are harmony models useful?

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

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

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

Outline

Representing harmony as a datatype, naively

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

Universiteit Utrecht

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

Representing harmony as a datatype, naively

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

Problem: the term $\mathsf{Piece}\ [\mathsf{PT}\ (\mathsf{T}_{\mathsf{IMaj}}\ \mathsf{II})]$ typechecks, but represents an invalid harmonic structure. $\mathsf{T}_{\mathsf{IMaj}}$ 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_IMaj (Deg I)
data Dom    = D_VMaj (Deg V) | D_SDom SDom Dom
data SDom   = S_IVMaj (Deg IV)
```

```
data Deg δ = Deg DegNum
```

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

Representing harmony as a datatype II

```
deg :: ToDegree δ ⇒ δ → Deg δ
deg d = Deg (toDegree d)
```

We need degrees at the type level:

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

And type-to-value conversions for degrees:

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

Representing harmony as a datatype III

Now the term Piece [PT (T_{IMaj} (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.

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

Outline

Parsing chord sequences I

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

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

Parsing chord sequences I

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

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

We use Doaitse's parser combinators, but we hide the details for this talk:

```
data ParserMusic α   -- abstract

instance Alternative ParserMusic
instance Applicative ParserMusic
instance Functor     ParserMusic
```

Parsing chord sequences II

Most instances are trivial:

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

Parsing chord sequences II

Most instances are trivial:

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

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

A generic parser

We use `instant-generics`:

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

instance Parse U where
  parse = pure U

instance (Parse α, Parse β) ⇒ Parse (α :+: β) where
  parse = L <$> parse <|> R <$> parse

instance (Parse α, Parse β) ⇒ Parse (α :×: β) where
  parse = (:×:) <$> parse <*> parse
```

A generic parser

We use `instant-generics`:

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

instance Parse U where
  parse = pure U

instance (Parse α, Parse β) ⇒ Parse (α :+: β) where
  parse = L <$> parse <|> R <$> parse

instance (Parse α, Parse β) ⇒ Parse (α :×: β) where
  parse = (:×:) <$> parse <*> parse


parseDefault :: (Representable α, Parse (Rep α))
             ⇒ ParserMusic α
parseDefault = fmap to parse
```

Parsing degrees

For degrees we need an adhoc parser:

```
instance (ToDegree δ) ⇒ Parse (Deg δ) where
  parse = pChord (toDegree (⊥ :: δ))

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

Note how it relies on scoped type variables. Again we use the type-to-value conversion class ToDegree.

Outline

Increasing complexity

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

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

Increasing complexity

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

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

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

Increasing complexity

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

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

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

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

Increasing complexity

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

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

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

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

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

Increasing complexity

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

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

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

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

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

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

...

Handling repeated chords

Easy: adapt the Deg type and pChord:

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

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

Secondary dominants—datatype representation

Here the fun begins! Adapt the Deg type again:

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

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

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

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

Secondary dominants—datatype representation

Here the fun begins! Adapt the Deg type again:

```
type Deg_SD δ = BDeg_SD δ
data BDeg_SD δ where
  Cons :: BDeg_SD (Perf_V δ) → Deg δ → BDeg_SD δ
  Base ::                      Deg δ → BDeg_SD δ

data Deg δ = Deg ...    -- as before

type family   Perf_V δ   -- computes the perfect fifth
type instance Perf_V I  = V
type instance Perf_V V  = II
...
```

Secondary dominants—datatype representation

Here the fun begins! Adapt the Deg type again:

```
type Deg_SD δ = BDeg_SD δ (Su (Su (Su Ze)))
data BDeg_SD δ η where
  Cons :: BDeg_SD (Perf_V δ) η → Deg δ → BDeg_SD δ (Su η)
  Base ::                        Deg δ → BDeg_SD δ (Su η)

data Deg δ = Deg ...    -- as before

type family   Perf_V δ   -- computes the perfect fifth
type instance Perf_V I  = V
type instance Perf_V V  = II
...
data Ze      -- type level zero
data Su η    -- type level successor
```

Secondary dominants—instances

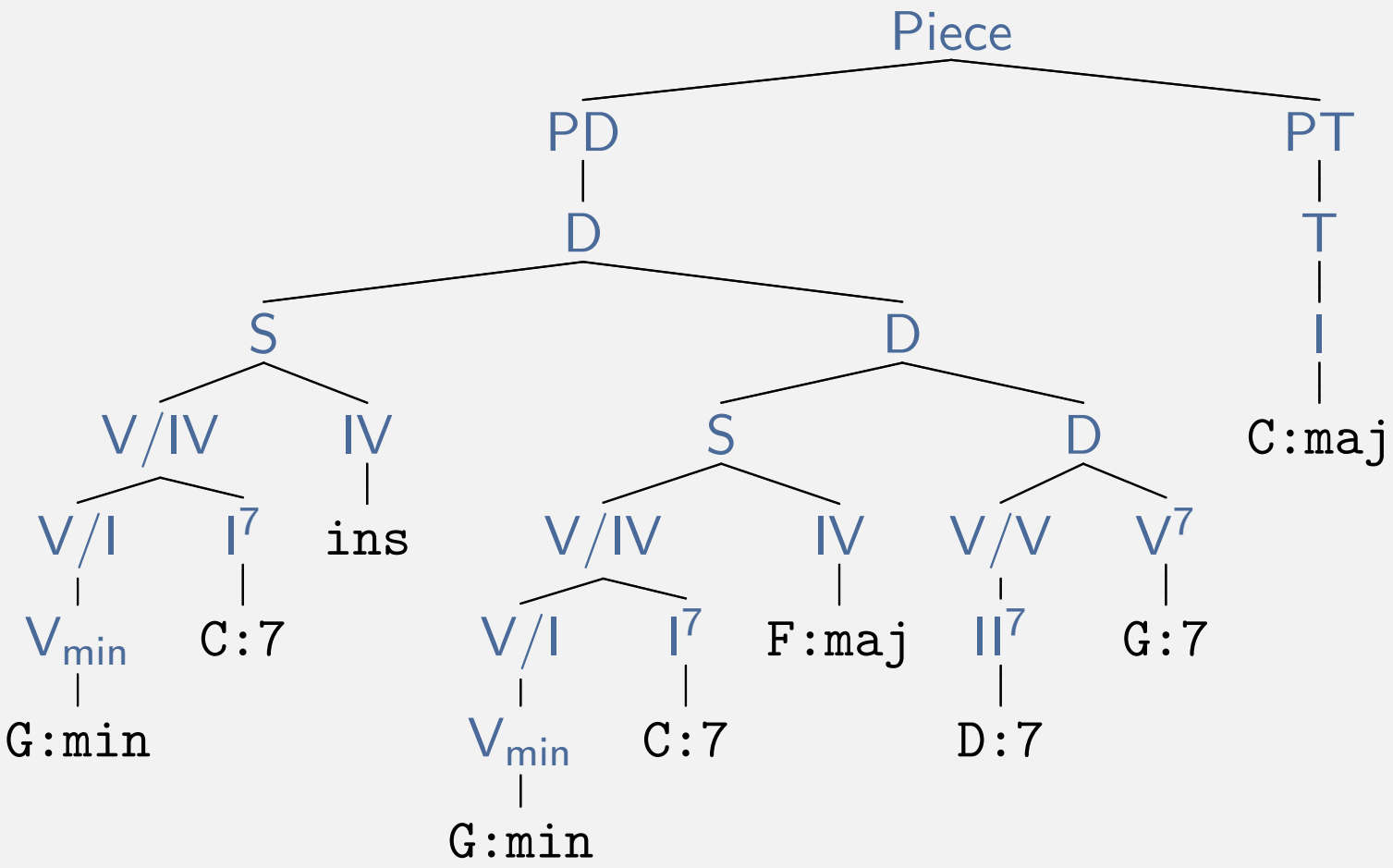
We also need an instance of Parse for $\mathsf{BDeg}_{\mathsf{SD}}$:

```
instance  (  Parse (Deg δ)
          ,  Parse (BDeg_SD (Perf_V δ) η      ))
          ⇒  Parse (BDeg_SD 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!

Example

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



Outline

Harmonic similarity

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

Generic diff

- On Hackage: `gdiff-ig`
- Relies on four generic functions: children, build, shallowEq, and typeOf
- We implement the first three in `instant-generics`, and take the latter from `Data.Typeable`
- We rely on existential quantification for diffing values of different types
- For performance we use unboxed Ints and strict constructor fields
- Memoization is crucial

Outline

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

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

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

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

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

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 η
  - At runtime: grammar ambiguity
- Model:
  - Instances for complicated GADTs
  - Modulations

Play with it!

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