4.3 Indeterminism and Evaluation Strategies

Monday, 01 June, 2015 8:30

Procedural Semantics has 2 indeterminisms:

Indeterminism 1: Which program clause K is used for the next res. step?

Indeterminism 2: Which A_i in the current goal is used for the next resolution step?

$\Rightarrow$ For one configuration (G_1, σ_1) there can be <u>several</u> successor configurations with
$(G_1, \sigma_1) \vdash_{\mathcal{S}} (G_2, \sigma_2)$.

<u>Ex. 4.3.1</u> Query: ?- ancestor(Z, aline).

$(\{\neg anc(Z, aline)\}, \emptyset) \vdash_{\mathcal{S}} (\{\neg mO(Z, aline)\}, \{V/Z, X/aline\})$

$(\{\neg anc(Z, aline)\}, \emptyset) \vdash_{\mathcal{S}} (\{\neg mO(Z, Y), \neg anc(Y, aline)\}, \{V/Z, X/aline\})$

Indet 1 influences the solution.

Indet 2 influences the termination

To implement LP (on a deterministic computer), one has to resolve these 2 indeterminisms.

We first look at indeterminism 2.

It will turn out that this indet. is "harmless": it does not influence the solution, i.e., if one resolves this indeterminism (e.g., by only taking the leftmost literal), then one still finds all solutions to the query.

Main reason: Exchangement Lemma:

For a query $\{\neg A_1, \dots, \neg A_k\}$, it does not matter whether one first resolves with A_i and then with A_j or vice versa.

Lemma 4.32 (Exchangement Lemma)

Let $\{\neg A_1, \dots, \neg A_k\}$, $\{B, \neg C_1, \dots, \neg C_n\}$, $\{D, \neg E_1, \dots, \neg E_m\}$ be variable-disjoint Horn clauses. Let σ_1 be the mgu of A_i and B, let σ_2 be the mgu of $\sigma_1(A_j)$ and D. Then the 2 resolution steps on the slide are possible (first resolve with $\neg A_i$, then with $\neg A_j$).

Then there exists an mgu σ_1' of A_j and D, and an mgu σ_2' of $\sigma_1'(A_i)$ and B. So it is also possible to resolve with A_j first and then with A_i.

Then $\sigma_2 \circ \sigma_1$ and $\sigma_2' \circ \sigma_1'$ are identical up to variable renaming (i.e., there is a variable renaming ν such that $\sigma_2' \circ \sigma_1' = \nu \circ \sigma_2 \circ \sigma_1$.

Ex 4.33 Illustration of the exchangement lemma:

p(Z,Z) :- r(Z).

q(W).

$?- p(X,Y),\ q(X).$ start resolving with the p-literal

$$(\{\neg \underline{p(X,Y)}, \neg q(X)\}, \emptyset) \vdash_{\mathcal{P}} (\{\neg r(Z), \neg \underline{q(Z)}\}, \{X/Z, Y/Z\})$$

$$\vdash_{\mathcal{P}} (\{\neg r(Z)\}, \underbrace{\{W/Z\} \circ \{X/Z, Y/Z\}}_{\{X/Z,\ Y/Z,\ W/Z\}})$$

Exchangement lemma states that one could exchange these 2 resolution steps (i.e., first resolve on q, then on p). Then we get the same substitution up to variable renaming.

$$(\{\neg p(X,Y), \neg \underline{q(X)}\}, \emptyset) \vdash_{\mathcal{P}} (\{\neg \underline{p(W,Y)}\}, \{X/W\})$$

$$\vdash_{\mathcal{P}} (\{\neg r(Y)\}, \underbrace{\{W/Y, Z/Y\} \circ \{X/W\}}_{\{X/Y,\ W/Y,\ Z/Y\}})$$

The resulting substitutions can be made equal by applying the variable renaming $\nu = \{Y/Z, Z/Y\}$.

Proof of the exchangement lemma 4.3.2:

Since the clauses are variable-disjoint, the mgu σ_1 of A_i and B does not modify the variables in D, i.e. $\sigma_1(D) = D$.

σ_2 is the mgu of $\sigma_1(A_j)$ and $\underbrace{D}_{\sigma_1(D)}$

$\Rightarrow \sigma_2 \circ \sigma_1(A_j) = \sigma_2 \circ \sigma_1(D)$

$\Rightarrow \sigma_2 \circ \sigma_1$ is a unifier of A_j and D.

$\Rightarrow A_j$ and D have an mgu σ_1' and there exists a substitution σ such that

$$\sigma_2 \circ \sigma_1 = \sigma \circ \sigma_1'. \qquad (*)$$

So we can perform the first resolution step using the mgu σ_1'. Now we have to show that one can also perform the second res. step, i.e., that $\sigma_1'(A_i)$ and $\mathcal{B}$ are unifiable. This indeed holds, since σ is a unifier of $\sigma_1'(A_i)$ and $\mathcal{B}$:

$$\begin{aligned}
\sigma(\sigma_1'(A_i)) &= \sigma_2(\sigma_1(A_i)) && \text{by } (*) \\
&= \sigma_2(\sigma_1(\mathcal{B})) && \text{since } \sigma_1 \text{ unifies } A_i \text{ and } \mathcal{B} \\
&= \sigma(\sigma_1'(\mathcal{B})) && \text{by } (*) \\
&= \sigma(\mathcal{B}) && \text{since } \sigma_1' \text{ does not modify the variables of } \mathcal{B} \text{ (} \sigma_1' \text{ is mgu of } A_j \text{ and } \mathcal{D}\text{)}
\end{aligned}$$

Since σ is a unifier of $\sigma_1'(A_i)$ and $\mathcal{B}$, they also have an mgu σ_2'. Thus, there exists a substitution δ with

$$\sigma = \delta \circ \sigma_2'. \qquad (**)$$

Hence, one can exchange the resolution steps and first perform resolution on $\neg A_j$, then on $\neg A_i$.

We still have to show that $\sigma_2 \circ \sigma_1$ and $\sigma_2' \circ \sigma_1'$ are the same up to variable renaming.

To show this: Prove that $\sigma_2' \circ \sigma_1'$ is an instance of $\sigma_2 \circ \sigma_1$ and $\sigma_2 \circ \sigma_1$ is an instance of $\sigma_2' \circ \sigma_1'$.

$\sigma_2 \circ \sigma_1 = \delta \circ \sigma_2' \circ \sigma_1'$ holds.

Reason: $\sigma_2 \circ \sigma_1 = \sigma \circ \sigma_1'$ by (*)

$= \delta \circ \sigma_2' \circ \sigma_1'$ by (**).

In a similar way, one can show that there also exists a subst. δ' with $\sigma_2' \circ \sigma_1' = \delta' \circ \sigma_2 \circ \sigma_1$. $\boxtimes$

The exchangement lemma implies that one can impose an arbitrary ordering on literals in a clause and restrict ourselves to resolution steps with the "first" literal in the clause (w.r.t. the ordering).

$\Rightarrow$ Regard clauses as <u>sequences</u> of literals and use an arbitrary <u>selection function</u> to select some literal from the clause for the next resolution step.

(<u>S</u>LD $\hat{=}$ selection fct.)

Prolog uses the selection fct. that always takes the leftmost literal. Computation steps that use the first literal in the goal are called <u>canonical</u>.

<u>Def 434</u> (Canonical Computations)

A computation $(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2) \vdash_{\mathcal{P}} \ldots$ is called <u>canonical</u> iff each resolution step is done using the first literal of the respective goal G_i.

<u>Thm 435</u> (Resolving Indeterminism 2)

Let $\mathcal{P}$ be a LP, let G be a query.

For every successful computation $(G, \emptyset) \vdash_{\mathcal{P}}^{+} (\Box, \sigma)$, there also exists a canonical computation there also exists a canonical computation $(G,\emptyset) \vdash_{\mathcal{S}}^{+} (\square, \sigma')$ of the same length and σ and σ' are identical up to variable renaming.

Proof: apply the exchangement lemma repeatedly to the original computation $(G,\emptyset) \vdash_{\mathcal{S}}^{+} (\square, \sigma)$ until it is canonical. $\blacksquare$

$\Rightarrow$ Completeness of SLD-resolution still holds if one is restricted to canonical computations.

$\Rightarrow$ improves efficiency $\rightarrow$ derivation tree does not have to explore the different possibilities resulting from indet. 2.

Ex 436 In the derivation tree of Ex 431, we can restrict ourselves to canonical computations without losing any solutions.
In this example the resulting tree becomes finite.

Ex 437 Indet 2 can influence the termination behavior:

```
p :- p.
q(a).

?- q(b), p.
```

terminates in Prolog, because there is no canonical computation starting in $(\{\neg q(b), \neg p\}, \emptyset)$.

But there exists an infinite non-canonical computation
$(\{\neg q(b), \neg \underline{p}\}, \emptyset) \vdash_{\mathcal{P}} (\{\neg q(b), \neg p\}, \emptyset) \vdash_{\mathcal{P}} \dots$

2 Indeterminisms

1. Which rule of the LP is used for the next res. step?
2. Which literal of the current goal is used for the next step?

Def 4.38 (SLD Tree)

Let $\mathcal{P}$ be a LP, G be a query. The <u>SLD tree</u> of $\mathcal{P}$ w.r.t. the query G is a finite or infinite tree whose nodes are marked with sequences of atomic formulas. Its edges are marked with substitutions. The SLD tree is the smallest tree such that

- If $G = \{\neg A_1, \dots, \neg A_k\}$, then the root of the tree is marked with $A_1, \dots, A_k$.
- If a node is marked with $B_1, \dots, B_n$ and B_1 is unifiable with the positive literals of the prog. clauses $K_1, \dots, K_k$ (where K_1 occurs before K_2, K_2 before K_3 etc. in $\mathcal{P}$), then the node has k successors. Then the i-th child is marked by those atoms that result from a canonical resolution step with K_i.

So if $(\{\neg B_1, \dots, \neg B_n\}, \emptyset) \vdash_{\mathcal{P}} (\{\neg C_1, \dots, \neg C_m\}, \sigma)$ is this canonical res. step, then the i-th child is marked with $C_1, \dots, C_m$ and the edge to this child is marked with σ (restricted to the variables in $B_1, \ldots, B_n$).

The <u>answer substitutions</u> can be obtained from paths ending in $\square$. If the edges from the root to the leaf $\square$ are marked with $\delta_1, \ldots, \delta_\ell$, then we obtain the answer subst: $\delta_\ell \circ \ldots \circ \delta_2 \circ \delta_1$ (restricted to the variables in G).

Thm 4.35 guarantees that by regarding the SLD-tree, we still find all answer substitutions.

Finite Paths that end in a clause different from $\square$: <u>finite failures</u>.

$B_1, \ldots, B_n$ where B_1 cannot be unified with the positive literal of any prog. clause.

Moreover, there can be infinite paths.

Indet 2: resolved by the SLD-tree

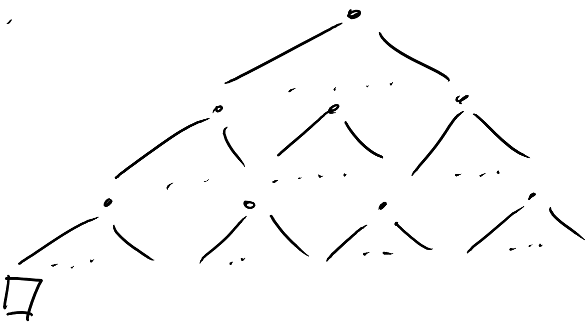
Indet 1: To resolve this indet., we have to fix the order in which the SLD-tree is constructed / traversed. (Evaluation Strategy) We also have to decide whether to stop as soon as one has found the first $\square$ or whether to search for all solutions? (In Prolog: Stop at the first $\square$, but entering ";" makes Prolog continue to the next $\square$, etc.)

<u>Options</u>:

- **breadth-first search**: first construct all nodes of height 0, then of height 1, etc.

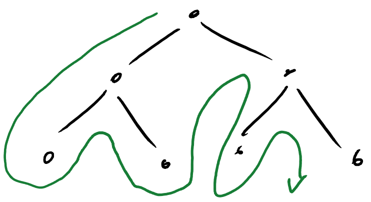
  advantage: completeness of canonical SLD-resolution is "preserved", i.e., every □ in the SLD-tree will be found. So if the query follows from the LP, this will be found out.

  disadvantage:



  building up the whole SLD-tree up to the level of the first □ can take very long
  $\Rightarrow$ too inefficient

- **depth-first search**: used by Prolog (left-to-right)

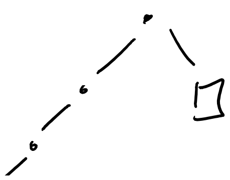


backtrack if a path ends in finite failure or if ";" is entered after reaching □

  advantage:

  □ is found very quickly if it is in the left-most paths

  disadvantage: this strategy is incomplete



Here, depth-first search does not terminate and then doesn't find □.

$\square$ does not terminate and thus, does not find $\square$ in the SLD-tree.

⇒ The programmer should take Prolog's evaluation strategy into account and ensure that solutions are found quickly before entering infinite paths.
(Choose suitable order of the literals in prog. clauses and of the clauses in the prog.)

(Violation of the principle of declarative programming:
The programmer should think (a bit) about how Prolog operates to solve queries:
1. Prog. clauses are used from top to bottom.
2. Literals in a goal are solved from left to right.)

Ex. 4.3.9 Illustrate the effect of exchanging literals in a prog. clause.
Here: the rightmost path of the tree becomes infinite

Ex. 4.3.10 Illustrate the effect of exchanging clauses in a program.
Here: the leftmost path becomes infinite
⇒ prog. does not terminate and does not find the solutions.

⇒ non-recursive clauses for a pred. p should usually

Come before recursive clauses.