Logic for declarative problem-solving and its applications

Gerhard Friedrich & Martin Gebser

Universität Klagenfurt

Austria

Structure

Part 1

- Why declarative problem solving?
- Preliminaries / brush-ups
- Introduction to answer set programming:
  - Logic-based
  - Expressive
  - Efficient reasoning for practical problems

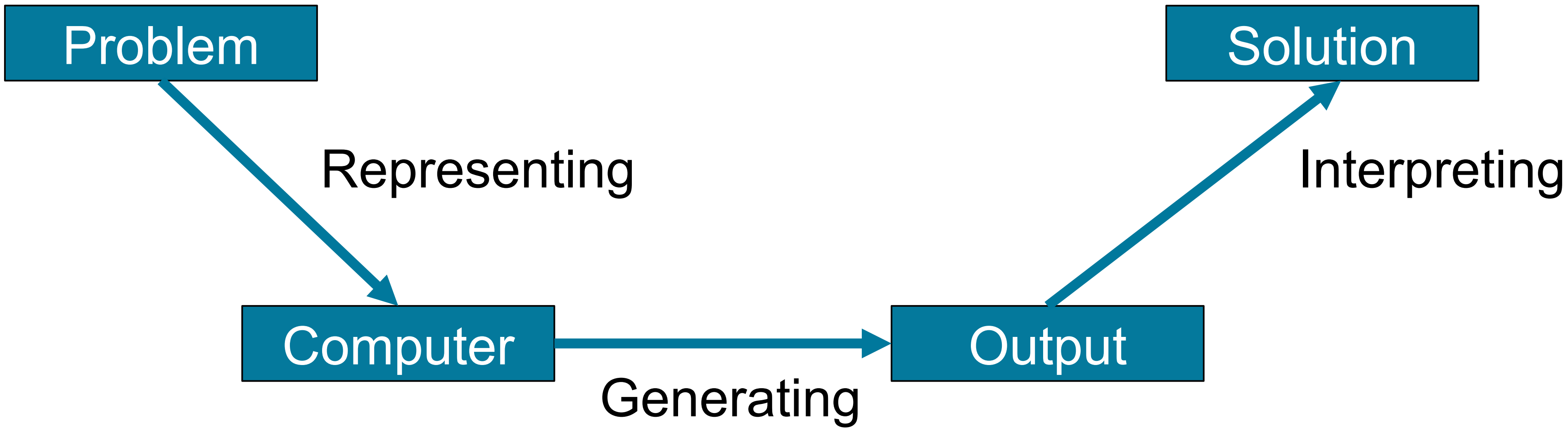
Part 2

Answer set programming for optimization and its application

- Traveling salesperson problem
- Linux package configuration
- Conclusion

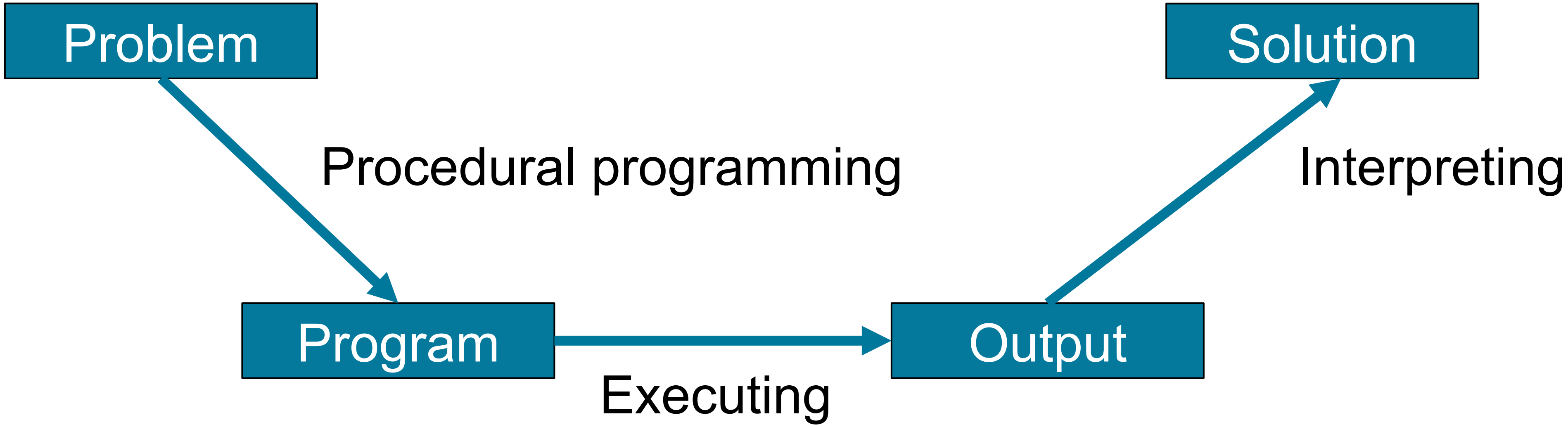
Why declarative problem-solving?

- Solving problems with a computer



Why declarative problem-solving?

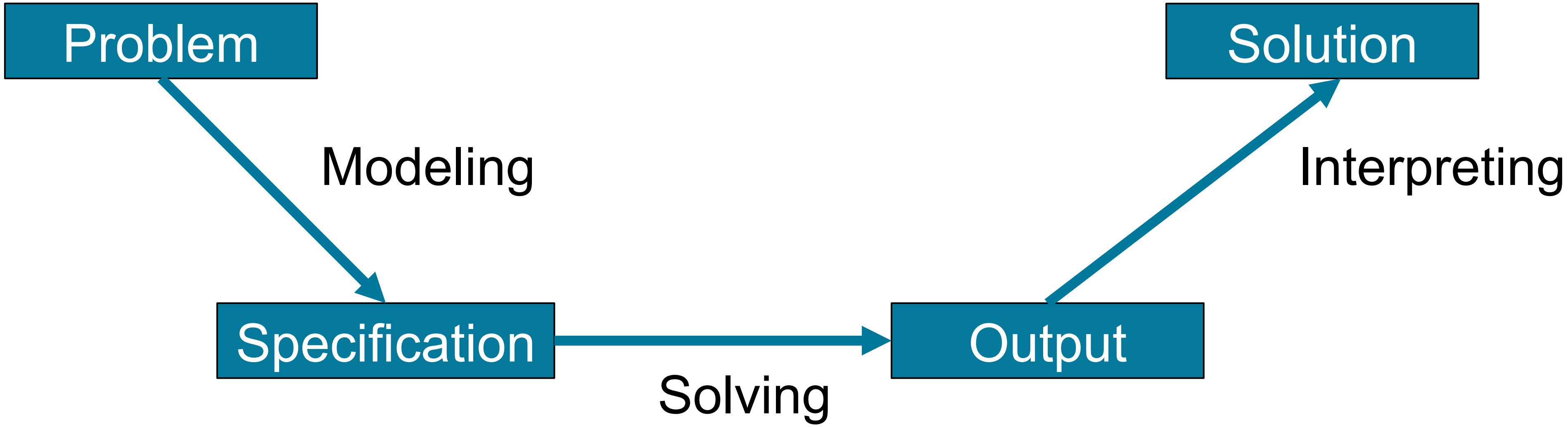
- Solving problems with a computer by **programming**



- A procedural program specifies "how to generate solutions"
- Solutions are generated by executing a program

Why declarative problem-solving?

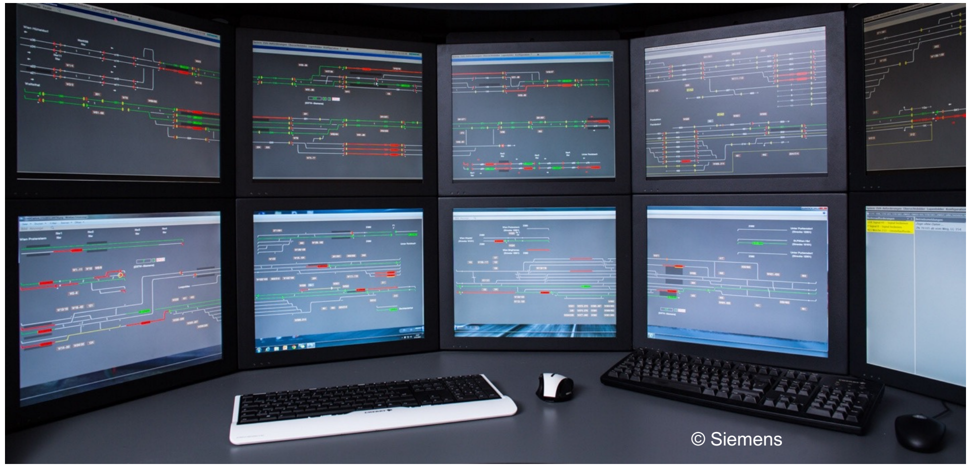
- Solving problems with a computer by **a declarative specification**

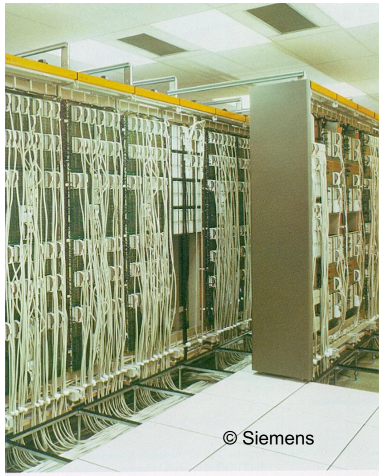


- A declarative program specifies "what are the solutions"
- Solutions are generated by a problem solver

Why declarative problem-solving?

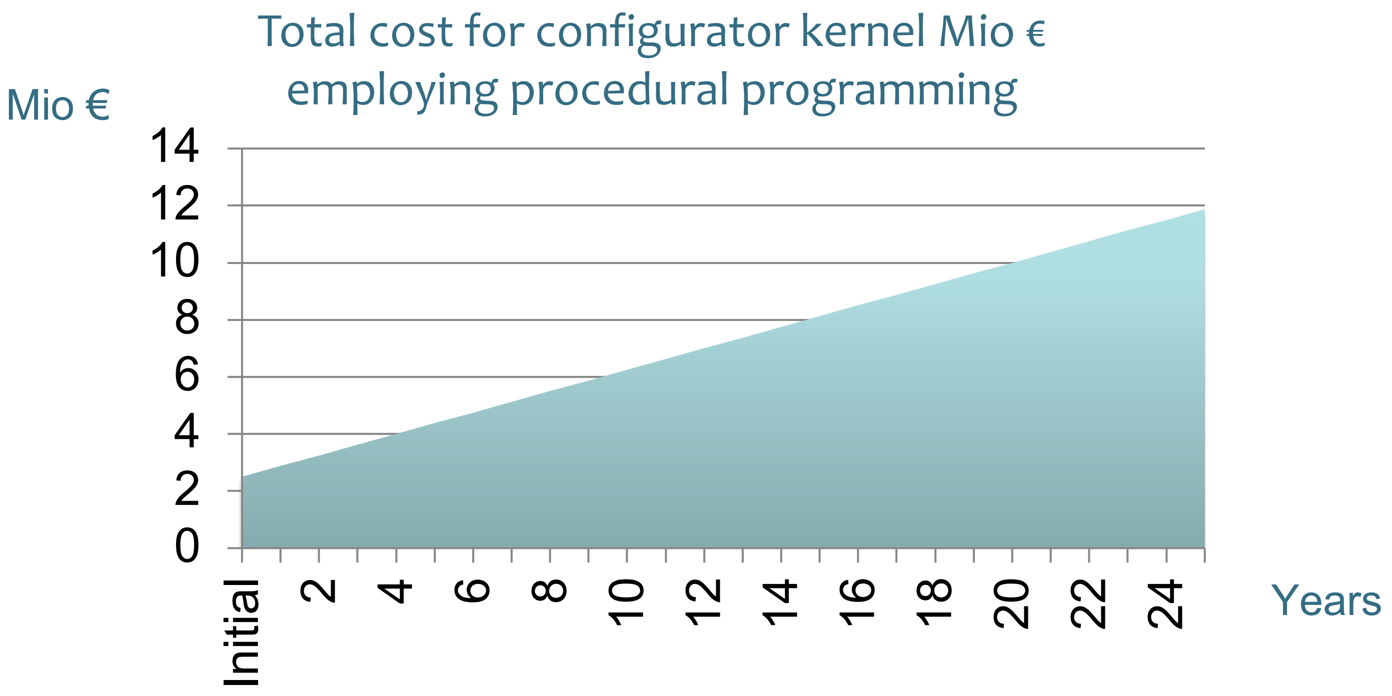
- E.g., configuration of technical system aka automated engineering





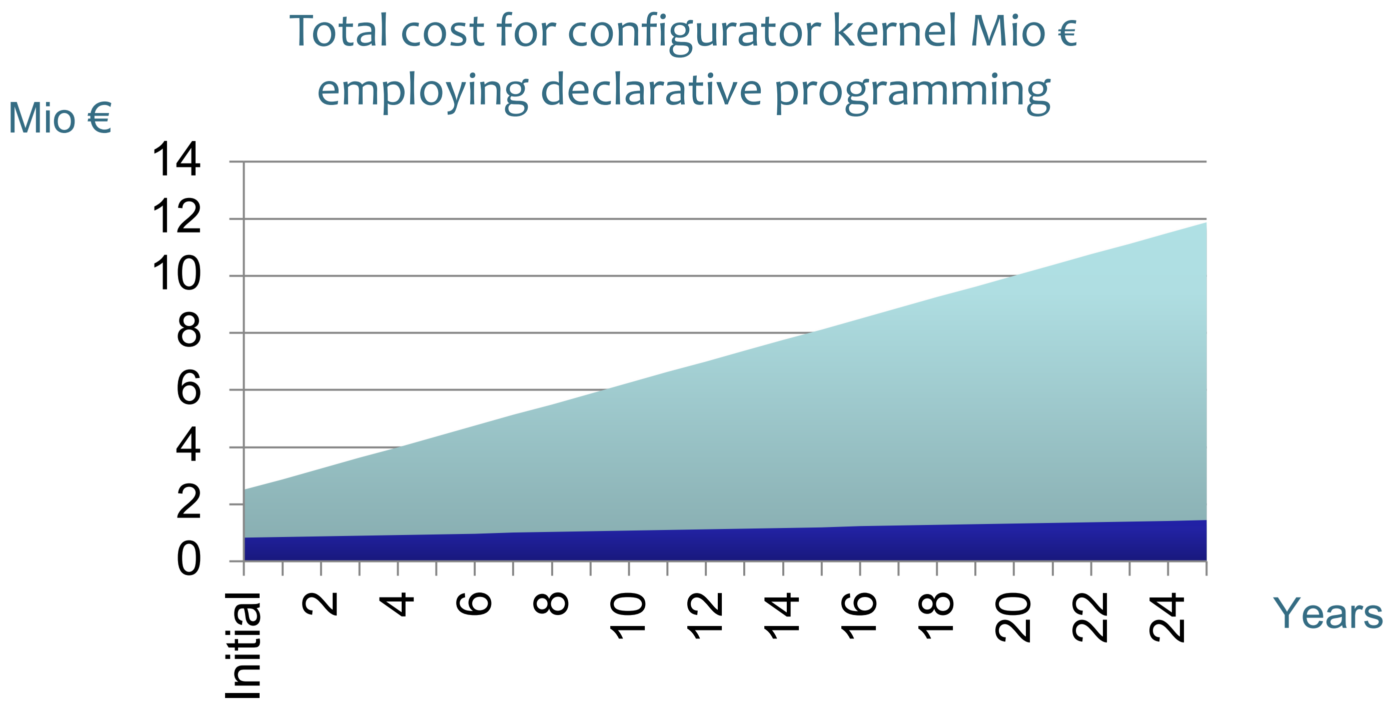
Why declarative problem-solving?

- Some insights of applying declarative problem-solving:



Maintenance cost: approx. 15% per year of initial development cost

Why declarative problem-solving?



- Reduction of initial development cost by 66%
- Reduction of yearly maintenance cost by 80%
- Productivity increase by 300% (no additional staff)
- ROI in 1 year for the telecommunication domain
- Enhanced user interaction:
  - explanations,
  - incremental configuration,
  - repair …

Wanted

- General-purpose framework for modeling and solving problems

Design requirements

- “Expressive” to succinctly formulate maintainable specifications
- Optimization
- Rules and constraints
- Aggregation
- Dealing with the absence of information

Proposal

- Answer set programming (ASP) paradigm

Preliminaries / brush-ups

Propositional logic (syntax):

- Finite set of atoms (propositional symbols): e.g., {a_1, a_2, b, ..., name_of(4711,joe), age_of(4711,20), ...}
- Logic operators: "$\lor$", "$\land$", "$\rightarrow$", "$\neg$", ...
- Literals are atoms and negated atoms: e.g., a, $\neg a$
- A clause is a set of literals connected by "$\lor$": e.g., $(\neg b \lor a_1 \lor c)$
- Propositional theory: a finite set of clauses connected by "$\land$": e.g., $(\neg a_1 \lor a_2) \land b \land (\neg b \lor a_1 \lor c)$

Propositional logic (semantics)

- Let the base B of a propositional theory T be the set of atoms of T

- An interpretation I is a subset of a base B, i.e., $I \subseteq B$

- There are two truth values, i.e., true and false. An interpretation I associates either true or false to **all** atoms of B, i.e., all atoms in I are true, all atoms in $B \setminus I$ are false

- The logic operators (“$\vee$”, “$\wedge$”, “$\rightarrow$”, “$\neg$”) are functions mapping truth values of their arguments to a truth value: e.g., (true $\wedge$ false) $\mapsto$ false, $\neg$ true $\mapsto$ false. Operators have the usual definition

- Given an interpretation I the truth value of a proposition theory T can be determined by recursively evaluating the logical operators, i.e., eval: $\{ I \} \times \{ T \} \mapsto$ {true, false}

- An interpretation I is a model of a propositional theory T iff eval(I,T) = true

Example

- Propositional theory
  $T = (\neg a_1 \vee a_2) \wedge b \wedge (\neg b \vee a_1 \vee c)$

- The base of T
  $B = \{a_1, a_2, b, c\}$

- Interpretation $\{a_1, b\}$ is not a model of T, because $(\neg a_1 \vee a_2)$ is evaluated to false

- Interpretations $\{a_1, a_2, b\}$ and $\{b, c\}$ are models of T, because all clauses of T are evaluated to true

Propositional logic programs

"$a \leftarrow b$" is equivalent to "$a \lor \neg b$", hence

clause c: $a_1 \lor \ldots \lor a_k \lor \neg b_1 \lor \ldots \lor \neg b_m$ is equivalent to
rule r: $(a_1 \lor \ldots \lor a_k) \leftarrow (b_1 \land \ldots \land b_m)$ written in ASP as
$a_1 \,|\, \ldots \,|\, a_k$:- $b_1 , \ldots , b_m$.

In answer set programming

- A propositional theory is called a (logic) program
- Clauses are rules . Rules are ended by "."
- "$a_1 \lor \ldots \lor a_k$" is called the head of rule r
- "$b_1 \land \ldots \land b_m$" is called the body of rule r
- "$\land$" corresponds to ","
- "$\lor$" corresponds to "|"
- "$\leftarrow$" corresponds to ":-"
- A rule "a_i ." with exactly one atom in the head and no atoms in the body is called a fact
- A rule "$\leftarrow b_1 \land \ldots \land b_m$." with empty head is called a constraint

Acknowledgements

Reused and extended presentations of

- Thomas Eiter
- Martin Gebser
- Torsten Schaub

Recommended reading:

- Answer Set Solving in Practice by Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub, University of Potsdam, Morgan & Claypool Publishers
- Answer Set Programming by Vladimir Lifschitz, University of Texas at Austin, Springer
- https://potassco.org
  https://github.com/potassco/guide/releases/

Rules in ASP

Logic program P comprises a set of rules:

$$a_1 \mid a_2 \mid \cdots \mid a_k \text{ :- } b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n.$$

```
e.g., head(X) | tail(X) :- coin(X), flippedCoin(X), not edge(X).
```

- a_i b_j c_l are atoms
- "not" is called negation as failure (naf) or default-negation
- All-quantified logical variables are allowed, but first we focus on the propositional case
- Rules must be "safe" (required for transforming all-quantified rules to propositional logic)
  - Roughly speaking, all variables in a_i, c_l must be contained in some b_j (more details follow)

Rules in ASP

```
e.g., p(a,X) :- q(Y,1), k("a_string"), X = Y + 20, X > 30, not exception.
```

- Terms are constants, variables, arithmetic terms. For simplicity no functional terms
- Constants are
  - symbolic constants (strings starting with some lowercase letter)
  - string constants (quoted strings)
  - integers
- Variables are strings starting with some uppercase letter. Variables are all-quantified
- Arithmetic terms have the form $-(t)$ or $(t \diamond u)$ for terms t and u with $\diamond \in \{$"+","−","*","/"$\}$
- Classical atoms have the form $p(t_1,\ldots,t_q)$ where t_i is a term and p is a predicate name, staring with some lowercase letter
  - p() with arity 0 is a classical atom. Parentheses can be dropped
- Built-in atoms have the form $t \prec u$ for terms t and u with $\prec \in \{$"<", "≤", "=", "≠", ">", "≥"$\}$
- Built-in atoms a as well as the expressions a and not a for a classical atom a are naf-literals
- Aggregate atoms will be defined later

Dinner example

- Wine bottles (brands) a, . . . ,e
- Plain ontology natively represented within the logic program
- Preference by facts

```
% A suite of wine bottles and their kinds
wineBottle(a).        isA(a,whiteWine).        isA(a,sweetWine).
wineBottle(b).        isA(b,whiteWine).        isA(b,dryWine).
wineBottle(c).        isA(c,whiteWine).        isA(c,dryWine).
wineBottle(d).        isA(d,redWine).          isA(d,dryWine).
wineBottle(e).        isA(e,redWine).          isA(e,sweetWine).

% Persons and their preferences
person(axel).         preferredWine(axel,whiteWine).
person(gibbi).        preferredWine(gibbi,redWine).
person(roman).        preferredWine(roman,dryWine).

% Available bottles a person likes
compliantBottle(X,Z) :- preferredWine(X,Y), isA(Z,Y).
```

Default negation ≠ classic negation

"not a" is negation as failure (default negation)

- We assume assertion not a as true, if there is no reason to **"believe"** in a
- There are no unnecessary facts in the model. I.e., a is not a fact or cannot be deduced

Example:

```
compliantBottle(axel,a).
bottleChosen(a) :-    not bottleSkipped(a),
                      compliantBottle(axel, a).
```

~~`bottleChosen(a) | bottleSkipped(a) :- compliantBottle(axel, a).`~~

Preferred minimal model:

M1 = `{ compliantBottle(axel,a), bottleChosen(a) }`

~~M2 = `{ compliantBottle(axel,a), bottleSkipped(a) }`~~.

Programs with negation

Extension of example:

```
compliantBottle(axel,a).

bottleChosen(X)    :- not bottleSkipped(X), compliantBottle(Y,X).
bottleSkipped(X)   :- not bottleChosen(X),  compliantBottle(Y,X).
```

Result ???

Problem: no single minimal model

Two alternatives:

- M1= `{ compliantBottle(axel,a), bottleChosen(a) },`
- M2 = `{ compliantBottle(axel,a), bottleSkipped(a) }.`

Which one to choose?

Social dinner example cont.

Extend the simple social dinner example:

```
% These rules generate multiple answer sets:
(1) bottleSkipped(X) :- not bottleChosen(X),
                         compliantBottle(Y,X).
(2) bottleChosen(X) :- not bottleSkipped(X),
                        compliantBottle(Y,X).
(3) hasBottleChosen(X) :- bottleChosen(Z), compliantBottle(X,Z).
```

- Rules (1) and (2) enforce that either `bottleChosen(X)` or `bottleSkipped(X)` is included in an answer set (but not both), if it contains `compliantBottle(Y,X)`
- Rule (3) computes which persons have a bottle

Social dinner example cont.

```
% Alternatively, we could use disjunction:

(4) bottleSkipped(X) | bottleChosen(X) :- compliantBottle(Y,X).

(3) hasBottleChosen(X) :- bottleChosen(Z), compliantBottle(X,Z).
```

- Rules (1) and (2) enforce that either `bottleChosen(X)` or `bottleSkipped(X)` is included in an answer set (but not both), if it contains `compliantBottle(Y,X)`
- Rule (3) computes which persons have a bottle
- Rule (4) (disjunction!) can be used for replacing (1)-(2), more in the appendix

Answer Set Semantics

- Variable-free, non-disjunctive programs first!

- Normal Rules

$$a \text{:- } b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n$$

  where all a, b_i , c_j are atoms

- A normal logic program P is a (finite) set of such rules

- HB(P) (Herbrand Base) is the set of all atoms with predicates and constants from P

Example

```
compliantBottle(axel,a).
wineBottle(a).
bottleSkipped(a)       :- not bottleChosen(a),  compliantBottle(axel,a).
bottleChosen(a)        :- not bottleSkipped(a), compliantBottle(axel,a).
hasBottleChosen(axel)  :- bottleChosen(a),      compliantBottle(axel,a).
```

- HB(P) = {
  wineBottle(a), wineBottle(axel),
  bottleSkipped(a), bottleSkipped(axel),
  bottleChosen(a), bottleChosen(axel),
  hasBottleChosen(a), hasBottleChosen(axel),
  compliantBottle(axel,a), compliantBottle(axel,axel),
  compliantBottle(a,a), compliantBottle(a,axel) }

Answer sets

UNIVERSITÄT KLAGENFURT

Let

- P be a normal logic program
- $M \subseteq HB(P)$ be a set of atoms

Gelfond-Lifschitz (GL) Reduct P^M

The reduct P^M is obtained as follows (based on "guessed" M):

❶ remove from P each rule

a:- $b_1, \ldots, b_m$, not $c_1, \ldots,$ not c_n where some c_i is in M

❷ remove all literals of form not c_i from all remaining rules

Answer sets

- The reduct P^M is a Horn program (clauses with at most one positive literal, i.e., facts and rules where the head may be empty)
- It has the least model $lm(P^M)$. There exists at most one minimal model

Definition:

$M \subseteq HB(P)$ is an answer set of P if and only if $M = lm(P^M)$

Intuition:

- M makes an assumption about what is true and what is false
- P^M derives positive facts under the assumption for which atom not $(\cdot)$ is true as defined by M
- If the result is M, then the assumption of M is “stable”

Computation of lm(P)

The least model of a not-free program can be computed by fixpoint iteration

Algorithm Compute_LM(P)

Input: Horn program P;

Output: lm(P)

new_M := ∅;

repeat

M := new_M;

new_M := $\{a \mid$ "a:-$b_1, \ldots, b_m$" $\in P, \{b_1, \ldots, b_m\} \subseteq M\}$

until new_M == M

return M

Examples 1

```
compliantBottle(axel,a).
wineBottle(a).
hasBottleChosen(axel) :- bottleChosen(a), compliantBottle(axel,a).
```

- P has no not (i.e., is Horn)
- thus, $P^M = P$ for every M
- the single answer set of P is
  M = lm(P) = { `wineBottle(a), compliantBottle(axel,a)` }.

Examples 2

```
(1) compliantBottle(axel,a).
    wineBottle(a).
(2) bottleSkipped(a)          :- not bottleChosen(a), compliantBottle(axel,a).
(3) bottleChosen(a)           :- not bottleSkipped(a), compliantBottle(axel,a).
(4) hasBottleChosen(axel)     :- bottleChosen(a), compliantBottle(axel,a).
```

Take M = `{ wineBottle(a), compliantBottle(axel,a), bottleSkipped(a) }`

- Rule (2) "survives" the reduction (delete `not bottleChosen(a)`)
- Rule (3) is dropped (`not bottleSkipped(a)` is false)

$Im(P^M) = M$, and thus M is an answer set

Examples 3

```
(1) compliantBottle(axel,a).
    wineBottle(a).
(2) bottleSkipped(a)        :- not bottleChosen(a), compliantBottle(axel,a).
(3) bottleChosen(a)         :- not bottleSkipped(a), compliantBottle(axel,a).
(4) hasBottleChosen(axel)   :- bottleChosen(a), compliantBottle(axel,a).
```

Take M = `{ wineBottle(a), compliantBottle(axel,a), bottleChosen(a), hasBottleChosen(axel) }`

- Rule (2) is dropped
- Rule (3) "survives" the reduction (delete `not bottleSkipped(a)`)

$Im(P^M) = M$, and therefore M is another answer set

Examples 4

```
(1) compliantBottle(axel,a).
    wineBottle(a).
(2) bottleSkipped(a)          :- not bottleChosen(a), compliantBottle(axel,a).
(3) bottleChosen(a)           :- not bottleSkipped(a), compliantBottle(axel,a).
(4) hasBottleChosen(axel)     :- bottleChosen(a), compliantBottle(axel,a).
```

Take M = { `wineBottle(a), compliantBottle(axel,a),`
`bottleChosen(a), bottleSkipped(a), hasBottleChosen(axel)` }

- Rules (2) and (3) are dropped

$Im(P^M)$ = { `wineBottle(a), compliantBottle(axel,a)` } ≠ M

Thus, M is not an answer set

Examples 5

```
(1) compliantBottle(axel,a).
    wineBottle(a).
(2) bottleSkipped(a)        :- not bottleChosen(a), compliantBottle(axel,a).
(3) bottleChosen(a)         :- not bottleSkipped(a), compliantBottle(axel,a).
(4) hasBottleChosen(axel)   :- bottleChosen(a), compliantBottle(axel,a).
```

Take M = `{ wineBottle(a), compliantBottle(axel,a)}`

- Rule (2) “survives” the reduction (delete `not bottleChosen(a)`)
- Rule (3) “survives” the reduction (delete `not bottleSkipped(a)`)

$Im(P^M)$ = `{ wineBottle(a), compliantBottle(axel,a), bottleSkipped(a), bottleChosen(a), hasBottleChosen(axel) }` ≠ M

Thus, M is not an answer set

Programs with variables

- Each rule is a shorthand for all its ground substitutions, i.e., replacements of variables with ground terms (variable-free, e.g., constants)
- Assumption: number of answer sets and the size of models are finite
- Assured, e.g., by syntactic restrictions such as no function symbols
- For simplicity we limit ground terms to constants

E.g., "`b(X) :- not s(X), c(Y,X).`" with constants `axel` and `a` is a shorthand for:

```
b(a) :- not s(a), c(a,a).
b(a) :- not s(a), c(axel,a).

b(axel) :- not s(axel), c(axel,axel).
b(axel) :- not s(axel), c(a,axel).
```

Programs with variables

- The Herbrand base of program P, HB(P), consists of all ground (variable-free) atoms with predicates and constant symbols from P
- The grounding of a rule r , Ground(r), consists of all rules obtained from r if each variable in r is replaced by some ground term (over P, unless specified otherwise)
- The grounding of program P, is $Ground(P) = \bigcup_{r \in P} Ground(r)$

Definition:

$M \subseteq HB(P)$ is an answer set of P if and only if M is an answer set of Ground(P)

Inconsistent programs

Program

```
p :- not p.
```

- This program has NO answer sets, both guesses `{ p }` and `{ }` are not answer sets
- Let P be a program and p be a new atom
- Adding

```
p :- not p.
```

to program P "kills" all answer sets of P

Constraints

- Adding

  `p :- q1,..., qm , not r1, ..., not rn, not p.`

  to P "kills" all answer sets of P that:

  - contain $q_1, \ldots, q_m$ and
  - do not contain $r_1, \ldots, r_n$

- Abbreviation:

  `:- q1,..., qm , not r1,..., not rn.`

- This is called a "constraint" (cf. integrity constraints in databases)
- A constraint is violated, if $q_1, \ldots, q_m$, not $r_1, \ldots,$ not r_n is satisfied

Social dinner example (cont.)

Task

- Add a constraint in order to filter answer sets in which for some person no bottle is chosen

```
% This rule generates multiple answer sets:
(1) bottleSkipped(X)  :- not bottleChosen(X), compliantBottle(Y,X).
(2) bottleChosen(X)   :- not bottleSkipped(X), compliantBottle(Y,X).

% Ensure that each person gets a bottle.
(3) hasBottleChosen(X) :- bottleChosen(Z), compliantBottle(X,Z).
(4) :- person(X), not hasBottleChosen(X).
```

Main reasoning tasks

Consistency: decide whether a given program P has an answer set

Cautious (resp. brave) reasoning: given a program P and ground literals $l_1, \ldots, l_n$
decide whether $l_1, \ldots, l_n$ simultaneously hold in every (resp., some) answer set of P

Query answering: Given a program P and non-ground atom a on variables $X_1, \ldots, X_k$,
list all assignments of values ν to $X_1, \ldots, X_k$ such that substituting the variables in a with ν is cautiously resp. bravely true

- Seamless integration of query language and rule language

Answer Set Computation: compute some / all answer sets of a given program P

Properties of Answer Sets

Minimality:

Each answer set M of P is a minimal model (w.r.t ⊆)

Generalization of stratified semantics:

If negation in a **normal** logic program P (no disjunction) is layered (“P is stratified”), then P has a unique answer set, which coincides with the perfect model

NP-Completeness:

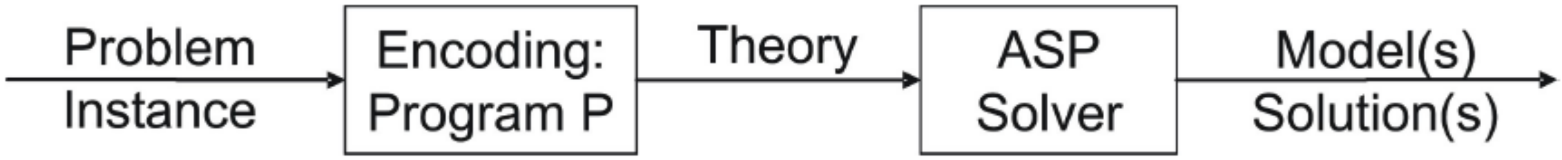
Deciding whether a normal propositional program P has an answer set is NP-complete in general
⇒ Answer Set Semantics is an expressive formalism

- Higher expressiveness through further language constructs (disjunction, weak/weight constraints)
- Computational complexity is beyond NP

Answer set programming paradigm

General idea: Models are solutions

Reduce solving a problem instance I to computing models



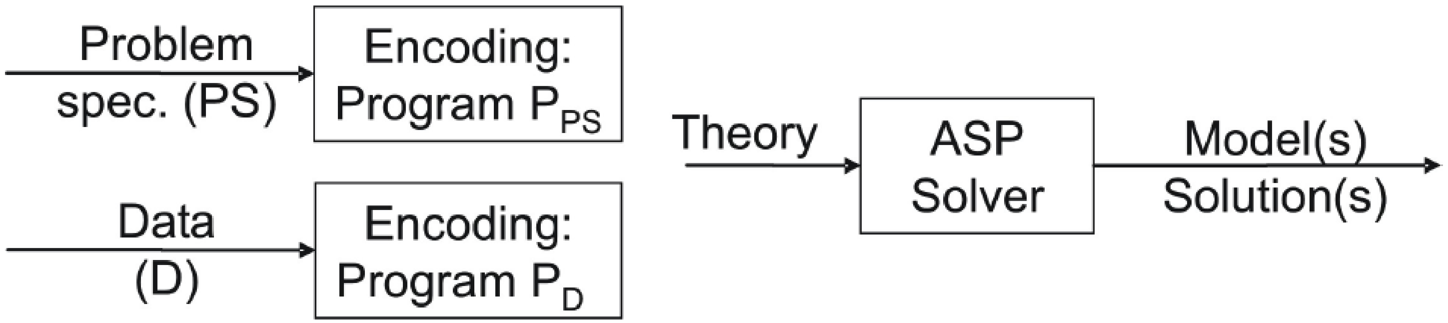
❶ *Encode* I as a logic program P, such that solutions of I are represented by models of P

❷ *Compute* some model M of P, using an ASP solver

❸ *Extract* a solution for I from M

Variant: Compute multiple models (for multiple resp. all solutions)

ASP in practice



Uniform encoding:

Separate problem specification PS and input data D (usually, facts)

- Compact, easily maintainable representation: logic programs with constraints
- Integration of knowledge representation, databases, and search techniques

Architecture of ASP Solvers

Typically, a two-level architecture

1. Grounding Step

- Input: a program P with variables
- Generate a (subset of its) grounding which has the same models
- Output: a program P with no variables

The safety property of rules is exploited by grounding to compress the resulting propositional ASP program

Architecture of ASP Solvers /2

2. Model search

This is applied for ground programs

Techniques:

- Translations to SAT (propositional satisfiability)
- Special search procedures for ASP such as
  - Conflict-driven answer set solving: From theory to practice, M. Gebser, B. Kaufmann, T. Schaub. AI Journal 187-188 (2012), Elsevier
  - Backtracking procedures for assigning truth value to atoms
  - Similar to DPLL algorithm for SAT solving
- Important: Heuristics (which atom/rule to assign a truth value)

Exception: Lazy grounding (e.g., Alpha system) to save memory consumption

Answer Set Solvers

- Clingo https://potassco.org/clingo/
- Cmodels https://www.cs.utexas.edu/users/tag/cmodels/
- DLV https://dlv.demacs.unical.it/home
- Smodels http://www.tcs.hut.fi/Software/smodels/
- Alpha (lazy grounding) https://github.com/alpha-asp/Alpha#

(see https://en.wikipedia.org/wiki/Answer_set_programming for a more extensive list)

Applications of ASP

- configuration
- scheduling
- routing
- diagnosis
- security analysis
- computer-aided verification
- ...

Extensions of ASP

- Many extensions of normal logic programs have been proposed such as
  - Weak constraints
  - Aggregation
  - Disjunction

- Some of these extensions are motivated by applications

- Some of these extensions are syntactic sugar, other strictly add expressiveness

Weak constraints

$:\sim q_1,\ldots, q_m$, not r_1, ..., not r_n. [Weight@Level, $t_1,\ldots,t_k$]

Syntax:

- Weight and Level are integers or variables bound in $q_1,\ldots, q_m$
- $t_1,\ldots,t_k$ are terms, e.g., constants or variables bound in $q_1,\ldots, q_m$
- @Level may be omitted. In this case Level = 0. $t_1,\ldots,t_k$ may be omitted

Semantics:

- Let PRIOS be the set of all Weight@Level, $t_1,\ldots,t_k$ elements of weak constraints in a ground program P and an answer set A of P, where $q_1,\ldots, q_m$, not r_1, ..., not r_n is satisfied in A
- Note, all elements in PRIOS are ground. PRIOS is a set. Duplicates are removed
- The answer sets of program P are those answer sets of P which minimize the sum of the weights of the violated weak constraints
- Minimize the violation of high-level constraints first
- Weak constraints provide means for optimizing objective functions

Weak constraints – Examples

```
a :- not b.    b :- not a.    c :- b.
:~ a. [1@0,1]
:~ b. [1@0,1]
:~ c. [1@0,1]
```

Optimal answer sets: `{a}, {b,c}` with sum of weights 1 at level 0

```
a :- not b.    b :- not a.    c :- b.        d :- b.
:~ a. [3,a]
:~ b. [1,b]
:~ c. [1,cd]
:~ d. [1,cd]
```

Optimal answer set: `{b,c,d}` with sum of weights 2 at level 0

Aggregate atoms

Aggregation allows to express properties over a set of literals that are true in a model

Example: Given a set of facts defining prices of wines, find the price of the most expensive one

```
costs(a,1). costs(b,7).  costs(c,5). costs(d,2). costs(e,5).

expensiveWine(Y) :- Y = #max{C : costs(W,C)}.

% grounded and evaluated to

expensiveWine(7) :- 7 = #max{1:costs(a,1); 7:costs(b,7); 5:costs(c,5);
                              2:costs(d,2); 5:costs(e,5)}.
```

- `Y = #max{C : costs(W,C)}` is an aggregate atom
- `#max` is a build-in aggregate function defined over
  - `{C : costs(W,C)}` a collection of aggregate elements
  - `C : costs(W,C)` is an aggregate element

Aggregate atoms

Aggregate atoms: $u_1 \prec_1$ #aggr $E \prec_2 u_2$ (either $u_1 \prec_1$ or $\prec_2 u_2$ may be omitted)

- u_1 and u_2 are terms (i.e., variables or integers)
- Aggregate relation: $\prec \in$ {“<”,“≤”,“=”,“≠”,“>”,“≥”}
- Aggregate function: #aggr ∈ {“#count”, “#sum”, “#max”, “#min”}
- E is a collection of aggregate elements separated by “;”

Aggregate element: $t_1,\ldots,t_m:l_1,\ldots,l_n$

- $t_1,\ldots,t_m$ are terms (e.g., variables or constants)
- $l_1,\ldots,l_n$ are naf-literals (atom a_i or not a_i)
- Aggregate elements are instantiated during grounding

Satisfaction of aggregate atoms

E.g., aggregate atom: `#sum{C,W : costs(W,C)} = Y`

Aggregate element (E): `C,W : costs(W,C)`

Facts: `costs(c,5). costs(e,5).`

Interpretation I = {`costs(c,5), costs(e,5)`}

Instantiation of E: E ={`5,c:costs(c,5); 5,e:costs(e,5)`}

- Given a collection E of aggregate elements and an interpretation $I \subseteq HB(P)$ of program P
- $eval(E,I) = \{(t_1,\ldots,t_m) \mid t_1,\ldots,t_m : l_1,\ldots,l_n$ occurs in E and $l_1,\ldots,l_n$ are true with respect to $I\}$

E.g., evaluation of E w.r.t. I: eval(E,I) = {`(5,c), (5,e)`}

- Aggregate atom $\#aggr\ E \prec u$ is true (or false) with respect to I if $\#aggr(eval(E,I)) \prec u$ is true (or false) with respect to I
- #aggr is applied on the **first** elements of the the tuples in eval(E,I)

E.g., application of aggregate function: `#sum({(5,c), (5,e)})=10`

Satisfaction of aggregate atoms

Notes:

- Truth of $u \prec \#aggr\ E$ is analogy defined
- The aggregate function #aggr is applied on the set provided by eval(E,I)
- Duplicates are removed!

Examples

```
costs(a,1). costs(b,7).  costs(c,5). costs(d,2). costs(e,5).

expensiveWine(Y):- #max{C : costs(W,C)} = Y.
```

Instantiation of aggregate element "`C : costs(W,C)`":

- {`1:costs(a,1); 7:costs(b,7); 5:costs(c,5); 2:costs(d,2); 5:costs(e,5)`}

Evaluation of instantiated aggregate element is

- {`(1), (7), (5), (2),` ~~`(5)`~~}

Application of `#max` on evaluation result provides `7`

- `expensiveWine(7)` is true

Examples of aggregate atoms

```
q :- 0 <= #count{X,Y : a(X,Z,k),b(1,Z,Y)} <= 3.

q(Z) :- 2 < #sum{V : d(V,Z)}, c(Z).

p(W) :- #min{S : c(S); T: d(T)} = W.

:- #max{V : d(V,Z)} > G, c(G).
```

Safety

- Let $a_1 \mid a_2 \mid \cdots \mid a_k$:- $b_1, \ldots, b_n$ be a rule r where $b_1, \ldots, b_n$ are naf-literals
- A variable is global in a rule or weak constraint r, if it appears outside of aggregate elements in r
- For a set V of variables and literals $b_1,\ldots,b_n$, $v \in V$ is bound by $b_1,\ldots,b_n$ if v occurs outside of arithmetic terms in some b_i for $1 \leq i \leq n$ such that b_i is
  - a classical atom, i.e., b_i is not negated, no built-in atom and no aggregate atom, or
  - a built-in atom t=v or v=t, and any member of V occurring in t is bound by $\{b_1,\ldots,b_n\} \backslash b_i$ (e.g., t can be an arithmetic term, a constant or a variable) or
  - an aggregate atom #aggr E = v, and any member of V occurring in E is bound by $\{b_1,\ldots,b_n\} \backslash b_i$
- The entire set V of variables is bound by $b_1,\ldots,b_n$ if each $v \in V$ is bound by $b_1,\ldots,b_n$

- A rule or weak constraint r is safe if the set V of global variables in r is bound by $b_1,\ldots,b_n$, and for each aggregate element $t_1,\ldots,t_q : l_1,\ldots,l_m$ in r with occurring variable set W, the set $W \backslash V$ of local variables is bound by $l_1,\ldots,l_m$

Examples

Safe rules and constraints

- `a(X) :- node(X), #count{V : edge(V,X)} > 0.`
- `a(X) :- node(X), not #count{V : edge(V,X)} = 0.`
- `a(X) :- #count{V : node(V), succ(V,Z), not node(Z)} = X.`
- `:- #count{V : edge(V,Y), not edge(Y,V)} = X, X > 2.`
- `:- not node(X), #count{V : edge(V,Y)} = X.`

Unsafe rules and constraints

- `a(X) :- not node(X), #count{V : edge(V,X)} > 0.`
- `a(X) :- node(X), #count{V : edge(V,X)} > Z.`
- `a(X) :- node(X), #count{V : edge(V,X), not edge(V,Y)} > 0.`
- `a(X) :- #count{V : node(V), not edge(V,Y), Y=V+Z} > 0.`
- `:- #count{V : edge(V,Y), not edge(Y,X)} > 0, X > 2.`
- `:- #count{V : edge(V,Y)} > 0, X > Y.`
- `:- not node(X), #count{V : edge(V,Y)} > X.`

Clingo

(one of the) most efficient ground-and-solve ASP systems

Various special language constructs

Conditional Literals:

Given **`r(a), r(b), r(c)`** as facts

- **`p :- q(X) : r(X).`** stands for
- **`p :- q(a), q(b), q(c).`**

Choice rules:

- **`2 {p(X,Y) : q(X)} 7 :- r(Y).`**

Conditional literals

Form of conditional literals:

$A_0 : A_1 , \dots , A_n$ where A_i is a literal and may contain variables

- Instantiate the “head literal” A_0 where the instantiations of the condition $A_1 , \dots , A_n$ is true

- The predicates of literals on the right-hand side of a colon (:) are usually defined from facts without any negative recursion, i.e., these facts can be fully evaluated by the grounder

- A conditional literal is terminated by “;” when further literals in the rule body follow

Conditional literals

Example

```
person(jane). important(jan).  available(jane).
person(john). important(john). available(john).
person(sam).

schedule.

meet :- available(X) : person(X), important(X);schedule.

% rule for meet is grounded to

meet :- available(jane), available(john), schedule.
```

Choice rules

By choice rules, we choose a subset of classical atoms of the head:

$\{A_1 ; \ldots ; A_m\}$:- $A_{m+1}, \ldots , A_n$, not $A_{n+1}, \ldots$, not A_q.

- $A_1 ; \ldots ; A_m$ are classical atoms or conditional literals where the head is a classical atom
- If body is satisfied in an answer set
  then any subset of $\{A_1 ; \ldots ; A_m\}$ can be included in the answer set
- This behavior can be implemented by generating new symbols and rules

Example: `{a} :- b. b.`

has two answer sets, i.e. ,
`{b}` and
`{a,b}`

Choice rules with restrictions

We can add optionally lower and upper bounds or restrictions to choice rules:

$u_1 \prec_1 \{A_1 \; ; \ldots ; A_m\} \prec_2 u_2$:- $A_{m+1}, \ldots , A_n$, not $A_{n+1}, \ldots$, not A_q.

- Aggregate relation: $\prec \in \{$“<”,“≤”,“=”,“≠”,“>”,“≥”$\}$, if omitted then “≤” is applied
- u_1 and u_2 are terms (i.e., variables or integers)
- If body is satisfied in an answer set, then any subset of $\{A_1 \; ; \ldots ; A_m\}$ can be included in the answer set but the restrictions on the cardinality of the subset must be satisfied

This behavior can be implemented by generating new symbols, rules, and aggregate constraints

Choice rules

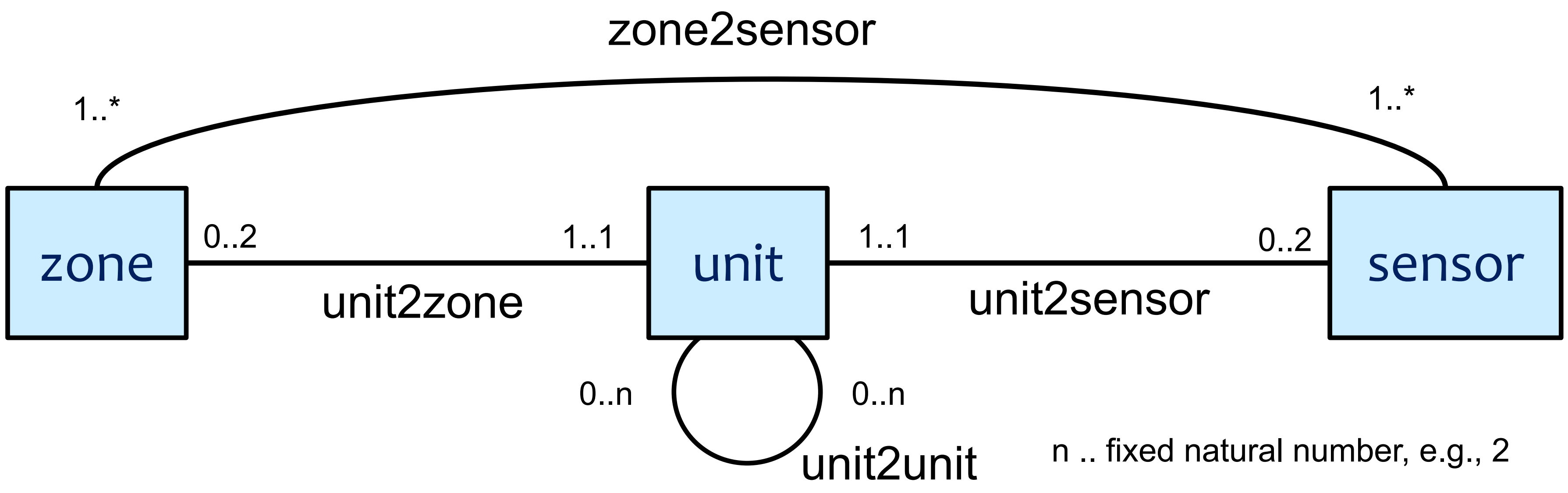
Example

```
c(2).
d(a).d(b).

1{a(Y):d(Y);b(Z):d(Z)} < X:- c(X).

Answer: 1
b(b)
Answer: 2
a(a)
Answer: 3
a(b)
Answer: 4
b(a)
```

Application: Partner Unit Problem

Subproblem of configuring railway safety systems

Partner Unit Problem



If zone Z is connected to sensor S then

Z is connected to a unit U and S is connected to this unit U or
Z is connected to a unit U1 and S is connected to a different unit U2
and U1 is connected to U2.

PUP in Clingo

```
% Facts (CUSTOMER REQUIREMENTS)

zone2sensor(z1,s1).
zone2sensor(z1,s2).
zone2sensor(z1,s3).
zone2sensor(z2,s3).
zone2sensor(z2,s4).

#const lower=2.
#const upper=4.
#const maxPU=2.
```

PUP in Clingo

```
% Rules (CONFIGURATION REQUIREMENTS)

comUnit(1..upper).

zone(Z) :- zone2sensor(Z,S).
sensor(S) :- zone2sensor(Z,S).

1 { unit2zone(U,Z) : comUnit(U)} 1 :- zone(Z).
:- comUnit(U), 3 <= #count { Z: unit2zone(U,Z)}.

1 { unit2sensor(U,S) : comUnit(U)} 1 :- sensor(S).
:- comUnit(U), 3 <= #count { S : unit2sensor(U,S)}.
```

PUP in Clingo

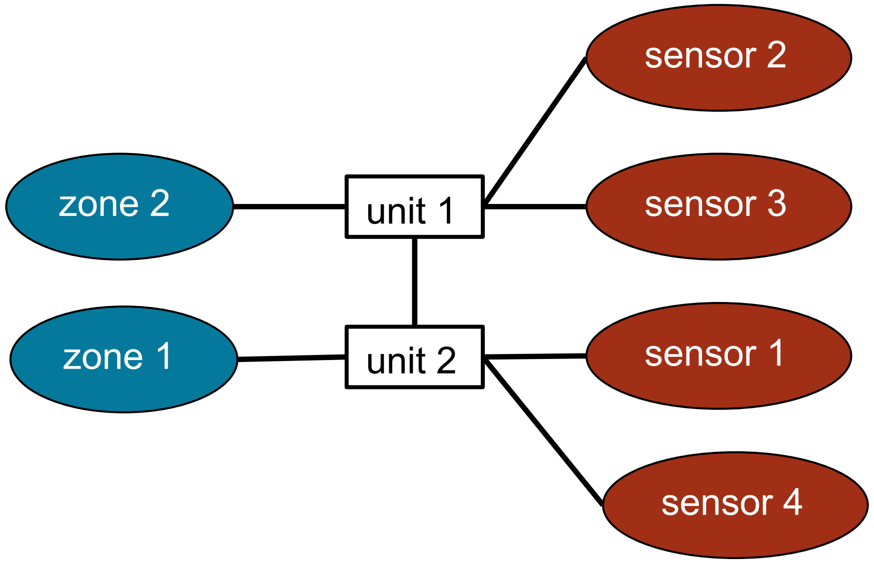
```
partnerunits(U,P):- zone2sensor(Z,S), unit2zone(U,Z), unit2sensor(P,S), U!=P.

partnerunits(U,P):- partnerunits(P,U).

:- comUnit(U), maxPU < #count { U : partnerunits(U,P)}.

% OPTIMIZATION

unitUsed(U):- unit2zone(U,Z).
unitUsed(U):- unit2sensor(U,S).
:~ unitUsed(U). [1,U]
```

PUP in Clingo

```
% SOME TUNING

lower{unitUsed(U): comUnit(U)}upper.
:- unitUsed(U), 1 < U, not unitUsed(U-1).

% SOLUTION SCHEMA

#show unit2zone/2.
#show unit2sensor/2.
#show partnerunits/2.
#show unitUsed/1.
```

PUP in Clingo

```
% SOLUTION

unitUsed(1)       unitUsed(2)

unit2zone(2,z1)   unit2zone(1,z2)

unit2sensor(2,s1) unit2sensor(2,s4)
unit2sensor(1,s2) unit2sensor(1,s3)

partnerunits(1,2) partnerunits(2,1)

Optimization: 2
```




```
Input:
zone2sensor(z1,s1).
zone2sensor(z1,s2).
zone2sensor(z1,s3).
zone2sensor(z2,s3).
zone2sensor(z2,s4).
```

Wrap up

- Declarative problem solving is based on a declarative specification of correct solutions
- Declarative problem solving can dramatically reduce development and maintenance costs
- ASP solvers implement currently one of the most efficient and expressive knowledge representation & reasoning frameworks for performing declarative problem solving

- ASP provides a **decidable** fragment of first-order logic including **disjunction** in the head of rules extended by:
  - **Non-monotonic reasoning** employing NAF, i.e., reasoning about the absence of information,
  - **Aggregation**, an instance of second-order reasoning,
  - **Weak constraints,** providing means for optimization

Appendix

Weak constraints – Example

```
employee(a). employee(b). employee(c).
employee(d). employee(e).

know(a,b). know(b,c). know(c,d). know(d,e).

same_skill(a,b).

married(c,d).

member(X,p1) :- employee(X), not member(X,p2).
member(X,p2) :- employee(X), not member(X,p1).

:~ member(X,P), member(Y,P), X != Y, not know(X,Y).      [1@1,X,Y,P]
:~ member(X,P), member(Y,P), X != Y, married(X,Y).       [1@2,X,Y,P]
:~ member(X,P), member(Y,P), X != Y, same_skill(X,Y).    [1@2,X,Y,P]
```

Weak constraints – Example

Best answer set (only member atoms):

```
{member(b,p1), member(c,p1),
 member(a,p2), member(d,p2), member(e,p2)}
```

Weight=0 at level 2, weight=6 at level 1

```
not know(X,Y): (c,b), (a,d), (a,e), (d,a), (e,a), (e,d)
```

Sub optimal answer set (only member atoms):

```
{member(b,p1), member(c,p1), member(e,p1),
 member(a,p2), member(d,p2)}
```

Weight=0 at level 2, weight=7 at level 1

```
not know(X,Y): (b,e), (c,b), (c,e), (e,b), (e,c), (a,d), (d,a)
```

Note: there is a symmetric best model to first solution, e.g., exchange **p1**/**p2**

Semantics of logic programs with negation

Two approaches

Single intended model approach:

- Select a single model of all classical models
- Agreement for so-called “stratified programs”: Perfect model

Multiple preferred model approach:

- Select a subset of all classical models
- Different selection principles for non-stratified programs

Stratified negation

Intuition: For evaluating the body of a rule containing **not r ⟨t⟩,**
the value of the „negative“ atoms **r⟨t⟩** should be known. Let ⟨t⟩ … (t_1, … ,t_n)

❶ Evaluate first r⟨t⟩
❷ if r⟨t⟩ is false, then *not* r⟨t⟩ is true,
❸ if r⟨t⟩ is true, then *not* r⟨t⟩ is false and rule is not applicable

Example:

```
compliantBottle(axel,a),
bottleChosen(X) :- not bottleSkipped(X), compliantBottle(Y,X).
```

Computed model

M = `{ compliantBottle(axel,a), bottleChosen(a) }.`

Program layers

- Evaluate predicates bottom up in layers
- Methods works if there is no cyclic negation (layered negation)

Example:

```
L0: compliantBottle(axel,a). wineBottle(a). expensive(a).
L0: bottleSkipped(X) :- expensive(X), wineBottle(X).

L1: bottleChosen(X) :- not bottleSkipped(X), compliantBottle(Y,X).
```

Unique (preferred) model resulting by layered evaluation ("perfect model"):

M = {`compliantBottle(axel,a), wineBottle(a),expensive(a), bottleSkipped(a)`}

Note: semantics defined by a procedure (violates declarativity)

Multiple preferred models

Unstratified Negation makes layering ambiguous:

```
L0: compliantBottle(axel,a).
L?: bottleChosen(X) :- not bottleSkipped(X), compliantBottle(Y,X).
L?: bottleSkipped(X) :- not bottleChosen(X), compliantBottle(Y,X).
```

- Assign to a program (theory) not one but **several** intended models!
  For instance: Answer sets!
- How to interpret these semantics? Answer set programming caters
  for the following views:
  ❶ *sceptical* reasoning: only take entailed answers, i.e., true in all models
  ❷ *brave* reasoning: each model represents a different solution to the problem
  ❸ *additionally*: one can define to consider only a subset of preferred models

Disjunctive ASP

- The use of disjunction in rule heads is natural

```
man(X) | woman(X) :- person(X).
```

- ASP has thus been extended with disjunction

$$a_1 \mid a_2 \mid \cdots \mid a_k \text{ :- } b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n .$$

- The interpretation of disjunction is “minimal”
- Disjunctive rules thus permit to encode choices

Social dinner example - disjunctive version

Replace the choice rules

```
bottleSkipped(X) :- not bottleChosen(X), compliantBottle(Y,X).
bottleChosen(X) :- not bottleSkipped(X), compliantBottle(Y,X).
```

with an equivalent (w.r.t. example) disjunctive rule

```
bottleSkipped(X) | bottleChosen(X) :- compliantBottle(Y,X).
```

- Very often, disjunction corresponds to such cyclic negation
- However, disjunction is more expressive in general, and cannot be efficiently eliminated

Answer sets of disjunctive programs

Define answer sets similar as for normal logic programs by Gelfond-Lifschitz Reduct P^M

Extend P^M to disjunctive programs:

❶ remove each rule in *Ground(P)* with some literal "not a" in the body if $a \in M$
❷ remove all literals "not a" from all remaining rules in *Ground(P)*

However, a single minimal model $lm(P^M)$ does not necessarily exist (multiple minimal models!)

Definition
$M \subseteq HB(P)$ is an answer set of P if and only if M is a minimal (wrt. $\subseteq$) model of P^M

Example

```
(1) compliantBottle(axel,a).
    wineBottle(a).
(2) bottleSkipped(a) | bottleChosen(a)  :- compliantBottle(axel,a).
(3) hasBottleChosen(axel)               :- bottleChosen(a), compliantBottle(axel,a).
```

This program contains no “not”, so $P^M = P$ for every M

Its answer sets are its minimal models:

- M1 = { `wineBottle(a), compliantBottle(axel,a), bottleSkipped(a)` }
- M2 = { `wineBottle(a), compliantBottle(axel,a), bottleChosen(a), hasBottleChosen(axel)` }

This is the same as in the non-disjunctive version

Implementation of choice rules

Choice rule

$\{A_1 ; \ldots ; A_m\}$:- $A_{m+1}, \ldots , A_n$, not $A_{n+1}, \ldots$, not A_q.

is translated into 2m+1 rules using new atoms $A, A_1', \ldots , A_m'$:

A:- $A_{m+1}, \ldots , A_n$, not $A_{n+1}, \ldots$, not A_q. % Condition

A_1 :- A, not A_1'. A_1' :- not A_1. % Choice

…

A_m :- A, not A_m'. A_m' :- not A_m. % Choice

Cardinality constraints

A (positive) cardinality constraint is of the form

low $\{A_1 ; \ldots ; A_m\}$ up % low and up are positive integers, $A_1, \ldots, A_m$ are classical atoms.

A cardinality constraint is satisfied in an answer set I
if the number of satisfied atoms of set $\{A_1, \ldots, A_m\}$ in I is between low and up (inclusive)

Frequently, conditional literals are employed: low $\{A_1{:}B_1 ; \ldots ; A_m{:}B_m\}$ up
where $B_1, \ldots, B_m$ are used to restrict the instantiations of variables occurring in $A_1 ; \ldots ; A_m$

Example:

```
2{a(X):b(X)}4.
b(r). b(s). b(t). b(u). b(v).
```

Cardinality rules

Are employed to control the cardinality of subsets

A_0 :- low $\{A_1 ; \ldots ; A_m ; \text{not } A_{m+1} ; \ldots ; \text{not } A_n\}$.

Informal meaning:

- If at least low elements of low $\{A_1 ; \ldots ; A_m ; \text{not } A_{m+1} ; \ldots ; \text{not } A_n\}$ are true in an interpretation, then add A_0 to this interpretation
- low is a lower bound on the truth assignments in the body

Example:

```
a  :- 1{b; c}. b.
```

has one answer set: `{a, b}`

Cardinality rules with bounds

A rule of the form

$$A_0 \text{ :- } low\ \{A_1 ; \ldots ; A_m ; not\ A_{m+1} ; \ldots ; not\ A_n\}\ up.$$

corresponds to

$$A_0 \text{ :- } B, not\ C.$$

$$B \text{ :- } low\ \{A_1, \ldots ,A_m, not\ A_{m+1}, \ldots , not\ A_n\}.$$

$$C \text{ :- } up{+}1\ \{A_1, \ldots ,A_m, not\ A_{m+1}, \ldots , not\ A_n\}.$$

Cardinality constraints in the head

A rule of the form

$low\ \{A_1 ; \ldots ; A_m\}\ up \text{ :- } A_{m+1}, \ldots , A_n, \text{not } A_{n+1}, \ldots , \text{not } A_q.$

corresponds to

$B \text{ :- } A_{m+1}, \ldots , A_n, \text{not } A_{n+1}, \ldots , \text{not } A_q.$

$\{A_1 ; \ldots ; A_m\} \text{ :- } B.$

$C \text{ :- } low\ \{A_1 ; \ldots ; A_m\}\ up.$

$\text{:- } B, \text{not } C.$