

AUTOMATIC MUSIC GENERATION USING CHRiSM

Jon Sneyers
K.U.Leuven, Belgium



DTAI seminar
November 2009

Part I

CHRiSM



- 1 Introduction: CHR
 - Constraint Handling Rules
 - CHR by example
- 2 CHRiSM
 - CHR(PRISM)
 - Syntax & semantics
 - PRISM features in **CHRiSM**
 - **CHRiSM** by example
- 3 Discussion and conclusion
 - Implementation of **CHRiSM**
 - Related formalisms
 - Conclusion



- 1 Introduction: CHR
 - Constraint Handling Rules
 - CHR by example
- 2 CHRiSM
- 3 Discussion and conclusion

Constraint Handling Rules [Frühwirth 1991]

5/47

- ▶ High-level language *extension*
 - ▶ different host languages (originally and mostly Prolog)
 - ▶ e.g. CHR(Prolog), CHR(Haskell), CHR(Java), CHR(C)
- ▶ Multi-headed committed-choice guarded rewrite rules
- ▶ Originally: designed for writing constraint solvers
- ▶ Today: general-purpose programming language

Example 1: less-or-equal solver

6/47

- ▶ Typical “solver” CHR program:

```
:- chr_constraint <= /2.
```

```
reflexivity @ X<=X <=> true.
```

```
antisymmetry @ X<=Y, Y<=X <=> X=Y.
```

```
idempotence @ X<=Y \ X<=Y <=> true.
```

```
transitivity @ X<=Y, Y<=Z ==> X<=Z.
```

- ▶ Example execution:

- ▶ Goal: $A \leq B, B \leq C, C \leq A$
- ▶ Store:

Example 1: less-or-equal solver

6/47

- ▶ Typical “solver” CHR program:

```
:- chr_constraint <= /2.
```

```
reflexivity @ X<=X <=> true.
```

```
antisymmetry @ X<=Y, Y<=X <=> X=Y.
```

```
idempotence @ X<=Y \ X<=Y <=> true.
```

```
transitivity @ X<=Y, Y<=Z ==> X<=Z.
```

- ▶ Example execution:

- ▶ Goal: $A \leq B$, $B \leq C$, $C \leq A$

- ▶ Store: $A \leq B$

Example 1: less-or-equal solver

6/47

- ▶ Typical “solver” CHR program:

```
:- chr_constraint <= /2.
```

```
reflexivity @ X<=X <=> true.
```

```
antisymmetry @ X<=Y, Y<=X <=> X=Y.
```

```
idempotence @ X<=Y \ X<=Y <=> true.
```

```
transitivity @ X<=Y, Y<=Z ==> X<=Z.
```

- ▶ Example execution:

- ▶ Goal: $A \leq B$, $B \leq C$, $C \leq A$

- ▶ Store: $A \leq B$, $B \leq C$

Example 1: less-or-equal solver

6/47

- ▶ Typical “solver” CHR program:

```
:- chr_constraint <= /2.
```

```
reflexivity @ X<=X <=> true.
```

```
antisymmetry @ X<=Y, Y<=X <=> X=Y.
```

```
idempotence @ X<=Y \ X<=Y <=> true.
```

```
transitivity @ X<=Y, Y<=Z ==> X<=Z.
```

- ▶ Example execution:

- ▶ Goal: $A \leq B$, $B \leq C$, $C \leq A$

- ▶ Store: $A \leq B$, $B \leq C$

Example 1: less-or-equal solver

6/47

- ▶ Typical “solver” CHR program:

```
:- chr_constraint <= /2.
```

```
reflexivity @ X<=X <=> true.
```

```
antisymmetry @ X<=Y, Y<=X <=> X=Y.
```

```
idempotence @ X<=Y \ X<=Y <=> true.
```

```
transitivity @ X<=Y, Y<=Z ==> X<=Z.
```

- ▶ Example execution:

- ▶ Goal: $A \leq B$, $B \leq C$, $C \leq A$
- ▶ Store: $A \leq B$, $B \leq C$, $A \leq C$

Example 1: less-or-equal solver

6/47

- ▶ Typical “solver” CHR program:

```
:- chr_constraint <= /2.
```

```
reflexivity @ X<=X <=> true.
```

```
antisymmetry @ X<=Y, Y<=X <=> X=Y.
```

```
idempotence @ X<=Y \ X<=Y <=> true.
```

```
transitivity @ X<=Y, Y<=Z ==> X<=Z.
```

- ▶ Example execution:

- ▶ Goal: $A \leq B$, $B \leq C$, $C \leq A$

- ▶ Store: $A \leq B$, $B \leq C$, $A \leq C$, $C \leq A$

Example 1: less-or-equal solver

6/47

- ▶ Typical “solver” CHR program:

```
:- chr_constraint <= /2.
```

```
reflexivity @ X<=X <=> true.
```

```
antisymmetry @ X<=Y, Y<=X <=> X=Y.
```

```
idempotence @ X<=Y \ X<=Y <=> true.
```

```
transitivity @ X<=Y, Y<=Z ==> X<=Z.
```

- ▶ Example execution:

- ▶ Goal: $A \leq B$, $B \leq C$, $C \leq A$

- ▶ Store: $A \leq B$, $B \leq C$, $A \leq C$, $C \leq A$

Example 1: less-or-equal solver

6/47

- ▶ Typical “solver” CHR program:

```
:- chr_constraint <= /2.
```

```
reflexivity @ X<=X <=> true.
```

```
antisymmetry @ X<=Y, Y<=X <=> X=Y.
```

```
idempotence @ X<=Y \ X<=Y <=> true.
```

```
transitivity @ X<=Y, Y<=Z ==> X<=Z.
```

- ▶ Example execution:

- ▶ Goal: $A \leq B, B \leq C, C \leq A$
- ▶ Store: $A \leq B, B \leq C, A = C$

Example 1: less-or-equal solver

6/47

- ▶ Typical “solver” CHR program:

```
:- chr_constraint <= /2.
```

```
reflexivity @ X<=X <=> true.
```

```
antisymmetry @ X<=Y, Y<=X <=> X=Y.
```

```
idempotence @ X<=Y \ X<=Y <=> true.
```

```
transitivity @ X<=Y, Y<=Z ==> X<=Z.
```

- ▶ Example execution:

- ▶ Goal: $A \leq B$, $B \leq C$, $C \leq A$
- ▶ Store: $A \leq B$, $B \leq A$, $A=C$

Example 1: less-or-equal solver

6/47

- ▶ Typical “solver” CHR program:

```
:- chr_constraint <= /2.
```

```
reflexivity @ X<=X <=> true.
```

```
antisymmetry @ X<=Y, Y<=X <=> X=Y.
```

```
idempotence @ X<=Y \ X<=Y <=> true.
```

```
transitivity @ X<=Y, Y<=Z ==> X<=Z.
```

- ▶ Example execution:

- ▶ Goal: $A \leq B$, $B \leq C$, $C \leq A$
- ▶ Store: $A \leq B$, $B \leq A$, $A=C$

Example 1: less-or-equal solver

6/47

- ▶ Typical “solver” CHR program:

```
:- chr_constraint <= /2.
```

```
reflexivity @ X<=X <=> true.
```

```
antisymmetry @ X<=Y, Y<=X <=> X=Y.
```

```
idempotence @ X<=Y \ X<=Y <=> true.
```

```
transitivity @ X<=Y, Y<=Z ==> X<=Z.
```

- ▶ Example execution:

- ▶ Goal: $A \leq B$, $B \leq C$, $C \leq A$

- ▶ Store: $A=B$, $A=C$

Example 1: less-or-equal solver

6/47

- ▶ Typical “solver” CHR program:

```
:- chr_constraint <= /2.
```

```
reflexivity @ X<=X <=> true.
```

```
antisymmetry @ X<=Y, Y<=X <=> X=Y.
```

```
idempotence @ X<=Y \ X<=Y <=> true.
```

```
transitivity @ X<=Y, Y<=Z ==> X<=Z.
```

- ▶ Example execution:

- ▶ Goal: $A \leq B$, $B \leq C$, $C \leq A$

- ▶ Store: $A=B$, $A=C$ ← answer

Example 2: Prime number generation

7/47

- ▶ Typical “general-purpose” CHR program:

```
:- chr_constraint upto/1, prime/1.  
upto(1) <=> true.  
upto(N) <=> N>1 | prime(N), upto(N-1).  
prime(I) \ prime(J) <=> J mod I ::= 0 | true.
```
- ▶ First two rules implement a loop, generating a sequence `prime(2), ..., prime(n)`
- ▶ Third rule filters out the non-prime numbers:
 - ▶ if I divides J, then J is not a prime
 - ▶ the rule removes such non-primes



1 Introduction: CHR

2 **CHRiSM**

- CHR(PRISM)
- Syntax & semantics
- PRISM features in **CHRiSM**
- **CHRiSM** by example

3 Discussion and conclusion

- ▶ New proposal for probabilistic CHR: CHRiSM
- ▶ based on CHR(PRISM)
- ▶ PRISM: PRogramming In Statistical Modeling
[Sato 1995, Sato & Kameya 1997]
- ▶ CHRiSM: CHance Rules induce Statistical Models



- ▶ PRISM built-in `msw/2` can be used in CHR(PRISM) programs
 - ▶ `msw(+Experiment, -Result)`: `Experiment` is ground at runtime; `Result` gets a random value based on a predefined discrete probability distribution

- ▶ For example:

```
values(coin, [head, tail]).  
:- set_sw(coin, [0.5, 0.5]).
```

```
toss <=> msw(coin, X), write(result=X).
```

- ▶ CHRiSM is “syntactic sugar” for CHR(PRISM)
 - ▶ relatively simple kind-of-source-to-source transformation

Syntax of CHRiSM

11/47

- ▶ Chance rules (may) have two kinds of probabilities:
 - ▶ Rule: application probability
 - ▶ Body: probabilistic disjunction

Syntax: rule with probability Prob

```
Prob ?? Head <=> Guard | Body.
```

normal CHR rules: "1 ??"

Syntax: probabilistic disjunction (in rule body)

fixed probability distribution: (cf. CP-Logic [Vennekens et al. 2006])

```
D1:Prob1 ; D2:Prob2 ; ... ; DN:ProbN
```

unknown/learnable probability distribution:

```
Prob ?? D1 ; D2 ; ... ; DN
```

Syntax of CHRiSM

11/47

- ▶ Chance rules (may) have two kinds of probabilities:
 - ▶ Rule: application probability
 - ▶ Body: probabilistic disjunction

Syntax: rule with probability Prob

Prob ?? Head \Leftrightarrow Guard | Body.

normal CHR rules: “1 ??”

Syntax: probabilistic disjunction (in rule body)

fixed probability distribution: (cf. CP-Logic [Vennekens et al. 2006])

D1:Prob1 ; D2:Prob2 ; ... ; DN:ProbN

unknown/learnable probability distribution:

Prob ?? D1 ; D2 ; ... ; DN

Syntax of CHRiSM

11/47

- ▶ Chance rules (may) have two kinds of probabilities:
 - ▶ Rule: application probability
 - ▶ Body: probabilistic disjunction

Syntax: rule with probability Prob

Prob ?? Head \Leftrightarrow Guard | Body.

normal CHR rules: “1 ??”

Syntax: probabilistic disjunction (in rule body)

fixed probability distribution: (cf. CP-Logic [Vennekens et al. 2006])

D1:Prob1 ; D2:Prob2 ; ... ; DN:ProbN

unknown/learnable probability distribution:

Prob ?? D1 ; D2 ; ... ; DN

- ▶ Operational semantics as usual ($\omega_t, \omega_r, \omega_p$)
- ▶ Two differences:
 - ▶ rule application can be skipped (with probability $1 - P$)
 - ▶ probabilistic disjunctions in the body: one disjunct is randomly chosen (committed-choice)

Probability expressions

13/47

- ▶ Different kinds of probability expressions Prob ?? allowed:
 - ▶ numbers:
head:0.5 ; tail:0.5
 - ▶ arithmetic expression which is ground at runtime:
eval(X/100) ?? foo(X) ==> bar.
 - ▶ probabilities are unknown:
roll <=> die ?? 1 ; 2 ; 3 ; 4 ; 5 ; 6.
 - ▶ probabilities are unknown and parametrized:
roll(X) <=> die(X) ?? 1 ; 2 ; 3 ; 4 ; 5 ; 6.
- ▶ Numbers and arithmetic expressions: fixed probabilities
- ▶ Parametrized probabilities (with 0 or more parameters):
 - ▶ initially: uniform distribution
 - ▶ actual distribution can be learned from examples

Features of PRISM

14/47

- ▶ PRISM has many nice features, a.o.:
 - ▶ Probabilistic execution (`sample`)
 - ▶ Probability computation (`prob`)
 - ▶ EM-learning (`learn`)
- ▶ These features can also be used in CHRiSM

PRISM features in CHRiSM

15/47

- ▶ Probabilistic execution: `sample goal`
 - ▶ starting from `goal`, apply CHRiSM rules (just like in CHR)
 - ▶ rules with probability P are skipped with probability $1 - P$
 - ▶ in a probabilistic disjunction, exactly one disjunct is chosen
- ▶ Probability computation: `prob goal <==> result`
 - ▶ compute probability that “`sample goal`” gives “`result`”
 - ▶ `prob goal ===> result`
compute probability that “`sample goal`” gives something of the form “`result, otherstuff`”
- ▶ EM-learning: `learn(observations)`
 - ▶ `observations`: a list of observations of the form “`goal <==> result`” or “`goal ===> result`”
 - ▶ compute an assignment to the unknown probabilities such that the likelihood of the observations is maximized

Demonstration of CHRiSM

16/47

Two examples:

- ▶ coin toss
- ▶ RISK





- 1 Introduction: CHR
- 2 CHRiSM
- 3 Discussion and conclusion
 - Implementation of **CHRiSM**
 - Related formalisms
 - Conclusion

Implementation of CHR(PRISM)

18/47

- ▶ First prototype used `toychr` [Duck 2004]
 - ▶ naive implementation of CHR, very inefficient
 - ▶ uses only pure Prolog
- ▶ Current implementation based on Leuven CHR system [Schrijvers et al 2004]
- ▶ Important remaining issue: how to get efficient explanation search?
 - ▶ tabling is not an option (CHR is bottom-up, has a large state)
 - ▶ need some mechanism to prune uninteresting derivations

Related formalisms

19/47

- ▶ CP-logic (LPADs) [Vennekens et al. 2006] can be encoded in CHRiSM
 - ▶ details in CHR'09 paper
- ▶ Many other probabilistic logic programming formalisms are sublogics of CP-logic:
 - ▶ PRISM itself
 - ▶ ProbLog [De Raedt et al. 2007]
 - ▶ ICL [Poole 1997]
- ▶ Bayesian network-inspired formalisms:
 - ▶ BLP [Kersting & De Raedt 2007] covered by CHRiSM
 - ▶ Others are more difficult (they support more complicated distributions):
 - ▶ RBN [Jaeger 1997]
 - ▶ CLP(BN) [Santos Costa et al. 2008]
 - ▶ Blog [Milch et al. 2007]

- ▶ New way to add probabilities to CHR: CHRiSM
 - ▶ based on CHR(PRISM)
 - ▶ has advantages over PCHR [Frühwirth et al. 2002]
- ▶ Download implementation:
<http://www.cs.kuleuven.be/~jon/chris>
- ▶ Subsumes other probabilistic-logic formalisms

- ▶ Language design
 - ▶ current syntax/semantics good in practice?
 - ▶ need to investigate more examples
- ▶ Efficient implementation
 - ▶ essential for feasible explanation search/learning
 - ▶ PRISM uses tabling, cf. work on CHR+tabling (e.g. in XSB)
 - ▶ perhaps consider CHR(ProbLog) too?
- ▶ Notion of probabilistic termination
 - ▶ program can terminate probabilistically but not classically
 - ▶ no problem for sampling
 - ▶ problem (of PRISM) for probability computation / learning
- ▶ Declarative semantics for CHRiSM
 - ▶ based on distribution semantics of PRISM?
 - ▶ direct model semantics for CHRiSM?
 - ▶ soundness/completeness of operational sem. w.r.t. decl. sem.

Part II

Automatic Music Generation

The image displays a musical score for the piece 'APopCALeasPs'. It is written in 3/4 time and consists of three systems of music. Each system includes a vocal line (treble clef), a piano accompaniment (treble clef), and a bass line (bass clef). The first system begins with a tempo marking of 120. The notation includes various rhythmic values such as quarter, eighth, and sixteenth notes, as well as rests and chords. The second system starts at measure 6, and the third system starts at measure 13.

- 4 Introduction
- 5 APopCALeasPs
 - Overview
 - Demo
 - A bit more detail
- 6 Some code fragments
 - Constraint declarations
 - Chord generation
 - Rhythm generation
 - Note generation
- 7 Conclusion
 - Conclusion
 - Future Work

Musical score for measures 1-5. The score is in 3/4 time and features a tempo marking of $\text{♩} = 120$. It consists of four staves: a vocal line (treble clef) and three piano accompaniment staves (treble and two bass clefs). The music begins with a quarter rest in the vocal line, followed by a series of quarter and eighth notes in the piano parts.

Musical score for measures 6-11. The score continues from the previous system. It features the same four-staff structure. The vocal line has a quarter rest in measure 6, followed by eighth notes in measures 7-11. The piano accompaniment provides a rhythmic and harmonic foundation.

Musical score for measures 12-15. The score continues from the previous system. It features the same four-staff structure. The vocal line has a quarter rest in measure 12, followed by eighth notes in measures 13-15. The piano accompaniment continues with its rhythmic pattern.

4 Introduction

5 APopCALeaPs

6 Some code fragments

7 Conclusion

- ▶ Goal: automatically generate original, royalty-free music
- ▶ Could be used in:
 - ▶ railway stations, airports, waiting rooms, stores (places where people are used to listening to crappy music)
 - ▶ computer games (music style influenced by game events)
 - ▶ tools for human composers, e.g. to get inspiration
- ▶ Should have a (probabilistic) learning component:
 - ▶ some musical rules are known, most are not
 - ▶ for some genres (e.g. Renaissance counterpoint), exhaustive enumeration of the rules is possible, but still:
 - ▶ need an expert who knows all rules
 - ▶ need to write out all rules in some formalism (tedious!)
 - ▶ from all pieces that satisfy the rules, some will be better than others; how to pick a solution?
 - ▶ *de gustibus non disputandum est* : the system has to be able to adjust to the musical taste of the user

The image displays a musical score for the piece 'APopCALeasPs'. It is written in 3/4 time and consists of three systems of music. The first system begins with a tempo marking of $\text{♩} = 120$. Each system contains four staves: a vocal line in the top staff, a piano accompaniment in the second staff, a bass line in the third staff, and a double bass line in the bottom staff. The notation includes various rhythmic values, rests, and chord symbols.

4 Introduction

5 APopCALeasPs

- Overview
- Demo
- A bit more detail

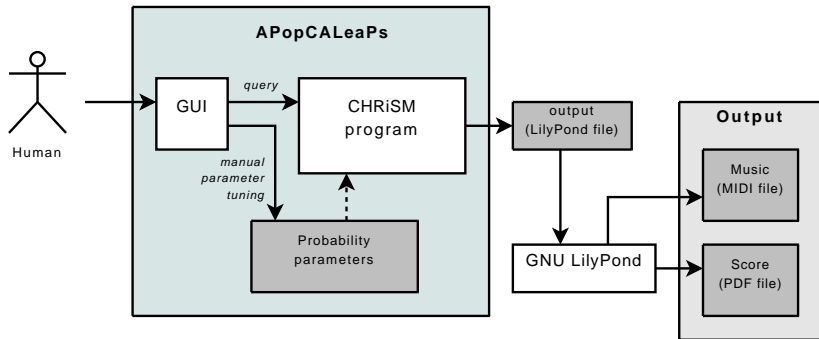
6 Some code fragments

7 Conclusion

APopCAlEaPs - current system

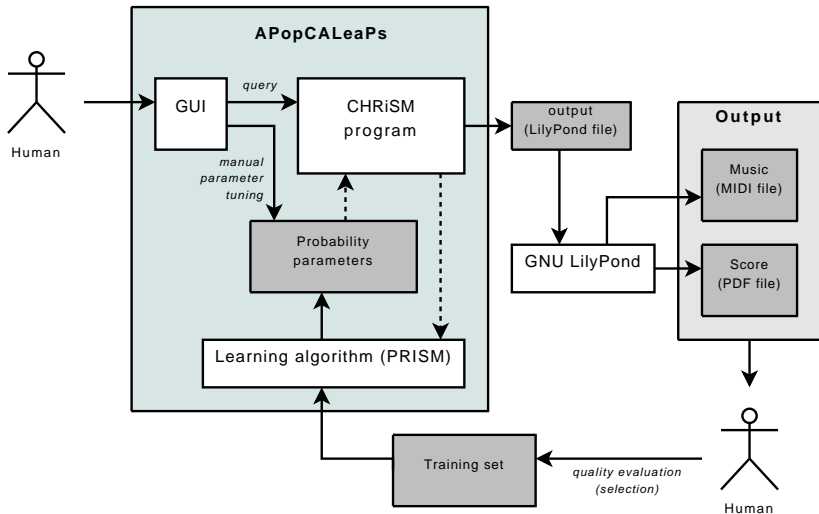
27/47

► Automatic Pop Composer (And Learner of Parameters)



APopCALeAPs - planned system

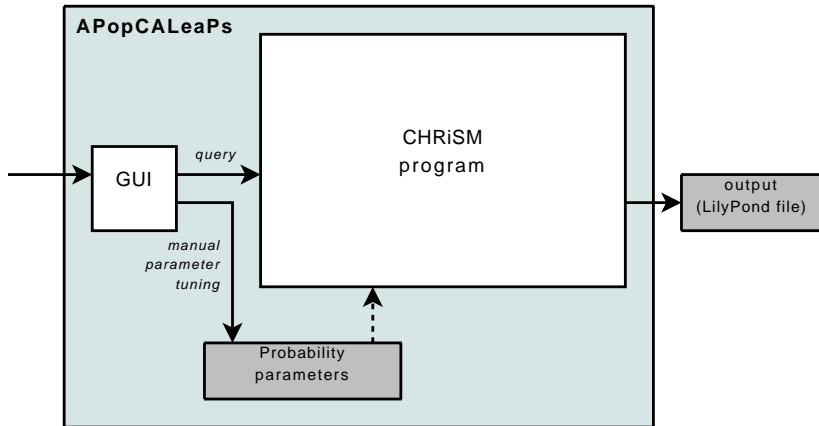
28/47



[demo of the APopCAlEaPs system]

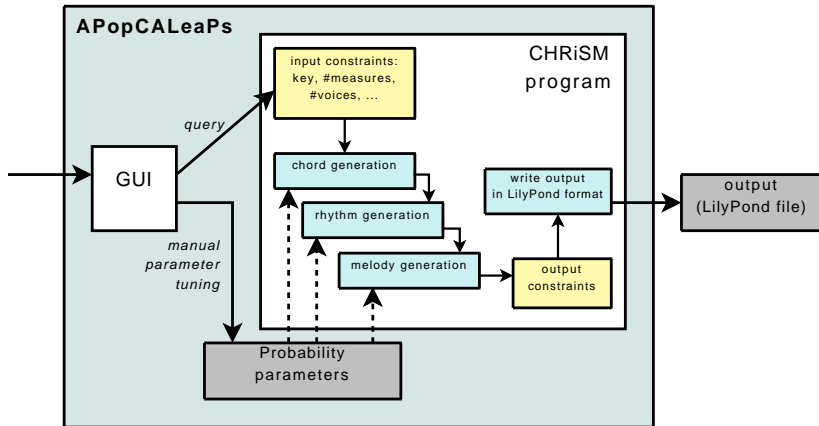
APopCAlEaPs generation process

30/47



APopCALeAPs generation process

31/47



120

4 Introduction

5 APopCALeAPs

6 Some code fragments

- Constraint declarations
- Chord generation
- Rhythm generation
- Note generation

7 Conclusion

Constraint declarations (1)

33/47

```
% inputs
:- chrism measures(+int), meter(+int,+duration), repeats(+int),
   key(+key), shortest_duration(+voice,+duration), tempo(+int),
   voice(+voice), range(+voice,+note,+int,+note,+int),
   max_jump(+voice,+int), instrument(+voice,+),
   chord_style(+cstyle), max_repeat(+voice,+int).

:- chr_type key ---> major ; minor.
:- chr_type voice ---> melody ; chords ; bass ; drums.
:- chr_type note ---> c ; d ; e ; f ; g ; a ; b.
:- chr_type duration ---> 2 ; 4 ; 8 ; 16 ; 32.
:- chr_type cstyle ---> offbeat ; long ; onbeat.
```

Constraint declarations (2)

34/47

```
% outputs
:- chrism measure(+measure), mchord(+int,+chord),
    beat(+voice,+measure,+int,+float,+duration),
    note(+voice,+measure,+int,+float,+),
    octave(+voice,+measure,+int,+float,+).

:- chr_type chord ---> c ; d ; e ; f ; g ; a ; b ;
    cm ; dm ; em ; fm ; gm ; am ; bm.
:- chr_type measure == int.
```

Constraint declarations (3)

35/47

```
% internals
:- chrism make_measures(+int), next_measure(+measure,+measure),
         make_beats(+int,+duration,+measure,+voice),
         next_beat(+voice,+measure,+int,+float,+measure,+int,+float),
         phase(+), chord(+,+,+,+,+),
         octave_d(+voice,+measure,+int,+float,+),
         octave_rangecheck(+voice,+measure,+int,+float,+),
         same_note_counter(+voice,+measure,+int,+float,+int),
         make_notes_measure(+int), find_octave_d(+,+,+,+,+).

% ...
```

Chord generation (1)

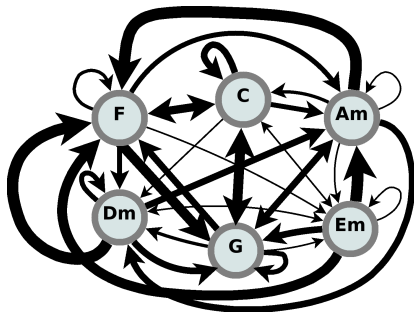
36/47

```
key(major), measure(1) ==> mchord(1,c).  
key(major), measures(N) ==> mchord(N,c).  
key(minor), measure(1) ==> mchord(1,am).  
key(minor), measures(N) ==> mchord(N,am).
```

Chord generation (2)

37/47

```
% simple Markov chain chord progression  
mchord(A,Chord), next_measure(A,B), measures(M)  
==> B < M |  
    msw(chord_choice(Chord),NextChord),  
    mchord(B,NextChord).
```



Rhythm generation (1)

38/47

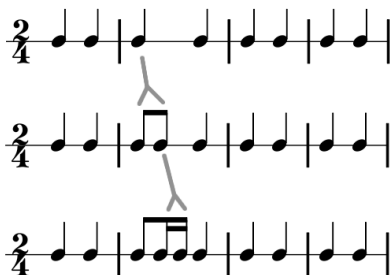


```
% create one beat per beat
meter(N,D), voice(V), measure(M) ==> make_beats(N,D,M,V).
make_beats(0,_,_,_) <=> true.
make_beats(N,D,M,V) <=> N > 0 |
    N1 is N-1, next_beat(V,M,N1,0,M,N,0),
    beat(V,M,N1,0,D), make_beats(N1,D,M,V).

meter(N,D), next_measure(M,M2)
    \ next_beat(V,A,B,C,M,N,E) <=> next_beat(V,A,B,C,M2,0,0).
```

Rhythm generation (2)

39/47



```
% split some of the beats in two  
split_beat(V) ??
```

```
meter(_,OD), measures(LastM), phase(split), shortest_duration(V,SD)  
\ beat(V,M,N,X,D), next_beat(V,M,N,X,NM,NN,NX) <=> D<SD, M \== LastM |  
  D2 is D*2, X2 is X+1/(D2/OD),  
  next_beat(V,M,N,X,M,N,X2), next_beat(V,M,N,X2,NM,NN,NX),  
  beat(V,M,N,X,D2), beat(V,M,N,X2,D2).
```

Note generation (1)

40/47

Special cases: instruments “drums” and “chords”

```
phase(make_notes), beat(drums,M,N,X,D) ==>
  abstract_beat(M,N,X,AB),
  msw(drum_choice(AB),Note),
  note(drums,M,N,X,Note).
```

```
phase(make_notes), beat(chords,M,N,X,D),
mchord(M,C), chord_style(Style) ==>
  abstract_beat(M,N,X,AB),
  msw(chord_type(Style,AB),Chord),
  chord(C,M,N,X,Chord).
```

Note generation (2)

41/47

```
% choose first note
make_notes_measure(1), beat(V,1,0,0,D), mchord(1,C) ==>
    V \== drums, V \== chords |
    abstract_beat(1,0,0,AB),
    soft_msw(note_choice(V,C,AB),Note),
    note(V,1,0,0,Note).

% choose next note and octave
make_notes_measure(M), beat(V,M,N,X,D), mchord(M,C),
octave(V,M1,N1,X1,00), next_beat(V,M1,N1,X1,M,N,X) ==>
    V \== drums, V \== chords |
    abstract_beat(M,N,X,AB),
    soft_msw(note_choice(V,C,AB),Note),
    note(V,M,N,X,Note),
    ( Note == r -> octave_d(V,M,N,X,0)
      ; find_octave_d(V,M,N,X,00) ).
```

Backtrackable msw/2

42/47

Why `soft_msw`?

- ▶ normal `msw` randomly picks a value and commits to it
- ▶ `soft_msw` picks a value, if it fails, it picks a different value
- ▶ useful construct to combine probabilistic choice and integrity constraints

```
% check max_jump constraint - fail (and backtrack) if it is violated
max_jump(V,MInt), octave(V,M1,N1,X1,00), note(V,M1,N1,X1,0N),
note(V,M,N,X,NN), next_beat(V,M1,N1,X1,M,N,X) \ octave(V,M,N,X,NO) <=>
    interval(0N,00,NN,NO,Int), Int > MInt | fail.
```

```
% check max_repeat constraint - fail (and backtrack) if it is violated
max_repeat(V,N), same_note_counter(V,A,B,C,N) <=> fail.
```

Rhythm revised

43/47



```
% two successive notes of the same pitch can be joined
join_notes(V,cond M=M2,cond N=N2) ??
  phase(join_notes), next_beat(V,M,N,X,M2,N2,X2),
  note(V,M2,N2,X2>Note) \ note(V,M,N,X>Note) <=>
    \+ has_tilde(Note), V \== drums |
    note(V,M,N,X>Note+' ~').
```

4 = 120

4

6

12

4 Introduction

5 APopCALeasPs

6 Some code fragments

7 Conclusion

- Conclusion
- Future Work

- ▶ CHRiSM allows very high-level music modelling
 - ▶ the entire music generation program is less than 500 lines (including whitespace and comments), most of which is output formatting etc.
- ▶ CHRiSM is an interesting hybrid programming paradigm
 - ▶ Usual approach in music generation: **either** probabilistic **or** constraint-based
 - ▶ First time a combined approach is tried (AFAIK)

This is work in progress...

46/47

... so there is a lot of future work:

- ▶ Current model of music is very simplistic
 - ▶ add (more) hidden states etc.
 - ▶ collaborate with a musicologist to improve it?
- ▶ Learning
 - ▶ EM learning from examples: infeasible?
 - ▶ semi-automatic parameter tuning?
 - ▶ learning for CHRiSM in general needs to be studied
- ▶ Music analysis / automatic classification
 - ▶ train several instances of the model with different genres/composers/styles/...
 - ▶ probability of a piece in each model indicates likelihood of belonging to a genre/...
- ▶ Experimental evaluation
 - ▶ can do this for analysis (but requires a lot of tedious data preparation)
 - ▶ how to do this for synthesis? Turing test?

▶ *Questions?*