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 Spring 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

Course overview

1. Course overview
2. Introduction to Logic
3. Declarative Programming with Prolog
4. Declarative Programming with Constraints
5. The Semantic Web

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, by taking the union of the literals in the inputs and removing the 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 $Y \longrightarrow \texttt{hasP}(X, Y)$; X has a friend $Y \longrightarrow \texttt{hasF}(X, Y)$
  - X is opt $\longrightarrow \texttt{opt}(X)$; $\texttt{s}$, $\texttt{f}$, $\texttt{m}$ 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:
     $\forall X, Y.(\texttt{opt}(X) \leftarrow \texttt{hasP}(X, Y) \land \texttt{opt}(Y))$
     `+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:
     $\forall X, Y.(\texttt{opt}(X) \leftarrow \texttt{hasF}(X, Y) \land \neg\texttt{opt}(Y))$
     `+opt(X) -hasF(X,Y) +opt(Y).`
  3. $\texttt{hasP}(\texttt{s}, \texttt{m})$ $\texttt{hasP}(\texttt{s}, \texttt{f})$ $\texttt{hasF}(\texttt{m}, \texttt{f})$
     `+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 `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], [3,4], [1,2,3,4]). 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
    ```
- The above app predicate is available as the built-in append/3

Example 3: Countdown

- Given the list of numbers `Is` and the target number `T`, obtain a solution `E`

```
countdown(Is, T, E) :-          % E is a solution of the task
                                % with ints Is and target T if
  subseq(Is, Is1, _),           %   Is has a subsequence Is1 and
  permutation(Is1, Is2),        %   Is1 has a permutation Is2 and
  expr_leaves(E, Is2),          %   E is a formula with
                                %   list of leaves Is2 and
  E =:= T.                      %   E evaluates to T.
```

- `subseq/3` and `permutation/2` are available from the `lists` library
- The third argument of `subseq/3` contains the remaining elements from the first argument. Using _ there means we do not care about that list.
- We only have to write `expr_leaves/2`

Countdown – expr_leaves/2

- We need expr_leaves/2 to generate the valid expressions in a tree form:

```
expr_leaves(E, Is) :-         % E is a valid formula with
                              % list of leaves Is if
  append(LIs, RIs, Is),       %   Is is the concatenation of
                              %   LIs and RIs and
  LIs \== [],                 %   LIs is not an empty list and
  RIs \== [],                 %   RIs is not an empty list and
  expr_leaves(LE, LIs),       %   LE is a formula with leaves LIs and
  expr_leaves(RE, RIs),       %   RE is a formula with leaves RIs and
  build_expr(LE, RE, E).      %   combining LE and RE may yield E.
expr_leaves(I, [I]) :-        % I is a valid formula with
                              % list of leaves [I] if
  integer(I).                 %   I is an integer.
```

Countdown – `build_expr/3`

- We still need `build_expr/3` to define the operations we can use:

```
build_expr(X, Y, X+Y).    % combining exprs X and Y may yield X+Y.
build_expr(X, Y, X*Y).    % combining exprs X and Y may yield X*Y.
build_expr(X, Y, X-Y) :- % combining exprs X and Y may yield X-Y if
  X > Y.                  %   X > Y.
build_expr(X, Y, X/Y) :- % combining exprs X and Y may yield X/Y if
  X mod Y =:= 0.          %   X divided by Y gives a 0 remainder.
```

- This program may give the same (or equivalent) solution several times because of the commutativity and associativity of the operators

Part III – Constraint technology

Example 4: a cryptarithmetic puzzle in Prolog

- Solve SEND+MORE=MONEY, where the letters represent different digits, and there are no leading zeroes
- We are using the permutation technique from the countdown example to make sure that the letters represents different numbers

```
sendmoney([S,E,N,D,M,O,R,Y]) :-
    subseq([0,1,2,3,4,5,6,7,8,9],L,_),
    permutation(L,[S,E,N,D,M,O,R,Y]),
    S > 0, M > 0,
    1000*S+100*E+10*N+D + 1000*M+100*O+10*R+E
      =:= 10000*M+1000*O+100*N+10*E+Y.
```

- This works, but is very slow
- However, we can use constraints to speed up the process

SEND MORE MONEY – Prolog and CLPFD solutions

Prolog: generate and test (check)

```
send0(SEND, MORE, MONEY) :-
    Ds = [S,E,N,D,M,O,R,Y],
    subseq([0,1,2,3,4,5,6,7,8,9],L,_),
    permutation(L,[S,E,N,D,M,O,R,Y]),
    S =\= 0, M =\= 0,
    SEND is 1000*S+100*E+10*N+D,
    MORE is 1000*M+100*O+10*R+E,
    MONEY is
    10000*M+1000*O+100*N+10*E+Y,
    SEND+MORE =:= MONEY.
```

CLPFD: test (constrain) and generate

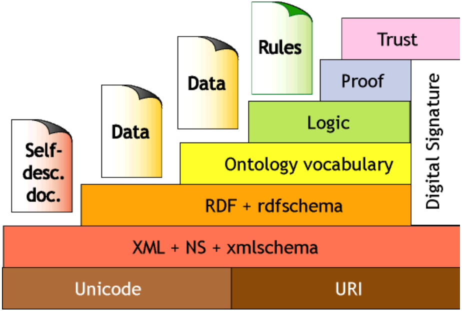
```
:- use_module(library(clpfd)).
send_clpfd(SEND, MORE, MONEY) :-
    Ds = [S,E,N,D,M,O,R,Y],
    domain(Ds, 0, 9),
    all_different(Ds),
    S #\= 0, M #\= 0,
    SEND #= 1000*S+100*E+10*N+D,
    MORE #= 1000*M+100*O+10*R+E,
    MONEY #=
    10000*M+1000*O+100*N+10*E+Y,
    SEND+MORE #= MONEY,
    labeling([], Ds).
```

How does it work?

- Variables have domains.
- Constraints can prune domains or cause failure.

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: $\forall X, Y.(\texttt{opt}(X) \leftarrow \texttt{hasP}(X, Y) \wedge \texttt{opt}(Y))$
  - 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: