UDK 004.43

Programming operational semantics of programming languages

Anureev I. S. (A.P. Ershov Institute of Informatics Systems SB RAS)

The paper presents a method for describing the operational semantics of programming languages. It is based on a domain-specific language designed to specify executable specifications for programming language constructs. This language is an extension of the Lisp language. The peculiarity of the method is to set the semantics for the program model, and not for the program itself. Since the specifications are executable, the method actually allows to 'program' the semantics of programming languages. The method can be used in teaching computer languages, as well as in creating new programming languages, since it allows you to describe the constructions of a new language immediately at the level of an abstract syntactic tree.

Keywords: programming language, operational semantics, domain-specific language, abstract syntax tree, program model, Lisp

1. Introduction

One of the key methods to ensure the reliability of software is the use of formal methods. However, such use requires a formal definition of the program. One of these natural definitions is the operational semantics of programming languages.

The existing definitions of operational semantics can be divided into three groups. The first group consists of semantics presented manually in a natural or mathematical language. There is a huge list of works of this kind for specific programming languages. Here ([12], [10], [3], [8], [7], [4], [5], [17], [2], [6], [16], [13]) are some of them. The second group consists of semantics represented by program executors (implementations of programming languages). The third group consists of semantics encoded in formal machine systems ([15], [1], [18], [9], [14]).

All of these ways of representing operational semantics have a number of common disadvantages. First, even a small modification of the semantics requires efforts to avoid making mistakes when editing the representation. With a significant change in semantics, it is necessary to rewrite a significant part of its representation. Secondly, such representations of semantics cannot provide an understandable and readable general structure at the same time as the completeness of the analysis of all cases. Thirdly, these representations are not univer-

sal, capable of accurately preserving the structure of programs of a formalized programming language, in particular, due to the fact that the structure of the representation is rigidly defined. Fourth, these representations are practically unsuitable for describing the semantics of programs composed of constructs of several programming languages (A way of combination can be found in [11]). Fifth, these representations are difficult to adapt to a subset of the same language, given the specifics of this subset, which makes it possible to simplify the general semantics.

We present metalanguage approach to the development of formal semantics of programming languages which is able to largely cope with these disadvantages.

2. Metalanguage approach

The proposed approach has the following features:

1. It applies to program models, not to programs themselves.
2. It is fine-grained, i. e. for each type of program constructs it has its own type of models and, thus, can be applied in isolation to individual program constructs, building their operational semantics.
3. It is flexible, since it allows to develop operational semantics only for a fragment of a target programming language represented by a variety of acceptable types of program constructs or to develop a family of different operational semantics for the same target programming language.
4. Models of program constructs are presented in a unified way in a metalanguage of description of models.
5. The metalanguage also allow to present models of execution contexts of these constructs.
6. The metalanguage has the means to transform these models, thus allowing develop operational semantics as a sequence of transformations of models of program constructs and their execution contexts and reducing process of development of formal semantics of programming languages to writing programs in a metalanguage that implements corresponding sequences of transformations.
7. The model transformation means make operational semantics of these constructs executable.
8. This approach can be applied to programs that use combinations of several programming languages.

2.1. Metalanguage Design

Within the framework of the proposed metalanguage approach, the metalanguage language PSML, a syntactic extension of Common Lisp, was designed.

PSML contains one extra type *mt*. It describes a finite set of model languages, as well as a set of operations for each language from this set. A model language specifies either models of programming language constructs, or models of execution contexts of these constructs.

This type is extensible, which means that its contents, which include many languages, many types of constructs for each of these languages, and many value constructs for these types, can be expanded by special PSML-functions. Let us list these functions.

Let letters s, o, os (possible with indices) denote Lisp symbols, objects and object sequences, respectively. The initial empty content of type *mt* is set by function (*mt-empty*). Function (*mt-lan* s) adds a new model language s to type *mt*. Function (*mt-clan* s) makes the model language s the current language. Model execution is always performed in the context of the current model language.

Let l, t, and c denote a model language, type of models, and value constructor of a type of models, respectively.

Function (*mt-type* (c t) os_1 $*$ os_2 $**$ os_3) adds a new type t of models and value constructor c for this type to the current model language. Sequences os_1, os_2, and os_3 are called positional, attributing, and tagging components and specify three kinds of arguments of c called positional, attributing and tagging arguments, respectively.

Component os_1 specifies positional arguments of c. Its elements have the form (s v), where v specifies type and optional property of a positional argument, and s specifies an argument name (called a positional attribute), allowing access to the argument not only by its position, but also by name of this attribute. Object v can have one of the following forms t', (t' l), (TC t'), (TC (t' l)), (*opt* t'), (*opt* (t' l)), (*opt* (TC t')), and (*opt* (TC (t' l))) where s specifies argument type, i. e. the type of models that it can take as a value, l indicates that type t' is a type of models of language l, $CT \in \{list, map\}$ *list* indicates that values of the argument are lists of models of type t, *map* indicates that values of the argument are are finite mappings from symbols to type t, and *option* indicates that this argument is optional and can be skipped. If $l = lisp$, t' is a built-in lisp type name (for example, *symbol*).

Component os_2 specifies named arguments of c that are used to annotate models of type c. Its elements have the form (s v), where s specifies an argument name (called an annotating

attribute), and v specifies type of the argument and has the same form as in component os_1 except for the optional property.

Component os_3 specifies tagging arguments of c that are used to tag models of type c. Its elements have the form s, where s is called a tag.

Function (*mt-utype* t (t_1 ... t_n)) adds a new type t of models which is a union of types t_1, ..., t_n. Let us note that this type does not have its own value constructor.

Function (*mt-ltype* t s) adds a new type t of models which values match the values of the Lisp type s.

PSML has functions of access to contents of mt and attribute update in it.

Function (*mt-aget* m a) returns the value of attribute a of model m. Function (*mt-aset* m a o) assigns value o to attribute a of model m. These functions have generalized forms (*mt-aget* m a_1 ... a_n) and (*mt-aset* m a_1 ... a_n o) that allow you to access the value and assign the value according to the hierarchy of attributes a_1, ..., a_n, respectively.

Functions (*mt-type* m) and (*mt-aname* p) return a type of model m and a name of positional attribute in position p.

Function (*mt-is-model* m) checks whether m is a model of the current model language. Function (*mt-clan-get* m) returns the current model language.

Function (*mt-lans*) returns a current list of model languages. Function (*mt-mtypes* l) returns a list of types of models of language l. Function (*mt-mtypes*) returns a list of types of models of the current language.

Function (*mt-function* f ((l t x) y) o) specifies actions given by body o for function f of two arguments, model x of type t of construct language l and execution context y. If l is omitted, the current language is assumed.

There are variants of the above functions wit prefix *mt-g-* instead of *mt-*. These variants have additionally a model language as the first argument.

2.2. Approach Steps

The application of the proposed approach consists of the following four steps:

1. *To program the content of type mt corresponding to the target programming language L (its program constructs and execution contexts) in PSML.*
2. *To develop a translator of L-constructs to their primary models.* A primary model of a program construct is a model that directly represents the construct without adding

additional information (for example, as result of semantic analysis). A primary model corresponds an AST of the construct obtained as a result of syntactic analysis, with one exception that this tree is additionally semantically marked with attributes and concepts (types of models) within the terminology of the subject area (a programming language). This is the only step that is performed by an external tool (for example, a parser generator), but because of one-to-one translation, it is simple.

3. *To program a translator of primary models of L-constructs to their secondary models in PSML.* A secondary model is a result of enriching the primary model of a program construct with additional information. This result is achieved by adding new annotating attributes to the construct and their meanings, as well as adding new tags. At the same time, the positional component of the construct does not change, just as the old annotating attributes and tags do not change. Enriching program construct models with additional information allows to simplify operational semantics development. Specialized external tools (for example, static analysis tools) can be used to execute this step.
4. *To specify function op-sem for each program construct model in PSML.* Let us note that although the full power of Common Lisp can be used in defining such a function, experiments show that it is possible to identify a limited subset of functions sufficient to describe the semantics of typical programming languages, thus defining DSL (Domain-Specific Language) for the operational semantics development.

2.3. Example

In this section the proposed approach is applied to a 'programming language' consisting exactly of the constructs contained in the program fragment below.

```
int a = 1; int b = 2; int c; int* p = &c;
asm(".intel_syntax noprefix\n\t" // GAS directive
  "mov eax, %1\n\t"
  "add eax, %2\n\t"
  "mov %0, eax\n\t"
  :"=r"(c)
  :"r"(a), "r"(b)
  : "eax"
);
```

This fragment denoted further by PF illustrates combination of two languages: C and Assembler.

In the first step, the content of type *mt* is set. Due to lack of space, this type will immediately contain the specification of the primary and secondary models, while attributes and tags related to the secondary model will be highlighted in bold.

```
(mt-empty) ; innitial intialization of type mt
(mt-lan C) ; adding model language C for C–constructs
; specifying model types for C language
(mt-clan C) ; making C current
(mt-type (seq declaration-sequence) (seq (list declaration))
(mt-utype expression (integer variable derefence addressof))
(mt-ltype integer int)
(mt-ltype variable symbol)
(mt-type (* derefence) (pointer (symbol name)))
(mt-type (& addressof) (variable (symbol name)))
(mt-utype declaration (variable-declaration asm-insert))
(mt-type (var variable-declaration) (type type) (name (symbol lisp))
     (value (opt expression)) * (always-points-to-var (symbol lisp))
     ** non-shared )
(mt-type (asm asm-insert-declaration) (code Assembler))
(mt-utype type (simple-type pointer-type))
(mt-ltype simple-type symbol)
(mt-type (pointer pointer-type) (type type))
; specifying model types for Inline Assembler language
(mt-clan Assembler) ; making Assembler current
(mt-type (insert insert) (GAS (string lisp)) (code (list instruction))
     (outputs (list binding)) (inputs (list binding))
          (destructed (list (symbol lisp))) )
(mt-type (binds binding) (register-type (symbol lisp)) (var (symbol lisp)))
(mt-utype instruction (mov add))
(mt-type (mov mov) (first operand) (second operand))
(mt-type (add add) (first operand) (second operand))
```

```
(mt-utype operand (simple-operand index-operand))
(mt-ltype simple-operand symbol)
(mt-type (& index-operand) (index (nat lisp)))
```

After completing the second and third steps, the following secondary model of fragment PF is obtained:

```
(seq (var int a 1 ** non-shared) (var int b 2 ** non-shared) (var int c 2)
    (var (pointer int) p (& c) * (always-points-to-var c))
    (asm (insert (GAS ".intel_syntax noprefix")
        (code (mov eax (% 1)) (add eax (% 2)) (mov (% 0) eax))
        (outputs ((binds (register-type r) (var c))))
        (inputs ((binds (register-type r) (var a))
            (binds (register-type r) (var b))))
        (destructed (eax)))))
```

Tag *non-shared* specifies variables values of which can be accessed in fragment PF only through their names. Annotating attribute *always-points-to-var* specifies pointers that always point to the same variable and returns this variable. From the arrangement of these entities in this model, it follows that a simple memory model without addresses can be used in the execution context for PF.

The language *C-context* specifies the execution context for C constructs of PF and is defined as follows:

```
(mt-lan C-context) ; adding C-context language
; specifying model types for C-context language
(mt-clan C-context) ; making C-context current
(mt-type (context context) * (var-values (map (object lisp)))
    (var-types (map type)) (pointers (map (symbol lisp))) )
```

Thus, this context stores the values of variables in annotating attribute *var-values* and the variables through which pointers get values in annotating attribute *pointers*. An example of model of language *C-context* is given below:

```
(context * (var-values (map (a 1) (b 2) (c 3)))
    (var-types (map (a int) (b int) (c int) (p (pointer int))))
    (pointers (map (p c))) )
```

The language *Assembler-context* specifies the execution context for Assembler constructs of PF and is defined as follows:

```
(mt-lan Assembler-context) ; adding Assembler-context language
; specifying model types for Assembler-context language
(mt-clan Assembler-context) ; making Assembler-context current
(mt-type (context context) * (reg-values (map (object lisp)))
    (index-values (list (object lisp)))
    (free-GP-registers (list (symbol lisp))) )
```

This context stores the values of registers in annotating attribute *reg-values* and unallocated general-purpose registers in annotating attribute *free-GP-registers*. An example of model of language *Assembler-context* is given below:

```
(context * (reg-values (map (eax 1)))
    (index-values (nil 1 2))
    (free-GP-registers (ecx edx ebx esp ...)) )
```

In the fourth step, function *op-sem* is specified for each program construct model. The PSML code for two cases of specification of this function for C model of type *variable* corresponding to variable access construct in C and similar Assembler model of type *index-operand* corresponding to C variable access construct in Assembler are given below.

```
(mt-clan C)
(mt-function op-sem ((variable x) y)
    (if (symbolp (mt-aget y var-types x))
        (mt-aget y var-values x)
        (mt-aget (mt-aget y pointers x) var-values))
(mt-clan Assembler)
(mt-function op-sem ((index-operand x) y)
    (mt-aget y index-values (mt-aget x index)) )
```

3. Conclusion

A metalanguage model approach has been proposed that allows programming the operational semantics of programming languages in terms of these languages themselves, preserving the

structure of the source program at the model level and representing operational semantics as a code of transformation of the model in a metalanguage.

In the near future, it is planned to finally fix the design of the metalanguage, clarify the property lists of global variable *mt*, modeling the corresponding type *mt* and its contents, and implement the designed functions in the form of Lisp macros.

References

1. Attali I., Caromel D., Russo M. A formal executable semantics for Java // Proceedings of Formal Underpinnings of Java, an OOPSLA / Citeseer. — 1998. — Vol. 98.
2. Börger E., Schulte W. A programmer friendly modular definition of the semantics of Java // Formal Syntax and Semantics of Java. — 1999. — P. 353–404.
3. Camilleri J. An operational semantics for occam // International Journal of Parallel Programming. — 1989. — Vol. 18. — P. 365–400.
4. A compositional operational semantics for Java mt / Ábrahám E., de Boer F. S., de Roever W.-P., and Steffen M. // Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday. — 2003. — P. 290–303.
5. Coscia E., Reggio G. An operational semantics for Java // Ter¬ chical report, DISI, Univ. of Genova, Italy. — 1998.
6. da Silva Feitosa S., Ribeiro R. G., Du Bois A. R. Formal Semantics for Java-like Languages and Research Opportunities // Revista de Informática Teórica e Aplicada. — 2018. — Vol. 25, no. 3. — P. 62–74.
7. Drossopoulou S., Eisenbach S. Towards an operational semantics and proof of type soundness for Java // Formal Syntax and Semantics of Java. — 1998. — Vol. 1523.
8. An event-based structural operational semantics of multi-threaded Java / Cenciarelli P., Knapp A., Reus B., and Wirsing M. // Formal syntax and semantics of Java. — 1999. — P. 157–200.
9. Hartel P. H. LETOS–a lightweight execution tool for operational semantics // Software: Practice and Experience. — 1999. — Vol. 29, no. 15. — P. 1379–1416.
10. Maffeis S., Mitchell J. C., Taly A. An operational semantics for JavaScript // Programming Languages and Systems: 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings 6 / Springer. — 2008. — P. 307–325.
11. Matthews J., Findler R. B. Operational semantics for multi-language programs // ACM

SIGPLAN Notices. — 2007. — Vol. 42, no. 1. — P. 3–10.

12. Matthews J., Findler R. B. An operational semantics for scheme1 // Journal of Functional Programming. — 2008. — Vol. 18, no. 1. — P. 47–86.
13. Ramananandro T. Mechanized Formal Semantics and Verified Compilation for C++ Objects : Ph. D. thesis ; Université Paris-Diderot-Paris VII. — 2012.
14. Van Tassel J. P. An operational semantics for a subset of VHDL // Formal Semantics for VHDL. — Springer, 1995. — P. 71–106.
15. Verdejo A., Martí-Oliet N. Executable structural operational semantics in Maude // The Journal of Logic and Algebraic Programming. — 2006. — Vol. 67, no. 1-2. — P. 226–293.
16. Wallace C. The semantics of the C++ programming language. — 1993.
17. The semantics of the Java programming language: Preliminary version : Rep. / Citeseer ; executor: Wallace C. : 1997.
18. Wesonga S. O. Javalite-An Operational Semantics for Modeling Java Programs. — Brigham Young University, 2012.