A METHOD FOR REMOVING UNUSED ARGUMENTS FROM LOGIC PROGRAMS

Emmanuil Marakakis, Chara Kounali, Kostas Vassilakis
Science Department, Technological Education Institute of Crete,
Heraklion - Crete, Greece
mmarak@cs.teicrete.gr, xkounali@venizeleio.gr, kostas@cs.teicrete.gr.

ABSTRACT
This paper studies the problem of removing unused arguments from logic programs which have been constructed by a schema-based method. Our schema-based method constructs logic programs semi-automatically. These programs have clear structure which depicts the design decisions that have been taken for their construction. On the other hand, these programs have unused arguments. We propose a method that automatically removes the unused arguments from such programs. This method is based on fold-unfold transformations.

KEY WORDS
Logic program transformation, fold, unfold, schema-based methods.

1. Introduction

This paper presents a transformation method for logic programs which are constructed by a schema-based method. This transformation method removes automatically the unused arguments from such programs. The derived programs are equivalent to the initial ones.

Our schema-based method constructs typed, moded logic programs by stepwise top-down design using five program schemata, data types (DTs) and modes [1], [2]. This schema-based method constructs logic programs with clear structure which depicts the design decisions taken during their construction. Such structured programs can be used for their maintenance and for proving their correctness [3]. However, running such programs may be inefficient due to their structured form. In addition, these logic programs have unused arguments due to the semi-automation of the construction method. These problems can be remedied by program transformation techniques. We propose a transformation method which removes the unused arguments without affecting the semantics of programs.

The ideas of program transformation methodology have been introduced in the early seventies for validating techniques such as those removing recursion in favour of iteration [4]. The program transformation approach to the development of programs has been first advocated in the case of functional languages in [5]. The unfolding and folding transformation rules have been adapted to the case of logic programs in [6].

The basic idea of the program transformation approach is to construct a sequence of programs, i.e. <P₀, P₁, …, P_r>, starting from program P₀ to program Pₙ such that P₀, P₁, …, and Pₙ have the same semantic value [7], [8]. The sequence of programs <P₀, P₁, …, P_r> is called transformation sequence, such that semantics(Pₙ) = semantics(P_i) for 0 ≤ i < r, where P_i is obtained from P_i by applying a transformation rule. During the program transformation process one is interested in reducing the computational complexity of the derived program Pₙ with respect to the initial program P₀. Meta-rules which prescribe suitable sequences of applications of transformation rules have been introduced in [7], [8] in order to ensure efficiency improvements of the transformed program. Such meta-rules are called transformation strategies. Transformation strategies formalize the transformation process so via this formalization it is possible for the programmer to perform “similar” transformations when starting from “similar” initial programs, thus avoiding the difficulty of deciding which transformation rule should be applied at every step [7], [8].

The transformation approach separates the efficiency aspect of a program from its correctness aspect. The task of writing a correct and efficient program is performed in two stages. The first stage consists in writing an initial, possibly inefficient program whose correctness can be easily shown. The second stage consists in transforming the initial program with the goal of deriving a new program which is more efficient.

In our approach a logic program is constructed by a semi-automatic schema-based method [1], [2]. Such programs have very structured form depicting the design decisions which had been followed for their construction. The correctness of such programs can be easily shown by following correctness proof schemes which correspond to the design schemata that have been used for their construction [3]. On the other hand, such programs have "unused" or "isolated" arguments due to the semi-automation of the method. The "isolated" arguments are not used in any computation of goals. In this paper, we propose an automatic transformation method based on the so called unfold/fold rules which removes the "isolated" arguments from these logic programs. The derived programs are equivalent to the initial ones and more efficient. The difficulty in program transformation is to identify the program clauses that need to be transformed and to select a transformation rule which is appropriate to
be applied in each case. These problems have been faced in our approach. We propose an iterative procedure which ends when all the unused arguments have been removed. The clauses to be transformed are the ones with unused arguments. These clauses are easily detected due to the structured form of the programs. In each iteration a specific sequence of transformations is applied which removes the identified isolated arguments.

This paper is organized as follows. An overview of our schema-based method is presented in Section 2. Then, in Section 3, we discuss the transformation of logic programs and the transformation operations which are used in our approach. Next, our transformation method is presented in Section 4. After that, a detailed example which illustrates our method is presented in Section 5. Finally, conclusions and future work are discussed in Section 6.

2. A Schema-based Method

Schemata capture commonly occurring patterns of programs. Algorithms can be expressed as schemata by abstracting problem representation details. Each schema represents a set of programs. A program can be derived from a schema by the instantiation operation. Using a library of schemata programming becomes selection of a particular schema and instantiation of its abstract components. A schema is constructed once but reused in several programs. In addition, composing instances of schemata nontrivial programs can be constructed. Logic programs are amenable to schema-based development methods due to the simplicity of their syntax. In addition, logic programmers tend to use implicitly certain program patterns.

Our schema-based method constructs typed, moded logic programs by stepwise top-down design using five program schemata, data types (DTs) and modes [1], [2]. This method uses just five program schemata. We have found this set of schemata to be very expressive. Each schema consists of a set of clause schemata together with type and mode schemata for each predicate variable occurring in them. A set of built-in DTs is also proposed which forms a kernel for defining and implementing user-defined DTs. User-defined DTs can be specified in terms of the built-in ones or other previously defined DTs. User-defined DTs are implemented by using this schema-based method.

A program is constructed using this method by successively refining undefined predicates. The lowest refinement level involves refinement by DT operations. Modes in this method support the application of the refinement operations. The programs which are constructed by this method are polymorphic many-sorted programs. They satisfy the head condition and the transparency condition [9] which ensures that no runtime type checking is needed. Finally, the programs satisfy declared input-output modes when run using the standard left-right depth-first computation rule.

3. Transformation of Logic Programs

Let’s assume that $P_0$ is an initial program and $D_0 = \emptyset$ is the initial set of new definitions. A transformation sequence is a sequence of pairs $(P_i, D_i)$ such that each pair $(P_i, D_i)$ (with either an unfold or a fold or a definition introduction transformation rule). $D_i$ denotes the conjunction of all clauses which are derived by the definition introduction transformation rule in the $i$-th transformation step. Let’s assume that C is a clause, $hd(C)$ and $bd(C)$ stand for the head and the body of clause C. $vars(C)$ stands for the variables of clause C respectively. The transformation rules are defined as follows [10], [11].

3.1 Definition Introduction Rule

From program $P_k$ program $P_{k+1}$ is derived by definition introduction by adding to $P_k$ the following $n$ ($\geq 1$) clauses, called definitions.

$$S_1: \text{newp}(X_1, ..., X_h) \leftarrow \text{Body}_1$$

$$...$$

$$S_n: \text{newp}(X_1, ..., X_h) \leftarrow \text{Body}_n$$

Such that:

1) newp is a new predicate symbol which does not occur in $P_0, ..., P_k$.
2) Each variable $X_i$ ($1 \leq i \leq h$) occurs in $\text{Body}_j$ for some $j$ ($1 \leq j \leq n$).
3) All predicates occurring in $\text{Body}_j$ for ($1 \leq j \leq n$) also occur in the initial program $P_0$.

Note that a variable occurring in $\text{Body}_j$ ($1 \leq j \leq n$) need not be in the set $\{X_1, ..., X_h\}$.

The new program $P_{k+1}$ and the set of new definitions $D_{k+1}$ are derived by adding the clauses $S_1, ..., S_n$ to $P_k$ and $D_k$ respectively. That is, $P_{k+1} = P_k \cup \{S_1, ..., S_n\}$ and $D_{k+1} = D_k \cup \{S_1, ..., S_n\}$.

In our approach, a new definition is introduced for each clause of program $P_i$ ($1 \leq i \leq r$) whose head predicate appears to have unused arguments. Each predicate that is introduced by the Definition Introduction Rule it is defined by one non-recursive clause whose body consists of precisely one atom.

3.2 Unfolding Rule

Let C be a clause in program $P_k$ of the form

$$C: H \leftarrow G_1, A, G_2$$

where A is an atom with a non-basic predicate, i.e. the predicate of A is not in the set {true, false, =} and $G_1$ and $G_2$ are (possibly empty) goals. Let $E_1, .., E_n$ with $n \geq 0$ be renamed clauses of $P_k$ such that $vars(C) \cap vars(E_i) \cap ... vars(E_n) = \emptyset$ and A is unified with the head of each $E_i$ ($i = 1, ..., n$) with most general unifier (mgu) $\theta$ i.e. $\theta_i = \text{mgu}(A, \text{hd}(E_i))$ where $\text{bd}(E_i)$ stands for the head of the clause $E_i$. Let $C_i$ ($i = 1, ..., n$) be a clause of the following form where $\text{bd}(E_i)$ stands for the body of clause $E_i$.

$$C_i: \{H \leftarrow G_1, \text{bd}(E_i), G_2\}.$$


Unfolding clause $C$ with respect to $A$ using clauses $E_{i_{1}}, \ldots , E_{i_{n}}$, we derive clauses $C_{1}, \ldots , C_{n}$. The atom $A$ is said to be the \textit{selected atom} for unfolding.

The new program $P_{k+1}$ is derived by replacing clause $C$ in $P_{k}$ with clauses $C_{1}, \ldots , C_{n}$, that is $P_{k+1} = P_{k} - \{C\} \cup \{C_{1}, \ldots , C_{n}\}$.

In our approach, the clauses which are going to be unfolded in the $i$th step of the transformation sequence are the program clauses of $P_{i}$ ($1 \leq i \leq r$) which had been introduced by the Definition Introduction Rule in the $(i-1)^{th}$ step of the transformation sequence.

### 3.3 Folding Rule

Let $C_{1}, \ldots , C_{n}$ be clauses in program $P_{k}$. Let $S_{1}, \ldots , S_{n}$ be clauses in $D_{k}$ which constitute the definition of predicate, say $\text{newp}$. That is, 

\begin{align*}
S_{1}: \text{newp}(X_{1}, \ldots , X_{k}) & \leftarrow \text{Body}_{1} \\
& \quad \ldots \\
S_{n}: \text{newp}(X_{1}, \ldots , X_{k}) & \leftarrow \text{Body}_{n}
\end{align*}

Suppose that there are two goals $G_{1}$ and $G_{2}$ and a substitution $\theta$ such that for $i=1, \ldots , n$ the following two conditions hold:

- Each clause $C_{i}(1 \leq i \leq n)$ is a variant of the clause $H \leftarrow G_{1}, \text{Body}_{i}\theta, G_{2}$. 
- For every variable $X$ in $\text{Body}_{i}$, $(1 \leq i \leq n)$ and $X \notin \{X_{1}, \ldots , X_{i}\}$, we have:
  - $X\theta$ is a variable which does not occur in $H$, $G_{1}$ and $G_{2}$.
  - For any variable $Y$ occurring in $\text{Body}_{i}$ $(1 \leq i \leq n)$, and different from $X$, i.e. $X \neq Y$, $X\theta$ does not occur in $Y\theta$.

Let $E$ be the clause $E : H \leftarrow G_{1}, \text{newp}(X_{1}, \ldots , X_{k})\theta, G_{2}$. Clause $E$ is derived by folding clauses $C_{1}, \ldots , C_{n}$ using clauses $S_{1}, \ldots , S_{n}$.

Let us assume that $P_{0}$ is a program that is constructed by the method in [1], [2] and $D_{0}$ is an empty set of new definitions. Suppose that predicate $p/m$ in program $P$ is defined by a set of clauses $C_{1}, \ldots , C_{k}$ as follows.

\begin{align*}
C_{1}: & \ p(X_{1}, \ldots , X_{j}, \ldots , X_{m}) \leftarrow \text{Body}_{1} \\
& \quad \ldots \\
C_{k}: & \ p(X_{1}, \ldots , X_{j}, \ldots , X_{m}) \leftarrow \text{Body}_{k}
\end{align*}

Initially, all predicates with "isolated" arguments and the corresponding isolated arguments of program $P_{0}$ are identified. For each predicate $p/m$ of program $P_{i}$ ($1 \leq i \leq r$) with "isolated" argument(s) the following three steps are performed.

#### Step 1: Definitions introduction.

Let’s assume that $X_{j}$ is an isolated argument in predicate $p(X_{1}, \ldots , X_{j}, \ldots , X_{m})$. A new clause, $C_{\text{new}}$, is introduced for predicate $p/m$ of the following form.

\begin{align*}
C_{\text{new}}: & \ p(X_{1}, \ldots , X_{j+1}, X_{j+1}^{1}, \ldots , X_{m}^{1}) \leftarrow p(X_{1}, \ldots , X_{j}, \ldots , X_{m})
\end{align*}

The derived program $P_{i}$ becomes $P_{i} = P_{0} \cup \{C_{\text{new}}\}$. The set of new definitions becomes $D_{i} = D_{0} \cup \{C_{\text{new}}\}$.

#### Step 2: Unfold.

The clause $C_{\text{new}}$ is unfolded using the definition of $p$, i.e. clauses $C_{1}, \ldots , C_{k}$. Clauses $C_{1}^{\text{new}}, C_{2}^{\text{new}}, \ldots , C_{k}^{\text{new}}$ are derived. That is,

\begin{align*}
C_{1}^{\text{new}}: & \ p(X_{1}, \ldots , X_{j+1}, X_{j+1}^{1}, \ldots , X_{m}^{1}) \leftarrow \text{Body}_{1}\theta_{1} \\
C_{2}^{\text{new}}: & \ p(X_{1}, \ldots , X_{j+1}, X_{j+1}^{1}, \ldots , X_{m}^{1}) \leftarrow \text{Body}_{2}\theta_{2} \\
& \quad \ldots
\end{align*}
The derived programs are equivalent to the initial ones due to the correctness properties of the fold/unfold transformations. That is, total correctness of the transformation rules for normal logic programs and total correctness of the fold/unfold sequence is shown in [8] and in [9].

Note that the removed unused arguments are variables. Compound terms in the programs that are constructed by the schema-based method in [1], [2] are confined in the data type operations which are either built-in or user defined. The data type operations do not have unused arguments. They are assumed to be well-designed and implemented efficiently.

5. Illustration of the Method with an Example

In the following example, the "isolated" ("unused") arguments are shown as a set of pairs. The first element of each pair consists of a predicate name and its cardinality and the second element consists of a list of "unused" arguments of the corresponding predicate. That is, \{(p₁/n₁, [unused arguments of predicate p₁]), ..., (pₖ/nₖ, [unused arguments of predicate pₖ])\}.

The predicate \text{sum}(X₁, X₂) is true if X₂ is the summation of the elements in list X₁. The following program P₀ for predicate \text{sum}/2 has been constructed by the method in [1], [2]. The definitions of the data type operations are not included because they are not used in this discussion. They are assumed as part of this program. The "isolated" ("unused") arguments are removed in first iteration.

C₁: \text{sum}(X₁, X₂) ← p₀(X₁) ∧ p₁(X₁, X₂) 
C₂: \text{sum}(X₁, X₂) ← p₀(X₁) ∧ p₂(X₁, X₃, X₄) ∧ \text{sum}(X₄, X₅) ∧ p₃(X₁, X₃, X₅, X₂) 
C₃: p₀(X₁) ← \text{empty_seq}(X₁) 
C₄: p₁(X₁, X₂) ← \text{neutral_add_subtr_int}(X₂) 
C₅: p₂(X₁, X₃, X₄) ← p₄(X₁, X₃, X₄) ∧ p₅(X₁, X₃, X₄) 
C₆: p₃(X₁, X₃, X₅, X₂) ← \text{plus_int}(X₃, X₅, X₂) 
C₇: p₄(X₁, X₃, X₄) ← \text{head}(X₁, X₃) 
C₈: p₅(X₁, X₃, X₄) ← \text{tail}(X₁, X₄)
program because they are part of the program. The implementations of the data type operations for this program are the following.

- empty_seq([])
- neutral_add_subtr_int(0)
- head([H|T], H)
- tail([H|T], T)
- plus_int(X1, X2) ← X3 is X1 +1

Program $P_0 = \{C1, C2, C3, C4, C5, C6, C7, C8\}$.

New Definitions $D_0 = \{}$

### 1st Iteration

**Step 1: New definitions introduction.**

The "isolated" arguments for program $P_0$ are found out. They are shown in the following set $S = \{(p1/2, [X1]), (p3/4, [X1]), (p4/3, [X4]), (p5/3, [X3])\}$.

New definitions are introduced for the predicates which have "isolated" arguments. That is,

- $C9: p1(X1,X2) ← p1(X1,X2)$
- $C10: p2(X3,X5,X2) ← p3(X1,X3,X5,X2)$
- $C11: p3(X1,X3) ← p4(X1,X3,X4)$
- $C12: p4(X1,X4) ← p5(X1,X3,X4)$

Derived program: $P_1 = P_0 \cup \{C9, C10, C11, C12\} = \{C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12\}$.

New definitions: $D_1 = D_0 \cup \{C9, C10, C11, C12\} = \{C9, C10, C11, C12\}$.

### Step 2: Unfold.

Clauses $C9$, $C10$, $C11$ and $C12$ are unfolded on literals $p1(X1,X2)$, $p3(X1,X3,X5,X2)$, $p4(X1,X3,X4)$ and $p5(X1,X3,X4)$ respectively by using the corresponding definitions of predicates $p1/2$, $p3/4$ and $p4/3$ and $p5/3$, i.e. clauses $C4$, $C6$, $C7$ and $C8$. Note that clauses $C4$, $C6$, $C7$ and $C8$ have to be clauses from $P_0 \land Def_0$.

- $C13: p1(X2') ← neutral_add_subtr_int(X2')$ unfold($C9$, $C4$)
- $C14: p2(X3', X5', X1') ← plus_int(X3',X5',X2')$ unfold($C10$, $C6$)
- $C15: p3(X1', X3') ← head(X1',X3')$ unfold($C11$, $C7$)
- $C16: p4(X1', X4') ← tail(X1',X4')$ unfold($C12$, $C8$)

Derived program: $P_2 = P_1 \cup \{C9, C10, C11, C12\} \cup \{C13, C14, C15, C16\} = \{C1, C2, C3, C4, C5, C6, C7, C8, C13, C14, C15, C16\}$

New definitions: $D_2 = D_1$

### Step 3: fold.

The calls of $p1/2$, $p3/4$, $p4/3$, $p5/3$ are folded using the clauses of new definitions $C9$, $C10$, $C11$, and $C12$ respectively.

- $C17: sum(X1, X2) ← p0(X1) \land p1(X2)$ fold($C1$, $C9$)
- $C18: sum(X1, X2) ← p0(X1) \land p2(X1,X3,X4) \land sum(X4,X5) \land p2(X3,X5,X2)$ fold($C5$, $C11$)
- $C19: p2(X1,X3,X4) ← p3(X1,X3) \land 5(X1,X3,X4)$ fold($C19$, $C12$)
- $C20: p2(X1,X3,X4) ← p3(X1,X3) \land p4(X1,X4)$

Derived program: $P_3 = P_2 \cup \{C17, C18, C19\} - \{C1, C2, C5\} - \{C19\} \cup \{C20\} = \{C3, C4, C6, C7, C8, C13, C14, C15, C16, C17, C18, C20\}$

New definitions: $D_3 = D_2$

### Step 4: Deletion of "useless" clauses.

The useless clauses are the ones that have been used in the unfold step. i.e. $\{C4, C6, C7, C8\}$. There is not any call in the program to these clauses.

- $C4: p1(X1,X2) ← neutral_add_subtr_int(X2)$
- $C6: p3(X1,X3,X5,X2) ← plus_int(X3,X5,X2)$
- $C7: p4(X1,X3,X4) ← head(X1,X3)$
- $C8: p5(X1,X3,X4) ← tail(X1,X4)$

Derived program: $P_4 = P_3 - \{C4, C6, C7, C8\} = \{C3, C13, C14, C15, C16, C17, C18, C20\}$

New definitions: $D_4 = D_3$

The program $P_4$ is the derived one from the first iteration of the transformation process. This iteration is also the last one for this example because program $P_4$ does not have "isolated" arguments. i.e. $P_4 = P_{final}$.

Program $P_{final} = P_4$:

- $C3: p0(X1) ← empty_seq(X1)$
- $C13: p1(X2) ← neutral_add_subtr_int(X2)$
- $C14: p2(X3,X5,X2) ← plus_int(X3,X5,X2)$.
- $C15: p3(X1,X3) ← head(X1,X3)$
- $C16: p4(X1,X4) ← tail(X1,X4)$
- $C17: sum(X1,X2) ← p0(X1) \land p1(X2)$
- $C18: sum(X1,X2) ← p0(X1) \land p2(X1,X3,X4) \land sum(X4,X5) \land p2(X3,X5,X2)$
- $C20: p2(X1,X3,X4) ← p3(X1,X3) \land p4(X1,X4)$

### 6. Conclusions and Future Research

Our method has the following advantages:

- This method is automatic. The predicates to be transformed are automatically identified. No decision is required by the programmer which predicates to transform.
- This method does not assume any knowledge of which arguments are unused. It just relies on the syntactic property of "isolated" argument. Therefore, it avoids the problem of deciding which call to unfold.
- It removes the unused arguments from all literals, whether they are negated or not.
- All unused arguments are removed. The derived programs are equivalent to the initial ones due to the correctness properties of the fold/unfold transformations.
• The "useless" clauses are identified easily without computational overhead. That is, it is not necessary to apply any abstract interpretation technique in order to discover them.
• The derived programs are more efficient than the initial ones.

Partial evaluators for logic programs are a class of meta-programs which have been used mainly for optimizing logic programs [13]. They have been used as well in the construction of logic programs. Partial evaluation supports program development methodologies based on data abstraction and on procedural abstraction by removing layers of procedure calls and by propagating data structures [14]. The derived programs by this method can be further transformed by applying to them partial evaluation techniques in order to remove their structure. This structure is created due to the semi-automation of the constructed method [1], [2]. A partial evaluation tool will propagate the data structures from the data type operations to predicates in higher levels in the program. The data structures are confined within data type operations in the programs constructed by the method in [2]. In addition, the data type operations are at the bottom of the structure of the constructed programs.

Acknowledgements

This study is supported by the project "Funding of research groups at Technological Educational Institute of Crete" of the program Archimedes II. This project is co-funded by the European Social Fund and National Resources.

References:


