THE UNIVERSITY OF MICHIGAN

COMPUTING RESEARCH LABORATORY[1]

RECONSIDERING TURING'S THESIS (TOWARD MORE REALISTIC SEMANTICS OF PROGRAMS)

Yuri Gurevich

CRL-TR-36-84

September 1984

Room 1079, East Engineering Building
Ann Arbor, Michigan 48109
USA
Tel: (313) 763-8000

[1]Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agency.

Reconsidering Turing's thesis
(Toward more realistic semantics of programs)

Yuri Gurevich[1]
Department of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor, MI 48109

Abstract. The classical computation model is based on the notion of a potentially infinite computing device (like a Turing machine or an idealized Pascal machine). We propose here an alternative computation model which explicitly recognizes finiteness of computers. The new operational semantics is especially appropriate in the case of algorithms sensitive to the bounds on machine resources (like algorithms for operating systems).

[1]Supported in part by NSF grant MCS83-01022

Introduction

In the classical computation model algorithms are closely related to potentially infinite computing devices (like Turing machines or idealized Pascal machines). These devices interleave computing with extending the necessary resources in a manner reminiscent of human computing with pen and paper. In §1 we explain our reservations with respect to the claim -- implicit in Turing's thesis -- that every Turing machine realises an algorithm. The same reservations apply to potentially infinite devices in general.

Our alternative to the notion of a potentially infinite machine is presented in §2. It is the notion of a uniform family of finite machines. This notion is to remain informal (like the notion of a potentially infinite machine in the classical case). A program for a uniform family of finite machines defines a global algorithm; every machine in the family executes a local version of the global algorithm. In particular, a program for a potentially infinite machine M defines a global algorithm for the family of finite versions of M; this global algorithm is insensitive to those resource bounds of the finite machines which vary from one machine to another. In §3 we discuss global algorithms which are sensitive to the bounds on varied resources. Such resource sensitive algorithms are most common in real world computing. Algorithms realized by operating systems obviously are resourse sensitive. But also many algorithms whose input-output relation is external to computers are resource sensitive for the sake of efficiency. An algorithm for sorting a large database, for example, may start with loading as much data as possible into the primary memory.

The notion of a global algorithm is to remain informal too. Its counterpart in the classical case is the informal notion of an algorithm. Even though resource sensitive global algorithms do not result directly from programs for potentially infinite machines, they can be simulated by potentially infinite machines. However we prefer to study global algorithms directly. An interesting question arises: what is a counterpart of Turing's thesis for global algorithms? This question is addressed in §4. Finally, §5 contains some additional remarks.

An important motivation for this paper is related to denotational semantics as presented in [St]. If you are not interested in denotational semantics just skip the rest of this introduction. The Scott-Strachey approach to the programming language theory is most impressive; I am not completely comfortable with the present form of denotational semantics however.

It was surprising to find untyped lambda calculus as a basis for denotational semantics. This calculus was never too popular among logicians; the underlying intuition of the calculus is not clear. Note that untyped calculus is used in [St] mostly for foundational purposes; when it comes closer to applications, a much more intuitive typed calculus is used. Untyped lambda calculus does not have (contrary to, say, Peano Arithmetic) a standard model. To be sure, untyped lambda calculus has (attractive) models [Sc]. Chosing such a model, current denotational semantics provides meaning to programs in a consistent way. But the meaning is somewhat arbitrary because of some arbitrariness of the underlying model.

More importantly, the finite character of a computer is not reflected properly in current denotational semantics (in spite of the presence of finite elements in lattices and the use of error elements). The operational counterpart of this denotational semantics is an infinite machine. It comes as no surprise that current denotational semantics deals more easily with a complicated recursion than with a simple utilization of, say, the primary memory.

It is our intent to use the alternative operational semantics as a basis for a reformed denotational semantics which provides unique meaning to programs and is able to deal naturally with limited resources.

Acknowledgements. This paper gained much from the constructive criticism of Andreas Blass. Issues related to hardware were an object of stimulating discussions with Gideon Frieder and with John Patrick Hayes who contributed some examples of uniform families of finite machines. Saharon Shelah contributed to related complexity issues (complexity considerations will be an object of a subsequent paper). Comments of Egon Boerger were useful as well. I am very grateful to all these people.

§1 Another look at potentially infinite machines

Turing's thesis states that every algorithm can be simulated by an appropriate Turing machine. An implicit (or the "trivial") part of the thesis is that every Turing machine realises an algorithm. This implicit part of the thesis is re-examined here.

Turing machines are potentially infinite idealised computing devices. Whenever a new portion of a tape is needed, we go and somehow get it. In other words, computation of a Turing machine can be interleaved with extension of its tape (or tapes). Generally, computation of a potentially infinite computing device can be interleaved with extension of resources.

One obvious criticism of potentially infinite devices is related to finite character of many relevant recources: eventually we will run out of tape material, ink or whatever. Here is another criticism which seems to us more important: the program of a potentially infinite computing device does not tell us how and where to get an additional portion of a needed resource.

(For proving theorems about Turing machines it is convenient sometimes to view them as actually infinite computing devices. Actually infinite computing devices are even more impractical and we restrict this discussion to potentially infinite machines.)

It isn't our intention to ridicule the implicit part of Turing's thesis. The finite character of resources is a philosophical question and physical limits on resources are far removed from the needs of ordinary computing. Moreover, in many cases computing can be interleaved with extension of resources. Consider for example the case (analysed by Turing [Tu]) of a human computer using pens and paper. (Juggling with more and more disks turns even the little Macintosh into a potentially infinite machine.)

However, existing electronic computers (under usual usage) do not seem potentially infinite to us. With all due respect to the classical computation model we would like to propose another computation model which deals only with finite computing devices.

§2. An alternative to potentially infinite machines

Our basic assumption is that the hardware of a computer is essentially fixed. Changing a bolt is not important but extending the primary memory (or adding a new processor to a multiprocessor machine) turns a given computer into a new one.

Since the hardware of a computer is fixed the computer can be viewed as a finite automaton: it can be only in finitely many essentially different states. However the conventional finite automata theory is not of much help in the case of a realistic size computer. The total number of essentially different states is overwhelming. It is not feasible to describe the behavior of a real computer by a state transformation table or to write down a regular expression describing the given computer.

Finite machines satisfy perfectly the implicit part of Turing's thesis but not the explicit part of it. Consider for example an ordinary Pascal program for computing, say, the factorial function. It defines an algorithm which cannot be adequately simulated by any finite machine. Of course, the program can run on a finite machine but the intended meaning of it will be distorted (when it comes to computing the factorial of a large enough number). In other words, no finite machine gives an adequate operational semantics to our program whereas an idealised potentially infinite Pascal machine is able to do that.

We seek a computational model which allows only finite machines but is nevetheless able to provide an adequate operational semantics for programs and algorithms. In this connection let us emphasize that philosophical rejection of the implicit part of Turing's thesis does not shake the strong feeling that a program for a potentially infinite machine gives a genuine algorithm that can be carried on *as long as we have the necessary resources*. Such program runs on all corresponding finite machines. All those finite machines together provide -- we believe -- an adequate operational semantics for the program. All finite (virtual) Pascal machines together provide an adequate operational semantics for Pascal programs. Similarly all finite versions of a given Turing machine provide an adequate semantics for its state transformation table . This brings us to our alternative to a potentially infinite machine. It is a *uniform family of finite machines*.

The machines in a uniform family are related by means of a common programming language. The same programs run on all the machines. Any particular machine is specified within the family by a bunch of parameters, the parameters reflect the resource bounds. The finite Pascal machines form a uniform family. Finite versions of a universal Turing machine form another uniform family. The finite versions of any potentially infinite idealised computing device form a uniform family. But a uniform family can also be finite. The notion of uniform families of finite machines will remain informal. Possible formalizations of this notion will be discussed later.

Here is an important criterion of uniformity of a family F of finite machines: there is a potentially infinite machine (with a fixed program) which, given the parameters of an arbitrary machine X in the family F, simulates X using only a finite -- and computable from the parameters of X -- portion of its resources.

Let us elaborate on the uniform family of finite versions of a universal Turing machine U. (An accurate description of finite Pascal machines is a complicated story. From our point of view it amounts to providing an adequate operational semantics to Pascal itself.)

We suppose the following about U. It's only tape has a specially marked leftmost cell and is potentially infinite to the right. Autonomous input and output devices are used. When the control is in a special reading mode (state) the input device provides the first unread character or a special end-of-file character. As a side effect of an instruction a character can be sent to the output device. An input consists of a program followed by a special character and data. First U writes the program on the tape and then executes it on the data. The data is not supposed to be necessarily finite: the machine U can work as an operating system or somebody can keep providing data from a terminal.

For every positive integer n consider the finite version of U with tape of length n and a specially marked rightmost cell. If this finite machine attempts to move the head (or a head if there are several heads) to the right from the rightmost cell then it halts in a special mode of the control; otherwise it works exactly like U. All these finite machines together form a uniform family.

§3. Global algorithms sensitive to resource bounds

A program for a uniform family of finite machines gives a *global algorithm* for the family; a particular finite machine executes a local version of this global algorithm. Like the notion of a uniform family of finite machines the notion of global algorithm is to remain informal.

Let us consider more closely the uniform family of finite versions of a potentially infinite machine. Some global algorithms for the family are given by programs for the potentially infinite machine. Let us call them Turing global algorithms for a moment. An interesting question arises : are there non-Turing global algorithms ? In other words, are there global algorithms for the family of finite machines which cannot be given by a program for the original potentially infinite machine ?

Our answer is YES. To be more specific we consider the family of finite Turing machines from §2. Using the ability to recognize the rightmost cell the finite machines can execute programs which are meaningless for the original machine with tape unbounded to the right. For example the following global algorithms can be programmed.

1. Partition the free part of the tape (which is all the tape except for an initial segment needed for the program) into two intervals of the same length (if the free part is of odd length let the left interval be one cell longer). Write the incoming data on the left interval till it is full or the symbol $ is encountered or the end-of-file is encountered. If and when $ is encountered write the subsequent data on the right interval till it is full or the end-of-file is encountered. Then compute relative frequences of the character a on each interval. (An algorithm of that sort can be useful in checking whether two parts of a text use the same code.)

2. As above, partition the free part of the tape into two intervals of the same or almost the same length . Perform a routine L with the incoming data on the left interval till a special symbol $ is encountered. Then perform a routine R with the incoming data on the right interval till $ is encountered. Then again perform L on the left interval, etc. Halt if end-of-file is encountered. (This algorithm provides some primitive time sharing).

The examples look somewhat superficial (like anything done with Turing machines) but they convey, we hope, the idea. Algorithms for finite machines can use the resource bounds -- the primary memory size, the maximal size of machine words, the number of processors (in the case of multiprocessor machines), etc. -- in an essential way. Many practical algorithms do, especially algorithms for operating systems.

Before we proceed let us introduce the following terminology. A global algorithm for a uniform family of finite machines is *resource insensitive* if it does not mention the machine resource bounds; otherwise it is *resource sensitive*. A resource insensitive global algorithm runs in exactly the same way on any two finite versions of the same potentially infinite machine all the time that the needed resources are available. Every global Turing algorithm is resource insensitive and, *vice versa*, every resource insensitive global algorithm for the family of finite versions of a potentially infinite machine is a Turing global algorithm.

Of course, our definition of the family of finite versions of the universal Turing machine U was biased. By introducing the new special mark for the rightmost cells we extended the programming language and allowed penetration of information about the recource bound into the programming language. If the rightmost cells are not specially marked then every global algorithm for the family is a Turing global algorithm. Was it fair to introduce a special mark for the rightmost cell? Again our answer is YES. After all, real programs run on finite machines and in many cases they are resource sensitive.

There is no doubt of the usefulness of resourse sensitive global algorithms. Knowing the length of machine words and the needed number of digits after the decimal point in a numeric computation such an algorithm can figure out whether single precision or double presision, etc. should be used. Knowing the number of processors (in the case of multiprocessor machines) such an algorithm can figure out how to divide between the processors the job of multiplying two matrices. Operating systems adaptable to many computers (like the kernel of UNIX) execute resource sensitive global algorithms. The global (even global resource sensitive) algorithms can be simulated by potentially infinite machines but we prefer to study them directly.

Some high level programming languages -- like FORTRAN -- do not reflect any information about resource bounds. All that a finite FORTRAN machine can do is simply to mimic the potentially infinite FORTRAN machine until one of the necessary resources is exhausted. On the other hand Pascal has an implementation-defined constant MAXINT. Several implementation-defined constants are found in ADA and APL; these constants reflect considerable information about the resource bounds. There seems to be a trend to include some implementation-defined constants in high level programming languages. We consider this trend symptomatic. It reflects, we believe, the fact that the operational semantics based on a uniform family of finite machines is in many cases closer to the intended meaning of usual programs than the operational semantics based on a potentially infinite machine. (Note that enriching a programming language by names for essential resource bounds does not make it implementation dependent. Pascal with MAXINT, for example, is not more implementation dependent than Pascal without MAXINT).

§4. On the new thesis

Now that we know about existence of "non-Turing" algorithms we would like to formalize the notion of global algorithms having Turing's thesis in mind as a model. Turing's thesis can be restated as follows: every potentially infinite computing device can be simulated by a Turing machine. It implies that every potentially infinite computing device can be simulated by a fixed universal Turing machine equipped with an appropriate program. Thus we would like to define formaly a special kind -- a tribe -- of uniform families of finite machines or a specific uniform family -- a universal family -- of finite machines; the new thesis would then state that any uniform family of finite machines can be simulated by an appropriate family in the tribe or by the universal family with an appropriate program respectively.

But what does it mean that a uniform family F2 of finite machines (with an appropriate program) simulates another uniform family F1 of finite machines? We restrict our attention here to the case when the F2 machines know or are able to find out their resource bounds. F2 (with a simulation program Q) *simulates* F1 if

(i) given the parameters of an F1 machine X, an arbitrary F2 machine Y (equipped with the program Q) computes and outputs either YES or NO; in the case of YES the machine Y is ready to simulate X, and

(ii) for every F1 machine X there is an F2 machine Y such that, given the parameters of X, Y (with Q) computes YES.

The notion of simulation of one finite machine X by another finite machine Y (with a fixed program Q) is to remain informal. Such simulation certainly implies (but not reduces to) computing the input-output relation of X.

One obvious candidate for the role of a universal uniform family of finite mashines is the family of finite versions of a universal Turing machines from §2. By the definition of uniform families there exist a potentially infinite machine M which, given the parameters of an arbitrary machine X in F, simulates X. Moreover, M spends only a finite -- and computable from the parameters of X -- portion of its resources for simulating X. By Turing's thesis M can be simulated by a Turing machine T. Hence, given the transformation table of T and the parameters of an arbitrary machine X in F, a universal Turing machine U simulates X.

Actually, Turing's original justification of his thesis [Tu] gives more: T needs only a finite -- and computable from the parameters of X -- portion of tape in order to simulate X. Hence the family of finite versions of U, equipped with an appropriate program Q, is able to simulate any uniform family F of finite machines.

Instead of Turing machines it is natural to use ramdom access memory machines (which better reflect modern computers); we skip the details here. Different formalizations of the notion of global algorithms are considered in a forthcoming paper of Andreas Blass and Yuri Gurevich.

A somewhat weaker form of Turing's thesis states that every computable function is computable by a Turing machine. It formalizes the notion of computable functions rather than the notion of algorithms. The first formalization of the notion of computable functions was given by Church's thesis [Ch]; many other formalizations of this notion are known by now.

Recall that a global algorithm is an algorithm for a uniform family of finite machines; each particular machine executes a local version of the global algorithm. A global algorithm computes a *global function;* each particular machine computes a local version of the global funcion. A notion of global functions was introduced (for a similar but different purpose) and studied in [Gu1, Gu2] but it needs to be generalized. We restrict ourself here to the following note. Suppose that an input to machines in a family F is a sequence of characters in an alphabet A, and the output of each F machine is a sequence of characters in an alphabet B. Fix a global algorithm for the family. Then each particular machine X computes a specific partial function f_X from sequences over A to sequences over B, and the whole family F computes a global function $\{f_X : X \text{ belongs to } F\}$.

Formalizing the notion of global algorithms provides a formalization of the notion of computable global functions. (In the classical case this corresponds to formalizing computable functions as Turing computable.) An interesting question of alternative formalizations of computable global functions (like recursive global functions, etc.) arises. We put off this question for now.

§5. Remarks

1. Intuitively speaking, uniform families of finite machines are finite or, at most, potentially infinite: like the family of virtual Pascal machines realized by existing computers and compilers, the family of feasible Pascal machines, etc. But if our metamathematics allows actually infinite objects like the set of all natural numbers then it is convenient, mathematically speaking, to deal with actually infinite families like the Platonic family of all finite Pascal machines. Actual infinity of a uniform familiy of finite machines seems to me less troublsome for semantical study of programs than actual or potential infinity of a single machine. The reason is that a program for a uniform family of finite machines runs separately on each machine. We are interested in behavior of a single (but arbitrary) member of the family. Infinity is reflected only in the range of parameters of an arbitrary member of the family.

1.1. Actually, my intuition is that even potential infinity is too much. The important difference seems to be between a finite collection which forms a mathematical set (so that the members can be counted) and a finite collection which does not: for example, English words in a given dictionary versus all English words in use.

1.2. If a computer C uses 96 bit addresses and we wish to consider the uniform family of finite machines similar to C, there is no need to generalize the address length.

2. Time is different from the resources mentioned in our examples; usually it is not bounded by hardware in an explicit way. (It is explicitly bounded in the case of a computer on batteries.) Nevetheless, it can be quite naturally incorporated into the picture. A computer rented for an hour and the same computer rented for two hours are different machines in our theory.

3. Not all infinite uniform families of finite machines are defined as families of finite versions of potentially infinite machines. Uniform families of boolean circuits, which are very popular now in complexity theory, are particulary interesting in this connection.

4. Consider the tribe of uniform families of finite Turing machines (with one or several tapes). One can formulate several versions of halting problem. Different complexity questions could be raised. If we stick to a fixed tape alphabet (which seems to be most natural for many relevant questions), then the results of conventional space complexity theory do not apply. Some questions and results related to these issues will appear in a forthcoming paper of Yuri Gurevich and Saharon Shelah.

References :

Ch A. Church, *An unsolvable problem of elementary number theory.* American Journal of Mathematics 58 (1936), 345-363.

Gu1 Y. Gurevich, *Algebras of feasible functions.* 24th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1983, 210-214.

Gu2 Y. Gurevich, *Toward logic tailored for computational complexity.* Technical report CRL-TR-3-84, University of Michigan, Jan 1983. To appear in Proceedings of European Logic Colloquium, Springer Lecture Notes in Mathematics.

Sc D. S. Scott, *Models for various type-free calculi.* In "Logic, Methodology and Philosophy of Science IV" (ed. P. Suppes *et al.*), North-Holland, Amsterdam, 1973.

St J. E. Stoy, *Denotational semantics : the Scott-Strachey approach to programming languages.* MIT Press, 1977.

Tu A. M. Turing, *On computable numbers, with an application to the Entscheidungsproblem.* Proceedings of London Mathematical Society 2, no. 42 (1936), 230-236, and no. 43 (1936), 544-546.