

AOPCALEAPS: AUTOMATIC MUSIC GENERATION WITH CHRiSM

Jon Sneyers and Danny De Schreye
K.U.Leuven, Belgium





1 CHRiSM

- Constraint Handling Rules (CHR)
- **CHRiSM**
- Syntax & semantics of **CHRiSM**
- PRISM features in **CHRiSM**

2 AOPCALEAPS

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

3 Conclusion



1 CHRiSM

- Constraint Handling Rules (CHR)
- CHRiSM
- Syntax & semantics of CHRiSM
- PRISM features in CHRiSM

2 AOPCALEAPS

3 Conclusion

Constraint Handling Rules [Frühwirth 1991]

4/31

- ▶ 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

5/31

- ▶ Typical “solver” CHR program:

```
:- chr_constraint  $\leq$ /2.
```

```
reflexivity @  $X \leq X \iff \text{true}$ .
```

```
antisymmetry @  $X \leq Y, Y \leq X \iff X=Y$ .
```

```
idempotence @  $X \leq Y \setminus X \leq Y \iff \text{true}$ .
```

```
transitivity @  $X \leq Y, Y \leq Z \implies X \leq Z$ .
```

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

Example 1: less-or-equal solver

5/31

- ▶ Typical “solver” CHR program:

```
:- chr_constraint  $\leq$ /2.
```

```
reflexivity @  $X \leq X \iff \text{true}$ .
```

```
antisymmetry @  $X \leq Y, Y \leq X \iff X = Y$ .
```

```
idempotence @  $X \leq Y \setminus X \leq Y \iff \text{true}$ .
```

```
transitivity @  $X \leq Y, Y \leq Z \implies X \leq Z$ .
```

- ▶ Example execution:

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

- ▶ Store: $A \leq B$

Example 1: less-or-equal solver

5/31

- ▶ Typical “solver” CHR program:

```
:- chr_constraint  $\leq$ /2.
```

```
reflexivity @  $X \leq X \iff \text{true}$ .
```

```
antisymmetry @  $X \leq Y, Y \leq X \iff X = Y$ .
```

```
idempotence @  $X \leq Y \setminus X \leq Y \iff \text{true}$ .
```

```
transitivity @  $X \leq Y, Y \leq Z \implies X \leq Z$ .
```

- ▶ 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

5/31

- ▶ Typical “solver” CHR program:

```
:- chr_constraint  $\leq$ /2.
```

```
reflexivity @  $X \leq X \iff \text{true}$ .
```

```
antisymmetry @  $X \leq Y, Y \leq X \iff X=Y$ .
```

```
idempotence @  $X \leq Y \setminus X \leq Y \iff \text{true}$ .
```

```
transitivity @  $X \leq Y, Y \leq Z \implies X \leq Z$ .
```

- ▶ 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

5/31

- ▶ Typical “solver” CHR program:

```
:- chr_constraint  $\leq$ /2.
```

```
reflexivity @  $X \leq X \iff \text{true}$ .
```

```
antisymmetry @  $X \leq Y, Y \leq X \iff X=Y$ .
```

```
idempotence @  $X \leq Y \setminus X \leq Y \iff \text{true}$ .
```

```
transitivity @  $X \leq Y, Y \leq Z \implies X \leq Z$ .
```

- ▶ 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

5/31

- ▶ 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

5/31

- ▶ 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

5/31

- ▶ Typical “solver” CHR program:

```
:- chr_constraint  $\leq$ /2.
```

```
reflexivity @  $X \leq X \iff \text{true}$ .
```

```
antisymmetry @  $X \leq Y, Y \leq X \iff X=Y$ .
```

```
idempotence @  $X \leq Y \setminus X \leq Y \iff \text{true}$ .
```

```
transitivity @  $X \leq Y, Y \leq Z \implies X \leq Z$ .
```

- ▶ 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

5/31

- ▶ Typical “solver” CHR program:

```
:- chr_constraint  $\leq$ /2.
```

```
reflexivity @  $X \leq X \iff \text{true}$ .
```

```
antisymmetry @  $X \leq Y, Y \leq X \iff X=Y$ .
```

```
idempotence @  $X \leq Y \setminus X \leq Y \iff \text{true}$ .
```

```
transitivity @  $X \leq Y, Y \leq Z \implies X \leq Z$ .
```

- ▶ 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

5/31

- ▶ Typical “solver” CHR program:

```
:- chr_constraint  $\leq$ /2.
```

```
reflexivity @  $X \leq X \iff \text{true}$ .
```

```
antisymmetry @  $X \leq Y, Y \leq X \iff X=Y$ .
```

```
idempotence @  $X \leq Y \setminus X \leq Y \iff \text{true}$ .
```

```
transitivity @  $X \leq Y, Y \leq Z \implies X \leq Z$ .
```

- ▶ 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

5/31

- ▶ 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

5/31

- ▶ 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

6/31

- ▶ 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

- ▶ Probabilistic CHR: CHRiSM [ICLP 2010]
- ▶ 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

9/31

- ▶ 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

9/31

- ▶ 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

9/31

- ▶ 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)

Features of PRISM

11/31

- ▶ 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

12/31

- ▶ 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



1 CHRiSM

2 AOPCALEAPS

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

3 Conclusion

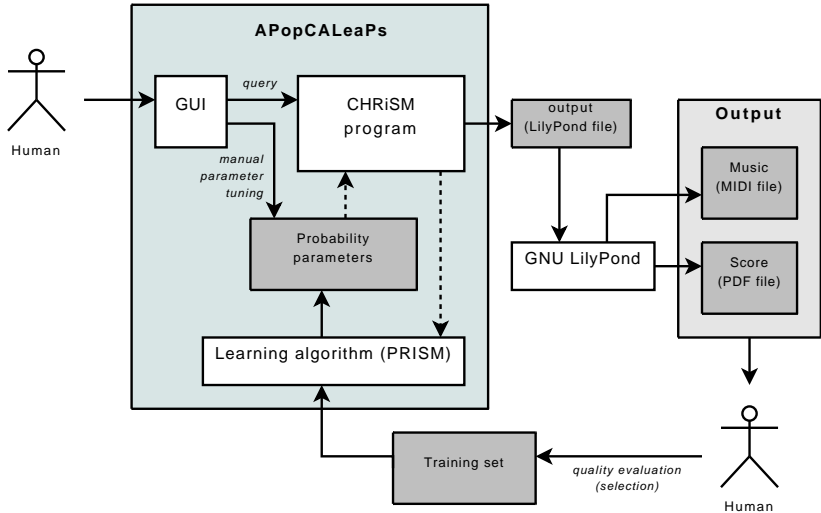
Automatic Music Generation

14/31

- ▶ 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

AOPCALEAPS - overview

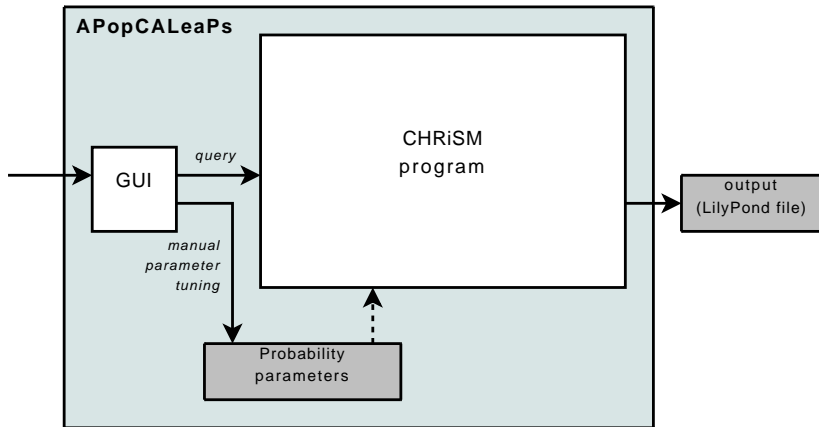
15/31



[demo of the APopCALeAPs system]

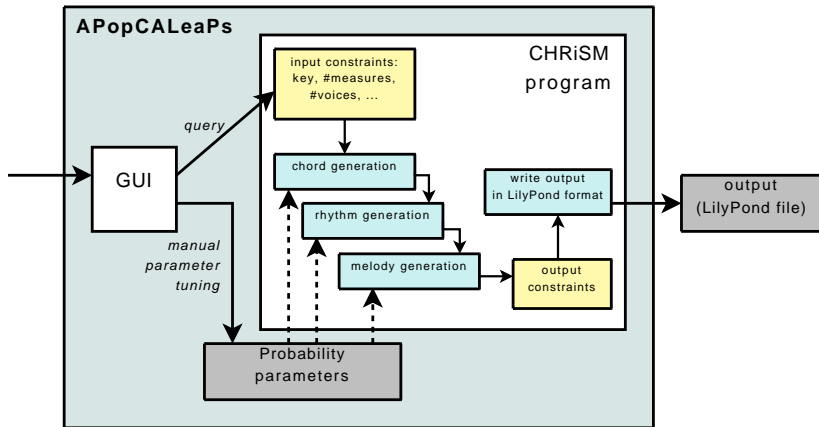
APopCALeAPs generation process

17/31



APopcALeaPs generation process

18/31



Constraint declarations (1)

19/31

```
% 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)

20/31

```
% 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.
```

Chord generation (1)

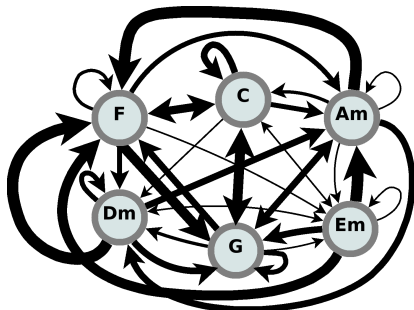
21/31

```
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)

22/31

```
% 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)

23/31

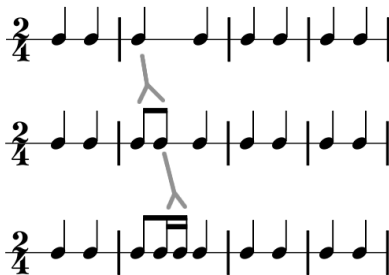


```
% create one beat per beat
meter(N,D), voice(V), measure(M) ==> make_beats(N,D,M,V).
make_beats(0,_,_,_) <=> true.
make_beats(N,D,M,V) <=> N > 0 |
    N1 is N-1, next_beat(V,M,N1,0,M,N,0),
    beat(V,M,N1,0,D), make_beats(N1,D,M,V).

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)

24/31



```
% 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

25/31

```
% 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,OO), 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,OO) ).
```

Why soft_msw?

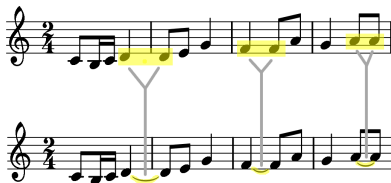
- ▶ normal msw randomly picks a value and commits to it
- ▶ soft_msw picks a value, if it fails, it picks a different value
- ▶ useful construct to combine probabilistic choice and integrity constraints

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

27/31



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



- 1 CHRiSM
- 2 AOPCALEAPS
- 3 Conclusion

- ▶ 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...

30/31

... 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?
 - ▶ trade-off with learning efficiency
- ▶ 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?*