6.1 Syntax and Semantics of Constraint Logic Programs

Goal: Extend logic programming by constraints

⇒ For the signature \((\Sigma, \Delta)\) introduce a sub-signature \(\Sigma', \Delta' \subseteq \Sigma, \Delta\) for constraints.

**Def. 6.1** (Constraint-Signature, Constraints)

See slide.

Constraints: * atomic formulas over sub-signature \((\Sigma', \Delta')\),

- \(S = T\), where \(S\) and \(T\) are arbitrary terms,
- \('true, fail\) can be applied to all funct. symbols

- Special predicates in \(\Delta'\) may only be applied to the special funct. symbols in \(\Sigma'\).

**Ex 6.1.2.** Constraint-Signature for integer numbers.

Predicates \(\#>\), etc., are different from \(\geq\) in \(\Delta_1' \subseteq \Delta_2\).

This Constraint-Signature is pre-defined in Prolog and called FD (finite domain).

Constraints:

\(X + Y \#> 7 \times 3\)

\(\max(X, Y) \#= X \mod 2\)

\(f(X) + 2 = Y + 2\)

\(\#\Sigma'\)

Idea: There should be a constraint solver to handle
constraints which has to be combined with the ordinary mechanism to evaluate logic programs. To determine whether a constraint is true, one needs a constraint theory CT.

**Def. 6.13 (Constraint Theory)**

Let \((\Sigma, \Delta, \Sigma', \Delta')\) be a constraint signature. CT is constraint theory iff \(CT \subseteq F(\Sigma', \Delta', \emptyset)\) is satisfiable and only contains closed formulas. \(^n\) no free variables, e.g. \(\forall X. X + 0 \neq X\)

**Idea.** We assume that we have a constraint solver to decide \(\forall \varphi \in CT\) for all closed formulas \(\varphi \in F(\Sigma', \Delta', \emptyset)\).

**Ex 6.14** For FD, \(CT_{FD}\) should contain all true closed formulas over integers. (\(CT_{FD}\) is not decidable, not even semi-decidable. \(\rightarrow\) see Sect. 6.2).

**Def 6.15 (Syntax of LP with Constraints)**

A non-empty finite set \(S\) of definite Horn clauses over a constraint signature \((\Sigma, \Delta, \Sigma', \Delta')\) is a logic program with constraints iff \(\{\text{true}\} \in S, \{X = X\} \in S\), and for all
Constraints iff \( \{ \text{true} \} \in \mathcal{P}, \{ X = X \} \in \mathcal{P}, \) and for all other clauses \( \{ \mathcal{B}, \neg \mathcal{C}_1, \ldots, \neg \mathcal{C}_n \} \in \mathcal{P} \) we have:

(a) if \( \mathcal{B} = p(\mathcal{t}_1, \ldots, \mathcal{t}_m) \), then \( p \notin \mathcal{A}' \cup \{ \text{true, false} \} \)

(b) if \( \mathcal{C}_i = p(\mathcal{t}_1, \ldots, \mathcal{t}_m) \) and \( p \in \mathcal{A}' \), then \( \mathcal{t}_1, \ldots, \mathcal{t}_m \in \gamma(\Sigma', \mathcal{V}) \).

Condition (b) also has to hold for all queries \( \{ \neg \mathcal{C}_1, \ldots, \neg \mathcal{C}_n \} \).

Ex 6.16 factorial as a CLP

Semantics of CLP: declarative + procedural semantics

Declarative Semantics: entailment from

- clauses of the program \( \mathcal{P} \)
- constraint theory \( \mathcal{CT} \)

Def 6.17 (Declarative Semantics of CLP)

Let \( \mathcal{P} \) be a LP with constraints, let \( \mathcal{CT} \) be the corresponding constraint theory. Let \( \mathcal{G} = \{ \neg \mathcal{A}_1, \ldots, \neg \mathcal{A}_k \} \) be a query. Then the declarative semantics of \( \mathcal{P} \) and \( \mathcal{CT} \) w.r.t. \( \mathcal{G} \) is defined as:

\[
\text{DIS}_{\mathcal{P}, \mathcal{CT}, \mathcal{G}} = \{ \sigma(\mathcal{A}_1 \ldots \mathcal{A}_k) | \mathcal{B} \cup \mathcal{CT} \vdash \sigma(\mathcal{A}_1 \ldots \mathcal{A}_k), \sigma \text{ ground subst.} \}
\]

Ex 6.18 \( \mathcal{P} \) from Ex 6.16.

\( \mathcal{G} = \{ \neg \text{fact}(X, Y) \} \)

\( \mathcal{G}' = \{ \neg \text{fact}(X, 1) \} \)
Definition \( S, CT_{\text{PC}}, GI = \{ \text{fact}(1,1) \} \).

Definition \( S, CT_{\text{PC}}, G'_{\text{II}} = \{ \text{fact}(0,1), \text{fact}(1,1) \} \)

\[ \text{?- fact}(X,Y). \]
\[ \text{?- fact}(X,Y). \]
\[ X = 0; \]
\[ X = 1 \]
\[ \text{prog. error} \]

Main advantages of CLP:
- Efficiency
- Bi-directionality

Corollary 6.19 Let \( \Sigma' = \emptyset, \Delta' = \emptyset \).

Then \( \text{DIL } S, \emptyset, G_{\text{II}} = \text{DIL } S, GI \).

(i.e.: CLP is a proper extension of CP)

Now we have to define the procedural semantics, i.e., how to evaluate CLP.

Pure CLP: binary SCID-resolution with prog. clauses of \( S \)

Problem: CT can contain arbitrary formulas (not just definite Horn clauses). Constraint solver should be used to handle CT.

Idea: also represent the SCID-resolution steps as constraints (to have a uniform representation of...
(evaluation steps with prog. clauses and with constraints)
these constraints are unification problems of the form:
"does the goal unify with the head of a clause?"

Ex 6.1.10. Illustrate how SLD-resolution steps can be represented as constraints.

add-program

Query:  \(?-\text{add}(s(0), s(0), U)\).

Idea: Do not perform the required unifications directly but only collect the unification problems that have to be solved.

Configurations now have the form \((G, CO)\).

Conjunction of unification problems \(A = B\).

Start with initial configuration \((G, \text{true})\).

In each step, check whether \(CO\) remains satisfiable (otherwise, one can’t perform the desired resolution step).

Final configuration of successful computation:
\((\Box, CO)\).

Now \(CO\) can be simplified to obtain the answer subst:

\(X' = s(0) \land Z = s(0) \land X = s(0) \land Y = 0 \land U = s(s(0))\)

In pure CL, "=" can only be applied to terms, not to
Formulas. Therefore, if \( A \) and \( B \) are atomic formulas, we write \( \overline{A=B} \) as an abbreviation for a corresponding conjunction of equalities between terms:

**Def 6.1.11.** Let \( A, B \) be atomic formulas. Then we define the formula \( \overline{A=B} \) as follows:

- \( \overline{A=B} \) is false, if \( A = p(...) \), \( B = q(...) \), \( p \neq q \).
- \( \overline{A=B} \) is true, if \( A = p \), \( B = p \).
- \( \overline{A=B} \) is the formula \( s_1 = t_1 \land s_2 = t_2 \land \ldots \land s_n = t_n \), if \( A = p(s_1, \ldots, s_n) \), \( B = p(t_1, \ldots, t_n) \). 

**Ex 6.1.12.** Add example using definition of \( \overline{A=B} \)

A configuration \((6_1, CO_1)\) should only be evaluated to \((6_2, CO_2)\) if \( CO_2 \) is still satisfiable (under the axioms for = and \( \text{true} \)).

Thus, we check:

\[ \{ \forall x \, x=x, \text{true} \} \models \exists x \, CO_2 \]

existential closure of \( CO_2 \),
i.e., all variables of \( CO_2 \) are existentially quantified.
This variant of the procedural semantics of CP can easily be extended in order to handle constraints

- Now one can add both unification constraints (with \(=\)) and constraints built with \(\Delta'\) (e.g. \(X \neq 0\))

- When checking satisfiability of constraints, one also has to regard \(CT:\ (G_1, C_0)\) can be evaluated to \((G_2, C_2)\) only if:

\[\forall X = X, \text{true} \cup CT \vdash F C_2\]

- This needs the constraint solver

- After each evaluation step, one can simplify the constraints (here one has to take \(CT\) into account again)

**Def 6.1.114** (Procedural Semantics of CP)

Let \(P\) be a CP and \(CT\) be the corresponding constraint theory.

A configuration is a pair \((G, C_0)\) where \(G\) is a query or \(\Pi\) and \(C_0\) is a conjunction of constraints.

**Computation step**: \((G_1, C_0) \rightarrow_\pi (G_2, C_2)\)

See slide

**PIT 3, CT, \(\Pi\)**: Here, the atoms of \(G\) are instantiated by all those ground subst. \(\sigma\) where \(\sigma(C_0)\) is true.

**Ex 6.1.115** Procedural semantics of fact.

Here: "1" omitted in the answer.
A computation for query $G$ is a (finite or infinite) sequence of configurations:

$$(G, tme) \xrightarrow{\text{s}} (G_1, CO_1) \xrightarrow{\text{s}} (G_2, CO_2) \xrightarrow{\text{s}} \ldots$$

A computation is successful iff it ends in

$$(\square, CO).$$

The answer constraints are $\text{simplify}(CO)$

where: $CT \cup \{ \forall X X = X, tme \} \models \neg \text{simplify}(CO)$

Simplification can also be used after each computation step.

Thm 6.1.16 (Equivalence of declarative+procedural semantics for CLP)

Let $S$ be a CLP and let $CT$ be the corresponding constraint theory. Let $G$ be a query.

Then: $D II S, CT, G II = P II S, CT, G II.$
CLP has the same indeterminisms as CP, and they are resolved in the same way:

Indet. 1: Which prog. clause is used for the next step?  
⇒ top to bottom

Indet. 2: Which literal of the goal is used for the next step?  
⇒ left to right

⇒ Construct SLD trees by depth-first search from left to right.

Ex 6.1.17  
?- \text{fac}(X, 1).

\[
\begin{array}{c}
\text{fac}(X, 1) \\
\{X/0\} \ \\
\quad \Box \ \\
\quad \quad X > 0, \ X \not< X-1, \ \text{fac}(X_1, Y_1), \ Y_1 \not< X \& Y_1
\end{array}
\]

1. Answer Subst: \(X = 0\);
Then: prog. error, because \(X\) is not instantiated in \(X > 0\).
⇒ \text{fac} is not bidirectional

?- \text{fact}(X, 1)
Instead of labeling edges by multisets, we now label them by the constraints:

if \((G_1, C_0) \Rightarrow G_2, C_2)\),
then this results in the edge:

\[
\begin{array}{c}
G_1 \\
\downarrow \\
C_0 \xrightarrow{\epsilon} \text{ or } \text{simplify } (C_0) \xrightarrow{\epsilon} \\
G_2
\end{array}
\]

CLP is bidirectional:

?- fact(X, Y)

finds both solutions for \(X\) (but runs into non-termination afterwards).

If one exchanged the last 2 literals in the recursive fact-rule, it would terminate.