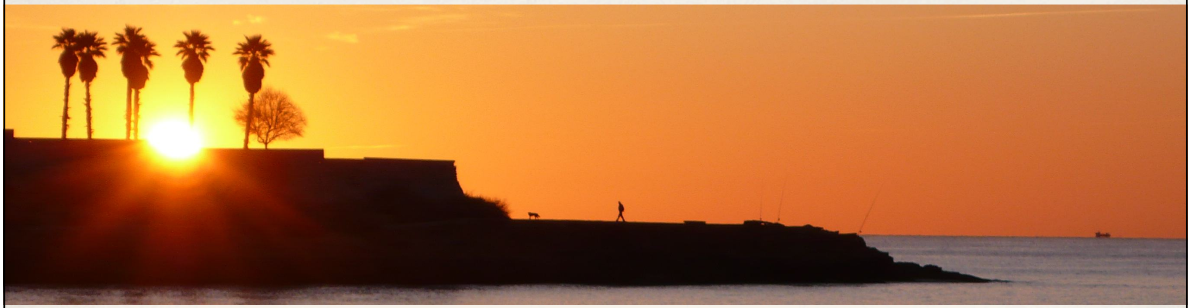


Abduction and language processing with CHR

Henning Christiansen, professor of Computer Science at Roskilde University, Denmark

CHR Summer School – September 2010

My background

- PhD in Computer Science: syntax and semantics of programming languages, 1988
- Later interest in logic programming, as specification+implementation language and an object of study by itself
- Leading to NLP (natural language processing) and automated reasoning, in particular with Constraint Handling Rules
    - with applications in teaching, from hardcore CS students to linguists
- Recent interests include also
    - probabilistic-logic models for bioinformatics
    - formal linguistics, in particular language evolution
- Various: Organizer of several conferences and workshops, coordinator for international student exchanges (Erasmus), a past as Head of CS Section and Study Director

Our principles

- Constraint store as a ***knowledge base***
- CHR rules as "business logic" or "integrity constraints" ≈ rules about knowledge
- Prolog or additional CHR rules as "driver algorithm"

A motivating example . . .

A motivation example (1:3)

Consider the following Prolog program:

```
happy(X):- rich(X).
happy(X):- professor(X), has(X,nice_students).
```

What is it supposed to mean?

Let's try it:

```
| ?- happy(henning).
! Existence error in user:rich/1
! procedure user:rich/1 does not exist
! goal:  user:rich(henning)
```

Another way of saying **no** :(

The problem: Prolog's ***closed world*** assumption

A motivation example (2:3)

Let's try with a little help from CHR:

```
:- use_module(library(chr)).
:- chr_constraint rich/1, professor/1, has/2.

happy(X):- rich(X).
happy(X):- professor(X), has(X,nice_students).
```

Intuition: Make certain predicates *"open world"*.

Let's try it:

```
| ?- happy(henning).
rich(henning) ? ;
professor(henning),
has(henning,nice_students) ? ;
no
```

Looks more like it, but still not perfect . . .

A motivation example (3:3)

Adding a bit of "universal knowledge" in terms of a CHR rule:

```
:- use_module(library(chr)).
:- chr_constraint rich/1, professor/1, has/2.

professor(X), rich(X) ==> fail.
happy(X):- rich(X).
happy(X):- professor(X), has(X,nice_students).
```

Let's try it:

```
| ?- happy(henning), professor(henning).
professor(henning),
has(henning,nice_students) ? ;
no
```

Thus:

- CHR constraints represent ***concrete facts*** about a given world.
- CHR rules represent ***universal knowledge*** valid in any world.

Historical background

- 1998: I found out that CHR existed and used it to implement a powerful automatic reasoning system [Christiansen, 1998]
- 1999: Visiting LMU, Munich, 1999, cooperating with Slim Abdennadher on $CHR^{\vee}$ for abduction [Abdennadher, Christiansen, 2000]
- Around 2000: developing CHR Grammars [Christiansen, TPLP 2005]
- 2002: Visiting Verónica Dahl in Canada; replacing $CHR^{\vee}$ by Prolog+CHR for abductive reasoning ≈ Hyprolog, [Christiansen, Dahl, ICLP 2005]
- 2002 and onwards: different applications
- Since 2005 or before: applied the principle in teaching AI
- 2006-2008: Probabilistic abduction [Christiansen, 2008]

See these and other references in the reference list.

Overview of this course

- **Abductive Reasoning with CHR**
  - Definition, implementation in CHR, applications, esp. for diagnosis
- **Language Analysis 1: With DCGs (= Prolog) plus CHR**
- **Language Analysis 2: CHR Grammars**
- **Probabilistic Abductive Reasoning with CHR**
  - Each branch of computation represented as a CHR constraint
  - Allows for best-first computations

A few remarks before we start

- All example programs available on the website *(TBA)*
  - Tested in SICStus 4; should be compatible with SWI
- No theorems (find them in the references), just programming :)
- Please feel free to ask questions, to disagree even.

Part I

Abductive reasoning with CHR

Abduction????

A term due to C.S.Pierce (1839-1914); the trilogy:

- ***Deduction***
  - Reason "forward" in a sound way from what we know already; finding its logic consequences; i.e., nothing really new
- ***Induction***
  - Creating rules from example, so we can use these rules in new situations
- ***Abduction***
  - Figure out which currently unknown facts that can explain an observation; unsound from logical point of view ;-)

Abduction with CHR

You've seen it already!

```
:- use_module(library(chr)).
:- chr_constraint rich/1, professor/1, has/2.
prof(X), rich(X) ==> fail.
happy(X):- rich(X).
happy(X):- professor(X), has(X,nice_students).
```

```
| ?- happy(henning), professor(henning).
professor(henning),
has(henning,nice_students) ? ;
no
```

In logic programming terms:

Figure out which facts should be added to the program to make a the given goal succeed

Traditional definition of Abductive Logic Programming (ALP)

- An abductive logic program consist of
  - A number of ***predicates***, some of which are called ***abducibles,*** *Abd*
  - A usual ***logic program***, *P*, in which abducibles do not occur in the head of rules
  - A set of ***integrity constraints***, *IC*, which are formulas that must always be true
- An abductive answer to a query *Q* is a set of abducible atoms *Ans* such that
  - $P \cup Ans \models Q$ and $P \cup Ans \models IC$
- (It is also possible to include an answer substitution, but we ignore that)

Translating ALP into Prolog+CHR

Abducible predicates	→	CHR constraints
Integrity constraints	→	CHR rules

Let us inspect our sample program:

```
:- use_module(library(chr)).
:- chr_constraint rich/1, professor/1, has/2.
prof(X), rich(X) ==> fail.
happy(X):- rich(X).
happy(X):- professor(X), has(X,nice_students).
```

Compare with "traditional" ALP

- Usually defined by difficult algorithms and implemented with complicated meta-interpreters; see references to work by Kowalski, Kakas & al, Decker, ...
- Our approach employs existing technology
  - in the most efficient way
  - with no meta-level overhead
  - and we can use all of Prolog and CHR (libraries, all sorts of dirty tricks)
- To my knowledge, far the most efficient implementation of ALP
- The cost? Only very limited use of negation (you can read about that)

Applications of abduction

- Language interpretation

Separate topic;

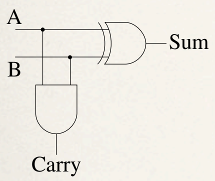
- Diagnosis
- Planning
- View update in databases

Not considered;

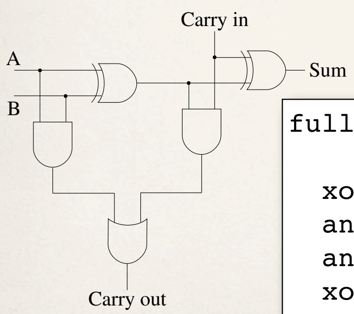
Diagnosis in Prolog+CHR

- Consider a complex system
  - we can only see it from the outside, i.e., observe ***symptoms***
  - we have a ***model*** about how the system works inside
  - we have an idea of possible ***diagnoses***, that can ***explain*** the symptoms
- Examples: a patient, a computer system, a car, . . .
- The problem: Given observed symptoms, suggest diagnoses
- Our example: Fault finding in logical circuits

A model of logical circuits in Prolog




```
halfadder(A, B, Carry, Sum):-
  and(A, B, Carry),
  xor(A, B, Sum).
```




```
fulladder(Carryin, A, B,
                Carryout, Sum):-
  xor(A, B, X),
  and(A, B, Y),
  and(X, Carryin, Z),
  xor(Carryin, X, Sum),
  or(Y, Z, Carryout).
```

```
not(0, 1).
not(1, 0).

and(0, 0, 0).
and(0, 1, 0).
and(1, 0, 0).
and(1, 1, 1).

xor(0, 0, 0).
xor(0, 1, 1).
xor(1, 0, 1).
xor(1, 1, 0).

or(0, 0, 0).
or(0, 1, 1).
or(1, 0, 1).
or(1, 1, 1).
```

Adapt for diagnosis with CHR

Each logical gate is given an *identifier*, so we can distinguish:

```
fulladder(Carryin, A, B,
            Carryout, Sum):-
  xor(A, B, X, g1),
  and(A, B, Y, g2),
  and(X, Carryin, Z, g3),
  xor(Carryin, X, Sum, g4),
  or(Y, Z, Carryout, g5).
```

A gate may be *perfect* or *defect* (ok or ko) for specific inputs

```
:- chr_constraint state/3.
```

```
disturb(0,1).
disturb(1,0).
```

```
and(A,B,X,Id):-
   and(A,B,X),
   state(Id,A+B,ok).
and(A,B,X,Id):-
   and(A,B,Z), disturb(Z,X),
   state(Id,A+B,ko).
or(A,B,X,Id):- . . .
```

Diagnosis may be based on different assumptions

1. ***Periodic faults***, i.e., sometimes a gate works and sometimes it doesn't
2. ***Consistent faults***, i.e., if something is wrong, it is always wrong
3. Consistent faults with ***correct-behavior-produced-in-correct-way***

Diagnosis may be based on different assumptions

1. ***Periodic faults***, i.e., sometimes a gate works and sometimes it doesn't
2. *Consistent faults*, i.e., if something is wrong, it is always wrong
3. Consistent faults with *correct-behavior-produced-in-correct-way*

```
%% No CHR rules needed
```

Let's try it:

```
| ?- fulladder(0,1,1,1,0),
     fulladder(0,1,0,0,1),
     fulladder(0,0,1,0,1),
     fulladder(1,0,1,1,1),
     fulladder(1,1,1,0,0),
     fulladder(0,0,0,0,1).

....
```

A total of 262144 solutions

Diagnosis may be based on different assumptions

1. *Periodic faults*, i.e., sometimes a gate works and sometimes it doesn't
2. ***Consistent faults***, i.e., if something is wrong, it is always wrong
3. Consistent faults with *correct-behavior-produced-in-correct-way*

```
state(Id,Input,S1) \ state(Id,Input,S2) <=> S1=S2.
```

Let's try it:

```
| ?- fulladder(0,1,1,1,0),
     fulladder(0,1,0,0,1),
     fulladder(0,0,1,0,1),
     fulladder(1,0,1,1,1),
     fulladder(1,1,1,0,0),
     fulladder(0,0,0,0,1).

....
```

A total of 72 solutions

Diagnosis may be based on different assumptions

1. *Periodic faults*, i.e., sometimes a gate works and sometimes it doesn't
2. *Consistent faults*, i.e., if something is wrong, it is always wrong
3. Consistent faults with ***correct-behavior-produced-in-correct-way***

```
state(Id,A,S1) \ state(Id,A,S2) <=> S1=S2.
```

Let's try it:

```
| ?- fulladder(0,1,1,1,0),
     fulladder(0,1,0,0,1),
     fulladder(0,0,1,0,1), !,
     fulladder(1,0,1,1,1),
     fulladder(1,1,1,0,0),
     fulladder(0,0,0,0,1).

state(g1,0+0,ko),
state(g3,0+1,ko),
state(g4,1+0,ko),
state(g4,1+1,ko),
state(g5,1+1,ko), .... (rest is ok) ?
```

Only 1 solution!!

Diagnosis may be based on different assumptions: *Summary*

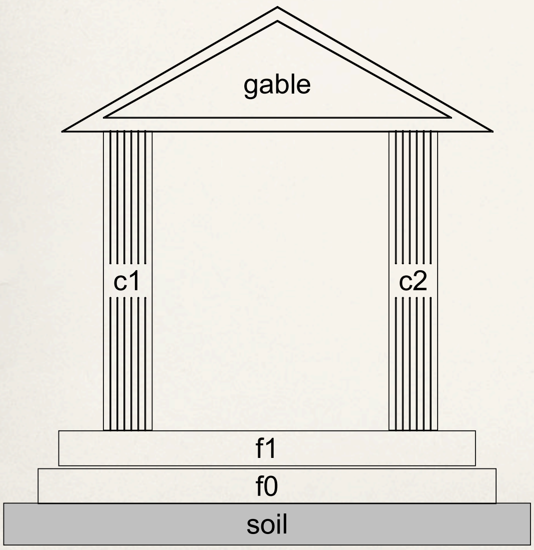
- Formulated in CHR with constraints for ok/not-ok for components
- Three alternative assumptions
  1. periodic faults
  2. consistent faults
  3. consistent faults with correct-behaviour-produced-in-correct way
- In practice, try 3, if it does not work, try 2 – and if that gives too many solutions, try to obtain more observations (i.e., test the device...)
- Problem for practical applications, say medical diagnosis, is the ***lack of priority*** between different diagnoses

Planning as Abduction

- ***Problem:*** Given a number of tasks + restrictions on the order in which they can be done.
- ***Solution:*** An assignment of a time point to each task so the restrictions are obeyed.
- ***In our terms***
  - Abducibles (CHR constraints): Assignment of a time point to a task
  - Integrity constraints (CHR ***rules***): The restrictions
  - The goal (≈ desired observation): *"The work has been done."*

Planning as Abduction, example

Architect's drawing:



CHR rules:

```
mount(P0,Time0), mount(P1,Time1) ==>
  supports(P0,P1), Time0 > Time1
  | fail.
mount(P,Time0), mount(P,Time1) ==>
  Time0 \= Time1
  | fail.
```

Prolog facts:

```
part(gable).
part(c1).
...
supports(soil,f0).
supports(f0,f1).
```

Driver algorithm in Prolog: *next slide*

CHR rules:

```
mount(P0,Time0), mount(P1,Time1) ==>
  supports(P0,P1), Time0 > Time1
  | fail.

mount(P,Time0), mount(P,Time1) ==>
  Time0 \= Time1
  | fail.
```

Prolog facts:

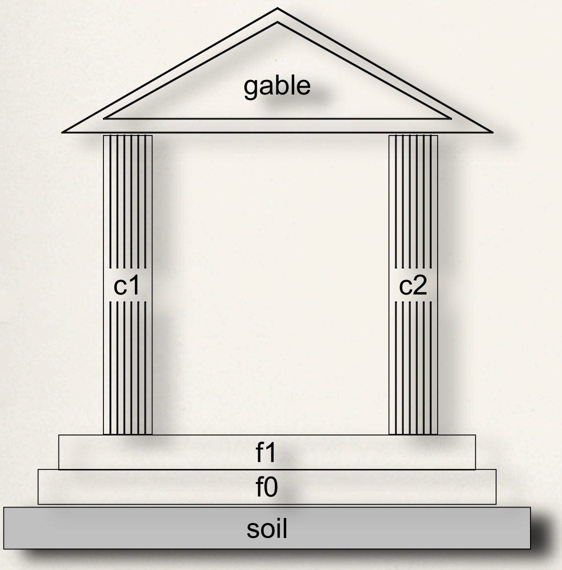
```
part(gable).
part(c1).
...
supports(soil,f0).
supports(f0,f1).
```

Driver algorithm in Prolog:

```
built:-  mount(soil,0), build(1).

build(6):- !.

build(Time):-
   part(P),
   mount(P,Time),
   Time1 is Time+1,
   build(Time1).
```

```
| ?- build.
mount(gable,5),
mount(c2,4),
mount(c1,3),
mount(f1,2),
mount(f0,1),
mount(soil,0) ? ;

mount(gable,5),
mount(c1,4),
mount(c2,3),
mount(f1,2),
mount(f0,1),
mount(soil,0) ? ;

no
```

Wanna see an animation of the first solution?




```
| ?- build.
mount(gable,5),
mount(c2,4),
mount(c1,3),
mount(f1,2),
mount(f0,1),
mount(soil,0) ?

mount(gable,5),
mount(c1,4),
mount(c2,3),
mount(f1,2),
mount(f0,1),
mount(soil,0) ? ;

no
```

More on planning

- With the same technique, we can extend with
  - ***Duration***, e.g., it takes 8 hours to mount a column
  - ***Resources***, e.g., to mount a column, we need 1 crane and 12 workers
  - ***Restrictions***+= At any time, the resources in use must not exceed the maximum available (say, 2 cranes and 30 workers)
- ***Your exercise (voluntary!):*** Extend the example and implement the scheme above
- ***Your next exercise (difficult & voluntary):*** Extend your program so it tries to find a solution that minimizes the no. of unoccupied workers — or, alternatively, the solution that finishes the building as early as possible.

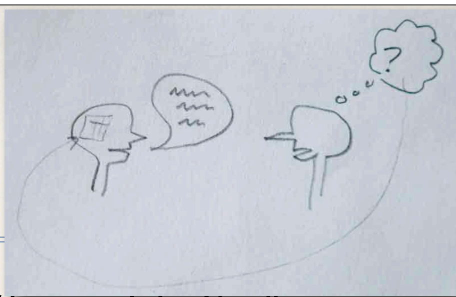
End of Part I

Abductive reasoning with CHR

Part II

Language analysis with Prolog and CHR

Overall principles



- My favourite metaphor: ***"Interpretation as abduction"***
  - Jerry R. Hobbs, Mark E. Stickel, Douglas E. Appelt, Paul A. Martin: Interpretation as Abduction. Artif. Intell. 63(1-2): 69-142 (1993)
  - Also Charniac, McDermott (1985), Gabbay & al (1997), Christiansen (2003)
- We use Prolog's Definite Clause Grammars (DCGs) extended with CHR
- Resulting method:
  - Integrates semantic and pragmatic analysis (in contrast to tradition methods)
  - A great experimental tool for students and researcher in linguistics; easy to approach and "advanced" analyses can be specified in very short time.

A short historical note

- Basic idea comes from CHR Grammars (Christiansen, 2001-2005), that we will look at later in the course
- Idea of using DCGs emerged through joint work with Verónica Dahl, 2002 and onwards....
  - Lead to the ***Hyprolog*** system (Christiansen, Dahl, ICLP, 2005)
  - adds a thing layer of syntactic sugar upon Prolog+CHR that supports abduction
  - and so-called assumptions, which another kind of tool (related to abduction, though), coming from Verónica Dahl's earlier work.
- Here we show things expressed directly in Prolog(DCG)+CHR

Overview

- Recall Definite Clause Grammars
- Adding semantics/pragmatics: Using CHR as knowledge base as we have seen already
- Examples
- Hyprolog and Assumptions
  - Basic idea
  - Examples
  - Briefly about implementation techniques
- A realistic application: Mapping Use Cases to UML (sketch)

Definite Clau

```
s(S0,S3):- np(S0,S1,N), v(S1,S2,N), v(S2,S3).
....
v([sees|S0],S0,sing).
```

- Syntactic sugar on top of Prolog
- System adds difference lists "behind the curtain"
- In Prolog from its very beginning
- Very popular for teaching, prototyping, and some realistic applications
- Easy to add features and "constraints"

```
s --> np(N), v(N), np(_).

s --> np(N), is(N), [at], np(_).

np(N) --> n(N).

v(sing)--> [sees].
v(plur)--> [see].
is(sing)--> [is].
is(plus)--> [are].

n(sing) --> [peter].
n(sing) --> [mary].
n(sing) --> [jane].
n(sing) --> [the,chr,summer,school].
n(sing) --> [hennings,course].
n(sing) --> [vacation].

n(plur) --> n(sing), [and], n(_).
```


Adding semantics/pragmatics

- Traditionally:
  - ***"Semantics"*** = context-independent (lambda) terms
  - ***"Pragmatics"*** = relating "Semantics" to context, e.g., mapping variables to (identifiers of) "real worlds"
- The present approach ***blurs this distinction***, which suits much better my intuition about how humans process language
- You may see this in the examples

A DGC with CHR for sem/pragm

First version: Only noting facts

```
:- chr_constraint at/2, see/2.

story --> [] ; s, ['.'], story.

s --> np(X), [sees], np(Y),
      {see(X,Y)}.

s --> np(X), [is,at], np(E),
      {at(E,X)}.

s --> np(X), [is,on,vacation],
      {at(vacation,X)}.

np(peter)     --> [peter].
np(mary)      --> [mary].
np(jane)      --> [jane].

np(chr_summer_school)
        --> [the,chr,summer,school].

np(hennings_course)
        --> [hennings,course].

np(vacation) --> [vacation].
```

```
:- phrase(story,
      [peter,sees,mary,'.',
       peter,sees,jane,'.',
       peter,is,at,the,
             chr,summer,school,'.',
       mary,is,at,hennings,course, '.',
       jane,is,on,vacation,'.']).

at(vacation,jane),
at(hennings_course,mary),
at(chr_summer_school,peter),
see(peter,jane),
see(peter,mary) ?
```


2nd version: Adding world knowledge

```
:- chr_constraint at/2, in/2, see/2, skypes/2.

at(chr_summer_school,X) ==> in(leuven,X).

in(Loc1,X) \ in(Loc2,X) <=> Loc1=Loc2.

at(hennings_course,X)   ==> at(chr_summer_school,X).

at(vacation,X)          ==> in(Loc,X), diff(Loc,leuven).

see(X,Y) ==> true |
    (in(L,X), in(L,Y)
     ; in(Lx,X), in(Ly,Y), diff(Lx,Ly), skypes(X,Y

diff(...) <=> ... . % Homemade version of dif/1 for nicer outpu

% Grammar rules: Exactly the same as before
```

```
| :- phrase(story,
      [peter,sees,mary,'.',
       peter,sees,jane,'.',
       peter,is,at,the,
             chr,summer,school,'.',
       mary,is,at,hennings,course, '.',
       jane,is,on,vacation,'.']).
```

```
course,'.'
```

```
at(vacation,jane),
at(chr_summer_school,mary),
at(hennings_course,mary),
at(chr_summer_school,peter),
in(_A,jane),
in(leuven,mary),
in(leuven,peter),
see(peter,jane),
see(peter,mary),
skypes(peter,jane),
diff(leuven,_A) ?
```

HYPROLOG and Assumptions

- Assumptions developed by [Dahl & al., 1997; Christiansen, Dahl, 2004, ...]
- Similar to abduction but with explicit creation and application + simplistic scoping
- Can be implemented in CHR more or less the same way as abduction; you may also take this as a lesson in implementing knowledge handling with CHR
- Included in the HYPROLOG system

+*A*	Assert linear assumption *A* for subsequent proof steps. Linear means "can be used once".
**A*	Assert intuitionistic assumption *A* for subsequent proof steps. Intuitionistic means "can be used any number of times".
-*A*	Expectation: consume/apply existing intuitionistic assumption in the state which unifies with *A*.
=+*A*, =**A*, =-*A*	Timeless versions of the above, meaning that order of assertion of assumptions and their application or consumption can be arbitrary.

Example of Assumptions in DCG

Semantic-pragmatic analysis with pronoun resolution/HYPROLOG syntax

```
assumptions acting/1.
abducibles fact/1.

sentence --> np(A,_), verb(V), np(B,_),
  {fact(A,V,B)}.

sentences --> [] ; sentence(S1),sentences(S2).

np(X,Gender) --> name(X,Gender),
  {*acting(X,Gender)}.

name(peter,masc) --> [peter].

...

np(X,Gender) --> {-acting(X,Gender)},
  pronoun(Gender).

pronoun(fem) --> [her].

...

verb(like) --> [likes].
```

"Peter likes Mary. She likes him"

```
*acting(peter,masc)
*acting(mary,fem)
-acting(X,fem)
```
leads to `X=mary`
```
-acting(X, masc)
```
leads to `X=peter`

```
fact(peter,like,mary)
fact(mary, like,peter)
```

Implementing Assumptions for DCGs in CHR

Example: Linear assumptions and expectations

- We want to be able to backtrack through alternative matches
- Incompatible (at first glance) with CHR's philosophy

```
:- chr_constraint (-)/1, (+)/1, assump_list/1.

+A, assump_list(L) <=> assump_list([A|L]).

+A <=> assump_list([A]).

-E, assump_list(L) <=> member(A,L,LRest), assump_list(LRest), A=E.
```

This is just one way on implementing Assumptions;
it is more efficient to maintain one `assump_list` per Assumption symbol

What is HYPROLOG, btw.?

- A system that adds a thin layer of syntactic sugar on top of Prolog+CHR
  - Special syntax for declaring abducibles (as you have seen)
  - Utilities and options for abductive reasoning (not shown here)
  - Assumptions implemented as you have just seen
- Implementation principles interesting if you want to do such things ...
  - Using same facilities as DCGs and CHR: `term_expansion`
  - Operator declarations in Prolog are fine and useful, but we need also:
  - When reading in a term from a Prolog source file, the system checks if there is a `term_expansion` clause that matches that term ...

Example, next slide

(A small parenthesis on Prolog programming)

- Implementing a “where” notation in Prolog which really surprises me that they did not put in from the beginning; included in HYPROLOG and CHR Grammars :)

Instead of

```
p(X):- r(X,Y), z(Y,4), q(X,17).
```

we would like to write

```
p(X):- Test, q(X, LastArg)
    where Test = (r(X,Y), z(Y,4)),
          LastArg = 17.
```

Implementation

```
:- op(1200,yfx, where).

term_expansion((Rule where Replace), Result):-
    Replace -> term_expansion(Rule, Result)
            ;  write('Error in "where" part'),
               abort.
```

(Recursive call to `term_expansion` important)

A realistic example: Extracting UML diagrams from Use Cases

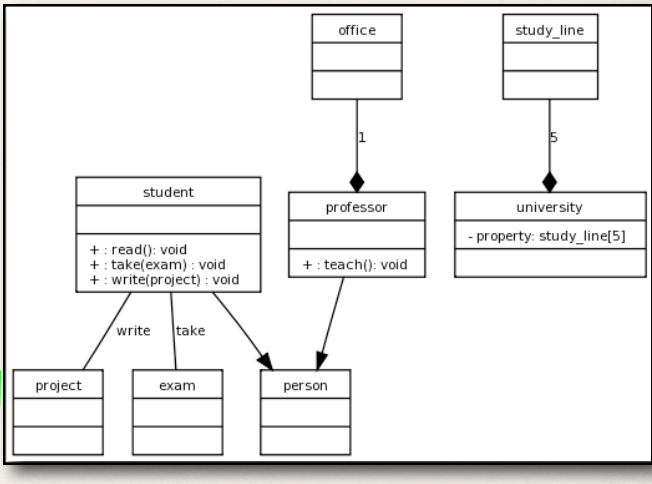
- Based on 4 week project work with two students [Christiansen, Have, Tveitane, 2007 a+b]
- Only a brief sketch; here using full power of CHR without caring about formal details ;-)
- Use cases?? In the OOA/OOP tradition, small *stories* about the world which the system to be developed will fit it.
- According to OOA principles, UML diagrams describing classes and their property, etc., are produced manually from use cases...
- But why not do it automatically, when we have a tool such as Prolog +CHR which is perfectly suited for semantic/pragmatic analysis

Example of input and output

From uses cases:

- *The professor teaches. A student reads, writes projects and takes exams. Henning is a professor. He has an office. The university has five study lines. Students and professors are persons.*

... extract info and produce



Examples of CHR rules for knowledge extraction (1:2)

Merging cardinalities, e.g.:

property(man, dog:1)
property(man, dog:5)

↓

property(man, dog:(1..5))

property(man, dog:(0..2))
property(man, dog:(1..n))

↓

property(man, dog:(0..n))

```
property(C,P:N), property(C,P:M) <=>
   count(N), count(M), N=<M
| property(C,P:(N..M)).
```

```
property(C,P:(N1..M1)),property(C,P:(N2..M2)) <=>
   min(N1,N2,N), max(M1,M2,M),
   property(C,P:(N..M)).
```

(NB: “n” is a special symbol meaning “many”)

Examples of CHR rules for knowledge extraction (2:2)

Pronoun resolution, e.g.,

Jack and John are teachers. Jack teaches music. John teaches computer science. Mary is a student. ***He*** *has many students.*

Our heuristics: Take most recent referent that matches gender and when no ambiguity arises; in case of ambiguity, we call it an error

~~*Jack and John are teachers. He ....*~~

```
sentence_no(Now), referent(No,G,Id,T) \  expect_referent(No,G,X) <=>
   T < Now,  there is no other relevant referent with Timestamp  > T
|
   if there is another relevant referent with  Timestamp  =  T  then
      X = errorcode(ambiguous)
   else
      X = Id.
```


Summary: Language analysis with DGC+CHR

- Natural and straightforward integration of semantic/pragmatic analysis with parsing
- 10^6 times easier for this purpose than any other, known tools
- DCGs (i.e., Prolog) provide parsing plus auxiliary predicates
- CHR constraint store as knowledge base; CHR rules for world knowledge
- We showed
  - Direct use of DCG+Prolog
  - HYPROLOG which provided syntactic sugar, Assumptions and various auxiliaries
  - A realistic example with pronoun resolution and semantic reasoning

End of part II

Language analysis with Prolog and CHR

Part III

CHR Grammars

CHR Grammars, background

- Around 2000, I noticed that it was easy to write bottom-op parsers with CHR
- Experiments showed that there was much more power in this principles than expected:
  - very flexible context-dependent rules, gaps, parallel matching, ...
  - interesting treatment of ambiguity
  - having parsing to depend on "semantics", and a lot of other stuff
- 2002: CHR Grammar system released; *only SICStus 3; beta versions for SICStus 4 and SWI exist; will be released soon (especially if you write to me ;-)*
- Main publication 2005 [JLP]
- Applications: The full power of CHR Grammars still needs to be discovered

CHR Grammars, overview

- Bottom-up parsing with CHR, our principle
- A grammar notation and its translation into CHR
- What we can do in CHR Grammars, derived from the translation into CHR
  - We have squeezed as much power as possible out of CHR without caring whether it is useful *(our preferred design methodology ;-)*
- Example: a biological application

Bottom-up parsing with CHR

Encode the string as a set of constraints with ***word boundaries***

"Peter likes Mary"

```
token(0,1,peter),token(1,2,likes),token(2,3,mary).
```

A bottom-parser that checks word / phrase boundaries

```
:- chr_constraint np/2, verb/2,
   sentence/2, token/3.

token(N0,N1,peter) ==> np(N0,N1).
token(N0,N1,mary)  ==> np(N0,N1).
token(N0,N1,likes) ==> verb(N0,N1).

np(N0,N1), verb(N1,N2), np(N2,N3)
                 ==> sentence(N0,N3).
```

```
?-  ... .
np(0,1),
verb(1,2),
np(2,3),
sentence(0,3),
token(0,1,peter),
token(1,2,likes),
token(2,3,mary) ?
```


A grammar notation upon CHR

Why write this?

```
:- chr_constraint np/2, verb/2,
   sentence/2, token/3.
token(N0,N1,peter) ==> np(N0,N1).
token(N0,N1,mary)  ==> np(N0,N1).
token(N0,N1,likes) ==> verb(N0,N1).
np(N0,N1), verb(N1,N2), np(N2,N3)
                 ==> sentence(N0,N3).
```

```
?- token(0,1,peter),
   token(1,2,likes),
   token(2,3,mary).
```

When we would like to write this:

```
:- grammar_symbol np/0, verb/0,
                  sentence/0.

[peter] ::> np.
[mary] ::> np.
[likes] ::> verb.

np, verb, np ::> sentence.

end_of_CHRG_source.
```

```
?- parse([peter,likes,mary]).
```

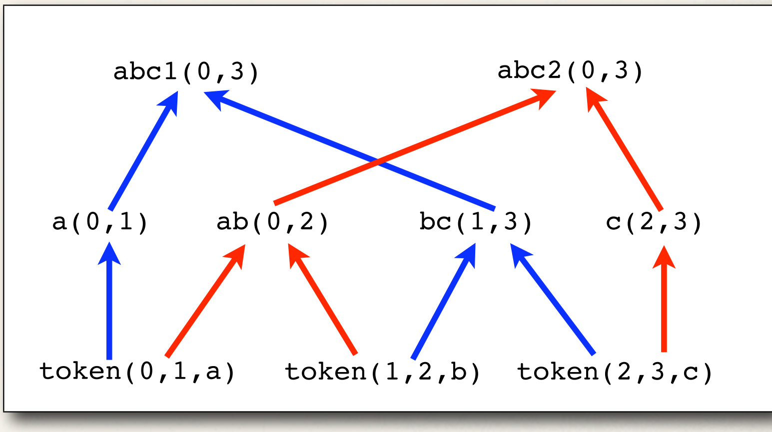
The CHR compiler

compile-on-load using `term_expansion`

Inherent handling of ambiguity

```
[a]   ::> a.
[b,c] ::> bc.
[a,b] ::> ab.
[c]   ::> c.
a, bc ::> abc1.
ab, c ::> abc2.
```

```
| ?- parse([a,b,c])
```



- I.e., all possible parses are run "in parallel"
- You can limit this by, e.g., simplification rules;
  - in the example, you would end up with {`abc1(0,3)`, `c(2,3)`}
- Thus the semantics ***very*** procedural! (good or bad?)

What else can we put in? (1:5)

- `::>` translates into `==>`
- `<:>` translates into `<=>`
- Order independent syntax for simpagations

```
!a, b, !c <:> ac.
```

translated into

```
b(N1,N2) \ a(N0,N1), c(N2,N3) <=> ac(N0,N3).
```

What else can we put in? (2:5)

Gaps in the head

```
[blip], 7...10, [blop] ::> blipblop
```

- translated into

```
a(N0,N1),b(N2,N3) ::>
   N2-N1 >= 7, N2-N1 =< 10
| ab(N0,N3).
```

- This may be relevant for biologic applications such as RNA folding

What else can we put in? (3:5)

Left and right context

- *left-context* `-\` *core-to-be-reduced* `/-` *right-context* `::>` ....
- For example

```
c1, ..., c2 -\ c3, c4 /- ..., c5 <:> c34.
```

- translated into

```
c1(_,N1),c2(N2,N3),c3(N3,N4),c4(N4,N5),
c5(N6,_)
<=> N1=<N2, N5=<N6 | c34(N3,N5).
```

What else can we put in? (4:5)

Parallel matching

- *one-reading-of-the-text* `$$` *another-reading-of-the-text* `::>` ....
- For example: `a $$ b <:> c.`
- translates into: `a(N0,N1), a(N0,N1) <=> c(N0,N1).`
- And: `a, 5...12 $$ b, c <:> d`
- translates into:

```
a(N0,N1), b(N0,N11), c(N11,N2)
<=> N1-N2 >= 5, N1-N2 =< 12 | d(N0,N2)
```

- ***Applications?*** *I forgot why I included it, but it is smart, isn't it?*

What else can we put in? (5:5)

- Assumptions as we have seen
- Further equipment for abduction (see paper on CHRG)
- All sorts of utilities and options (see online User's Guide)
- Extra-grammatical constraints in the head and body of rules (...)

Example: Simplification and context for disambiguation

An abstract and ~~highly ambiguous~~ grammar:

```
e, [+], e /- (['+'];[')'];[eof])   <:> e.
e, [*], e /- ([*];[+];[')'];[eof]) <:> e.
e, [^], e /- [X] <:> X \= ^ | e.
['('], e, [')'] <:> e.
[N] <:> integer(N) | e.
```

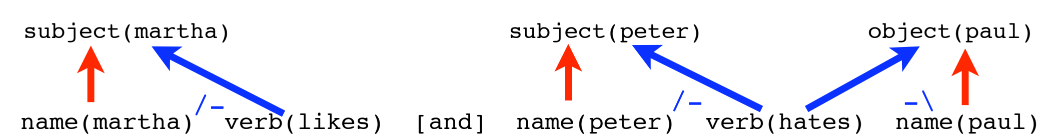
Here we used LR(1) items as right context to disambiguate...

just one special case of what we can do

Example: Context used for tagger-like rules

Classify np's according to position of the verb

```
            name(A) /- verb(_) <:>   subject(A).
verb(_) -\ name(A)             <:>    object(A).
name(A),    [and], subject(B)  <:> subject(A+B).
object(A), [and], name(B)      <:>  object(A+B).
```



Martha likes and Peter hates Paul

A little voluntary exercise

- Write the remaining rules for a grammar that may parse the entire phrase given in the previous slide.
  - to make certain terminal symbols into nonterminals such as `name(mary)`
  - to make certain terminal symbols into nonterminals `verb(likes)`
  - to parse complete sentences, i.e., that include explicit object.
  - to parse incomplete sentences that has implicit object, given by another sentence after "`and`".
- Next, add at attribute to each sentence of the form `fact(`*subject*`,`*verb*`,`*object*`)` and modify your grammar so that it generates the correct "meaning" for each sentence, also the incomplete ones.
  - For example, the first incomplete sentence in the previous example should generate the "meaning" `fact(martha,like,paul)`.
- Extend the grammar with whatever you find interesting.

Biological example

- RNA folding — to be developed over summer

Summary of CHRGs

- A powerful language specification language
- A powerful language processing system
- Exemplifies how you can use CHR to implement fairly advanced, knowledge-based systems
- A compile-on-load implementation technique, you can use for other purposes
- The power of CHRGs has not been explored fully; biological applications are under consideration

End of part III

CHR Grammars

Part IV

Probabilistic abduction with best-first search

Probabilistic Abduction with CHR

Our approach

- Use constraint store to hold a bunch of processes; CHR rules perform derivation steps
- Each such process holds its "own constraint store"
- This principle can be used for other purposes!!!
- Recall abduction: To figure out which facts that are missing in order to have a given goal to succeed; integrity constraints to avoid nonsense

This presentation

- Shows the ***propositional*** case only and by example, but ...
- See [Christiansen, 2008 - LNCS 5388] for all details with variables and non-ground abducibles.
  - No system available, but you can copy-paste from the paper

Towards probabilistic abduction: (1:4) Processes as CHR constraints

This is a Prolog program:

```
p:- q.
p:- r.
q:- s.
q.
s.
```

This is its translation into an all-solutions CHR version:

```
:- chr_constraint process/1.

process([p|More]) <=>
    process([q|More]),
    process([r|More]).

process([q|More]) <=>
    process([s|More]),
    process(More).

process([s|More]) <=> process(More).
```

Trying it:

```
| ?- process([p]).
process([r]) ;
process([]) ;
process([]) ;
no
```

≈ a failed branch (i.e., could not continue)

≈ successful branches

Towards probabilistic abduction: (2:4) Adding abducibles

This is an *abductive* logic program:

```
abducible(a).
abducible(b).
abducible(c).
p:- a,q.
p:- r.
q:- b,s.
q:- c.
s.
```

This is its translation into an all-solutions CHR version:

```
:- chr_constraint process/2.

process([p|More],Abd) <=>
    process([a,q|More],Abd),
    process([r|More],Abd).

process([q|More],Abd) <=>
    process([b,s|More],Abd),
    process([c|More],Abd).

process([s|More],Abd) <=> process(More,Abd).

process([A|More],Abd) <=>
    abducible(A),
  |
    (member(A,Abd) -> process(More,Abd)
     ; process(More,[A|Abd])).
```

Trying it:

```
| ?- process([p],[]).
process([r],[]) ;
process([],[c,a]) ;
process([],[b,a]) ;
no
```

≈ a failed branch (i.e., could not continue)

≈ successful branches

Towards probabilistic abduction: (3:4) Adding integrity constraints

This is an *abductive* logic program:

```
abducible(a).
abducible(b).
abducible(c).
p:- .... as before

:- a,c.
```

This is its translation into an all-solutions CHR version:

```
:- chr_constraint process/2.
process([p|More],Abd) <=>
    .... as before
process([A|More],Abd) <=>
     abducible(A), \+ violate([A|Abd])
   |
     (member(A,Abd) -> process(More,Abd)
      ; process(More,[A|Abd])).
violate(Abd):- member(a,Abd), member(c,Abd).
```

Trying it:

```
| ?- process([p],[]).
process([r],[]),
process([c],[a]),
process([],[b,a]) ? ;
no
```

≈ failed branches (i.e., could not continue)

≈ a successful branch

Towards probabilistic abduction: (4:4) Finally, probabilities

This is a *probabilistic abductive* logic program:

```
abducible(a, 0.2).
abducible(b, 0.7).
abducible(c, 0.9).
p:- .... as before
 ...
:- a,c.
```

This is its translation into an all-solutions CHR version:

```
:- chr_constraint process/3.
process([p|More],Abd,Prob) <=>
   process([a,q|More],Abd,Prob),
   process([r|More],Abd,Prob).
....
process([A|More],Abd,Prob) <=>
     abducible(A,P), \+ violate([A|Abd])
   |
     (member(A,Abd) -> process(More,Abd,Prob)
      ; Prob1 is Prob*P,
        process(More,[A|Abd],Prob1)).
violate(Abd):- member(a,Abd), member(c,Abd).
```

Trying it:

```
| ?- process([p],[],1).
process([r],[],1),
process([c],[a],0.2),
process([],[b,a],0.14) ? ;
no
```

≈ failed branches (i.e., could not continue)

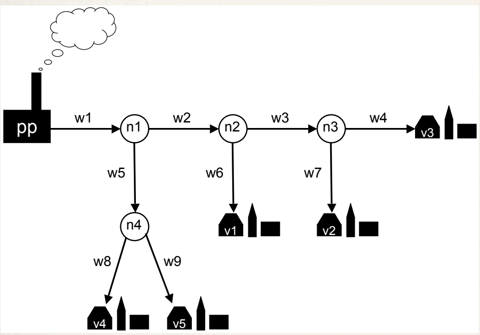
≈ a successful branch

Extending with best-first search

- Add a little bit of control encoding
  - only a `process(`G`,`Abd`,`P`)` with a highest P can be expanded
  - when the first `process([],`Abd`,`P`)` is encountered, *Abd* and P are printed and the user asked if he / she wants more solution
- Advantages
  - more efficient: executes until first and guaranteed best solution is found
  - can work even with programs that would otherwise loop
- Find details in the paper [Christiansen, 2008 - LNCS 5388]

Example: Diagnosis with probabilities

A power supply network:




```
abducible(up(_),   0.9).
abducible(down(_), 0.1).

:- up(X),down(X).
```

```
link(w1, pp, n1). ...

haspower(pp):- up(pp).
haspower(N2):-
      link(W,N1,N2), up(W), haspower(N1).

hasnopower(pp):- down(pp).
hasnopower(N2):- link(W,_,N2), down(W).
hasnopower(N2):-
      edge(_,N1,N2), hasnopower(N1).
```

Trying it:

```
?- haspower(v5), nohaspower(v1).
to be tested and included later
```

A voluntary project work

- Write a compile-on-load implementation of probabilistic logic programs using `term_expansion`.
- If you decide to do this, write to me!!

Summary of our approach to Probabil R

Non-ground abducibles?

```
| ?- happy(peter).
married(peter,X), blond(X), rich(X).
Probability = 0.2689
```

- Probabilities (or other
- Abductive logic programs with probabilities are powerful modeling tools for systems to be diagnosed
- Comparison:

	Our approach	**Poole** [1993,200]	**Sato & al** [2001, ...] PRISM
Non-ground abducibles	*yes*	*no*	*yes*
Integrity constraints	*yes*	*no*	*no*
Other features			*Very powerful machine learning techniques and lots of facilities*

- To be done:
  - an efficient priority queue for selecting currently best
  - express and utilize that, e.g., `down(X)`,`up(X)` are each other's negation, e.g.,

```
[down(w1), down(w2)] + [down(w1), up(w2)]  =  [down(w2)]
```

End of part IV

Probabilistic abduction with best-first search

Summary of the course

- CHR is for more than numbers, inequalities and stuff like that
- CHR is a ***powerful knowledge representation & manipulation language***
- I have showed methods for abductive reasoning and language processing, that are
  - executed directly by the underlying CHR and Prolog systems
  - thus efficient for the right kind of problems
- I have intended that, after this course and a bit of reading, ***you can***
  - use the methods as described directly
  - invent your own ways to work with knowledge and experiment with in Prolog+CHR

The End