Advanced Functional Programming for Fun and Profit

José Pedro Magalhães

Department of Computer Science, University of Oxford
http://dreixel.net

August 30, 2013
Utrecht University, The Netherlands

Introduction

This is a two-part talk:

- Part I: new and (even more) advanced functional programming in GHC 7.6
- Part II: Applying advanced functional programming techniques to music modelling

Part I

Advanced AFP

More features, more fun

Exciting new features in GHC 7.6:

- Data kinds
- Kind polymorphism
- Type-level literals
- Deferred type errors

We'll go through a few examples of how to put these new features to good use.

Colours

In this talk I will use:

- Blue for constructors (most of the time)
  Nothing, False, Left True, 3, "abc", 'p'

Colours

In this talk I will use:

- Blue for constructors (most of the time)
  Nothing, False, Left True, 3, "abc", 'p'
- Red for types
  Int, String, Show a, **data** () = ()

Colours

In this talk I will use:

- Blue for constructors (most of the time)
  Nothing, False, Left True, 3, "abc", 'p'
- Red for types
  Int, String, Show a, **data** () = ()
- Green for kinds
  $\star$, $\star \rightarrow \star$

What are kinds?

Just like types classify values...

```
3        :: Num a ⇒ a
'p'      :: Char
Just ()  :: Maybe ()
"abc"    :: String
```

What are kinds?

Just like types classify values...

3 :: Num a $\Rightarrow$ a

'p' :: Char

Just () :: Maybe ()

"abc" :: String

... kinds classify types:

Int :: $\star$

Char :: $\star$

Maybe :: $\star \rightarrow \star$

[] :: $\star \rightarrow \star$

The language of kinds

However, the language of kinds, unlike that of types, is rather limited:

$$\begin{aligned} k ::= & \star \\ \mid & \; k \to k \end{aligned}$$

In particular: no user defined kinds, no kind variables.

Diversion: the Constraint kind

With `-XConstraintKinds` we get one new base kind to classify constraints:

$$\begin{aligned}
\text{Show} &:: \star \to \text{Constraint} \\
\text{Functor} &:: (\star \to \star) \to \text{Constraint} \\
\text{Num Int} &:: \text{Constraint} \\
\text{Int} \sim \text{Bool} &:: \text{Constraint}
\end{aligned}$$

Why do we need a better kind system? I

We often want to restrict type arguments to a particular kind:

```
data Ze
data Su n

data Vec :: ⋆ → ⋆ → ⋆ where
  Nil  :: Vec Ze a
  Cons :: a → Vec n a → Vec (Su n) a
```

Types like Vec Int Int, Vec Int Bool, and Vec () () are valid (albeit uninhabited). We want to say that the first argument of Vec should only be Ze or Su!

Why do we need a better kind system? II

Lack of kind polymorphism leads to code duplication:

```
class Typeable  (a :: ⋆)              where
  typeOf  :: a      → TypeRep
class Typeable_1 (a :: ⋆ → ⋆)        where
  typeOf_1 :: a b   → TypeRep
class Typeable_2 (a :: ⋆ → ⋆ → ⋆) where
  typeOf_2 :: a b c → TypeRep
```

We would rather have a single, kind-polymorphic Typeable class!

Datatype promotion I

With `-XDataKinds`, the following code is valid:

```
data Nat = Ze | Su Nat

data Vec :: Nat → ⋆ → ⋆ where
  Nil  :: Vec Ze a
  Cons :: a → Vec n a → Vec (Su n) a
```

Note the implicit promotion of the constructors Ze and Su to types Ze and Su, and of the type Nat to the kind Nat.
Types like Vec Int Int now trigger a kind error!

Datatype promotion II

Type families can also be indexed over promoted types:

```
type family Add (m :: Nat) (n :: Nat) :: Nat
type instance Add Ze     n = n
type instance Add (Su m) n = Su (Add m n)
```

Datatype promotion II

Type families can also be indexed over promoted types:

```
type family Add (m :: Nat) (n :: Nat) :: Nat
type instance Add Ze     n = n
type instance Add (Su m) n = Su (Add m n)

append :: Vec m a → Vec n a → Vec (Add m n) a
append Nil         v = v
append (Cons h t) v = Cons h (append t v)
```

This was all possible before, but now we can express the right kind of Add.

Promoted lists and tuples

Haskell lists are natively promoted, so we can encode heterogeneous lists as follows:

```
data HList :: [⋆] → ⋆ where
  HNil  :: HList []
  HCons :: a → HList t → HList (a : t)
```

As an example, here is a heterogeneous collection:

```
hetList :: HList [Int, Bool]
hetList = HCons 3 (HCons False HNil)
```

Tuples are also promoted, e.g. $(\star, \star \to \star, \text{Constraint})$.

Kind-polymorphic type equality

Kind polymorphism reduces code duplication:

```
data Eq_T a b where
  Refl :: Eq_T a a
```

Previously the kind of $\mathrm{Eq_T}$ would default to $\star \to \star \to \star$. With `-XPolyKinds` it doesn't, so the following types are all valid: $\mathrm{Eq_T}$ a Int, $\mathrm{Eq_T}$ f Maybe, $\mathrm{Eq_T}$ t Either.

Kind-polymorphic Typeable

Now we can define a single kind-polymorphic Typeable class:

```
data Proxy (t :: k) = Proxy
class Typeable (t :: k) where
  typeRep :: Proxy t → TypeRep
```

Note that Proxy is kind polymorphic!

Kind-polymorphic Typeable

Now we can define a single kind-polymorphic Typeable class:

```
data Proxy (t :: k) = Proxy
class Typeable (t :: k) where
  typeRep :: Proxy t → TypeRep
```

Note that Proxy is kind polymorphic!

We can now give Typeable instances for types of various kinds:

```
instance Typeable Char   where ...
instance Typeable []     where ...
instance Typeable Either where ...
```

Type-level literals

Thanks to Iavor Diatchki's work, we now have efficient type-level naturals:

$$0, 1, 2, \ldots :: \mathsf{Nat}$$

Note the colours!
These type-level naturals come with associated operations:

$$(\leqslant) :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Constraint}$$
$$(+) :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$$
$$(*) :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$$
$$(\hat{}) :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$$

Value-level reflection

How do we manipulate values representing type-level naturals? There is a family of singleton types, parameterised by literals:

newtype Sing :: a $\rightarrow \star$

Value-level reflection

How do we manipulate values representing type-level naturals? There is a family of singleton types, parameterised by literals:

```
newtype Sing :: a → ⋆
```

From types to values:

```
fromSing :: Sing a → SingRep a

type family SingRep a
type instance SingRep (a :: Nat)    = Integer
type instance SingRep (a :: Symbol) = String
```

Note that we can have type-level literals other than naturals, and SingRep is a *kind-indexed* family!

Revisiting vectors

Revisiting vectors, now with type-level naturals:

```
data Vec :: Nat → ⋆ → ⋆ where
  Nil  :: Vec 0 a
  Cons :: a → Vec n a → Vec (n + 1) a
```

Vector concatenation uses type-level natural number addition:

```
append :: Vec m a → Vec n a → Vec (m + n) a
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Why are type-level naturals hard to implement?

```
append :: Vec m a → Vec n a → Vec (m + n) a
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Function append requires GHC to prove equalities between natural number expressions:

- Could not deduce $(n \sim (0 + n))$ from the context $(m \sim 0)$ bound by a pattern with constructor $\mathsf{Nil} :: \forall a.\mathsf{Vec}\ 0\ a$
- Could not deduce $((m + n) \sim ((n' + n) + 1))$ from the context $(m \sim (n' + 1))$ bound by a pattern with constructor $\mathsf{Cons} :: \forall a\ (n :: \mathsf{Nat}).a \to \mathsf{Vec}\ n\ a \to \mathsf{Vec}\ (n + 1)\ a$

We need an equation solver!

The illogical next step

What is the next thing that you want, when you have data kinds, polymorphic kinds, and type-level literals?

The illogical next step

What is the next thing that you want, when you have data kinds, polymorphic kinds, and type-level literals?

Naturally, to turn off type checking! :-)

Why would you want to do that?

For instance:

- Prototyping
- Large refactoring
- IDE

Example I

With the flag `-fdefer-type-errors`, this example:

```
p, q :: Int
p = 1
q = '1'
main = print p
```

Compiles with warning: "couldn't match expected type Int with actual type Char in an equation for q: q = '1'". Runs and returns 1!

Example II

```
p, q :: Int
p = 1
q = '1'
main = print q
```

Fails at runtime with: "couldn't match expected type Int with actual type Char in an equation for q: q = '1'".

Example III

```
t1 :: Int
t1 = '1'

t2 :: a → String
t2 = show

data T a where
  T1 :: Int → T Int
  T2 :: a   → T a

t3 :: T a
t3 = T1 0

main = print 1
```

Runs fine!

How it works

GHC's core language uses *coercions* to (safely) cast terms:

$$\begin{aligned}
&\textbf{data}\ \mathsf{T}\ \mathsf{a} = \mathsf{T}_1\ (\mathsf{a} \sim \mathsf{Int})\ \mathsf{Int} \mid \mathsf{T}_2\ \mathsf{a} \\
&\mathsf{unT} :: \mathsf{T}\ \mathsf{a} \to \mathsf{a} \\
&\mathsf{unT}\ (\mathsf{T}_1\ \mathsf{c}\ \mathsf{n}) = \mathsf{n} \rhd (\mathsf{sym}\ \mathsf{c}) \\
&\mathsf{unT}\ (\mathsf{T}_2\ \mathsf{x}) \quad = \mathsf{x} \\
&\rhd :: \mathsf{b} \to (\mathsf{b} \sim \mathsf{a}) \to \mathsf{a}
\end{aligned}$$

Evidence, or values of type $(\sim)$, is automatically generated by GHC during type checking. Deferring type errors simply means generating runtime errors as evidence!

It's not dynamic typing!

Note that deferring type errors doesn't mean any form of checks are performed at runtime. Consider this example:

```
f :: ∀a.a → a → a
f x y = x ∧ y
main = print (f True False)
```

It still fails at runtime!

Part I summary

A better kind system gives us:

- Increase type safety
- Increase expressivity
- Reduce code duplication
- Allow for writing clearer code

Part I summary

A better kind system gives us:

- Increase type safety
- Increase expressivity
- Reduce code duplication
- Allow for writing clearer code

And if we get tired of it we can always defer errors to runtime!

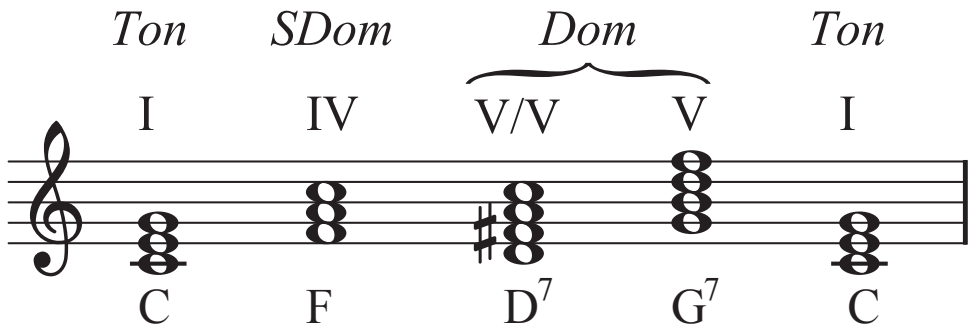
Part II

Haskell, music, fun. . . and maybe even profit!

Introduction

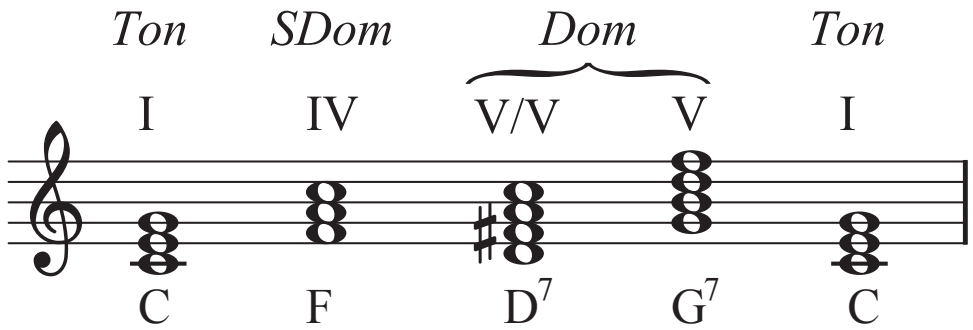
- Modelling musical harmony using Haskell
- Avoiding programming
- Applications of a model of harmony:
  - Musical analysis
  - Finding cover songs
  - Generating chords for melodies
  - Correcting errors in chord extraction from audio sources

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

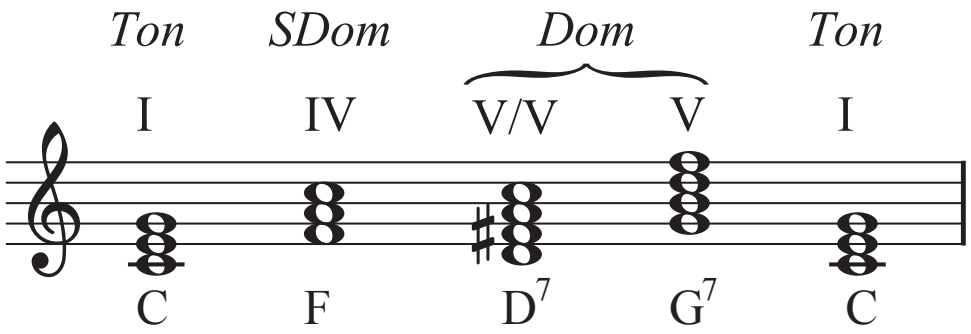
What is harmony?

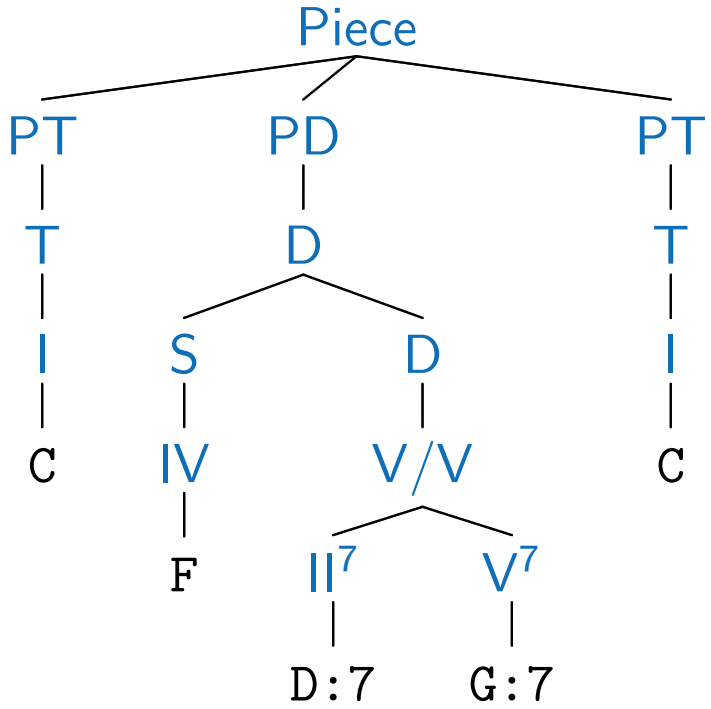


- Harmony arises when at least two notes sound at the same time
- Harmony induces tension and release patterns, that can be described by music theory and music cognition
- The internal structure of the chord has a large influence on the consonance or dissonance of a chord
- The surrounding context also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

Demo: how harmony affects melody

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)

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 ...
```

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 ...
```

Problem: the term Piece [PT (T_{IMaj} II)] typechecks, but represents an invalid harmonic structure. T_{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.

Parsing chord sequences I

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

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

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*.

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 = ...
```

Again we use the type-to-value conversion class ToDegree.

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
```

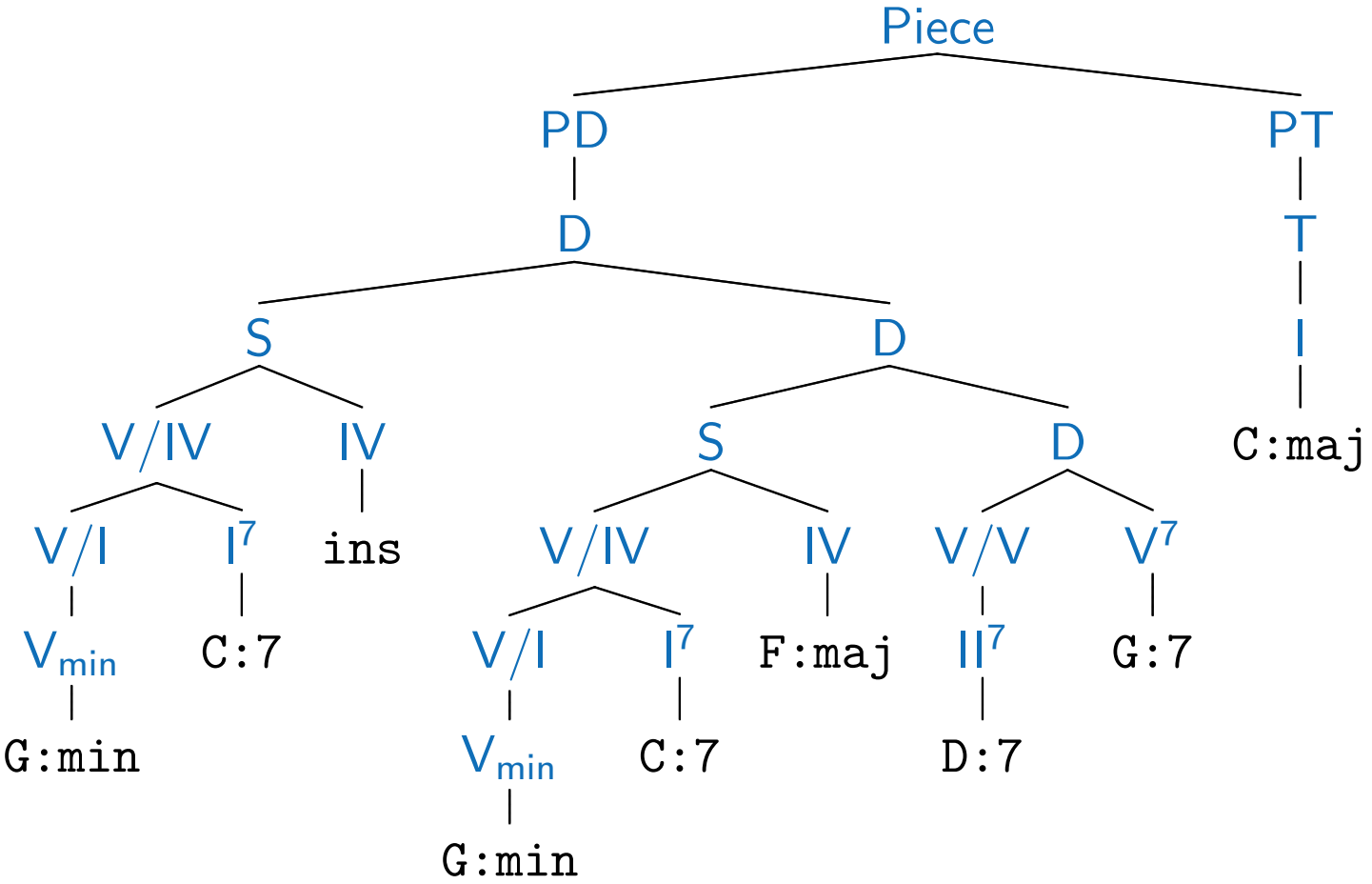
Secondary dominants—datatype representation

Here the fun really begins! We 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
...
```

Application: harmony analysis

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

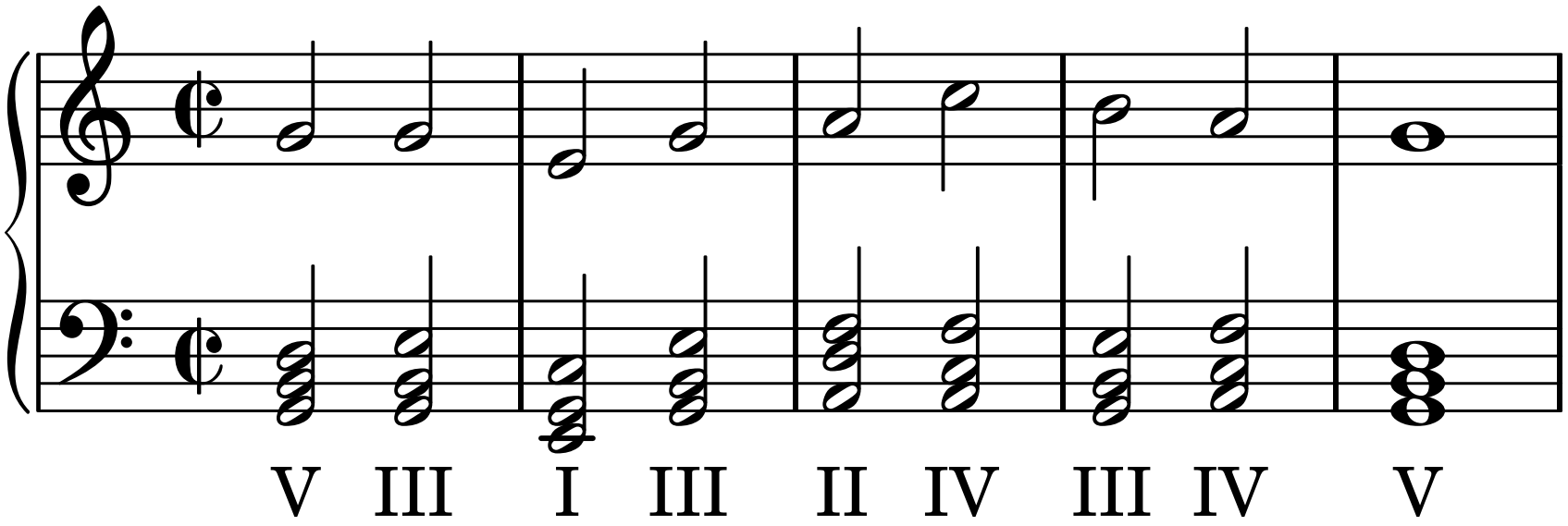


Application: harmonic similarity

- 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

Application: automatic harmonisation of melodies

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.

Application: 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.

Demo: `http://chordify.net`

Summary

Musical modelling with Haskell:

- A model for musical harmony as a 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
- Harmonising melodies
- Recognising harmony from audio sources

Play with it!

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

chordify®

`http://chordify.net`