Semantic and Declarative Technologies

László Kabódi, Péter Tóth, Péter Szeredi

kabodil@gmail.com
peter@toth.dev
szeredi@cs.bme.hu

Aquincum Institute of Technology

Budapest University of Technology and Economics
Department of Computer Science and Information Theory

2024 Fall Semester

Course information

- Course layout
  - Introduction to Logic — Weeks 1–2
  - Declarative Programming
    - Prolog – Programming in Logic — Weeks 3–7
    - Constraint Programming — Weeks 8–12
  - Semantic Technologies
    - Logics for the Semantic Web — Weeks 13–14
- Requirements
  - 2 assignments (150 points each) — 300 points
  - 2 tests (mid-term and final, 200 points each) — 400 points total
  - many small exercises + class activity — 300 points total
- Course webpage: `http://cs.bme.hu/~szeredi/ait`
- Course rules: `http://cs.bme.hu/~szeredi/ait/course-rules.pdf`

Part I – *practical* mathematical logic

Propositional Logic

- Basic Boolean functions (bitwise ops in C, Python, etc.)
    - `and`: $\wedge$ (&)
    - `or`: $\vee$ (|)
    - `not`: $\neg$ (~)
    - `implies`: $\rightarrow$ $A \rightarrow B$ (A implies B) is the same as $(\neg A \vee B)$
- The puzzle below is cited from “What Is The Name Of This Book?” by Raymond M. Smullyan, chapter “From the cases of Inspector Craig”
- Puzzles in this chapter involve suspects of a crime, named A, B, etc. Some of them are guilty, some innocent.
- Example:

  An enormous amount of loot had been stolen from a store. The criminal (or criminals) took the heist away in a car. Three well-known criminals A, B, C were brought to Scotland Yard for questioning. The following facts were ascertained:

  1. No one other than A, B, C was involved in the robbery.
  2. C never works without A (and possibly others) as an accomplice.
  3. B does not know how to drive.

  Is A innocent or guilty?

Inspector Craig puzzle – transforming to formal logic

- Let’s recall the facts
  1. No one other than A, B, C was involved in the robbery.
  2. C never works without A (and possibly others) as an accomplice.
  3. B does not know how to drive.
- Transform each statement into a formula involving the letters A, B, C as atomic propositions. Proposition A stands for “A is guilty”, etc.
  1. A is guilty or B is guilty or C is guilty: $A \vee B \vee C$
  2. If C is guilty then A is guilty: $C \rightarrow A$
  3. It cannot be the case that only B is guilty: $B \rightarrow (A \vee C)$
- Transform each propositional formula into conjunctive normal form (CNF), then show the clauses in simplified form:

	Original formula	CNF	Simplified clausal form
1	$A \vee B \vee C$	$A \vee B \vee C$	`+A +B +C.`
2	$C \rightarrow A$	$\neg C \vee A$	`-C +A.`
3	$B \rightarrow (A \vee C)$	$\neg B \vee A \vee C$	`-B +A +C.`

- A clause is a set of signed atomic propositions, called *literals*

Inspector Craig puzzle – resolution proof

- Collect the clauses, giving each a reference number:

`(1)`	`+A +B +C.`	Only *A*, *B*, *C* was involved in the robbery.
`(2)`	`-C +A.`	*C* never works without *A* as an accomplice.
`(3)`	`-B +A +C.`	*B* does not know how to drive.

- A resolution step requires two input clauses which have opposite literals e.g. literal 3 of clause `(1)` is `+C` while lit 1 of clause `(2)` is `-C`
- The resolution step creates a new clause, called the resolvent. It takes the union of the literals in the inputs and removes a single pair of opposite literals, e.g. resolving (1) lit 3 with (2) lit 1 results in `+A +B`
- The resolvent follows from (is a consequence of) the input clauses, as $(U \vee V) \wedge (\neg U \vee W) \rightarrow (V \vee W)$ always holds (is a tautology)
- A sample resolution proof:

		resolve `(1)` lit 2 with `(3)` lit 1 resulting in `(4)`
`(4)`	`+A +C.`	resolve `(4)` lit 2 with `(2)` lit 1 resulting in `(5)`
`(5)`	`+A.`	

- We deduced that *A* is true, so the solution of the puzzle is: *A* is guilty

Clauses in First Order Logic (FOL)

- Example: There is an island where some people are optimistic (opt)
- The following statements hold on this island:
  1. Someone having an opt parent is bound to be opt.
  2. Someone having a non-opt friend is also bound to be opt.
  3. Susan's mother has Susan's father as a friend.
- To formalize this in FOL we introduce some task-specific symbols:
  - X has a parent $Y \longrightarrow \mathtt{hasP}(X, Y)$; X has a friend $Y \longrightarrow \mathtt{hasF}(X, Y)$
  - X is opt $\longrightarrow \mathtt{opt}(X)$; $\mathtt{s}$, $\mathtt{f}$, $\mathtt{m}$ stand for Susan, her father and her mother, resp.
- The FOL form and the clausal form of the above statements:
  1. For all X and Y, X is opt if X has a parent Y and Y is opt:
     $\forall X, Y.(\mathtt{opt}(X) \leftarrow \mathtt{hasP}(X, Y) \wedge \mathtt{opt}(Y))$
     `+opt(X) -hasP(X,Y) -opt(Y).`
  2. For all X and Y, X is opt if X has a friend Y and Y is not opt:
     $\forall X, Y.(\mathtt{opt}(X) \leftarrow \mathtt{hasF}(X, Y) \wedge \neg\mathtt{opt}(Y))$
     `+opt(X) -hasF(X,Y) +opt(Y).`
  3. $\mathtt{hasP}(\mathtt{s}, \mathtt{m})$ $\mathtt{hasP}(\mathtt{s}, \mathtt{f})$ $\mathtt{hasF}(\mathtt{m}, \mathtt{f})$
     `+hasP(s,m). +hasP(s,f). +hasF(m,f).`
- We will also learn FOL resolution, on which Prolog execution is based

Part II – Prolog

Example 1: checking if an integer is a prime

- A Prolog program consists of predicates (functions returning a Boolean)
- Let's write a predicate, which is true if and only if the argument is a prime
- Programming by specification: first describe when the predicate is true, then transform the decription to Prolog code

```
prime(P) :-                  % P is a prime if
  integer(P), P > 1,         %     P is an integer and P > 1 and
  P1 is P-1,                 %     P1 = P-1 and
  \+ (                       %     it is not the case that
                             %     (there exists an integer I such that)
    between(2, P1, I),       %         2 =< I =< P1 and
    P mod I =:= 0            %         P is divisible by I
  ).                         %
```

Are you convinced of the correctness of the code? `:-)`

Example 2: append - multiple uses of a single predicate

- `app(L1, L2, L3)` is true if `L3` is the concatenation of lists `L1` and `L2`.

```
app([], L, L).                % appending an empty list with L gives L.
app([H|L1], L2, [H|L3]) :-    % appending a list composed of
                              % head H and tail L1 with a list L2
                              % gives a list with head H and tail L3 if
    app(L1, L2, L3).          %     appending L1 and L2 gives L3.
```

- app can be used, for example,
  - to check whether the relation holds:

```
| ?- app([1,2], [3], [1,2,3]).   ⟹  yes
```

  - to append two lists:

```
| ?- app([1,2], [3,4], L).       ⟹  L = [1,2,3,4] ? ; no
```

  - to split a list into two:

```
| ?- app(L1, L2, [1,2,3]).       ⟹  L1 = [], L2 = [1,2,3] ? ;
                                     L1 = [1],  L2 = [2,3] ? ;
                                     L1 = [1,2],  L2 = [3] ? ;
                                     L1 = [1,2,3], L2 = [] ? ; no
```

- Predicate `app` is available as a built-in: `append/3` (`append` with 3 args)

Example 3: A number puzzle

- An arithmetic expression is simple if it uses the four basic operations only
- Let's write a Prolog program for solving the following task:
  Given a set of integers, e.g. $\{1, 3, 4, 6\}$, and a target integer *n*, e.g. 14, build a simple arithmetic expression that contains each element of the given set exactly once, and evaluates to *n*
- Some further clarification:
  - you cannot "glue" together integers to form larger ones, e.g. forming 13 from 1 and 3 is not allowed
  - each operation can be used 0 or more times
  - parentheses can be used freely
- Examples: $1 + 6 * (3 + 4) = 43$, $(1 + 3)/4 - 6 = -5$
- The list of integers contained within an expression (in order of occurence) is called its list of leaves, e.g. the list of leaves of $6 * (3 + 4)$ is `[6,3,4]`
- A fairly hard task is to construct an expression that evaluates to 24, using integers $\{1, 3, 4, 6\}$

The number puzzle in Prolog

Blue/orange color indicates built-in/library predicates

```
% Expr uses all integers in L and evaluates to Val.
leaves_value_expr(L, Val, Expr) :-
    permutation(L, PL),       % PL is a permutation of L,
    leaves_expr(PL, Expr),    % PL is a list of leaves of Expr,
    catch(Expr =:= Val, _,    % Expr evaluates to Val, if any error
          fail).              % occurs (e.g. division by 0), simply fail

% Expr is an (arbitrary) expression having a given list of leaves L.
leaves_expr(L, Expr) :-
    L = [Expr].               % If L is a singleton, Expr is the element
leaves_expr(L, Expr) :-
    append(L1, L2, L),        % Split L into L1 ⊕ L2
    L1 \= [], L2 \= [],       % so that neither L1, nor L2 is empty ([])
    leaves_expr(L1, E1),      % Let E1 be an arbitrary expr with leaves L1
    leaves_expr(L2, E2),      % Let E2 be an arbitrary expr with leaves L2
    member(Op, [+,-,*,/]),    % Let Op be one of the four allowed operations
    Expr =.. [Op,E1,E2].      % Let Expr be a binary expresion
                              % with operation Op and operands E1 and E2
```

Part III – Constraint technology

Example 7: The 711 problem (David Gries, May 1982)

`https://www.cs.cornell.edu/gries/TechReports/82-493.pdf`

One day, a customer bought four items at a 711 store (a chain of stores in the US). The cashier bagged them and said:

- *That will be $7.11, please.*
- The customer asked: *Is it $7.11 because this is a 711 store?*
- *No*, replied the cashier, *I multiplied the prices together and got $7.11.*
- *But you are supposed to **add** them, not multiply them*, said the customer.
- *Oh, you're right!* exclaimed the cashier
- *Let me recalculate . . . that will be $7.11.*

Can you find out the price of each of the four items, based on the above conversation?

Note: calculations are assumed to be exact, no rounding!

We will use `library(clpfd)`: Constraint Logic Programming over Finite Domains

Solving the 711 problem using CLPFD: constrain-and-generate

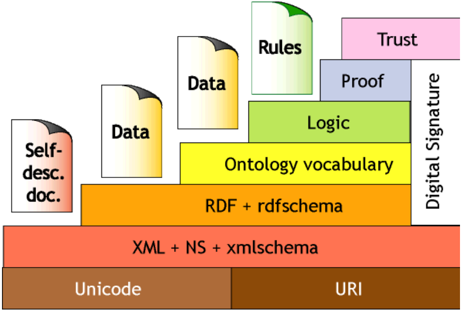
```
:- use_module(library(clpfd)).

problem711(Vs) :-
    Vs = [A,B,C,D],               % Prices of the 4 items
    domain(Vs, 1, 711),           % Prices are in cents
    A+B+C+D #= 711,               % Prices add up to 711 cents
  % A*B*C*D/100^4 = 711/100,      % Prices in $s multiply to 7.11
                                  % multiply both sides by 100^4:
    A*B*C*D      #= 711*100^3,    %
    A #=< B, B #=< C, C #=< D,    % Ensure increasing order
    labeling([ff], Vs).           % Search, using the first fail
                                  % principle: explore the narrowest
                                  % choice point first

| ?- problem711(Vs).   ⟹    Vs = [120,125,150,316] ? ; no
```

Some statistics, using SICStus Prolog (exploring the whole search space):

- Prunings: 21712 (how many times was the domain of a variable reduced)
- Run time: 0.015 sec, backtracks (branches of the search tree): 147 (brute force search would require $711^4 = 2.56 * 10^{11}$ backtracks)

Part IV – Semantic Web

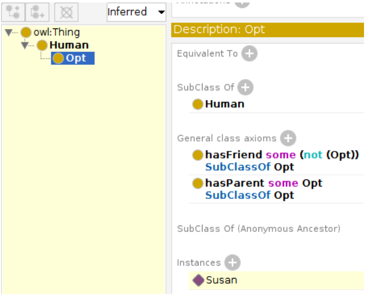
- The main goal of the Semantic Web (SW) approach:
  - make the information on the web processable by computers
  - machines should be able to understand the web, not only read it
- Achieving the vision of the Semantic Web
  - Adding (computer processable) meta-information to the web
  - Formalizing background knowledge – building so called ontologies
  - Developing reasoning algorithms and tools
- The Semantic Web layer cake – Tim Berners-Lee



Making Susan Optimistic using OWL and Protégé

- Recall a statement from the Susan example discussed earlier
  - English: Someone having an opt parent is bound to be opt.
  - FOL: $\forall X, Y.(\mathtt{opt}(X) \leftarrow \mathtt{hasP}(X, Y) \land \mathtt{opt}(Y))$
  - clausal form: `+opt(X) -hasP(X,Y) -opt(Y).`
  - OWL (Web Ontology Language): `hasParent some Opt SubClassOf Opt`
    (The set of those having some parents who are Opt is a subset of Opt)

- OWL (Web Ontology Language) represents a subset of FOL: e.g. predicates can have one or two arguments only, but efficient reasoners are available for this subset
- Protégé is a free, open source ontology editor and knowledge-base framework:



hasFriend some (not (Opt)) SubClassOf Opt

hasParent some Opt SubClassOf Opt

Susan