Music as Language

Putting Probabilistic Temporal Graph Grammars to Good Use

Orestis Melkonian
Information and Computing Sciences
Utrecht University
The Netherlands
melkon.or@gmail.com

Abstract

Music composers have long been attracted by the idea of an automated tool for music generation, that is able to aid them in their day-to-day compositional process. We focus on algorithmic composition techniques that do not aim to produce complete music pieces, but rather provide an expert composer with a source of copious amounts of musical ideas to explore.

A promising formalism towards this direction are *probabilistic temporal graph grammars* (PTGGs), which allow for the automatic generation of musical structures by defining a set of expressive rewrite rules.

However, the primary focus so far has been on generating harmonic structures, setting aside the other two main pillars of music: melody and rhythm. We utilize the expressiveness of PTGGs to transcribe grammars found in the musicology literature. In order to do so, we make slight modifications to the original PTGG formalism and provide a concise domain-specific language (DSL) embedded in Haskell to define such grammars. Furthermore, we employ a heuristics-driven post-processing step that interprets the abstract musical structures produced by our grammars into concrete musical output.

Lastly, parametrizing over different musical configurations enables more user control over the generative process. We produce multiple variations of four configurations to demonstrate the flexibility of our framework and motivate the use of *formal grammars* in automated music composition.

CCS Concepts • **Theory of computation → Grammars and context-free languages**; • **Applied computing → Sound and music computing**; • **Software and its engineering** → *Domain specific languages.*

Keywords algorithmic music composition, music grammars

FARM '19, August 23, 2019, Berlin, Germany

ACM Reference Format:
Orestis Melkonian. 2019. Music as Language: Putting Probabilistic Temporal Graph Grammars to Good Use. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design (FARM '19), August 23, 2019, Berlin, Germany*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3331543.3342576

1 Introduction

1.1 Probabilistic Temporal Graph Grammars

Our work relies on a class of generative grammars for harmonic structure, called probabilistic temporal graph grammars (PTGGs) [Quick and Hudak 2013]. These grammars consist of weighted rules (*probabilistic*), which are moreover parametrized by time duration (*temporal*) and allow repetition of a rewritten symbol via the use of the *Let* construct (*graph*).

We eschew from giving a formal definition of PTGGs here, as we will actually make some non-trivial modifications to the original formulation and present our variant thoroughly in Section 2.

1.2 Euterpea

Throughout our development, we use the Haskell music library *Euterpea* [Hudak and Quick 2018], as it offers definitions for common musical structures and the ability to render music to MIDI files.

We further define the type of musical intervals, giving way to chords and scales:

```
data Interval
  = P1  | Mi2 | M2  | Mi3 | M3 | P4  | A4  | P5
  | Mi6 | M6  | Mi7 | M7  | P8 | Mi9 | ... | P15
  deriving (Eq, Enum)

type ChordType = [Interval]    -- ≡ ScaleType
type SemiChord = [PitchClass]  -- ≡ SemiScale
type Chord     = [Pitch]       -- ≡ Scale
(⊩) :: PitchClass → ChordType → SemiChord
(⊩) = ...
```

A *ChordType* is defined by the intervals that comprise it, a *SemiChord* is a *ChordType* with a specific tonic pitch class and a *Chord* is a concrete voicing of a *SemiChord*. Scales

have exactly the same representation as chords, but we give different type aliases to distinguish them semantically. We will use the $\Vdash$ operator to instantiate a chord/scale type with a tonic.

Another useful operation we will frequently use is that of *transposition* (upwards or downwards). As this notion applies to several musical elements, we define a typeclass to overload the corresponding operators:

```
class Transposable a where
  (↑), (↓) :: a → Interval → a

instance Transposable PitchClass where ...
instance Transposable Chord where ...
```

For convenience, we define short-hands for widely used types of chords and scales:

```
-- Chord types.
maj   = [P1, M3, P5]
m7b5 = [P1, Mi3, A4, Mi7]
⋮
allChords = [maj, ...] :: [ChordType]

-- Scale types.
ionian = [P1, M2, M3, P4, P5, M6, M7]
major  = ionian
lydian = mode 4 ionian
⋮
allScales = [ionian, ...] :: [ScaleType]
```

Last but not least, we will need the action of randomly selecting an item from a weighted list:

```
equally :: [a] → [(Double, a)]
equally = zip (repeat 1.0)

choose :: MonadRandom m ⇒ [(Double, a)] → m a
choose xs = do
  i ← getIndex ⟨$⟩ getRandomR (0, sum (fst ⟨$⟩ xs))
  return (xs !! i)

chooseBy :: MonadRandom m
         ⇒ (a → Double) → [(Double, a)] → m a
chooseBy = choose ∘ fmap (λa → (f a, a))
```

Notice the use of *MonadRandom*, which allows the operators to be used in any monadic stack that provides a randomness source[1], as well as reproducible results via randomness *seeds*. For brevity, we will replace these qualified signatures with their instantiation in the *IO* monad[2].

Overview The rest of the paper is structured as follows: Section 2 describes our variant of the original PTGG formulation, which makes it more convenient to transcribe grammars found "in the wild". Sections 3, 4 and 5 give grammars for tonal harmony, melodic improvisation, and tabla improvisation, respectively. Section 6 presents several example configurations we use to generate music pieces and Section 7 concludes with an overview of possible next steps.

2 Extending PTGG

In this section, we describe several extensions and alterations we make to the original PTGG formalism introduced in [Quick and Hudak 2013]. For the sake of self-containment, we provide all relevant definitions, but choose to omit technicalities pertaining to music manipulation tasks. Nonetheless, readers who are not familiar with this particular grammar formalism are encouraged to read the original paper first.
All of our code development is available on Github[3].

2.1 Basic Definitions

We start out with the datatype of grammars, which is parametrized over the type of metadata it carries and the type of symbols it manipulates. A grammar then consists of an initial symbol and a list of rules:

```
data Grammar meta a
  = a |: [Rule meta a]
```

Rules rewrite atomic symbols to (possibly more complex) grammar terms, which can depend on the time duration of the current symbol. We assign probabilistic weights to each rule; when multiple rules can fire simultaneously, we randomly select one based on their weight:

Firing Guards Modelling the right-hand side of rules as a function from time values is certainly flexible and compositional, but may prove overly unconstrained to allow rewriting in any possible fraction of time. For instance, it is perfectly reasonable to disallow rewriting of terms that have a short duration (e.g. restrict the number of chords in a bar).

To express these constraints, we have augmented the left-hand side of the grammar rules with a predicate on time values; the corresponding rule fires only whenever this predicate holds for the current time. Below is the rule definition that combines all aforementioned elements:

```
data Rule meta a
  = (a, Double, Dur → Bool) ↣ (Dur → Term meta a)
```

[1] https://hackage.haskell.org/package/MonadRandom/docs/Control-Monad-Random-Class.html

[2] We write $(\ldots \to IO\ a)$ instead of $(MonadRandom\ m \Rightarrow \ldots \to m\ a)$.

[3] https://github.com/omelkonian/music-grammars

Terms are sequences of user-supplied symbols of a certain duration; duration is assigned using the : operator and sequencing is possible with the binary operator ⊗. We can also wrap existing terms with auxiliary metadata using ▷ and repeat terms using *Let*:

```
data Term meta a
  = a : Dur
  | Term meta a ⊗ Term meta a
  | meta ▷ Term meta a
  | Let (Term meta a) (Term meta a → Term meta a)
```

Higher-order Abstract Syntax for Let The grammars discussed in this work do not require repetition in their rules, but we nevertheless provide a higher-order variant of the original *Let* construct that enables node sharing.

Instead of using variables to refer to let-bound terms, we utilize *higher-order abstract syntax* [Pfenning and Elliott 1988], as evidenced by the second argument of the *Let* constructor, which is another function. During the final step of the rewriting process, when the fixpoint is reached, we "unlet" the resulting term by applying the supplied function to the bound term. This higher-order variant gives a stricter notion of repetition than that proposed in [Quick and Hudak 2013]; it is not possible to continue rewriting inside the continuation, so multiple uses of the variable will result in exact copies that cannot diverge.

Auxiliary Metadata The original PTGG formalism had introduced a separate term constructor for harmonic modulation. Since we are dealing with more general grammars here, we replace this constructor with a generic one that annotates other terms with auxiliary information and make the *Term* datatype polymorphic on the type of annotations. In Section 3, where we implement a grammar for tonal harmony, the type variable of annotations will be instantiated to the type of musical intervals, hence modelling key modulation.

2.2 Typeclasses

The type of symbols *a* is provided by the user, but has to adhere to certain constraints in order to allow for rewriting and translation to actual music. To enforce these constraints, we use Haskell's typeclasses [Wadler and Blott 1989] and require the type of symbols to be an instance of several of them[4].

Expansion Any metadata-carrying grammar term must be expanded to a stripped-down grammar term with no metadata (i.e. *Term a* ()), possibly producing terms of a different type *b* and requiring some *input*:

```
class Expand input a meta b where
  expand :: input → Term meta a → IO (Term () b)
```

Notice that expansion can be a probabilistic process, so we allow execution in the *IO* monad.

We also give a default instance for terms that already carry no metadata:

```
instance Expand input a () a where
  expand = const return
```

Interpretation When the rewriting process is done, we have to somehow convert grammar symbols to actual MIDI events. For that, we use Euterpea's *ToMusic1* typeclass and require that we can convert symbols to that type:

```
class ToMusic1 c ⇒ Interpret input b c where
  interpret :: input → Music b → IO (Music c)
```

Similar to expansion, this can be a random process requiring some configuration as *input*.

If our symbol type is already an instance of *ToMusic1*, we can trivially interpret it:

```
instance ToMusic1 b ⇒ Interpret () b Note1 where
  interpret _ = return ∘ toMusic1
```

Lastly, we require that we can compare symbols for equality and package all class constraints in a single type synonym[5]:

```
type Grammarly input meta a b c =
  (Eq a, Eq meta, ToMusic1 c
  , Expand input a meta b
  , Interpret input b c)
```

2.3 Rewriting

The rewriting process is rather straightforward, with few deviations from the original algorithm. Given a desired total duration, a well-formed grammar and the required input for expansion and interpretation, the initial symbol gets rewritten up to fixpoint. Then, we expand all metadata wrappers in the result, which we then interpret as a core music type:

```
runGrammar :: Grammarly input meta a b c
           ⇒ Grammar meta a → Dur → input
           → IO (Music b, Music1)
runGrammar (init |: rs) t0 input = do
  rewritten  ← fixpointM rewrite (init : t0)
  expanded   ← expand input (unlet rewritten)
  let abstract = toMusic expanded
  concrete   ← toMusic1 ⟨$⟩ interpret input abstract
```

[4] We also make use of several typeclass-related GHC extensions, including *MultiParamTypeClasses*, *TypeSynonymInstances*, *FlexibleInstances* and *FunctionalDependencies*.

[5] The *ConstraintKinds* extension is needed to capture constraints in type synonyms.

```
  return (abstract, concrete)
  where
    rewrite :: Term meta a → IO (Term meta a)
    rewrite (a : t) = pickRule a t (filter isActive rs)
      where isActive ((a′, _, p) ↣ _) = a′ ≡ a ∧ p t
            pickRule = ...
    toMusic :: Term () b → Music b
    toMusic (a : t)  = Prim (Note t a)
    ...
    unlet :: Term meta a → Term meta a
    unlet (Let x k) = unlet (k x)
    ...
    fixpointM :: Eq x ⇒ (x → IO x) → x → IO x
    ...
```

A single step of the initial rewriting traverses the whole term structure and rewrites atomic symbols, randomly picking one of the active rules for the current symbol and duration. As we mentioned previously, the traversal does not continue in the body of a *Let* and we "unlet" only when the fixpoint is reached. Notice that we also return the abstract musical structure we construct prior to interpretation.

Fixpoint Iteration Our variant of the PTGG rewriting process iterates until a fixpoint is reached. This restricts the grammars we can implement; we will not be able to extract infinite sequences of abstract musical structure by writing a non-converging grammar. However, this variant is more aligned with the type of rewriting systems that grammar writers often have in mind and covers all the grammars we will investigate in this work.

The following utility functions will come in handy when we later define grammars:

```
always :: Duration → Bool
always = const True

fillBars :: (Dur, Dur) → a → Term meta a
fillBars (t, t′) = fold1 ⊗ ∘ replicate (t / t′) ∘ (: t′)
```

The firing guard *always* does not pose any restriction on the current time value, while *fillBars* is used to repeat a symbol of duration t' across the total duration t.

3 Harmony

We initially set out to transcribe a generative grammar for jazz chord sequences [Steedman 1984], which exploits the recursive character of 12-bar blues progressions. Alas, the rules used in Steedman's grammar resemble those of a far more expressive formalism than what we have currently at hand, namely that of *context-sensitive* grammars.

Fortunately, there exists a suitable context-free grammar, quite well-known in computational musicology circles, that aspires to provide a universal context-free grammar for diatonic progressions [Rohrmeier 2011]. The grammar is based on a Schenkerian view of harmony, where an intricate progression can be abstracted away as a single scale degree and, conversely, it is possible to elaborate degrees using a fixed set of musical transformations [Schenker and Jonas 1935].

In fact, such a grammar is more suited to our requirements, since we prefer to have genre-agnostic grammars and let the user further constrain generation according to ad-hoc requirements. We will see example of such constraints in Section 3.2.

3.1 Grammar

The symbols manipulated by our grammar of tonal harmony consist of scale degrees as terminals and non-terminal symbols that guide the rewriting process:

```
data Degree
  = I | II  | III | IV | V | VI | VII -- terminals
  | P | TR | DR | SR | T | D  | S     -- non-terminals
  deriving (Eq, Enum)
```

Grammar symbols are divided in three levels: the phrase level (P) divides the piece in tonic regions, the functional level (TR, DR, SR, T, D, S) distinguishes between tonic/dominant/sub-dominant regions and the scale degree level ($I - VII$) assigns specific scale degrees to each region. Apart from elaborating simple regions to more elaborate ones, the functional level also modulates the key of some grammar sub-terms:

```
harmony :: Grammar Interval Degree
harmony = P |:
  [ -- Phrase level
   (P, 1, always)    ↣ λt → fillBars (t, 4 ∗ wn) TR
    -- Functional level: Expansion
  , (TR, 1, (> wn)) ↣ λt → TR : t/2 ⊗ DR : t/2
  , (TR, 1, always) ↣ λt → DR : t/2 ⊗ T  : t/2
  , (DR, 1, always) ↣ λt → SR : t/2 ⊗ D  : t/2
  ] ++
  [(x, 1, (> wn)) ↣ λt → Let (x : t/2) (λy → y ⊗ y)
    | x ← [TR, SR, DR]
  ] ++
  [(TR, 1, always) ↣ (T :)
  , (DR, 1, always) ↣ (D :)
  , (SR, 1, always) ↣ (S :)
    -- Functional level: Modulation
  , (D, 1, (⩾ qn))  ↣ λt → P5 ▷ D : t
  , (S, 1, (⩾ qn))  ↣ λt → P4 ▷ S : t
    -- Scale-degree level: Secondary dominants
  ] ++
  [(x, 1, (⩾ hn)) ↣ λt → Let  (x : t/2)
                              (λy → (P5 ▷ y) ⊗ y)
```

```
  | x ← [T, D, S]
] ++
[ -- Scale-degree level: Functional-Scale interface
  (T, 1, (≥ wn))  ↣ λt → I : t/2 ⊗ IV : t/4 ⊗ I : t/4
, (T, 1, always)  ↣ (I   :)
, (S, 1, always)  ↣ (IV  :)
, (D, 1, always)  ↣ (V   :)
, (D, 1, always)  ↣ (VI  :) ]
```

The grammar above uses metadata-enriched decorations that carry musical intervals; these model key modulations relative to the current key of the harmonic configuration. Also note the use of (strict) repetitions using the *Let* construct, something not apparent in Rohrmeier's original grammar.

3.2 Expansion

In order to expand the auxiliary modulations our grammar terms carry, we need to have a certain harmonic context:

```
data HarmonyConfig
  = HarmonyConfig
      { basePc    :: PitchClass
      , baseScale :: ScaleType
      , chords    :: [(Double, ChordType)] }
```

We now traverse the grammar emitted from the rewriting process, locally transposing the configuration's key whenever we encounter a modulation and probabilistically choosing a chord type for each scale degree[6]:

```
instance Expand HarmonyConfig Degree Interval
                                  SemiChord where
  expand :: HarmonyConfig → Term Interval Degree
         → IO (Term () SemiChord)
  expand cfg h =
    case h of
      (i ▷ t) → expand (cfg { basePc = basePc cfg ↑ i}) t
      ...
      (a : t) → (: t) ⟨$⟩ choose chs
    where
      tonic = basePc cfg ⊩ baseScale cfg
      tone  = tonic !! fromEnum degree
      chs   = (tone ⊩)
            ⟨$⟩ filter (match tonic) (chords cfg)
```

[6] There are many possible chord types for a given scale degree. For instance, both major triads and major seventh chords can be used for the tonic of a major scale.

3.3 Interpretation

After expanding the results, we get a abstract chord sequence that we need to instantiate with concrete chord voicings. For reasons of euphony, one has to ensure a smooth transition between subsequent chords, a process commonly known as *voice leading*.

```
instance Interpret HarmonyConfig SemiChord
                                 Chord where
  interpret :: HarmonyConfig → Music SemiChord
            → IO (Music Chord)
  interpret cfg = fold1 f
    where f m (sc, d) = do
      ch ← chooseBy (chordDistance m) (inversions sc)
      return (m ⊗ ch)
```

While the original grammar proposed in [Rohrmeier 2011] considers only the standard major and minor scales, we allow degrees to be obtained by any user-supplied scale by simple harmonization[7].

4 Melody

To generate the melodic part of a music piece, we transcribe a probabilistic grammar used for generating jazz solos in the educational software tool Impro-Visor [Keller and Morrison 2007]. The grammar consists of quite a few rules with intricate weights, derived from statistical analysis of large corpora of jazz solos.

Fortunately, it turns out that our current grammar formalism can easily express the original grammar. The grammar proceeds in two levels; first, the rhythmic structure of the improvisation is decided through the non-terminals *P* and *Q*. Second, the rhythmic values are expanded to (possibly more than one) melodic non-terminals *V* and *N*, which are finally rewritten to terminal symbols that characterize a note's function with respect to the current harmonic background.

```
data M
  = HT | CT | L  | AT | ST | R -- terminals
  | P  | Q  | V | N             -- non-terminals
  deriving Eq
```

CT terminals correspond to chord tones (i.e. notes belonging to the current chord in the harmony), *AT* to approach tones (i.e. notes a semitone apart from a chord tone), *L* to tensions tones (i.e. notes not belonging to the current chord, but providing further tension) and *HT* terminals mean either *CT*, *AT* or *L*. *ST* terminals represent scale tones (i.e. notes belonging to the current scale), Finally, *R* terminals correspond to rests.

[7] The most standard technique for scale harmonization is based on stacking thirds on each scale degree.

4.1 Grammar

Most of the rewriting rules of the grammar act on elements of a specific duration, so it is convenient to define a shorthand for duration-independent rules:

```
(↠) :: Head a → Term meta a → Rule meta a
a ↠ b = a ↣ const b
```

Below, we give the grammar of melodic improvisation, acting on the symbols we just defined:

```
melody :: Grammar () M
melody = P |:
  [ -- Rhythmic Structure: Expand P to Q
    (P, 1,  (≡ qn))  ↣ (Q :)
  , (P, 1,  (≡ hn))  ↣ (Q :)
  , (P, 1,  (≡ hn·)) ↠ Q : hn ⊗ Q : qn
  , (P, 25, (> hn·)) ↣ λt → Q : hn ⊗ P : (t − hn)
  , (P, 75, (> wn)) ↣ λt → Q : wn ⊗ P : (t − wn)
    -- Melodic Structure: Expand Q to V, V to N
  , (Q, 52, (≡ wn)) ↠ Q : hn ⊗ V : qn ⊗ V : qn
  , (Q, 47, (≡ wn)) ↠ V : qn ⊗ Q : hn ⊗ V : qn
  , (Q, 1,  (≡ wn)) ↠ V : en ⊗ N : qn ⊗ N : qn
                      ⊗ N : qn ⊗ V : en
  , (Q, 60, (≡ hn)) ↠ Let (V : qn) (λx → x ⊗ x)
  , (Q, 16, (≡ hn)) ↠ HT : qn· ⊗ N : en
  , (Q, 12, (≡ hn)) ↠ V : en ⊗ N : qn ⊗ V : en
  , (Q, 6,  (≡ hn)) ↠ N : hn
  , (Q, 6,  (≡ hn)) ↠ HT : qn³ ⊗ HT : qn³ ⊗ HT : qn³
  , (Q, 1,  (≡ qn)) ↠ CT : qn
  , (V, 1,  (≡ wn)) ↠ Let (V : qn)
                          (λx → x ⊗ x ⊗ x ⊗ x)
  , (V, 72, (≡ qn)) ↠ Let (V : en) (λx → x ⊗ x)
  , (V, 22, (≡ qn)) ↠ N : qn
  , (V, 5,  (≡ qn)) ↠ Let (HT : en³) (λx → x ⊗ x ⊗ x)
  , (V, 1,  (≡ qn)) ↠ Let (HT : en³)
                          (λx → x ⊗ x ⊗ AT : en³)
  , (V, 99, (≡ en)) ↠ N : en
  , (V, 1,  (≡ en)) ↠ HT : sn ⊗ AT : sn
    -- Melodic Structure: Expand N to terminals
  , (N, 1,  (≡ hn)) ↠ CT : hn
  , (N, 50, (≡ qn)) ↠ CT : qn
  , (N, 50, (≡ qn)) ↠ ST : qn
  , (N, 45, (≡ qn)) ↠ R  : qn
  , (N, 20, (≡ qn)) ↠ L  : qn
  , (N, 1,  (≡ qn)) ↠ AT : qn
  , (N, 40, (≡ en)) ↠ CT : en
  , (N, 40, (≡ en)) ↠ ST : en
  , (N, 20, (≡ en)) ↠ L  : en
  , (N, 20, (≡ en)) ↠ R  : en
  , (N, 1,  (≡ en)) ↠ AT : en ]
```

The above is faithful to the original probabilistic grammar, modulo the insertion of repetitive *Let* constructs in several rules that produce duplicate terms. The grammar formalism previously presented in [Keller and Morrison 2007] did not have sharing capabilities (i.e. repetition), but it seemed appropriate to repeat melodic structures identically now that we are able to. Notice also that this grammar does not use any metadata and, therefore, its *meta* variable is instantiated to ().

4.2 Interpretation

The result we get from the previous grammar describes a melodic improvisation abstractly; there is a concrete rhythmic structure, but only a vague description of how the notes should function under a specific harmonic context. We now need to interpret this structure as a concrete melodic improvisation, thus translating the terminal symbols of our grammar to actual musical notes.

In order to do so, we require an abstract harmonic structure as input (*Music SemiChord*), indicating the chord progression the overall piece adheres to. For instance, this could be the result we get by rewriting on our grammar of tonal harmony in Section 3.

Moreover, we allow some control over the melodic output of the interpretation by letting the user provide a weighted choice of scales and octaves to use:

```
data MelodyConfig
  = MelodyConfig
    { scales  :: [(Double, ScaleType)]
    , octaves :: [(Double, Octave)] }
type MelodyInput = (MelodyConfig, Music SemiChord)
```

Below we present the conceptual skeleton of the process, omitting the gory details of the heuristics we employ to make the melody sound better:

```
instance Interpret MelodyInput M Pitch where
  interpret (cfg, chs) symbols = mapM interpretSymbol
                                 ∘ synchronize chs
    where
      interpretSymbol :: (SemiChord, M) → IO Pitch
      interpretSymbol (ch, s) =
        case s of
          CT → choose ch
          AT → choose $ (ch ↓ Mi2) ++ (ch ↑ Mi2)
          ST → choose $ filter (match chord) (scales cfg)
          ...
```

```
synchronize :: Music a → Music b → Music (a, b)
...
```

The main interpretation steps are to synchronize the harmonic background with the grammar terms and then select a valid note with respect to the current chord. In the case of scale tones, for instance, we select a note out of the user-supplied scales that match the current chord.

5 Rhythm

Moving on to the rhythmic part of the generation process, we attempt to model rhythmic improvisations in the tradition of North Indian classical music. We specifically focus on the traditional instrument *tabla*, since musicologists believe tabla improvisation follows precise rules akin to natural language [Kippen 2005]. Moreover, we can utilize an existing context-free grammar for a particular improvisation technique, called *qa'ida*, which was developed in collaboration with expert tabla players and later employed in the expert system Bol Processor BP1 [Bel 1992].

It is important to note that subsequent versions of the Bol Processor go even further to context-sensitive formalisms that allow inspection of a terminal's context while rewriting, but we refrain from extending our grammar formalism to accommodate such features.

5.1 Grammar

The grammar is rather simple, employing neither assignment of probabilistic weights nor dependence on durations. Note that non-determinism only arises when multiple rules match the same symbol. For convenience, we define a shorthand for this particular rule format[8]:

```
(↠) :: a → [a] → Rule meta a
x ↠ xs = (x, 1, always) ↣ fold1 ⊗ ((: en) ⟨$⟩ xs)
```

Our terminal symbols consist of syllables (traditionally known as *bols*), representing a different stroke of the instrument; a common form of transliteration in written Carnatic rhythms. Capitalized symbols represent non-terminals that guide the rewriting process:

```
data Syllable
  = -- terminals
    Tr | Kt | Dhee | Tee | Dha | Ta
  | Ti | Ge | Ke   | Na  | Ra  | Noop
    -- non-terminals
  | S   | XI  | XD | XJ  | XA | XB | XG | XH | XC
  | XE  | XF  | TA7 | TC2 | TE1 | TF1 | TF4 | TD1
  | TB2 | TE4 | TC1 | TB3 | TA8 | TA3 | TB1 | TA1
  deriving Eq
```

[8] We assume an eighth-note beat, but this can easily be controlled by the user.

The grammar can be conceptually divided in two layers: first, non-terminal symbols are rewritten to ones that have a fixed duration and, second, these are expanded to syllables of the corresponding duration. One could rewrite the top layer until fixpoint and then continue with the rest of the rules independently:

```
rhythm :: Grammar () Syllable
rhythm = S |:
  [S  ↠ [TE1, XI]
  ,XI  ↠ [TA7, XD], XD ↠ [TA8]
  ,XI  ↠ [TF1, XJ] ,XJ ↠ [TC2, XA]
  ,XA  ↠ [TA1, XB], XB ↠ [TB3, XD]
  ,XI  ↠ [TF1, XG], XG ↠ [TB2, XA]
  ,S   ↠ [TA1, XH]
  ,XH  ↠ [TF4, XB] ,XH ↠ [TA3, XC]
  ,XC  ↠ [TE4, XD], XC ↠ [TA3, XE]
  ,XE  ↠ [TA1, XD], XE ↠ [TC1, XD]
  ,XC  ↠ [TB1, XB]
  ,S   ↠ [TB1, XF]
  ,XF  ↠ [TA1, XJ] ,XF ↠ [TD1, XG]
  ---------------------------------
  ,TA7 ↠ [Kt, Dha, Tr, Kt, Dha, Ge, Na]
  ,TC2 ↠ [Tr, Kt], TE1 ↠ [Tr], TF1 ↠ [Kt]
  ,TF4 ↠ [Ti, Dha, Tr, Kt]
  ,TE4 ↠ [Ti, Noop, Dha, Ti]
  ,TD1 ↠ [Noop], TB2 ↠ [Dha, Ti], TC1 ↠ [Ge]
  ,TB3 ↠ [Dha, Tr, Kt]
  ,TA8 ↠ [Dha, Ti, Dha, Ge, Dhee, Na, Ge, Na]
  ,TA3 ↠ [Tr, Kt, Dha], TB1 ↠ [Ti], TA1 ↠ [Dha] ]
```

5.2 Interpretation

Interpreting the tabla syllables is as simple as giving a *ToMusic1* instance for the *Syllable* type, essentially converting a syllable to the MIDI number of that tabla sound[9]:

```
instance ToMusic1 Syllable where
  toMusic1 = toMusic ∘ pitch ∘ percussionMap
    where percussionMap :: Syllable → Int
          percussionMap s = case s of Tr → 38
                                      Kt → 45
                                      ...
```

6 Generated Results

In this section, we define different musical configurations and for each of them, generate several variations by repeating the non-deterministic rewriting procedure. For each

[9] These numbers are dictated by the specific MIDI map that the user's synthesizer will use to play the tabla sounds.

variation, we generate separate MIDI files for each part (melody, harmony, tabla), which we combine to produce sheet music and playable music files. All output is available on Github[10], but one can also listen to the generated songs on Soundcloud[11].

These outputs do not aim to be perceived as complete pieces of music, but rather demonstrations that our results exhibit a certain amount of variability and flexibility. For this reason, there has been no manual musical editing, apart from assigning different instruments to each voice and running a post-processing step that introduces more natural dynamics to the whole piece.

In order to generate a music piece, we require harmonic and melodic configurations as input and then run all three grammars to produce the final MIDI output:

```
generate :: FilePath → Dur
         → HarmonyConfig → MelodyConfig
         → IO ()
generate f t hCfg mCfg = do
  (absHarm, harm) ← runGrammar harmony t hCfg
  (_, mel) ← runGrammar melody t (mCfg, absHarm)
  (_, rhy) ← runGrammar rhythm t ()
  writeToMidiFile f (harm :=: mel :=: rhy)
```

We will ignore the tabla output for all configurations, except the last one in Subsection 6.4.

6.1 Sonata in E Minor

We start out with a very simple configuration for a sonata in E Minor. The pool of available chords and scales to choose are very limited, with only three triads for the harmony and two scales for improvisation. Both parts (chords and melody) are meant to be played by a single piano, resembling Classical-era pieces.

```
sonata
 = generate "sonata" (12 * wn)
   HarmonyConfig
     { basePc    = E
     , baseOct   = 4
     , baseScale = minor
     , chords    = equally [mi, maj, dim] }
   MelodyConfig
     { scales  = equally [ionian, harmonicMinor]
     , octaves = [(5, 4), (20, 5), (10, 6)] }
```

6.2 Romanian Elegy for Piano & Cello

The second piece is based on the Romanian scale, which is the fourth mode of the harmonic minor scale[12]. It usually evokes a feeling of mysticism that leads to interesting melodic improvisation, but also more unusual harmonizations. Nonetheless, to keep a classical sound, we restrict ourselves to a handful of chords without any tensions (9ths, 13ths, etc..):

```
romanianElegy
 = generate "romanian" (12 * wn)
   HarmonyConfig
     { basePc    = C
     , baseOct   = 4
     , baseScale = romanian
     , chords    = equally [mi, maj, aug, dim, m7, m7b5] }
   MelodyConfig
     { scales  = equally allScales
     , octaves = [(20, 3), (15, 4), (10, 5)] }
```

6.3 Byzantine Dance for Harpsichord

For the next piece, we use the even more exotic Byzantine scale, also known as double harmonic major scale. The configuration places no restriction on the chords and scales used by the generative process, hopefully leading to interesting results:

```
byzantineDance
 = generate "byzantine" (8 * wn)
   HarmonyConfig
     { basePc    = Fs
     , baseOct   = 4
     , baseScale = byzantine
     , chords    = equally allChords }
   MelodyConfig
     { scales  = equally allScales
     , octaves = [(1, 3), (20, 4), (15, 5), (1, 6)] }
```

6.4 Oriental Algebras for Metalophone, Sitar & Tablas

The final piece additionally uses the grammar of tabla improvisation, hence it is appropriate to switch to the more eastern Arabic scale, a heptatonic variant of the whole-half diminished scale. We leave chords completely unconstrained; the generation is allowed to pick freely from a comprehensive list of chords and scales we have defined. Moreover, we intend to play the melody on the Sitar instrument, so we

[10] https://github.com/omelkonian/music-grammars/tree/master/output
[11] https://soundcloud.com/haskell-music-grammars/sets

[12] Equivalently, the Romanian scale can be defined as the Dorian mode of major with its 4th augmented.

make sure that the melodic generation picks notes from appropriate octaves:

```
orientalAlgebras
 = generate "oriental" (12 * wn)
   HarmonyConfig
     { basePc    = A
     , baseOct   = 3
     , baseScale = arabian
     , chords    = equally allChords}
   MelodyConfig
     { scales  = equally allScales
     , octaves = [(20, 4), (15, 5), (5, 6)]}
```

7 Discussion

Jazz Harmony We abandoned our attempt to transcribe the grammar of jazz chord progressions presented in [Steedman 1984], because of its overly expressive context-sensitive rules; one has to inspect the surrounding context of a symbol to determine which rules to rewrite with. Nonetheless, Steedman later demonstrated an equivalent grammar that is (weakly) context-free [Steedman 1996]. It would certainly be worthwhile to examine whether it is possible to transcribe this equivalent grammar, although its presentation as a categorial grammar [Wood 2014] may prove hard to model with our current PTGG formulation.

This direction may also shine some light on the relative expressiveness of PTGGs; we believe them to lie somewhere between context-free and context-sensitive grammars and one should investigate intermediate grammar formalisms, such as tree-adjoining [Joshi and Schabes 1997] and multiple context-free grammars [Seki et al. 1991].

Better Interpretation Our work put most emphasis on the usage of PTGGs to generate abstract musical structures, rather than the a-posteriori interpretation of these abstract musical structures to actual music. This was not by accident, given that our interpretation is an ad-hoc mixture of brittle heuristics and only admits an informal presentation.

Nevertheless, [Quick and Hudak 2012] already provides a mathematically elegant, but also computationally feasible solution to the task of interpreting chord progressions (i.e. voice leading), based on the notion of chord spaces. In the same vein, it would be interesting to explore similar mathematically-ground techniques for melodic improvisation.

Inherently-Typed Grammars There are various ways in which our DSL permits the definition of ill-formed grammars, e.g. one could break the rewriting process by giving a non-converging grammar. Many such subtleties are currently the responsibility of the user of our DSL, but recent advances in Haskell's type system [Eisenberg et al. 2014; Peyton Jones et al. 2006; Yorgey et al. 2012] allow for the encoding of such constraints in the type of grammars [Lindley and McBride 2014]. For instance, it is possible to encode in the type-level that grammar rules preserve the total duration of the rewritten symbol, thus enforcing the linear usage of time values statically. We plan to enrich the type of grammars to statically enforce that grammars written in our DSL are well-formed, in a way that is as opaque as possible to the user.

Non-musical Domains While we utilized our grammatical framework to express music generation, it seems likely that the grammar formalism presented here is suitable for other domains than music. We believe it would be worthwhile to investigate similarities described in the *Algebraic Theory of Polymorphic Temporal Media* [Hudak 2004; Hudak and Janin 2014], e.g. to generate random graphic animations.

Weight Inference Many musical grammars found in the literature have been suggested for analytical purposes, e.g. a song is parsed using a grammar of tonal harmony, resulting in the harmonic structure of the piece.

In order to extract the generative dual of those grammars, we need to manually assign the probabilistic weights to overlapping rules, an error-prone process that requires domain-specific expertise. To tackle this problem, it seems appropriate to investigate statistical techniques that automatically infer the weights of the rules, based on some existing corpus of musical data. One could go even further to deduce the grammar rules themselves, but restriction to a less expressive grammar formalism would be necessary for tractable inference.

8 Conclusion

We have presented an extension of the original PTGG formalism and demonstrated that it is adequate to model complicated music generation schemes, just as those found in the musicology literature.

Furthermore, we are the first, to our knowledge, to attempt to model other aspects of music using formal grammars. We give example transcriptions of grammars found in the music literature: a grammar for Western tonal harmony [Rohrmeier 2011], one for melodic jazz improvisation [Keller and Morrison 2007] and one for North-Indian tabla drum improvisation [Bel 1992].

As a final contribution, we combine the grammars we transcribed to form complete music compositions and extract several variations for each configuration. We wish to emphasize that these example compositions are in no way chosen to form a coherent whole, since they employ techniques from unrelated music genres and, consequently, the generated out is doomed not to sound authentic enough.

Nonetheless, they act as an effective medium to demonstrate the capabilities of our grammatical framework, as well as evaluate the current prototypes we have at hand.

There is certainly room for further attempts to model a stylistically coherent set of grammars, which would eventually generate well-formed musical compositions. For instance, one could restrict the scope to the jazz genre, replacing our current grammar for harmony with one that models jazz chord progression as in [Steedman 1984]. The output of our melodic grammar would now be interpreted over these jazz chord progression, arguably leading to more realistic results. As discussed in the Section 7, this might require extending the formalism even further with context-sensitive features.

References

Bernard Bel. 1992. Modelling improvisatory and compositional processes. *Languages of Design, Formalisms for Word, Image and Sound* 1, 1 (1992), 11–26.

Richard A Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed type families with overlapping equations. *ACM SIGPLAN Notices* 49, 1 (2014), 671–683.

Paul Hudak. 2004. An algebraic theory of polymorphic temporal media. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 1–15.

Paul Hudak and David Janin. 2014. Tiled polymorphic temporal media. In *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*. ACM, 49–60.

Paul Hudak and Donya Quick. 2018. *The Haskell School of Music: From signals to Symphonies*. Cambridge University Press.

Aravind K Joshi and Yves Schabes. 1997. Tree-adjoining grammars. In *Handbook of formal languages*. Springer, 69–123.

Robert M Keller and David R Morrison. 2007. A grammatical approach to automatic improvisation. In *Proceedings, Fourth Sound and Music Conference, Lefkada, Greece, July*.

James Kippen. 2005. *The Tabla of Lucknow: A cultural analysis of a musical tradition*. Manohar Publishers.

Sam Lindley and Conor McBride. 2014. Hasochism: the pleasure and pain of dependently typed Haskell programming. *ACM SIGPLAN Notices* 48, 12 (2014), 81–92.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *ACM SIGPLAN Notices*, Vol. 41. ACM, 50–61.

Frank Pfenning and Conal Elliott. 1988. Higher-order abstract syntax. In *ACM sigplan notices*, Vol. 23. ACM, 199–208.

Donya Quick and Paul Hudak. 2012. Computing with chord spaces. In *ICMC*.

Donya Quick and Paul Hudak. 2013. Grammar-based automated music composition in Haskell. In *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design*. ACM, 59–70.

Martin Rohrmeier. 2011. Towards a generative syntax of tonal harmony. *Journal of Mathematics and Music* 5, 1 (2011), 35–53.

Heinrich Schenker and Oswald Jonas. 1935. *Der freie Satz*. Vol. 3. Universal Edition.

Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. 1991. On multiple context-free grammars. *Theoretical Computer Science* 88, 2 (1991), 191–229.

Mark Steedman. 1996. The blues and the abstract truth: Music and mental models. *Mental models in cognitive science* (1996), 305–318.

Mark J Steedman. 1984. A generative grammar for jazz chord sequences. *Music Perception: An Interdisciplinary Journal* 2, 1 (1984), 52–77.

Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 60–76.

Mary McGee Wood. 2014. *Categorial Grammars (RLE Linguistics B: Grammar)*. Routledge.

Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*. ACM, 53–66.