Meta-Programming & ILASP

Presented by Jiong-Da Wang

2025-07-17

Contents

1. Meta-Programming 2
   1.1 Motivation 3
   1.2 A simple example 4
   1.3 Optimization 8
2. ILASP 12
   2.1 Introduction 13
   2.2 Background knowledge 14
   2.3 Hypothesis space 16
   2.4 Examples 19

1. Meta-Programming

1.1 Motivation

“程序即数据”

Meta-Programming 将“问题描述”与“求解策略”分离。

问题描述：将 Clingo 语句转换为统一的问题描述，例如 atom/1, rule/2 ...

求解策略：即约束求解器，读取问题描述，应用算法求解。

这么做的好处：

- 可以引入新的求解算法（如 GAC 等等），而不必修改问题描述。
- 可以扩展语言本身的功能：例如，ASP 本身不使用经典逻辑，即不包含排中律($p \vee \neg p$)。如果我们想要在求解中使用经典逻辑，就可以编写一个新的元程序，在其中遍历每个原子并为其动态添加排中律的约束。
- 实现更加复杂的优化功能：Clingo 自身支持标准的优化功能 (#minimize)，但只是数值层面的最优化。我们可以用 meta-programming 实现更复杂的、定性的偏好，例如子集最小化、帕累托最优、有条件的偏好等。

1.2 A simple example

example.lp:

{ a }.

b :- a .

c :- not a .

两个稳定模型:

{ a, b }

{ c }

生成问题描述： `clingo --output=reify example.lp`

Reification(具化)：不想“运行”这个程序，而是想把它“描述”出来。这就是具化（Reification）的作用。当我们对这个文件进行具化时，Clingo 会把它翻译成一系列事实。

过程:

1. 描述原子
2. 描述规则

1.2 A simple example

example.lp:

{ a }.

b :- a .

c :- not a .

symbol 1: a | symbol 2: b | symbol 3: c

Output:

```
atom_tuple(0). % define an atom
atom_tuple(0,1). % map atom 0 to
symbol 1 (a)
literal_tuple(0).
rule(choice(0),normal(0)). % {a}
atom_tuple(1).
atom_tuple(1,2).
literal_tuple(1).
literal_tuple(1,-1).
rule(disjunction(1),normal(1)).
atom_tuple(2).
atom_tuple(2,3).
literal_tuple(2).
literal_tuple(2,1).
rule(disjunction(2),normal(2)).
output(a,2).
literal_tuple(3).
literal_tuple(3,3).
output(b,3).
literal_tuple(4).
literal_tuple(4,2).
output(c,4).
```

1.2 A simple example

meta.lp (一个标准的前向解释器，模拟基本的前向推理):

```
conjunction(B) :- literal_tuple(B), %判断规则的body是否满足
        hold(L) : literal_tuple(B, L), L > 0;
    not hold(L) : literal_tuple(B,-L), L > 0.
body(normal(B)) :- rule(_,normal(B)), conjunction(B).
body(sum(B,G))  :- rule(_,sum(B,G)),
    #sum { W,L : hold(L), weighted_literal_tuple(B, L,W), L > 0 ;
           W,L : not hold(L), weighted_literal_tuple(B,-L,W), L >
0 } >= G.
  hold(A) : atom_tuple(H,A)   :- rule(disjunction(H),B), body(B).
{ hold(A) : atom_tuple(H,A) } :- rule(     choice(H),B), body(B).
#show T : output(T,B), conjunction(B), not hide(T).
```

简单来说，就是从已知的 hold 事实出发，通过 conjunction 规则判断哪些规则体被满足，然后在通过 hold 规则推导新的 hold 事实，循环往复。

1.2 A simple example

另一个元求解器：metaD.lp

其实现思路与 meta.lp 相反，metaD.lp 自顶向下，先通过析取规则生成所有可能的解，然后用约束规则过滤不符合要求的解。

1.3 Optimization

使用元编程处理复杂的优化标准。metasp 项目为我们提供了一个 encoding.lp，其中包括许多不同的偏好优化编码。

其核心思想为：实现一个通用的“支配检查”机制。（如果解 A 在某个评价标准上严格优于解 B，我们就说 A 支配 B）.一个最优解就是那个不被任何其他合法解所支配的解。这个 encoding 框架将“支配”的定义模块化，允许用户插入不同的评价标准，如基数最小化、子集最小化等。

例子：

```
1 { p; t }   :- 1 { r; s; not t } 2.
  { q; r } 1 :- 1 { p; t }.
    s        :- not q, not r.
#minimize { 1,p:p; 1,q:q; 1,r:r; 1,s:s }.
```

1.3 Optimization

encoding.lp 中优化的机制：

1. 找到一个初始的解
2. 优化循环：
    1. 将对象程序具化，转换为问题描述
    2. 将当前最优解 S_current 具化，转换为一系列事实 (true/1 谓词)
    3. 加载 encoding.lp
    4. 执行支配检查：使用 clingo 寻找一个新的答案集 S_new，并且 S_new 必须支配 S_current
    5. 若 Clingo 找到一个新的 S_new，则用 S_new 替换 S_current，然后回到第二步。若 Clingo 报告 UNSATISFIABLE，则 S_current 就是最优解，循环结束

1.3 Optimization

此处有几种不同优化方式：

1. 基数最小化，对应于 Clingo 中标准的 #minimize 语句。
2. 子集最小化，旨在找到包含原子数量最少的答案集。
3. 自定义偏好，例如包含谓词 A 的解比包含谓词 B 的解更好。

实际上，对于不同的偏好优化方式，其实现方式的不同就是自定义不同的 worse/1 谓词。

例如，自定义帕累托最优，只需要定义新的 worse 谓词以及辅助谓词，检查新的解是否在所有目标上都没有更差且在一个目标上严格更好，就可以实现帕累托最优的优化。

此外，元编程还可以帮助我们实现模拟不同逻辑（如经典逻辑、Here-and-There (HT) 逻辑、Supported Models 等）、程序分析与验证（给定一个程序，对其进行分析。例如，给定一个程序，判断其是否是“austere program”，这是一种结构上更简单的程序，拥有一些特殊的性质）。关于更多元编程的示例，请参照 https://github.com/potassco/clingo/tree/master/examples/reify。

2. ILASP

2.1 Introduction

ASP 可以表示复杂的组合问题；而 ILASP (Inductive Learning of Answer Set Programs) 的作用就是从示例中学习 ASP 程序。 ILASP 解决的学习任务由三个主要组件构成：背景知识 (the background knowledge)、模式偏置(the mode bias)和示例(examples).

1. 背景知识B是一个 ASP 程序，描述了一组在学习之前已经掌握的概念。
2. 模式偏差/语言偏差：用于表达 ILASP 可能学习的所有规则的空间。
3. 示例描述了学习程序应当满足的一组语义属性。（暂时忽略噪声/错误示例的可能性时），ILASP 的目标就是找到一个程序（通常称为假设），当其与背景知识结合时，能够覆盖任务中的每个示例。

2.2 Background knowledge

目前支持四种 ASP 规则：普通规则、选择规则、硬约束、软约束。

1. 常规规则：`h :- b_1, ..., b_n.`
2. 选择规则：`lb { h_1; ...; h_m } ub :- b_1, ..., b_n.`
3. 硬约束：`:- b_1, ..., b_n`
4. 软约束：`:~ b_1, ..., b_n.[wt@lev,t_1, ..., t_m]`

wt, lev 是算术表达式，t_1, ..., t_m 是 terms

2.2 Background knowledge

例子：Sudoku 背景知识（4*4 的数独网格，分成 4 个块）

```
cell((1..4,1..4)).
% Note that the '..' notation can be used to define a range of
atoms. In this case 16 cell atoms are defined.

block((X, Y), tl) :- cell((X, Y)), X < 3, Y < 3.
block((X, Y), tr) :- cell((X, Y)), X > 2, Y < 3.
block((X, Y), bl) :- cell((X, Y)), X < 3, Y > 2.
block((X, Y), br) :- cell((X, Y)), X > 2, Y > 2.

same_row((X1,Y),(X2,Y)) :- X1 != X2, cell((X1,Y)), cell((X2, Y)).
same_col((X,Y1),(X,Y2)) :- Y1 != Y2, cell((X,Y1)), cell((X, Y2)).
same_block(C1,C2) :- block(C1, B), block(C2, B), C1 != C2.
```

2.3 Hypothesis space

定义假设空间最简单的方法是明确定义空间中的每条规则（及其长度）。其标准定义形式为：`length ~ rule`

一个例子：

```
1 ~ p.
2 ~ p :- r.
2 ~ p :- not s.
3 ~ p :- r, not s.
```

但是这个假设空间的约束太强了，一般不会使用。

2.3 Hypothesis space

另一种方法是定义一组模式声明/模式偏差（mode declarations/mode bias）
var(t) / const(t): 占位符，用于某个常量项 t，这些可以被任何类型为 t 的变量/常量分别替换。
模式声明是一个其参数可以为占位符的命题。如果模式声明 m 中的每个占位符都被替换为正确类型的常量和变量后得到命题 a，那么我们说命题 a 和模式声明 m 兼容。
模式声明有四种类型：(其中可添加召回参数，指定该模式最多使用次数)

1. #modeh: the normal head declarations
2. #modeha: the aggregate head declarations
3. #modeb: the body declarations
4. #modec: the condition declarations
5. #modeo: the optimisation body declarations

2.3 Hypothesis space

例子：

#modeha(r(var(t1), const(t2))). 允许学习形如 `r(V, C) :- ...`的规则

#modeh(p). 允许学习形如 `p :- ...`的规则

#modeb(1, p). 规则的 body 可以包含 p，但最多只能包含 1 次

#modeb(2, q(var(t1))). 规则的 body 可以包含 q(V)形式的文字，但最多只能存在两个，如`... :- q(V1), q(V2), ...`

#constant(t2, c1).

#constant(t2, c2). 定义 t2 类型包含两个常量 c1 和 c2

#maxv(2). 规定 ILASP 学习的每一个规则中，最多只能出现两个不同的变量

注：#modeha 经常用在存在性归纳的场景，若 body 中的某个变量没有出现在 head，ILASP 可以假定这个变量存在一个值使得规则成立。

2.4 Examples

ILASP 中的示例有两种类型：正例(#pos)和负例(#neg).

正例形如 `#pos(Example_ID, {Context}, {Inclusions}, {Exclusions}).`

Example_ID (optional): 例子的 ID。

Context：上下文，即只在这个特定例子中有效的事实，它们和全局的背景知识结合在一起，共同作为该例子的局部背景知识。

Inclusions：原子集合，在该上下文中必须为真的结论。

Exclusions (optional): 原子集合，在该上下文中必须为假的结论。

负例形如 `#neg(Example_ID, {Context}, {Exclusions}, {Inclusions}).`

与正例不同的地方在于，Exclusions 变为必需，而 Inclusions 变为可选。

2.4 Examples

```
#modeh(heads(var(coin))).
#modeh(tails(var(coin))).

#modeb(heads(var(coin))).
#modeb(tails(var(coin))).
#modeb(coin(var(coin))).

#modeh(heads(const(coin))).
#modeh(tails(const(coin))).

#constant(coin, c1).
#constant(coin, c2).
#constant(coin, c3).
```

抛掷三枚硬币两次，得到两个示例：

```
#pos(
  {heads(c1), tails(c2), heads(c3)},
  {tails(c1), heads(c2), tails(c3)}).
#pos(
  {heads(c1), heads(c2), tails(c3)},
  {tails(c1), tails(c2), heads(c3)}).
```

背景知识：`coin(c1).coin(c2).coin(c3).`

ILASP 学习到的规则：

```
heads(V1) :- coin(V1), not tails(V1).
tails(V1) :- coin(V1), not heads(V1).
```