What is a “Pointer Machine”?

Amir M. Ben-Amram*

April 19, 1995

Abstract

A “Pointer Machine” is many things. Authors who consider referring to this term are invited to read the following note first.

1 Introduction

In a 1992 paper by Galil and the author we referred to a “pointer machine” model of computation. A subsequent survey of related literature has produced over twenty references to papers having to do with “pointer machines”, naturally containing a large number of cross-references. These papers address a range of subjects that range from the model considered in the above paper to some other ones which are barely comparable. The fact that such different notions have been discussed under the heading of “pointer machines” has produced the regrettable effect that cross references are sometimes found to be misleading. Clearly, it is easy for a reader who does not follow a paper carefully to misinterpret its claims when a term that is so ill-defined is used.

This note is an attempt to rectify the situation. We start with a survey of the different notions that have appeared under the heading of “pointer machines,” in order to create a common terminology which allows the necessary distinctions. The main distinction is between *Abstract Machines*, which are described in Sections 2 and 3, and *Pointer Algorithms*, which are described in Section 4. Under each heading, finer distinctions are made and some pointers to literature are provided. We conclude with suggestions on how to avoid the confusion in future literature as well as in references to the past.

*DIKU, Department of Computer Science, University of Copenhagen. Work supported by a scholarship from the Josef and Regina Nachemsohn Memorial Foundation.

2 Abstract Machines: Atomistic Models

Abstract machines were introduced to Computer Science, at the first place, as a means of formalizing the notion of "algorithm". The main example is the Turing Machine; another model introduced with this goal in mind is the Kolmogorov-Uspenskii machine, to be described in more detail below. Such machines have "input/output media," conventionally described as tapes on which symbols from some finite alphabet are written: thus algorithms in this class compute functions from Σ^*, where Σ is a finite alphabet, into Σ^*. The fact that this is common to all models has the following important consequences. First, it is possible to prove that all of them compute the same class of functions, thus giving supportive evidence to Church's Thesis. Secondly, it is possible to compare the *power* of different models by studying the relationship between the complexity of problems on the different models, or the complexity of *simulating* one model by another.

The differences between the models are expressed in their *internals,* which are typically specified by a *storage structure* and a set of operations that can be put together to form a *program.* These operations include input/output operations, operations that modify or inspect the internal storage etc.

Much as the input/output language of these machines is the language of strings over a finite alphabet, so is it often desired that the internal operations of the machines have a "discrete" nature, since this seems to support the claim that they represent effective algorithms: this has been argued by Kolmogorov and Uspenskii [10] as follows.

> The mathematical notion of Algorithm has to preserve two properties...
>
> 1. The computational operations are carried out in discrete steps, where every step only uses a bounded part of the results of all preceding operations.
> 2. The unboundedness of memory is only quantitative: i.e., we allow an unbounded number of elements to be accumulated, but they are drawn from a finite set of types, and the relations that connect them have limited complexity.

Schönhage [12] referred to abstract machines that have these properties as *atomistic.*

The form of storage suggested by Kolmogorov and Uspenskii as well as by Schönhage is a collection of *nodes* that are interconnected by *pointers* (these are Schönhage's terms, which agree with programming practice). All machine models that have this type of storage structure can be described as *pointer machines*. But they are far from being the same. The following paragraphs contain a list of all machine models known to the author that might be grouped under the heading *atomistic pointer machines*. For simplicity in giving details, we consider in what follows the input/output alphabet to be binary.

2.1 Kolmogorov-Uspenskii Machines

The Kolmogorov-Uspenskii model [10] represents storage as an undirected finite graph, in which every edge is has one of a finite set of *labels*, and edges incident to the same node must have different labels. This naturally implies a finite bound on the degree of the graph. At any moment, one specific node is designated as *the active node*. The neighborhood of the active node of some fixed radius is called *the active zone*. In the original formulation, each step of the machine consists of applying a fixed transformation that maps every possible form of the active zone into a (generally different) subgraph that has the same "boundary" (so the rest of the storage graph remains accessible). For example, the active zone can be contracted, with the effect that nodes that were previously too far are pulled inside for inspection.

A more conventional programming formulation, which serves to bring this model under the common framework described above, defines a program to be a sequence of instructions that include the following types: unconditional branch, input, output, conditional branch and storage modification. The conditional branch instruction specifies two strings of labels, not longer than the radius of the active zone; the destination of the branch is determined by whether the two paths starting at the active node, and specified by the given labels, lead to the same node. A storage modification instruction can create a new node, and redirect an edge (there is a certain default setting for the edges of a newly created node).

2.2 Storage Modification Machines

The SMM model, introduced by Schönhage [12], differs from the KUM by representing storage as a directed finite graph; apart from that the description in the last paragraph applies. Schönhage did not include the requirement of a fixed radius for the active zone, but mentioned that this change may give a “more precise and realistic” measure of running time.

In contrast with KUMs, here the finite size of the set of labels only restricts the *out-degree* of nodes; an unbounded number of pointers may lead to a single node.

Observe that using a set of two distinct labels, i.e., nodes of out-degree two, any algorithm using a larger out-degree can be simulated with at most a constant factor loss in running time as well as in the number of nodes in storage. This is true for both the SMM and the KUM.

2.3 Knuth's Linking Automaton

The Linking Automaton was defined by Knuth [9] as a model that may help to understand better the capabilities of algorithms that operate on linked structures. It is defined the same as the SMM except that every node has, in addition to the fixed number of outgoing pointers, also a fixed number of *value fields*. Each value field stores one symbol out of a given alphabet. The program has the capabilities of moving symbols around and of comparing value fields for equality of contents.

2.4 APLM and AFLM

The above acronyms stand for Atomistic Pure-LISP Machine and Atomistic Full-LISP Machine, respectively. They are the atomistic versions of the corresponding PLM and FLM models [3], which are not atomistic. The AFLM is the same as Knuth's automaton, except that every node contains precisely two fields, which can store either pointers or alphabet symbols (one may say that Knuth's automaton has a strictly typed language while the AFLM does not). The APLM is the same model with the following restriction: the fields of a node may be set only at the time it is created. Once set, they cannot be altered. This implies that the graph created is acyclic, as nodes can be ordered according to the time of their creation and pointers will only point to earlier nodes.

The names of these models are due to the fact that the above restriction is characteristic to a subset of the LISP language known as *pure* LISP. The advantages of the restricted model are that functions do not have side effects, and the semantics are simpler (as well as implementation issues such as garbage collection).

2.5 General Atomistic Pointer Machines

Shvachko [13] suggests to study machines which operate on graphs, under different specifications of the class of allowable graphs. He mentions the KUM and SMM as particular examples, where the former is obtained by selecting undirected graphs of bounded degree and the latter by selecting directed graphs of bounded out-degree. In addition to these models, he considers "tree pointer machines." These are naturally obtained by restricting the graphs to be bounded-degree trees.

2.6 Jones' `I` Language

An interesting observation on the "abstract machine" way of formalizing algorithms, is that once a programming language is defined, with proper semantics and cost functions (to evaluate complexity), a computational model has been fully specified. In a model thus obtained, the notion of *algorithm* is identified with that of a *program,* and not of *machine.*

Jones [8] took this approach, and presented two programming languages, `I` and $\mathtt{I}^{su}$, that can be used as general computational models. He used denotational semantics; however, the natural choice of a model for operational semantics is, in the case of $\mathtt{I}^{su}$, an SMM of out-degree two. This makes the $\mathtt{I}^{su}$ model a programming-oriented alternative to the SMM. It only differs by the restriction of the out-degree (which could be easily relaxed), and the fact that the $\mathtt{I}^{su}$ language is structured, while the SMM is programmed using `goto`s.

The `I` language is a restricted version of $\mathtt{I}^{su}$, corresponding to the APLM mentioned above; namely, in this language, the destination of a pointer may only be set when the node it emanates from is created.

2.7 Relationships among the Models

The following are simple observations: the AFLM can be seen as a restricted version of Knuth's automaton (with some programming nuances), and so can the SMM and Jone's $\mathtt{I}^{su}$. On the other hand, the three other models can simulate Knuth's automaton in real time, using a constant-factor overhead in storage (see [12] for a formal definition of real-time simulation). The constant factor in both time and space complexities depends on the size of the automaton's alphabet.

Due to the above equivalences, one model may be chosen as a representative for the above group, and historically this has been the SMM.

The APLM model is a restriction of the AFLM, same as the `I` model is to $\mathtt{I}^{su}$. Thus the restricted models are not stronger, but possibly weaker, than the SMM. Whether they actually are weaker is yet unknown. Similarly, it is easy to see that the KUM can be easily and efficiently simulated by the SMM; it may be weaker than the latter, but this is also open.

2.8 Pointers to Literature

Among the above models, the one most extensively studied is the SMM. Most of the results regarding this model are reported in [15]. Some of these results also pertain to the other, possibly weaker models, while some make use of the full power of the SMM. An interesting discussion of the KUM may be found in [6]. Jones [8] presents several results on his `I` and $\mathtt{I}^{su}$ models.

3 Abstract Machines: High-Level Models

Recall that machines described in the last section only manipulate a single type of "values," namely symbols from the chosen alphabet. Let us extend the framework by replacing the finite alphabet by an arbitrary set of data (e.g., the integers) which is furthermore structured (i.e., is the domain of some algebraic structure). Let us call this set our *domain* and further equip the model by a set of instructions for effecting various operations that are defined on elements of the domain (e.g., addition and subtraction of integers). All instructions will count as a single step in the calculation of time complexity. Consider the input and output operations too as transporting an arbitrary element of the domain in one step instead of a symbol from a finite set. In accordance with programming-language practice, we refer to the algebraic structure (domain + operations) as a *data type*.

We refer to a computational model thus obtained as a *high-level* machine: clearly, it does not belong to the atomistic class of Section 2, and thus the rich theory of computation developed for these models does not always apply. Moreover, models of different domains compute functions out of different function spaces. It is thus not possible in general to compare such models for efficiency. But it is quite possible if the models share the data type and only differ on "intrinsic" features such as the style of programming or the structure of memory.

The main justification for high-level models is that they are closer to the way computer algorithms are described and analyzed in practice. This position was advocated in [3]. This paper also claimed that, to to get closer to this goal, it is worthwhile to use models that reflect the capabilities of useful programming languages. In this vein it introduced two computational models called the $\mathcal{T}$-FLM (Full LISP Machine) and $\mathcal{T}$-PLM (Pure LISP Machine). It may be simplest to define these models as the high-level extensions of the corresponding atomistic machines (Section 2.4), where $\mathcal{T}$ stands for the *data type* used in the extension. The name LISP *Machine* (LM) is used in this paper when the distinction between the two models may be neglected.

These models have not received much theoretical treatment; the best account so far is [1].

Interestingly, the first use of the term *pointer machine* (known to the author) refers to a model quite similar to the FLM [14]. The main distinction between Tarjan's presentation of the model and the one of [3] are that Tarjan neglected to endow his model with input/output instructions; this is not a coincidence, as the main result of that paper is better interpreted not in the framework of abstract machines, but in a framework of *classes of data-structure algorithms* (as the title of the paper actually suggests). This is the issue of the next section.

4 Pointer Algorithms

4.1 Definition

When a class of problems is too hard to analyse in the most general fashion, one often chooses to concentrate on a particular means of solution that is natural for the problems at hand, and study the capabilities of this means in isolation. A well-known example is the study of comparison algorithms. In this area, we ask for the number of comparison operations that are necessary for solving certain order-related problems (e.g., sorting). We thus *assume* that the data are amenable to pairwise comparisons, and that the algorithm only uses information gained by such comparisons. We *ignore* any other work that a program, implementing this algorithm, might have to perform.

The study of *pointer algorithms* falls into the same framework. Here the problems we are interested at may be generally described as *data-structure problems*. A primary example, and the one first considered in this framework, is the *maintenance of disjoint sets*. In this problem, the data structure represents a collection of disjoint, named subsets of some universe U. Operations include the query: given an element of U, what set contains it (known as *find*) and various update operations, such as merging two sets into one (known as *union*).

A common representation for such structures is a directed graph, including a node for every element and set name, such that it is possible to go from an element to its set name by following arcs [1]. This graph could be implemented as a network of records in memory with the arcs represented by pointers from one record to another; however, we are not interested in implementation details. Instead, let us *assume* that our data structure is a graph of the above kind; we can then ask for the complexity of performing data-structure operations as measured by the number of modifications, or accesses, to the graph.

We thus define the class of *pointer algorithms* as algorithms for performing some given data structure operations on a graph representation of the structure. Clearly, the concrete relationship of the graph to the represented structure has to be defined specifically for each problem; in the example of disjoint-set maintenance problems, we require each element as well as set-name to be represented by a node. In general this relationship should be of such nature that the data-structure operations accept pointers to nodes in the graph as input and deliver their output in form of such pointers. For instance, the *find* operation accepts a pointer to a node that represents an element and has to deliver a pointer to the node that represents the appropriate set name. Note that sometimes there is more than one input pointer (e.g., two pointers are input for a *union*); and sometimes there is more than one output (as in a query that reports all elements satisfying some condition). To measure the *time complexity* of a pointer algorithm we consider the number of arcs followed by the algorithm, i.e., the length of the shortest (possibly branching) paths leading from the set of input nodes up to every member of the set of output nodes. To this we add the number of arcs added to or removed from the graph during the operation. Note that, while a particular algorithm may traverse the same arc more than once, we only count the number of arcs used for the cost of the algorithm. This may underestimate the cost of an algorithm, but such underestimation is inherent in the use of this kind of model, and it turns out that tight lower bounds are nonetheless obtained for various problems. Space complexity is sometimes measured by the number of nodes in the graph.

An even more precise way of defining this model is as an implementation of a given *abstract data structure* (e.g., a union-find structure) by means of the abstract data structure *directed graph*. We impose the restriction, that

[1] We use the term *arc* for a directed graph while *edge* is used for undirected graphs.

the abstract data types “element” and “set name” must be implemented as the data type “pointer to node”. The abstract data structure *directedgraph* provides a well-defined set of operations: add node, add pointer, get pointer and delete pointer. The time complexity of the implementation is defined by the number of these operations, and the space by the number of *add node* operations.

We finally remark that, except as noted below, it is assumed that the out-degree of the graph is bounded by a constant independent of the problem instance (for example, it cannot be a function of the number of elements). We call this the *fixed degree requirement*.

4.2 Separable Pointer Algorithms

In the context of the disjoint set maintenance problem a special type of pointer algorithms has been considered, known as *separable pointer algorithms*. This class of algorithms is defined by imposing the *separability condition*: after every operation is completed, the graph can be partitioned into disjoint subgraphs that correspond to the current disjoint sets. The subgraph corresponding to a given set contains the node for its name. No edge leads from one subgraph to another.

It turns out that for such algorithms tight lower bounds may be obtained even if the fixed degree requirement is omitted. Therefore, the class of separable pointer algorithms is defined without this requirement.

4.3 Representative Results

Since the above description has been somewhat abstract, we mention some particular results on pointer algorithms for three different data-structure problems. All the results mentioned are lower-bound results, because it is such proofs that promoted the use of this model.

Disjoint Set Maintenance. Tarjan [14] showed that any separable pointer algorithm must require, in the worst case, $\Omega(m\alpha(m,n))$ time to execute a sequence of $m \geq n$ find operations and $n-1$ unions on n elements.

Disjoint Set Maintenance has many variants and many papers in this area used the pointer algorithm model. See [5] for a survey of this area as of 1991.

Queries on Rooted Trees. Harel and Tarjan [7] show that any pointer algorithm, that implements *nearest common ancestor* queries on binary trees of n nodes, must take $\Omega(\log\log n)$ time in the worst case.

Reporting Points in Space. Chazell [2] proves a time-space tradeoff for orthogonal range queries (report all points of a given set that fall within a query box), when carried out by a pointer algorithm.

Many pointer-algorithm lower bounds have been broken by RAM algorithms. For instance, [7] shows a constant-time solution on a RAM. Gabow and Tarjan [4] give a linear-time RAM algorithm for a case of the union-find problem to which Tarjan's lower bound applies.

4.4 Discussion

The important message of this section is the distinction between the pointer-algorithm model (or more generally, the isolated-means-of-solution approach) and the abstract machine approach to computational complexity of problems. An essential characteristic of the abstract machine approach is the machine-independent representation of the problem, as a function over strings, integers etc. This machine-independence allows us to ask for the complexity of the same problem on different models and in fact to compare different models for their power. This approach obviously applies to off-line problems, where the input is given at one time, but it can also be used with data-structure maintenance problems, or more generally on-line problems. Here the computation can be viewed as a sequence of "operations", and the input to the next operation is (generally) given only after the previous operation has been completed.

To contrast, in the pointer-algorithm model the problem definition is all but machine independent: in fact, it makes use of *pointers*, an entity which has no external representation. The fact that input and output are given as pointers is essential to the model; for instance, consider a union-find structure representing n singleton sets, so that there are n nodes that represent set names. If a node's name had to be given to the *find* procedure in an external representation, say a string or a number, and the procedure only maintains a finite number of pointer variables, then $\Omega(\log n)$ arcs would have to be followed in most cases just to locate the desired set-name node.

Thus such algorithms are a natural model only for data-structure problems, where we may assume that the procedures we consider are intended for use by a "client" program: this program may obtain a pointer from one procedure call and at a later time pass it to the next. In fact, this "client" may even be the one that supplies the pointers in the first place: think of a program that manages a set of records which, among other uses, have to be organized in a union-find structure. The requirements of this user naturally call for a pointer algorithm.

The historical connection of Pointer Algorithms to machine models such as the SMM and the LISP Machines stems from the fact that it is natural to implement a Pointer Algorithm using the capabilities of such machines. It is for this purpose that these abstract machines were mentioned in papers, whose essential contribution relates to Pointer Algorithms, such as Tarjan [14] and Paige [11]. In fact, Tarjan describes in detail an implementation of an optimal algorithm for his problem which uses only the capabilities of the SMM. Taking another point of view, Paige's contribution shows that if the client program, makes use of a certain repertory of set operations, then pointer algorithms are as efficient as any other data-structuring technique.

4.5 Pointers to Literature

Many of the publications on pointer algorithms have to do with the set-union problem. See [5] for a survey of this area as of 1991.

5 Conclusion

In this short survey we have seen two essentially different notions, i.e., pointer algorithms and abstract machine models, of which there are also different variants; unfortunately, all have been referred to as "pointer machines". As mentioned in the introduction, the multiple uses of "pointer machine" have caused some confusion in the past (examples appeared in print but will not be cited). To counter this problem, I strongly recommend to authors of related papers in the future to *avoid the use of this term* in favour of terms that have not acquired a confusing multitude of denotations. The above survey used a unique term for every notion mentioned, and it is my hope that it may serve as a source of precise terminology. In fact, I have only used the words "pointer machines" in the qualified term "atomistic pointer machines," to which I have found no satisfactory substitute; but as long as the qualified term is used, no confusion can arise.

As for past literature, the best we can do is to classify old papers using well-defined terms. The result of this effort, undertaken by the author, will be an annotated bibliography covering all the areas considered in this note (and others if discovered). In order to make this work as complete as possible, *readers who know of related references are urged to write to the author* at the Email address above. The bibliography will be made available both electronically and in print.

References

[1] A. M. Ben-Amram, "On the power of random access machines," thesis, Tel-Aviv University, 1994.

[2] B. Chazelle, "Lower bounds for orthogonal range searching," *J. of the ACM* 37 (1990), 299–212.

[3] A. M. Ben-Amram and Z. Galil, "On Pointers versus Addresses," *J. of the ACM* 39:3 (1992).

[4] H. Gabow and R. E. Tarjan, "A linear-time algorithm for a special case of disjoint set union," *J. of Computer and System Sciences* 30 (1985) 209–221.

[5] Z. Galil and G. F. Italiano, "Data structures and algorithms for disjoint set union problems," *ACM Computing Surveys* 23:3 (1991), 319–344.

[6] Y. Gurevich, "Kolmogorov machines and related issues: the column on Logic in Computer Science," *Bull. of the EATCS* 35 (1988), 71–82.

[7] D. Harel and R. E. Tarjan, "Fast algorithms for finding nearest common ancestors," *SIAM J. Comput.* 13 (1984), 338–355.

[8] N. D. Jones, "Constant time factors do matter," *Proc. 25th Annual ACM Symp. on Theory of Computing* (1993), 602–611.

[9] D. E. Knuth, *The Art of Computer Programming*, Vol. 1: *Fundamental Algorithms*. Addison-Wesley, Reading, Mass. 1968, 1973.

[10] A. N. Kolmogorov and V. A. Uspenskii, "On the definition of an algorithm," *Uspehi Mat. Nauk.* 13 (1958), 3–28. English translation in *AMS transl.* II Vol. 29 (1963), 217–245.

[11] R. Paige, "Real-time simulation of a set machine on a RAM," in *Computing and Information ICCI '89*, Vol. 2, R. Janicki and W. Koczkodaj (eds.), 1989.

[12] A. Schönhage, "Storage modification machines," *SIAM J. Comput.* 9:3 (1980), 490–508.

[13] K. V. Shvachko, "Different Modifications of Pointer Machines and their Computational Power," *Proc. Symp. Mathematical Foundations*

of Computer Science (MFCS) 1991, LNCS 520, Springer, Berlin, 426–435.

[14] R. E. Tarjan, "A class of algorithms which require nonlinear time to maintain disjoint sets," *J. Comput. System Sci.* 18 (1979), 110–127.

[15] P. van Emde Boas, "Machine models and simulations," in: *Handbook of Theoretical Computer Science*, Vol. A, Elsevier, 1990, 1–66.