Trace Analysis for Predicting the Effectiveness of Partial Evaluation

UNIVERSIDAD POLITECNICA DE VALENCIA

Germán Vidal

DSIC, Technical University of Valencia

International Conference on Logic Programming, ICLP 2008, Udine (Italia)

Introduction

partial evaluation

- Given a program and **part** of its input data—the so called **static** data—a partial evaluator returns a new, **residual** program which is specialized for the given data

Motivation

- Very few approaches devoted to formally analyze the effects of partial evaluation (mainly experimental)

We introduce a **symbolic** approach for predicting the potential effects of PE:

1. generate a finite representation that safely describes all possible **call traces**
2. analyze how this finite representation would **change** by a particular partial evaluation
3. compare the original and the transformed representations (in some cases one can **predict** the effects of running the partial evaluator beforehand)

Trace Analysis for Logic Programs

Overview of the method

- construct a context-free grammar (**CFG**) that approximates the call traces of a logic program (**LP**)
- approximate the **CFG** by a strongly regular grammar (**SRG**) (if needed)
- transform the **SRG** into a finite automaton (**FA**)

LP ⇒ CFG ⇒ SRG ⇒ FA

- **Definition** (call trace)

  We say that $\tau = p_0\, p_1 \ldots p_{n-1} \in \Pi^*$, $n \geq 1$, is a call trace for Q_0 with P iff there exists a successful SLD derivation $Q_0 \overset{p_0}{\leadsto} Q_1 \overset{p_1}{\leadsto} \ldots \overset{p_{n-1}}{\leadsto} Q_n$ where each SLD step is labeled with the predicate symbol of the selected atom

From Logic Programs to Context-Free Grammars

- A CFG is a tuple $G = \langle \Sigma, N, R, S \rangle$, where Σ and N are two disjoint sets of **terminals** and **non-terminals**, respectively, $S \in N$ is the **start** symbol, and R is a set of **rules**
- **Definition** (trace CFG)

  Let P be a program and $q \in \Pi$ a predicate symbol. The associated trace CFG is $\mathrm{CFG}^P_q = \langle \Pi, \overline{\Pi} \cup \{\mathrm{START}\}, R, \mathrm{START} \rangle$, where

  $$\begin{array}{rl} R = & \{\mathrm{START} \to \overline{q}\} \\ \cup & \{\overline{pred}(A_0) \to pred(A_0)\overline{pred}(B_1)\ldots\overline{pred}(B_n) \mid A_0 \leftarrow B_1,\ldots,B_n \in P, n \geq 0\} \end{array}$$

  where
  - $\overline{pred}(A)$ returns a **non-terminal** associated to the predicate symbol of A
  - $pred(A)$ returns a **terminal** associated to the predicate symbol of A

CFG^P_q is a **correct approximation** of the call traces for P w.r.t. the leftmost computation rule

Example

(c_1) $mlist([\,], I, [\,]).$
(c_2) $mlist([X|R], I, L) \leftarrow ml(X, R, I, L).$
(c_3) $ml(X, R, I, [XI|RI]) \leftarrow mult(X, I, XI),\ mlist(R, I, RI).$
(c_4) $mult(0, Y, 0).$ (c_5) $mult(s(X), Y, Z) \leftarrow mult(X, Y, Z1),\ add(Z1, Y, Z).$
(c_6) $add(X, 0, X).$ (c_7) $add(X, s(Y), s(Z)) \leftarrow add(X, Y, Z).$

Associated trace CFG:

$\mathrm{CFG}^P_{mlist} = \langle \{mlist, ml, mult, add\}, \{\mathrm{START}, \mathrm{MLIST}, \mathrm{ML}, \mathrm{MULT}, \mathrm{ADD}\}, R, \mathrm{START} \rangle$

where the set of rules R is as follows:

$\mathrm{START} \to \mathrm{MLIST}$	$\mathrm{ML} \to ml\ \mathrm{MULT}\ \mathrm{MLIST}$	
$\mathrm{MLIST} \to mlist$	$\mathrm{MULT} \to mult$	$\mathrm{ADD} \to add$
$\mathrm{MLIST} \to mlist\ \mathrm{ML}$	$\mathrm{MULT} \to mult\ \mathrm{MULT}\ \mathrm{ADD}$	$\mathrm{ADD} \to add\ \mathrm{ADD}$

From Context-Free Grammars to Strongly Regular Grammars

Basic idea

- We use Mohri and Nederhof's technique for approximating CFGs with SRGs, which can then be mapped to equivalent finite-state automata
- We consider **left-linear** grammars: every rule has either the form $(A \to t)$ or $(A \to t\, B)$ where t is a finite sequence of terminals and A, B are non-terminals

- **Definition** (trace SRG)

  For each set M of mutually recursive non-terminals such that their rules are not all left-linear w.r.t. the non-terminals of M, we apply the following transformation:

  1. For each non-terminal $A \in M$, we introduce a fresh non-terminal A' and add the rule $A' \to \epsilon$
  2. For each non-terminal $A \in M$ and each rule

     $$A \to t_0\, B_1\, t_1\, B_2\, t_2 \ldots B_m\, t_m \quad \text{of } \mathrm{CFG}^P_q$$

     with $m \geq 0$, $B_1, \ldots, B_m \in M$, $t_0, \ldots, t_m \in (\Pi \cup (\overline{\Pi} \setminus M))^*$, we replace this rule by the following set:

     $$\begin{array}{rcl} A & \to & t_0\, B_1 \\ B'_1 & \to & t_1\, B_1 \\ & \ldots & \\ B'_{m-1} & \to & t_{m-1}\, B_m \\ B'_m & \to & t_m\, A' \end{array}$$

- We let $\mathrm{SRG}^P_q = \langle \Pi, \overline{\Pi} \cup N \cup \mathrm{START}, R', \mathrm{START} \rangle$, where R' are the rules obtained as described above and N are the fresh non-terminals added during this process.

Example

- The sets of mutually recursive non-terminals are

  $$\{\{\mathrm{MLIST}, \mathrm{ML}\}, \{\mathrm{MULT}\}, \{\mathrm{ADD}\}\}$$

- Therefore,
  - The rules for MLIST and ML are left-linear w.r.t. $\{\mathrm{MLIST}, \mathrm{ML}\}$
  - The rules for ADD are clearly left-linear too
- However, the second rule of MULT:

  $$\mathrm{MULT} \to mult\ \mathrm{MULT}\ \mathrm{ADD}$$

  is not left-linear because, even if ADD is treated as a terminal, it appears **to the right** of the non-terminal MULT
- Therefore, in SRG^P_{mlist} we replace the original rules for MULT by the following ones:

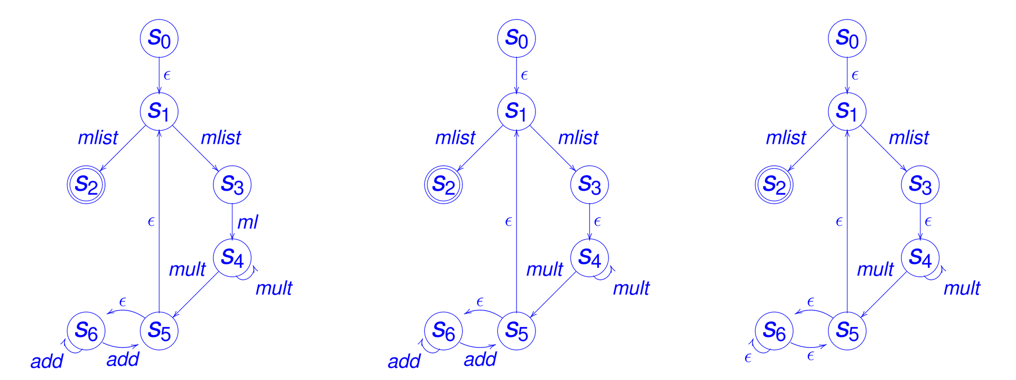
  $$\begin{array}{l} \mathrm{MULT}' \to \epsilon \\ \mathrm{MULT} \to mult\ \mathrm{MULT}' \\ \mathrm{MULT} \to mult\ \mathrm{MULT} \\ \mathrm{MULT}' \to \mathrm{ADD}\ \mathrm{MULT}' \end{array}$$

From SRGs to Finite Automata (and Regular Expressions)

- The language associated to a SRG can be represented by
  - a finite-state automaton (FA) or
  - a regular expression (RE)
- **Trace FA**
  - A FA is is specified by a tuple $\langle Q, \Sigma, \delta, s_0, F \rangle$, where
    - Q is a set of states
    - Σ is an input alphabet
    - $\delta \subseteq Q \times \Sigma \times Q$ is a set of transitions
    - $s_0 \in Q$ is the start state
    - $F \subseteq Q$ is a set of final states
  - We follow the classical approach to construct a FA from a SRG
    - there is a start state associated to the start symbol of the SRG
    - for each reduction $w \to w'$ with a rule $A \to t\, B$, we have a transition (s, α, s') in the FA, where states s, s' are associated with the sequence of non-terminals in w, w' and character α is set to the sequence t in the applied rule

Example

The trace FA associated to SRG^P_{mlist} is the FA (a) below:



(a) (b) (c)

where

$s_0 = \mathrm{START}$	$s_1 = \mathrm{MLIST}$	$s_2 = \epsilon$	final state
$s_3 = \mathrm{ML}$	$s_4 = \mathrm{MULT}\ \mathrm{MLIST}$	$s_5 = \mathrm{MULT}'\ \mathrm{MLIST}$	
$s_6 = \mathrm{ADD}\ \mathrm{MULT}'\ \mathrm{MLIST}$			

Towards Predicting the Speedup of Partial Evaluation

- The trace analysis gives us the **context** where every predicate call appears
- We introduce two **transformations** that modify the computed traces to account for the potential effects of a partial evaluation

Elimination of intermediate predicates

- For every state with exactly one input transition and one output transition, replace the label of the output transition by ϵ (i.e., delete calls to predicates which are called from a single program point)
- For the trace FA (a) above, we get the trace FA (b) by eliminating the intermediate state s_3

Removal of unfoldable predicates

- The transformation is parameterized by the output of a BTA, which annotates each predicate with either **unfold** or **memo**. Basically, the labels of unfoldable predicates are replaced with ϵ
- Given a BTA that annotates *mlist*, *ml*, and *mult* with **memo** and *add* with **unfold**, the trace FA (b) above is transformed into the trace FA (c)
- **Is this PE useful?** YES, we will achieve a significant improvement since, in every iteration for *mlist*, we will save the (recursive) evaluation of the calls to *add*

Conclusions

- The closest approach to our trace analysis is that of Gallagher and Lafave (1996), though we offer a different trade-off between analysis cost and accuracy:
  - they generate trace **terms** abstracting computation trees **independently of a computation rule**, while we generate **sequences** of predicate calls for a **specific computation rule**
  - they do not include a technique for enumerating the (possibly infinite) set of trace terms of a program, while this is a key ingredient of our approach
- A proof-of-concept implementation of our technique, called PEPE, is publicly available from `http://german.dsic.upv.es/pepe.html`
- Our approach is a first step towards the development of automated techniques and tools for predicting the potential speedup of partial evaluation