RAM Simulation of BGS model of Abstract State Machines*

Comandur Seshadhri[1], Anil Seth[2], and Somenath Biswas[2] **

[1] Department of Computer Science
Princeton University, Princeton NJ 08544, USA,
[2] Department of Computer Science and Engineering
IIT Kanpur, Kanpur 208016, India

Abstract. We show in this paper that the BGS model of abstract state machines can be simulated by random access machines with at most a polynomial time overhead. This result is already stated in [BGS99] with a very brief proof sketch. The present paper gives a detailed proof of the result. We represent hereditarily finite sets, which are the typical BGS ASM objects, by membership graphs of the transitive closure of the sets. Testing for equality between BGS objects can be done in linear time in our representation.

1 Introduction

Blass, Gurevich, and Shelah defined in [BGS99] a version of abstract state machines, now known as BGS ASMs. BGS ASMs provide us with a rich model of computation that can deal directly with a structure as input (rather than its string encoding, as would be the case with a Turing machine), and can work with the universe of hereditarily finite sets over the domain of the input structure. The computation model, BGS ASM, when viewed as a logic, is one where order is not given as a logical primitive, therefore this model defines generic, i.e., isomorphism closed, properties of arbitrary unordered first order structures. BGS ASMs are of further interest because a natural polynomial restriction on computational resources of BGS ASMs defines a subclass of generic polynomial time properties, or equivalently queries, which includes many queries not expressible in known logics for fragments of **P**. At the same time, due to their rich syntax, BGS ASMs are arguably more natural to use. As in [BGS99], we use the term *PTime BGS ASMs* to denote the class of BGS ASMs with restricted computational resources referred above. For more on comparison of queries expressed by *PTime BGS ASMs* with other common logics for fragments of **P**, see [BGS99].

We provide in this paper a step-by-step method of how to implement algorithms expressed in BGS ASMs by random access machines (RAMs). Our

* This work was partially supported by a grant from Microsoft Research to the Computer Science and Engineering Department, IIT Kanpur.

** E-mail address for correspondence on this work: `sb@cse.iitk.ac.in`

implementation shows that BGS ASM programs can be simulated by RAM programs with at most a polynomial time overhead. As it is well known that all RAM programs can be simulated by Turing machines, again with at most a polynomial time overhead [AHU74], we conclude that PTime BGS ASMs can be simulated by polynomial time Turing machines.

We should mention that the above conclusion will not be unexpected to the ASM community. Indeed, [BGS99] states this conclusion as Theorem 3, and also provides a brief proof sketch. What may be seen as our contribution is that we make all the details explicit which make the above conclusion possible.

We refer to [BGS99] for the definition of the BGS version of ASMs, and [BS03], [Gu**] for background information and the current status of ASMs. (Chapter 9 of [BS03] provides both a history of ASM concept and a survey of ASM research).

2 States

A state in any ASM is a first order structure, and an ASM program is a recipe which defines what the next state will be, given what the present state is. The base set of every state in a given BGS ASM is the same, and it consists of two parts: a finite set of *atoms*, and the collection of *all* hereditarily finite sets built from the atoms. The input instance to the ASM defines what the set of atoms will be. In the BGS ASM terminology, atoms and the hereditarily finite sets of an ASM are called its *objects*.

A BGS ASM that terminates on an input instance will, of course, make use of only a finite subset of the infinite collection of hereditarily finite sets on its run on the input instance. A natural RAM implementation of the ASM will generate a representation of an element of this finite subset as and when its need arises during the simulation of the run.

2.1 Representation of objects

In our simulation, we will represent an ASM object as a partially labelled rooted directed acyclic graph that captures the transitive closure of the object.[1] For an object A, we use the term *TC dag of A* to denote this representation of A.

Let us assume that the set of atoms for the ASM to be simulated is $X = \{a_1, a_2, \ldots, a_n\}$, where all a_is are atoms.

Definition 1 (TC dag).
Given any atom, say a_i, its representative TC dag is simply a node labelled a_i. Similarly, the representative TC dag for an empty set is a single node with label 'ϕ'. The root to such a single node TC dag is clearly the node it consists of.

[1] We are indebted to Yuri Gurevich and Andreas Blass for suggesting to us this representation.

If X is a non-empty hereditarily finite set then $TC(X)$ denotes the transitive closure of X.[2] *The representative TC dag G_X for X has* **exactly one** *node corresponding to the each element of $TC(X)$, and there is an edge from u to v in G_X iff the set or the atom that corresponds to v is an element of the set that corresponds to u. All nodes in G_X that correspond to atoms or to the empty set ϕ are labelled– for an atom a_i the label is a_i, and for the empty set, the label is 'ϕ'. The root of G_X is the node that corresponds to X.* [3]

Access to a representative TC dag:
For an object X, if G_X is the TC dag representing X, then our RAM simulation will use a pointer to the root of G_X to access this representation of X.

As can be seen subsequently, in our implementation we may have the representative TC dag of one object sharing in part substructures of one or more representative TC dags. To avoid any confusion about which dag we mean in a context, we will use the following convention.

A convention: Suppose x is a vertex. Whenever we say *the dag subtended at x*, or simply, *the dag at x*, we mean the largest, downward closed dag that is there with x as its root vertex.

A crucial property of the representation is that it allows equality checking of ASM objects to be carried out in polynomial time, as a matter of fact, in linear time.

Proposition 1. *Two ASM objects X and Y are equal iff their TC dags are isomorphic. Further, the isomorphism testing problem of TC dag representations of a BGS ASM objects admits a RAM algorithm that runs in time linear in the size of the two TC dag representations being tested for isomorphism.*

The first statement of the above proposition is, more or less, obvious. We defer the proof of the second statement to the Appendix. [4]

---

[2] We recall that $TC(X)$ is the least set Y such that $X \in TC(X)$, and for all x, y, if $y \in x \in Y$ then $y \in Y$.

[3] Hereditarily finite sets are nested entities, and a natural representation for any nested entity is a tree. However, as pointed out by Blass and Gurevich [BG04], a RAM simulation of BGS ASMs that uses trees to represent hereditarily finite sets cannot carry out the simulation with only polynomial time overhead: it is easy to see that the von Neumann encoding of a natural number n can be constructed by a BGS ASM in time polynomial in n, but the tree representation of the encoding is exponential in n.

[4] That our isomorphism problem is linear time solvable is of interest because of the fact that the dag isomorphism testing, (even when the dags are rooted, with the added constraint that every path in a dag is of length at most two) can easily be shown to be as hard as the general graph isomorphism problem. Why is our problem simpler? The rooted dags that we use to represent BGS objects can be seen to be essentially compacted versions of rooted trees, and two of our rooted dags are isomorphic iff the two rooted trees they came from are isomorphic. Therefore, our isomorphism testing problem is really a variant of the tree isomorphism problem, which is known to be solvable in linear time.

2.2 Compacting rooted dags to TC dags

Clearly, not all dags with lebelled out-degree zero vertices will be TC dags, although we can naturally associate a hereditarily finite set (over the set of labels of zero out-degree vertices) with each non-labelled vertex of the graph, by treating each edge (u, v) to capture that the hereditarily finite set (or atom) associated with v is an element of the set associated with u. We shall see that in our simulation we do create such dags as above, and we will need to obtain the TC dag for the set associated with some vertex in the graph. The result below states that this can be done efficiently.

Definition 2. *Let G be a dag whose leaves, i.e., nodes with zero out-degree, are labelled with atoms or the empty set ϕ. We associate with each non-leaf node u of G a hereditarily finite set, $H(u)$, over atoms and ϕ. This is done by induction on the rank of a node in G, as follows.*

$H(u)$ = label of n, if u is a leaf.

$H(u) = \{H(u_1), \cdots, H(u_k)\}$, if u is a non-leaf node with children $u_1, \cdots, u_k$.

Proposition 2. *Given a labelled dag $G = (V, E)$ whose zero out-degree vertices are labelled with atoms or the empty set ϕ, and u a vertex of G, there is an efficient algorithm for creating a dag $G = (V, E')$, such that all $v \in V$ denote the same hereditarily finite set as in G, dag at u in G' is a* **TC dag** *and for all $v \in V$ if dag at v is a TC dag in G then dag at v is also a TC dag in G'. G' arises by manipulating the edges only in the dag at u in G and the algorithm works in time $O(|V_u|^4)$, where V_u is the set of vertices in dag at u in G.*

We defer the proof to the Appendix.

3 Basic Structure and Operations

In this section, the RAM implementation of each ASM component is given.

Logical attributes

- Logical values are directly implemented through the set definition of the empty set ϕ as false and $\{\phi\}$ as true.
- Boolean operations can be easily implemented directly in a RAM machine.
- The equality sign (a boolean predicate) which compares two objects is implemented through testing for isomorphism of the two corresponding TC dag representations. In other words, to evaluate $X = Y$, where X and Y are objects, we check if the two rooted TC dags G_X and G_Y, representing X and Y respectively, are isomorphic. If they are, then we conclude $X = Y$, otherwise, we conclude $X \neq Y$. From Proposition 1, we know that equality predicate can be evaluated in linear time.

Set theoretic functions

- The binary predicate $\in$ can be implemented through the rooted TC dag isomorphism that we use to test the equality predicate. For example, to evaluate $a \in S$, where both a and S are objects, we simply compare the representative TC dag of a with the TC dags at $x_1, x_2, \dots x_k$, where the root vertex of representation of S has edges to the k vertices $x_1, x_2, \dots, x_k$. Since the dag at each x_i represents an element of S, we conclude $a \in S$ if the representation of a is equal to one of these k rooted TC dags.
- Unary $\bigcup$ is defined as :
  If $a_1, \dots, a_j$ are atoms and $b_1, \dots, b_k$ are sets, then
  $\bigcup\{a_1, \dots, a_j, b_1, \dots, b_k\} = b_1 \cup \dots \cup b_k$.
  The TC dag representation clearly tells us whether an object is an atom or a set (an atom is simply a single node dag). Once one detects all the b_is, the representation of the union is created by first creating a new vertex v and then creating edges from v to the root vertex of the representation of each b_i. At this point $H(v) = b_1 \cup \dots \cup b_k$ (as per Definition 2) but the dag at v need not be a TC dag, as there may be nodes in different b_i's representing same hereditarily finite sets. We therefore invoke the algorithm of Proposition 2 to modify the dag at v (in the big graph which has represenentations for all the objects of asm at this point in computation) so that it becomes a TC dag.
  The resulting dag at the new vertex v is the RAM representation of $\bigcup\{a_1, \dots, a_j, b_1, \dots, b_k\}$

  It is easy to see that essentially we make polynomially many calls to TC dag isomorphism testing, and to the algorithm of Proposition 2. Therefore, $\bigcup$ can be implemented in polynomial time.
- TheUnique(a) is defined as the unique element of a if a is a singleton set, otherwise it is the null set. The number of children of the root of (the TC dag representing) a tells us the cardinality of a. Once this is determined, we can evaluate TheUnique(a).
- Pair(a,b) = $\{a, b\}$. Suppose the TC dag for a is rooted at v_a and that of b is rooted at v_b. We create a new vertex v, add edges from v to v_a and to v_b and invoke the algorithm of Proposition 2 to make the dag at v to be a TC dag. The dag at v represents Pair(a, b).

Dynamic Functions: Associated with each dynamic function f, we define the extent of f as the set:

$$\{(x_0, \dots, x_j) : f(x_0, \dots, x_{j-1}) = x_j \neq 0\}$$

The extent for any dynamic function is always finite. This is stored as a linear list of *blocks*. Each block is simply an ordered list of pointers to $x_0, \dots, x_j$ (in that order). We have a block for each element in the extent. Given some values $x_0, \dots, x_{j-1}$, we can find the value of $f(x_0, \dots, x_{j-1})$ by scanning the list of blocks.

Term Formation: Suppose v is a variable, $t(v)$ is a term, r is a term without free occurrences of v and $g(v)$ is a Boolean term, then $\{t(v) : v \in r : g(v)\}$ is a term. To implement this, we first create a new node, say w. We consider each child vertex u of the root of the TC dag for r, check if the (object corresponding to the) dag at u satisfies the predicate g. If it does, we create the TC dag for $t(v)$ where v is the object represented by the dag at u. We add an edge from the new node w to the root of the dag for this $t(v)$. Clearly, $H(w) = \{t(v) : v \in r : g(v)\}$, we now invoke the algorithm of Proposition 2 to get a TC dag at w. The dag at w represents $\{t(v) : v \in r : g(v)\}$.

4 Rules

We will instantiate any variable as and when it comes, so during our implementation we will never encounter free variables (because a program is a rule without any free variables).

Skip : This is a null instruction, so nothing is to be done.

Update Rules : Update rules are of the form $f(t_1, \ldots, t_r) := t_0$, where f is a dynamic function of arity r and $t_0, \cdots, t_r$ are terms. Since in our implementation, all variables are instantiated, the terms will always be objects. To carry out this rule, we search for a block in the sequence of blocks that represents f, such that the sequence formed by the first $r-1$ objects of the block is equal[5] to $t_1, \ldots, t_r$. If we find one, the last object of the block is updated to t_0. If such a block does not exist, we create a new block with $t_1, \ldots, t_r, t_0$.

Conditional Rules : If g is a boolean term and R_1 and R_2 are rules, then a conditional rule is:
`if` g `then` R_1 `else` R_2 `endif`

Since we always have instantiated variables, g will not have any free variables and can be evaluated. Accordingly, our RAM program can simulate R_1 or R_2 accordingly.

Do-forall Rules : If v is a variable, r is a term without v free and $R_0(v)$ is a rule, then a Do-forall rule is -

`do forall` $v \in r$
    $R_0(v)$
`enddo`

Here, we instantiate v with all elements of r and execute $R_0(v)$. These executions have to be done in parallel, however, we are using a serial RAM. This

[5] Here, the equality is the equality of two tuples of objects, the implementation of object tuples equality extends obviously from equality of objects.

can be done by using temporary storage. This instantiation ensures that we do not have any free variables at any stage of the implementation. Therefore, our initial assumption holds.

5 Complexity Analysis

Let us consider an ASM program running in polynomial time. Such a program is denoted as $(\Pi, p(n), q(n))$ which captures the fact that the program Π has a run of length $\leq p(n)$ and the total number of active objects is $\leq q(n)$, where n is the size of the input, and $p(\cdot)$ and $q(\cdot)$ are two fixed polynomials.[6] Let us now consider the RAM machine implementation of Π and calculate its time and space complexity.

New objects are fundamentally created either through the $\bigcup$ operator, through Pair or through term formation. That implies that new active objects are created through already present active objects. As the TC dag for any object X will have at most one sub-TC dag representing any element of $TC(X)$, we conclude that the TC dag for any object will have at most $O(q(n))$ nodes, which implies that each object can be stored in $O(q^2(n))$ space. Space is used for storing the extent of each dynamic function, and as temporary workspace, e.g., in the simulation of a Do-forall rule. The number of entries of each dynamic function is bounded by $q(n)$, and the size of each entry is $O((k+1) \times \log q(n))$, where k is the arity of the function. (Recall that each block for the function is a list of pointers). Since the number of functions is a constant and the arity of each is also a constant, the space-complexity is $O((\log q(n))q(n)^2)$. The temporary space required for the Do-forall rules would also be of order $O(q(n)^2)$.

Let us try to see the amount of time that each step in the run takes. The number of objects accessed in any step is bounded by $q(n)$. Each $\bigcup$ operation and update can take at most $O(q^4(n))$ time, taking into account the isomorphism testings involved. Same bound holds for Pair also. Term formation and Do-forall

[6] Roughly, an object is active if either it is an atom, or a constant (e.g., a truth value representation), or an object which is created (even for a temporary reason) at any time during the execution of the ASM program on the given input. The number of active objects is a measure (as explained in [BGS99]) of the number of microsteps ([BG97]) in the execution of a program on an input. In our implementation, an object is represented as a rooted TC dag. We may have two TC dags occurring in the storage which are isomorphic. That is, several copies of the same object may be present in the storage. Thus, the number of active objects may not, in general, correspond to the total number of rooted dags our implementation deals with. The correspondence is achieved, if for the purpose of this complexity analysis, we assume that every time an object gets created, even temporarily, it is a new object. In any case, as the total number of active objects will be polynomially related to the total number of all copies of all objects for a PTime BGS ASM, the distinction is not crucial for our result that such a program can be implemented by a PTime RAM.

operations can only have $O(q(n))$ loops. Therefore, the time complexity for one step can be expressed as $O(q(n)^k)$, where k is some constant (this comes due to the possible presence of nested Do-forall operations and updates, term formations, etc. present in these loops). The total time-complexity then comes out to be $O(p(n)(q(n))^k)$, for some constant k, which is bounded by a fixed polynomial in n.

We conclude, therefore, that our implementation takes a BGS model of a PTime bounded ASM program and converts it to a RAM, which runs in polynomial time and uses polynomial space.

6 Concluding remarks

The randomized algorithm for testing tree isomorphism as given in [Ko92] provides an obvious fingerprinting scheme for trees. We can extend this scheme easily to work for our rooted dags used for representing objects. In practice, with every object representation, we can keep also its fingerprint. The implementation of the equality predicate testing will become more efficient when the fingerprints are taken into account. Our implementation, in general, will have several copies of the same non-atomic object. This is what happens also in usual implementations of Lisp, Java, etc. However, it is possible to have an implementation that ensures only a single copy of any object will be present in the storage at any time. The challenge then will be how best to reduce the increase in time complexity which the decrease in space usage will entail.

As explained in [BG00], BGS model is an instance of ASMs with background classes of structures. It is easy to see that for the other examples of ASMs with finitary background classes as given in [BG00], the same result as ours holds: each of such ASMs can be simulated by Turing machines with at most a polynomial time overhead. It is not clear to us, however, if the axioms as provided in [BG00] to formalize the notion of background classes can even guarantee that testing for equality of two objects can be done effectively. Besides equality, one would like to have term formation operations, as well as those operations that break a compound term into its constituents, [7] to be effective for simulation with Turing machines. For simulation with at most polynomial time overheads, it is necessary that all these should be feasibly implementable on Turing machines. Is the condition also sufficient? This appears to us to be a question worth investigation.

References

[AHU74] *The Design and Analysis of Computer Algorithms*, Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, Addison-Wesley, 1974.

[BG97] The Linear Time Hierarchy Theorems for RAMs and Abstract State Machines, Andreas Blass and Yuri Gurevich, Springer Jl. of Universal Computer Science, Vol 3, No. 4, pp 247–278, 1997.

[7] For example, `cons` and `car, cdr` as in Lisp are such operations for lists.

[BG00] Background, Reserve, and Gandy Machines, Proc. CSL 2000, LNCS Vol. 1862, pp 1 – 17, 2000.
[BG04] Personal communication, July 2004.
[BGS02] On Polynomial Time Computation over Unordered Structures, Andreas Blass, Yuri Gurevich, and Sahron Shelah, Journal of Symbolic Logic, 67:3, pp 1093 – 1125, 2002.
[BGS99] Choiceless Polynomial Time, Andreas Blass, Yuri Gurevich, and Sahron Shelah, Annals of Pure and Applied Logic, 100, pp 141 – 187, 1999.
[BS03] *Abstract State Machines*, Egon Börger and Robert Stärk, Springer-Verlag, 2003.
[Gu**] See home page of Yuri Gurevich `http://www.research.microsoft.com/g̃urevich` for extensive information on ASMs.
[Ko92] *The Design and Analysis of Algorithms*, Dexter C. Kozen, Springer-Verlag, 1992.

Appendix

We provide here the proofs of Propositions 1 and 2.

We need a notion of *rank* for nodes in a dag. Let G be a dag. All nodes in G that have zero out-degree have rank 0. The rank of any other node y is $m+1$ if m is the maximum rank among the vertices to each of which there is an edge from y.

Proposition 1. *The isomorphism testing problem of rooted TC dag representations of objects of a BGS ASM admits a RAM algorithm that runs in time linear in the size of the two representations being tested for isomorphism.*

Proof. We sketch an algorithm which is an adaptation of the tree isomorphism testing algorithm of [AHU74].

Let X and Y be two objects and G_X and G_Y be their corresponding rooted dag representations. The algorithm essentially tries to construct an isomorphic mapping between G_X and G_Y by extending a map that maps vertices of rank i in G_X to vertices of rank i in G_Y, for successive values of i starting with i equal to 0.

We recall that the rank 0 vertices in both the dags are labelled. The algorithm constructs a list consisting of labels of rank 0 vertices of G_X, and a list of labels of rank 0 vertices of G_Y, and then sorts the two lists. If the two sorted lists are not identical, the algorithm terminates declaring that the two dags are not isomorphic. If the sorted lists are identical, then each rank 0 vertex is given a (new) label which is the integer j, where j is the position of the old label in the sorted list.

Suppose the algorithm has completed assigning integer labels to each node at rank i in both G_X and G_Y. Next, we consider each vertex v at rank $(i+1)$. Suppose there is an edge from v to each $v_1, v_2, \ldots, v_k$, let the ranks of these k vertices be $r_1, r_2, \ldots, r_k$ respectively, and the integer labels of the k vertices be $n_1, n_2, \ldots, n_k$ respectively. The label that we assign to v is the *sorted version* of the list consisting of $(r_1, n_1), (r_2, n_2), \ldots (r_k, n_k)$. Then we create two lists, one each for the vertices of rank $(i+1)$ of G_X and G_Y, and then sort the two lists. If these two sorted lists are not identical then the algorithm terminates declaring G_X and G_Y to be non-isomorphic. Else, each rank $(i+1)$ vertex is relabelled with an integer: vertex u gets the new integer label m, if its old label was at the mth position in the sorted list of old labels of rank $(i+1)$ vertices of the corresponding dag.

Iterating in the above manner, when the two root vertices get their integer labels, (provided the algorithm has not already terminated by then), the algorithm returns with the decision that the two dags are isomorphic if the two root labels are equal, else returns a negative answer.

For sorting, we use the linear time algorithm in [AHU74] to sort lists with variable sized tuples, where the tuple components are integers in the range 0 to a value polynomial in the size of our instance size. We use sorting in two types of steps: first, for sorting initial labels of all vertices of a certain rank, and second, in obtaining the initial label of a vertex. The first type of sorting, amortized over all vertices, will take total time proportional to the total number of vertices; and the second type of sorting, amortized over all edges, can be seen to take time proportional to the total number of edges. Hence, the algorithm runs in time linear to the size of the input instance.

Proposition 2. *Given a labelled dag $G = (V, E)$ whose zero out-degree vertices are labelled with atoms or the empty set ϕ, and u a vertex of G, there is an efficient algorithm for creating a dag $G = (V, E')$, such that all $v \in V$ denote the same hereditarily finite set as in G, dag at u in G' is a* **TC dag** *and for all $v \in V$ if dag at v is a TC dag in G then dag at v is also a TC dag in G'. G' arises by manipulating the edges only in the dag at u in G and the algorithm works in time $O(|V_u|^4)$, where V_u is the set of vertices in dag at u in G.*

Proof. Let G_u be the dag at u in G and let m be the rank of the node u in G_u. For $i = 0$ to m, we repeat the following steps (in increasing order of i).

{

Let V_i be the set of nodes of rank i in G_u. Choose a subset V_i' of V_i, s.t. for all $l \in V_i$, there is a unique $l' \in V_i'$, s. t. $H(l) = H(l')$. This can easily be done, as for $i = 0$ we have assumed that equality of atoms can be tested easily and for $i > 0$ we use the fact that for all $v, v' \in V_i$, $H(v) = H(v')$ iff set of their children are the same (note that this step does not involve equality checking recursively, by virtue of inductive step in our construction). Now for each $l \in V_i - V_i'$ and for any $v \in V(G_u)$, if there was an edge from v to l then it is replaced by an edge from v to l', where l' is the unique representative of l in set V_i'. A duplicate edge between any pair of nodes is deleted.

}

It is clear from the construction that any node represents the same hereditarily finite set in G' as in G and any node which has a TC dag subtending at it in G also has a TC dag subtending at it in G'.

The complexity of the above algorithm is easily seen as upper bounded by $O(|V(G_u)|^4)$.