Operational semantics

Operational semantics is a category of formal programming language semantics in which certain desired properties of a program, such as correctness, safety or security, are verified by constructing proofs from logical statements about its execution and procedures, rather than by attaching mathematical meanings to its terms (denotational semantics).

Operational semantics are classified in two categories: **structural operational semantics** (or **small-step semantics**) formally describe how the *individual steps* of a computation take place in a computer-based system; by opposition **natural semantics** (or **big-step semantics**) describe how the *overall results* of the executions are obtained. Other approaches to providing a formal semantics of programming languages include axiomatic semantics and denotational semantics.

The operational semantics for a programming language describes how a valid program is interpreted as sequences of computational steps. These sequences then *are* the meaning of the program. In the context of functional programming, the final step in a terminating sequence returns the value of the program. (In general there can be many return values for a single program, because the program could be nondeterministic, and even for a deterministic program there can be many computation sequences since the semantics may not specify exactly what sequence of operations arrives at that value.)

Perhaps the first formal incarnation of operational semantics was the use of the lambda calculus to define the semantics of Lisp.[1] Abstract machines in the tradition of the SECD machine are also closely related.

History

The concept of operational semantics was used for the first time in defining the semantics of Algol 68. The following statement is a quote from the revised ALGOL 68 report:

> *The meaning of a program in the strict language is explained in*

> *terms of a hypothetical computer which performs the set of actions that constitute the elaboration of that program. (Algol68, Section 2)*

The first use of the term "operational semantics" in its present meaning is attributed to Dana Scott (Plotkin04). What follows is a quote from Scott's seminal paper on formal semantics, in which he mentions the "operational" aspects of semantics.

> *It is all very well to aim for a more ‘abstract’ and a ‘cleaner’*

> *approach to semantics, but if the plan is to be any good, the operational aspects cannot be completely ignored. (Scott70)*

Approaches

Gordon Plotkin introduced the structural operational semantics, Matthias Felleisen and Robert Hieb the reduction semantics,[2] and Gilles Kahn the natural semantics.

Small-step semantics

Structural operational semantics

Structural operational semantics (SOS, also called **structured operational semantics** or **small-step semantics**) was introduced by Gordon Plotkin in (Plotkin81) as a logical means to define operational semantics. The basic idea behind SOS is to define the behavior of a program in terms of the behavior of its parts, thus providing a structural, i.e., syntax-oriented and inductive, view on operational semantics. An SOS specification defines the behavior of a program in terms of a (set of) transition relation(s). SOS specifications take the form of a set of inference rules that define the valid transitions of a composite piece of syntax in terms of the transitions of its components.

For a simple example, we consider part of the semantics of a simple programming language; proper illustrations are given in Plotkin81 and Hennessy90, and other textbooks. Let C_1, C_2 range over programs of the language, and let s range over states (e.g. functions from memory locations to values). If we have expressions (ranged over by E), values

(V) and locations (L), then a memory update command would have semantics:

$$\frac{\langle E, s\rangle \Rightarrow V}{\langle L := E\ ,\ s\rangle \longrightarrow (s \uplus (L \mapsto V))}$$

Informally, the rule says that "**if** the expression E in state s reduces to value V, **then** the program $L := E$ will update the state s with the assignment $L = V$".

The semantics of sequencing can be given by the following three rules:

$$\frac{\langle C_1, s\rangle \longrightarrow s'}{\langle C_1; C_2\ , s\rangle \longrightarrow \langle C_2, s'\rangle} \qquad \frac{\langle C_1, s\rangle \longrightarrow \langle C_1', s'\rangle}{\langle C_1; C_2\ , s\rangle \longrightarrow \langle C_1'; C_2\ , s'\rangle} \qquad \frac{}{\langle \mathbf{skip}, s\rangle \longrightarrow s}$$

Informally, the first rule says that, if program C_1 in state s finishes in state s', then the program $C_1 ; C_2$ in state s will reduce to the program C_2 in state s'. (You can think of this as formalizing "You can run C_1, and then run C_2 using the resulting memory store.) The second rule says that if the program C_1 in state s can reduce to the program C_1' with state s', then the program $C_1 ; C_2$ in state s will reduce to the program $C_1' ; C_2$ in state s'. (You can think of this as formalizing the principle for an optimizing compiler: "You are allowed to transform C_1 as if it were stand-alone, even if it is just the first part of a program.") The semantics is structural, because the meaning of the sequential program $C_1; C_2$, is defined by the meaning of C_1 and the meaning of C_2 .

If we also have Boolean expressions over the state, ranged over by B, then we can define the semantics of the **while** command:

$$\frac{\langle B, s\rangle \Rightarrow \mathbf{true}}{\langle \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle \longrightarrow \langle C; \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle} \quad \frac{\langle B, s\rangle \Rightarrow \mathbf{false}}{\langle \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle \longrightarrow s}$$

Such a definition allows formal analysis of the behavior of programs, permitting the study of relations between programs. Important relations include simulation preorders and bisimulation. These are especially useful in the context of concurrency theory.

Thanks to its intuitive look and easy-to-follow structure, SOS has gained great popularity and has become a de facto standard in defining operational semantics. As a sign of success, the original report (so-called Aarhus report) on SOS (Plotkin81) has attracted more than 1000 citations according to the CiteSeer [1] (http://citeseer.ist.psu.edu/673965.html) , making it one of the most cited technical reports in Computer Science.

Reduction semantics

Reduction semantics is an alternative presentation of operational semantics. Its key ideas were first applied to purely functional call by name and call by value variants of the lambda calculus by Gordon Plotkin in 1975[3] and generalized to higher-order functional languages with imperative features by Matthias Felleisen in his 1987 dissertation.[4] The method was further elaborated by Matthias Felleisen and Robert Hieb in 1992 into a fully equational theory for control and state.[2] The phrase “reduction semantics” itself was first coined by Felleisen and Daniel Friedman in a PARLE 1987 paper.[5]

Reduction semantics are given as a set of *reduction rules* that each specify a single potential reduction step. For example, the following reduction rule states that an assignment statement can be reduced if it sits immediately beside its variable declaration:

$$\mathbf{let\ rec}\ x = v_1\ \mathbf{in}\ x \leftarrow v_2;\ e \quad\longrightarrow\quad \mathbf{let\ rec}\ x = v_2\ \mathbf{in}\ e$$

To get an assignment statement into such a position it is “bubbled up” through function applications and the right-hand side of assignment statements until it reaches the proper point. Since intervening $\mathbf{let}$ expressions may declare distinct variables, the calculus also demands an extrusion rule for $\mathbf{let}$ expressions. Most published uses of reduction semantics define such “bubble rules” with the convenience of evaluation contexts. For example, the grammar of evaluation contexts in a simple call by value language can be given as

$$E ::= [\,] \mid v\,E \mid E\,e \mid x \leftarrow E \mid \mathbf{let\ rec}\ x = v\ \mathbf{in}\ E \mid E;\ e$$

where e denotes arbitrary expressions and v denotes fully-reduced values. Each evaluation context includes exactly one hole $[\,]$ into which a term is plugged in a capturing fashion. The shape of the context indicates with this hole where reduction may occur. To describe "bubbling" with the aid of evaluation contexts, a single axiom suffices:

$$E[\,x \leftarrow v;\; e\,] \;\longrightarrow\; x \leftarrow v;\; E[\,e\,] \qquad \text{(lift assignments)}$$

This single reduction rule is the lift rule from Felleisen and Hieb's lambda calculus for assignment statements. The evaluation contexts restrict this rule to certain terms, but it is freely applicable in any term, including under lambdas.

Following Plotkin, showing the usefulness of a calculus derived from a set of reduction rules demands (1) a Church-Rosser lemma for the single-step relation, which induces an evaluation function, and (2) a Curry-Feys standardization lemma for the transitive-reflexive closure of the single-step relation, which replaces the non-deterministic search in the evaluation function with a deterministic left-most/outermost search. Felleisen showed that imperative extensions of this calculus satisfy these theorems. Consequences of these theorems are that the equational theory—the symmetric-transitive-reflexive closure—is a sound reasoning principle for these languages. However, in practice, most applications of reduction semantics dispense with the calculus and use the standard reduction only (and the evaluator that can be derived from it).

Reduction semantics are particularly useful given the ease by which evaluation contexts can model state or unusual control constructs (e.g., first-class continuations). In addition, reduction semantics have been used to model object-oriented languages,[6] contract systems, exceptions, futures, call-by-need, and many other language features. A thorough, modern treatment of reduction semantics that discusses several such applications at length is given by Matthias Felleisen, Robert Bruce Findler and Matthew Flatt in *Semantics Engineering with PLT Redex*.[7]

Big-step semantics

Natural semantics

Big-step structural operational semantics is also known under the names **natural semantics, relational semantics** and **evaluation semantics**.[8] Big-step operational semantics was introduced under the name *natural semantics* by Gilles

Kahn when presenting Mini-ML, a pure dialect of ML.

One can view big-step definitions as definitions of functions, or more generally of relations, interpreting each language construct in an appropriate domain. Its intuitiveness makes it a popular choice for semantics specification in programming languages, but it has some drawbacks that make it inconvenient or impossible to use in many situations, such as languages with control-intensive features or concurrency.

A big-step semantics describes in a divide-and-conquer manner how final evaluation results of language constructs can be obtained by combining the evaluation results of their syntactic counterparts (subexpressions, substatements, etc.).

Comparison

There are a number of distinctions between small-step and big-step semantics that influence whether one or the other forms a more suitable basis for specifying the semantics of a programming language.

Big-step semantics have the advantage of often being simpler (needing fewer inference rules) and often directly correspond to an efficient implementation of an interpreter for the language (hence Kahn calling them "natural".) Both can lead to simpler proofs, for example when proving the preservation of correctness under some program transformation.[9]

The main disadvantage of big-step semantics is that non-terminating (diverging) computations do not have an inference tree, making it impossible to state and prove properties about such computations.[9]

Small-step semantics give more control over the details and order of evaluation. In the case of instrumented operational semantics, this allows the operational semantics to track and the semanticist to state and prove more accurate theorems about the run-time behaviour of the language. These properties make small-step semantics more convenient when proving type soundness of a type system against an operational semantics.[9]

See also

- Algebraic semantics
- Axiomatic semantics
- Denotational semantics

- Formal semantics of programming languages

References

1. *McCarthy, John. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I" (https://web.archive.org/web/20131004215327/http://www-formal.stanford.edu/jmc/recursive.html) . Archived from the original (http://www-formal.stanford.edu/jmc/recursive.html) on 2013-10-04. Retrieved 2006-10-13.*
2. *Felleisen, M.; Hieb, R. (1992). "The Revised Report on the Syntactic Theories of Sequential Control and State". Theoretical Computer Science. **103** (2): 235–271. doi:10.1016/0304-3975(92)90014-7 (http*

s://doi.org/10.1016%2F0304-3975%2892%2990014-7) .

3. *Plotkin, Gordon (1975). "Call-by-name, call-by-value and the λ-calculus" (https://www.sciencedirect.com/science/article/pii/0304397575900171/pdf?md5=db2e67c1ada7163a28f124934b483f3a&pid=1-s2.0-0304397575900171-main.pdf) (PDF). Theoretical Computer Science.* ***1*** *(2): 125–159. doi:10.1016/0304-3975(75)90017-1 (https://doi.org/10.1016%2F0304-3975%2875%2990017-1) . Retrieved July 22, 2021.*
4. *Felleisen, Matthias (1987). The calculi of Lambda-v-CS conversion: a syntactic theory of control and state in imperative higher-order programming languages (https://www2.ccs.neu.edu/racket/pubs/dissertation-f*

elleisen.pdf) *(PDF) (PhD). Indiana University. Retrieved July 22, 2021.*

5. *Felleisen, Matthias; Friedman, Daniel P. (1987). "A Reduction Semantics for Imperative Higher-Order Languages". Proceedings of the Parallel Architectures and Languages Europe. International Conference on Parallel Architectures and Languages Europe. Vol. 1. Springer-Verlag. pp. 206–223. doi:10.1007/3-540-17945-3_12 (https://doi.org/10.1007%2F3-540-17945-3_12) .*
6. *Abadi, M.; Cardelli, L. (8 September 2012). A Theory of Objects (https://books.google.com/books?id=AgzSBwAAQBAJ&q=%22operational+semantics%22) . ISBN 9781441985989.*

7. *Felleisen, Matthias; Findler, Robert Bruce; Flatt, Matthew (2009). Semantics Engineering with PLT Redex (https://mitpress.mit.edu/books/semantics-engineering-plt-redex) . The MIT Press. ISBN 978-0-262-06275-6.*
8. *University of Illinois CS422 (https://web.archive.org/web/20131019133339/https://fsl.cs.illinois.edu/images/6/63/CS422-Spring-2010-BigStep.pdf)*
9. *Xavier Leroy. "Coinductive big-step operational semantics".*

Further reading

- Gilles Kahn. "Natural Semantics". *Proceedings of the 4th Annual Symposium on Theoretical Aspects of*

Computer Science. Springer-Verlag. London. 1987.

- Gordon D. Plotkin. A Structural Approach to Operational Semantics (http://citeseer.ist.psu.edu/673965.html) . (1981) Tech. Rep. DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark. (Reprinted with corrections in J. Log. Algebr. Program. 60-61: 17-139 (2004), preprint (http://homepages.inf.ed.ac.uk/gdp/publications/sos_jlap.pdf)).
- Gordon D. Plotkin. The Origins of Structural Operational Semantics. J. Log. Algebr. Program. 60-61:3-15, 2004.

(preprint (http://homepages.inf.ed.ac.uk/gdp/publications/Origins_SOS.pdf)).

- Dana S. Scott. Outline of a Mathematical Theory of Computation, Programming Research Group, Technical Monograph PRG–2, Oxford University, 1970.
- Adriaan van Wijngaarden et al. Revised Report on the Algorithmic Language ALGOL 68. IFIP. 1968. ([2] (http://vestein.arb-phys.uni-dortmund.de/~wb/RR/rr.pdf))
- Matthew Hennessy. Semantics of Programming Languages. Wiley, 1990. available online (https://www.cs.tcd.ie/

matthew.hennessy/splexternal2015/resources/sembookWiley.pdf) .

External links

- Media related to Operational semantics at Wikimedia Commons