HaMusic: Haskell toolkit for musical analysis

Samuel Silva [silva.samuel@alumni.uminho.pt]

October 29, 2008

Contents

1 Top module

MusicAnalysis is a collection of Haskell modules to improve music analysing. Actuallly this library offers simple transformations like transpositions, tempo changes, and so on.

To run MusicAnalysis library it must use these Haskell language extensions:

MultiParamtypeClasses This extension allows classes parametrics,

TypeSynonymInstances This extension allows instances over types,

FlexibleInstances This allows powerfull instances.

This is main module. Figure expose architecture of Music Analysis library.

```
-- |
-- Maintainer : silva.samuel@alumni.uminho.pt
-- Stability : experimental
-- Portability: HaXML, Haskore
-- main module
module Music.Analysis (
  module Music.Analysis.PF,
  module Music.Analysis.Base,
  module Music.Analysis.Script
  ) where


import Music.Analysis.PF
import Music.Analysis.Base hiding (toInteger)
import Music.Analysis.Script
import Script ()
```

2 Base definition's

2.1 Point-Free combinators

This module imprements a Point-Free library using Formal Methods approach. See Mpi.hs or CP.hs from http://www.di.uminho.pt/

```
-- |
-- Maintainer : silva.samuel@alumni.uminho.pt
```

```
  -- Stability : experimental
  -- Portability: portable
  -- This module implements Point-Free library
module Music.Analysis.PF where
import Data.Maybe (Maybe (..), maybe)
import Data.Either ((..) + ·, [·, ·])
import Data.Bool (Bool)
import Data.Tuple (uncurry, curry, π1, π2)
import Data.Function ((·), ($), id, _)
import Data.List ((++))
import Prelude ()
infix 5 ×
infix 4 +

  -- * Product
  -- ** Combinators
  -- | split
< ·, · > ::(a → b) → (a → c) → a → (b, c)
< f, g >  x = (f x, g x)
  -- | product
(×) :: (a → b) → (c → d) → (a, c) → (b, d)
f × g =< f · π1, g · π2 >
  -- | the 0-adic split
(!) :: a → ()
(!) = _()

  -- ** Renamings
  -- | fst
π1 :: (a, b) → a
π1 = π1
  -- | snd
π2 :: (a, b) → b
π2 = π2

  -- * Coproduct
  -- ** Renamings
  -- | Left
i1 :: a → a + b
i1 = i1
  -- | Right
i2 :: b → a + b
i2 = i2
  -- ** Combinators
  -- either is predefined
  -- | sum
(+) :: (a → b) → (c → d) → a + c → b + d
f + g = [i1 · f, i2 · g]
  -- | McCarthy's conditional:
cond :: (b → Bool) → (b → c) → (b → c) → b → c
cond p f  g = ([f, g]) · (grd p)

  -- * Exponentiation
  -- ** Combinators
```

```
  -- curry is predefined
  -- | ap
ap :: (a → b, a) → b
ap = uncurry ($)
  -- | expn
expn :: (b → c) → (a → b) → a → c
expn f = curry (f · ap)
  -- exponentiation functor is (a->) predefined

  -- * Others
  -- @const :: a -> b -> a st @
  -- @const a x = a is predefined@
  -- | guard
grd :: (a → Bool) → a → a + a
grd p x = if p x then i1 x else i2 x

  -- * Natural isomorphisms
  -- | swap
swap :: (a, b) → (b, a)
swap =< π2, π1 >
  -- | assoc right
assocr :: ((a, b), c) → (a, (b, c))
assocr =< π1 · π1, < π2 · π1, π2 >>
  -- | assoc left
assocl :: (a, (b, c)) → ((a, b), c)
assocl =< id × π1, π2 · π2 >
  -- | dist right
distr :: (a, b + c) → (a, b) + (a, c)
distr (a, i1 b) = i1 (a, b)
distr (a, i2 c) = i2 (a, c)
  -- | undist right
undistr :: (a, b) + (a, c) → (a, b + c)
undistr = [id × i1, id × i2]
  -- | flat right
flatr :: (a, (b, c)) → (a, b, c)
flatr (a, (b, c)) = (a, b, c)
  -- | flat left
flatl :: ((a, b), c) → (a, b, c)
flatl ((b, c), d) = (b, c, d)
  -- | unflat right
unflatr :: (a, b, c) → (a, (b, c))
unflatr (a, b, c) = (a, (b, c))
  -- | unflat left
unflatl :: (a, b, c) → ((a, b), c)
unflatl (b, c, d) = ((b, c), d)
  -- | pair with nil
pwnil :: a → (a, ())   -- pwnil means 'pair with nil'
pwnil =< id, ! >
  -- | coswap
coswap :: a + b → b + a
coswap = [i2, i1]
  -- | coassoc right
coassocr :: (a + b) + c → a + (b + c)
coassocr = [id + i1, i2 · i2]
```

```
  -- * More funtions
  -- | maybe 2 Either
maybe2either :: Maybe a → () + a
maybe2either Nothing = i1 ()
maybe2either (Just v) = i2 v
  -- | either 2 maybe
either2maybe :: () + a → Maybe a
either2maybe (i1 ()) = Nothing
either2maybe (i2 a) = Just a
  -- | Binding to either2maybe
e2m :: () + a → Maybe a
e2m = either2maybe
  -- | Binding to maybe2either
m2e :: Maybe a → () + a
m2e = maybe2either


  -- * Algebras and Co-Algebras
  -- | out
outL :: [a] → Maybe (a, [a])
outL [] = Nothing
outL (h : t) = Just (h, t)
  -- | catamorphism
cataL :: (Maybe (a, c) → c) → [a] → c
cataL g = g · e2m · (id + id × cataL g) · m2e · outL
  -- | in
inL :: Maybe (a, [a]) → [a]
inL Nothing = []
inL (Just v) = uncurry (:) v
  -- | anamorphism
anaL :: (c → Maybe (a, c)) → c → [a]
anaL g = inL · e2m · (id + id × anaL g) · m2e · g
  -- | hylomorphism
hyloL :: (Maybe (c, b) → b) → (a → Maybe (c, a)) → a → b
hyloL g h = g · e2m · (id + id × hyloL g h) · m2e · h


  -- | mapping
mapL :: (a → b) → [a] → [b]
mapL f = cataL (maybe [] (uncurry (:) · (f × id)))
  -- | concat
concatL :: [[a]] → [a]
concatL = cataL (maybe [] (uncurry (++)))
  -- | reverse
reverseL :: [a] → [a]
reverseL = cataL (maybe [] (uncurry (++) · swap · ((:[]) · id × id)))


  -- * Algebras and Co-Algebras
  -- | out
outL1 :: [a] → Maybe (a + (a, [a]))
outL1 [] = Nothing
outL1 [x] = Just (i1 x)
outL1 (h : t) = Just (i2 (h, t))
  -- | catamorphism
```

```
cataL1 :: (Maybe (a + (a, c)) → c) → [a] → c
cataL1 g = g · e2m · (id + (id + id × cataL1 g)) · m2e · outL1
  -- | in
inL1 :: Maybe (a + (a, [a])) → [a]
inL1 Nothing = []
inL1 (Just v) = [:[], uncurry (:)] v
  -- | anamorphism
anaL1 :: (c → Maybe (a + (a, c))) → c → [a]
anaL1 g = inL1 · e2m · (id + (id + id × anaL1 g)) · m2e · g
  -- | hylomorphism
hyloL1 :: (Maybe (d + (c, b)) → b) → (a → Maybe (d + (c, a))) → a → b
hyloL1 g h = g · e2m · (id + (id + id × hyloL1 g h)) · m2e · h
```

2.2 Base definition's

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- This module implements common types
module Music.Analysis.Base where
import Data.Char (String)
import Data.Bool (Bool)
import Data.Int (Int)
import Data.Maybe (Maybe (..))
import Data.Ratio (Ratio, (%))
import Prelude (Double, truncate)
```

This types are simple redefinitions. We explicit Double as Number. This Number is a Real Number. Delta is a number that means increment/decrement. Also explicit String as Text.

This wrapper get Integer number from Real Number. Invariant class allows check properties to datatypes.

```
  -- * Types
  -- | Number is Double
type Number = Double
  -- | Delta type is a number
type Delta = Number
  -- | Text is String
type Text = String
  -- | Integer Number definition
type IntegerNumber = Int
  -- | Ratio Number definition
type RatioNumber = Ratio IntegerNumber
  -- | wrapper to get Integer Number
toInteger :: Number → IntegerNumber
toInteger = truncate
  -- | wrapper to get Ratio Number
toRatio :: IntegerNumber → RatioNumber
toRatio i = i % 1
  -- | Invariant class specification
```

```
class Invariant a where
  invariant :: a → Bool
```

We also presents utilitary functions, such zip and unzip in Maybe version.

```
  -- * Auxiliary functions
  -- | like @unzip@
unzipMaybe :: [Maybe (a, b)] → ([Maybe a], [Maybe b])
unzipMaybe [] = ([], [])
unzipMaybe (Just (a1, a2) : as) =
  let (x, y) = unzipMaybe as in (Just a1 : x, Just a2 : y)
unzipMaybe (Nothing : as) =
  let (x, y) = unzipMaybe as in (Nothing : x, Nothing : y)
  -- | like @zip@
zipMaybe :: ([Maybe a], [Maybe b]) → [Maybe (a, b)]
zipMaybe ([], []) = []
zipMaybe (Just x : xs, Just y : ys) = Just (x, y) : zipMaybe (xs, ys)
zipMaybe (Nothing : xs, Nothing : ys) = Nothing : zipMaybe (xs, ys)
zipMaybe _ = []
```

3 Abstract

3.1 Settings


Settings are defined using mapping from text to value. This value can be text or number.

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- This module implements configurations
module Music.Analysis.Abstract.Settings where
import Music.Analysis.PF (< ·, · >, π1, π2, grd)
import Music.Analysis.Base (Text, Number)
import Data.Map (· ↦ ·, empty, lookup, alter, union, fromList, map, unionWith)
  -- foldWithKey, filterWithKey)
import Data.Maybe (Maybe (..), maybe)
import Data.Either ((..) + ·, [·, ·])
import Data.Function ((·), id, ., flip)
  -- import Data.List ((++))
import Data.Bool (Bool (..))
  -- import Data.Eq (Eq(..))
import Prelude (Double)
```

Settings is a mapping from keys to values with additional information of keys. This additional information is very helpful to disambiguate merging settings when some settings are more priority than others.

```
  -- | Definition of Settings to make general configurations
  -- It is possible grow
type Settings = Text ↦ (Bool, Text + Number)
  -- (Map Text (Either Text Number), [Text])
```

These functions lets new settings from scratch. text and number functions are wrapper to settings value. These last two functions are helpful at fromList uses, because hide Either datatype.

```
  -- * Building Settings
  -- | empty configurations
empty :: Settings
empty = Data.Map.empty
  -- | fromList
fromList :: [(Text, (Bool, Text + Number))] → Settings
fromList = Data.Map.fromList
  -- | wrapper to settings
text' :: Text → (Bool, Text + Number)
text' t = (True, i1 t)
text :: Text → Bool → (Bool, Text + Number)
text t =< id, (i1 t) >
  -- |wrapper to settings
number' :: Number → (Bool, Text + Number)
number' n = (True, i2 n)
number :: Number → Bool → (Bool, Text + Number)
number n =< id, (i2 n) >
  -- |
priority :: Bool
priority = True
```

We present raw access functions. These functions, such as, addSettings and getSettings are wrapper to function on mapping. These functions allow manipulation over Settings like add new setting or get value from previous added key.

```
  -- * Directly access
  -- | Get value from configuration
getSettings :: Text → Settings → Maybe (Text + Number)
getSettings t = lookup t · Data.Map.map π2
  -- maybe Nothing (Just . p2) . lookup t
  -- |
changeSettings' :: Text → (Bool, Text + Number) → Settings → Settings
changeSettings' k v = alter ((_ · Just) v) k
changeSettings :: Text → (Bool → (Bool, Text + Number)) → Settings → Settings
changeSettings k v = alter (Just · maybe (v False) (v · π1)) k
```

We recommend these functions instead raw access functions due to easy use. These functions explicit type of values and are designed to Point-Free approach.

```
  -- * Easy access
  -- | get Text value from Configurations
getText :: Text → Settings → Maybe Text
getText k = maybe Nothing ([Just, Nothing]) · getSettings k
  -- | get Number value from Configurations
getNumber :: Text → Settings → Maybe Number
getNumber k = maybe Nothing ([Nothing, Just]) · getSettings k
  -- | Change Text
changeText :: Text → Text → Settings → Settings
changeText k v = changeSettings k (text v)
  -- | Change Number
changeNumber :: Text → Number → Settings → Settings
changeNumber k v = changeSettings k (number v)
```

Joining and splitting settings are crucial operations at next modules to mix settings. This function doesn't work with keys that it have more priority than others.

```
  -- * Merge and splitting
  -- | Union
union :: Settings → Settings → Settings
union a b = Data.Map.union a b
union1 :: Settings → Settings → Settings
union1 = Data.Map.unionWith ([_,flip _] · grd π1)
  -- if p1 then const else const) a b
```

3.2 Motive

 This module implements generic Motive. This specifications fits Melodic Motive, Rhythm Motive and so on.

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- This module implements a generic Motive
module Music.Analysis.Abstract.Motive where
import Music.Analysis.PF ((×), cataL, mapL, π1, π2, < ·, · >,
  anaL, grd, (+), e2m)
import Music.Analysis.Abstract.Settings (Settings, union, empty)
import Data.Tuple (curry, uncurry)
import Data.List (zip, head, tail)
import Data.Function ((·), id, _)
import Data.Maybe (maybe)
import Data.Eq (Eq (..))
import Prelude (Show, Read)
```

Basic definition is product between Settings and Sequence of nodes. These nodes contains music information.

```
  -- | Motive Definition
data Motive a = Motive (Settings, [a])
  deriving (Eq, Show, Read)
  -- | make new motive
mkMotive :: Settings → [a] → Motive a
mkMotive = curry Motive
  -- | get Internal Motive representation
fromMotive :: Motive a → (Settings, [a])
fromMotive (Motive x) = x
  -- | get Motive from internal representation
toMotive :: (Settings, [a]) → Motive a
toMotive = Motive
```

Function meta is responsible for update Settings, nodes will be same. Next functions, like cataMotive and mapMotive are catamorphism and mapping applied to Motive data type. These functions doesn't change metadata at Settings.

```
  -- * Combinators
meta :: (Settings → Settings) → Motive a → Motive a
meta f = toMotive · (f × id) · fromMotive
  -- | General cata
cataMotive :: b → (Settings → (a, b) → b) → Motive a → (Settings, b)
```

```
cataMotive z f =< π1, λ(s, x) → cataL (maybe z (f s)) x > ·fromMotive
  -- | General map
mapMotive :: (Settings → a → b) → Motive a → Motive b
mapMotive f = toMotive · (< π1, λ(s, x) → mapL (f s) x >) · fromMotive
```

These functions are used to reusing functions.

```
  -- | join Pair of Motive into Motive of Pair
joinMotivePair :: (Motive a, Motive b) → Motive (a, b)
joinMotivePair =
  toMotive ·
  (uncurry union × uncurry zip) ·
   < π1 × π1, π2 × π2 > ·
  (fromMotive × fromMotive)
  -- | Split Motive of Pair into Pair of Motive
splitMotivePair :: Motive (a, b) → (Motive a, Motive b)
splitMotivePair =< mapMotive π1, mapMotive π2 >
  -- | join List of Motive into Motive of List
joinMotiveList :: [Motive a] → Motive [a]
joinMotiveList =
  toMotive ·
  cataL (maybe (empty, []) (λ((a, b), (c, d)) → (a `union` c, b : d))) ·
  mapL fromMotive
  -- | split Motive of List into List of Motive
splitMotiveList :: Eq a ⇒ Motive [a] → [Motive a]
splitMotiveList =
  mapL toMotive ·
  anaL (e2m · (())+ < id × head, id × tail >) · grd ((≡ []) · π2)) ·
  fromMotive
```

3.3 Melodic

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- This module implements Melodic Motive
module Music.Analysis.Abstract.Melodic where
import Music.Analysis.Base (Number, Text, Delta,
  zipMaybe, unzipMaybe, toInteger, IntegerNumber)
import Music.Analysis.Abstract.Settings (Settings,
  getNumber, fromList, number, text, changeText, priority)
import Music.Analysis.PF ((×), grd, swap, (+), π1, < ·, · >,
  assocl, mapL, cataL, e2m, hyloL)
import Music.Analysis.Abstract.Motive (Motive, toMotive, fromMotive,
  mapMotive, cataMotive, meta)
import Data.Maybe (Maybe (..), maybe, fromJust, isNothing)
import Data.Either ([·, ·])
import Data.Function (id, ., (·))
import Data.Tuple (uncurry)
import Data.List ((++), (!!), init, tail, last)
import Data.Bool (otherwise, Bool (..), (∧))
```

```
import Data.Ord (Ord (..))
import Data.Eq (Eq (..))
import Data.Fixed (div', mod')
import Prelude (Num (..), mod, Show, Read)
```

Accident is defined as number, however Natural isn't supported. Alternative to support Natural is defined using `Maybe Number`. Remember that Accident will be number of half-tones from base note. Settings are fundamental, in our model. Settings change some operations results. Mandatory settings are:

ClefName by default is text "ClefG",

ClefNumber by default is number 2,

Key by default is number 0,

Mode by default is text "Major",

Octave by default is number 0.

Optional settings can be Type to expression motive type, such as, Relative, 7-Absolute, 12-Absolute or Alpha. Some operations expect specific Type, but doesn't test Type.

```
  -- | Melodic node
type Pitch = Number
type MelodicNode = Maybe (Delta, Accident)
type MelodicRelative = Maybe (Delta, Accident)
type MelodicAbsolute = Maybe (Pitch, Accident)
type MelodicClass = Maybe (PitchClass, Accident)
  -- | Accident is defined as number /provisional/.
  -- It doesn't support natural (only supports flats and sharps)
  -- To supports sharps, flats and natural, it will be @Maybe Number@
  -- This number is number of half-tones.
type Accident = Maybe Number
  -- |
data AccidentClass = DoubleSharp
   | Sharp
   | Natural
   | Flat
   | DoubleFlat
   | UnknowAccident Text
  deriving (Eq, Show)
  -- | Pitch Class definition
data PitchClass = C   -- C
   | D   -- D
   | E   -- E
   | F   -- F
   | G   -- G
   | A   -- A
   | B   -- B
  deriving (Eq, Show, Read)
  -- | MelodicNode with PitchClass
type MelodicClassNode = Maybe (PitchClass, Accident)
  -- | default settings
settings :: Settings
settings = fromList [
```

```
  ("ClefName",    text "ClefG" priority),
  ("ClefNumber", number 2 priority),
  ("Key",           number 0 priority),
  ("Mode",         text "Major" priority),
  ("Octave",       number 0 priority)]
```

Rest cab ne used as combinator, like maybe combinator. This combinator has same behaviour that maybe combinator.

```
  -- * Combinators
  -- | rest combinator
rest :: b → ((Delta, Accident) → b) → MelodicNode → b
rest = maybe
  -- * Auxiliary functions
  -- | Default rest
mkRest :: MelodicNode
mkRest = Nothing
  -- | default non-rest
mkNoRest :: MelodicNode
mkNoRest = Just (0, Nothing)
```

We presents simple functions that are reused at next stages.

```
  -- | Transforms 7-number into Char notation
pitch :: (Number, Accident) → (IntegerNumber, (PitchClass, Accident))
pitch (x, y) =
  (div' (toInteger x) 7, ([C, D, E, F, G, A, B] !! mod' (toInteger x) 7, pitch_alter y))
  where pitch_alter :: Accident → Accident
    pitch_alter = id
  -- | Transforms 7-number into Char notation
pitch' :: (Number, Accident) → (PitchClass, Accident)
pitch' (x, y) = ([C, D, E, F, G, A, B] !! mod (toInteger x) 7, pitch_alter' y)
  where pitch_alter' :: Accident → Accident
    pitch_alter' = id
  -- | Transforms Char into 7-Number notation
absPitch :: (PitchClass, Accident) → (Number, Accident)
absPitch = absPitch_class × absPitch_alter
  where absPitch_class :: PitchClass → Number
    absPitch_class pc = case pc of
      C → 0; D → 1; E → 2; F → 3; G → 4; A → 5; B → 6;
    absPitch_alter :: Accident → Accident
    absPitch_alter = id
  -- | Transpose to 12 level
transpose12 :: Number → MelodicAbsolute → MelodicAbsolute
transpose12 n = rest Nothing (Just · ((n+) × id))
  -- | transforms 7 level to 12-level notation
f7to12 :: (Number, Accident) → (Number, Accident)
f7to12 (n, ac) | n < 0 = let (x, y) = f7to12 (n + 7, ac) in (x − 12, y)
  | n > 6 = let (x, y) = f7to12 (n − 7, ac) in (x + 12, y)
  | otherwise = case n of
    _ → ([0, 2, 4, 5, 7, 9, 11] !! (toInteger n), ac)
  -- | transforms 12-level into 7-level notation
f12to7 :: (Number, Accident) → (Number, Accident)
f12to7 (n, Nothing) = f12to7 (n, Just 0)
f12to7 (n, ac@(Just a)) | n < 0 = let (x, y) = f12to7 (n + 12, ac) in (x − 7, y)
```

```
  | n > 11 = let (x, y) = f12to7 (n − 12, ac) in (x + 7, y)
  | otherwise = case n of
      0 → (0, ac); 2 → (1, ac); 4 → (2, ac);
      5 → (3, ac); 7 → (4, ac); 9 → (5, ac); 11 → (6, ac);
      _ →
        if a > 0
        then let (x, y) = f12to7 (n − 1, ac) in
          case y of
            (Just y′) → (x, Just (y′ + 1))
            Nothing → (x, Just 1)
        else let (x, y) = f12to7 (n + 1, ac) in
          case y of
            (Just y′) → (x, Just (y′ − 1))
            Nothing → (x, Just (−1))
```

This stage presents some combinators, like transpose.

```
  -- | Transposes over 7 absolute level
transpose :: Number → Motive MelodicAbsolute → Motive MelodicAbsolute
transpose n =
  mapMotive ([id, Just · f12to7 · ((n+) × id) · f7to12 · fromJust] · grd isNothing)
  -- | Reverse
reverse :: Motive MelodicNode → Motive MelodicNode
reverse =
  toMotive · cataMotive [] (uncurry (++) · swap · ((:[]) × id))
  -- | symmetric melodic
symmetric :: Number → Motive MelodicAbsolute → Motive MelodicAbsolute
symmetric n =
  mapMotive (rest Nothing (Just · ((n+) · ((−) n) × id)))
```

We expect that Type of input Motive Melodic to

toAlpha is 7-Absolute and result will be Alpha,

fromAlpha is Alpha and result will be 7-Absolute,

to12 is 7-Absolute and result will be 12-Absolute,

from12 is 12-Absolute and result will be 7-Absolute,

Function toAlpha' is deprecated, because isn't so expressive as toAlpha.

```
  -- | Convert to alpha notation
toAlpha :: Motive MelodicAbsolute →
  Motive (Maybe (IntegerNumber, (PitchClass, Accident)))
toAlpha = meta (changeText "Type" "Alpha") ·
  mapMotive (maybe Nothing (Just · pitch))
  -- | Convert to alpha notation
toAlpha′ :: Motive MelodicNode → Motive MelodicClassNode
toAlpha′ = meta (changeText "Type" "Alpha") ·
  mapMotive (rest Nothing (Just · pitch′))
  -- | get 7-Absolute music from alpha notation
fromAlpha :: Motive MelodicClassNode → Motive MelodicAbsolute
fromAlpha = meta (changeText "Type" "7-Absolute") ·
  mapMotive (maybe Nothing (Just · absPitch))
  -- | Convert 7-absolute into 12-absolute notation
```

```
to12 :: Motive MelodicAbsolute → Motive MelodicAbsolute
to12 = meta (changeText "Type" "12-Absolute") ·
  mapMotive (rest Nothing (Just · f7to12))
  -- | Convert 12-absolute into 7-absolute notation
from12 :: Motive MelodicAbsolute → Motive MelodicAbsolute
from12 = meta (changeText "Type" "7-Absolute") ·
  mapMotive (rest Nothing (Just · f12to7))
```

relative function isn't PF-approach, but works. Absolute functionat that computes absolute motive from relative is written using Point-Free approach. To get Absolute is built list of all initials Melodics and sum it.

```
  -- | relative melodic.
relative :: Motive MelodicAbsolute → Motive MelodicRelative
relative =
  toMotive ·
  < changeText "Type" "Relative" · π1,
      λ(m1, m2) → aux1 (maybe 0 id (getNumber "Key" m1)) m2 > ·
  fromMotive
  where aux1 :: Number → [MelodicNode] → [MelodicNode]
        aux1 _ [] = []
        aux1 acc (x : l) =
          maybe Nothing (Just · ((λa → a − acc) × id)) x :
          aux1 (maybe acc π1 x) l
  -- | absolute PF
absolute :: Motive MelodicRelative → Motive MelodicAbsolute
absolute = toMotive ·
  < changeText "Type" "7-Absolute" · π1,
    zipMaybe ·
      (tail ·
        mapL ([last, Just · sumRelatives] · grd guard) ·
        hyloL reverses getInits ·    -- get inits and reverse reversed result
        uncurry (:)   -- append key
        × id) ·       -- doesn't change alter
    (assocl · (id × unzipMaybe)) ·
    (maybe (Just 0) Just · getNumber "Key" × id) >    -- get key
  · fromMotive
  where
    -- | condition
    guard :: [Maybe Delta] → Bool
    guard l = l ≢ [] ∧ last l ≡ Nothing
    -- | sum with Maybe
    sumRelatives :: [Maybe Delta] → Number
    sumRelatives = cataL (maybe 0 (uncurry (+) · (maybe 0 id × id)))
    -- | get inits
    getInits :: [Maybe Number] → Maybe ([Maybe Number], [Maybe Number])
    getInits = e2m · (()+ < id, init >) · grd (≡ [])
    -- | reverse
    reverses :: Maybe ([Maybe Number], [[Maybe Number]]) → [[Maybe Number]]
    reverses = maybe [] (uncurry (++) · swap · ((:[]) × id))
```

3.4 Rhythm

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- This module implements Rhythm Motive
module Music.Analysis.Abstract.Rhythm where
import Music.Analysis.PF ((×), swap, π2, < ·, · >, π1, grd, (+), assocl,
  mapL, cataL, e2m, hyloL)
import Music.Analysis.Abstract.Settings (Settings, getNumber, changeText,
  fromList, number, priority)
import Music.Analysis.Abstract.Motive (Motive, fromMotive, toMotive,
  mapMotive, cataMotive)
import Music.Analysis.Base (Number, Delta, Text,
  IntegerNumber, RatioNumber, toRatio, toInteger)
import Data.Maybe (maybe)
import Data.Function (id, ∴, (·))
import Data.Ord (Ord (..))
import Data.Eq (Eq (..))
import Data.Tuple (uncurry)
import Data.List ((++), zip, tail, init, unzip)
import Prelude (Num (..), (/), Show (..), (↑), read)
```

RhythmNode is defined as pair of Delta, variation on durations, and number of dots. Mandatory settings are:

TempoPitch by default is number 4,

TempoNumber by default is number 60,

CompassUp by default is number 4

CompassDown by default is number 4.

```
  -- * Types
  -- | Rhythm node
type RhythmNode = (Delta, Dots)
  -- type RhythmAbsolute = (Number, Dots)
  -- type RhythmRelative = (Delta, Dots)
type RhythmAbsolute = (RatioNumber, Dots)
type RhythmRelative = (RatioNumber, Dots)
  -- | Dots is defined by number.
  -- Only Integers and positive numbers are allowed.
type Dots = IntegerNumber
type Duration = Number
data DurationClass = Whole
   | Half
   | Quarter
   | Eighth
   | Th16
   | Th32
   | Th64
   | UnkownDuration Text
  deriving (Eq, Show)
  -- | sefault settings
settings :: Settings
```

```
settings = fromList [
  ("TempoPitch", number 4   priority),
  ("TempoNumber", number 60 priority),
  ("CompassUp",  number 4   priority),
  ("CompassDown", number 4  priority)]


  -- * Auxiliary functions
  -- | computes duration /PW/
durationNode :: RhythmNode → Number
durationNode =
  hyloL
      (maybe 0 (uncurry (+)))
      (e2m · (()+ < π1, (/2) × pred >) · grd ((⩽ 0) · π2)) ·
    (id × succ)
      where pred x = x − 1
            succ x = x + 1
  -- | computes compass duration
compass :: (Number, Number) → Number
compass (x, y) = x / y
```

These functions are combinators over Rhythm.

```
  -- | changes duration
tempo :: RatioNumber → Motive RhythmAbsolute → Motive RhythmAbsolute
tempo n = mapMotive ((∗n) × id)
  -- | computes duration
duration :: Motive RhythmNode → Number
duration = π2 ·
  cataMotive 0 (uncurry (+) · (durationNode × id))
  -- | reverse
reverse :: Motive RhythmNode → Motive RhythmNode
reverse = toMotive ·
  cataMotive [] (uncurry (++) · swap · ((:[]) × id))
  -- | symmetric trsnformation
symmetric :: RatioNumber → Motive RhythmAbsolute → Motive RhythmAbsolute
symmetric n = mapMotive ((n+) · (/n) × id)
```

Relative function assume that Motive have Absolute Type, and change it into Relative Type. Absolute function is computed using Point-Free approach. Actually, it built initial lists from nodes and reverse reversed result, summing it. Attention: Relative and absolute functions use wrong initial number from settings.

```
  -- | Computes relative Rhythm
relative :: Motive RhythmAbsolute → Motive RhythmRelative
relative = toMotive ·
   < changeText "Type" "Relative" · π1,
     mapL (< uncurry (/) · (π1 × π1), π2 · π1 >) ·
     uncurry zip · (π2 × uncurry (:) · (initial × id)) ·
      < id, id > ·
     (maybe (toRatio 4) (toRatio · toInteger) ·
       getNumber "CompassDown" × id) >
  · fromMotive
  where initial :: RatioNumber → RhythmAbsolute
        initial =< id, 0 >
```

```
  -- | Absolute PF
absolute :: Motive RhythmRelative → Motive RhythmAbsolute
absolute = toMotive ·
   < changeText "Type" "Absolute" · π1,
     uncurry zip · (tail ·
       (mapL (cataL (maybe (toRatio 1) (uncurry (*)))) ·
       (hyloL (maybe [] (uncurry (++) · swap · ((:[]) × id)))
          (e2m · ((_)+ < id, init >) · grd (≡ [])) ) ·
       (uncurry (:))
        × id) ·
     (assocl · (id × unzip)) ·
     (maybe (toRatio 4) (toRatio · toInteger) · getNumber "CompassDown" × id) >
     · fromMotive


durationNumber :: DurationClass → Duration
durationNumber Whole = 4
durationNumber Half  = 2
durationNumber Quarter = 1
durationNumber Eighth = 0.5
durationNumber Th16 = 0.25
durationNumber Th32 = 0.125
durationNumber Th64 = 0.0625
durationNumber (UnkownDuration _) = 1


  -- |
durationTotalNumber :: (DurationClass, Dots) → Duration
durationTotalNumber =
  hyloL
       (maybe 0 (uncurry (+)))
       (e2m · ((_)+ < π1, (/2) × pred >) · grd ((⩽ 0) · π2)) ·
    (durationNumber × succ)
      where pred x = x − 1
            succ x = x + 1
  -- |
getDurationClass :: (Duration, Dots) → DurationClass
getDurationClass (4, _) = Whole
getDurationClass (2, _) = Half
getDurationClass (1, _) = Quarter
getDurationClass (0.5, _) = Eighth
getDurationClass (0.25, _) = Th16
getDurationClass (0.125, _) = Th32
getDurationClass (0.0625, _) = Th64
getDurationClass (b, 0) = UnkownDuration (show b)
getDurationClass (t, i) = getDurationClass (((2 ↑ i) / ((2 ↑ (i + 1)) − 1)) ∗ t, 0)
  -- |
getDuration :: (DurationClass, Dots) → Duration
getDuration =
  hyloL
       (maybe 0 (uncurry (+)))
       (e2m · ((_)+ < π1, (/2) × pred >) · grd ((⩽ 0) · π2)) ·
    (durationNumber × succ)
      where pred x = x − 1
```

```
        succ x = x + 1
  -- |
getDuration_aux1 :: DurationClass → Duration
getDuration_aux1 Whole = 4
getDuration_aux1 Half = 2
getDuration_aux1 Quarter = 1
getDuration_aux1 Eighth = 0.5
getDuration_aux1 Th16 = 0.25
getDuration_aux1 Th32 = 0.125
getDuration_aux1 Th64 = 0.0625
getDuration_aux1 (UnkownDuration x) = read x
```

3.5 Zip

Zip module is result of merging Melodic and Rhyhm modules.

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- This module join Melodic and Rhythm
module Music.Analysis.Abstract.Zip where
import Music.Analysis.PF ((×), π2)
import Music.Analysis.Base (Number, toRatio, toInteger)
import Music.Analysis.Abstract.Settings (Settings, union)
import Music.Analysis.Abstract.Motive
import Music.Analysis.Abstract.Melodic as Melodic
import Music.Analysis.Abstract.Rhythm as Rhythm
import Data.Function ((·), id)
import Prelude ()
```

At this stage will be join pieces, like melodic and rhythm.

```
  -- * Types
  -- | VoiceZipNode definition
type VoiceZipNode = (MelodicNode, RhythmNode)
type VoiceZipAbsolute = (MelodicAbsolute, RhythmAbsolute)
type VoiceZipRelative = (MelodicRelative, RhythmRelative)
  -- | default settings
settings :: Settings
settings = Melodic.settings `union` Rhythm.settings


  -- | transposes
transpose :: Number → Motive VoiceZipAbsolute → Motive VoiceZipAbsolute
transpose n =
  joinMotivePair · (Melodic.transpose n × id) · splitMotivePair
  -- | changes duration
tempo :: Number → Motive VoiceZipAbsolute → Motive VoiceZipAbsolute
tempo n = joinMotivePair ·
  (id × (Rhythm.tempo · toRatio · toInteger) n) ·
  splitMotivePair
  -- | computes duration
duration :: Motive VoiceZipNode → Number
```

```
duration = Rhythm.duration · π2 · splitMotivePair
  -- | reverse
reverse :: Motive VoiceZipNode → Motive VoiceZipNode
reverse =
  joinMotivePair · (Melodic.reverse × Rhythm.reverse) · splitMotivePair
  -- | relative
relative :: Motive VoiceZipAbsolute → Motive VoiceZipRelative
relative =
  joinMotivePair · (Melodic.relative × Rhythm.relative) · splitMotivePair
  -- | absolute
absolute :: Motive VoiceZipRelative → Motive VoiceZipAbsolute
absolute =
  joinMotivePair · (Melodic.absolute × Rhythm.absolute) · splitMotivePair
```

3.6 Notations

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- This module implements specific music notation
module Music.Analysis.Abstract.Notations where
import Music.Analysis.PF ((×), π1)
import Music.Analysis.Base (Number, Text, Invariant (..))
import Music.Analysis.Abstract.Settings (Settings)
import Music.Analysis.Abstract.Motive
import Music.Analysis.Abstract.Zip as Zip
import Data.Ord (Ord (..))
import Data.Bool (Bool (..))
import Data.Function (id, (·))
import Data.Either ((..) + ·)
import Data.Maybe (Maybe (..))
import Prelude ()
```

This are defined settings to Motive NotationNode, that will be equal to Zip settings.

```
  -- | Info
type NotationInfo = Text + Number
  -- | New Notation Node
type NotationNode = [(NotationPosition, NotationInfo)]   -- for each note
  -- | New Notation Position
type NotationPosition = Maybe Position   -- Nothing -> Point; Just -> Interval (more x)
  -- | Position
type Position = Number
  -- | sefault settings
settings :: Settings
settings = Zip.settings
instance Invariant NotationPosition where
  invariant (Just n) | n < 0 = False
  invariant _ = True


  -- |
```

```
addNotation :: NotationPosition → NotationInfo → NotationNode → NotationNode
addNotation a b = (:) (a, b)
```

Next functions are combinators like transpose.

```
  -- | transpose using above layers
transpose :: Number → Motive (VoiceZipAbsolute, NotationNode) →
  Motive (VoiceZipAbsolute, NotationNode)
transpose n = joinMotivePair ·
  ((Zip.transpose n) × id) ·
  splitMotivePair
  -- | tempo transformation using above layers
tempo :: Number → Motive (VoiceZipAbsolute, NotationNode) →
  Motive (VoiceZipAbsolute, NotationNode)
tempo n =
  joinMotivePair · ((Zip.tempo n) × id) · splitMotivePair
  -- | duration computation using above layers
duration :: Motive (VoiceZipNode, NotationNode) → Number
duration = Zip.duration · π1 · splitMotivePair
  -- | reverse using above layers
reverse :: Motive (VoiceZipNode, NotationNode) →
  Motive (VoiceZipNode, NotationNode)
reverse =
  joinMotivePair · (Zip.reverse × id) · splitMotivePair
  -- | relative transformation using above layers
relative :: Motive (VoiceZipAbsolute, NotationNode) →
  Motive (VoiceZipRelative, NotationNode)
relative =
  joinMotivePair · (Zip.relative × id) · splitMotivePair
  -- | absolute transformation using above layers
absolute :: Motive (VoiceZipRelative, NotationNode) →
  Motive (VoiceZipAbsolute, NotationNode)
absolute =
  joinMotivePair · (Zip.absolute × id) · splitMotivePair
```

3.7 Voices

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- This module implements multiple voices
module Music.Analysis.Abstract.Voices where
import Music.Analysis.PF ((×), < ·, · >, π1, π2)
import Music.Analysis.Base (Number, IntegerNumber)
import Music.Analysis.Abstract.Settings (Settings)
import Music.Analysis.Abstract.Motive
import Music.Analysis.Abstract.Zip
import Music.Analysis.Abstract.Notations as Notations
import Data.Function (id, (·), ˙)
import Data.Tuple (uncurry)
import Prelude ()
```

```
  -- * Types
  -- |
type MultiVoiceNode = ((VoiceZipNode, IntegerNumber), NotationNode)
type MultiVoiceAbsolute = ((VoiceZipAbsolute, IntegerNumber), NotationNode)
type MultiVoiceRelative = ((VoiceZipRelative, IntegerNumber), NotationNode)
  -- | default settings
settings :: Settings
settings = Notations.settings


joinVoices :: IntegerNumber → (a, NotationNode) →
  ((a, IntegerNumber), NotationNode)
joinVoices b =<< π1, b >, π2 >
splitVoices :: ((a, IntegerNumber), NotationNode) →
  (IntegerNumber, (a, NotationNode))
splitVoices =< π2 · π1, < π1 · π1, π2 >>
```

This are defined some combinators, like are transpose. These functions are defined using functions from Notations module.

```
  -- | Transpose using above layers
transpose :: Number → Motive MultiVoiceAbsolute → Motive MultiVoiceAbsolute
transpose n =
  mapMotive (uncurry joinVoices) ·
    joinMotivePair · (id × Notations.transpose n) · splitMotivePair ·
  mapMotive splitVoices
  -- | tempo transformation using above layers
tempo :: Number → Motive MultiVoiceAbsolute → Motive MultiVoiceAbsolute
tempo n =
  mapMotive (uncurry joinVoices) ·
    joinMotivePair · (id × Notations.tempo n) · splitMotivePair ·
  mapMotive splitVoices
  -- | duration computation using above layers
duration :: Motive MultiVoiceNode → Number
duration =
  Notations.duration · π2 · splitMotivePair · mapMotive splitVoices
  -- | reverse using above layers
reverse :: Motive MultiVoiceNode → Motive MultiVoiceNode
reverse =
  mapMotive (uncurry joinVoices) ·
    joinMotivePair · (id × Notations.reverse) ·
    splitMotivePair ·
  mapMotive splitVoices
  -- | absolute transformation using above layers
absolute :: Motive MultiVoiceRelative → Motive MultiVoiceAbsolute
absolute =
  mapMotive (uncurry joinVoices) ·
    joinMotivePair · (id × Notations.absolute) · splitMotivePair ·
  mapMotive splitVoices
  -- | relative transformation using above layers
relative :: Motive MultiVoiceAbsolute → Motive MultiVoiceRelative
relative =
  mapMotive (uncurry joinVoices) ·
```

```
    joinMotivePair · (id × Notations.relative) · splitMotivePair ·
  mapMotive splitVoices
```

3.8 Instruments

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- This module implements multiple instruments
module Music.Analysis.Abstract.Instruments where
import Music.Analysis.Base (Number)
import Music.Analysis.PF (mapL)
import Music.Analysis.Abstract.Settings (Settings,
  union, fromList, text, priority)
import Music.Analysis.Abstract.Motive
import Music.Analysis.Abstract.Voices as Voices
import Data.Function ((·))
import Prelude ()
```

This are defined default settings.

```
  -- * Types
type MultiInstrumentNode = [MultiVoiceNode]
type MultiInstrumentAbsolute = [MultiVoiceAbsolute]
type MultiInstrumentRelative = [MultiVoiceRelative]
  -- | default settings
settings :: Settings
settings = Voices.settings `union`
  fromList [("InstrumentName", text "Piano" priority)]


  -- | Transpose using above layers
transpose :: Number → Motive MultiInstrumentAbsolute →
  Motive MultiInstrumentAbsolute
transpose n = joinMotiveList · mapL (Voices.transpose n) · splitMotiveList
  -- | tempo transformation using above layers
tempo :: Number → Motive MultiInstrumentAbsolute →
  Motive MultiInstrumentAbsolute
tempo n = joinMotiveList · mapL (Voices.tempo n) · splitMotiveList
  -- | duration computation using above layers
duration :: Motive MultiInstrumentNode → [Number]
duration = mapL Voices.duration · splitMotiveList
  -- | reverse using above layers
reverse :: Motive MultiInstrumentNode → Motive MultiInstrumentNode
reverse = joinMotiveList · mapL Voices.reverse · splitMotiveList
  -- | absolute transformation using above layers
absolute :: Motive MultiInstrumentRelative → Motive MultiInstrumentAbsolute
absolute = joinMotiveList · mapL Voices.absolute · splitMotiveList
  -- | relative transformation using above layers
relative :: Motive MultiInstrumentAbsolute → Motive MultiInstrumentRelative
relative = joinMotiveList · mapL Voices.relative · splitMotiveList
```

3.9 Annotation

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- This module implements annotation over music notation
module Music.Analysis.Abstract.Annotation where
import Music.Analysis.PF ((×), π1)
import Music.Analysis.Base (Text, Number)
import Music.Analysis.Abstract.Settings (Settings)
import Music.Analysis.Abstract.Motive
import Music.Analysis.Abstract.Melodic
import Music.Analysis.Abstract.Rhythm
import Music.Analysis.Abstract.Notations
import Music.Analysis.Abstract.Instruments as Instruments
import Data.Maybe (Maybe)
import Data.Function (id, (·))
import Data.Bool (Bool)
import Prelude ()
```

Development of Recursive and powerfull Annotation. It still at alpha version.

```
  -- * Types
  -- |
type A = Text
  -- |
type Annot =
  ([((((MelodicNode, A), (RhythmNode, A)), Bool),
  [((NotationPosition, NotationInfo), A)])], A)
  -- |
type AnnotationNode = [(Maybe Number, Text)]
  -- | Definition of annotation
type MultiAnnotationNode = (MultiInstrumentNode, AnnotationNode)
type AnnotationAbsolute = (MultiInstrumentAbsolute, AnnotationNode)
type AnnotationRelative = (MultiInstrumentRelative, AnnotationNode)
  -- | sefault settings
settings :: Settings
settings = Instruments.settings


  -- | Transpose using above layers
transpose :: Number → Motive AnnotationAbsolute → Motive AnnotationAbsolute
transpose n =
  joinMotivePair · ((Instruments.transpose n) × id) · splitMotivePair
  -- | tempo transformation using above layers
tempo :: Number → Motive AnnotationAbsolute → Motive AnnotationAbsolute
tempo n = joinMotivePair · ((Instruments.tempo n) × id) · splitMotivePair
  -- | duration computation using above layers
duration :: Motive MultiAnnotationNode → [Number]
duration = Instruments.duration · π1 · splitMotivePair
  -- | reverse using above layers
reverse :: Motive MultiAnnotationNode → Motive MultiAnnotationNode
```

```
reverse = joinMotivePair · (Instruments.reverse × id) · splitMotivePair
  -- | absolute transformation using above layers
absolute :: Motive AnnotationRelative → Motive AnnotationAbsolute
absolute = joinMotivePair · (Instruments.absolute × id) · splitMotivePair
  -- | relative transformation using above layers
relative :: Motive AnnotationAbsolute → Motive AnnotationRelative
relative = joinMotivePair · (Instruments.relative × id) · splitMotivePair
```

4 MusicXML

4.1 MusicXML

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: HaXml
  -- This module make interface with MusicXML using HaXML library.
module Music.Analysis.MusicXML where
import Music.Analysis.Base
import Music.Analysis.PF
import qualified Music.Analysis.MusicXML.Level1 as Layer1
import qualified Music.Analysis.MusicXML.Level2 as Layer2
import qualified Music.Analysis.MusicXML.Level3 as Layer3
import qualified Music.Analysis.MusicXML.Level4 as Layer4
import qualified Music.Analysis.MusicXML.Level5 as Layer5
import qualified Music.Analysis.MusicXML.Level6 as Layer6 ()
import qualified Text.XML.MusicXML as MusicXML
import qualified Text.XML.MusicXML.Partwise as Partwise
import qualified Text.XML.MusicXML.Timewise as Timewise
import Data.Char (isDigit)
import Prelude


  -- |
toTimewise :: Partwise.Score_Partwise → Timewise.Score_Timewise
toTimewise = (id × (id × transpose))
  -- |
toPartwise :: Timewise.Score_Timewise → Partwise.Score_Partwise
toPartwise = (id × (id × transpose))
  -- |
transpose :: [(a, [(b, c)])] → [(b, [(a, c)])]
transpose [] = []
transpose ((_, []) : xss) = transpose xss
transpose ((a, ((b, x) : xs)) : xss) =
  (b, (a, x) : [(a, h) | (_, (_, h) : _) ← xss]) :
  (transpose ((a, xs) : [t | t ← xss]))


  -- |
abst_Score_Partwise :: MusicXML.Score_Partwise → Layer5.Score_Partwise
abst_Score_Partwise = (id × (id × fmap abst_Part))
```

```
  -- |
abst_Part :: Partwise.Part → Layer5.Part
abst_Part = (id × fmap abst_Measure)
  -- |
abst_Measure :: Partwise.Measure → Layer5.Measure
abst_Measure = (id × fmap abst_Music_Data)
  -- |
abst_Music_Data :: MusicXML.Music_Data_ → Layer5.Music_Data
abst_Music_Data (MusicXML.Music_Data_1 x) = Layer5.Music_Data_1 (abst_Note x)
abst_Music_Data (MusicXML.Music_Data_2 x) = Layer5.Music_Data_2 x
abst_Music_Data (MusicXML.Music_Data_3 x) = Layer5.Music_Data_3 x
abst_Music_Data (MusicXML.Music_Data_4 x) = Layer5.Music_Data_4 x
abst_Music_Data (MusicXML.Music_Data_5 x) =
  Layer5.Music_Data_5 (abst_Attributes x)
abst_Music_Data (MusicXML.Music_Data_6 x) = Layer5.Music_Data_6 x
abst_Music_Data (MusicXML.Music_Data_7 x) = Layer5.Music_Data_7 x
abst_Music_Data (MusicXML.Music_Data_8 x) = Layer5.Music_Data_8 x
abst_Music_Data (MusicXML.Music_Data_9 x) = Layer5.Music_Data_9 x
abst_Music_Data (MusicXML.Music_Data_10 x) = Layer5.Music_Data_10 x
abst_Music_Data (MusicXML.Music_Data_11 x) = Layer5.Music_Data_11 x
abst_Music_Data (MusicXML.Music_Data_12 x) = Layer5.Music_Data_12 x
abst_Music_Data (MusicXML.Music_Data_13 x) = Layer5.Music_Data_13 x
  -- |
abst_Note :: MusicXML.Note → Layer5.Note
abst_Note = (id × ((λ(x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13) →
  -- x4' <- fmap abst_Type x4
    (abst_Note_ x1, fmap abst_Instrument x2,
      abst_Editorial_Voice x3,
      (maybe Nothing id · fmap abst_Type) x4, fmap abst_Dot x5,
      (maybe Nothing id · fmap abst_Accidental) x6,
      fmap abst_Time_Modification x7, fmap abst_Stem x8,
      fmap abst_Notehead x9, fmap abst_Staff x10, fmap abst_Beam x11,
      fmap abst_Notations x12, fmap abst_Lyric x13))))
  -- |
abst_Note_ :: MusicXML.Note_ → Layer5.Note_
abst_Note_ (MusicXML.Note_1 (x1, x2, x3)) =
  Layer5.Note_1 (abst_Grace x1, abst_Full_Note x2,
    fmap (abst_Tie × fmap abst_Tie) x3)
abst_Note_ (MusicXML.Note_2 (x1, x2, x3)) =
  Layer5.Note_2 (abst_Cue x1, abst_Full_Note x2, abst_Duration x3)
abst_Note_ (MusicXML.Note_3 (x1, x2, x3)) =
  Layer5.Note_3 (abst_Full_Note x1, abst_Duration x2,
    fmap (abst_Tie × fmap abst_Tie) x3)
  -- |
abst_Grace :: MusicXML.Grace → Layer5.Grace
abst_Grace = id
  -- |
abst_Cue :: MusicXML.Cue → Layer5.Cue
abst_Cue = id
  -- |
abst_Instrument :: MusicXML.Instrument → Layer5.Instrument
abst_Instrument = id
  -- |
abst_Duration :: MusicXML.Duration → Layer5.Duration
```

```
abst_Duration = maybe 1 id · read_IntegerNumber
  -- |
abst_Full_Note :: MusicXML.Full_Note → Layer5.Full_Note
abst_Full_Note = fmap id × abst_Full_Note_
  -- |
abst_Full_Note_ :: MusicXML.Full_Note_ → Layer5.Full_Note_
abst_Full_Note_ (MusicXML.Full_Note_1 x) = Layer5.Full_Note_1 (abst_Pitch x)
abst_Full_Note_ (MusicXML.Full_Note_2 x) = Layer5.Full_Note_2 (abst_Unpitched x)
abst_Full_Note_ (MusicXML.Full_Note_3 x) = Layer5.Full_Note_3 (abst_Rest x)
  -- |
abst_Pitch :: MusicXML.Pitch → Layer2.Pitch
abst_Pitch (a, b, c) =
  ((maybe Layer1.C id · abst_Step) a,
    (maybe Nothing id · fmap abst_Alter) b, abst_Octave c)
  -- |
abst_Step :: MusicXML.Step → Maybe Layer1.Step
abst_Step "A" = Just Layer1.A
abst_Step "B" = Just Layer1.B
abst_Step "C" = Just Layer1.C
abst_Step "D" = Just Layer1.D
abst_Step "E" = Just Layer1.E
abst_Step "F" = Just Layer1.F
abst_Step "G" = Just Layer1.G
abst_Step _ = Nothing
  -- |
abst_Alter :: MusicXML.Alter → Maybe Layer1.Alter
abst_Alter = read_Number
  -- |
abst_Octave :: MusicXML.Octave → Layer1.Octave
abst_Octave = maybe 4 id · read_IntegerNumber
  -- |
abst_Unpitched :: MusicXML.Unpitched → Layer4.Unpitched
abst_Unpitched = id
  -- |
abst_Rest :: MusicXML.Rest → Layer4.Rest
abst_Rest = id
  -- |
abst_Tie :: MusicXML.Tie → Layer5.Tie
abst_Tie = id
  -- |
abst_Editorial_Voice :: MusicXML.Editorial_Voice → Layer5.Editorial_Voice
abst_Editorial_Voice = id
  -- |
abst_Type :: MusicXML.Type → Maybe Layer5.Type
abst_Type (a, b) = do    -- (id >< abst_Type_)
  b' ← abst_Type_ b
  return (a, b')
  -- |
abst_Type_ :: MusicXML.PCDATA → Maybe Layer1.Type_
abst_Type_ "long"    = Just Layer1.Long
abst_Type_ "breve"   = Just Layer1.Breve
abst_Type_ "whole"   = Just Layer1.Whole
abst_Type_ "half"    = Just Layer1.Half
abst_Type_ "quarter" = Just Layer1.Quarter
```

```
abst_Type_ "eighth" = Just Layer1.Eighth
abst_Type_ "16th"   = Just Layer1.Th16
abst_Type_ "32nd"   = Just Layer1.Th32
abst_Type_ "64th"   = Just Layer1.Th64
abst_Type_ "128th"  = Just Layer1.Th128
abst_Type_ "256th"  = Just Layer1.Th256
abst_Type_ _        = Nothing
  -- |
abst_Dot :: MusicXML.Dot → Layer5.Dot
abst_Dot = id
  -- |
abst_Accidental :: MusicXML.Accidental → Maybe Layer5.Accidental
abst_Accidental (a, b) = do
  b' ← abst_Accidental_ b
  return (a, b')
abst_Accidental_ :: MusicXML.PCDATA → Maybe Layer1.Accidental_
abst_Accidental_ "sharp"                = Just Layer1.Sharp
abst_Accidental_ "natural"              = Just Layer1.Natural
abst_Accidental_ "flat"                 = Just Layer1.Flat
abst_Accidental_ "double-sharp"         = Just Layer1.Double_Sharp
abst_Accidental_ "sharp-sharp"          = Just Layer1.Sharp_Sharp
abst_Accidental_ "flat-flat"            = Just Layer1.Flat_Flat
abst_Accidental_ "natural-sharp"        = Just Layer1.Natural_Sharp
abst_Accidental_ "natural-flat"         = Just Layer1.Natural_Flat
abst_Accidental_ "quarter-sharp"        = Just Layer1.Quarter_Sharp
abst_Accidental_ "quarter-flat"         = Just Layer1.Quarter_Flat
abst_Accidental_ "three-quarters-sharp" = Just Layer1.Three_Quarters_Sharp
abst_Accidental_ "three-quarters-flat"  = Just Layer1.Three_Quarters_Flat
abst_Accidental_ _                      = Nothing
  -- |
abst_Time_Modification :: MusicXML.Time_Modification → Layer5.Time_Modification
abst_Time_Modification = id
  -- |
abst_Stem :: MusicXML.Stem → Layer5.Stem
abst_Stem = id
  -- |
abst_Notehead :: MusicXML.Notehead → Layer5.Notehead
abst_Notehead = id
  -- |
abst_Staff :: MusicXML.Staff → Layer5.Staff
abst_Staff = maybe 1 id . read_IntegerNumber
  -- |
abst_Beam :: MusicXML.Beam → Layer5.Beam
abst_Beam = id
  -- |
abst_Notations :: MusicXML.Notations → Layer5.Notations
abst_Notations = id
  -- |
abst_Lyric :: MusicXML.Lyric → Layer5.Lyric
abst_Lyric = id
  -- |
abst_Attributes :: MusicXML.Attributes → Layer5.Attributes
abst_Attributes (x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12) =
```

```
  (abst_Editorial x1, fmap abst_Divisions x2, fmap abst_Key x3,
    fmap abst_Time x4, fmap abst_Staves x5, fmap abst_Part_Symbol x6,
    fmap abst_Instruments x7, fmap abst_Clef x8,
    fmap abst_Staff_Details x9, fmap abst_Transpose x10,
    fmap abst_Directive x11, fmap abst_Measure_Style x12)
  -- |
abst_Editorial :: MusicXML.Editorial → Layer5.Editorial
abst_Editorial = id
  -- |
abst_Divisions :: MusicXML.Divisions → Layer5.Divisions
abst_Divisions = maybe 1 id · read_IntegerNumber
  -- |
abst_Key :: MusicXML.Key → Layer5.Key
abst_Key = id × (abst_Key_ × fmap abst_Key_Octave)
  -- |
abst_Key_ :: MusicXML.Key_ → Layer5.Key_
abst_Key_ (MusicXML.Key_1 (x1, x2, x3)) =
  Layer5.Key_1 (x1, abst_Fifths x2, (maybe Nothing id · fmap abst_Mode) x3)
abst_Key_ (MusicXML.Key_2 x) =
  Layer5.Key_2 (fmap (abst_Key_Step × abst_Key_Alter) x)
  -- |
abst_Fifths :: MusicXML.Fifths → Layer2.Fifths
abst_Fifths = maybe 0 id · read_IntegerNumber
  -- |
abst_Mode :: MusicXML.Mode → Maybe Layer2.Mode
abst_Mode "major"     = Just Layer2.Major
abst_Mode "minor"     = Just Layer2.Minor
abst_Mode "dorian"    = Just Layer2.Dorian
abst_Mode "phrygian"  = Just Layer2.Phrygian
abst_Mode "lydian"    = Just Layer2.Lydian
abst_Mode "mixolydian" = Just Layer2.Mixolydian
abst_Mode "aeolian"   = Just Layer2.Aeolian
abst_Mode "ionian"    = Just Layer2.Ionian
abst_Mode "locrian"   = Just Layer2.Locrian
abst_Mode _           = Nothing
  -- |
abst_Key_Step :: MusicXML.Key_Step → Layer2.Key_Step
abst_Key_Step = maybe Layer1.C id · abst_Step
  -- |
abst_Key_Alter :: MusicXML.Key_Alter → Layer2.Key_Alter
abst_Key_Alter = maybe 0 id · abst_Alter
  -- |
abst_Key_Octave :: MusicXML.Key_Octave → Layer5.Key_Octave
abst_Key_Octave = id × abst_Octave
  -- |
abst_Time :: MusicXML.Time → Layer5.Time
abst_Time = id × abst_Time_B
  -- |
abst_Time_B :: MusicXML.Time_B → Layer3.Time_B
abst_Time_B (MusicXML.Time_5 x) =
  Layer3.Time_5 (fmap (abst_Beats × abst_Beat_Type) x)
abst_Time_B (MusicXML.Time_6 x) = Layer3.Time_6 x
  -- |
abst_Beats :: MusicXML.Beats → Layer3.Beats
```

```
abst_Beats =
  maybe (4, Nothing) id ·
  (λ(a, b) → maybe Nothing (λa′ → Just (a′, b)) a) ·
  (read_IntegerNumber ×
    ([Nothing, read_IntegerNumber · tail] · grd null)) ·
  span (≢ '+')
  -- |
abst_Beat_Type :: MusicXML.Beat_Type → Layer3.Beat_Type
abst_Beat_Type =
  maybe 4 id ·
  maybe Nothing (e2m · (() + id) · grd (<0)) ·
  read_IntegerNumber
  -- |
abst_Staves :: MusicXML.Staves → Layer5.Staves
abst_Staves = id
  -- |
abst_Part_Symbol :: MusicXML.Part_Symbol → Layer5.Part_Symbol
abst_Part_Symbol = id
  -- |
abst_Instruments :: MusicXML.Instruments → Layer5.Instruments
abst_Instruments = id
  -- |
abst_Clef :: MusicXML.Clef → Layer5.Clef
abst_Clef =
  id × (flatl ·
    ((maybe Layer2.Clef_Sign_None id · abst_Sign ×
      fmap abst_Line) × fmap abst_Clef_Octave_Change) ·
    unflatl)
  -- |
abst_Sign :: MusicXML.Sign → Maybe Layer2.Sign
abst_Sign "G"           = Just Layer2.Clef_Sign_G
abst_Sign "F"           = Just Layer2.Clef_Sign_F
abst_Sign "C"           = Just Layer2.Clef_Sign_C
abst_Sign "percussion" = Just Layer2.Clef_Sign_Percussion
abst_Sign "TAB"         = Just Layer2.Clef_Sign_TAB
abst_Sign "none"        = Just Layer2.Clef_Sign_None
abst_Sign _             = Nothing
  -- |
abst_Line :: MusicXML.Line → Layer2.Line
abst_Line = maybe 1 id · read_IntegerNumber
  -- |
abst_Clef_Octave_Change ::
  MusicXML.Clef_Octave_Change → Layer2.Clef_Octave_Change
abst_Clef_Octave_Change = maybe 0 id · read_IntegerNumber
  -- |
abst_Staff_Details :: MusicXML.Staff_Details → Layer5.Staff_Details
abst_Staff_Details = id
  -- |
abst_Transpose :: MusicXML.Transpose → Layer5.Transpose
abst_Transpose = id
  -- |
abst_Directive :: MusicXML.Directive → Layer5.Directive
abst_Directive = id
```

```
  -- |
abst_Measure_Style :: MusicXML.Measure_Style → Layer5.Measure_Style
abst_Measure_Style = id

  -- |
rep_Score_Partwise :: Layer5.Score_Partwise → MusicXML.Score_Partwise
rep_Score_Partwise = (id × (id × fmap rep_Part))
  -- |
rep_Part :: Layer5.Part → Partwise.Part
rep_Part = (id × fmap rep_Measure)
  -- |
rep_Measure :: Layer5.Measure → Partwise.Measure
rep_Measure = (id × fmap rep_Music_Data)
  -- |
rep_Music_Data :: Layer5.Music_Data → MusicXML.Music_Data_
rep_Music_Data (Layer5.Music_Data_1 x) = MusicXML.Music_Data_1 (rep_Note x)
rep_Music_Data (Layer5.Music_Data_2 x) = MusicXML.Music_Data_2 x
rep_Music_Data (Layer5.Music_Data_3 x) = MusicXML.Music_Data_3 x
rep_Music_Data (Layer5.Music_Data_4 x) = MusicXML.Music_Data_4 x
rep_Music_Data (Layer5.Music_Data_5 x) =
  MusicXML.Music_Data_5 (rep_Attributes x)
rep_Music_Data (Layer5.Music_Data_6 x) = MusicXML.Music_Data_6 x
rep_Music_Data (Layer5.Music_Data_7 x) = MusicXML.Music_Data_7 x
rep_Music_Data (Layer5.Music_Data_8 x) = MusicXML.Music_Data_8 x
rep_Music_Data (Layer5.Music_Data_9 x) = MusicXML.Music_Data_9 x
rep_Music_Data (Layer5.Music_Data_10 x) = MusicXML.Music_Data_10 x
rep_Music_Data (Layer5.Music_Data_11 x) = MusicXML.Music_Data_11 x
rep_Music_Data (Layer5.Music_Data_12 x) = MusicXML.Music_Data_12 x
rep_Music_Data (Layer5.Music_Data_13 x) = MusicXML.Music_Data_13 x
  -- |
rep_Note :: Layer5.Note → MusicXML.Note
rep_Note = (id × ((λ(x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13) →
  -- x4' <- fmap abst_Type x4
    (rep_Note_ x1, fmap rep_Instrument x2,
      rep_Editorial_Voice x3,
      (fmap rep_Type) x4, fmap rep_Dot x5,
      (fmap rep_Accidental) x6,
      fmap rep_Time_Modification x7, fmap rep_Stem x8,
      fmap rep_Notehead x9, fmap rep_Staff x10, fmap rep_Beam x11,
      fmap rep_Notations x12, fmap rep_Lyric x13))))
  -- |
rep_Note_ :: Layer5.Note_ → MusicXML.Note_
rep_Note_ (Layer5.Note_1 (x1, x2, x3)) =
  MusicXML.Note_1 (rep_Grace x1, rep_Full_Note x2,
    fmap (rep_Tie × fmap rep_Tie) x3)
rep_Note_ (Layer5.Note_2 (x1, x2, x3)) =
  MusicXML.Note_2 (rep_Cue x1, rep_Full_Note x2, rep_Duration x3)
rep_Note_ (Layer5.Note_3 (x1, x2, x3)) =
  MusicXML.Note_3 (rep_Full_Note x1, rep_Duration x2,
    fmap (rep_Tie × fmap rep_Tie) x3)
  -- |
rep_Grace :: Layer5.Grace → MusicXML.Grace
rep_Grace = id
  -- |
```

```
rep_Cue :: Layer5.Cue → MusicXML.Cue
rep_Cue = id
  -- |
rep_Instrument :: Layer5.Instrument → MusicXML.Instrument
rep_Instrument = id
  -- |
rep_Duration :: Layer5.Duration → MusicXML.Duration
rep_Duration = show
  -- |
rep_Full_Note :: Layer5.Full_Note → MusicXML.Full_Note
rep_Full_Note = fmap id × rep_Full_Note_
  -- |
rep_Full_Note_ :: Layer5.Full_Note_ → MusicXML.Full_Note_
rep_Full_Note_ (Layer5.Full_Note_1 x) = MusicXML.Full_Note_1 (rep_Pitch x)
rep_Full_Note_ (Layer5.Full_Note_2 x) = MusicXML.Full_Note_2 (rep_Unpitched x)
rep_Full_Note_ (Layer5.Full_Note_3 x) = MusicXML.Full_Note_3 (rep_Rest x)
  -- |
rep_Pitch :: Layer2.Pitch → MusicXML.Pitch
rep_Pitch (a, b, c) =
  (rep_Step a, fmap rep_Alter b, rep_Octave c)
  -- |
rep_Step :: Layer1.Step → MusicXML.Step
rep_Step Layer1.A = "A"
rep_Step Layer1.B = "B"
rep_Step Layer1.C = "C"
rep_Step Layer1.D = "D"
rep_Step Layer1.E = "E"
rep_Step Layer1.F = "F"
rep_Step Layer1.G = "G"
  -- |
rep_Alter :: Layer1.Alter → MusicXML.Alter
rep_Alter = show
  -- |
rep_Octave :: Layer1.Octave → MusicXML.Octave
rep_Octave = show
  -- |
rep_Unpitched :: Layer4.Unpitched → MusicXML.Unpitched
rep_Unpitched = id
  -- |
rep_Rest :: Layer4.Rest → MusicXML.Rest
rep_Rest = id
  -- |
rep_Tie :: Layer5.Tie → MusicXML.Tie
rep_Tie = id
  -- |
rep_Editorial_Voice :: MusicXML.Editorial_Voice → Layer5.Editorial_Voice
rep_Editorial_Voice = id
  -- |
rep_Type :: Layer5.Type → MusicXML.Type
rep_Type = id × rep_Type_
  -- |
rep_Type_ :: Layer1.Type_ → MusicXML.PCDATA
rep_Type_ Layer1.Long    = "long"
rep_Type_ Layer1.Breve   = "breve"
```

```
rep_Type_ Layer1.Whole      = "whole"
rep_Type_ Layer1.Half       = "half"
rep_Type_ Layer1.Quarter    = "quarter"
rep_Type_ Layer1.Eighth     = "eighth"
rep_Type_ Layer1.Th16       = "16th"
rep_Type_ Layer1.Th32       = "32nd"
rep_Type_ Layer1.Th64       = "64th"
rep_Type_ Layer1.Th128      = "128th"
rep_Type_ Layer1.Th256      = "256th"
  -- |
rep_Dot :: Layer5.Dot → MusicXML.Dot
rep_Dot = id
  -- |
rep_Accidental :: Layer5.Accidental → MusicXML.Accidental
rep_Accidental = id × rep_Accidental_
  -- |
rep_Accidental_ :: Layer1.Accidental_ → MusicXML.PCDATA
rep_Accidental_ Layer1.Sharp                = "sharp"
rep_Accidental_ Layer1.Natural              = "natural"
rep_Accidental_ Layer1.Flat                 = "flat"
rep_Accidental_ Layer1.Double_Sharp         = "double-sharp"
rep_Accidental_ Layer1.Sharp_Sharp          = "sharp-sharp"
rep_Accidental_ Layer1.Flat_Flat            = "flat-flat"
rep_Accidental_ Layer1.Natural_Sharp        = "natural-sharp"
rep_Accidental_ Layer1.Natural_Flat         = "natural-flat"
rep_Accidental_ Layer1.Quarter_Sharp        = "quarter-sharp"
rep_Accidental_ Layer1.Quarter_Flat         = "quarter-flat"
rep_Accidental_ Layer1.Three_Quarters_Sharp = "three-quarters-sharp"
rep_Accidental_ Layer1.Three_Quarters_Flat = "three-quarters-flat"
  -- |
rep_Time_Modification :: MusicXML.Time_Modification → Layer5.Time_Modification
rep_Time_Modification = id
  -- |
rep_Stem :: Layer5.Stem → MusicXML.Stem
rep_Stem = id
  -- |
rep_Notehead :: Layer5.Notehead → MusicXML.Notehead
rep_Notehead = id
  -- |
rep_Staff :: Layer5.Staff → MusicXML.Staff
rep_Staff = show
  -- |
rep_Beam :: Layer5.Beam → MusicXML.Beam
rep_Beam = id
  -- |
rep_Notations :: Layer5.Notations → MusicXML.Notations
rep_Notations = id
  -- |
rep_Lyric :: Layer5.Lyric → MusicXML.Lyric
rep_Lyric = id
  -- |
rep_Attributes :: Layer5.Attributes → MusicXML.Attributes
rep_Attributes (x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12) =
```

```
  (rep_Editorial x1, fmap rep_Divisions x2, fmap rep_Key x3,
    fmap rep_Time x4, fmap rep_Staves x5, fmap rep_Part_Symbol x6,
    fmap rep_Instruments x7, fmap rep_Clef x8,
    fmap rep_Staff_Details x9, fmap rep_Transpose x10,
    fmap rep_Directive x11, fmap rep_Measure_Style x12)
  -- |
rep_Editorial :: Layer5.Editorial → MusicXML.Editorial
rep_Editorial = id
  -- |
rep_Divisions :: Layer5.Divisions → MusicXML.Divisions
rep_Divisions = show
  -- |
rep_Key :: Layer5.Key → MusicXML.Key
rep_Key = id × (rep_Key_ × fmap rep_Key_Octave)
  -- |
rep_Key_ :: Layer5.Key_ → MusicXML.Key_
rep_Key_ (Layer5.Key_1 (x1, x2, x3)) =
  MusicXML.Key_1 (x1, rep_Fifths x2, fmap rep_Mode x3)
rep_Key_ (Layer5.Key_2 x) =
  MusicXML.Key_2 (fmap (rep_Key_Step × rep_Key_Alter) x)
  -- |
rep_Fifths :: Layer2.Fifths → MusicXML.Fifths
rep_Fifths = show
  -- |
rep_Mode :: Layer2.Mode → MusicXML.Mode
rep_Mode Layer2.Major      = "major"
rep_Mode Layer2.Minor      = "minor"
rep_Mode Layer2.Dorian     = "dorian"
rep_Mode Layer2.Phrygian   = "phrygian"
rep_Mode Layer2.Lydian     = "lydian"
rep_Mode Layer2.Mixolydian = "mixolydian"
rep_Mode Layer2.Aeolian    = "aeolian"
rep_Mode Layer2.Ionian     = "ionian"
rep_Mode Layer2.Locrian    = "locrian"
  -- |
rep_Key_Step :: Layer2.Key_Step → MusicXML.Key_Step
rep_Key_Step = rep_Step
  -- |
rep_Key_Alter :: Layer2.Key_Alter → MusicXML.Key_Alter
rep_Key_Alter = rep_Alter
  -- |
rep_Key_Octave :: Layer5.Key_Octave → MusicXML.Key_Octave
rep_Key_Octave = id × rep_Octave
  -- |
rep_Time :: Layer5.Time → MusicXML.Time
rep_Time = id × rep_Time_B
  -- |
rep_Time_B :: Layer3.Time_B → MusicXML.Time_B
rep_Time_B (Layer3.Time_5 x) =
  MusicXML.Time_5 (fmap (rep_Beats × rep_Beat_Type) x)
rep_Time_B (Layer3.Time_6 x) = MusicXML.Time_6 x
  -- |
rep_Beats :: Layer3.Beats → MusicXML.Beats
rep_Beats = uncurry (++) · swap · (show × maybe "" ((++"+") · show))
```

```
  -- maybe (4, Nothing) id .
  -- (\(a,b) -> maybe Nothing (\a' -> Just (a',b)) a) .
  -- (read_IntegerNumber ><
  -- (either (const Nothing) (read_IntegerNumber . tail) . grd null)) .
  -- span (/='+')
  -- |
rep_Beat_Type :: Layer3.Beat_Type → MusicXML.Beat_Type
rep_Beat_Type = show
  -- maybe 4 id .
  -- maybe Nothing (e2m . (const () -|- id) . grd (<0)) .
  -- read_IntegerNumber
  -- |
rep_Staves :: Layer5.Staves → MusicXML.Staves
rep_Staves = id
  -- |
rep_Part_Symbol :: Layer5.Part_Symbol → MusicXML.Part_Symbol
rep_Part_Symbol = id
  -- |
rep_Instruments :: Layer5.Instruments → MusicXML.Instruments
rep_Instruments = id
  -- |
rep_Clef :: Layer5.Clef → MusicXML.Clef
rep_Clef =
  id × (flatl ·
    ((rep_Sign × fmap rep_Line) × fmap rep_Clef_Octave_Change) ·
    unflatl)
  -- |
rep_Sign :: Layer2.Sign → MusicXML.Sign
rep_Sign Layer2.Clef_Sign_G     = "G"
rep_Sign Layer2.Clef_Sign_F     = "F"
rep_Sign Layer2.Clef_Sign_C     = "C"
rep_Sign Layer2.Clef_Sign_Percussion = "percussion"
rep_Sign Layer2.Clef_Sign_TAB = "TAB"
rep_Sign Layer2.Clef_Sign_None = "none"
  -- |
rep_Line :: Layer2.Line → MusicXML.Line
rep_Line = show
  -- |
rep_Clef_Octave_Change ::
  Layer2.Clef_Octave_Change → MusicXML.Clef_Octave_Change
rep_Clef_Octave_Change = show
  -- |
rep_Staff_Details :: Layer5.Staff_Details → MusicXML.Staff_Details
rep_Staff_Details = id
  -- |
rep_Transpose :: Layer5.Transpose → MusicXML.Transpose
rep_Transpose = id
  -- |
rep_Directive :: Layer5.Directive → MusicXML.Directive
rep_Directive = id
  -- |
rep_Measure_Style :: Layer5.Measure_Style → MusicXML.Measure_Style
rep_Measure_Style = id
```

```
dur_Duration :: String → IntegerNumber
dur_Duration = maybe 0 id · read_IntegerNumber
dur_Backup :: MusicXML.Backup → IntegerNumber → IntegerNumber
dur_Backup (x, _) = λy → y − (dur_Duration x)
dur_Forward :: MusicXML.Forward → IntegerNumber → IntegerNumber
dur_Forward (x, _, _) = λy → y + (dur_Duration x)
dur_Note_ :: MusicXML.Note_ → IntegerNumber → IntegerNumber
dur_Note_ (MusicXML.Note_1 _) = id
dur_Note_ (MusicXML.Note_2 (_, _, x)) = λy → y + dur_Duration x
dur_Note_ (MusicXML.Note_3 (_, x, _)) = λy → y + dur_Duration x
dur_Note :: MusicXML.Note → IntegerNumber → IntegerNumber
dur_Note (_, (x, _, _, _, _, _, _, _, _, _, _, _, _)) = dur_Note_ x
dur_Attributes :: MusicXML.Attributes → Maybe MusicXML.Divisions
dur_Attributes (_, x, _, _, _, _, _, _, _, _, _) = dur_Divisions x
dur_Divisions :: Maybe a → Maybe a
dur_Divisions (Nothing) = Nothing
dur_Divisions (Just x) = Just x
dur_Music_Data_ :: MusicXML.Music_Data_ →
  (MusicXML.Divisions, IntegerNumber) → (MusicXML.Divisions, IntegerNumber)
dur_Music_Data_ (MusicXML.Music_Data_1 x) (a, b) = (a, dur_Note x b)
dur_Music_Data_ (MusicXML.Music_Data_2 x) (a, b) = (a, dur_Backup x b)
dur_Music_Data_ (MusicXML.Music_Data_3 x) (a, b) = (a, dur_Forward x b)
dur_Music_Data_ (MusicXML.Music_Data_5 x) (a, b) = (maybe a id (dur_Attributes x), b)
dur_Music_Data_ (_) (a, b) = (a, b)


  -- |
map_Score_Partwise' :: (MusicXML.Music_Data_ → MusicXML.Music_Data_) →
  MusicXML.Score_Partwise → MusicXML.Score_Partwise
map_Score_Partwise' f = (id × (id × fmap (map_Part' f)))
  -- |
map_Part' :: (MusicXML.Music_Data_ → MusicXML.Music_Data_) →
  Partwise.Part → Partwise.Part
map_Part' f = (id × fmap (map_Measure' f))
  -- |
map_Measure' :: (MusicXML.Music_Data_ → MusicXML.Music_Data_) →
  Partwise.Measure → Partwise.Measure
map_Measure' f = (id × fmap (map_Music_Data' f))
  -- |
map_Music_Data' :: (MusicXML.Music_Data_ → MusicXML.Music_Data_) →
  MusicXML.Music_Data_ → MusicXML.Music_Data_
map_Music_Data' f = f
  -- map_Music_Data (MusicXML.Music_Data_1 x) = MusicXML.Music_Data_1 x
  -- map_Music_Data (MusicXML.Music_Data_2 x) = MusicXML.Music_Data_2 x
  -- map_Music_Data (MusicXML.Music_Data_3 x) = MusicXML.Music_Data_3 x
  -- map_Music_Data (MusicXML.Music_Data_4 x) = MusicXML.Music_Data_4 x
  -- map_Music_Data (MusicXML.Music_Data_5 x) = MusicXML.Music_Data_5 x
  -- map_Music_Data (MusicXML.Music_Data_6 x) = MusicXML.Music_Data_6 x
  -- map_Music_Data (MusicXML.Music_Data_7 x) = MusicXML.Music_Data_7 x
  -- map_Music_Data (MusicXML.Music_Data_8 x) = MusicXML.Music_Data_8 x
  -- map_Music_Data (MusicXML.Music_Data_9 x) = MusicXML.Music_Data_9 x
  -- map_Music_Data (MusicXML.Music_Data_10 x) = MusicXML.Music_Data_10 x
  -- map_Music_Data (MusicXML.Music_Data_11 x) = MusicXML.Music_Data_11 x
```

```
  -- map_Music_Data (MusicXML.Music_Data_12 x) = MusicXML.Music_Data_12 x
  -- map_Music_Data (MusicXML.Music_Data_13 x) = MusicXML.Music_Data_13 x


  -- |
read_Number :: String → Maybe Number
read_Number =
  e2m · (() + read) ·
  grd (λx → null x ∨ ¬ (all isDigit x))
  -- |
read_IntegerNumber :: String → Maybe IntegerNumber
read_IntegerNumber =
  e2m · (() + read) ·
  grd (λx → null x ∨ ¬ (all isDigit x))
  -- |
coread_Number :: String → Number + String
coread_Number =
  coswap · (id + read) · grd (λx → null x ∨ ¬ (all isDigit x))
  -- |
coread_IntegerNumber :: String → IntegerNumber + String
coread_IntegerNumber =
  coswap · (id + read) · grd (λx → null x ∨ ¬ (all isDigit x))
```

4.2 Functions

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- This module implements functions on MusicXML format
module Music.Analysis.MusicXML.Functions where
import Music.Analysis.PF
import qualified Text.XML.MusicXML as MusicXML


  -- |
isNote :: MusicXML.Music_Data_ → Bool
isNote (MusicXML.Music_Data_1 _) = True
isNote _ = False
  -- |
isGraceNote :: MusicXML.Note → Bool
isGraceNote (_, (MusicXML.Note_1 _, _, _, _, _, _, _, _, _, _, _, _, _)) = True
isGraceNote _ = False
  -- |
isCueNote :: MusicXML.Note → Bool
isCueNote (_, (MusicXML.Note_2 _, _, _, _, _, _, _, _, _, _, _, _, _)) = True
isCueNote _ = False
  -- |
isNormalNote :: MusicXML.Note → Bool
isNormalNote (_, (MusicXML.Note_3 _, _, _, _, _, _, _, _, _, _, _, _, _)) = True
isNormalNote _ = False
```

```
  -- |
filterNote :: MusicXML.Music_Data → MusicXML.Music_Data
filterNote = filter isNote
  -- |
filterNotNote :: MusicXML.Music_Data → MusicXML.Music_Data
filterNotNote = filter (¬ · isNote)
  -- |
filterGraceNote :: MusicXML.Music_Data → MusicXML.Music_Data
filterGraceNote =
  filter (λx → if isNote x
    then (isGraceNote · (λ(MusicXML.Music_Data_1 y) → y)) x
    else True)
  -- |
filterNotGraceNote :: MusicXML.Music_Data → MusicXML.Music_Data
filterNotGraceNote =
  filter (λx → if isNote x
    then (¬ · isGraceNote · (λ(MusicXML.Music_Data_1 y) → y)) x
    else True)
  -- |
filterCueNote :: MusicXML.Music_Data → MusicXML.Music_Data
filterCueNote =
  filter (λx → if isNote x
    then (isCueNote · (λ(MusicXML.Music_Data_1 y) → y)) x
    else True)
  -- |
filterNotCueNote :: MusicXML.Music_Data → MusicXML.Music_Data
filterNotCueNote =
  filter (λx → if isNote x
    then (¬ · isCueNote · (λ(MusicXML.Music_Data_1 y) → y)) x
    else True)
  -- |
filterNormalNote :: MusicXML.Music_Data → MusicXML.Music_Data
filterNormalNote =
  filter (λx → if isNote x
    then (isNormalNote · (λ(MusicXML.Music_Data_1 y) → y)) x
    else True)
  -- |
filterNotNormalNote :: MusicXML.Music_Data → MusicXML.Music_Data
filterNotNormalNote =
  filter (λx → if isNote x
    then (¬ · isNormalNote · (λ(MusicXML.Music_Data_1 y) → y)) x
    else True)


  -- |
filterNote' :: (MusicXML.Note → Bool) →
  MusicXML.Music_Data → MusicXML.Music_Data
filterNote' p =
  filter (λx → if isNote x
    then (p · (λ(MusicXML.Music_Data_1 y) → y)) x
    else False)
  -- |
count_part :: MusicXML.MusicXMLDoc → Int
```

```
count_part (MusicXML.Score (MusicXML.Partwise x)) =
  (length · π2 · π2) x
count_part (MusicXML.Score (MusicXML.Timewise x)) =
  (sum · fmap length · fmap π2 · π2 · π2) x
count_part _ = 0
  -- |
count_measure :: MusicXML.MusicXMLDoc → Int
count_measure (MusicXML.Score (MusicXML.Partwise x)) =
  (sum · fmap length · fmap π2 · π2 · π2) x
count_measure (MusicXML.Score (MusicXML.Timewise x)) =
  (length · π2 · π2) x
count_measure _ = 0
  -- |
count_music_data :: MusicXML.MusicXMLDoc → Int
count_music_data (MusicXML.Score (MusicXML.Partwise x)) =
  (sum · fmap sum · (fmap · fmap) (length · π2) · fmap π2 · π2 · π2) x
count_music_data (MusicXML.Score (MusicXML.Timewise x)) =
  (sum · fmap sum · (fmap · fmap) (length · π2) · fmap π2 · π2 · π2) x
count_music_data _ = 0
  -- |
count_note :: MusicXML.MusicXMLDoc → Int
count_note (MusicXML.Score (MusicXML.Partwise x)) =
  (sum · fmap sum · (fmap · fmap) (length · filterNote · π2) · fmap π2 · π2 · π2) x
count_note (MusicXML.Score (MusicXML.Timewise x)) =
  (sum · fmap sum · (fmap · fmap) (length · filterNote · π2) · fmap π2 · π2 · π2) x
count_note _ = 0
  -- |
count_note_grace :: MusicXML.MusicXMLDoc → Int
count_note_grace (MusicXML.Score (MusicXML.Partwise x)) =
  (sum · fmap sum · (fmap · fmap) (length · filterNote′ isGraceNote · π2) ·
    fmap π2 · π2 · π2) x
count_note_grace (MusicXML.Score (MusicXML.Timewise x)) =
  (sum · fmap sum · (fmap · fmap) (length · filterNote′ isGraceNote · π2) ·
    fmap π2 · π2 · π2) x
count_note_grace _ = 0
  -- |
count_note_cue :: MusicXML.MusicXMLDoc → Int
count_note_cue (MusicXML.Score (MusicXML.Partwise x)) =
  (sum · fmap sum · (fmap · fmap) (length · filterNote′ isCueNote · π2) ·
    fmap π2 · π2 · π2) x
count_note_cue (MusicXML.Score (MusicXML.Timewise x)) =
  (sum · fmap sum · (fmap · fmap) (length · filterNote′ isCueNote · π2) ·
    fmap π2 · π2 · π2) x
count_note_cue _ = 0
  -- |
count_note_normal :: MusicXML.MusicXMLDoc → Int
count_note_normal (MusicXML.Score (MusicXML.Partwise x)) =
  (sum · fmap sum · (fmap · fmap) (length · filterNote′ isNormalNote · π2) ·
    fmap π2 · π2 · π2) x
count_note_normal (MusicXML.Score (MusicXML.Timewise x)) =
  (sum · fmap sum · (fmap · fmap) (length · filterNote′ isNormalNote · π2) ·
    fmap π2 · π2 · π2) x
count_note_normal _ = 0
```

```
  -- |
mapMusicXML :: (MusicXML.Music_Data → MusicXML.Music_Data) →
  MusicXML.MusicXMLDoc → MusicXML.MusicXMLDoc
mapMusicXML f (MusicXML.Score (MusicXML.Partwise x)) =
  (MusicXML.Score · MusicXML.Partwise ·
    (id × (id × fmap (id × fmap (id × f))))) x
mapMusicXML f (MusicXML.Score (MusicXML.Timewise x)) =
  (MusicXML.Score · MusicXML.Timewise ·
    (id × (id × fmap (id × fmap (id × f))))) x
mapMusicXML _ x = x
```

4.3 Layer1

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  --
module Music.Analysis.MusicXML.Level1 (
  module Music.Analysis.MusicXML.Level1,
  ) where
import Music.Analysis.Base


  -- |
type Score_Partwise = [Music_Data]
  -- |
data Music_Data =
  Music_Data_1 Note
  deriving (Eq, Show)
  -- |
type Note = (Note_, Maybe Type, [Dot], Maybe Accidental)
  -- |
data Note_ =
  Note_3 Full_Note
  deriving (Eq, Show)
  -- |
type Full_Note = Full_Note_
  -- |
data Full_Note_ = Full_Note_1 Pitch
   | Full_Note_3 Rest
     deriving (Eq, Show)
  -- |
type Pitch = (Step, Maybe Alter, Octave)
  -- |
data Step = C | D | E | F | G | A | B
  deriving (Eq, Show, Ord, Enum)
  -- |
type Alter = Number
  -- |
type Octave = IntegerNumber
```

```
  -- |
type Rest = ()
  -- |
type Type = Type_
  -- |
data Type_ = Long | Breve |
  Whole | Half | Quarter | Eighth |
  Th16 | Th32 | Th64 | Th128 | Th256
  deriving (Eq, Show, Ord, Enum)
  -- |
type Dot = ()
  -- |
type Accidental = Accidental_
  -- |
data Accidental_ =
  Sharp | Natural | Flat |
  Double_Sharp | Sharp_Sharp | Flat_Flat |
  Natural_Sharp | Natural_Flat |
  Quarter_Sharp | Quarter_Flat |
  Three_Quarters_Sharp | Three_Quarters_Flat
  deriving (Eq, Show, Ord, Enum)
```

4.4 Layer2

```
module Music.Analysis.MusicXML.Level2 (
  module Music.Analysis.MusicXML.Level2,
  ) where
import qualified Music.Analysis.MusicXML.Level1 as Layer1
import Music.Analysis.PF
import Music.Analysis.Base (IntegerNumber)
import Data.Maybe (catMaybes)
import Prelude


  -- |
type Score_Partwise = [Measure]
  -- |
type Measure = [Music_Data]
  -- |
data Music_Data =
    Music_Data_1 Note
  | Music_Data_5 Attributes
    deriving (Eq, Show)
  -- |
type Note = (Note_, Maybe Type, [Dot], Maybe Accidental)
  -- |
data Note_ =
  Note_3 (Full_Note, Duration)
  deriving (Eq, Show)
  -- |
type Full_Note = Full_Note_
  -- |
```

```
data Full_Note_ = Full_Note_1 Pitch
   | Full_Note_3 Layer1.Rest
     deriving (Eq, Show)
  -- |
type Pitch = (Layer1.Step, Maybe Layer1.Alter, Layer1.Octave)
  -- |
type Duration = IntegerNumber
  -- |
type Type = Layer1.Type_
  -- |
type Dot = ()
  -- |
type Accidental = Layer1.Accidental_
  -- |
type Attributes = (Maybe Divisions, [Key], [Time], [Clef])
  -- |
type Divisions = IntegerNumber
  -- |
type Key = (Key_, [Key_Octave])
  -- |
data Key_ = Key_1 (Fifths, Maybe Mode)
   | Key_2 [(Key_Step, Key_Alter)]
     deriving (Eq, Show)
  -- |
type Fifths = IntegerNumber
  -- |
data Mode = Major | Minor |
  Dorian | Phrygian | Lydian | Mixolydian |
  Aeolian | Ionian | Locrian
  deriving (Eq, Show)
  -- |
type Key_Step = Layer1.Step
  -- |
type Key_Alter = Layer1.Alter
  -- |
type Key_Octave = Layer1.Octave
  -- |
type Time = Time_B
  -- |
data Time_B = Time_5 [(Beats, Beat_Type)]
  deriving (Eq, Show)
  -- | MusicXML Schema specify 'xs:string'
type Beats = (IntegerNumber, Maybe IntegerNumber)
  -- | MusicXML Schema specify 'xs:string'
type Beat_Type = IntegerNumber
  -- |
type Clef = (Sign, Maybe Line, Maybe Clef_Octave_Change)
  -- |
data Sign =
  Clef_Sign_G | Clef_Sign_F | Clef_Sign_C |
  Clef_Sign_Percussion | Clef_Sign_TAB |
  Clef_Sign_None
  deriving (Eq, Show, Enum)
  -- |
```

```
type Line = IntegerNumber
  -- |
type Clef_Octave_Change = IntegerNumber

  -- |
abst_Score_Partwise :: Score_Partwise → Layer1.Score_Partwise
abst_Score_Partwise = catMaybes · fmap abst_Music_Data · concat
  -- (id -|- fmap abst_Measure . head) . grd null . p2 . p2
  -- |
  -- abst_Measure :: Measure -> Layer1.Measure
  -- abst_Measure = catMaybes . fmap abst_Music_Data
  -- |
abst_Music_Data :: Music_Data → Maybe Layer1.Music_Data
abst_Music_Data (Music_Data_1 x) = Just (Layer1.Music_Data_1 (abst_Note x))
  -- x' <- abst_Note x
  -- return (Layer1.Music_Data_1 x')
  -- Layer1.Music_Data_1 ((catMaybes . (fmap abst_Note)) x)
  -- abst_Music_Data (Music_Data_2 _) = Nothing
  -- abst_Music_Data (Music_Data_3 _) = Nothing
  -- abst_Music_Data (Music_Data_4 _) = Nothing
abst_Music_Data (Music_Data_5 _) = Nothing
  -- Just (Layer1.Music_Data_5 (abst_Attributes x))
  -- abst_Music_Data (Music_Data_6 _) = Nothing
  -- abst_Music_Data (Music_Data_7 _) = Nothing
  -- abst_Music_Data (Music_Data_8 _) = Nothing
  -- abst_Music_Data (Music_Data_9 _) = Nothing
  -- abst_Music_Data (Music_Data_10 _) = Nothing
  -- abst_Music_Data (Music_Data_11 _) = Nothing
  -- abst_Music_Data (Music_Data_12 _) = Nothing
  -- abst_Music_Data (Music_Data_13 _) = Nothing
  -- |
abst_Note :: Note → Layer1.Note
abst_Note =
  (λ(x1, x4, x5, x6) →
    (abst_Note_ x1, fmap abst_Type x4,
      fmap abst_Dot x5, fmap abst_Accidental x6))
  -- |
abst_Note_ :: Note_ → Layer1.Note_
  -- abst_Note_ (Note_1 _) = Nothing
  -- Layer1.Note_1 (abst_Grace x1, abst_Full_Note x2,
  -- fmap (abst_Tie >< fmap abst_Tie) x3)
  -- abst_Note_ (Note_2 _) = Nothing
  -- Layer1.Note_2 (abst_Cue x1, abst_Full_Note x2, abst_Duration x3)
abst_Note_ (Note_3 (x1, _)) = Layer1.Note_3 (abst_Full_Note x1)
  -- x1' <- abst_Full_Note x1
  -- return (Layer1.Note_3 (x1', abst_Duration x2))
  -- |
abst_Full_Note :: Full_Note → Layer1.Full_Note
abst_Full_Note = abst_Full_Note_
  -- b' <- abst_Full_Note_ b
  -- return (fmap id a, b')
  -- abst_Full_Note :: Full_Note -> (Maybe Layer1.Chord, Maybe Layer1.Full_Note_)
  -- abst_Full_Note = (fmap id >< abst_Full_Note_)
  -- abst_Full_Note (x,y) = do
```

```
  -- y' <- abst_Full_Note_ y
  -- return (fmap id x >< fmap id y'))
  -- (fmap id >< (\x -> do
  -- y <- abst_Full_Note_ x
  -- ))
  -- |
abst_Full_Note_ :: Full_Note_ → Layer1.Full_Note_
abst_Full_Note_ (Full_Note_1 x) = (Layer1.Full_Note_1 (abst_Pitch x))
  -- abst_Full_Note_ (Full_Note_2 _) = Nothing
abst_Full_Note_ (Full_Note_3 x) = (Layer1.Full_Note_3 x)
  -- |
abst_Pitch :: Pitch → Layer1.Pitch
abst_Pitch = id
  -- |
abst_Type :: Type → Layer1.Type
abst_Type = id
  -- |
abst_Dot :: Dot → Layer1.Dot
abst_Dot = id
  -- |
abst_Accidental :: Accidental → Layer1.Accidental
abst_Accidental = id


  -- |
split_Measure :: Measure → ((), [Maybe Layer1.Music_Data])
split_Measure =< (), fmap (π2 · split_Music_Data) >
  -- catMaybes . fmap abst_Music_Data . concat
  -- (id -|- fmap abst_Measure . head) . grd null . p2 . p2
  -- |
  -- abst_Measure :: Measure -> Layer1.Measure
  -- abst_Measure = catMaybes . fmap abst_Music_Data
  -- |
split_Music_Data :: Music_Data → (Music_Data, Maybe Layer1.Music_Data)
  -- Maybe Layer1.Music_Data
split_Music_Data (Music_Data_1 x) =
   < Music_Data_1, Just · Layer1.Music_Data_1 · π2 · split_Note > x
  -- x' <- abst_Note x
  -- return (Layer1.Music_Data_1 x')
  -- Layer1.Music_Data_1 ((catMaybes . (fmap abst_Note)) x)
  -- abst_Music_Data (Music_Data_2 _) = Nothing
  -- abst_Music_Data (Music_Data_3 _) = Nothing
  -- abst_Music_Data (Music_Data_4 _) = Nothing
split_Music_Data (Music_Data_5 x) =< Music_Data_5, Nothing > x
  -- Just (Layer1.Music_Data_5 (abst_Attributes x))
  -- abst_Music_Data (Music_Data_6 _) = Nothing
  -- abst_Music_Data (Music_Data_7 _) = Nothing
  -- abst_Music_Data (Music_Data_8 _) = Nothing
  -- abst_Music_Data (Music_Data_9 _) = Nothing
  -- abst_Music_Data (Music_Data_10 _) = Nothing
  -- abst_Music_Data (Music_Data_11 _) = Nothing
  -- abst_Music_Data (Music_Data_12 _) = Nothing
  -- abst_Music_Data (Music_Data_13 _) = Nothing
  -- |
```

```
split_Note :: Note → (Note, Layer1.Note)
split_Note =< id, λ(a, b, c, d) → ((Layer1.Note_3 · π2 · split_Note_) a,
  fmap (π2 · split_Type) b, fmap (π2 · split_Dot) c, fmap (π2 · split_Accidental) d) >
  -- (\(x1,x4,x5,x6) ->
  -- (abst_Note_ x1, fmap abst_Type x4,
  -- fmap abst_Dot x5, fmap abst_Accidental x6))
  -- |
split_Note_ :: Note_ → (Duration, Layer1.Full_Note_)
  -- abst_Note_ (Note_1 _) = Nothing
  -- Layer1.Note_1 (abst_Grace x1, abst_Full_Note x2,
  -- fmap (abst_Tie >< fmap abst_Tie) x3)
  -- abst_Note_ (Note_2 _) = Nothing
  -- Layer1.Note_2 (abst_Cue x1, abst_Full_Note x2, abst_Duration x3)
  -- split_Note_ (Note_3 (x1,x2)) = split (const x2) (Layer1.Note_3 (abst_Full_Note x1))
split_Note_ =< π2, π2 · split_Full_Note · π1 > ·(λ(Note_3 x) → x)
  -- split (const x2) (Layer1.Note_3 (abst_Full_Note x1))
  -- x1' <- abst_Full_Note x1
  -- return (Layer1.Note_3 (x1', abst_Duration x2))
  -- |
split_Full_Note :: Full_Note → (Full_Note, Layer1.Full_Note)
split_Full_Note = split_Full_Note_
  -- b' <- abst_Full_Note_ b
  -- return (fmap id a, b')
  -- abst_Full_Note :: Full_Note -> (Maybe Layer1.Chord, Maybe Layer1.Full_Note_)
  -- abst_Full_Note = (fmap id >< abst_Full_Note_)
  -- abst_Full_Note (x,y) = do
  -- y' <- abst_Full_Note_ y
  -- return (fmap id x >< fmap id y'))
  -- (fmap id >< (\x -> do
  -- y <- abst_Full_Note_ x
  -- ))
  -- |
split_Full_Note_ :: Full_Note_ → (Full_Note_, Layer1.Full_Note_)
split_Full_Note_ (Full_Note_1 x) =
  ((id × Layer1.Full_Note_1)· < Full_Note_1, π2 · split_Pitch >) x
  -- (Layer1.Full_Note_1 (split_Pitch x))
  -- abst_Full_Note_ (Full_Note_2 _) = Nothing
split_Full_Note_ (Full_Note_3 x) =< Full_Note_3, Layer1.Full_Note_3 >  x
  -- (Layer1.Full_Note_3 x)
  -- |
split_Pitch :: Pitch → (Pitch, Layer1.Pitch)
split_Pitch =< id, id >
  -- |
split_Type :: Type → (Type, Layer1.Type)
split_Type =< id, id >
  -- |
split_Dot :: Dot → (Dot, Layer1.Dot)
split_Dot =< id, () >
  -- |
split_Accidental :: Accidental → (Accidental, Layer1.Accidental)
split_Accidental =< id, id >
```

4.5 Layer3

```
module Music.Analysis.MusicXML.Level3 (
  module Music.Analysis.MusicXML.Level3,
  ) where
import Music.Analysis.Base (IntegerNumber)
import Music.Analysis.PF
import qualified Music.Analysis.MusicXML.Level1 as Layer1
import qualified Music.Analysis.MusicXML.Level2 as Layer2
import qualified Music.Analysis.MusicXML.Level2Num as Layer2Num
import qualified Music.Analysis.MusicXML.Level3Num as Layer3Num
import Data.Maybe
import qualified Text.XML.MusicXML as MusicXML


  -- |
type Score_Partwise =
  (MusicXML.Document_Attributes, (MusicXML.Score_Header, [Part]))
  -- |
type Part = [Measure]
  -- |
type Measure = [Music_Data]
  -- |
data Music_Data =
    Music_Data_1 Note
  | Music_Data_2 MusicXML.Backup
  | Music_Data_3 MusicXML.Forward
  | Music_Data_5 Attributes
  | Music_Data_10 Barline
    deriving (Eq, Show)
  -- |
type Barline = MusicXML.Barline
  -- |
type Note =
  (Note_, Maybe Instrument, Editorial_Voice,
    Maybe Type, [Dot], Maybe Accidental,
    Maybe Staff)
  -- |
data Note_ =
  Note_3 (Full_Note, Duration)
  deriving (Eq, Show)
  -- |
type Full_Note = (Maybe MusicXML.Chord, Full_Note_)
  -- |
data Full_Note_ = Full_Note_1 Layer2.Pitch
  | Full_Note_3 Rest
    deriving (Eq, Show)
  -- |
type Rest = ()
  -- |
type Duration = IntegerNumber
  -- |
type Editorial_Voice = MusicXML.Editorial_Voice
```

```
  -- |
type Instrument = MusicXML.Instrument
  -- |
type Type = Layer1.Type_
  -- |
type Dot = MusicXML.Dot
  -- |
type Accidental = Layer1.Accidental_
  -- | positive number
type Staff = IntegerNumber
  -- |
type Attributes = (Maybe Divisions, [Key], [Time],
  Maybe Staves, Maybe Instruments,
  [Clef], Maybe Transpose)
  -- |
type Editorial = MusicXML.Editorial
  -- |
type Divisions = IntegerNumber
  -- |
type Key = (Key_, [Layer2.Key_Octave])
  -- |
data Key_ = Key_1 (Maybe MusicXML.Cancel, Layer2.Fifths, Maybe Layer2.Mode)
   | Key_2 [(Layer2.Key_Step, Layer2.Key_Alter)]
     deriving (Eq, Show)
  -- |
type Time = Time_B
  -- |
data Time_B = Time_5 [(Beats, Beat_Type)]
      | Time_6 MusicXML.Senza_Misura
        deriving (Eq, Show)
  -- | MusicXML Schema specify 'xs:string'
type Beats = (IntegerNumber, Maybe IntegerNumber)
  -- | MusicXML Schema specify 'xs:string'
type Beat_Type = IntegerNumber
  -- |
type Staves = MusicXML.Staves
  -- |
type Part_Symbol = MusicXML.Part_Symbol
  -- |
type Instruments = MusicXML.Instruments
  -- |
type Clef = Layer2.Clef
  -- |
type Staff_Details = MusicXML.Staff_Details
  -- |
type Transpose = MusicXML.Transpose
  -- |
type Directive = MusicXML.Directive
  -- |
type Measure_Style = MusicXML.Measure_Style

  -- |
abst_Score_Partwise :: Score_Partwise → Layer2.Score_Partwise
abst_Score_Partwise =
```

```
  ([[], fmap abst_Measure · head] · grd null) ·
  π2 · π2
  -- (id -|- fmap abst_Measure . head) . grd null . p2 . p2
  -- |
abst_Measure :: Measure → Layer2.Measure
abst_Measure = catMaybes · fmap abst_Music_Data
  -- |
abst_Music_Data :: Music_Data → Maybe Layer2.Music_Data
abst_Music_Data (Music_Data_1 x) = Just (Layer2.Music_Data_1 (abst_Note x))
  -- x' <- abst_Note x
  -- return (Layer2.Music_Data_1 x')
  -- Layer2.Music_Data_1 ((catMaybes . (fmap abst_Note)) x)
abst_Music_Data (Music_Data_2 _) = Nothing
abst_Music_Data (Music_Data_3 _) = Nothing
  -- abst_Music_Data (Music_Data_4 _) = Nothing
abst_Music_Data (Music_Data_5 x) =
  Just (Layer2.Music_Data_5 (abst_Attributes x))
  -- abst_Music_Data (Music_Data_6 _) = Nothing
  -- abst_Music_Data (Music_Data_7 _) = Nothing
  -- abst_Music_Data (Music_Data_8 _) = Nothing
  -- abst_Music_Data (Music_Data_9 _) = Nothing
abst_Music_Data (Music_Data_10 _) = Nothing
  -- abst_Music_Data (Music_Data_11 _) = Nothing
  -- abst_Music_Data (Music_Data_12 _) = Nothing
  -- abst_Music_Data (Music_Data_13 _) = Nothing
  -- |
abst_Note :: Note → Layer2.Note
abst_Note =
  (λ(x1, _, _, x4, x5, x6, _) →
    (abst_Note_ x1, fmap abst_Type x4,
      fmap abst_Dot x5, fmap abst_Accidental x6))
  -- |
abst_Note_ :: Note_ → Layer2.Note_
  -- abst_Note_ (Note_1 _) = Nothing
  -- Layer2.Note_1 (abst_Grace x1, abst_Full_Note x2,
  -- fmap (abst_Tie >< fmap abst_Tie) x3)
  -- abst_Note_ (Note_2 _) = Nothing
  -- Layer2.Note_2 (abst_Cue x1, abst_Full_Note x2, abst_Duration x3)
abst_Note_ (Note_3 (x1, x2)) =
  Layer2.Note_3 (abst_Full_Note x1, abst_Duration x2)
  -- x1' <- abst_Full_Note x1
  -- return (Layer2.Note_3 (x1', abst_Duration x2))
  -- |
abst_Full_Note :: Full_Note → Layer2.Full_Note
abst_Full_Note = abst_Full_Note_ · π2
  -- b' <- abst_Full_Note_ b
  -- return (fmap id a, b')
  -- abst_Full_Note :: Full_Note ->
  -- (Maybe Layer2.Chord, Maybe Layer2.Full_Note_)
  -- abst_Full_Note = (fmap id >< abst_Full_Note_)
  -- abst_Full_Note (x,y) = do
  -- y' <- abst_Full_Note_ y
  -- return (fmap id x >< fmap id y'))
```

```
  -- (fmap id >< (\x -> do
  -- y <- abst_Full_Note_ x
  -- ))
  -- |
abst_Full_Note_ :: Full_Note_ → Layer2.Full_Note_
abst_Full_Note_ (Full_Note_1 x) = (Layer2.Full_Note_1 x)
  -- abst_Full_Note_ (Full_Note_2 _) = Nothing
abst_Full_Note_ (Full_Note_3 _) = (Layer2.Full_Note_3 ())
  -- |
abst_Duration :: Duration → Layer2.Duration
abst_Duration = id
  -- |
abst_Type :: Type → Layer2.Type
abst_Type = id
  -- |
abst_Dot :: Dot → Layer2.Dot
abst_Dot = ()
  -- |
abst_Accidental :: Accidental → Layer2.Accidental
abst_Accidental = id
  -- |
abst_Divisions :: Divisions → Layer2.Divisions
abst_Divisions = id
  -- |
abst_Attributes :: Attributes → Layer2.Attributes
abst_Attributes (x2, x3, x4, _, _, x8, _) =
  (fmap abst_Divisions x2, fmap abst_Key x3,
    fmap abst_Time x4, fmap abst_Clef x8)
  -- |
abst_Key :: Key → Layer2.Key
abst_Key = (abst_Key_ × fmap id)
  -- |
abst_Key_ :: Key_ → Layer2.Key_
abst_Key_ (Key_1 x) = (Layer2.Key_1 · π2 · unflatr) x
abst_Key_ (Key_2 x) = Layer2.Key_2 x
  -- |
abst_Time :: Time → Layer2.Time
abst_Time (Time_5 l) = Layer2.Time_5 l
abst_Time (Time_6 _) = Layer2.Time_5 []
  -- |
abst_Clef :: Clef → Layer2.Clef
abst_Clef = id


  -- |
map_Score_Partwise :: (Music_Data → b) → Score_Partwise →
  (MusicXML.Document_Attributes, (MusicXML.Score_Header, [[[b]]]))
map_Score_Partwise f = (id × (id × fmap (map_Part f)))
  -- |
map_Part :: (Music_Data → b) → Part → [[b]]
map_Part f = fmap (map_Measure f)
  -- |
map_Measure :: (Music_Data → b) → Measure → [b]
map_Measure f = fmap (map_Music_Data f)
```

```
  -- |
map_Music_Data :: (Music_Data → b) → Music_Data → b
map_Music_Data f = f
  -- map_Music_Data :: Music_Data -> Music_Data
  -- map_Music_Data f (Music_Data_1 x) = f (Music_Data_1 x)
  -- map_Music_Data f (Music_Data_2 x) = f (Music_Data_2 x)
  -- map_Music_Data f (Music_Data_3 x) = f (Music_Data_3 x)
  -- map_Music_Data (Music_Data_4 x) = Music_Data_4 x
  -- map_Music_Data f (Music_Data_5 x) = f (Music_Data_5 x)
  -- map_Music_Data (Music_Data_6 x) = Music_Data_6 x
  -- map_Music_Data (Music_Data_7 x) = Music_Data_7 x
  -- map_Music_Data (Music_Data_8 x) = Music_Data_8 x
  -- map_Music_Data (Music_Data_9 x) = Music_Data_9 x
  -- map_Music_Data f (Music_Data_10 x) = f (Music_Data_10 x)
  -- map_Music_Data (Music_Data_11 x) = Music_Data_11 x
  -- map_Music_Data (Music_Data_12 x) = Music_Data_12 x
  -- map_Music_Data (Music_Data_13 x) = Music_Data_13 x
toNum_Music_Data :: Music_Data → Layer3Num.Music_Data
toNum_Music_Data (Music_Data_1 x) = Layer3Num.Music_Data_1 (toNum_Note x)
toNum_Music_Data (Music_Data_2 x) = Layer3Num.Music_Data_2 x
toNum_Music_Data (Music_Data_3 x) = Layer3Num.Music_Data_3 x
toNum_Music_Data (Music_Data_5 x) =
  Layer3Num.Music_Data_5 (toNum_Attributes x)
toNum_Music_Data (Music_Data_10 x) = Layer3Num.Music_Data_10 x
toNum_Note :: Note → Layer3Num.Note
toNum_Note ((Note_3 (a1, a2)), b, c, d, e, f, g) =
  ((Layer3Num.Note_3 (toNum_Full_Note a1, a2)), b, c,
    fmap fromEnum d, e, fmap fromEnum f, g)
toNum_Full_Note :: Full_Note → Layer3Num.Full_Note
toNum_Full_Note (a, Full_Note_1 (b1, b2, b3)) =
  (a, Layer3Num.Full_Note_1 ((fromEnum b1) + (b3 * 7), b2))
toNum_Full_Note (a, Full_Note_3 b) = (a, Layer3Num.Full_Note_3 b)
toNum_Attributes :: Attributes → Layer3Num.Attributes
toNum_Attributes (a, b, c, d, e, f, g) =
  (a, fmap toNum_Key b, fmap toNum_Time c, d, e, fmap toNum_Clef f, g)
toNum_Key :: Key → Layer3Num.Key
toNum_Key (Key_1 (a1, a2, a3), b) =
  (Layer3Num.Key_1 (a1, a2, fmap toNum_Mode a3), b)
toNum_Key (Key_2 a, b) =
  (Layer3Num.Key_2 (fmap (λ(x, y) → (fromEnum x, y)) a), b)
toNum_Mode :: Layer2.Mode → Layer2Num.Mode
toNum_Mode Layer2.Major = Layer2Num.Major
toNum_Mode Layer2.Minor = Layer2Num.Minor
toNum_Mode Layer2.Dorian = Layer2Num.Dorian
toNum_Mode Layer2.Phrygian = Layer2Num.Phrygian
toNum_Mode Layer2.Lydian = Layer2Num.Lydian
toNum_Mode Layer2.Mixolydian = Layer2Num.Mixolydian
toNum_Mode Layer2.Aeolian = Layer2Num.Aeolian
toNum_Mode Layer2.Ionian = Layer2Num.Ionian
toNum_Mode Layer2.Locrian = Layer2Num.Locrian
toNum_Time :: Time → Layer3Num.Time
toNum_Time (Time_5 a) = Layer3Num.Time_5 a
```

```
toNum_Time (Time_6 a) = Layer3Num.Time_6 a
toNum_Clef :: Clef → Layer2Num.Clef
toNum_Clef (a, b, c) = (fromEnum a, b, c)
```

4.6 Layer4

```
module Music.Analysis.MusicXML.Level4 (
  module Music.Analysis.MusicXML.Level4,
  ) where
import Music.Analysis.Base (IntegerNumber)
import Music.Analysis.PF
import qualified Music.Analysis.MusicXML.Level1 as Layer1
import qualified Music.Analysis.MusicXML.Level2 as Layer2
import qualified Music.Analysis.MusicXML.Level3 as Layer3
import Data.Maybe (catMaybes)
import qualified Text.XML.MusicXML as MusicXML


  -- |
type Score_Partwise =
  (MusicXML.Document_Attributes, (MusicXML.Score_Header, [Part]))
  -- |
type Part = [Measure]
  -- |
type Measure = [Music_Data]
  -- |
data Music_Data =
    Music_Data_1 Note
  | Music_Data_2 MusicXML.Backup
  | Music_Data_3 MusicXML.Forward
  | Music_Data_4 MusicXML.Direction
  | Music_Data_5 Attributes
  | Music_Data_6 MusicXML.Harmony
  | Music_Data_7 MusicXML.Figured_Bass
  | Music_Data_8 MusicXML.Print
  | Music_Data_9 MusicXML.Sound
  | Music_Data_10 MusicXML.Barline
  | Music_Data_11 MusicXML.Grouping
  | Music_Data_12 MusicXML.Link
  | Music_Data_13 MusicXML.Bookmark
    deriving (Eq, Show)
  -- |
type Note =
  (Note_, Maybe Instrument, Editorial_Voice,
    Maybe Type, [Dot], Maybe Accidental,
    Maybe Time_Modification, Maybe Stem,
    Maybe Notehead, Maybe Staff, [Beam],
    [Notations], [Lyric])
  -- |
data Note_ =
      Note_1 (Grace, Full_Note, Maybe (Tie, Maybe Tie))
```

```
      | Note_2 (Cue, Full_Note, Duration)
      | Note_3 (Full_Note, Duration, Maybe (Tie, Maybe Tie))
        deriving (Eq, Show)
  -- |
type Grace = MusicXML.Grace
  -- |
type Cue = MusicXML.Cue
  -- |
type Tie = MusicXML.Tie
  -- |
type Full_Note = (Maybe MusicXML.Chord, Full_Note_)
  -- |
data Full_Note_ = Full_Note_1 Layer2.Pitch
   | Full_Note_2 Unpitched
   | Full_Note_3 Rest
     deriving (Eq, Show)
  -- |
type Unpitched = MusicXML.Unpitched
  -- |
type Rest = MusicXML.Rest
  -- |
type Duration = IntegerNumber
  -- |
type Editorial_Voice = MusicXML.Editorial_Voice
  -- |
type Instrument = MusicXML.Instrument
  -- |
type Type = Layer1.Type_
  -- |
type Dot = MusicXML.Dot
  -- |
type Accidental = Layer1.Accidental_
  -- |
type Time_Modification = MusicXML.Time_Modification
  -- |
type Stem = MusicXML.Stem
  -- |
type Notehead = MusicXML.Notehead
  -- |
type Beam = MusicXML.Beam
  -- | positive number
type Staff = IntegerNumber
  -- |
type Lyric = MusicXML.Lyric
  -- |
type Notations = MusicXML.Notations
  -- |
type Attributes = (Editorial, Maybe Divisions, [Key], [Time],
  Maybe Staves, Maybe Part_Symbol, Maybe Instruments,
  [Clef], [Staff_Details], Maybe Transpose, [Directive],
  [Measure_Style])
  -- |
type Editorial = MusicXML.Editorial
  -- |
```

```
type Divisions = IntegerNumber
  -- |
type Key = (Key_, [Layer2.Key_Octave])
  -- |
data Key_ = Key_1 (Maybe MusicXML.Cancel, Layer2.Fifths, Maybe Layer2.Mode)
          | Key_2 [(Layer2.Key_Step, Layer2.Key_Alter)]
            deriving (Eq, Show)
  -- |
type Time = Layer3.Time_B
  -- |
type Staves = MusicXML.Staves
  -- |
type Part_Symbol = MusicXML.Part_Symbol
  -- |
type Instruments = MusicXML.Instruments
  -- |
type Clef = (Layer2.Sign, Maybe Layer2.Line, Maybe Layer2.Clef_Octave_Change)
  -- |
type Staff_Details = MusicXML.Staff_Details
  -- |
type Transpose = MusicXML.Transpose
  -- |
type Directive = MusicXML.Directive
  -- |
type Measure_Style = MusicXML.Measure_Style


  -- |
abst_Score_Partwise :: Score_Partwise → Layer3.Score_Partwise
abst_Score_Partwise = (id × (id × map abst_Part))
  -- |
abst_Part :: Part → Layer3.Part
abst_Part = fmap abst_Measure
  -- |
abst_Measure :: Measure → Layer3.Measure
abst_Measure = catMaybes · fmap abst_Music_Data
  -- |
abst_Music_Data :: Music_Data → Maybe Layer3.Music_Data
abst_Music_Data (Music_Data_1 x) = do
  x' ← abst_Note x
  return (Layer3.Music_Data_1 x')
abst_Music_Data (Music_Data_2 x) = Just (Layer3.Music_Data_2 x)
abst_Music_Data (Music_Data_3 x) = Just (Layer3.Music_Data_3 x)
abst_Music_Data (Music_Data_4 _) = Nothing
abst_Music_Data (Music_Data_5 x) =
  Just (Layer3.Music_Data_5 (abst_Attributes x))
abst_Music_Data (Music_Data_6 _) = Nothing
abst_Music_Data (Music_Data_7 _) = Nothing
abst_Music_Data (Music_Data_8 _) = Nothing
abst_Music_Data (Music_Data_9 _) = Nothing
abst_Music_Data (Music_Data_10 x) = Just (Layer3.Music_Data_10 x)
abst_Music_Data (Music_Data_11 _) = Nothing
abst_Music_Data (Music_Data_12 _) = Nothing
abst_Music_Data (Music_Data_13 _) = Nothing
```

```
  -- |
abst_Note :: Note → Maybe Layer3.Note
abst_Note =
  (λ(x1, x2, x3, x4, x5, x6, _, _, _, x10, _, _, _) → do
    x1' ← abst_Note_ x1
    return (x1', fmap abst_Instrument x2, abst_Editorial_Voice x3,
      fmap abst_Type x4, fmap abst_Dot x5,
      fmap abst_Accidental x6, fmap abst_Staff x10))
  -- |
abst_Note_ :: Note_ → Maybe Layer3.Note_
abst_Note_ (Note_1 _) = Nothing
abst_Note_ (Note_2 _) = Nothing
abst_Note_ (Note_3 (x1, x2, _)) = do
  x1' ← abst_Full_Note x1
  return (Layer3.Note_3 (x1', abst_Duration x2))
  -- |
abst_Full_Note :: Full_Note → Maybe Layer3.Full_Note
abst_Full_Note (a, b) = do
  b' ← abst_Full_Note_ b
  return (fmap id a, b')
  -- abst_Full_Note :: Full_Note -> (Maybe Layer3.Chord, Maybe Layer3.Full_Note_)
  -- abst_Full_Note = (fmap id >< abst_Full_Note_)
  -- abst_Full_Note (x,y) = do
  -- y' <- abst_Full_Note_ y
  -- return (fmap id x >< fmap id y'))
  -- (fmap id >< (\x -> do
  -- y <- abst_Full_Note_ x
  -- ))
  -- |
abst_Full_Note_ :: Full_Note_ → Maybe Layer3.Full_Note_
abst_Full_Note_ (Full_Note_1 x) = Just (Layer3.Full_Note_1 x)
abst_Full_Note_ (Full_Note_2 _) = Nothing
abst_Full_Note_ (Full_Note_3 _) = Just (Layer3.Full_Note_3 ())
  -- |
abst_Duration :: Duration → Layer3.Duration
abst_Duration = id
  -- |
abst_Editorial_Voice :: Editorial_Voice → Layer3.Editorial_Voice
abst_Editorial_Voice = id
  -- |
abst_Instrument :: Instrument → Layer3.Instrument
abst_Instrument = id
  -- |
abst_Type :: Type → Layer3.Type
abst_Type = id
  -- |
abst_Dot :: Dot → Layer3.Dot
abst_Dot = id
  -- |
abst_Accidental :: Accidental → Layer3.Accidental
abst_Accidental = id
  -- |
abst_Staff :: Staff → Layer3.Staff
abst_Staff = id
```

```
  -- |
abst_Editorial :: Editorial → Layer3.Editorial
abst_Editorial = id
  -- |
abst_Divisions :: Divisions → Layer3.Divisions
abst_Divisions = id
  -- |
abst_Staves :: Staves → Layer3.Staves
abst_Staves = id
  -- |
abst_Attributes :: Attributes → Layer3.Attributes
abst_Attributes (_, x2, x3, x4, x5, _, x7, x8, _, x10, _, _) =
  (fmap abst_Divisions x2, fmap abst_Key x3,
    fmap abst_Time x4, fmap abst_Staves x5,
    fmap abst_Instruments x7, fmap abst_Clef x8,
    fmap abst_Transpose x10)
  -- |
abst_Key :: Key → Layer3.Key
abst_Key = (abst_Key_ × fmap id)
  -- |
abst_Key_ :: Key_ → Layer3.Key_
abst_Key_ (Key_1 x) = Layer3.Key_1 x
abst_Key_ (Key_2 x) = Layer3.Key_2 x
  -- |
abst_Time :: Time → Layer3.Time
abst_Time = id
  -- |
abst_Part_Symbol :: Part_Symbol → Layer3.Part_Symbol
abst_Part_Symbol = id
  -- |
abst_Instruments :: Instruments → Layer3.Instruments
abst_Instruments = id
  -- |
abst_Clef :: Clef → Layer3.Clef
abst_Clef = id
  -- |
abst_Staff_Details :: Staff_Details → Layer3.Staff_Details
abst_Staff_Details = id
  -- |
abst_Transpose :: Transpose → Layer3.Transpose
abst_Transpose = id
  -- |
abst_Directive :: Directive → Layer3.Directive
abst_Directive = id
  -- |
abst_Measure_Style :: Measure_Style → Layer3.Measure_Style
abst_Measure_Style = id


  -- |
rep_Score_Partwise :: Layer3.Score_Partwise → Score_Partwise
rep_Score_Partwise = (id × (id × map rep_Part))
  -- |
rep_Part :: Layer3.Part → Part
rep_Part = fmap rep_Measure
```

```
  -- |
rep_Measure :: Layer3.Measure → Measure
rep_Measure = fmap rep_Music_Data
  -- |
rep_Music_Data :: Layer3.Music_Data → Music_Data
rep_Music_Data (Layer3.Music_Data_1 x) = (Music_Data_1 (rep_Note x))
  -- x' <- rep_Note x
  -- return (Music_Data_1 x')
  -- Layer3.Music_Data_1 ((catMaybes . (fmap abst_Note)) x)
rep_Music_Data (Layer3.Music_Data_2 x) = (Music_Data_2 x)
rep_Music_Data (Layer3.Music_Data_3 x) = (Music_Data_3 x)
  -- rep_Music_Data (Layer3.Music_Data_4 _) = Nothing
rep_Music_Data (Layer3.Music_Data_5 x) =
  (Music_Data_5 (rep_Attributes x))
  -- rep_Music_Data (Layer3.Music_Data_6 _) = Nothing
  -- rep_Music_Data (Layer3.Music_Data_7 _) = Nothing
  -- rep_Music_Data (Layer3.Music_Data_8 _) = Nothing
  -- rep_Music_Data (Layer3.Music_Data_9 _) = Nothing
rep_Music_Data (Layer3.Music_Data_10 x) = (Music_Data_10 x)
  -- rep_Music_Data (Layer3.Music_Data_11 _) = Nothing
  -- rep_Music_Data (Layer3.Music_Data_12 _) = Nothing
  -- rep_Music_Data (Layer3.Music_Data_13 _) = Nothing
  -- |
rep_Note :: Layer3.Note → Note
rep_Note (x1, x2, x3, x4, x5, x6, x10) =
  (rep_Note_ x1, fmap rep_Instrument x2, rep_Editorial_Voice x3,
    fmap rep_Type x4, fmap rep_Dot x5, fmap rep_Accidental x6,
    Nothing, Nothing, Nothing, fmap rep_Staff x10, [], [], [])
  -- (\(x1,x2,x3,x4,x5,x6,_,_,_,x10,_,_,_) -> do
  -- x1' <- abst_Note_ x1
  -- return (x1',fmap abst_Instrument x2, abst_Editorial_Voice x3,
  -- fmap abst_Type x4, fmap abst_Dot x5,
  -- fmap abst_Accidental x6, fmap abst_Staff x10))
  -- |
rep_Note_ :: Layer3.Note_ → Note_
  -- rep_Note_ (Note_1 _) = Nothing
  -- Layer3.Note_1 (abst_Grace x1, abst_Full_Note x2,
  -- fmap (abst_Tie >< fmap abst_Tie) x3)
  -- rep_Note_ (Note_2 _) = Nothing
  -- Layer3.Note_2 (abst_Cue x1, abst_Full_Note x2, abst_Duration x3)
rep_Note_ (Layer3.Note_3 (x1, x2)) =
  (Note_3 (rep_Full_Note x1, rep_Duration x2, Nothing))
  -- x1' <- abst_Full_Note x1
  -- return (Layer3.Note_3 (x1', abst_Duration x2))
  -- |
rep_Full_Note :: Layer3.Full_Note → Full_Note
rep_Full_Note (a, b) = (a, rep_Full_Note_ b)
  -- b' <- rep_Full_Note_ b
  -- return (fmap id a, b')
  -- abst_Full_Note :: Full_Note -> (Maybe Layer3.Chord, Maybe Layer3.Full_Note_)
  -- abst_Full_Note = (fmap id >< abst_Full_Note_)
  -- abst_Full_Note (x,y) = do
  -- y' <- abst_Full_Note_ y
  -- return (fmap id x >< fmap id y'))
```

```
  -- (fmap id >< (\x -> do
  -- y <- abst_Full_Note_ x
  -- ))
  -- |
rep_Full_Note_ :: Layer3.Full_Note_ → Full_Note_
rep_Full_Note_ (Layer3.Full_Note_1 x) = (Full_Note_1 x)
  -- rep_Full_Note_ (Full_Note_2 _) = Nothing
rep_Full_Note_ (Layer3.Full_Note_3 _) = (Full_Note_3 Nothing)
  -- |
rep_Duration :: Layer3.Duration → Duration
rep_Duration = id
  -- |
rep_Editorial_Voice :: Layer3.Editorial_Voice → Editorial_Voice
rep_Editorial_Voice = id
  -- |
rep_Instrument :: Layer3.Instrument → Instrument
rep_Instrument = id
  -- |
rep_Type :: Layer3.Type → Type
rep_Type = id
  -- |
rep_Dot :: Layer3.Dot → Dot
rep_Dot = id
  -- |
rep_Accidental :: Layer3.Accidental → Accidental
rep_Accidental = id
  -- |
rep_Staff :: Layer3.Staff → Staff
rep_Staff = id
  -- |
rep_Editorial :: Layer3.Editorial → Editorial
rep_Editorial = id
  -- |
rep_Divisions :: Layer3.Divisions → Divisions
rep_Divisions = id
  -- |
rep_Staves :: Layer3.Staves → Staves
rep_Staves = id
  -- |
rep_Attributes :: Layer3.Attributes → Attributes
rep_Attributes (x2, x3, x4, x5, x7, x8, x10) =
  ((Nothing, Nothing), fmap rep_Divisions x2, fmap rep_Key x3,
    fmap rep_Time x4, fmap rep_Staves x5, Nothing,
    fmap rep_Instruments x7, fmap rep_Clef x8, [],
    fmap rep_Transpose x10, [], [])
  -- (_,x2,x3,x4,x5,_,x7,x8,_,x10,_,_) =
  -- (fmap abst_Divisions x2, fmap abst_Key x3,
  -- fmap abst_Time x4, fmap abst_Staves x5,
  -- fmap abst_Instruments x7, fmap abst_Clef x8,
  -- fmap abst_Transpose x10)
  -- |
rep_Key :: Layer3.Key → Key
rep_Key = (rep_Key_ × fmap id)
  -- |
```

```
rep_Key_ :: Layer3.Key_ → Key_
rep_Key_ (Layer3.Key_1 x) = Key_1 x
rep_Key_ (Layer3.Key_2 x) = Key_2 x
  -- |
rep_Time :: Layer3.Time → Time
rep_Time = id
  -- |
rep_Part_Symbol :: Layer3.Part_Symbol → Part_Symbol
rep_Part_Symbol = id
  -- |
rep_Instruments :: Layer3.Instruments → Instruments
rep_Instruments = id
  -- |
rep_Clef :: Layer3.Clef → Clef
rep_Clef = id
  -- |
rep_Staff_Details :: Layer3.Staff_Details → Staff_Details
rep_Staff_Details = id
  -- |
rep_Transpose :: Layer3.Transpose → Transpose
rep_Transpose = id
  -- |
rep_Directive :: Layer3.Directive → Directive
rep_Directive = id
  -- |
rep_Measure_Style :: Layer3.Measure_Style → Measure_Style
rep_Measure_Style = id


  -- |
map_Score_Partwise :: (Music_Data → Music_Data) →
  Score_Partwise → Score_Partwise
map_Score_Partwise f = (id × (id × fmap (map_Part f)))
  -- |
map_Part :: (Music_Data → Music_Data) → Part → Part
map_Part f = fmap (map_Measure f)
  -- |
map_Measure :: (Music_Data → Music_Data) → Measure → Measure
map_Measure f = fmap (map_Music_Data f)
  -- |
map_Music_Data :: (Music_Data → Music_Data) → Music_Data → Music_Data
map_Music_Data f = f
  -- map_Music_Data :: Music_Data -> Music_Data
  -- map_Music_Data f (Music_Data_1 x) = Music_Data_1 x
  -- map_Music_Data f (Music_Data_2 x) = Music_Data_2 x
  -- map_Music_Data f (Music_Data_3 x) = Music_Data_3 x
  -- map_Music_Data f (Music_Data_4 x) = Music_Data_4 x
  -- map_Music_Data f (Music_Data_5 x) = Music_Data_5 x
  -- map_Music_Data f (Music_Data_6 x) = Music_Data_6 x
  -- map_Music_Data f (Music_Data_7 x) = Music_Data_7 x
  -- map_Music_Data f (Music_Data_8 x) = Music_Data_8 x
  -- map_Music_Data f (Music_Data_9 x) = Music_Data_9 x
  -- map_Music_Data f (Music_Data_10 x) = Music_Data_10 x
  -- map_Music_Data f (Music_Data_11 x) = Music_Data_11 x
```

```
  -- map_Music_Data f (Music_Data_12 x) = Music_Data_12 x
  -- map_Music_Data f (Music_Data_13 x) = Music_Data_13 x
```

4.7 Layer5

```
module Music.Analysis.MusicXML.Level5 (
  module Music.Analysis.MusicXML.Level5,
  ) where
import Music.Analysis.Base
import Music.Analysis.PF
import qualified Music.Analysis.MusicXML.Level1 as Layer1
import qualified Music.Analysis.MusicXML.Level2 as Layer2
import qualified Music.Analysis.MusicXML.Level3 as Layer3
import qualified Music.Analysis.MusicXML.Level4 as Layer4
import Data.Maybe ()
import qualified Text.XML.MusicXML as MusicXML


  -- |
type Score_Partwise =
  (MusicXML.Document_Attributes, (MusicXML.Score_Header, [Part]))
  -- |
type Part = (MusicXML.ID, [Measure])
  -- |
type Measure = ((MusicXML.CDATA, Maybe MusicXML.Yes_No,
  Maybe MusicXML.Yes_No, Maybe MusicXML.Tenths), [Music_Data])
  -- |
data Music_Data =
    Music_Data_1 Note
  | Music_Data_2 MusicXML.Backup
  | Music_Data_3 MusicXML.Forward
  | Music_Data_4 MusicXML.Direction
  | Music_Data_5 Attributes
  | Music_Data_6 MusicXML.Harmony
  | Music_Data_7 MusicXML.Figured_Bass
  | Music_Data_8 MusicXML.Print
  | Music_Data_9 MusicXML.Sound
  | Music_Data_10 MusicXML.Barline
  | Music_Data_11 MusicXML.Grouping
  | Music_Data_12 MusicXML.Link
  | Music_Data_13 MusicXML.Bookmark
    deriving (Eq, Show)
  -- |
type Note =
  ((MusicXML.Print_Style, MusicXML.Printout, Maybe MusicXML.CDATA,
    Maybe MusicXML.CDATA, Maybe MusicXML.CDATA, Maybe MusicXML.CDATA,
    Maybe MusicXML.CDATA, Maybe MusicXML.Yes_No),
  (Note_, Maybe Instrument, Editorial_Voice,
    Maybe Type, [Dot], Maybe Accidental,
    Maybe Time_Modification, Maybe Stem,
    Maybe Notehead, Maybe Staff, [Beam],
    [Notations], [Lyric]))
```

```
  -- |
data Note_ =
       Note_1 (Grace, Full_Note, Maybe (Tie, Maybe Tie))
     | Note_2 (Cue, Full_Note, Duration)
     | Note_3 (Full_Note, Duration, Maybe (Tie, Maybe Tie))
       deriving (Eq, Show)
  -- |
type Grace = MusicXML.Grace
  -- |
type Cue = MusicXML.Cue
  -- |
type Tie = MusicXML.Tie
  -- |
type Full_Note = (Maybe MusicXML.Chord, Full_Note_)
  -- |
data Full_Note_ = Full_Note_1 Layer2.Pitch
   | Full_Note_2 Layer4.Unpitched
   | Full_Note_3 Layer4.Rest
     deriving (Eq, Show)
  -- |
type Duration = IntegerNumber
  -- |
type Editorial_Voice = MusicXML.Editorial_Voice
  -- |
type Instrument = MusicXML.Instrument
  -- |
type Type = (Maybe MusicXML.Symbol_Size, Layer1.Type_)
  -- |
type Dot = MusicXML.Dot
  -- |
type Accidental = ((Maybe MusicXML.Yes_No, Maybe MusicXML.Yes_No,
  MusicXML.Level_Display, MusicXML.Print_Style), Layer1.Accidental_)
  -- |
type Time_Modification = MusicXML.Time_Modification
  -- |
type Stem = MusicXML.Stem
  -- |
type Notehead = MusicXML.Notehead
  -- |
type Beam = MusicXML.Beam
  -- | positive number
type Staff = IntegerNumber
  -- |
type Lyric = MusicXML.Lyric
  -- |
type Notations = MusicXML.Notations
  -- |
type Attributes = (Editorial, Maybe Divisions, [Key], [Time],
  Maybe Staves, Maybe Part_Symbol, Maybe Instruments,
  [Clef], [Staff_Details], Maybe Transpose, [Directive],
  [Measure_Style])
  -- |
type Editorial = MusicXML.Editorial
  -- |
```

```
type Divisions = IntegerNumber
  -- |
type Key = (
  (Maybe MusicXML.CDATA, MusicXML.Print_Style, MusicXML.Print_Object),
  (Key_, [Key_Octave]))
  -- |
data Key_ = Key_1 (Maybe MusicXML.Cancel, Layer2.Fifths, Maybe Layer2.Mode)
        | Key_2 [(Layer2.Key_Step, Layer2.Key_Alter)]
          deriving (Eq, Show)
  -- |
type Key_Octave = ((MusicXML.CDATA, Maybe MusicXML.Yes_No), Layer1.Octave)
  -- |
type Time = ((Maybe MusicXML.CDATA, Maybe MusicXML.Time_A,
  MusicXML.Print_Style, MusicXML.Print_Object), Layer3.Time_B)
  -- |
type Staves = MusicXML.Staves
  -- |
type Part_Symbol = MusicXML.Part_Symbol
  -- |
type Instruments = MusicXML.Instruments
  -- |
type Clef = (
  (Maybe MusicXML.CDATA, Maybe MusicXML.Yes_No, Maybe MusicXML.Symbol_Size,
     MusicXML.Print_Style, MusicXML.Print_Object),
  (Layer2.Sign, Maybe Layer2.Line, Maybe Layer2.Clef_Octave_Change))
  -- |
type Staff_Details = MusicXML.Staff_Details
  -- |
type Transpose = MusicXML.Transpose
  -- |
type Directive = MusicXML.Directive
  -- |
type Measure_Style = MusicXML.Measure_Style

  -- |
abst_Score_Partwise :: Score_Partwise → Layer4.Score_Partwise
abst_Score_Partwise = (id × (id × map abst_Part))
  -- |
abst_Part :: Part → Layer4.Part
abst_Part = map abst_Measure · π2
  -- |
abst_Measure :: Measure → Layer4.Measure
abst_Measure = map abst_Music_Data · π2
  -- |
abst_Music_Data :: Music_Data → Layer4.Music_Data
abst_Music_Data (Music_Data_1 x) = Layer4.Music_Data_1 (abst_Note x)
abst_Music_Data (Music_Data_2 x) = Layer4.Music_Data_2 x
abst_Music_Data (Music_Data_3 x) = Layer4.Music_Data_3 x
abst_Music_Data (Music_Data_4 x) = Layer4.Music_Data_4 x
abst_Music_Data (Music_Data_5 x) = Layer4.Music_Data_5 (abst_Attributes x)
abst_Music_Data (Music_Data_6 x) = Layer4.Music_Data_6 x
abst_Music_Data (Music_Data_7 x) = Layer4.Music_Data_7 x
abst_Music_Data (Music_Data_8 x) = Layer4.Music_Data_8 x
abst_Music_Data (Music_Data_9 x) = Layer4.Music_Data_9 x
```

```
abst_Music_Data (Music_Data_10 x) = Layer4.Music_Data_10 x
abst_Music_Data (Music_Data_11 x) = Layer4.Music_Data_11 x
abst_Music_Data (Music_Data_12 x) = Layer4.Music_Data_12 x
abst_Music_Data (Music_Data_13 x) = Layer4.Music_Data_13 x
  -- |
abst_Note :: Note → Layer4.Note
abst_Note =
  (λ(x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13) →
    (abst_Note_ x1, fmap abst_Instrument x2,
    abst_Editorial_Voice x3, fmap abst_Type x4, fmap abst_Dot x5,
    fmap abst_Accidental x6, fmap abst_Time_Modification x7,
    fmap abst_Stem x8, fmap abst_Notehead x9, fmap abst_Staff x10,
    fmap abst_Beam x11, fmap abst_Notations x12, fmap abst_Lyric x13)) ·
  π2
  -- |
abst_Note_ :: Note_ → Layer4.Note_
abst_Note_ (Note_1 (x1, x2, x3)) =
  Layer4.Note_1 (abst_Grace x1, abst_Full_Note x2,
    fmap (abst_Tie × fmap abst_Tie) x3)
abst_Note_ (Note_2 (x1, x2, x3)) =
  Layer4.Note_2 (abst_Cue x1, abst_Full_Note x2, abst_Duration x3)
abst_Note_ (Note_3 (x1, x2, x3)) =
  Layer4.Note_3 (abst_Full_Note x1, abst_Duration x2,
    fmap (abst_Tie × fmap abst_Tie) x3)
  -- |
abst_Grace :: Grace → Layer4.Grace
abst_Grace = id
  -- |
abst_Cue :: Cue → Layer4.Cue
abst_Cue = id
  -- |
abst_Tie :: Tie → Layer4.Tie
abst_Tie = id
  -- |
abst_Full_Note :: Full_Note → Layer4.Full_Note
abst_Full_Note = (fmap id × abst_Full_Note_)
  -- |
abst_Full_Note_ :: Full_Note_ → Layer4.Full_Note_
abst_Full_Note_ (Full_Note_1 x) = Layer4.Full_Note_1 x
abst_Full_Note_ (Full_Note_2 x) = Layer4.Full_Note_2 x
abst_Full_Note_ (Full_Note_3 x) = Layer4.Full_Note_3 x
  -- |
abst_Duration :: Duration → Layer4.Duration
abst_Duration = id
  -- |
abst_Editorial_Voice :: Editorial_Voice → Layer4.Editorial_Voice
abst_Editorial_Voice = id
  -- |
abst_Instrument :: Instrument → Layer4.Instrument
abst_Instrument = id
  -- |
abst_Type :: Type → Layer4.Type
abst_Type = π2
  -- |
```

```
abst_Dot :: Dot → Layer4.Dot
abst_Dot = id
  -- |
abst_Accidental :: Accidental → Layer4.Accidental
abst_Accidental = π2
  -- |
abst_Time_Modification :: Time_Modification → Layer4.Time_Modification
abst_Time_Modification = id
  -- |
abst_Stem :: Stem → Layer4.Stem
abst_Stem = id
  -- |
abst_Notehead :: Notehead → Layer4.Notehead
abst_Notehead = id
  -- |
abst_Beam :: Beam → Layer4.Beam
abst_Beam = id
  -- |
abst_Staff :: Staff → Layer4.Staff
abst_Staff = id
  -- |
abst_Notations :: Notations → Layer4.Notations
abst_Notations = id
  -- |
abst_Editorial :: Editorial → Layer4.Editorial
abst_Editorial = id
  -- |
abst_Divisions :: Divisions → Layer4.Divisions
abst_Divisions = id
  -- |
abst_Key_Octave :: Key_Octave → Layer2.Key_Octave
abst_Key_Octave = π2
  -- |
abst_Staves :: Staves → Layer4.Staves
abst_Staves = id
  -- |
abst_Attributes :: Attributes → Layer4.Attributes
abst_Attributes (x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12) =
  (abst_Editorial x1, fmap abst_Divisions x2, fmap abst_Key x3,
    fmap abst_Time x4, fmap abst_Staves x5, fmap abst_Part_Symbol x6,
    fmap abst_Instruments x7, fmap abst_Clef x8, fmap abst_Staff_Details x9,
    fmap abst_Transpose x10, fmap abst_Directive x11,
    fmap abst_Measure_Style x12)
  -- |
abst_Lyric :: Lyric → Layer4.Lyric
abst_Lyric = id
  -- |
abst_Key :: Key → Layer4.Key
abst_Key = (abst_Key_ × fmap abst_Key_Octave) · π2
  -- |
abst_Key_ :: Key_ → Layer4.Key_
abst_Key_ (Key_1 x) = Layer4.Key_1 x
abst_Key_ (Key_2 x) = Layer4.Key_2 x
  -- |
```

```
abst_Time :: Time → Layer4.Time
abst_Time = π2
  -- |
abst_Part_Symbol :: Part_Symbol → Layer4.Part_Symbol
abst_Part_Symbol = id
  -- |
abst_Instruments :: Instruments → Layer4.Instruments
abst_Instruments = id
  -- |
abst_Clef :: Clef → Layer4.Clef
abst_Clef = π2
  -- |
abst_Staff_Details :: Staff_Details → Layer4.Staff_Details
abst_Staff_Details = id
  -- |
abst_Transpose :: Transpose → Layer4.Transpose
abst_Transpose = id
  -- |
abst_Directive :: Directive → Layer4.Directive
abst_Directive = id
  -- |
abst_Measure_Style :: Measure_Style → Layer4.Measure_Style
abst_Measure_Style = id

empty_Print_Style :: MusicXML.Print_Style
empty_Print_Style =
  ((Nothing, Nothing, Nothing, Nothing),
    (Nothing, Nothing, Nothing, Nothing),
    Nothing)
empty_Printout :: MusicXML.Printout
empty_Printout = (Nothing, Nothing, Nothing, Nothing)
empty_Level_Display :: MusicXML.Level_Display
empty_Level_Display = (Nothing, Nothing, Nothing)

  -- |
rep_Score_Partwise :: Layer4.Score_Partwise → Score_Partwise
rep_Score_Partwise = (id × (id ×
  (fmap (((++) · show) × id) ·
    uncurry zip · (([1 ..] :: [Int]) × id) · unzip ·
    fmap rep_Part)))
  -- |
rep_Part :: Layer4.Part → Part
rep_Part =< "P1", uncurry
  zip ·
  ((fmap f2 · uncurry zip ·
    ((fmap show · ([1 ..] :: [Int])) × id) ·
    unzip · fmap f1) × id) ·
  unzip · fmap rep_Measure >
  where f1 (a, b, c, d) = (a, (b, c, d))
    f2 (a, (b, c, d)) = (a, b, c, d)
  -- |
rep_Measure :: Layer4.Measure → Measure
rep_Measure =< ("1", Nothing, Nothing, Nothing), fmap rep_Music_Data >
```

```
  -- |
rep_Music_Data :: Layer4.Music_Data → Music_Data
rep_Music_Data (Layer4.Music_Data_1 x) = Music_Data_1 (rep_Note x)
rep_Music_Data (Layer4.Music_Data_2 x) = Music_Data_2 x
rep_Music_Data (Layer4.Music_Data_3 x) = Music_Data_3 x
rep_Music_Data (Layer4.Music_Data_4 x) = Music_Data_4 x
rep_Music_Data (Layer4.Music_Data_5 x) = Music_Data_5 (rep_Attributes x)
rep_Music_Data (Layer4.Music_Data_6 x) = Music_Data_6 x
rep_Music_Data (Layer4.Music_Data_7 x) = Music_Data_7 x
rep_Music_Data (Layer4.Music_Data_8 x) = Music_Data_8 x
rep_Music_Data (Layer4.Music_Data_9 x) = Music_Data_9 x
rep_Music_Data (Layer4.Music_Data_10 x) = Music_Data_10 x
rep_Music_Data (Layer4.Music_Data_11 x) = Music_Data_11 x
rep_Music_Data (Layer4.Music_Data_12 x) = Music_Data_12 x
rep_Music_Data (Layer4.Music_Data_13 x) = Music_Data_13 x
  -- |
rep_Note :: Layer4.Note → Note
rep_Note (x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13) =
    (emptyAttrs,
    (rep_Note_ x1, fmap rep_Instrument x2,
      rep_Editorial_Voice x3, fmap rep_Type x4, fmap rep_Dot x5,
      fmap rep_Accidental x6, fmap rep_Time_Modification x7,
      fmap rep_Stem x8, fmap rep_Notehead x9, fmap rep_Staff x10,
      fmap rep_Beam x11, fmap rep_Notations x12, fmap rep_Lyric x13))
  where emptyAttrs = (empty_Print_Style, empty_Printout,
    Nothing, Nothing, Nothing, Nothing, Nothing, Nothing)
  -- (\(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13) ->
  -- (abst_Note_ x1, fmap abst_Instrument x2,
  -- abst_Editorial_Voice x3, fmap abst_Type x4, fmap abst_Dot x5,
  -- fmap abst_Accidental x6, fmap abst_Time_Modification x7,
  -- fmap abst_Stem x8, fmap abst_Notehead x9, fmap abst_Staff x10,
  -- fmap abst_Beam x11, fmap abst_Notations x12, fmap abst_Lyric x13)) .
  -- p2
  -- |
rep_Note_ :: Layer4.Note_ → Note_
rep_Note_ (Layer4.Note_1 (x1, x2, x3)) =
  Note_1 (rep_Grace x1, rep_Full_Note x2,
    fmap (rep_Tie × fmap rep_Tie) x3)
rep_Note_ (Layer4.Note_2 (x1, x2, x3)) =
  Note_2 (rep_Cue x1, rep_Full_Note x2, rep_Duration x3)
rep_Note_ (Layer4.Note_3 (x1, x2, x3)) =
  Note_3 (rep_Full_Note x1, rep_Duration x2,
    fmap (rep_Tie × fmap rep_Tie) x3)
  -- |
rep_Grace :: Layer4.Grace → Grace
rep_Grace = id
  -- |
rep_Cue :: Layer4.Cue → Cue
rep_Cue = id
  -- |
rep_Tie :: Layer4.Tie → Tie
rep_Tie = id
  -- |
```

```
rep_Full_Note :: Layer4.Full_Note → Full_Note
rep_Full_Note = (fmap id × rep_Full_Note_)
  -- |
rep_Full_Note_ :: Layer4.Full_Note_ → Full_Note_
rep_Full_Note_ (Layer4.Full_Note_1 x) = Full_Note_1 x
rep_Full_Note_ (Layer4.Full_Note_2 x) = Full_Note_2 x
rep_Full_Note_ (Layer4.Full_Note_3 x) = Full_Note_3 x
  -- |
rep_Duration :: Layer4.Duration → Duration
rep_Duration = id
  -- |
rep_Editorial_Voice :: Layer4.Editorial_Voice → Editorial_Voice
rep_Editorial_Voice = id
  -- |
rep_Instrument :: Layer4.Instrument → Instrument
rep_Instrument = id
  -- |
rep_Type :: Layer4.Type → Type
rep_Type =< Nothing, id >
  -- |
rep_Dot :: Dot → Layer4.Dot
rep_Dot = id
  -- |
rep_Accidental :: Layer4.Accidental → Accidental
rep_Accidental =
   < (Nothing, Nothing, empty_Level_Display, empty_Print_Style),
  -- ((Nothing,Nothing,Nothing,Nothing),
  -- (Nothing,Nothing,Nothing,Nothing),Nothing)))
     id >
  -- |
rep_Time_Modification :: Layer4.Time_Modification → Time_Modification
rep_Time_Modification = id
  -- |
rep_Stem :: Layer4.Stem → Stem
rep_Stem = id
  -- |
rep_Notehead :: Layer4.Notehead → Notehead
rep_Notehead = id
  -- |
rep_Beam :: Layer4.Beam → Beam
rep_Beam = id
  -- |
rep_Staff :: Layer4.Staff → Staff
rep_Staff = id
  -- |
rep_Notations :: Layer4.Notations → Notations
rep_Notations = id
  -- |
rep_Editorial :: Layer4.Editorial → Editorial
rep_Editorial = id
  -- |
rep_Divisions :: Layer4.Divisions → Divisions
rep_Divisions = id
```

```
  -- |
rep_Key_Octave :: Layer2.Key_Octave → Key_Octave
rep_Key_Octave =< ("0", Nothing), id >
  -- |
rep_Staves :: Layer4.Staves → Staves
rep_Staves = id
  -- |
rep_Attributes :: Layer4.Attributes → Attributes
rep_Attributes (x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12) =
  (rep_Editorial x1, fmap rep_Divisions x2, fmap rep_Key x3,
    fmap rep_Time x4, fmap rep_Staves x5, fmap rep_Part_Symbol x6,
    fmap rep_Instruments x7, fmap rep_Clef x8, fmap rep_Staff_Details x9,
    fmap rep_Transpose x10, fmap rep_Directive x11,
    fmap rep_Measure_Style x12)
  -- |
rep_Lyric :: Layer4.Lyric → Lyric
rep_Lyric = id
  -- |
rep_Key :: Layer4.Key → Key
rep_Key =
  < (Nothing, empty_Print_Style, Nothing),
    rep_Key_ × fmap rep_Key_Octave >
  -- |
rep_Key_ :: Layer4.Key_ → Key_
rep_Key_ (Layer4.Key_1 x) = Key_1 x
rep_Key_ (Layer4.Key_2 x) = Key_2 x
  -- |
rep_Time :: Layer4.Time → Time
rep_Time =
  < (Nothing, Nothing, empty_Print_Style, Nothing),
    id >
  -- |
rep_Part_Symbol :: Layer4.Part_Symbol → Part_Symbol
rep_Part_Symbol = id
  -- |
rep_Instruments :: Layer4.Instruments → Instruments
rep_Instruments = id
  -- |
rep_Clef :: Layer4.Clef → Clef
rep_Clef =
  < (Nothing, Nothing, Nothing, empty_Print_Style, Nothing),
    id >
  -- |
rep_Staff_Details :: Layer4.Staff_Details → Staff_Details
rep_Staff_Details = id
  -- |
rep_Transpose :: Layer4.Transpose → Transpose
rep_Transpose = id
  -- |
rep_Directive :: Layer4.Directive → Directive
rep_Directive = id
  -- |
rep_Measure_Style :: Layer4.Measure_Style → Measure_Style
```

```
rep_Measure_Style = id


  -- |
map_Score_Partwise :: (Music_Data → Music_Data) →
  Score_Partwise → Score_Partwise
map_Score_Partwise f = (id × (id × fmap (map_Part f)))
  -- |
map_Part :: (Music_Data → Music_Data) → Part → Part
map_Part f = (id × fmap (map_Measure f))
  -- |
map_Measure :: (Music_Data → Music_Data) → Measure → Measure
map_Measure f = (id × fmap (map_Music_Data f))
  -- |
map_Music_Data :: (Music_Data → Music_Data) →
  Music_Data → Music_Data
map_Music_Data f = f
  -- map_Music_Data (Music_Data_1 x) = Music_Data_1 x
  -- map_Music_Data (Music_Data_2 x) = Music_Data_2 x
  -- map_Music_Data (Music_Data_3 x) = Music_Data_3 x
  -- map_Music_Data (Music_Data_4 x) = Music_Data_4 x
  -- map_Music_Data (Music_Data_5 x) = Music_Data_5 x
  -- map_Music_Data (Music_Data_6 x) = Music_Data_6 x
  -- map_Music_Data (Music_Data_7 x) = Music_Data_7 x
  -- map_Music_Data (Music_Data_8 x) = Music_Data_8 x
  -- map_Music_Data (Music_Data_9 x) = Music_Data_9 x
  -- map_Music_Data (Music_Data_10 x) = Music_Data_10 x
  -- map_Music_Data (Music_Data_11 x) = Music_Data_11 x
  -- map_Music_Data (Music_Data_12 x) = Music_Data_12 x
  -- map_Music_Data (Music_Data_13 x) = Music_Data_13 x
  -- map_Score_Partwise :: (Layer4.Score_Partwise -> Layer4.Score_Partwise) ->
  -- Score_Partwise -> Score_Partwise
  -- map_Score_Partwise f = (id >< (id >< f))
  -- map_Part :: (Layer4.Part -> Layer4.Part) -> Part -> Part
  -- map_Part f = split p1 (f . abst_Part)
  -- ( >< f)
  -- map_Measure :: (Layer4.Measure -> Layer4.Measure) -> Measure -> Measure
  -- map_Measure f = (id >< f )
```

4.8 Layer6

```
module Music.Analysis.MusicXML.Level6 (
  module Music.Analysis.MusicXML.Level6,
  ) where
import Music.Analysis.PF
import qualified Music.Analysis.MusicXML.Level5 as Layer5
import Data.Maybe (catMaybes)
import qualified Text.XML.MusicXML as MusicXML


  -- |
type Score_Partwise =
```

```
  (MusicXML.Document_Attributes, (MusicXML.Score_Header, [Part]))
  -- |
type Part = (MusicXML.ID, [Measure])
  -- |
type Measure = ((MusicXML.CDATA, Maybe MusicXML.Yes_No,
  Maybe MusicXML.Yes_No, Maybe MusicXML.Tenths), [Music_Data])
  -- |
data Music_Data =
    Music_Data_1 Note
  | Music_Data_2 MusicXML.Backup
  | Music_Data_3 MusicXML.Forward
  | Music_Data_4 MusicXML.Direction
  | Music_Data_5 Attributes
  | Music_Data_6 MusicXML.Harmony
  | Music_Data_7 MusicXML.Figured_Bass
  | Music_Data_8 MusicXML.Print
  | Music_Data_9 MusicXML.Sound
  | Music_Data_10 MusicXML.Barline
  | Music_Data_11 MusicXML.Grouping
  | Music_Data_12 MusicXML.Link
  | Music_Data_13 MusicXML.Bookmark
  | Music_Data_14 Annotation
    deriving (Eq, Show)
  -- |
type Annotation = (Maybe MusicXML.Start_Stop, MusicXML.PCDATA)
  -- |
type Note = Layer5.Note
  -- |
type Attributes = Layer5.Attributes

  -- |
abst_Score_Partwise :: Score_Partwise → Layer5.Score_Partwise
abst_Score_Partwise = (id × (id × map abst_Part))
  -- |
abst_Part :: Part → Layer5.Part
abst_Part = id × map abst_Measure
  -- |
abst_Measure :: Measure → Layer5.Measure
abst_Measure = id × catMaybes · fmap abst_Music_Data
  -- |
abst_Music_Data :: Music_Data → Maybe Layer5.Music_Data
abst_Music_Data (Music_Data_1 x) = return (Layer5.Music_Data_1 (abst_Note x))
abst_Music_Data (Music_Data_2 x) = return (Layer5.Music_Data_2 x)
abst_Music_Data (Music_Data_3 x) = return (Layer5.Music_Data_3 x)
abst_Music_Data (Music_Data_4 x) = return (Layer5.Music_Data_4 x)
abst_Music_Data (Music_Data_5 x) = (Just · Layer5.Music_Data_5 · abst_Attributes) x
abst_Music_Data (Music_Data_6 x) = return (Layer5.Music_Data_6 x)
abst_Music_Data (Music_Data_7 x) = return (Layer5.Music_Data_7 x)
abst_Music_Data (Music_Data_8 x) = return (Layer5.Music_Data_8 x)
abst_Music_Data (Music_Data_9 x) = return (Layer5.Music_Data_9 x)
abst_Music_Data (Music_Data_10 x) = return (Layer5.Music_Data_10 x)
abst_Music_Data (Music_Data_11 x) = return (Layer5.Music_Data_11 x)
abst_Music_Data (Music_Data_12 x) = return (Layer5.Music_Data_12 x)
abst_Music_Data (Music_Data_13 x) = return (Layer5.Music_Data_13 x)
```

```
abst_Music_Data (Music_Data_14 x) =
  return (Layer5.Music_Data_11 (abst_Annotation x))
  -- |
abst_Note :: Note → Layer5.Note
abst_Note = id
  -- |
abst_Attributes :: Attributes → Layer5.Attributes
abst_Attributes = id
  -- |
abst_Annotation :: Annotation → MusicXML.Grouping
abst_Annotation (a, ann) =
  ((abst_SSS a, "1", Just "annotation"), [(Nothing, ann)])
  -- |
abst_SSS :: Maybe MusicXML.Start_Stop → MusicXML.Start_Stop_Single
abst_SSS (Just MusicXML.Start_Stop_1) = MusicXML.Start_Stop_Single_1
abst_SSS (Just MusicXML.Start_Stop_2) = MusicXML.Start_Stop_Single_2
abst_SSS Nothing = MusicXML.Start_Stop_Single_3

  -- |
rep_Score_Partwise :: Layer5.Score_Partwise → Score_Partwise
rep_Score_Partwise = (id × (id × fmap rep_Part))
  -- |
rep_Part :: Layer5.Part → Part
rep_Part = id × fmap rep_Measure
  -- |
rep_Measure :: Layer5.Measure → Measure
rep_Measure = id × map rep_Music_Data
  -- |
rep_Music_Data :: Layer5.Music_Data → Music_Data
rep_Music_Data (Layer5.Music_Data_1 x) = Music_Data_1 (rep_Note x)
rep_Music_Data (Layer5.Music_Data_2 x) = Music_Data_2 x
rep_Music_Data (Layer5.Music_Data_3 x) = Music_Data_3 x
rep_Music_Data (Layer5.Music_Data_4 x) = Music_Data_4 x
rep_Music_Data (Layer5.Music_Data_5 x) = Music_Data_5 (rep_Attributes x)
rep_Music_Data (Layer5.Music_Data_6 x) = Music_Data_6 x
rep_Music_Data (Layer5.Music_Data_7 x) = Music_Data_7 x
rep_Music_Data (Layer5.Music_Data_8 x) = Music_Data_8 x
rep_Music_Data (Layer5.Music_Data_9 x) = Music_Data_9 x
rep_Music_Data (Layer5.Music_Data_10 x) = Music_Data_10 x
rep_Music_Data (Layer5.Music_Data_11 x) =
  [Music_Data_14 , Music_Data_11] (rep_Annotation x)
rep_Music_Data (Layer5.Music_Data_12 x) = Music_Data_12 x
rep_Music_Data (Layer5.Music_Data_13 x) = Music_Data_13 x
  -- |
rep_Note :: Layer5.Note → Note
rep_Note = id
  -- |
rep_Attributes :: Layer5.Attributes → Attributes
rep_Attributes = id
  -- |
rep_Annotation :: MusicXML.Grouping → Annotation + MusicXML.Grouping
rep_Annotation ((a, _, Just "annotation"), [(_, ann)]) = i1 (rep_SSS a, ann)
rep_Annotation grouping = i2 grouping
  -- |
```

```
rep_SSS :: MusicXML.Start_Stop_Single → Maybe MusicXML.Start_Stop
rep_SSS MusicXML.Start_Stop_Single_1 = Just MusicXML.Start_Stop_1
rep_SSS MusicXML.Start_Stop_Single_2 = Just MusicXML.Start_Stop_2
rep_SSS MusicXML.Start_Stop_Single_3 = Nothing


  -- |
map_Score_Partwise :: (Music_Data → Music_Data) →
  Score_Partwise → Score_Partwise
map_Score_Partwise f = (id × (id × fmap (map_Part f)))
  -- |
map_Part :: (Music_Data → Music_Data) → Part → Part
map_Part f = (id × fmap (map_Measure f))
  -- |
map_Measure :: (Music_Data → Music_Data) → Measure → Measure
map_Measure f = (id × fmap (map_Music_Data f))
  -- |
map_Music_Data :: (Music_Data → Music_Data) →
  Music_Data → Music_Data
map_Music_Data f = f
  -- map_Music_Data (Music_Data_1 x) = Music_Data_1 x
  -- map_Music_Data (Music_Data_2 x) = Music_Data_2 x
  -- map_Music_Data (Music_Data_3 x) = Music_Data_3 x
  -- map_Music_Data (Music_Data_4 x) = Music_Data_4 x
  -- map_Music_Data (Music_Data_5 x) = Music_Data_5 x
  -- map_Music_Data (Music_Data_6 x) = Music_Data_6 x
  -- map_Music_Data (Music_Data_7 x) = Music_Data_7 x
  -- map_Music_Data (Music_Data_8 x) = Music_Data_8 x
  -- map_Music_Data (Music_Data_9 x) = Music_Data_9 x
  -- map_Music_Data (Music_Data_10 x) = Music_Data_10 x
  -- map_Music_Data (Music_Data_11 x) = Music_Data_11 x
  -- map_Music_Data (Music_Data_12 x) = Music_Data_12 x
  -- map_Music_Data (Music_Data_13 x) = Music_Data_13 x
  -- map_Score_Partwise :: (Layer4.Score_Partwise -> Layer4.Score_Partwise) ->
  -- Score_Partwise -> Score_Partwise
  -- map_Score_Partwise f = (id >< (id >< f))
  -- map_Part :: (Layer4.Part -> Layer4.Part) -> Part -> Part
  -- map_Part f = split p1 (f . abst_Part)
  -- ( >< f)
  -- map_Measure :: (Layer4.Measure -> Layer4.Measure) -> Measure -> Measure
  -- map_Measure f = (id >< f )
```

5 Formats

5.1 Haskore

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- This module implements lite interface to Haskore
```

```
  --
  -- Bugs:
  -- - Chords
  -- - more than one Divisions(changes duration)
module Music.Analysis.Haskore where
import qualified Haskore ()
import qualified Haskore.Music as HMusic ()
import qualified Haskore.Basic.Pitch as HPitch ()
  -- import qualified Haskore.Basic.Duration as HDuration
import qualified Haskore.Melody as HMelody ()
  -- import qualified Haskore.Melody.Standard as HMelodyStd
  -- import qualified Haskore.Music.Rhythmic as HRhythmic
import qualified Haskore.Music.GeneralMIDI as GeneralMIDI ()
import qualified Medium.Controlled.List as Medium ()
import qualified Haskore.Interface.MIDI.Render as Render ()
```

5.2 ABC

```
module Music.Analysis.ABC where
import Music.Analysis.Base
import Data.List
data ABCMusic = ABCMusic ABCMetaData [ABCMusicData]
  deriving (Eq)
data ABCMetaData = ABCMetaData ABCIndex ABCTitle ABCMeter ABCKey
  deriving (Eq)
data ABCIndex = ABCIndex deriving (Eq, Show)
data ABCTitle = ABCTitle deriving (Eq, Show)
data ABCMeter = ABCMeter deriving (Eq, Show)
data ABCKey = ABCKey deriving (Eq, Show)
data ABCMusicData = Single ABCNote
   | Tuplet Int [ABCMusicData]
   | Chord [ABCMusicData]
   | Tie ABCMusicData ABCMusicData
   | Slur [ABCMusicData]
   | Staccato ABCMusicData
   | GraceNotes [ABCNote] ABCMusicData
   | Symbol String ABCNote
  deriving (Eq)
data ABCNote = Pitch Pitch Octave Accident Duration Dotted
   | Rest Bool Duration
  deriving (Eq)
data Pitch = C | D | E | F | G | A | B
  deriving (Show, Eq)
data Accident = Accident (Maybe Accidental)
  deriving Eq
data Accidental = Sharp (Number, Int) | Natural | Flat (Number, Int)
  deriving (Eq)
```

```
data Dotted = Increase Int | Decrease Int
  deriving (Eq)
type Octave = Int
data Duration = Duration (Int, Int)
  deriving (Eq)

instance Show ABCMusic where
  show (ABCMusic _ xs) = concat (intersperse " " (map show xs))
instance Show ABCMusicData where
  show (Single x) = show x
  show (Tuplet n xs) = "(" ++ show n ++ show xs
  show (Chord xs) = "[" ++ show xs ++ "]"
  show (Tie x y) = show x ++ show y
  show (Slur xs) = "(" ++ show xs ++ ")"
  show (Staccato x) = "." ++ show x
  show (GraceNotes xs y) = "{" ++ show xs ++ "}" ++ show y
  show (Symbol _ y) = show y
instance Show ABCNote where
  show (Pitch a b c d e) = show a ++ show b ++ show c ++ show d ++ show e
  show (Rest True b) = "z" ++ show b
  show (Rest False b) = "x" ++ show b
instance Show Accident where
  show (Accident (Just x)) = show x
  show (Accident Nothing) = []
instance Show Accidental where
  show (Sharp n) = "^" ++ show n
  show (Natural) = "="
  show (Flat n) = "_" ++ show n
instance Show Dotted where
  show (Increase n) | n ≡ 0 = []
     | otherwise = ">" ++ show n
  show (Decrease n) | n ≡ 0 = []
     | otherwise = "<" ++ show n
instance Show Duration where
  show (Duration (n, 0)) = show n
  show (Duration (1, n)) = "/" ++ show n
  show (Duration (0, n)) = "/" ++ show n
  show (Duration (x, y)) = show x ++ "/" ++ show y
```

5.3 Lilypond

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- Implements interface to Lilypond
module Music.Analysis.Lilypond where
```

6 Translations

6.1 Abstract2ABC

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- This module implements
module Music.Analysis.Abstract2ABC where
  -- import Music.Analysis.Base
import Music.Analysis.PF
import Music.Analysis.Abstract.Settings
import qualified Music.Analysis.Abstract.Motive as Motive
  -- import qualified Music.Analysis.Abstract.Zip as Abstract
import qualified Music.Analysis.Abstract.Melodic as Melodic
import qualified Music.Analysis.Abstract.Rhythm as Rhythm
import Music.Analysis.ABC as ABC
import Data.Ratio


mk_Music :: Motive.Motive (Melodic.MelodicClass, Rhythm.RhythmAbsolute)
   -> ABCMusic
mk_Music = uncurry ABCMusic· < mk_Meta, mk_Notes >
  -- |
mk_Meta :: Motive.Motive (Melodic.MelodicClass, Rhythm.RhythmAbsolute)
   -> ABCMetaData
mk_Meta = mk_Meta_aux · π1 · Motive.fromMotive where
  mk_Meta_aux s =
    let a = ABCIndex s
        b = maybe ABCTitle ABCTitle (getText "title" s)
        c = maybe ABCMeter ABCMeter (getText "meter" s)
        d = maybe ABCKey ABCKey (getText "key" s)
    in ABCMetaData a b c d
  -- |
mk_Notes :: Motive.Motive (Melodic.MelodicClass, Rhythm.RhythmAbsolute)
   -> [ABC.ABCMusicData]
mk_Notes = map (ABC.Single · mk_Note) · π2 · Motive.fromMotive
  -- |
mk_Note :: (Melodic.MelodicClass, Rhythm.RhythmAbsolute) -> ABC.ABCNote
mk_Note (x, y) =
  let (duration, dotted) = mk_Duration y
  in case mk_Note' x of
    Just (pitch, accidental) ->
      ABC.Pitch pitch 0 accidental duration dotted
    Nothing -> ABC.Rest True duration
mk_Duration :: Rhythm.RhythmAbsolute -> (ABC.Duration, ABC.Dotted)
mk_Duration (x, y) = (ABC.Duration (denominator x, numerator x), ABC.Increase y)
mk_Note' :: Melodic.MelodicClass -> Maybe (ABC.Pitch, ABC.Accident)
mk_Note' Nothing = Nothing
mk_Note' (Just (x, y)) = Just (mk_Pitch x, mk_Accident y)
mk_Pitch :: Melodic.PitchClass -> ABC.Pitch
```

```
mk_Pitch Melodic.C = ABC.C
mk_Pitch Melodic.D = ABC.D
mk_Pitch Melodic.E = ABC.E
mk_Pitch Melodic.F = ABC.F
mk_Pitch Melodic.G = ABC.G
mk_Pitch Melodic.A = ABC.A
mk_Pitch Melodic.B = ABC.B
```

```
mk_Accident :: Melodic.Accident → ABC.Accident
mk_Accident Nothing = ABC.Accident Nothing
mk_Accident (Just n) | n > 0 = ABC.Accident (Just (ABC.Sharp (n, 1)))
    | n ≡ 0 = ABC.Accident (Just (ABC.Natural))
    | n < 0 = ABC.Accident (Just (ABC.Flat (n, 1)))
    | otherwise = ABC.Accident Nothing
```

6.2 Abstract2Lilypond

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- Implements interface to Lilypond
module Music.Analysis.Abstract2Lilypond where
```

```
import Music.Analysis.PF ((×), (+), grd, < ·, · >, e2m, hyloL)
import Music.Analysis.Base (IntegerNumber)
import Music.Analysis.Abstract.Melodic (Accident)
import Music.Analysis.Abstract.Zip (VoiceZipNode)
import Data.Tuple (uncurry)
import Data.List ((++))
import Data.Function ((·), id)
import Data.Maybe (maybe)
import Data.Either ([·, ·])
import Data.Char (String)
import Data.Ord (Ord (..))
import Data.Eq (Eq (..))
import Prelude (Num (..), show)
```

```
  -- * Output Functions
  -- |
output :: VoiceZipNode → String
output = uncurry (++) ·
  (maybe "r" (uncurry (++) · (show × outputAccidents)) ×
    uncurry (++) · (show × outputDots))
  -- |
outputDots :: IntegerNumber → String
outputDots =
  hyloL (maybe [] (uncurry (++)))
    (e2m · (id+ < ".", pred >) · grd (⩽ 0))
  where pred = λx → x − 1
  -- |
outputAccidents :: Accident → String
```

```
outputAccidents =
  maybe "" (hyloL (maybe [] (uncurry (++)))
    (e2m · ((,) + [< "is", pred >,
      < "es", succ >] · grd (>0)) ·
      grd (≡ 0)))
  where pred = λx → x − 1
    succ = λx → x + 1
```

6.3 MusicXML2Abstract

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- This module implements common types
module Music.Analysis.MusicXML2Abstract where
import Music.Analysis.Abstract.Settings
import Music.Analysis.Abstract.Motive
  -- import Music.Analysis.Abstract.Melodic
  -- import Music.Analysis.Abstract.Rhythm
  -- import Text.XML.MusicXML.Partwise as Partwise
import Music.Analysis.MusicXML.Level2 as Level2
import Music.Analysis.MusicXML.Level1 as Level1
import Music.Analysis.Abstract.Melodic as Melodic
import Music.Analysis.Abstract.Rhythm as Rhythm
  -- import Music.Analysis.Abstract.Zip as Abstract
import Data.Ratio


mk_Music :: Level2.Score_Partwise →
  Motive (Melodic.MelodicClass, Rhythm.RhythmAbsolute)
  -- [(Melodic.MelodicClass, Rhythm.RhythmAbsolute)]
mk_Music = mkMotive empty · concat · map mk_MusicData
mk_MusicData :: [Level2.Music_Data] →
  [(Melodic.MelodicClass, Rhythm.RhythmAbsolute)]
mk_MusicData = map mk_Note · map (λ(Level2.Music_Data_1 x) → x) · filter p
  where p (Level2.Music_Data_1 _) = True
    p _ = False
  -- |
mk_Note :: Level2.Note → (Melodic.MelodicClass, Rhythm.RhythmAbsolute)
mk_Note (Level2.Note_3 (a1, a2), b, c, d) =
  (mk_MelodicNode a1 d, mk_RhythmNode a2 b c)
  -- |
mk_MelodicNode :: Level2.Full_Note → Maybe Level2.Accidental
  → Melodic.MelodicClass
mk_MelodicNode (Level2.Full_Note_1 (a, _, _)) _ = Just (mk_Pitch a, Nothing)
mk_MelodicNode (Level2.Full_Note_3 _) _ = Nothing
  -- |
mk_Pitch :: Level1.Step → Melodic.PitchClass
mk_Pitch x = case x of
  Level1.C → Melodic.C; Level1.D → Melodic.D;
```

```
  Level1.E → Melodic.E; Level1.F → Melodic.F;
  Level1.G → Melodic.G; Level1.A → Melodic.A;
  Level1.B → Melodic.B;
  -- |
mk_RhythmNode :: Level2.Duration → Maybe Level2.Type → [Level2.Dot]
  → Rhythm.RhythmAbsolute
mk_RhythmNode x _ ly = (x % 1, length ly)
```

6.4 MusicXML2Haskore

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- This module implements lite interface to Haskore
  --
  -- Bugs:
  -- - Chords
  -- - more than one Divisions(changes duration)
module Music.Analysis.MusicXML2Haskore where
  -- import Music.Analysis.Base
  -- import Music.Analysis.PF (p1, p2, (><), grd)
  -- import Music.Analysis.Settings (union)
  -- import Music.Analysis.Motive (Motive, toMotive, fromMotive)
import Music.Analysis.PF
  -- cataMotive, splitMotiveList)
  -- import Music.Analysis.Melodic (MelodicNode)
  -- import Music.Analysis.Rhythm (RhythmNode)
  -- import Music.Analysis.Zip (VoiceZipNode)
  -- import Music.Analysis.Voices (MultiVoiceNode)
  -- import Music.Analysis.Instruments (MultiInstrumentNode, settings)
import Music.Analysis.Interface ()
  -- import Haskore (Music(..), Pitch, Dur, PitchClass)
  -- import Haskore.Basics (Music(..), Pitch, Dur, PitchClass(..))
  -- import Haskore.Performance ()
import qualified Haskore ()
import qualified Haskore.Music as HMusic
import qualified Haskore.Basic.Pitch as HPitch
  -- import qualified Haskore.Basic.Duration as HDuration
import qualified Haskore.Melody as HMelody
  -- import qualified Haskore.Melody.Standard as HMelodyStd
  -- import qualified Haskore.Music.Rhythmic as HRhythmic
import qualified Haskore.Music.GeneralMIDI as GeneralMIDI
import qualified Medium.Controlled.List as Medium
import qualified Haskore.Interface.MIDI.Render as Render
  -- import qualified Haskore.Performance
import Data.List
import Data.Maybe
  -- import Data.Function (const, (.))
  -- import Data.Either (either)
  -- import Data.Maybe (Maybe(..), isNothing)
```

```
  -- import Data.Tuple (uncurry)
import Numeric.NonNegative.Wrapper as NonNeg
import qualified Music.Analysis.MusicXML as IMusicXML
import qualified Music.Analysis.MusicXML.Level5 as Layer5
import qualified Music.Analysis.MusicXML.Level1 as Layer1
import qualified Text.XML.MusicXML as MusicXML
import qualified Text.XML.MusicXML.Partwise as Partwise
  -- import Data.Ratio
import System.Info
import System.Cmd
import System.Exit
import Prelude


  -- |
from_Score_Partwise :: MusicXML.Score_Partwise → MusicXML.Score_Partwise
from_Score_Partwise = (id × (id × fmap from_Part))
  -- |
from_Part :: Partwise.Part → Partwise.Part
from_Part = (id × fmap from_Measure)
  -- |
from_Measure :: Partwise.Measure → Partwise.Measure
from_Measure = (id × fmap from_Music_Data)
  -- |
from_Music_Data :: MusicXML.Music_Data_ → MusicXML.Music_Data_
from_Music_Data (MusicXML.Music_Data_1 x) =
  MusicXML.Music_Data_1 (from_Note x)
from_Music_Data (MusicXML.Music_Data_2 x) = MusicXML.Music_Data_2 x
from_Music_Data (MusicXML.Music_Data_3 x) = MusicXML.Music_Data_3 x
from_Music_Data (MusicXML.Music_Data_4 x) = MusicXML.Music_Data_4 x
from_Music_Data (MusicXML.Music_Data_5 x) = MusicXML.Music_Data_5 x
from_Music_Data (MusicXML.Music_Data_6 x) = MusicXML.Music_Data_6 x
from_Music_Data (MusicXML.Music_Data_7 x) = MusicXML.Music_Data_7 x
from_Music_Data (MusicXML.Music_Data_8 x) = MusicXML.Music_Data_8 x
from_Music_Data (MusicXML.Music_Data_9 x) = MusicXML.Music_Data_9 x
from_Music_Data (MusicXML.Music_Data_10 x) = MusicXML.Music_Data_10 x
from_Music_Data (MusicXML.Music_Data_11 x) = MusicXML.Music_Data_11 x
from_Music_Data (MusicXML.Music_Data_12 x) = MusicXML.Music_Data_12 x
from_Music_Data (MusicXML.Music_Data_13 x) = MusicXML.Music_Data_13 x
  -- |
from_Note :: MusicXML.Note → MusicXML.Note
from_Note = id   -- (id >< ((\(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13) ->
  -- x4' <- fmap abst_Type x4
  -- (abst_Note_ x1, fmap abst_Instrument x2,
  -- abst_Editorial_Voice x3,
  -- (maybe Nothing id . fmap from_Type) x4, fmap from_Dot x5,
  -- (maybe Nothing id . fmap from_Accidental) x6,
  -- fmap from_Time_Modification x7, fmap from_Stem x8,
  -- fmap from_Notehead x9, fmap from_Staff x10, fmap from_Beam x11,
  -- fmap from_Notations x12, fmap from_Lyric x13))))


  -- |
group_Part :: Partwise.Part → [[[[MusicXML.Note]]]]
group_Part =
```

```
  fmap (concat · group_Measure) · π2
  -- |
group_Measure :: Partwise.Measure → [[[[MusicXML.Note]]]]
group_Measure =
  (fmap · fmap) (groupBy group_Instrument · sortBy cmp_Instrument) ·
  fmap (groupBy group_Staff · sortBy cmp_Staff) ·
  groupBy group_Voice · sortBy cmp_Voice ·
  toNote · π2
  -- |
toNote :: [MusicXML.Music_Data_] → [MusicXML.Note]
toNote = catMaybes · fmap f
  where f (MusicXML.Music_Data_1 x) = return x
        f _ = fail []
  -- |
group_Music_Data_ :: (MusicXML.Note → MusicXML.Note → Bool) →
  MusicXML.Music_Data_ → MusicXML.Music_Data_ → Bool
group_Music_Data_ p (MusicXML.Music_Data_1 x) (MusicXML.Music_Data_1 y) = p x y
group_Music_Data_ _ _ _ = False
  -- |
group_Instrument :: MusicXML.Note → MusicXML.Note → Bool
group_Instrument = (λ(_, (_, x2, _, _, _, _, _, _, _, _, _, _, _))
  (_, (_, y2, _, _, _, _, _, _, _, _, _, _, _)) → x2 ≡ y2)
  -- |
group_Voice :: MusicXML.Note → MusicXML.Note → Bool
group_Voice = (λ(_, (_, _, (_, _, x3), _, _, _, _, _, _, _, _, _))
  (_, (_, _, (_, _, y3), _, _, _, _, _, _, _, _, _)) → x3 ≡ y3)
  -- |
group_Staff :: MusicXML.Note → MusicXML.Note → Bool
group_Staff = (λ(_, (_, _, _, _, _, _, _, _, _, x10, _, _, _))
  (_, (_, _, _, _, _, _, _, _, _, y10, _, _, _)) → x10 ≡ y10)
  -- |
cmp_Instrument :: MusicXML.Note → MusicXML.Note → Ordering
cmp_Instrument = (λ(_, (_, x2, _, _, _, _, _, _, _, _, _, _, _))
  (_, (_, y2, _, _, _, _, _, _, _, _, _, _, _)) → x2 `compare` y2)
  -- |
cmp_Voice :: MusicXML.Note → MusicXML.Note → Ordering
cmp_Voice = (λ(_, (_, _, (_, _, x3), _, _, _, _, _, _, _, _, _))
  (_, (_, _, (_, _, y3), _, _, _, _, _, _, _, _, _)) → x3 `compare` y3)
  -- |
cmp_Staff :: MusicXML.Note → MusicXML.Note → Ordering
cmp_Staff = (λ(_, (_, _, _, _, _, _, _, _, _, x10, _, _, _))
  (_, (_, _, _, _, _, _, _, _, _, y10, _, _, _)) → x10 `compare` y10)
  -- |
get_Instrument :: MusicXML.Note → Maybe MusicXML.Instrument
get_Instrument (_, (_, x2, _, _, _, _, _, _, _, _, _, _, _)) = x2
  -- |
get_Voice :: MusicXML.Note → Maybe MusicXML.Voice
get_Voice (_, (_, _, (_, _, x3), _, _, _, _, _, _, _, _, _)) = x3
  -- |
get_Staff :: MusicXML.Note → Maybe MusicXML.Staff
get_Staff (_, (_, _, _, _, _, _, _, _, _, x10, _, _, _)) = x10

  -- |
group_Measure' :: Partwise.Measure →
```

```
  [(Maybe MusicXML.Instrument,
    [(Maybe MusicXML.Staff,
      [(Maybe MusicXML.Voice, [MusicXML.Note])])])]
group_Measure' =
  map (id × map (id × map (< headM · map get_Voice, id >))) ·
  map (id × map (< (headM · headM) · (map · map) get_Staff, id >)) ·
  map (< (headM · headM · headM) · (map · map · map) get_Instrument, id >) ·
  group_Measure
  -- |
headM :: Monad m ⇒ [m a] → m a
headM [] = fail "empty list"
headM (x : _) = x


  -- |
toMedium_ :: [[[[Medium.T control a]]]] → Medium.T control a
toMedium_ =
  Medium.parallel ·
  ((map) Medium.parallel) ·
  ((map · map) Medium.parallel) ·
  ((map · map · map) Medium.serial)
  -- |
toMedium :: [[[[a]]]] → Medium.T control a
toMedium =
  Medium.parallel ·
  ((map) Medium.parallel) ·
  ((map · map) Medium.parallel) ·
  ((map · map · map) Medium.serial) ·
  (map · map · map · map) Medium.prim
  -- toMedium' :: [[[[a]]]] -> Medium.T () a
  -- toMedium' =
  -- (Medium.Control () . Medium.parallel) .
  -- ((map) (Medium.Control () . Medium.parallel)) .
  -- ((map . map) (Medium.Control () . Medium.parallel)) .
  -- ((map . map . map) (Medium.Control () . Medium.serial)) .
  -- (map . map . map . map) Medium.prim
  -- |
toMedium' :: [(Maybe MusicXML.Instrument,
    [(Maybe MusicXML.Staff,
      [(Maybe MusicXML.Voice, [MusicXML.Note])])])] →
        Medium.T ControlID MusicXML.Note
toMedium' =
  Medium.parallel ·
  (map (uncurry Medium.Control ·
    (Control_Instrument × Medium.parallel))) ·
  (map (id × map (uncurry Medium.Control ·
    (Control_Staff × Medium.parallel)))) ·
  (map (id × map (id × map (uncurry Medium.Control ·
    (Control_Voice × Medium.serial))))) ·
  (map (id × map (id × map (id × map Medium.prim)))) ·
  id
  -- where ctrl = Medium.Control
  -- |
data ControlID = Control_Instrument (Maybe MusicXML.Instrument)
```

```
    | Control_Staff (Maybe MusicXML.Staff)
    | Control_Voice (Maybe MusicXML.Voice)
  deriving (Eq, Show)

  -- |
abst_Step :: MusicXML.Step → Maybe Layer1.Step
abst_Step = IMusicXML.abst_Step
  -- |
abst_Octave :: MusicXML.Octave → Layer1.Octave
abst_Octave = IMusicXML.abst_Octave
  -- |
abst_Alter :: MusicXML.Alter → Maybe Layer1.Alter
abst_Alter = IMusicXML.abst_Alter
  -- |
toClass :: Layer1.Step → Maybe Layer1.Alter → HPitch.Class
toClass step Nothing = toEnum (3 * (fromEnum step) + 1)
toClass step (Just alter) =
  toEnum (3 * (fromEnum step) + 1 + (truncate ((3 / 2) * alter)))
  -- |
abst_Pitch :: MusicXML.Pitch → (Layer1.Octave, HPitch.Class)
abst_Pitch =
  swap · (uncurry toClass × id) · unflatl · IMusicXML.abst_Pitch
  -- |
abst_Full_Note_ :: MusicXML.Full_Note_ → Maybe HPitch.T
abst_Full_Note_ (MusicXML.Full_Note_1 x) = return (abst_Pitch x)
abst_Full_Note_ _ = fail []
  -- |
abst_Full_Note :: MusicXML.Full_Note → Maybe HPitch.T
abst_Full_Note = abst_Full_Note_ · π2
  -- |
abst_Note_ :: MusicXML.Note_ → (Maybe HPitch.T, Maybe Layer5.Duration)
abst_Note_ (MusicXML.Note_1 _) = (Nothing, Nothing)
abst_Note_ (MusicXML.Note_2 x) =
  ((abst_Full_Note × Just · abst_Duration) · π2 · unflatr) x
  -- in maybe Nothing (\a' -> return (a', b)) a
abst_Note_ (MusicXML.Note_3 x) =
  ((abst_Full_Note × Just · abst_Duration) · π1 · unflatl) x
  -- in maybe Nothing (\a' -> return (a', b)) a
  -- |
abst_Duration :: MusicXML.Duration → Layer5.Duration
abst_Duration = IMusicXML.abst_Duration
  -- |
toDur :: Layer5.Duration → HMusic.Dur
toDur n = NonNeg.fromNumber (fromIntegral n)
  -- |
abst_Note :: MusicXML.Note → HMusic.T HPitch.T
abst_Note (_, (a, _, _, _, _, _, _, _, _, _, _, _)) =
  case abst_Note_ a of
    (Just a1, Just a2) → HMusic.atom (toDur a2) (Just a1)
    (Nothing, Just a2) → HMusic.rest (toDur a2)
    (_, Nothing) → HMusic.rest 0
  -- |
abst_Note' :: MusicXML.Note → HMelody.T MusicXML.Note
abst_Note' n@(_, (a, _, _, _, _, _, _, _, _, _, _, _)) =
```

```
  case abst_Note_ a of
    (Just a1, Just a2) → HMusic.atom (toDur a2) (Just (HMelody.Note n a1))
    (Nothing, Just a2) → HMusic.rest (toDur a2)
    (_, Nothing) → HMusic.rest 0
  -- Just (a1,a2) -> Just (toDur a2, a1)
  -- Nothing -> Nothing
  -- map_abst_Note (a,b,c) =
  -- (id >< fmap swap) . split id abst_Note'


  -- |
measure2haskore :: Partwise.Measure → HMelody.T MusicXML.Note
measure2haskore = toMedium_ · (map · map · map · map) abst_Note' · group_Measure
  -- |
part2haskore :: Partwise.Part → HMelody.T MusicXML.Note
part2haskore = Medium.Serial · map measure2haskore · π2
  -- |
partwise2haskore :: Partwise.Score_Partwise → HMelody.T MusicXML.Note
partwise2haskore = Medium.Parallel · fmap part2haskore · π2 · π2
toMidi :: MusicXML.MusicXMLDoc → GeneralMIDI.T
toMidi (MusicXML.Score (MusicXML.Partwise music)) =
  (GeneralMIDI.fromMelodyNullAttr GeneralMIDI.AcousticGrandPiano ·
  fmap fun · partwise2haskore) music
  where fun (HMusic.Atom x Nothing) = HMusic.Atom x Nothing
    fun (HMusic.Atom x (Just (HMelody.Note _ y))) =
      HMusic.Atom x (Just (HMelody.Note () y))
toMidi (_) =
  (GeneralMIDI.fromMelodyNullAttr GeneralMIDI.AcousticGrandPiano ·
  fmap fun · Medium.Parallel) []
  where fun (HMusic.Atom x Nothing) = HMusic.Atom x Nothing
    fun (HMusic.Atom x (Just (HMelody.Note _ y))) =
      HMusic.Atom x (Just (HMelody.Note () y))
saveMidi :: FilePath → GeneralMIDI.T → IO ()
saveMidi filepath = Render.fileFromGeneralMIDIMusic filepath
playMidi :: FilePath → GeneralMIDI.T → IO ExitCode
playMidi filepath music =
  Render.fileFromGeneralMIDIMusic filepath music ≫ playGeneric filepath
playGeneric :: FilePath → IO ExitCode
playGeneric filepath =
  case System.Info.os of
    "mingw32" → rawSystem "mplay32" [filepath]
    "linux" → rawSystem "playmidi" ["-rf", filepath]
    _ →       rawSystem "timidity" ["-B8,9", filepath]
```

6.5 MusicXML2ABC

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- This module implements
```

```
module Music.Analysis.MusicXML2ABC where
import Music.Analysis.MusicXML.Level2 as Level2
import Music.Analysis.ABC as ABC
import qualified Music.Analysis.Abstract2ABC as Abstract2ABC
import qualified Music.Analysis.MusicXML2Abstract as MusicXML2Abstract

mk_Music :: Level2.Score_Partwise → ABC.ABCMusic
mk_Music = Abstract2ABC.mk_Music · MusicXML2Abstract.mk_Music
```

7 Executable

7.1 Script interface

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  --
module Music.Analysis.Script where
import Script
import qualified Text.XML.MusicXML as MusicXML
import Music.Analysis.PF
import Music.Analysis.MusicXML2Haskore as IHaskore
import Music.Analysis.MusicXML as Interface
import Music.Analysis.MusicXML.Functions
import Music.Analysis.MusicXML.Level5 as Layer5
import Music.Analysis.MusicXML.Level4 as Layer4
  -- import Music.Analysis.Definition.Layer3 as Layer3
  -- import Music.Analysis.Definition.Layer2 as Layer2
import Text.XML.HaXml.OneOfN
import Control.Exception (throw, Exception (..))
import qualified Haskore.Music.GeneralMIDI
  -- import System.Directory
  -- import Text.XML.HaXml.XmlContent
  -- import Data.List
import System.IO (hFlush, stdout)
import Prelude


  -- |
procFilter :: Filter → MusicXML.MusicXMLDoc → MusicXML.MusicXMLDoc
procFilter (Filter Filter_select_note Nothing) =
  procFilter (Filter Filter_select_note (Just Filter_mode_yes))
procFilter (Filter Filter_select_note (Just Filter_mode_yes)) =
  mapMusicXML filterNote
procFilter (Filter Filter_select_note (Just Filter_mode_no)) =
  mapMusicXML filterNotNote
procFilter (Filter Filter_select_note_normal Nothing) =
  procFilter (Filter Filter_select_note_normal (Just Filter_mode_yes))
procFilter (Filter Filter_select_note_normal (Just Filter_mode_yes)) =
  mapMusicXML filterNormalNote
```

```
procFilter (Filter Filter_select_note_normal (Just Filter_mode_no)) =
  mapMusicXML filterNotNormalNote
procFilter (Filter Filter_select_note_cue Nothing) =
  procFilter (Filter Filter_select_note_cue (Just Filter_mode_yes))
procFilter (Filter Filter_select_note_cue (Just Filter_mode_yes)) =
  mapMusicXML filterCueNote
procFilter (Filter Filter_select_note_cue (Just Filter_mode_no)) =
  mapMusicXML filterNotCueNote
procFilter (Filter Filter_select_note_grace Nothing) =
  procFilter (Filter Filter_select_note_grace (Just Filter_mode_yes))
procFilter (Filter Filter_select_note_grace (Just Filter_mode_yes)) =
  mapMusicXML filterGraceNote
procFilter (Filter Filter_select_note_grace (Just Filter_mode_no)) =
  mapMusicXML filterNotGraceNote
  -- |
onlyPartwise :: (MusicXML.Score_Partwise → MusicXML.Score_Partwise) →
  MusicXML.MusicXMLDoc → MusicXML.MusicXMLDoc
onlyPartwise f (MusicXML.Score (MusicXML.Partwise x)) =
  MusicXML.Score (MusicXML.Partwise (f x))
onlyPartwise _ x = x
  -- |
procReification :: Reification →
  MusicXML.Score_Partwise → MusicXML.Score_Partwise
procReification (Reification Reification_value_5) =
  Interface.rep_Score_Partwise ·
  Interface.abst_Score_Partwise
procReification (Reification Reification_value_4) =
  Interface.rep_Score_Partwise ·
  Layer5.rep_Score_Partwise ·
  Layer5.abst_Score_Partwise ·
  Interface.abst_Score_Partwise
procReification (Reification Reification_value_3) =
  Interface.rep_Score_Partwise ·
  Layer5.rep_Score_Partwise ·
  Layer4.rep_Score_Partwise ·
  Layer4.abst_Score_Partwise ·
  Layer5.abst_Score_Partwise ·
  Interface.abst_Score_Partwise
procReification (Reification Reification_value_2) =
  Interface.rep_Score_Partwise ·
  Layer5.rep_Score_Partwise ·
  Layer4.rep_Score_Partwise ·
  -- Layer3.rep_Score_Partwise .
  -- Layer3.abst_Score_Partwise .
  Layer4.abst_Score_Partwise ·
  Layer5.abst_Score_Partwise ·
  Interface.abst_Score_Partwise
procReification (Reification Reification_value_1) =
  Interface.rep_Score_Partwise ·
  Layer5.rep_Score_Partwise ·
  Layer4.rep_Score_Partwise ·
  -- Layer3.rep_Score_Partwise .
  -- Layer2.rep_Score_Partwise .
```

```
  -- Layer2.abst_Score_Partwise .
  -- Layer3.abst_Score_Partwise .
  Layer4.abst_Score_Partwise ·
  Layer5.abst_Score_Partwise ·
  Interface.abst_Score_Partwise
procCount :: Count → MusicXML.MusicXMLDoc → (String, Int)
procCount (Count Count_select_part)          =
  λm → ("part", count_part m)
procCount (Count Count_select_measure)       =
  λm → ("measure", count_measure m)
procCount (Count Count_select_music_data)    =
  λm → ("music-data", count_music_data m)
procCount (Count Count_select_note)          =
  λm → ("note", count_note m)
procCount (Count Count_select_note_normal)   =
  λm → ("note-normal", count_note_normal m)
procCount (Count Count_select_note_grace)    =
  λm → ("note-grace", count_note_grace m)
procCount (Count Count_select_note_cue)      =
  λm → ("note-cue", count_note_cue m)
procStat :: Stat → MusicXML.MusicXMLDoc → String
procStat (Stat (Stat_Attrs Nothing) list) =
  procStat (Stat (Stat_Attrs (Just Stat_verbose_yes)) list)
procStat (Stat (Stat_Attrs (Just Stat_verbose_no)) list) =
  λm → unlines (map (show · π2) (map (flip procCount m) list))
procStat (Stat (Stat_Attrs (Just Stat_verbose_yes)) list) =
  λm → unlines (map (uncurry (++) · ((++": ") × show))
    (map (flip procCount m) list))
  -- |
procParttime :: a → MusicXML.MusicXMLDoc → MusicXML.MusicXMLDoc
procParttime _ (MusicXML.Score (MusicXML.Partwise music)) =
  (MusicXML.Score (MusicXML.Timewise (Interface.toTimewise music)))
procParttime _ music = music
  -- |
procTimepart :: a → MusicXML.MusicXMLDoc → MusicXML.MusicXMLDoc
procTimepart _ (MusicXML.Score (MusicXML.Timewise music)) =
  (MusicXML.Score (MusicXML.Partwise (Interface.toPartwise music)))
procTimepart _ music = music
procHaskore :: a → MusicXML.MusicXMLDoc → String
procHaskore _ (MusicXML.Score (MusicXML.Partwise music)) = "music = " ++
  show (IHaskore.partwise2haskore music)
procHaskore _ (MusicXML.Score (MusicXML.Timewise music)) = "music = " ++
  show (IHaskore.partwise2haskore (Interface.toPartwise music))
procHaskore _ _ = []
procMidi :: Midi → MusicXML.MusicXMLDoc → Haskore.Music.GeneralMIDI.T + ()
procMidi (Midi Nothing) = procMidi (Midi (Just Midi_play_no))
procMidi (Midi (Just Midi_play_no)) = i2 · ()‾
procMidi (Midi (Just Midi_play_yes)) = i1 · toMidi · procTimepart Timepart
  -- (procTimepart Timepart m)
  -- playMidi 'test.mid' (procTimepart Timepart music) ■ return ()
  -- procMidi (Midi (Just filepath) Nothing) music =
  -- procMidi (Midi (Just filepath) (Just Midi_play_no)) music
```

```
  -- procMidi (Midi (Just filepath) (Just Midi_play_no)) music =
  -- saveMidi filepath music
  -- procMidi (Midi (Just filepath) (Just Midi_play_yes)) music =
  -- saveMidi filepath music ■
  -- playMidi filepath (procTimepart Timepart music) ■ return ()


  -- |
procWarn :: Maybe Action_warnings → Bool
procWarn Nothing = procWarn (Just Action_warnings_no)
procWarn (Just Action_warnings_no) = False
procWarn (Just Action_warnings_yes) = True
  -- |
procOutput :: Maybe FilePath →
  (MusicXML.MusicXMLDoc + String) + ·
  (Haskore.Music.GeneralMIDI.T + ()) → IO ()
procOutput Nothing x = do
  let f = MusicXML.show_CONTENTS MusicXML.show_MusicXMLDoc
  [[putStrLn · f, putStrLn],
    [return · (), return · ()]] x
  hFlush stdout
procOutput (Just output) x = do
  let f = MusicXML.show_FILE MusicXML.show_MusicXMLDoc
  [[f output, writeFile output],
    [λm → saveMidi output m ≫
      playMidi output m ≫ return (),
      return · ()]] x
  -- |
fromError :: MusicXML.Result a → String
fromError (MusicXML.Error e) = e
fromError (MusicXML.Ok _) = throw (ErrorCall "internal error")
  -- |
procAction_ ::
  [OneOf7 Filter Reification Stat Parttime Timepart Haskore Midi] →
  MusicXML.MusicXMLDoc →
  (MusicXML.MusicXMLDoc + String) + ·
  (Haskore.Music.GeneralMIDI.T + ())
procAction_ [] = i1 · i1
  -- procAction_ ((OneOf8 a):_) = i1 . i2 . procLength a
procAction_ ((OneOf7 b) : t) = λm → procAction_ t (procFilter b m)
procAction_ ((TwoOf7 c) : t) =
  λm → procAction_ t (onlyPartwise (procReification c) m)
procAction_ ((ThreeOf7 d) : _) = i1 · i2 · procStat d
procAction_ ((FourOf7 e) : t) = λm → procAction_ t (procParttime e m)
procAction_ ((FiveOf7 f) : t) = λm → procAction_ t (procTimepart f m)
procAction_ ((SixOf7 g) : _) = i1 · i2 · procHaskore g
procAction_ ((SevenOf7 h) : _) = i2 · procMidi h
procAction :: Action → IO ()
procAction (Action (Action_Attrs input output w) x) = do
  musicxml ← MusicXML.read_FILE MusicXML.read_MusicXMLDoc input
  case MusicXML.isOK musicxml of
    True → procOutput output
      (procAction_ x (MusicXML.fromOK musicxml))
    False →
```

```
        if procWarn w
        then procOutput output ((i1 · i2) (fromError musicxml))
        else return ()
  -- |
procScript :: Script → IO ()
procScript (Script _ actions) = mapM_ procAction actions
```

7.2 HaMusic

```
module Main where
import Music.Analysis ()
import Music.Analysis.Script
import Control.Monad
import Data.List
import System.IO
import System.Directory
import System.Environment
import System.Console.GetOpt
import Text.XML.HaXml.XmlContent


  -- |
filepath_default :: FilePath
filepath_default = "HaMusic.xml"
  -- |
directory_default :: FilePath
directory_default = "."
  -- |
warn :: String → IO ()
warn msg = putStrLn msg ≫ hFlush stdout
  -- |
version :: String
version = "0.1.1027"


  -- |
data Option = Help | Version | Input FilePath | Path FilePath | Check
  deriving (Eq, Show)
  -- |
options :: [OptDescr Option]
options = [
  Option ['v', 'V'] ["version"] (NoArg Version) "show version number"
  , Option ['h', 'H', '?'] ["help"] (NoArg Help) "show help"
  , Option ['c'] ["check"] (NoArg Check) "check script"
  , Option ['s', 'S'] ["script"] opt1 "input file script"
  , Option ['p', 'P'] ["path"] opt2 "change current path"
  ]
  where opt1 = OptArg (Input · maybe filepath_default id) "FILE"
    opt2 = OptArg (Path · maybe directory_default id) "PATH"
  -- |
header :: String → String
header prog = "Usage: " ++ prog ++ " [OPTIONS...] FILES..."
```

```
  -- |
procScriptFile :: Bool → FilePath → IO ()
procScriptFile boolean filepath = do
  curdir ← getCurrentDirectory
  script ← fReadXml filepath
  -- maybe (return ())
  (λx → if null x then return () else setCurrentDirectory x)
    ((concat · init · groupBy (λ_ y → y ≢ '/')) filepath)
  unless boolean (procScript script)
  setCurrentDirectory curdir
  -- script <- (fReadXml filepath :: IO Script)
  -- maybe (return ())
  -- (\x -> if null x then return () else setCurrentDirectory x)
  -- ((concat.init.groupBy (\ _ y -> y/='/')) filepath)


  -- |
parseOptions :: Option → Maybe (FilePath + FilePath)
parseOptions (Input filepath) = Just (i1 filepath)
parseOptions (Path path) = Just (i2 path)
parseOptions _ = Nothing


  -- |
main :: IO ()
main = do
  argv ← getArgs
  prog ← getProgName
  curdir ← getCurrentDirectory
  case getOpt Permute options argv of
    (o, n, []) | Help ∈ o → putStrLn (usageInfo (header prog) options)
       | Version ∈ o → putStrLn (unwords [prog, version])
       | Check ∈ o →
         mapM_ (maybe (return ())
           ([procScriptFile True, setCurrentDirectory]) ·
           parseOptions) (o ++ map Input n)
          ≫ setCurrentDirectory curdir
       | otherwise →
         mapM_ (maybe (return ())
           ([procScriptFile False, setCurrentDirectory]) ·
           parseOptions) (o ++ map Input n)
          ≫ setCurrentDirectory curdir
    (_, _, errs) → putStrLn (unlines errs ++ usageInfo (header prog) options)
```

7.3 Translator

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- This module implements
module Main where
```

```
import qualified Text.XML.MusicXML as MusicXML
import qualified Music.Analysis.MusicXML as IMusicXML
import qualified Music.Analysis.MusicXML.Level3 as Level3
import qualified Music.Analysis.MusicXML.Level4 as Level4
import qualified Music.Analysis.MusicXML.Level5 as Level5
import qualified Music.Analysis.MusicXML2Haskore as MusicXML2Haskore
import qualified Music.Analysis.MusicXML2Abstract as MusicXML2Abstract
import qualified Music.Analysis.MusicXML2ABC as MusicXML2ABC
import qualified Music.Analysis.Abstract2ABC as Abstract2ABC
import Control.Monad
import Data.List
import System.IO
import System.Environment
import System.Console.GetOpt
import System.FilePath
import Control.Exception (throw, Exception (..))


  -- |
warn :: String → IO ()
warn msg = putStrLn msg ≫ hFlush stdout
  -- |
version :: String
version = "0.1.1028"


  -- |
data Option = InputFormat Format
   | OutputFormat Format
   | Help
   | Version
     deriving (Eq, Show)
data Format = MusicXML | Abstract | ABC | Lilypond | Haskore | Unknown
  deriving (Eq, Show)
  -- |
isInput :: Option → Bool
isInput (InputFormat _) = True
isInput _               = False
isOutput :: Option → Bool
isOutput (OutputFormat _) = True
isOutput _ = False
  -- |
importFormat :: String → Format
importFormat "musicxml" = MusicXML
importFormat "abstract" = Abstract
importFormat "abc"      = ABC
importFormat "lilypond" = Lilypond
importFormat "haskore" = Haskore
importFormat _          = Unknown
  -- |
options :: [OptDescr Option]
options = [
  Option ['v', 'V'] ["version"] (NoArg Version) "show version number"
  , Option ['h', 'H', '?'] ["help"] (NoArg Help) "show help"
  , Option ['i'] ["input"] (ReqArg (InputFormat · importFormat) "FORMAT")
```

```
    "input format"
  , Option ['o'] ["output"] (ReqArg (OutputFormat . importFormat) "FORMAT")
    "output format"
  ]
  -- |
header :: String -> String
header prog = "Usage: " ++ prog ++ " [OPTIONS...] FILES..."


  -- |
fromError :: MusicXML.Result a -> String
fromError (MusicXML.Error e) = e
fromError (MusicXML.Ok _) = throw (ErrorCall "internal error")
  -- |
forFILES :: [x] -> (x -> IO ()) -> IO ()
forFILES xs f = mapM_ f xs
  -- |
todo :: Format -> Format -> IO ()
todo a b = putStrLn
  ("Can't translate from " ++ show a ++ "to " ++ show b ++ "formats")


convert :: Format -> Format -> [FilePath] -> IO ()
convert MusicXML Abstract xs =
  forFILES xs (\x -> do
    musicxml <- MusicXML.read_FILE MusicXML.read_MusicXML_Partwise x
    case MusicXML.isOK musicxml of
      True -> writeFile (replaceExtension x "abs")
        ((show . MusicXML2Abstract.mk_Music .
          Level3.abst_Score_Partwise .
          Level4.abst_Score_Partwise .
          Level5.abst_Score_Partwise .
          IMusicXML.abst_Score_Partwise .
          MusicXML.fromOK) musicxml)
      False -> putStrLn (fromError musicxml))
convert MusicXML ABC xs =
  forFILES xs (\x -> do
    musicxml <- MusicXML.read_FILE MusicXML.read_MusicXML_Partwise x
    case MusicXML.isOK musicxml of
      True -> writeFile (replaceExtension x "abc")
        ((show . MusicXML2ABC.mk_Music .
          Level3.abst_Score_Partwise .
          Level4.abst_Score_Partwise .
          Level5.abst_Score_Partwise .
          IMusicXML.abst_Score_Partwise .
          MusicXML.fromOK) musicxml)
      False -> putStrLn (fromError musicxml))
convert MusicXML Haskore xs = do
  forFILES xs (\x -> do
    musicxml <- MusicXML.read_FILE MusicXML.read_MusicXML_Partwise x
    case MusicXML.isOK musicxml of
      True -> writeFile (replaceExtension x "hs")
        ((show . MusicXML2Haskore.partwise2haskore .
          MusicXML.fromOK) musicxml)
      False -> putStrLn (fromError musicxml))
```

```
convert MusicXML Lilypond xs = do
  let _ = map (flip replaceExtension "ly") xs
  todo MusicXML Lilypond
convert ABC MusicXML xs = do
  let _ = map (flip replaceExtension "xml") xs
  todo ABC MusicXML
convert ABC Abstract xs = do
  let _ = map (flip replaceExtension "abs") xs
  todo ABC Abstract
convert ABC Haskore xs = do
  let _ = map (flip replaceExtension "hs") xs
  todo ABC Haskore
convert ABC Lilypond xs = do
  let _ = map (flip replaceExtension "ly") xs
  todo ABC Lilypond
convert Abstract MusicXML xs = do
  let _ = map (flip replaceExtension "xml") xs
  todo Abstract MusicXML
convert Abstract ABC xs = do
  forFILES xs (λx → do
    y ← readFile x
    writeFile (replaceExtension x "abc")
      ((show · Abstract2ABC.mk_Music · read) y))
convert Abstract Lilypond xs = do
  let _ = map (flip replaceExtension "ly") xs
  todo Abstract Lilypond
convert Abstract Haskore xs = do
  let _ = map (flip replaceExtension "hs") xs
  todo Abstract Haskore
convert Haskore ABC xs = do
  let _ = map (flip replaceExtension "abc") xs
  todo Haskore ABC
convert Haskore Abstract xs = do
  let _ = map (flip replaceExtension "abs") xs
  todo Haskore Abstract
convert Haskore MusicXML xs = do
  let _ = map (flip replaceExtension "xml") xs
  todo Haskore MusicXML
convert Haskore Lilypond xs = do
  let _ = map (flip replaceExtension "ly") xs
  todo Haskore Lilypond
convert Lilypond MusicXML xs = do
  let _ = map (flip replaceExtension "ly") xs
  todo Lilypond MusicXML
convert Lilypond ABC xs = do
  let _ = map (flip replaceExtension "abc") xs
  todo Lilypond ABC
convert Lilypond Abstract xs = do
  let _ = map (flip replaceExtension "abs") xs
  todo Lilypond Abstract
convert Lilypond Haskore xs = do
  let _ = map (flip replaceExtension "hs") xs
  todo Lilypond Haskore
convert a b _ | a ≡ b = todo a b
```

```
    | a ≡ Unknown = todo a b
    | b ≡ Unknown = todo a b
    | otherwise = todo a b


main :: IO ()
main = do
  argv ← getArgs
  prog ← getProgName
  case getOpt Permute options argv of
    ([InputFormat a, OutputFormat b], n, []) → convert a b n
    ([OutputFormat a, InputFormat b], n, []) → convert b a n
    (o, n, []) | Help ∈ o → putStrLn (usageInfo (header prog) options)
       | Version ∈ o → putStrLn (unwords [prog, version])
       | otherwise →
         putStrLn (unlines n ++ usageInfo (header prog) options)
    (_, _, errs) → putStrLn (unlines errs ++ usageInfo (header prog) options)
```

7.4 MusicCount

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: portable
  -- This module implements
module Main where
import qualified Text.XML.MusicXML as MusicXML
  -- import Music.Analysis.MusicXML2Haskore
import Music.Analysis.MusicXML.Functions
import Control.Monad
import Data.List
import System.IO
import System.Environment
import System.Console.GetOpt
  -- import System.FilePath
  -- import Control.Exception (throw, Exception(..))


  -- |
warn :: String → IO ()
warn msg = putStrLn msg ≫ hFlush stdout
  -- |
version :: String
version = "0.1.1027"


  -- |
data Option = Part | Measure | Note | Sharp | Flat | GraceNote
  -- | MusicXML | Abstract | ABC | Lilypond | Haskore
  -- | Format
  -- | MusicXML2Abstract
  -- | Abstract2ABC
```

```
  -- | Abstract2Lilypond
  -- | MusicXML2Haskore
   | Help
   | Version
     deriving (Eq, Show)
data Options = Options{ part :: Bool, measure :: Bool, note :: Bool }
data ResultOptions = Result (Maybe Int) (Maybe Int) (Maybe Int)
  deriving (Show)
  -- |
options :: [OptDescr Option]
options = [
  Option ['v', 'V'] ["version"] (NoArg Version) "show version number"
  , Option ['h', 'H', '?'] ["help"] (NoArg Help) "show help"
  , Option ['p'] ["part"] (NoArg Part) "counts part"
  , Option ['m'] ["measure"] (NoArg Measure) "counts measure"
  , Option ['n'] ["note"] (NoArg Note) "counts note"
  -- , Option ['i'] ['input'] (ReqArg Input 'FILE') 'input music file'
  -- , Option ['o'] ['output'] (ReqArg Output 'FILE') 'output music file'
  -- , Option ['a'] ['abstract'] (NoArg Abstract) 'abstract format'
  -- , Option ['M'] ['musicxml'] (NoArg MusicXML) 'musicxml format'
  -- , Option ['H'] ['haskore'] (NoArg Haskore) 'haskore format'
  -- , Option ['A'] ['abc'] (NoArg ABC) 'abc format'
  -- , Option ['L'] ['lilypond'] (NoArg Lilypond) 'lilypond format'
  -- , Option ['c'] ['convert'] opt1
  -- 'convert input format into output format'
  -- , Option [] ['musicxml2abstract'] (NoArg MusicXML2Abstract)
  -- 'converts musicxml file into abstract music'
  -- , Option [] ['musicxml2haskore'] (NoArg MusicXML2Haskore)
  -- 'converts musicxml file into haskore music'
  -- , Option [] ['abstract2abc'] (NoArg Abstract2ABC)
  -- 'converts abstract music into abc notation'
  -- , Option [] ['abstract2lilypond'] (NoArg Abstract2Lilypond)
  -- 'converts abstract music into lilypond notation'
  ]
  -- |
header :: String → String
header prog = "Usage: " ++ prog ++ " [OPTIONS...] FILES..."


  -- |
counts :: Options → MusicXML.MusicXMLDoc → ResultOptions
counts opt m = Result (if part opt then Just (count_part m) else Nothing)
  (if measure opt then Just (count_measure m) else Nothing)
  (if note opt then Just (count_note m) else Nothing)
  -- Nothing Nothing
importMusicXML :: FilePath → IO (Maybe MusicXML.MusicXMLDoc)
importMusicXML input = do
  musicxml ← MusicXML.read_FILE MusicXML.read_MusicXMLDoc input
  case MusicXML.isOK musicxml of
    True → return (Just (MusicXML.fromOK musicxml))
    False → putStrLn ("Can't read " ++ show input) ≫ return Nothing
defaultOptions :: Options
defaultOptions = Options{ part = False, measure = False, note = False }
```

```
convert :: [Option] → Options → Options
convert [] = id
convert (Part : xs) = λopts → convert xs opts{ part = True }
convert (Measure : xs) = λopts → convert xs opts{ measure = True }
convert (Note : xs) = λopts → convert xs opts{ note = True }
convert (_ : xs) = convert xs


main :: IO ()
main = do
  argv ← getArgs
  prog ← getProgName
  case getOpt Permute options argv of
    (o, n, []) | Help ∈ o → putStrLn (usageInfo (header prog) options)
       | Version ∈ o → putStrLn (unwords [prog, version])
       | otherwise → mapM_ (λinput → do
      m ← importMusicXML input
      maybe (return ()) (print · counts (convert o defaultOptions)) m) n
    (_, _, errs) → putStrLn (unlines errs ++ usageInfo (header prog) options)
```

8 Tests

8.1 Main

```
import qualified Test.Recordare
import qualified Test.Wikifonia
import Prelude


main :: IO ()
main = putStrLn "=> Test program: Recordare" ≫ Test.Recordare.main
  ≫ putStrLn "=> Test program: Wikifonia" ≫ Test.Wikifonia.main
```

8.2 Recordare

```
import qualified Test.Recordare
main :: IO ()
main = Test.Recordare.main
```

8.3 Test.Recordare

```
module Test.Recordare where
import Music.Analysis   -- HaMusic package
import Text.XML.MusicXML hiding (String)
```

```
import Music.Analysis.MusicXML as Interface
import Music.Analysis.MusicXML.Level5 as Layer5
import Music.Analysis.MusicXML.Level4 as Layer4
  -- import Music.Analysis.Definition.Layer3 as Layer3
  -- import Music.Analysis.Definition.Layer2 as Layer2
import System.IO
import Prelude


  -- |
pathto :: FilePath
pathto = "../examples/Recordare/"
  -- |
files_Recordare :: [FilePath]
files_Recordare = [
  "Echigo-Jishi",
  "elite",
  "ActorPreludeSample",
  "BeetAnGeSample",
  "Binchois",
  "BrahWiMeSample",
  "BrookeWestSample",
  "Chant",
  "DebuMandSample",
  "Dichterliebe01",
  "FaurReveSample",
  "MahlFaGe4Sample",
  "MozaChloSample",
  "MozaVeilSample",
  "MozartPianoSonata",
  "MozartTrio",
  "Saltarello",
  "SchbAvMaSample",
  "Telemann"]
  -- |
input_partwise, input_timewise :: [FilePath]
input_partwise = map (λx → pathto ++ "partwise/" ++ x ++ ".xml") files_Recordare
input_timewise = map (λx → pathto ++ "timewise/" ++ x ++ ".xml") files_Recordare
mkoutput :: FilePath → FilePath
mkoutput = reverse · ("lmx.tuptuo-"++) · drop 4 · reverse


  -- |
putLn :: String → IO ()
putLn msg = putStrLn msg ≫ hFlush stdout
put :: String → IO ()
put msg = putStr msg ≫ hFlush stdout


notPartwise :: MusicXMLDoc → Bool
notPartwise (Score (Partwise _)) = False
notPartwise _ = True
inout :: FilePath → IO ()
inout filepath = do
  putLn ("filepath: " ++ filepath)
```

```
  x ← read_FILE read_MusicXMLDoc filepath
  case isOK x of
    True → do
  -- put '\tPart[simple]: ' ■ (return.fromOK) x ■= print . length_Part
  -- put '\tPart[map]: ' ■ (return.fromOK) x ■= print . length_Part_map
  -- put '\tPart[concat]: ' ■ (return.fromOK) x ■= print . length_Part_concat
  -- put '\tMeasure[simple]: ' ■ (return.fromOK) x ■= print . length_Measure
  -- put '\tMeasure[map]: ' ■ (return.fromOK) x ■= print . length_Measure_map
  -- put '\tMeasure[concat]: ' ■ (return.fromOK) x ■= print . length_Measure_concat
  if (notPartwise · fromOK) x then return () else do
    let todown = (Layer4.abst_Score_Partwise ·
      Layer5.abst_Score_Partwise ·
      Interface.abst_Score_Partwise ·
      (λ(Score (Partwise x')) → x') · fromOK) x
    let toup = (Score · Partwise · Interface.rep_Score_Partwise ·
      Layer5.rep_Score_Partwise ·
      Layer4.rep_Score_Partwise) todown
    show_FILE show_MusicXMLDoc (mkoutput filepath) (toup)
  -- put '\tPart[simple]: ' ■ return toup ■= print . length_Part
  -- put '\tPart[map]: ' ■ return toup ■= print . length_Part_map
  -- put '\tPart[concat]: ' ■ return toup ■= print . length_Part_concat
  -- put '\tMeasure[simple]: ' ■ return toup ■= print . length_Measure
  -- put '\tMeasure[map]: ' ■ return toup ■= print . length_Measure_map
  -- put '\tMeasure[concat]: ' ■ return toup ■= print . length_Measure_concat
    False → return ()


  -- |
main :: IO ()
main = putLn "=> Test number: 1 \t<partwise>"
  ≫ mapM_ inout input_partwise
  ≫ putLn "=> Test number: 2 \t<timewise>"
  ≫ mapM_ inout input_timewise
```

9 Wikifonia

```
import qualified Test.Wikifonia
main :: IO ()
main = Test.Wikifonia.main
```

10 Test/Wikifonia

```
module Test.Wikifonia where
import Music.Analysis ()   -- HaMusic package
import System.IO
import Prelude
```

```
  -- |
pathto :: FilePath
pathto = "../examples/Wikifonia/"
  -- |
files_Wikifonia :: [FilePath]
files_Wikifonia = []
  -- |
input :: [FilePath]
input = map (λx → pathto ++ x ++ ".xml") files_Wikifonia
mkoutput :: FilePath → FilePath
mkoutput = reverse · ("lmx.tuptuo-"++) · drop 4 · reverse


  -- |
put :: String → IO ()
put msg = putStrLn msg ≫ hFlush stdout


  -- |
main :: IO ()
main = return ()
```