MANAGEMENT OF INFORMATION ON PROGRAM FLOW ANALYSIS

Margarita Knyazeva, Dmitry Volkov

Abstract: The article proposes the model of management of information about program flow analysis for conducting computer experiments with program transformations. It considers the architecture and context of the flow analysis subsystem within the framework of Specialized Knowledge Bank on Program Transformations and describes the language for presenting flow analysis methods in the knowledge bank.

Keywords: Knowledge bank; Ontology; Knowledge base; Ontology editor; Database editor; Flow analysis; Editor of flow analysis methods

ACM Classification Keywords: I.2.5 Artificial intelligence: programming languages and software

Introduction

The impossibility of carrying out computer experiments opportunely constitutes the main problem of program optimization science. Their goal is to determine how often transformations can be applied in real programs, what effect can be achieved, and what strategy is the best to be applied for the specified set of optimizing transformations. At present, optimizing compilers are the only means of conducting such experiments [Bacon, 1994] [GNU, 2007]. However, the period between the moment when a new transformation description is published and the moment when the realization of an optimizing compiler containing this transformation (if such a compiler is being developed) ends is so long that the results of computer experiments with this transformation appear to be out-of-date. Besides, an optimizing compiler usually contains a wide set of transformations and built-in strategy of their application so it is impossible to obtain reliable results of computer experiments related to a particular transformation (not to the whole set) or other strategy.

The absence of tools for conducting experiments results in transformations and transformation application strategies, whose characteristics are not known completely, being included in optimizing compilers. This adversely affects their making. Therefore to create a system for program transformation experiments aimed to solve the above-mentioned problems is a topical issue. Artificial intelligence methods applied in program transformations serve as a basis for this system.

Based on the results of the paper [Orlov, 2006], the paper [Kleshchev, 2005] proposes Specialized Knowledge Bank on Program Transformations (SKB_PT) as the concept of program transformation information management to solve scientific, practical and educational problems in the sphere of program transformations. This article proposes the model of management of information about knowledge-managed program flow analysis that is a tool of getting reliable information about program performance without its execution in the program transformation system in SKB_PT. The multipurpose computer knowledge bank is used as the general concept within the framework of which the program transformation system is realized with the knowledge-managed program flow analysis [Orlov, 2006] (http://www.iacp.dvo.ru/es/mpkbank).

The paper has been financially supported by the Far Eastern Branch of the Russian Academy of Sciences, initiative-based research project “Internet system for controlling information about program transformations”.

Concept of knowledge-managed program flow analysis

The extraction of certain semantic characteristics of a program takes place during flow analysis that is traditionally divided into control flow analysis and data flow analysis [Kasyanov, 1988] [Voevodin, 2002].

The main task of control flow analysis is to present and structure sets of program executions, to find characteristics of statements and branches in these executions, to choose an order of program statements processing. During data flow analysis each program is executed in parallel over all values from a symbolic and very simplified version of its real data area.

Let us consider the architecture of the subsystem of the knowledge-managed flow analysis within the framework of the program transformation system (fig. 1).

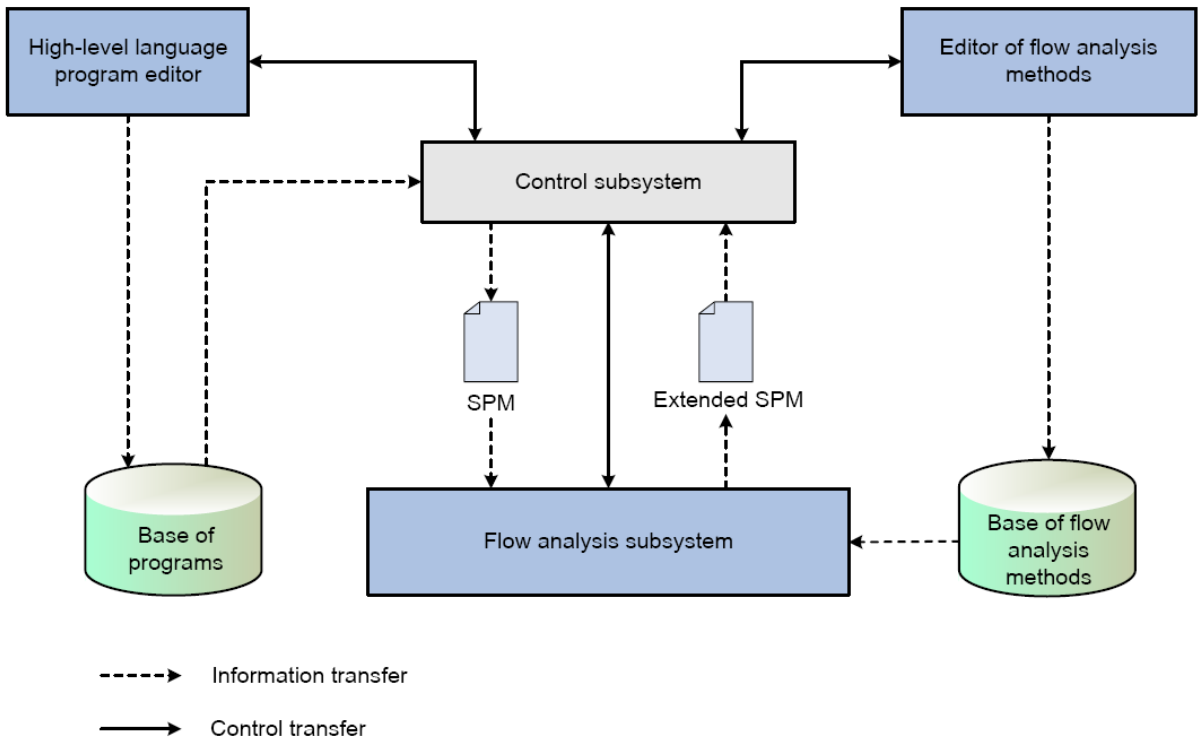

Fig. 1. Architecture of subsystem of knowledge-managed flow analysis

Structural program model (SPM), flow analysis methods and task to do a program flow analysis are input data of a knowledge-managed flow analysis subsystem in the system of program transformation. SPM extended with the terms of the program flow analysis is formed at the output of the subsystem.

Structural program model defined in [Knyazeva, 2005a] is a single internal presentation at which the program flow analysis takes place. It is presented as a graph. Extended SPM is control and information graphs of the program [Knyazeva, 2005b]. To extend SPM is to add special control arcs to the program presentation and enter new program fragments which result from the program flow analysis into SPM. Extended SPM is the basis for program transformations. Functions and relationships assigning some program characteristics are defined on a set of fragments and identifiers of the program. Functions that have one argument are called attributes.

In order to apply new flow analysis methods in experiments, the flow analysis subsystem gives the user the opportunity to exploit a specialized language, that describes flow analysis methods (FAM), to assign methods of program flow analysis.

The task to do a flow analysis is a description of the knowledge out of the whole volume of the knowledge about flow analysis methods that are to be applied in this situation.

Source data are entered into the corresponding databases by means of High-level language program editor and Editor of flow analysis methods. Control subsystem provides the interaction between the flow analysis subsystem, program transformation system and data sources.

The base of programs contains high-level language programs in terms of language ontologies.

The base of flow analysis methods contains flow analysis methods in the language of flow analysis methods.

Language of flow analysis methods

Main forms of notation of flow analysis methods described in relevant works were analyzed when the language being developed. It contains variables that may take on references to various elements of the program model as values; basic constructions of algorithmic programming languages (such as loop, selection, assignment); operations with sets as variable sets and identifier sets are operated on while the information about the program is being accumulated; tree walk operations and operations with tree structures.

The syntax of the language of flow analysis methods is described in extended BNF notation:

<Flow analysis method>::= " Flow_analysis_method" "(" <Method name> ")" <Variable declaration block> <Sequence of constructions>

< Method name >::=<String>

<String>::=<Letter> | <String> < Letter > | <String> <Digit>

< Letter >::=A | ... | Z | a | ... | z| - | _ |

< Digit >::=0 | 1 … | 9

<Sequence of constructions>::= "{" [<Construction>] "}"

< Variable declaration block >::= [<Variable declaration >]

< Variable declaration >::= <Variable type> ":" (<Variable-fragment> | <Variable-attribute> | <Variable-arc> | <Variable-relation> | <Variable> [","]) ";"

<Variable type>::="Variable-fragment" | "Variable-attribute" | "Variable-arc" | "Variable-relation" | "Integer" | "Real"

<Construction>::=<Formula> | < Walk> | <Selection> | <Loop> | <Assignment> | <Program modification>

<Formula>::=<Formula with fragment> | <Formula with set> | <Logical formula>

< Walk>::=<Program tree walk> | <Expression tree walk>

<Selection>::="If" "(" <Logical formula> ")" "Then" <Sequence of constructions> ["Else" <Sequence of constructions>]

<Loop>::="While" <Condition> <Sequence of constructions>

<Assignment>::= <Left part of assignment> "=" <Right part of assignment>

<Program modification>::= <Fragment creation> | <Attribute creation> | <Attribute change> | <Arc creation> | <Relation creation> | <Variable creation>

<Formula with fragment>::= <Arc fragment> | <Fragment attribute> | <To get class> | <To get expression variable> | <First arc fragment of sequence> | <Next arc fragment of sequence>

<Arc fragment>::= " Arc_fragment" "("<Variable-fragment>, <Arc name>, <Variable-fragment> ")"

<Fragment attribute>::= “Fragment_attribute” “(” <Variable-fragment>, <Attribute name>, <Variable-attribute> “)”

<To get class>::= “To_get_class” “(” <Variable-fragment>, <Fragment class>“)”

<To get expression variable>::= “To_get_expression_variable” “(” <Variable-fragment>, <Variable> “)”

<First arc fragment of sequence>::= “ First_arc_fragment_of_sequence” “(” <Variable-fragment>, <Variable-fragment> “)”

<Next arc fragment of sequence>::= “Next_arc_fragment_of_sequence” “(” <Variable-fragment>, <Variable-fragment>, <Variable-fragment> “)”

<Formula with set>::= <Intersection of sets> | <Union of sets> | <Equality of sets>

<Intersection of sets>::= “Intersection_of_sets” “(” <Variable-set> <Variable-set> < Variable-set> “)”

<Union of sets>::=“ Union_of_sets” “(” <Variable-set> <Variable-set> <Variable-set> “)”

<Equality of sets>::= “ Equality_of_sets” “(” <Argument-set> <Variable-set> <Boolean-set> “)”

<Logical formula>::=<Term of logical formula>

<Compound logical formula>::= <Term of logical formula> <Logical operator> <Term of logical formula>

<Term of logical formula>::=<Compound logical formula> | <Boolean set> | < Equality of sets> | <Fragment class> | <Arc name> | <Variable-fragment> | <Variable-attribute> | <Variable-arc> | <Variable-relation> | <Attribute name> | <Relation name> | <Variable>

<Logical operator>::= “>” | “<” | “>=” | “<=” | “<>” | “==” | “AND” | “OR” | “NOT”

<Program tree walk>::=“Program_tree_walk” “(“ <Variable-fragment>, <Variable-fragment>, <Logical formula> “)” <Sequence of constructions>

<Expression tree walk>::=“Expression_tree_walk” “(“<Variable-fragment>, <Variable-fragment> “)” <Sequence of constructions>

<Program modification>::=<Fragment creation> | <Attribute creation> | <Arc creation> | <Relation creation> | <Variable creation>

<Fragment creation>::=“To_create_fragment” “(“ <Variable-fragment>,<Fragment class> “;”

<Variable-fragment> “)”

<Attribute creation>::= “To_create_attribute” “(“ <Variable-fragment>, <Attribute name>, <Variable-attribute> “)”

<Arc creation>::=“ To_create_arc” “(“ <Variable-fragment>, <Variable-fragment>, <Arc name>, <Varaiable-arc> “)”

<Relation creation>::= “To_create_relation” “(“ <Variable-fragment>, <Variable-fragment>[, <Variable-fragment><Variable-relation> “)”

<Value>::=<Integer> <Real> <Boolean set>

<Integer>::=(<Digit>)

<Real>::=(<Digit>)[,(<Digit>)]

<Assignment>::=<Left part of assignment> = <Right part of assignment>

<Left part of assignment>::=<Variable-fragment> | <Variable-attribute> | <Variable-arc> | <Variable-relation> | <Variable>

<Right part of assignment>::=<Variable-fragment> | <Variable-attribute> | <Variable-arc> | <Variable-relation> | <Variable> | <Value> | <Arithmetic expression>

<Arithmetic expression>::=<Term of arithmetic expression> <Arithmetic operator> <Term of arithmetic expression>

<Term of arithmetic expression>::=<Arithmetic expression> <Variable> <Bracketed arithmetic expression> <Variable value>

<Arithmetic operator>::= “+” | “- ” | “* ” | “/ ” | “^”

<Fragment class>::=“Variable_declaration” | “Function_declaration” | “Parameter_declaration” | “Variables_declaration” | “Functions_declaration” | “Parameters_declaration” | “Assignment” | “Input” | “Output” | “Program_block” | “Conditional_statement” | “Loop_with_step” | “Loop_with_precondition” | “Loop_with_postcondition” | “Procedure_call” | “Dynamic_variable_elimination” | “Expression” | “Sequence_of_statements”

<Attribute name>::=“Reverse_Polish_notation” | “Result_array” | “Pointer” | “Function_recursive” | “Side_effect” | “Reference_to_memory_space” | “Nesting_level” | “Priority” | “Type” | “Reference_parameters” | “Value_parameters” | “Actual_reference_parameters” | “Changeable_actual_reference_parameters” | “Actual_value_parameters” | “Argument_set” | “Result_set” | “Obligatory_result_set” | “Function_declaration_statement” | “Contiguous_sequence_of_fragments” | “Classes_of_fragments_of_sequences” | “Quantity_of_fragments” | “Result_identifier” | “Pseudovariable” | “Design_of_new_types” | “Acceptable_left_expression”

<Arc name>::=“If” | “Then” | “Else” | “Condition_of_loop” | “For” | “Until” | “Step” | “Statement_body” | “Parameter_block” | “Local_parameter_block” | “Embedded_function_block” | “Right_expression” | “Left_expression” | “First_element_of_sequence” | “Lasy_element_of_sequence” | “Arc_statement_sequence” | “Matches_fragments” | “Next_fragment” | “Parameter_list”

<Relation name>::=“Immediate_precedence” | “Precedence” | “Similarity” | “To_be_part” | “To_be_submodel” | “Precedence_of_submodels” | “Joint_sequence” | “Intermediate_sequence” | “Preceding_sequence” | “Next_sequence”

<Boolean-set>::=“true” | “false”

<Variable>::=<String>

<Variable-set>::=<String>

<Variable-fragment>::=<String>

<Variable-attribute>::=<String>

<Variable-arc>::=<String>

<Variable-relation>::=<String>

Example of presenting flow analysis method in FAM language

Context conditions for transformations are described either in terms of a program model or in terms derived from them. The model is to semantically ensure formulating of the context of the current transformation. The justification for the transformation lies in proving the theorem that context conditions for transformations are sufficient conditions for functional equivalency of transformed and source program models [Pottosin, 1980].

The enclosure of any optimizing transformation into the compiler assumes simultaneous forming of the transformation and context condition; provided the condition is met, the given transformation is applied to the program.

This can be exemplified by argument set that is a set of variables the values of which may affect a statement performance in the program. The information about argument sets of statements is made use of in optimizing transformations “unused variable elimination”, “loop invariant statement removal” and others [Bacon, 1994]. The correct selection of an optimization area in a source program and the transformation efficiency on the whole depend on the flow analysis quality.

Method of copying argument set of each program statement into result set of program in FAM language:

```
Flow_analysis_method(Copying_of_sets)
Type-fragment: Current_fragment;
Type-attribute: Temporary_attribute;
{
Program_tree_walk(Function_Main; Current_fragment; Fragment_class(Current_fragment) == Assignment){
Fragment_attribute(Current_fragment; Argument_set; Temporary_attribute);
Attribute_creation(Current_fragment; Result_set; Temporary_attribute);
}
}
```

The first string is as follows:

```
Flow_analysis_method(Copying_of_sets)
```

The first sentence in the FAM language starts with the key word Flow_analysis_method that is followed by the flow analysis method name in parenthesis.

The section declaring variables follows:

```
Type-fragment: Current_fragment;
Type-attribute: Temporary_attribute;
```

In this example there are two variables described: the first one has Type-fragment type and is called Current_fragment, the second one has Type-attribute type and is called Temporary_attribute. Type-fragment variable type means that this variable may take on a reference to a fragment of a particular program on SPM or an object in the memory that reflects all characteristics of SPM fragment. The declarations of variables of different types are separated with semicolons. If there are declarations of several variables of one type, they can be separated with commas.

The method body immediately follows the variable declarations and consists of a sequence of constructions:

```
{
Program_walk_tree(…)
{
Fragment_attribute(…);
Attribute_creation(…);
}
}
```

The method body is in braces. The inside constructions are separated with semicolons. In this example, the method body consists of one Program_walk_tree construction which consists of two constructions: Fragment_attribute and Attribute_creation.

Program_walk_tree construction is a function with three arguments that follows the key word. They are in parentheses and separated with a semicolon and the body in braces:

Program_walk_tree (Function_Main; Current_fragment; Fragment_class(Current_fragment) == Assignment)

{

…}

This function realizes SPM fragments tree walk. The first argument specifies SPM fragment which is the root the subtree that is to be walked. The second argument is a variable that takes on the fragment value at the next walk step. The third argument is a logical formula whose verity ensures the execution of the body constructions sequence. In this example, the first argument is Function_Main constant the value of which is a reference to SPM root fragment. The second argument is Current_fragment variable that takes on the next fragment value at each walk step. The third argument is a logical formula. It takes on the verity value if SPM fragment, Current_fragment refers to, has Assignment class.

The construction Fragment_attribute is a function with three arguments:

Fragment_attribute(Current_fragment, Argument set, Temporary_attribute);

The first argument is SPM fragment Current_fragment variable refers to. The second argument is the name of SPM argument that is necessary to get. The third argument is a Type-attribute variable which is the result of the function and takes on the value of the reference to the specified attribute of the current fragment.

Fragment_creation construction is a function with three arguments:

Fragment_creation(Current_fragment, Result_set, Temporary_attribute);

This function creates the attribute with the specified name and value for the specified fragment. The first argument is SPM fragment Current_fragment variable refers to. The second argument is the name of SPM attribute that is necessary to create; in this case it is Result_set. The third argument is a Type-attribute variable whose value is to be copied for a newly-created attribute.

Conclusion and Acknowledgements

This paper presents the knowledge-managed flow analysis concept. It provides examples how various flow analysis methods can be defined by means of the described language. At present, based on the knowledge-managed flow analysis concept, the flow analysis subsystem within the framework of the program transformation system in SKB_PT is developed.

Bibliography

[Bacon, 1994] Bacon D.F., Graham S.L., Sharp O.J. Compiler transformations for high-performance computing //ACM Computing Surveys 1994 V.26 № 4. PP.345-420/

[GNU, 2007] GNU Compilers Collection 3.3.2. http://gcc.gnu.org/onlinedocs/gcc-3.3.2/gcc/

[Kasyanov, 1988] Kasyanov V. N. Optimizing transformations of the programs. Moskow: Nauka, 1988. 336 p. (In Russian).

[Kleshchev, 2005] Kleshchev A.S., Knyazeva M.A. Controlling Information on Program Transformations: I. Analysis of Problems and Ways of Their Solution with methods of Artificial Intelligence. Journal of Computer and Systems Sciences International, Vol.44, No5, 2005, pp. 784-792.

[Knyazeva, 2005a] Knyazeva M.A., Kupnevich O.A. Domain ontology model for the domain "Sequential program optimization". Defining the language of structural program model. In The Scientific and Technical Information, Ser. 2.-2005.-№ 2.-P. 17-21. (In Russian).

[Knyazeva, 2005b] Knyazeva M.A., Kupnevich O.A. Domain ontology model for the domain "Sequential program optimization". Defining the extension of the language of structural program model with flow analysis terms. In The Scientific and Technical Information, Ser. 2.-2005.-№ 4. (In Russian).

[Orlov, 2006] Orlov V.A., Kleshchev A.S. Computer banks of knowledge. Multi-purpose bank of knowledge. In The Information Technologies. 2006. №2. P.2-8. (In Russian).

[Pottosin, 1980] Pottosin I.V., Yugrinova O.V. Justification for purging transformations for loops. In The Programming 1980. - №5. - P.3 - 16. (In Russian).

[Voevodin, 2002] Voevodin V.V., Voevodin Vl.V. Parallel computing. Saint Petersburg: BHV-Pereburg, 2002. 608 p. (In Russian).

Authors' Information:

Margarita A. Knyazeva, Dmitry A. Volkov - Institute for Automation & Control Processes, Far Eastern Branch of the Russian Academy of Sciences, 5 Radio Street, Vladivostok, Russia mak@nt.pin.dvgu.ru, vd2000@mail.ru