A compositional semantics for the Turing machine

Peter van Emde Boas[*]

Departments of Mathematics and Computer Science, University of Amsterdam, Nieuwe Achtergracht 166, 1018 WV Amsterdam

Dedicated to J.W. de Bakker at the occasion of his 25-th anniversary at the CWI
March 9, 1989

Abstract

In the current tradition of constructing compositional semantics for programming languages the machine language of the most popular model in structural complexity theory seems to have been overlooked. We show that this gap in the literature can easily be filled. Notwithstanding the fact that the contrary is generally believed, we show that the standard semantics for Turing machines is compositional, provided the right syntax is used for describing their programs. This presents one more example for Janssen's observation that lack of compositionality in well understood semantic situations is frequently caused by syntactic prejudices and not by real semantic problems.

1 Why a semantics for the Turing machine is needed

The study of semantics of programming languages has been an important research area in computer science in general and, in particular at the CWI (formerly the Mathematical Centre) for the past 25 years. The present volume and the seminar organized at the occasion of de Bakkers 25^{th} anniversary at the CWI gives witness to his important contributions to this research area.

I have always considered myself to be a side-liner watching this game of semantics. When I encountered semantical investigations for the first time about 20 years ago, I was amazed by the amount of intellectual effort invested in issues of programming which I believed at that time to be self evident and well understood. It took only a short period to understand that the gap between our intuitive operational understanding of computational processes and their formal representation by mathematical models is huge. The trails which are designed for traversing this gap are frequented by toads and dragons, with pitfalls and swampy pools everywhere around, bringing misfortune to the careless wanderer. I have also learned that our original semantic intuition remains the only compass which will guide us out of the murky waters. Our intuition promises us the existence of the other side of the river and shows us in which direction to move. If we stick to its course we will not be swept away into a bottomless whirlpool of formalisms and symbol manipulation strategies which remain stuck in the translation.

[*]also CWI-AP6, Amsterdam

My experience of the past two decades also shows that semantical investigations follow fashions which arrive and go away. When I got introduced into the area characterizing recursive functions by least fixed-point semantics was the hot topic. Next we had a long period where models and proof theories for concurrency formed the backbone of the semantics activities. We have seen the algebraic approach to data types, and more recently we have seen a large amount of interest in the study of semantical problems arising in the weird language we call Prolog. The bare fact that the semantics of this language has become a research topic already presents a convincing argument against the claim that a Prolog has an evident semantics defined by its logic clauses.

A semantic topic which I have never encountered in these studies is the semantics of the Turing machine. This device represents (together with the Random access machine and, more recently, the parallel versions of the RAM) the work-horse of computational complexity theory. Being one of the most general accepted formalisms for universal computability it pops up in almost every corner of theoretical and practical computer science. It is also a device which predates the real computers by almost a decade [9], and a device which has hardly changed its fundamental structure during the half century it has been among us.

As far as semantic theory is concerned it seems that complexity theory still lives within the stone age of pure operationalism. In the basic courses and textbooks of introductory computation theory Turing machines are given an operational interpretation. Turing machine programs are only explicitly constructed in very simple examples and the corresponding exercises. In more advanced theory explicit programs are never specified; every algorithm is suggested and made believed to be programmed on Turing machines by force of intimidation. This practice is justified on the basis of the so-called *Inessential use of Church' thesis: Whatever we feel to be computable can be brought within the scope of our formal models.*

Since there are no real Turing machine programs there is no need for proving that these programs achieve what they are intended to do. Since there is no need to prove programs correct there is no specified semantics relative which these programs should meet their specifications. Neither does there exist a specification language aimed at Turing machines.

Faced with this absence of even the bare minimum of a semantic theory for a machine formalism which is generally believed to be of some importance one might ask for the cause to this deplorable state of affairs. The following observations are in order before trying to answer this question:

- There exists a well understood operational semantics for Turing machine computations. This semantics is presented in all textbooks on introduction to computation theory which describe the model. See for example [3].

- It is common knowledge that proving properties of Turing machine programs is an undecidable problem. Starting from the undecidability of the Halting problem it is an elementary exercise to show that virtually every behavioral property of Turing machines is also undecidable. The same holds for all nontrivial extensional properties (properties which are completely determined by the function computed by the Turing machine), as follows from Rice's theorem [8]

- Proving properties of Turing machine programs is not only hard in theory as exemplified by the above undecidability results, but even the small instances are highly intractable. This is illustrated for example by the problem of evaluating the initial values of the so-called busy beaver function: the largest number of 1-symbols printed by a k-state Turing machine program over the alphabet $\{0,1\}$) as a function of k. The evaluation of this function for $k = 5$ was the subject of a competition organized at the occasion of the 1983 GI TCS meeting at Dortmund; the winner only could have obtained his result using special purpose hardware [2]

- It is generally believed that Turing machine programs rank among the most structureless programs one encounters in the universe. Every additional instruction may screw up the meaning of a well understood program in such a way that even the cleverest machine code hackers may fail to understand what's going on

Each of the above four observations can be invoked as an excuse for the absence of a well explored semantics theory of the Turing machine. In the line of the first remark one simply can deny that there exists a problem. We know perfectly well what we are talking about. But since the standard semantics is operational it is evident that this belief is unfounded and should be shattered.

The second observation leads to the excuse that it doesn't make sense to design a theory of Turing machines since we know beforehand that all problems will be unsolvable. This is an invalid argument in the light of the observation that researchers in semantics have never hesitated to look at semantical theories where the underlying validity problem is even far harder. The halting problem of Turing automata is only Σ_1^0-complete whereas many programming logics have Π_1^1-complete validity/satisfiability problems.

The third observations illustrates the lack of well designed paradigmatic examples of solved semantical problems in the area of Turing machine programs. This is just a matter of lack of interest of the research community. If nothing else is invented one can always redo the factorial example or rebuild the towers of Hanoi.

The last argument simply illustrates the fundamental discrepancy between the average researcher in semantics and its true vocation. He should investigate the semantics problems of real programmers using real-life programming languages and not be side-tracked by the clean well-structured toy languages designed by the theoreticians for their own benefit. We simply can't wait until humanity has decided that it will believe the prophets and start to program in a well structured manner [1]. As long as we can't cope with real Turing programs we have not yet fulfilled our task.

I believe therefore that after 25 years time has arrived that the semantics community should attack the problem of the Turing machine by taking its semantics serious. As a small contribution towards completion of this program I will in this note discuss the issue of compositionality of the semantics of Turing machines. This clearly represents a necessary preliminary step since it is generally accepted that there exist no alternative for working compositional in semantics [6].

2 The standard semantics of the Turing machine

For the purpose of this note I will use the single-tape Turing automaton in stead of the usual model from complexity theory where one has special input tapes, output tapes, and work tapes. It is folklore knowledge that all versions of Turing automata are equivalent in the sense that they can simulate each other with polynomial time overhead and constant factor overhead in space [10]. This implies that all models are equivalent from the perspective of structural complexity theory, and this provides us the freedom of selecting a convenient version to start our semantical investigations on.

It is usual to define a Turing machine by means of a sixtuple of sets. However, for the purpose of semantics it is advantageous to start from a more syntactic definition. For this purpose I introduce a pair of infinite alphabets $Q = \{q_0, q_f, q_1, \ldots\}$ and $\Sigma = \{s_0, s_b, s_1, \ldots\}$. The elements of the alphabet Q are called ***states*** and the elements of Σ are called ***(tape)symbols***. The state q_0 is called the ***start state*** and the state q_f is called the ***accepting state***. The symbol s_b is called the ***blank symbol***. We denote moreover $M = \{L, 0, R\}$.

Definition 1 A *Turing machine program* is a string A in the language described by the regular expression $([Q\Sigma Q\Sigma M])^*$.

Readers familiar with the traditional definition of a Turing machine will certainly recognize that the above definition yields the quintuple programs of nondeterministic Turing machines. The substrings of the form $[Q\Sigma Q\Sigma M]$ in A are called ***instructions***. It should be observed that the observed complexity of Turing machine semantics is therefore certainly not due to the complexity of the syntactic structure. The language of Turing programs as defined above is a simple regular language.

The traditional semantics of the Turing machine is defined in terms of computations and configurations. These concepts have syntactic definitions as well.

Definition 2 A *configuration* is a string c in the language described by the regular expression $\$\Sigma^*Q\Sigma\Sigma^*\$$. An *initial configuration* is described by the regular expression $\$q_0\Sigma\Sigma^*\$$. An *accepting configuration* is described by the regular expression $\$\Sigma^*q_f\Sigma\Sigma^*\$$.

It should be noted that the above definition yields a machine-independent concept of configuration in general and initial and final configurations in particular. Our next notion however requires the interaction between a configuration and a Turing program:

Definition 3 A configuration c is *terminal relative the Turing program A* if the unique substring of the form $Q\Sigma$ in c does not occur embedded in a substring of the form $[Q\Sigma$ in A.

It is usual to express this notion by requiring that for the pair q_is_j occurring in the configuration c there is no instruction $[q_is_jq_{i'}s_{j'}m]$ in A. As described above it seems that the property of being a terminal configuration is context sensitive, but for fixed program A the terminal (and therefore also the non-terminal) configurations form a regular language.

The next notion which must be introduced is the notion of a transition. It is common to attach context sensitive production rules to instructions in the program. When this has been done configurations no longer need a separate definition since they become nothing but productions in the grammar described by these production rules.

Definition 4 With the program A we associate the following collection $P(A)$ of context sensitive production rules:

- for every instruction $[q_i s_j q_{i'} s_{j'} 0]$ in A we introduce the production $(q_i s_j \mapsto q_{i'} s_{j'})$ in $P(A)$
- for every instruction $[q_i s_j q_{i'} s_{j'} R]$ in A and every symbol q_k which occurs in A we introduce the productions $(q_i s_j s_k \mapsto s_{j'} q_{i'} s_k)$ in $P(A)$; to this we add the production $(q_i s_j \$ \mapsto s_{j'} q_{i'} s_b \$)$ in $P(A)$
- for every instruction $[q_i s_j q_{i'} s_{j'} L]$ in A and every symbol q_k which occurs in A we introduce the productions $(s_k q_i s_j \mapsto q_{i'} s_k s_{j'})$ in $P(A)$; to this we add the production $(\$ q_i s_j \mapsto \$ q_{i'} s_b s_{j'})$ in $P(A)$
- These are the only productions in $P(A)$

Definition 5 A *transition* between two configurations c and c' relative A, denoted $c \overset{A}{\mapsto} c'$ is a one step production from c into c' by the grammar $P(A)$. A *computation* from c to c' of machine A, denoted $c \overset{A}{\mapsto}^{*} c'$ is a production from c to c' under the grammar $P(A)$. A *full computation* is a computation starting in an initial state and leading to a terminal configuration. The full computation is *accepting* if its last configuration is.

The crucial part of the above definition is the transition between one step transitions to computations. This is nothing but the step between a binary relation on the set of all configurations to its transitive reflexive closure, which is a well known ingredient in semantics. As we will see in the sequel it is precisely this transitive closure from which the problems of understanding the semantics of Turing machine programs originate.

It remains to assign inputs and results to Turing machine computations. This is again easily done by some syntactic definitions.

Definition 6 The *result* of some configuration c is equal to $U(c)$ where the mapping U is some simple gsm mapping, depending on the purpose of the computation intended.

The most common examples of results covered by this definition are:

- existence of a terminal configuration: here the gsm mapping U tests for the occurrence of a state-symbol pair for which no instruction is available in A. This is the result which is relevant for *accepting* a set by halting.
- existence of an accepting configuration: here the gsm mapping U tests for the occurrence of the accepting state (which is assumed not to be active in any instruction in A). This is the result which is relevant for *recognizing* a set by accepting.
- the result of a function evaluation: here the gsm mapping U tests for the occurrence of the accepting state symbol, but at the same time all occurrences of the \$-symbol, the blank symbol s_b and the state-symbols will be erased.

Definition 7 The *initial configuration* for some string x is the configuration $i(x) = \$q_0 x\$$

It is usual to presume that x is nonempty and that the blank symbol doesn't occur in x. In the case of an empty input the initial configuration $\$q_0 s_b \$$ is used.

The above notions suffice for defining the operational semantics of Turing machine programs. In order to assign to a program A some meaning $M(A)$ which is a multi-valued partial function from strings to results argues as follows: for the input string x the initial configuration $i(x)$ is constructed. Next all full computations $i(x)\overset{A}{\mapsto}{}^{*}c'$ are obtained and for every resulting terminal configuration c' the result $U(c')$ is computed. Then $M(A)$ is the function which maps x onto the set of results $U(c')$ obtained in this way.

3 Is the semantics of Turing machines compositional?

In the previous section I have given a definition of the traditional operational semantics of Turing machines. But why is this called an operational semantics? It seems that this name is used only because the semantics has been obtained using computation sequences consisting of configurations connected by computation steps, where each computation step is performed by execution of some instruction in the program. This interpretation, however, only refers to the names we have given to our syntactic objects used in the definitions. In fact the whole definition above is given in terms of well known mathematical structures, like regular languages, production rules, grammars, gsm mappings, a transition relation and its reflexive transitive closure.

If we put everything together we can rephrase the entire definition by the formula:

$$M(A) = \lambda x[U((\overset{A}{\mapsto}{}^{*})(i(x)))] \qquad (i)$$

It should be self evident that the above formula is as mathematical and as denotational as one might ever require; all operators in this formula have a clear mathematical meaning which, in principle, is easily formalized in the language of set theory. If the above semantics therefore is called operational this is not caused by the semantics itself but rather by the way it has been traditionally been looked upon within computer science.

The more relevant question to ask is whether the semantics as described by formula i is compositional or not. Here I refer to the well known Fregean principle of compositionality which is expressed by the assertion:

The meaning of some compound expression is composed from the meanings of its parts

According to the reconstruction of the algebraic content of the compositionality principle which was given by Theo Janssen in his 1983 dissertation [4], we may obtain a compositional semantics according to i provided we can assign both to the language of Turing machine programs and to its semantics the structure of an algebra, and do so in such a way that the above mapping M becomes a homomorphism. The algebra on the semantic structure may be obtained by combining operators from another algebra like set theory into so-called polynomial operators, according to the guideline *polynomial is safe*. At the same time we may exploit all imaginable tricks in restructuring the syntactic structure of the programs in order to tune the syntactic algebra towards the semantic algebra.

Now the following observations can be made.

1. The transformation from instructions in A to production rules in $P(A)$ is of a straightforward permutational nature; the symbols are slightly permuted. This operation is easily expressed by polynomial operators over the algebra of strings and sets

2. The transformation from a production rules to the set of all single step productions performed using this production is a polynomial operator in the algebra of sets of strings
3. The single step production relation $\mapsto$ according to a set of production rules is the union of the relations according to the individual productions (a clear instance of homomorphic behavior where the operator concatenation is mapped onto union)
4. The transition from single step productions to many step productions which represent computations is represented by the transitive reflexive closure operator which belongs to the algebra of sets and strings and relations
5. The final semantics i is obtained by pre- and post-multiplication with simple mappings belonging to the algebra of strings and sets

The above observations show us how to restructure the syntax of our Turing machine programs. The individual 5 steps are reflected if we restructure the grammar describing Turing machine programs as follows:

```
instruction ::= '['state,symbol,state,symbol,move']'    $ step 1
instruction1 ::= instruction                            $ step 2
program1 ::= (instruction1)*                            $ step 3
program2 ::= program1                                   $ step 4
program  ::= program2                                   $ step 5
```

In this restructuring of the grammar we have introduced three unit-rules which express the changes of types for the underlying semantic objects from (set of) rules to subsets of the transition relation, and from the transition relation to its transitive closure and next to its input-result relation. Such lifting rules are a common tool in Montague grammar.

It should be noted at this point that we have obtained a compositional semantics for Turing machine programs without modifying the traditional semantics (widely believed to be an operational semantics) just by reconsidering the semantics and slightly restructuring the syntax. The key idea is that the juxtaposition of instructions should be mapped onto the union of relations, and that this union should be performed before subjecting the transition relation to the transitive closure (as expressed by the lifting rules).

One could make the objection that the second step in the semantics described above, which transforms production rules into single step productions requires knowledge of the complete tape alphabet of the involved Turing machine. There are several solutions to this problem. One can specify the tape alphabet at the start of the construction and turn it into a constant of the semantics for this particular device A. An alternative is to work with the infinite alphabet Σ as introduced in the previous section. The fact that in the transition many tape symbols are allowed which won't be used by the actual machine has no influence on the eventual semantics according to our construction. Finally one can restrict oneself to Turing automata over a fixed alphabet; as is well known this can be done without loss of generality. Note that a similar problem with the state alphabet doesn't arise, due to the fact that we may restrict ourselves to those strings which contain exactly one symbol from Q.

The observation that compositionality for a well understood semantical interpretation can be obtained by modifying the syntax is not new. In the dissertation of Theo Janssen mentioned before [5] there are presented a number of examples in early extensions of Montague grammar where authors have deviated from the compositionality principle mainly on behalf of syntactic prejudice; the semantic phenomena they were after were easily brought within the scope of the compositionality principle by having another look at the syntax.

The reconstruction of a compositional semantics for Turing machines in this note illustrates that similar prejudices may lead to misconceptions in programming language semantics as well.

The positive result of our exercise does not mean that the compositionality achieved will be very helpful in understanding the behavior of Turing machines. The transitive closure operator easily will obliterate all structure of the union operator on the instructions. The complexity is caused by the fact that union and transitive closure are non-commuting operators, but that is a situation which is not unusual in mathematics.

Only for programs which are structured in some very restricted way this union may sort of commute with the transitive closure and reappear in the shape of a meaningful operation like functional composition, nondeterministic choice, iteration or one of the other composition operators which are well known in the theory of semantics of programming languages. See for example the description of the combination of Turing machines, called *machine schemata* in section 4.3 of [7]. What we have achieved therefore by our analysis is that we may start to understand why such restrictions on the program structure are required and what kind of restrictions we should aim for. The real work, however still has to be done. Clearly an interesting research topic for the next 25 years.

References

[1] Dijkstra, E.W., *A Discipline of Programming*, Prentice Hall Series in Automatic Computation 1976.

[2] Report on the Busy beaver competition 1982/83 distributed at the 6th conference on theoretical computer science, Jan 05-07, 1983, Dortmund. See also EATCS bulletin vol. 19, Feb 1983, pp. 77

[3] Hopcroft, J.E. and Ullman, J.D., *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley (1979)

[4] Janssen, T.M.V., *Foundations and applications of Montague grammar, Part 1: Philosophy, framework, computer science*, CWI Tract 19 (1986)

[5] Janssen, T.M.V., *Foundations and applications of Montague grammar, Part 2: Applications to natural language*, CWI Tract 28 (1986)

[6] Janssen, T.M.V. and van Emde Boas, P., *Some observations on compositional semantics*, in Kozen, D., ed., Logic of programs, Proceedings 1981, Springer Lecture notes in computer science 131 (1982) pp. 137-149

[7] Lewis, H.R. and Papadimitriou, C.H., *Elements of the Theory of Computation*, Prentice-Hall (1981)

[8] Rogers, H. Jr, *Theory of Recursive Functions and Effective Computability*, Mc Graw-Hill Series in Higher Mathematics (1967)

[9] Turing, A.M., *On computable numbers, with an application to the entscheidungsproblem*, Proc. London Math. Soc. ser. 2, 42 (1936) 230-265

[10] van Emde Boas, P. *Machine models and simulations*, to appear in J. van Leeuwen (ed.), Handbook of theoretical computer science, North Holland Publ. Comp. 1989. Preprint: rep. ITLI-CS-89-02.