Constraint Programming

or Answer Set Solving in Practice

Dominik Moritz, slides based on work from Torsten Schaub (Uni Potsdam), Thomas Eiter (TU Wien), Vladimir Lifschitz (UT Austin)

Goal:

Solve hard computational programs with a simple and intuitive modeling language.

Overview

Motivation

Basics

Answer Set Programming

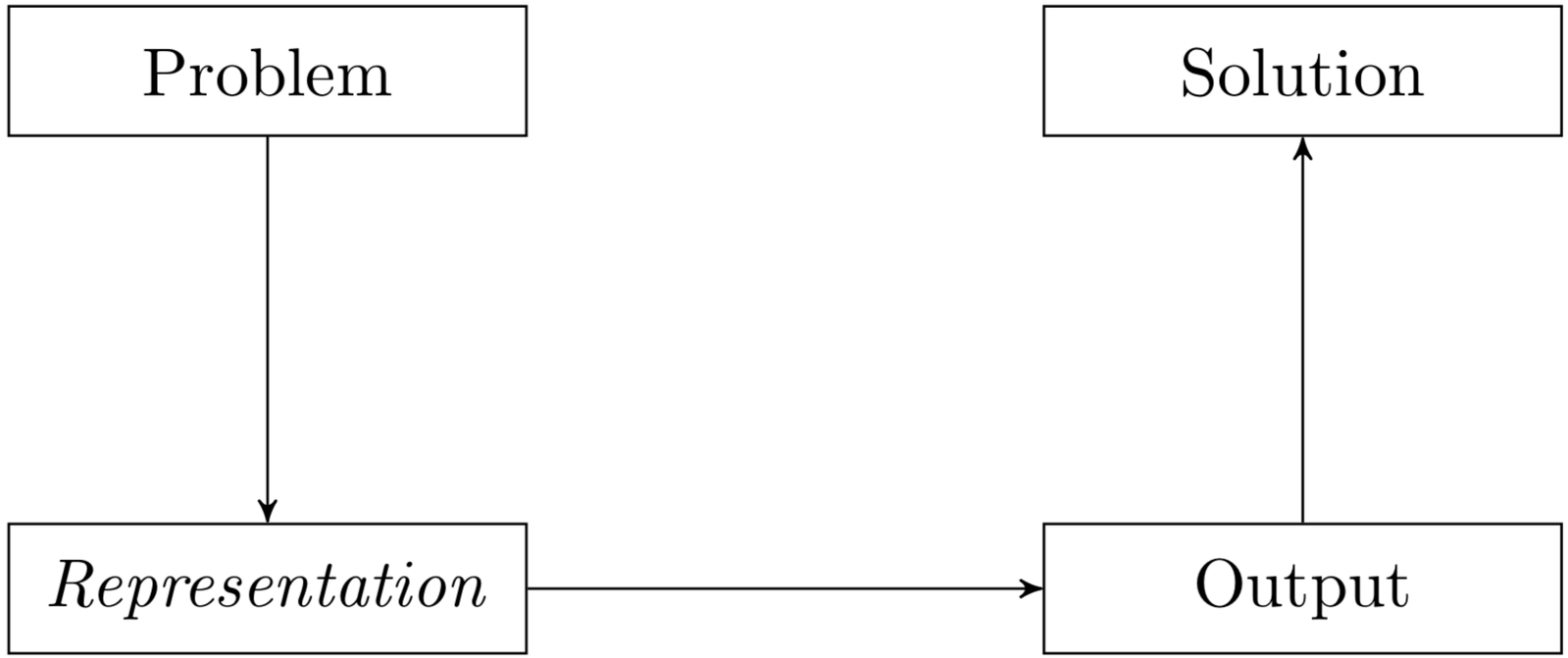
Practical Examples

Looking Behind the Curtain

Motivation

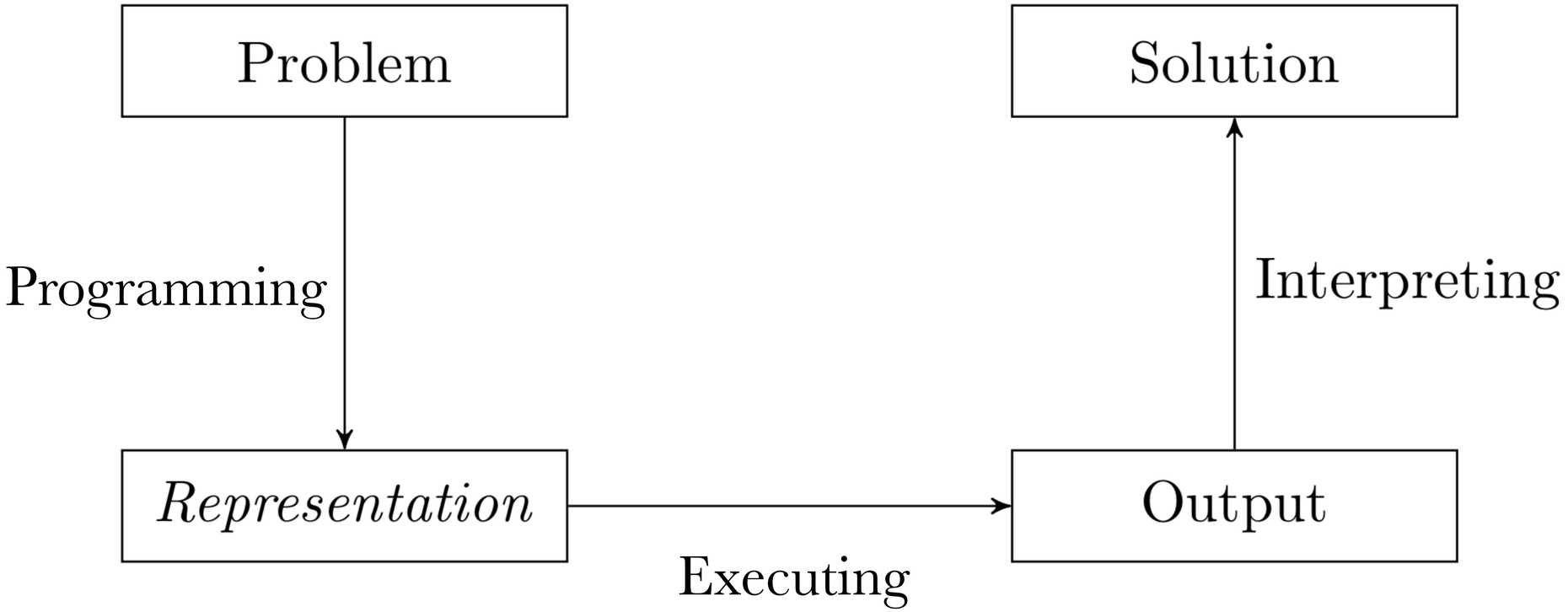
Declarative Problem Solving

"What is the problem?" versus **"How to solve the problem?"**



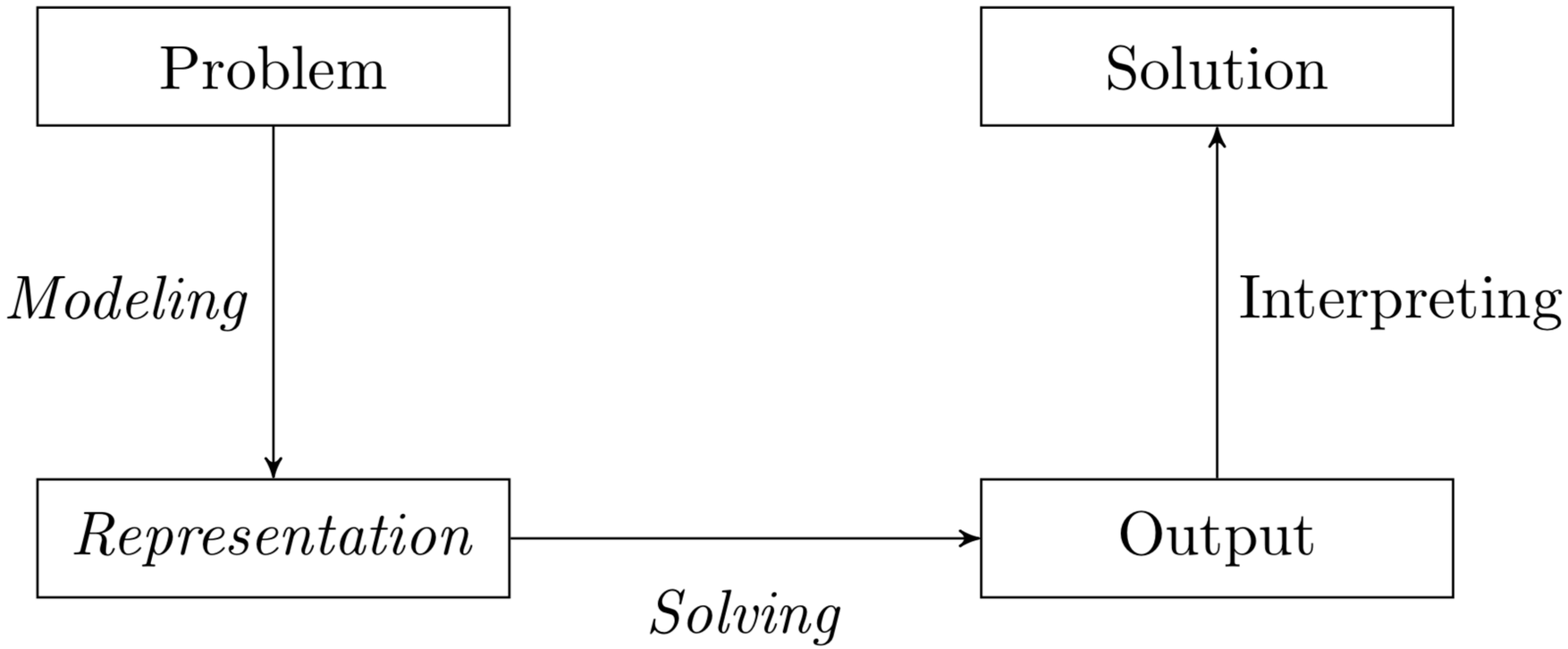
Declarative Problem Solving

"How to solve the problem?" versus **"What is the problem?"**



Declarative Problem Solving

"How to solve the problem?" versus **"What is the problem?"**



Approaches to Constraint Solving

Theorem Proving based approach (e.g. Prolog)

1. Provide a representation of the problem
2. A solution is given by a derivation of a query

Model Generation based approach (e.g. SATisfiability testing)

1. Provide a representation of the problem
2. A solution is given by a model of the representation

Answer Set Programming (ASP)

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way
- Stable model semantics [Lifschitz 1988]

Use Cases for ASP

Combinatorial search problems in the realm of P, NP, and NP^{NP} (some with substantial amount of data), like

- Automated planning
- Code optimization
- Database integration
- Decision support for NASA shuttle controllers
- Model checking
- Music composition Product configuration
- Robotics
- Systems biology System design
- Team building
- and many many more…

Basics

Terminology

Atoms or **Facts** are elementary propositions (factual statements) that may be true or false.

Literals are atoms a and their negations not a.

Constants or **predicate symbols**: x, y, z, …

Variables: X, Y, Z, …

Function symbols: f, g, h, …

Rules are expressions of the form: **head** ← **body**, or more precisely

$a \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n.$

Constraints are rules with an empty head.

Notation

	true, false	if	and	or	iff	default negation	classical negation
source code		`:-`	`,`	`|`		`not`	`-`
logic program		$\leftarrow$	,	;		$\sim$	$\neg$
formula	$\bot, \top$	$\rightarrow$	$\wedge$	$\vee$	$\leftrightarrow$	$\sim$	$\neg$

Logic 101

Commutative	$p \wedge q \iff q \wedge p$	$p \vee q \iff q \vee p$
Associative	$(p \wedge q) \wedge r \iff p \wedge (q \wedge r)$	$(p \vee q) \vee r \iff p \vee (q \vee r)$
Distributive	$p \wedge (q \vee r) \iff (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) \iff (p \vee q) \wedge (p \vee r)$
Identity	$p \wedge T \iff p$	$p \vee F \iff p$
Negation	$p \vee \sim p \iff T$	$p \wedge \sim p \iff F$
Double Negative	$\sim (\sim p) \iff p$	
Idempotent	$p \wedge p \iff p$	$p \vee p \iff p$
Universal Bound	$p \vee T \iff T$	$p \wedge F \iff F$
De Morgan's	$\sim (p \wedge q) \iff (\sim p) \vee (\sim q)$	$\sim (p \vee q) \iff (\sim p) \wedge (\sim q)$
Absorption	$p \vee (p \wedge q) \iff p$	$p \wedge (p \vee q) \iff p$
Conditional	$(p \implies q) \iff (\sim p \vee q)$	$\sim (p \implies q) \iff (p \wedge \sim q)$

Logic 101

$a \rightarrow b = \sim a \wedge b$

$\rightarrow a = F \rightarrow a = a$

$a \leftarrow = a$

$\leftarrow a = \sim a$

$\leftarrow \sim a = a$

Answer Set Programming

ASP vs Prolog

ASP features "pure" declarative programming

- the order of program rules does not matter;
- the order of subgoals in a rule does not matter;
- termination is not an issue.

Nondeterminism in ASP: Possibility to make guesses

Prolog uses a top-down approach to solving (with unification).

Unification

?- mia = X

X = mia

?- f(A,B) = f(1,2)

A = 1, B = 2

?- k(s(g), Y) = k(X, t(k)).

X = s(g), Y = t(k)

```
on(a,b).
on(b,c).
above(X,Y) :- above(X,Z), on(Z,Y).
above(X,Y) :- on(X,Y).

?- above(a,c).
Fatal Error: local stack overflow.
```

Answer Sets (Stable Models)

Answer sets are stable models. Stable models are models that are *justified* and *minimal*.

Defined as a model that satisfies $Cn(P^S) = S$. (Cn = consequence)

P^S is the reduct of S

1. delete each clause with some $\sim C_i$ such that $C_i \in S$
2. delete each $\sim C_i$ (such that $C_i \notin S$)

A program can have no, one, or multiple stable models.

Constructive flavor of ASP. Negative literals must only be true, while positive ones must also be provable. This excludes "circular derivation".

Examples of Answer Sets

$$P_4 \quad = \quad \{a \leftarrow \sim a\}$$

X	P_4^X				$Cn(P_4^X)$		
$\emptyset$	$P_3^{\emptyset}$	$=$	$\{a \leftarrow$	$\}$	$Cn(P_3^{\emptyset})$	$=$	$\{a\}$
$\{a\}$	$P_3^{\{a\}}$	$=$	$\{$	$\}$	$Cn(P_3^{\{a\}})$	$=$	$\emptyset$

Only one model: {a}. No stable model.

Examples of Answer Sets

$$P_1 \quad = \quad \left\{ \begin{array}{l} a \leftarrow \\ c \leftarrow \sim b, \sim d \\ d \leftarrow a, \sim c \end{array} \right\}$$

X	P_1^X			$\subseteq$-minimal model of P_1^X
$\{a, c\}$	$P_1^{\{a,c\}}$	$=$	$\{a \leftarrow,\ c \leftarrow\}$	$\{a, c\}$
$\{a, b, c, d\}$	$P_1^{\{a,b,c,d\}}$	$=$	$\{a \leftarrow\}$	$\{a\}$

Examples of Answer Sets

```
p :- not q.
r :- p.
s :- r, not p.
```

`{p}` is not an answer set because it is not a model

`{r, s}` is not an answer set because r is included for no reason

`{p, r}` is a model and answer set (it's the only one)

Examples of Answer Sets

```
p :- q.
p :- r.
q :- not r.
p :- not r.
```

There are two models: `{p, q}` and `{p, r}`. Only the first one is stable.

Note that Prolog cannot derive p.

Practical Examples

Example: Roads out of Berlin

```
road(berlin,potsdam).
road(potsdam,werder).
road(werder,brandenburg).
road(X,Y) :- road(Y,X).

blocked(werder,brandenburg).

route(X,Y) :- road(X,Y), not blocked(X,Y).
route(X,Y) :- route(X,Z), route(Z,Y).

drive(X) :- route(berlin ,X).

#show drive/1.
```

Example: Roads out of Berlin

```
$ clingo roads.lp
clingo version 5.2.2
Reading from roads.lp
Solving...
Answer: 1
drive(potsdam) drive(berlin) drive(werder)
SATISFIABLE

Models       : 1
Calls        : 1
Time         : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.002s
```

Basic Components

Basic rules

Integrity constraints

Choice rules

Cardinality rules

Weight rules

Basic Rules

Facts like

```
road(werder,brandenburg).
road(X,Y) :- road(Y,X).
```

Integrity Constraints

Remember logic 101. Right side implies false -> right side must be false.

Read: "it cannot be the case that"

```
:- edge(3,7), color(3,red), color(7,red).
```

Choice Rules

Lets us express a choice that the solver can make.

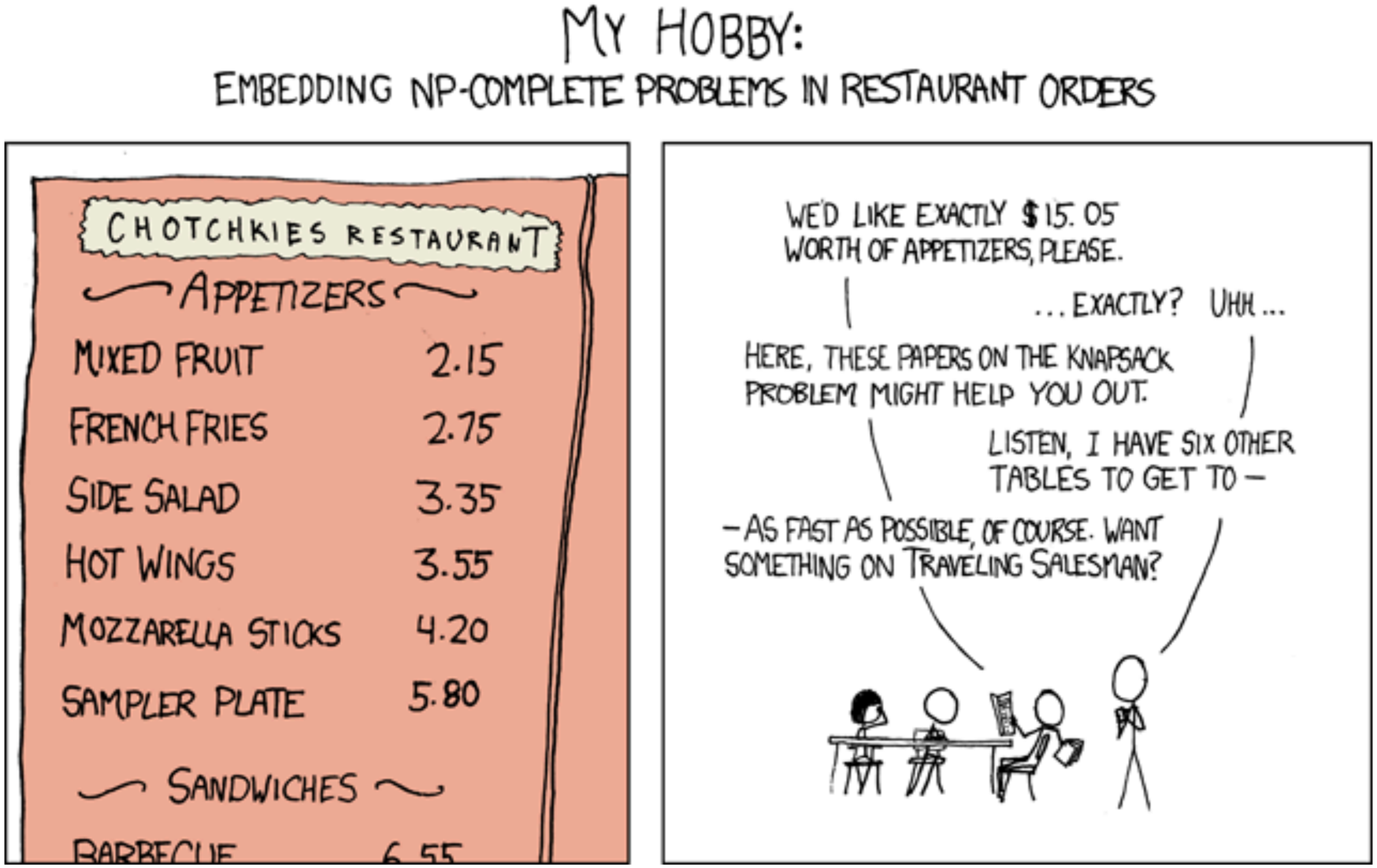
```
{ buy(pizza), buy(wine), buy(corn) } :- at(grocery).
```

Problem Modelling

Principle: Generate, Define, Test (, Optimize)

In other words: describe the search space, describe what is invalid, the remainder of the search space the solution. I think this is more natural than constructing what is valid (as done in Z3 etc).

Example: XKCD



Example: XKCD

```
#const total = 1505.

#const n = 10.
amount(0..n).

food(mixed_fruit;french_fries;side_salad;hot_wings;mozzarella_sticks;samples_place).

price(mixed_fruit,215).
price(french_fries,275).
price(side_salad,335).
price(hot_wings,355).
price(mozzarella_sticks,420).
price(samples_place,580).

prices(P) :- price(_, P).

% each food has exactly one amount
1 { food_amount(Food, Amount) : amount(Amount) } 1 :- food(Food).

% prices sums to total
total = #sum{ Price*Amount,F:food_amount(F, Amount) : price(F, Price), prices(Price), amount(Amount) }.

#show food_amount/2.
```

Example: XKCD

```
$ clingo examples/xkcd.lp 0
clingo version 5.2.2
Reading from examples/xkcd.lp
Solving...
Answer: 1
food_amount(mixed_fruit,7) food_amount(french_fries,0) food_amount(side_salad,0)
food_amount(hot_wings,0) food_amount(mozzarella_sticks,0) food_amount(samples_place,0)
Answer: 2
food_amount(mixed_fruit,1) food_amount(french_fries,0) food_amount(side_salad,0)
food_amount(hot_wings,2) food_amount(mozzarella_sticks,0) food_amount(samples_place,1)
SATISFIABLE

Models       : 2
Calls        : 1
Time         : 0.005s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.004s
```

Example: Towers of Hanoi

```
%%%%%%%%%
% Instance

peg(a;b;c).
disk(1..4).
init_on(1..4,a).
goal_on(1..4,c).
moves(15).

%%%%%%%%%
% Generate

% at each point T in time (other than 0), exactly one
% move of a disk D to some peg P must be executed.
{ move(D,P,T) : disk(D), peg(P) } = 1 :- moves(M), T =
1..M.

%%%%%%%
% Define

% projection
move(D,T)   :- move(D,_,T).

% capture state of towers

% start
on(D,P,0)   :- init_on(D,P).
% move
on(D,P,T)   :- move(D,P,T).
% inertia
on(D,P,T+1) :- on(D,P,T), not move(D,T+1), not moves(T).
% a smaller disk, with a number greater than D-1, is located on a peg P
blocked(D-1,P,T+1) :- on(D,P,T), not moves(T).
% propagate to larger disks
blocked(D-1,P,T)   :- blocked(D,P,T), disk(D).

%%%%%
% Test

% a disk D must not be moved to a peg P if D-1 is blocked at time point T
:- move(D,P,T), blocked(D-1,P,T).

% a disk D cannot be moved at time point T if it is blocked by
% some smaller disk on the same peg P
:- move(D,T), on(D,P,T-1), blocked(D,P,T).

% the goal situation, given in an instance, must be achieved at
% maximum time point M
:- goal_on(D,P), not on(D,P,M), moves(M).

% for every disk D and time point T, there is exactly one peg P
% such that on(D,P,T) holds
:- { on(D,P,T) } != 1, disk(D), moves(M), T = 1..M.
% note that this is already implied but adding it improves performance

%%%%%%%%
% Display
#show move/3.
```

Example: Graph Coloring

```
country(belgium;denmark;france;germany;netherlands;luxembourg).

% 3 color is not enough
% color(red;green;blue).
color(red;green;blue;white).

arc(france,belgium;france,luxembourg;france,germany).
arc(luxembourg,germany;luxembourg,belgium).
arc(netherlands,belgium).
arc(germany,belgium;germany,netherlands;germany,denmark).

neighbor(X,Y) :- arc(X,Y).
neighbor(Y,X) :- arc(X,Y).

% Ensure that each country has exactly one color,
1 {color(X, C) : color(C) } 1 :- country(X).

% Two neighboring countries cannot have the same color.
:- color(X1, C), color(X2, C), neighbor(X1,X2).

% symmetry breaking
:- color(germany, red).
:- color(france, blue).

#show color/2.
```

Example: Jobs

There are four people: Roberta, Thelma, Steve, and Pete.
Among them, they hold eight different jobs.
Each holds exactly two jobs.
The jobs are chef, guard, nurse, clerk, police officer (gender not implied), teacher, actor, and boxer.
The job of nurse is held by a male.
The husband of the chef is the clerk.
Roberta is not a boxer.
Pete has no education past the ninth grade.
Roberta, [and] the chef, and the police officer went golfing together.

Question: Who holds which jobs

Frame: https://pastebin.com/raw/sYAJu7F0

Example: Jobs (Answer)

```
person(roberta;thelma;steve;pete).
job(chef;guard;nurse;clerk;police_officer;teacher;actor;boxer).

male(steve;pete).
female(roberta;thelma).
require_higher_education(nurse;police_officer;teacher).

% just one person has a specific job
1 { has_job(P,J) : person(P) } 1 :- job(J).

% Each person has exactly 2 jobs
2 { has_job(P,J) : job(J) } 2 :- person(P).

% The job of nurse is held by a male.
:- person(P), has_job(P,nurse), not male(P).

% The husband of the chef is the clerk.
:- has_job(P,chef), has_job(P, clerk).
:- has_job(P,clerk), not male(P).
:- has_job(P,chef), not female(P).

% Roberta is not a boxer.
:- has_job(roberta, boxer).

% Pete has no education past the ninth grade.
:- has_job(pete, J), require_higher_education(J).

% Roberta, [and] the chef, and the police officer went golfing together.
:- has_job(roberta, chef).
:- has_job(roberta, police_officer).
:- person(P), has_job(P, chef), has_job(P, police_officer).

% From the name of the job (actor: male)
:- has_job(P,actor), not male(P).

#show has_job/2.
```

Strong Negation

```
cross :- not train.
```

Cross in the absence of knowledge about whether there is a train coming.

```
cross :- -train.
```

Cross if we have evidence that there is no train.

Looking Behind the Curtain

ASP Solver

Grounding Step

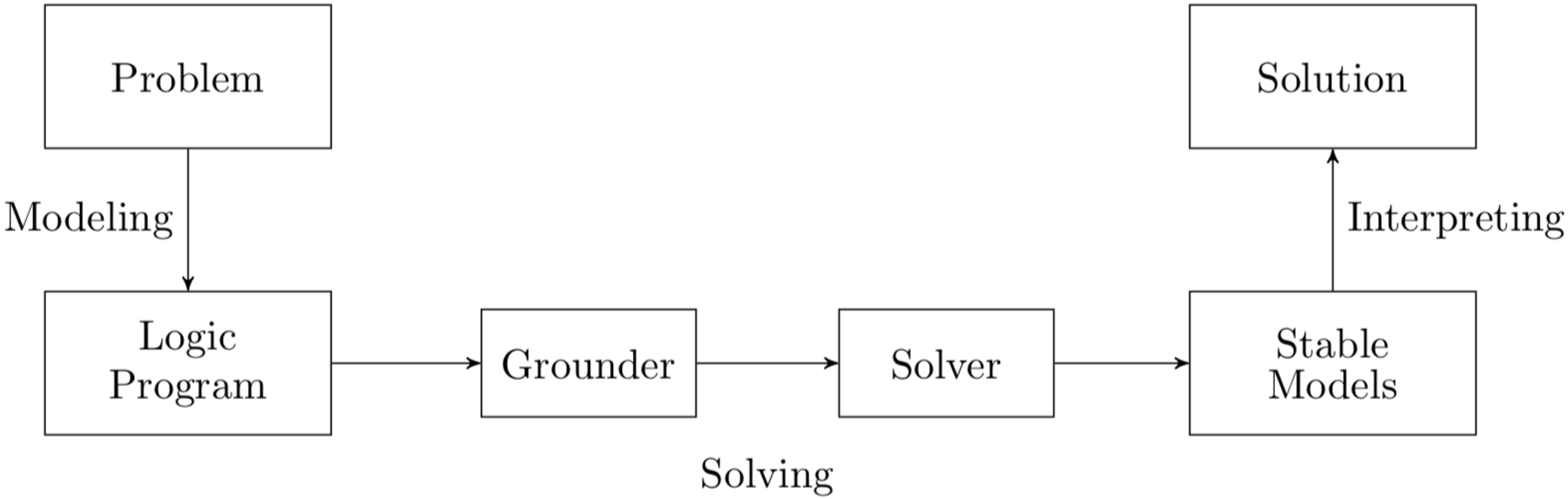
- Given a program P with variables, generate a (subset) of its grounding which has the same models
- program: `gringo`

Model Search

- Candidate generation (classical model)
- Model checking (stability!)
- Similar to (Davis-Putnam-Logemann-Loveland) DPPL algorithm for SAT + CDCL (conflict driven clause learning) with backjumping
- program: `clingo`

`clingo = gringo + clasp`

ASP Solver


Problem
Modeling
Logic Program
Grounder
Solver
Stable Models
Solution
Interpreting
Solving