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
Introduction: CHR
- Constraint Handling Rules
- CHR by example

CHRiSM

Discussion and conclusion
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

Typical “solver” CHR program:

```prolog
:- 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≤B, B≤C, C≤A
- Store:
Example 1: less-or-equal solver

Typical “solver” CHR program:

```prolog
:- 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 <= B, B <= C, C <= A
- Store: A <= B
Example 1: less-or-equal solver

Typical “solver” CHR program:

```prolog
:- 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 <= B, B <= C, C <= A
- Store: A <= B, B <= C
Example 1: less-or-equal solver

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 <= B, B <= C, C <= A
- Store: A <= B, B <= C
Typical “solver” CHR program:

```
:- chr_constraint \leq/2.
reflexivity  @ X \leq X <=> true.
antisymmetry @ X \leq Y, Y \leq X <=> X=Y.
idempotence  @ X \leq Y \ \ X \leq Y <=> true.
transitivity @ X \leq Y, Y \leq Z ==> 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

Typical “solver” CHR program:

```prolog
:- 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 <= B, B <= C, C <= A
- Store: A <= B, B <= C, A <= C, C <= A
Example 1: less-or-equal solver

Typical “solver” CHR program:

```prolog
:- 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 ≤ B, B ≤ C, C ≤ A
- Store: A ≤ B, B ≤ C, A ≤ C, C ≤ A
Example 1: less-or-equal solver

Typical “solver” CHR program:

\[
\text{:- chr_constraint } \leq/2.
\]

- reflexivity \( @ X \leq X \iff \text{true.} \)
- antisymmetry \( @ X \leq Y, Y \leq X \iff X = Y. \)
- idempotence \( @ X \leq Y \lor X \leq Y \iff \text{true.} \)
- 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

Typical “solver” CHR program:

```prolog
:- 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 <= B, B <= C, C <= A
- Store: A <= B, B <= A, A = C
Typical “solver” CHR program:

```prolog
:- 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 <= B, B <= C, C <= A
- Store: A <= B, B <= A, A = C
Example 1: less-or-equal solver

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 <= B, B <= C, C <= A
- Store: A = B, A = C
Example 1: less-or-equal solver

Typical “solver” CHR program:
\[
\text{:- chr_constraint } \leq /2.
\]

reflexivity @ \( X \leq X \iff \text{true} \).
antisymmetry @ \( X \leq Y, Y \leq X \iff X = Y \).
idempotence @ \( X \leq Y \backslash X \leq Y \iff \text{true} \).
transitivity @ \( X \leq Y, Y \leq Z \Rightarrow X \leq Z \).

Example execution:
- Goal: \( A \leq B, B \leq C, C \leq A \)
- Store: \( A = B, A = C \) ← answer
Introduction: CHR

Discussion and conclusion

Constraint Handling Rules

CHR by example

Example 2: Prime number generation

Typical “general-purpose” CHR program:

```prolog
:- 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 CH RiSM
   - CHR(PRISM)
   - Syntax & semantics
   - PRISM features in CH RiSM
   - CH RiSM by example

3 Discussion and conclusion
New proposal for probabilistic CHR: CHRIISM

based on CHR(PRISM)

PRISM: PRogramming In Statistical Modeling
[Sato 1995, Sato & Kameya 1997]

CHRIISM: 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:
  ```prolog
  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
Chance rules (may) have two kinds of probabilities:

- Rule: application probability
- Body: probabilistic disjunction

Syntax: rule with probability Prob

\[
\text{Prob ?? Head } \leftrightarrow \text{Guard | Body.}
\]

normal CHR rules: “1 ??”

Syntax: probabilistic disjunction (in rule body)

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

\[
\text{D1:Prob1 ; D2:Prob2 ; \ldots ; DN:ProbN}
\]

unknown/learnable probability distribution:

\[
\text{Prob ?? D1 ; D2 ; \ldots ; DN}
\]
Syntax of CHRIISM

- 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
```
Chance rules (may) have two kinds of probabilities:

- Rule: application probability
- Body: probabilistic disjunction

Syntax: rule with probability \( \text{Prob} \)

\[
\text{Prob} \ ? ? \ \text{Head} \ \Leftrightarrow \ \text{Guard} \mid \text{Body}.
\]

normal CHR rules: “1 ??”

Syntax: probabilistic disjunction (in rule body)

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

\[
\text{D1:Prob1} \ ; \ \text{D2:Prob2} \ ; \ldots \ ; \ \text{DN:ProbN}
\]

unknown/learnable probability distribution:

\[
\text{Prob} \ ? ? \ \text{D1} \ ; \ \text{D2} \ ; \ldots \ ; \ \text{DN}
\]
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)
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

- PRISM has many nice features, a.o.:
  - Probabilistic execution (sample)
  - Probability computation (prob)
  - EM-learning (learn)
- These features can also be used in CHRIISM
Probabilistic execution: sample goal
- starting from goal, apply CHRIISM 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 $\iff$ result
- compute probability that “sample goal” gives “result”
- prob goal $\implies$ 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 $\iff$ result” or “goal $\implies$ result”
- compute an assignment to the unknown probabilities such that the likelihood of the observations is maximized
Two examples:

- coin toss
- RISK
1 Introduction: CHR

2 CHRIISM

3 Discussion and conclusion
   - Implementation of CHRIISM
   - Related formalisms
   - Conclusion

Jon Sneyers
Automatic Music Generation using CHRIISM
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

- CP-logic (LPADs) [Vennekens et al. 2006] can be encoded in CHRI\textsc{m}
  - 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 CHRI\textsc{m}
  - 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: CHRIISM
- based on CHR(PRISM)
- has advantages over PCHR [Frühwirth et al. 2002]

Download implementation:

Subsumes other probabilistic-logic formalisms
Future work

- **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 CHRI SM**
  - based on distribution semantics of PRISM?
  - direct model semantics for CHRI SM?
  - soundness/completeness of operational sem. w.r.t. decl. sem.
Part II

Automatic Music Generation
4 Introduction
5 APopCALeaPs
   • Overview
   • Demo
   • A bit more detail
6 Some code fragments
   • Constraint declarations
   • Chord generation
   • Rhythm generation
   • Note generation
7 Conclusion
   • Conclusion
   • Future Work
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 listing 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
Introduction

APopCALeaPs

Some code fragments

Conclusion

Overview

Demo

A bit more detail

4 Introduction

5 APopCALeaPs
   - Overview
   - Demo
   - A bit more detail

6 Some code fragments

7 Conclusion
Automatic Pop Composer (And Learner of Parameters)
Automatic Music Generation using CHRiSM

Overview
- Demo
- A bit more detail

Introduction
- APopCALeaPs
- Some code fragments
- Conclusion

APopCALeaPs - planned system

Human

GUI

query

CHRiSM program

output
(LilyPond file)

GNU LilyPond

Music
(MIDI file)

Score
(PDF file)

Probability parameters

manual parameter tuning

Learning algorithm (PRISM)

Training set

quality evaluation
(selection)

Human

Jon Sneyers
[demo of the APopCALeaPs system]
Automatic Music Generation using CHRiSM

APopCALeaPs generation process

- APopCALeaPs
- GUI
- CHRiSM program
- Probability parameters
- Output (LilyPond file)

Query process:
- GUI
- Manual parameter tuning
- Probability parameters
- CHRiSM program
- Output (LilyPond file)
APopCALeaPs generation process

- **GUI**: input constraints: key, #measures, #voices, ...
- **chord generation**
- **rhythm generation**
- **melody generation**
- **write output in LilyPond format**
- **output constraints**
- **output** (LilyPond file)

**Manual parameter tuning**

**Probability parameters**

**CHRiSM program**
4 Introduction

5 APopCALeaPs

6 Some code fragments
- Constraint declarations
- Chord generation
- Rhythm generation
- Note generation

7 Conclusion
Constraint declarations (1)

% 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.
% 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.
% 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)

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

Jon Sneyers
Automatic Music Generation using CHiRSM
Rhythm generation (1)

% 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).
% 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,NN,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,NN,NN,NX),
beat(V,M,N,X,D2), beat(V,M,N,X2,D2).
Special cases: instruments “drums” and “chords”

\[
\text{phase(make_notes), beat(drums,M,N,X,D) } \Rightarrow \\
\text{abstract_beat(M,N,X,AB),} \\
\text{msw(drum\_choice(AB),Note),} \\
\text{note(drums,M,N,X,Note).}
\]

\[
\text{phase(make_notes), beat(chords,M,N,X,D),} \\
\text{mchord(M,C), chord_style(Style) } \Rightarrow \\
\text{abstract_beat(M,N,X,AB),} \\
\text{msw(chord\_type(Style,AB),Chord),} \\
\text{chord(C,M,N,X,Chord).}
\]
Note generation (2)

% 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 \rightarrow 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),
ote(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.
% 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+’ ~’).
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)
... 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?