Algorithms in the World of Bounded Resources

Yuri Gurevich[*]

Abstract. Writing an algorithm, one has in mind some addressee (or addressees), i.e., a computing agent, a possible executor of the algorithm. In the classical theory of algorithms, one addresses a computing agent with unbounded resources. We argue in favor of a more realistic theory of algorithms which recognizes the multiplicity of addressees and the fact that their resources may be bounded. We are especially interested in the case when the resources of each relevant computing agent are bounded.

1. *The Assumption of Unbounded Resources*

An algorithm is supposed to be given by an exact prescription, defining a computational process. Writing the prescriptions, we make assumptions about the computing agent. The classical theory of algorithms makes a simplifying assumption that the resources of the computing agent are unbounded. Consider for example an algorithm for computing the factorial function in the language of primitive recursive functions:

$$\begin{aligned} FACT(0) &= 1, & FACT(x') &= MULT(FACT(x), x'). \\ MULT(x, 0) &= 0, & MULT(x, y') &= SUM(MULT(x, y), x), \\ SUM(x, 0) &= x, & SUM(x, y') &= (SUM(x, y))'. \end{aligned}$$

Notice that no provision is made for the case of insufficient computing resources. Let us express the same algorithm as a Pascal program:

```
program FACTORIAL1;
{computes the factorial f of a given natural number n}
  var f, k, n : integer;
begin
  readln(n);
  writeln('n = ', n);
```

* Partially supported by NSF grants MCS 83-01022 and DCR 85-03275.

```
    k := 0;
    f := 1;
    while k < n do
      begin
      k := k + 1;
      f := f * k
      end;
    writeln('n! = ', f)
  end.
```

When I tried to run FACTORAL1 with $n = 8$ on my Macintosh, the result was a message stating that the expression $f * k$ is out of range. I guess we are supposed to view our computers as imperfect approximations to an ideal computing agent.

Another, more realistic point of view is that a program may be executed by different computing agents, and some (or even all) of them may have bounded resources. When you write a program (or create a programming language), think about the family of (real or virtual) computing devices that may run your program (or use your language), rather than one ideal computing agent with unbounded resources.

Imagine that FACTORIAL1 is a link in a chain of computations, and its output is the input for another algorithm. To give a standard form to the output of FACTORIAL1, one can elaborate FACTORIAL1 as follows:

```
program FACTORIAL2;
  var f, k, n integer;
begin
  readln(n)
  writeln ('n = ', n)
  k := 0
  f := 1;
  while (k < n) and (f <= maxint /(k + 1)) do
    begin
    k := k + 1;
    f := k * f
    end;
  if k < n then
    writeln ('n! exceeds maxint')
  else
    writeln ('n! = ', f)
end.
```

Is FACTORIAL2 an exact prescription for computing the factorial of an arbitrary natural number n (given in the decimal notation) in the world where resources may

be bounded? No, FACTORIAL2 does not ensure the rejection of inputs exceeding maxint in any standard way. Further modification is needed. In particular, we need to change the type of n. Pragmatically speaking, it may be even more important to change the type of f; that would allow us to compute $f = n!$ for many more inputs n. The assumption of unbounded resources simplifies the design of an algorithm. Taking bounded resources into account makes us work harder, but the resulting prescription may be more exact and more useful. Even though this phenomenon is well known, it is not reflected properly in the theory of algorithms.

Sometimes it is easier to explain one's arguments in discussion. To this end, please allow me the liberty of introducing an imaginary opponent, a skeptical graduate student.

Opponent: "In spite of your criticism, I think that the abstraction of unbounded resources is very useful."

It is indeed. Nevertheless its applicability is restricted. It reflects computations where the effort to acquire additional resources—whenever it is necessary—is negligible; after acquiring additional resources, the computation resumes as if it was never interrupted. This mode of computation reflects very well routine computations of human computers. In his celebrated paper (Turing *1936-7*), Turing analyzed a routine computation of a human computer with a pile of sheets of paper before him/her. The pile was abstracted as the Turing tape. It is easy to believe that the supply of paper exceeds the patience of the human computer.

Here is another situation where the abstraction of unbounded resources is justifiable. Consider a relatively small computation process living in a huge resource-sharing environment managed by a flexible operating system. Whenever the process needs more space or another processor or whatever, the operating system quietly and efficiently provides the necessary additional resources.

But, as we saw, it is easy to find situations where the abstraction of unbounded resources is not justifiable. Imagine that your Macintosh runs out of memory during a computation. What can you do?

Opponent: "I can buy a hard disk; it will take care of all of my needs for a while. But I see your point. It is not negligible to acquire additional resources in the case of the Macintosh. What if I keep inserting and removing floppy disks (I can always buy more of those)?"

Then you become essentially a part of a new machine which contains a Macintosh. To make the point, let me consider the following ridiculous example. Imagine that you have bought a personal Turing machine. After a while you may run out of tape. You go and buy some additional tape. Your nearest shop may run out of tape. Then you have to figure out where to go. You use your brain to solve all these problems; the transition table of your Turing machine does not give you all the answers.

2. Machines With Bounded Resources

A possible alternative to the assumption of unbounded resources is an assumption that the resources of all relevant computing devices are bounded.

Opponent: "For some purposes you need machines with unbounded resources. For example, no single machine with bounded resources is able to provide an adequate operational semantics for a high-level programming language like Pascal."

It is true that no machine with bounded resources is able to provide an adequate operational semantics for Pascal. But we may consider a family of machines with bounded resources. This creates a situation which is similar to that in logic where the meaning of a first-order formula is given by a family of structures. Each structure of the appropriate signature gives only a local meaning to the formula; the global meaning of a formula is given by all structures of the appropriate signature. A family of Pascal machines with bounded resources is sketched in Gurevich *1987*. Each machine gives a local meaning to a Pascal program, and the global meaning of the program is given by the whole family of Pascal machines. When you write a Pascal program, you may imagine the addressee, i.e., the computing agent, as an ideal machine with unbounded resources. Alternatively, you may think about a machine with bounded resources. This should not make your program implementation dependent. Proving a theorem about finite groups, you may speak about the group of discourse; this does not mean that you prove your theorem for just that one group.

Opponent: "Please, give me a simpler example of a family of machines with bounded resources."

Let T be a Turing machine. Consider bounded-tape versions of T with or without special end-of-tape marks.

Opponent: "The computation of any bounded-tape version of T without the end-of-tape mark is an initial segment of the computation of T, and the cut-off point is irrelevant to the meaning of the program of T. I would prefer to consider the computation of T itself rather then a pretty arbitrary collection of initial segments of the computation."

Yes, in many cases, a machine with unbounded resources gives a cleaner operational semantics. But not always. Machines with bounded resources may know their resources and utilize this knowledge. A program for bounded-tape machines may use the end-of-tape mark for different purposes; it may divide the tape equally into a left and a right part and execute two different processes in a time-sharing fashion. More convincing examples of how machines with bounded resources may use the knowledge of their resources come from real life. Think about operating systems. In particular, think about an operating system which runs on many computers and starts with an inventory of the available resources. Of course, this program (the operating system) can be modeled by a machine with unbounded resources, but this is not necessarily the best way to provide an operational semantics to the program.

Some high level programming languages—such as FORTRAN—do not reflect any information about the resource bounds of the implementation. On the other hand, Pascal has an implementation-defined constant maxint. Implementation-defined constants found in ADA or APL reflect considerable information about the resource bounds. There seems to be a trend to include 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 machines with bounded resources is closer to the intended meaning of many programs than the operational semantics based on machines with unbounded resources. Enriching a programming language with 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.

3. *Dynamic Structures*

I believe we need a computational model appropriate to deal with machines with bounded resources. To start, it may be reasonable to make some simplifying assumptions. For example, one may suppose the following:

A machine with bounded resources is a finite-state machine which works in discrete time. The states form a legitimate mathematical set. The machine may be nondeterministic and able to do many things at once, but all parts of it obey one clock. The set of states, the input alphabet, the output alphabet and the transition function do not change in time.

Opponent: "Isn't any finite collection a legitimate mathematical set?"

The answer depends on how you define finiteness. But a subcollection of a perfectly good finite set may be not a legitimate mathematical set. For example, the collection of English words is not a legitimate mathematical set; the membership relation takes more than two truth values. Questions like "Is the number of elements even?" or "Is the number of elements prime?" should be meaningful for legitimate finite mathematical sets.

Opponent: "There is already a computational model of machines with bounded resources: the classical theory of finite-state machines?"

Unfortunately, the classical theory of finite-state machines is not adequate to deal with real computers or virtual computing devices like Pascal interpreters. One can view Apple's Macintosh as a finite transducer (or even a finite automaton), but it is unfeasible to write down the full transition table (or a regular expression) for the Macintosh: there are too many states. The problem arises to elaborate the notion of finite-state machines into a notion which is more appropriate to deal with real computers and complicated virtual computing devices. One problem with formalizing

machines and some other objects of interest in computer science is their dynamic character.

Opponent: "What other objects?"

Consider for example relational data bases. They are defined often as collections of mathematical relations (predicates). But collections of mathematical relations do not evolve in time, whereas data bases do. Traditional mathematical structures—such as graphs and groups—are static. There is a tendency to formalize dynamic processes by means of static structures; a common trick is to introduce time as another dimension. You can trace that tendency also in the field of foundations. Think about the process of "cleaning" mathematics from constant and variable quantities in favor of functions.

Opponent: "If static structures have served us so well for so long in so many different applications of mathematics, why shouldn't they be appropriate in computer science? What is so special in the applications of mathematics to computer science versus, say, the applications of mathematics to physics?"

The special features of many computer applications include discrete time, the existence and relative simplicity of static configurations and the relative simplicity of transitions from one static configuration to another. (The relative simplicity of static configurations and transitions is often accompanied by an overwhelming number of states and an overwhelming complexity of the whole process.)

Opponent: "I can give you examples of neat mathematical structures of dynamic nature: Turing machines, random-access machines."

I would like to see a more general notion of dynamic structures. Complicated abstract machines, such as virtual Pascal, Ada, and Smalltalk machines, should be covered. I believe we need a theory of dynamic structures. Operational semantics for programming languages would be one application for such a theory.

Opponent: "Plotkin *1981* speaks about transition systems. A transition system is a set of elements (called configurations) with a binary relation (called the transition relation). You were willing to restrict the discussion to the case of discrete time when all parts of the machine obey one discrete clock. The notion of transition systems seems sufficiently general for that case."

I agree. However, transition systems give us static representations of dynamic structures. In the case of a finite-state machine, the transition system is simply the presentation of the machine as a finite automaton, and we know that the theory of finite automata is inadequate to deal with complicated dynamic structures.

It is natural to see configurations of a dynamic structure as static structures. The evolution of a dynamic structure is governed by transition rules. The notion of a family of dynamic structures may be restricted by the requirement that members of the family have the same transition rules (so that there is a single finite formulation of transition rules appropriate to all members of the family).

Different classes of dynamic structures may be defined by imposing syntactical restrictions on transition rules, by allowing or forbidding the evolution of the signature (the language) of the current configuration, by allowing or forbidding the creation of new universes (sorts, types) and the elimination of old ones, and so on.

The notion of dynamic structures is briefly discussed in Gurevich *1987*. The primary example there is a family of Pascal machines. A detailed description of Modula-2 machines will appear in Gurevich and Morris *1987*. The case of Modula-2 is more interesting because of some parallelism allowed there. We are also looking into Ada and Smalltalk from that point of view.

4. *The New Thesis Problem*

The problem is to define a modest class (let us call it U for 'universal') of abstract machines with bounded resources such that every "real" computing device with bounded resources can be closely simulated by an appropriate U-machine of comparable size, and every family of computing devices with bounded resources can be appropriately simulated by a family of U-machines.

It is easy to come up with a cheap solution if one ignores the quality of simulation or the size of the simulating machine (Gurevich *1984*), but a good solution should have important applications. The problem was briefly discussed in Gurevich *1987*, and we intended to take up the issue here but the limitation of time has proved to be too severe. The new thesis problem will be discussed in Blass and Gurevich (in preparation).

By the way, bringing the issue of simulation into the open invites attempts to improve Turing's thesis by imposing a restriction on the allowed simulations and possibly using different machines instead of Turing machines. One may be interested in polynomial-time simulations, linear-time simulations, real-time simulations, etc. There seems to be a consensus among computer scientists that Turing machines are sufficiently good simulators from the point of view of polynomial time. In particular, the polynomial-time version of Turing's thesis seems to be accepted by many. However, one should be a little careful about what is a legitimate computing device in the polynomial version of Turing's thesis. For example, Schönhage *1979* constructed a deterministic random access machine with built-in arithmetical operations which decides the satisfiability of boolean formulas in conjunctive normal form (a known NP complete problem) in polynomial time with respect to the so-called uniform cost criterion. (Under the uniform cost criterion each instruction requires one unit of time whereas a more realistic logarithmic cost criterion takes into account the limited size of machine words in real random access machines; see Aho, Hopcroft, and Ullman *1974*.)

With respect to linear or real time, Turing machines are inadequate simulators. Kolmogorov and Uspenski *1958* introduced more flexible machines. They wrote humbly that the only purpose of their paper was simply to re-examine for themselves Church-Turing's thesis. It is possible nevertheless that they had in mind something more ambitious like the following thesis: Every sequential computing device can be simulated in real time by an appropriate Kolmogorov-Uspenski machine.

The notion of real time simulation is defined as follows in Schönhage *1980*: A machine M' is said to *simulate* another machine M *in real time* if there is a constant c such that for every input sequence x the following holds: if x causes M to read an input symbol, or to print an output symbol, or to halt at time steps

$$0 = t_0 < t_1 < \ldots < t_k,$$

respectively, then x will cause M' to act in the very same way with regard to those externally visible operations at time steps

$$0 = t'_0 < t'_1 < \ldots < t'_k,$$

where $(t'_j - t'_{j-1}) \leq c(t_j - t_{j-1})$ for $1 \leq j \leq k$.

Schönhage *1980* introduced a machine model which is similar to but different from the Kolmogorov-Uspenski model. He "posed the intuitive thesis that this model possesses extreme flexibility and should therefore serve as a basis for an adequate notion of time complexity". A related and blunter thesis is: Every sequential computing device can be simulated in real time by an appropriate Schönhage machine.

To get rid of the restriction to sequential devices, one may use parallel versions of both Kolmogorov-Uspenski machines and Schönhage machines.

Gandy *1980* formulated four assumptions about computing devices, put forward a thesis that every deterministic mechanical device satisfies the four assumptions, and proved that whatever can be calculated by a device satisfying the four assumptions is Turing computable. Much more detailed analysis is needed to argue for or against the polynomial-time version of Turing's thesis or either of the real-time simulation theses. The task does not seem easy. It should involve in particular a closer examination of the assumption of unbounded resources.

Acknowledgements. I am very grateful to Egon Boerger, Bernie Galler, Kit Fine, Saharon Shelah and especially Andreas Blass for enjoyable and useful discussions.

References

Aho, A.V., J.E. Hopcroft, and J.D. Ullman

1974 *The Design and Analysis of Computer Algorithms.* Reading, MA: Addison-Wesley (1974).

Blass, A., and Y. Gurevich

in prep. A new thesis (tentative title), in preparation.

Church, A.

1936 An unsolvable problem of elementary number theory. *Am. J. Math.* **58** (1936) 345–363.

Gandy, R.

1980 Church's thesis and principles for mechanisms. In: *The Kleene Symposium,* eds. J. Barwise et al., pp. 123–148. North-Holland Publ. Co. (1980).

Gurevich, Y.

1984 Reconsidering Turing's thesis (toward more realistic semantics of programs). Tech. report CRL-TR-36-84, University of Michigan (1984).

1985 A new thesis. AMS Abstracts, Aug. 1985, p. 317.

1987 Logic and the challenge of computer science. In: *Current Trends in Theoretical Computer Science,* ed. E. Börger, pp. 1–57. Rockville, MD: Computer Science Press (1987).

Gurevich, Y., and J.M. Morris

1987 Algebraic operational semantics for Modula-2. Tech. Report CRL-TR-10-87, July 1987, University of Michigan.

Kolmogorov, A.N., and V.A. Uspenski

1958 On the definition of an algorithm. *Uspekhi Mat. Nauk* **13** (1958) 3–28 (Russian); *AMS Translations* **29** (1963) 217–245.

Plotkin, G.D.

1981 A structural approach to operational semantics. Tech. Report DAIMI FN-19, Aarhus University, Aarhus, Denmark.

Schönhage, A.

1979 On the power of random access machines. In: *Automata, Languages and Programming,* ed. H.A. Mauer, pp. 520–529. Berlin: Springer-Verlag (1979).

1980 Storage modification machines. *SIAM J. Comp.* **9** (1979) 490–508.

Turing, A.M.

1936-7 On computable numbers, with an application to the Entscheidungsproblem. *P. Lond. Math. Soc. (2)* **42** (1936-7) 230–236; A correction, ibid. **43** (1937) 544–546.