Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 376 (2007) 17–29
Theoretical Computer Science
www.elsevier.com/locate/tcs

Can abstract state machines be useful in language theory?

Yuri Gurevich[a,*], Margus Veanes[a], Charles Wallace[b]

[a] *Microsoft Research, Redmond, WA, USA*
[b] *Michigan Tech, Houghton, MI, USA*

Abstract

The abstract state machine (ASM) is a modern computation model. ASMs and ASM based tools are used in academia and industry, albeit on a modest scale. They allow you to give high-level operational semantics to computer artifacts and to write executable specifications of software and hardware at the desired abstraction level. In connection with the 2006 conference on Developments in Language Theory, we point out several ways that we believe abstract state machines can be useful to the DLT community.

Keywords: Abstract state machine; AsmL; Evolutionary algorithms; Graph-partition algorithm; Language theory

1. Introduction

Abstract state machines (originally called evolving algebras) constitute a modern computation model [14,16]. ASMs describe algorithms without compromising on the abstraction level. There are numerous ASM based tools, in particular the specification language AsmL, the ASM Language, designed and implemented at Microsoft Research [2]. ASMs and ASM based tools are used in academia and industry, albeit still on a modest scale. They are used to give precise semantics to computer programs, to programming languages and to other computing artifacts, and they are used to specify software and hardware [1,2,9,17].

This paper started as a write-up of the invited talk that the first author gave at the 10th International Conference on Developments in Language Theory, DLT 2006. Language theory is understood broadly by the DLT community [12], and we use the term in the same broad sense. We argue that ASMs can be useful to the DLT community and we point out a few specific ways that ASMs can be useful.

In Section 2 we describe some properties of ASMs that appear to us most relevant to language theory: their strong universality, their facility for abstraction and interaction, and their ability to capture concurrency.

In Section 3 we illustrate how ASMs can be used to program traditional language-theory algorithms on their natural abstraction levels, devoid of unnecessary details. Typically these algorithms are described in words or are programmed in a conventional programming language, which does involve unnecessary implementation details. We consider two

* Corresponding author. Tel.: +1 425 703 5664.
E-mail addresses: gurevich@microsoft.com (Y. Gurevich), margus@microsoft.com (M. Veanes), wallace@mtu.edu (C. Wallace).

0304-3975/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2007.01.009

well known algorithms and program them in AsmL. The level of abstraction is that of traditional verbal description, but of course we have to be precise.

The DLT community has been keenly interested in evolutionary computing. In Section 4, we illustrate how ASMs can be used to write evolutionary algorithms, and what benefits one may expect from using ASMs. We do not presume that the reader is familiar with evolutionary algorithms.

In Section 5 we illustrate how ASMs compute with inputs in the form of finite abstract structures. The ASM model was designed to work with abstract structures. In fact, the current models for computations with abstract structures and the related complexity theory build upon ASMs [7,8]. Computing with abstract structures is not prominent yet on the DLT horizon, but it is not completely foreign to the DLT community either. The list of topics at the DLT 2006 website [12] starts with "grammars, acceptors and transducers for strings, trees, graphs, arrays". But the conventional computation models cannot deal directly with abstract structures: you cannot put an abstract graph on the tape of a Turing machine. Instead, the conventional models deal with presentations of abstract structures. In the case of trees, a linear order is imposed on the children of the same node. In the case of graphs, a linear order is imposed on the vertices. We are convinced that the significance of computing with abstract structures is bound to grow. As interest in grammars, acceptors and transducers for abstract structures grows, ASMs will become more and more relevant to the DLT community.

Finally, let us say a few words about the AsmL programs of this paper. To enhance their readability, we allow ourselves to use mathematical symbols instead of some AsmL symbols: '$\in$' for '`in`', '$\notin$' for '`notin`', '$\neq$' for '`ne`', '$\mapsto$' for '`->`', '$\forall$' for '`forall`', '$\exists$' for '`exists`', '$\leftrightarrow$' for '`iff`', '$\emptyset$' for '{}', '$\cap$' for '`intersect`'. When a method or function M takes no parameters, we abbreviate $M()$ to M.

The public distribution of the model-based testing and model exploration tool Spec Explorer, developed at Microsoft Research, includes a number of AsmL samples [21]. We have put there the AsmL programs of Sections 3 and 5, complete with example data and using the standard AsmL symbols, so that you can execute the programs and play with them. We have not done the same with the programs of Section 4 because those are not complete in themselves; they require a number of external application-dependent functions.

2. The ASM computation model: How is it different?

We do not define ASMs here; instead, we discuss some of the distinctive qualities of ASMs that may be of interest to the DLT community. The examples in the subsequent sections should be largely self-explanatory.

For those interested in learning more about ASMs, the standard reference [14] to the definition of ASMs is still a good place to start. But there has been a substantial advance in the meantime. Two developments are worth mentioning here. One is the development of the specification language AsmL [2], a richer and executable alternative to the frugal mathematical model of [14]. The other is the enhancement of the interactive aspect of ASMs [16].

2.1. A richer notion of universality

Turing's original model of computation was intended to capture the notion of computable function from strings to strings. Turing convincingly argued that every string-to-string computable function is computable by some Turing machine [22]. His thesis is now widely accepted.

The string-to-string restriction does not seem severe because usually inputs and outputs can be satisfactorily coded by strings. But the restriction is not innocuous. Some algorithms work with inputs that do not admit string encoding. Think for example of the Gaussian elimination algorithm that deals with genuine reals. One can argue that (a) in any actual computation one deals with finite objects, e.g. with finite approximations of genuine real numbers, and (b) finite objects can be represented by strings. Clause (a) is true of course, but it may be useful to write down an algorithm on its natural level of abstraction. Clause (b) is true as well, but only to a point. A graph, for example, can be represented by an adjacency matrix (and a matrix can be written as a string: row by row) but this requires that the vertices be linearly ordered. Different orders give different adjacency matrices in general. If a graph input is given by an adjacency matrix, then the computation may depend on the choice of the adjacency matrix. This is related to a well-known problem of database theory: how to deal with databases in an implementation independent way? Besides, the problem of whether two given adjacency matrices represent the same graph is not known to have a polynomial time solution.

There is another restriction of Turing's model. A Turing machine simulation of a given algorithm is guaranteed to preserve only the input/output behavior. But there may be more to an algorithm than the function it computes.

The abstract state machine (ASM) computation model is universal in a stronger sense. The ASM thesis asserts that, for every algorithm A, there is an ASM B that is behaviorally equivalent to A [14]. In particular, B step-for-step simulates A. This applies also to the Gaussian elimination algorithm and other algorithms that deal with infinite objects. Substantial parts of the thesis have been proved from first principles [15,6,16]; these developments are put into broad perspective in [5].

2.2. Abstraction

While the Turing machine is perfect for its intended purpose, its abstraction level is essentially that of single bits. The low abstraction level of the Turing machine inhibits its ability to faithfully simulate algorithms. On the other hand, an ASM simulator operates at the abstraction level of the original algorithm. Each ASM is endowed with a vocabulary of function names. A state of the ASM is a (first-order) structure of that vocabulary: a collection of elements, along with interpretations of the function names as functions over the elements. The author of an ASM program has flexibility in choosing the level of abstraction. For example, atoms, sets of atoms, sets of sets of atoms, etc. can be treated as elements of a structure of the vocabulary with the containment relation $\in$. Similarly, other complex data – maps, sequences, trees, graphs, sets of maps, etc. – can be dealt with directly. This makes ASMs appropriate for various applications – e.g., specifications of software – that deal with high-level abstractions.

2.3. Non-determinism and interaction

An ASM can be non-deterministic. For example, you may have an assignment of the form

```
current := any $x \mid x \in s$
```

where s is a set-valued term. (If $s = \emptyset$ then an error occurs.)

From the point of view of ASM theory, algorithms are inherently deterministic. Nondeterministic choices are made by the algorithm's environment, e.g. the operating system. The any construct is really a query asking the environment to make a choice. Similarly, the creation of a new object (in the sense of object oriented programming) is a query. See more about that in [16].

2.4. Concurrency

In the ASM world, parallelism is the default. For example consider a statement (a.k.a. rule)

```
if cond then
   $R_1$
   $R_2$
```

where cond is a Boolean-valued expression and R_1, R_2 are statements. If cond evaluates to true, then the statements R_1 and R_2 are executed in parallel. You pay a syntactic price for requiring that the rule R_1 is executed first, and the rule R_2 is executed second:

```
if cond then
   step
      $R_1$
   step
      $R_2$
```

Parallelism can be massive. For example, you may have a comprehension expression

```
current := {$f(x) \mid x \in s$ where cond($x$)}
```

or a statement

```
forall $x \in s$
   $R(x)$
```

where s is a set-valued term, cond(x) is Boolean-valued term and $R(x)$ is a statement. (If $s = \emptyset$ then the `forall` command does nothing and thus is equivalent to the `skip` command.)

A distributed ASM is a dynamic set of agents operating asynchronously over a global state [14]. The global state could reflect a physical common memory or be a pure mathematical abstraction with no physical counterpart. Agents can be created or removed.

There are various new computational paradigms that exploit the possibility of massive parallelism: quantum computing, DNA computing, etc. They all seem well suited for description in terms of ASMs. In fact, Grädel and Nowack proved that every model of quantum computing in the literature can be viewed as a specialized ASM model [13].

2.5. *Executability*

The last, but certainly not the least, important aspect of the ASM computation model that we address here is the executability of ASMs. There are several ASM-based high-level programming languages. The most powerful is AsmL [2]. No, AsmL cannot execute the Gaussian elimination algorithm over genuine reals; the reals would have to be approximated.

3. Traditional language theory algorithms: Examples

3.1. *Finite automata minimization*

We construct an ASM that executes the well-known algorithm for minimizing a deterministic finite automaton [18]. Given a finite automaton $A = \langle Q, \Sigma, q_0, \delta, F\rangle$, the algorithm computes an equivalence relation E on Q and then uses the equivalence classes of E as the states of a minimal finite automaton $A' = \langle Q', \Sigma, q'_0, \delta', F'\rangle$ equivalent to A.

The desired ASM is a sequential composition of three submachines:

```
MinimizeFA
   step
      InitializeEquivalence
   step
      ComputeEquivalence
   step
      ComputeNewAutomaton
```

Now we describe the three submachines. The first one is the simplest.

```
InitializeEquivalence
   $E := \{(p,q) \mid p \in Q, q \in Q$ where $p \in F \leftrightarrow q \in F\}$
```

The second submachine iterates an assignment to E until a fixed point is reached. Reaching a fixed point means in this case that the value of E stabilizes and does not change any more.

```
ComputeEquivalence
step until fixpoint
   $E := \{(p,q) \mid (p,q) \in E$ where $\forall\sigma \in \Sigma$ holds $(\delta(p,\sigma), \delta(q,\sigma)) \in E\}$
```

The final submachine uses a function `Eclass` that maps an original finite automaton state q into the E-equivalence class that contains q. The type `FAstate` is abstract. We do not presume that the states of the given finite automaton are numbers or strings.

```
type FAstate
Eclass($q$ as FAstate) as Set of FAstate
   return $\{p \mid p \in Q$ where $(p,q) \in E\}$

ComputeNewAutomaton
   $Q' := \{$Eclass$(q) \mid q \in Q\}$
   $q'_0 :=$ Eclass$(q_0)$
   $\delta' := \{($Eclass$(q), \sigma) \mapsto$ Eclass$(\delta(q,\sigma)) \mid q \in Q,\ \ \sigma \in \Sigma\}$
   $F' := \{$Eclass$(q) \mid q \in F\}$
```

The four assignments are executed in parallel, because parallel execution is the default in AsmL.

3.2. Finite automata determinization

We construct an ASM that executes the Rabin–Scott algorithm that, given a non-deterministic finite automaton $A = \langle Q, \Sigma, q_0, \delta, F\rangle$, where $\delta : Q \times \Sigma \to 2^Q$, computes an equivalent deterministic finite automaton $A' = \langle Q', \Sigma, q'_0, \delta', F'\rangle$. Again, new states are sets of original states. The desired ASM is the sequential composition of two submachines

```
DeterminizeFA
   step
      InitializeNewAutomaton
   step until Frontier = $\emptyset$
      ExploreFrontier
```

where the second submachine is an iteration of a simpler submachine. We describe the submachines `InitializeNewAutomaton` and `ExploreFrontier`. To this end, we need an auxiliary variable `Frontier`. As before, the type `FAstate` is abstract.

```
type FAstate
var Frontier as Set of Set of FAstate

InitializeNewAutomaton
   $Q' := \{\{q_0\}\}$
   Frontier $:= \{\{q_0\}\}$
   $q'_0 := \{q_0\}$
   $F' :=$ if $q_0 \in F$ then $\{\{q_0\}\}$ else $\emptyset$
   $\delta' := \{\mapsto\}$
```

Here $\{\mapsto\}$ is the empty map. In the typed world of AsmL, we have to distinguish between the empty set and the empty map.

```
ExploreFrontier
   forall $S \in$ Frontier
      forall $\sigma \in \Sigma$
         let $T = \{q \mid q \in Q$ where $\exists p \in S$ where $q \in \delta(p, \sigma)\}$
         $\delta'(S, \sigma) := T$
         if $T \notin Q'$ then
            add $T$ to $Q'$
            add $T$ to Frontier
         if $T \cap F \neq \emptyset$ then
            add $T$ to $F'$
      remove $S$ from Frontier
```

4. Evolutionary algorithms

We illustrate AsmL specifications (a.k.a. high-level descriptions, high-level programming) of evolutionary algorithms with a couple of examples. The idea is to promote AsmL as a better alternative to the pseudocode specification of algorithms. It is common to specify algorithms in pseudocode form, and the case of evolutionary algorithms is no exception. Typical drawbacks of pseudocode include unintended ambiguities, inconsistent abstraction levels, unnecessary details, missing essential information, insufficient clarity, and insufficient precision. Of course pseudocode specifications carefully written by best field experts may be of high quality, and the use of AsmL does not guarantee perfect specifications. But AsmL helps one to keep a consistent abstraction level, and it imposes certain degrees of precision without forcing unnecessary details or determinism. And there is a crucial advantage of AsmL specifications: they are executable. You can play with them, and in the process debug your specification. Note, however, that in examples like those below, when the environment of the algorithm evaluates so-called external functions, the executability of specifications requires programming those evaluations in one way or another.

4.1. The canonical evolutionary algorithm

There are various computing methods for solving optimization problems, inspired by the metaphor of evolution in the natural world [4]. What is the essence of an evolutionary algorithm? Many authors attempt to answer this question by giving a “general scheme” or “canonical algorithm”, in pseudocode form. In an influential book [4] on evolutionary computing, Bäck [3] gives a “general framework” for the “basic instances of evolutionary algorithms”. We quote:

> We define I to denote an arbitrary space of individuals $a \in I$, and $F : I \to \mathbb{R}$ to denote a real-valued fitness function of individuals. Using μ and λ to denote parent and offspring population sizes, $P(t) = (a_1(t), \ldots, a_\mu(t)) \in I^\mu$ characterizes a population at generation t. Selection, mutation, and recombination are described as operators $s : I^\lambda \to I^\mu$, $m : I^\kappa \to I^\lambda$, and $r : I^\mu \to I^\kappa$ that transform complete populations.
>
> These operators typically depend on additional sets of parameters Θ_s, Θ_m, and Θ_r which are characteristic for the operator and the representation of individuals. Additionally, an initialization procedure generates a population of individuals …, an evaluation routine determines the fitness values of the individuals of a population, and a termination criterion is applied to determine whether or not the algorithm should stop.
>
> Putting this all together, a basic evolutionary algorithm reduces to the simple recombination–mutation–selection loop as outlined below:

```
Input:    $\mu, \lambda, \Theta_\iota, \Theta_r, \Theta_m, \Theta_s$
Output:   $a^*$, the best individual found during the run, or
          $P^*$, the best population found during the run.
1  $t \leftarrow 0$;
2  $P(t) \leftarrow$ initialize($\mu$);
3  $F(t) \leftarrow$ evaluate($P(t), \mu$);
4  while ($\iota(P(t), \Theta_\iota) \neq$ true) do
5      $P'(t) \leftarrow$ recombine($P(t), \Theta_r$);
6      $P''(t) \leftarrow$ mutate($P'(t), \Theta_m$);
7      $F(t) \leftarrow$ evaluate($P''(t), \lambda$);
8      $P(t+1) \leftarrow$ select($P''(t), F(t), \mu, \Theta_s$);
9      $t \leftarrow t+1$;
   od
```

That completes the quotation. As with many pseudocode representations of algorithms (and we see plenty of these in software engineering), certain questions arise. Writing an ASM forces us to be precise and confront the questions.

Consider line 3. The line is superfluous unless the termination condition utilizes $F(t)$, in which case we would expect F to appear as a parameter of ι. Of course F may appear in Θ_ι, but the application specific Θ_ι is inappropriate to hide F. We will make $F(t)$ an explicit parameter of ι.

Once our confidence in the pseudocode is slightly shaken, we may question whether the parts of it that are clear have their intended meaning. $P(t)$ is defined as a sequence, but no advantage is gained from ordering the individuals (or from the fact that the same individual can appear several times in $P(t)$). Is $P(t)$ ordered here because it is ordered in the implementations? We will treat $P(t)$ as a set.

Consider lines 3 and 7. As far as we can see, the second parameter of `evaluate` is superfluous, as its value is just the cardinality of the first parameter value. We will remove the second parameter of `evaluate`. Another question concerns the order of lines 5 and 6: is it necessary that mutation follows recombination? At this point, we will follow the pseudocode.

Finally, something that appears to be missing from the pseudocode is the computation of the output. We will incorporate a computation of the best population into our ASM (and will not compute the best individual). But first we have to make the notion of best population precise. One way to do that is this. We presume that possible results of `evaluate` are linearly ordered. The best population is the population $P(t^*)$ such that

$$\texttt{evaluate}(P(t^*)) = \max_t \texttt{evaluate}(P(t))$$

and t^* is the least possible. If the algorithm terminates, then the sequence $\langle P(0), P(1), \ldots\rangle$ is finite and the best population is defined.

The functions `evaluate`, `initialize`, `mutate`, `recombine`, `select`, and `terminate` that appear in pseudocode are external functions in ASM terms. They are evaluated by the environment (e.g. the implementation) of our high-level algorithm. We do not program these external functions here. However, for clarity, it is useful to declare their types. Besides, AsmL is a typed language and requires us to declare the types of functions. However, the types could be abstract; when needed, the abstract types could be refined. We use that flexibility of AsmL. The type `Params` of the application specific parameters is abstract. In the pseudocode above, parameters Θ_ι, Θ_r, Θ_m, Θ_s may have different types. Accordingly we may need different refinements of `Params`. The type `Real` can be concretized to be a numeric type that approximates reals (e.g. `Float`). Individuals are abstract entities represented by the abstract type `Individual`.

```
type Params
type Real
type Individual
type Population = Set of Individual
evaluate($p$ as Population) as Real
initialize($m$ as Integer) as Population
mutate($p$ as Population, $t$ as Params) as Population
recombine($p$ as Population, $t$ as Params) as Population
select($p$ as Population, $f$ as Real,
       $s$ as Integer, $t$ as Params) as Population
terminate($p$ as Population, $f$ as Real, $t$ as Params) as Boolean
```

The main ASM program *CanonicEvolution* contains three submachines that are defined below.

```
CanonicEvolution
   step
      InitializeComputation
   step while not terminate($P$, evaluate($P$), $\Theta_\iota$)
      step
         CreateNextGeneration
      step
         UpdateBestPopulation
```

Variables P and P^* are defined as follows.

```
var $P$ as Population
var $P^*$ as Population
```

Note that P does not have the argument t. The algorithm computes the best population only, and for this purpose the explicit time parameter will not be needed. The situation would be different if the goal was to compute the whole generation history.

Now we give the three submachines. The initializing submachine returns a set of individuals of size μ.

```
InitializeComputation
   step
      $P :=$ initialize($\mu$)
   step
      $P^* := P$
```

```
CreateNextGeneration
   let $P'$ = recombine($P$, $\Theta_r$)
   let $P''$ = mutate($P'$, $\Theta_m$)
   let $F$ = evaluate($P''$)
   $P$ := select($P''$, $F$, $\mu$, $\Theta_s$)
```

The four external functions have the intuitive meaning explained above by Bäck.

```
UpdateBestPopulation
   if evaluate($P$) $>$ evaluate($P^*$) then $P^* := P$
```

If and when the algorithm terminates, the variable P^* gives the best population.

4.2. Parallel evolutionary algorithms

How much can the canonical algorithm be parallelized? Nowostawski and Poli [19] give a taxonomy of parallel versions of the canonical evolutionary algorithm. Here we focus on "static demes with migration" scenarios, where the population is partitioned into a set of disjoint *demes* (subpopulations), each one maintained separately. Each individual is in one deme at any given time, but individuals may migrate from deme to deme. We consider two particular scenarios.

4.2.1. Synchronous scenario

In a synchronous scenario, generations of the total population are well defined, and the optimal solution is chosen over all generations, as in the canonical algorithm. Migration happens between generations. The desired ASM is a modification of the *CanonicEvolution* ASM above. The top-level structure of the new ASM is the following.

```
SynchronousParallelEvolution
   step
      InitializeComputation
   step while not terminate($P$, evaluate($P$), $\Theta_t$)
      step
         CreateDemeCandidates
      step
         MoveIndividualsAround
      step
         UpdateBestPopulation
```

It is similar to that of the top-level structure of *CanonicEvolution*, but there are two differences: the `CreateNewGeneration` submachine is split into two submachines, and P is now a derived function rather than a variable. P is, of course, the union of the populations of all the demes. But what is a deme exactly? Note that a deme may be empty during some points of the evolution. Are two empty demes equal? Not necessarily. If follows that demes are not uniquely defined by their contents.

```
class Deme
   var $p$ as Set of Individual
Demes as Set of Deme
```

Intuitively, the members of Deme are deme identifiers of some kind. The field p of a deme d gives the current population of d. More formally, p is a function that maps every deme to a set of individuals.

```
$P$ as Set of Individual
   return $\{i \mid d \in \text{Demes}, i \in d.p\}$
```

Now we describe the submachines of *SynchronousParallelEvolution*.

```
InitializeComputation
   step
      forall $d \in$ Demes
         $d.p$ := initialize($d$, $\mu(d)$)
   step
      $P^* := \{i \mid d \in \text{Demes}, i \in d.p\}$
```

Note that μ is now not an integer, as it was above, but a function from demes to integers.

Creating a new generation now involves parallelism and migration.

```
CreateDemeCandidates
   forall $d \in$ Demes
      let $p'$ = recombine($d.p$, $\Theta_r$)
      let $p''$ = mutate($p'$, $\Theta_m$)
      let $f$ = evaluate($p''$)
      $d.p$ := select($p''$, $f$, $\mu(d)$, $\Theta_s$)
```

The following migration rule presumes that the demes are disjoint; otherwise inconsistency may arise. For an individual i, a dynamic external function NewDeme(i) gives the new deme of i.

```
MoveIndividualsAround
   forall $d \in$ Demes
      forall $i \in d.p$ where NewDeme($i$) $\neq d$
         remove $i$ from $d.p$
         add $i$ to NewDeme($i$).$p$
```

NewDeme(i) is dynamic in the sense that a subsequent call to NewDeme(i) may produce a different result. One may want to make NewDeme(i) static by providing additional parameters. We are accustomed to dynamic external functions and feel comfortable using them. The final submachine is almost identical to the one used by *CanonicEvolution*, but P is derived now.

```
UpdateBestPopulation
   if evaluate($P$) > evaluate($P^*$) then $P^* := P$
```

4.2.2. Asynchronous case

In an asynchronous scenario, demes evolve independently, and generations of the total population may be ill defined. The algorithm runs until termination, and the union of the final demes constitutes the final population.

Demes do not communicate directly with each other. To enable migration, a special station is used.

```
var Station as Set of (Individual, Deme) $= \emptyset$
```

It is composed of pairs (i, d), where i is an individual and d is a deme to which i is being transferred. The station is initially empty.

We program the behavior of a deme including the interaction between the deme and the station. We omit the part on choosing the best individual among the final population.

Demes are initialized upon creation. When a new deme d comes to existence, its population p is initialized to initialize(d, $\mu(d)$).

```
class Deme
   Deme
      $p$ := initialize(this,$\mu$(this))
```

A deme evolves by executing its program Evolve.

```
class Deme
   Evolve
      step while not terminate($p$, evaluate($p$), $\Theta_t$)
         step
            CreateDemeCandidate
         step
            TransferIndividuals

   CreateDemeCandidate
      let $p'$ = recombine($p$)
      let $p''$ = mutate($p'$)
      let $f$ = evaluate($p''$)
      $p$ := select($p''$, $f$, $\mu$, $\Theta_s$)

   TransferIndividuals
      forall $i \in p$ where NewDeme($i$) $\neq$ this
         remove $i$ from $p$
         add ($i$, NewDeme($i$)) to Station
      forall $(i, d) \in$ Station where $d =$ this
         remove $(i, d)$ from Station
         add $i$ to $p$
```

Notice that some individuals may be placed in the station and some other individuals may be removed from the station simultaneously during one step of Evolve. However, it is not possible that some individual is simultaneously removed and added, because the populations of distinct demes are disjoint.

5. Computing with abstract structures

One reason to compute with abstract structures, rather than with their string representations, is to compute on a higher abstraction level. We illustrate this below by writing AsmL code for a useful graph partition algorithm $\mathcal{A}$. We give an example execution of $\mathcal{A}$ and discuss some uses of $\mathcal{A}$.

Another reason to compute with abstract structures is to guarantee that the result does not depend on representation. This aspect is especially important when the structures in question (e.g. databases) are inputs to large automated systems. Due to time and page limitations, we do not illustrate it here.

5.1. Graph partition algorithm

Our graph partition algorithm $\mathcal{A}$ is a humble relative of graph labeling algorithms used in practice to solve the graph isomorphism problem [10,11]. We think that, taking into account the character of this article (as well as time and page limitations), a simpler graph partitioning algorithm fits our purposes better.

Define a *partition* P of (the vertex set of) a graph $G = (V, E)$ as a map from V to the power set 2^V of V such that every $v \in P(v)$ and every two distinct sets $P(u)$ and $P(v)$ are disjoint. Each $P(v)$ is a *part* of P. We presume that E is symmetric and irreflexive. Here is the algorithm $\mathcal{A}$.

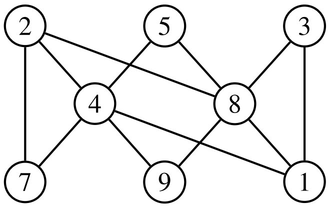

Fig. 1. Sample graph.

```
type Vertex
type Edge = (Vertex, Vertex)
var $V$ as Set of Vertex
var $E$ as Set of Edge
var $P$ as Map of Vertex to Set of Vertex

ComputePartition
   step
      $P := \{x \mapsto V \mid x \in V\}$
      let Nbh $= \{x \mapsto \{y \mid y \in V$ where $(x, y) \in E\} \mid x \in V\}$
   step until fixpoint
      $P := \{x \mapsto \{y \mid y \in P(x)$ where
                          Bag$\{P(z) \mid z \in$ Nbh$(x)\} =$
                          Bag$\{P(z) \mid z \in$ Nbh$(y)\}\} \mid x \in V\}$
```

The type Vertex is abstract. We do not presume that vertices are numbers or strings. Nbh alludes to "neighborhood". Bag$\{P(z) \mid z \in$ Nbh$(x)\}$ is the multiset of sets $P(z)$ where z ranges over Nbh(x). Running $\mathcal{A}$ on a graph $G = (V, E)$, produces a partition $P = \mathcal{A}(G)$ that respects automorphisms of G: if α is an automorphism of G then every $P(\alpha(v)) = P(v)$.

To execute $\mathcal{A}$, we need to extend the AsmL program above, and, in particular, to concretize the type Vertex. For simplicity, the extended program is designed to run on one particular graph, the graph in Fig. 1.

```
Main
   step
      InitializeGraph
   step
      ComputePartition

type Vertex = Integer

InitializeGraph
   $V := \{2, 3, 5, 7, 8, 9, 1, 4\}$
   $E :=$ Symmetrize($\{(7, 4), (7, 2), (5, 4), (5, 8), (3, 8), (3, 1),$
                     $(4, 2), (4, 9), (4, 1), (8, 1), (8, 9), (8, 2)\}$)

Symmetrize(es as Set of Edge) as Set of Edge
   return es $+ \{(y, x) \mid (x, y) \in$ es$\}$
```

Let P_i denote the set of the parts of P before the ith iteration of the assignment of the step-until-fixpoint loop. The following values of P_i can be observed by inserting the AsmL statement WriteLine(P.Values) right before that assignment:

$P_1 = \{\{2, 3, 5, 7, 8, 9, 1, 4\}\}$
$P_2 = \{\{8, 4\}, \{1, 2\}, \{9, 7, 5, 3\}\}$
$P_3 = \{\{8, 4\}, \{1, 2\}, \{9, 5\}, \{7, 3\}\}$

5.2. Using the graph partition algorithm

Given a property π of graphs, let ϕ_n be the fraction of π graphs on $V_n = \{1, 2, \dots, n\}$, that is the number of graphs on $\{1, 2, \dots, n\}$ that possess property π divided by the total number of graphs on V_n. Consider the uniform probability distribution on the collection of graphs of V_n (according to which every two graphs are equally probable), and let G_n be a random graph on V_n. Then ϕ_n is the probability that G_n possesses property π. The property π is called *almost sure* in finite model theory if ϕ_n approaches 1 when n grows to infinity.

Call a partition P of a graph $G = (V, E)$ *ultimate* if every $P(v) = \{v\}$. Almost surely, the partition $\mathcal{A}(G)$ is ultimate. In other words, the probability that $\mathcal{A}(G_n)$ is ultimate, where G_n is a random graph as above, converges to 1 when n grows to infinity. We omit the supporting computation.

Recall that a graph G is *rigid* if the only automorphism of G is the identity automorphism. The *graph rigidity* problem is the problem of determining whether a given graph is rigid. The problem is not known to be polynomial time. Algorithm $\mathcal{A}$ is polynomial time, and it does not always determine the rigidity of a given graph. But it can be used to solve the rigidity problem in practice. Almost surely, the partition $\mathcal{A}(G)$ is ultimate, in which case G is rigid. (It follows that graphs are almost surely rigid.) If, however, $\mathcal{A}(G)$ is not ultimate, then the question whether G is rigid remains open.

There is a trivial algorithm $\mathcal{B}$ for the graph rigidity problem: give the positive answer in all cases. Almost surely, $\mathcal{B}$ is correct. Does $\mathcal{A}$ have any advantage over $\mathcal{B}$? Yes. One advantage is that $\mathcal{A}$ does not have false positives. If partition $\mathcal{A}(G)$ is ultimate, then G is rigid. But there are rigid graphs G such that $\mathcal{A}(G)$ is not ultimate. In that sense, $\mathcal{A}$ has false negatives, while the trivial algorithm does not have false negatives (because it does not have any negatives). Another advantage of $\mathcal{A}$ that it can be helpful in solving the graph isomorphism problem.

The *graph isomorphism problem* is the problem of determining whether two given graphs are isomorphic. It is not known to be in polynomial time. The partition algorithm can be used for a practical solution of the problem in most cases. Given two graphs G and H, run $\mathcal{A}$ on the disjoint sum $G + H$. Almost surely, the result is the ultimate partition of $G + H$, which establishes that G and H are not isomorphic. Clearly this remains true for probabilities conditional on the event that G and H are not isomorphic.

What about the event when G and H are isomorphic? This case is a bit more subtle. So let us clarify what the event is. Given a positive integer n, choose G randomly among the graphs on $\{1, \dots, n\}$, and then choose H randomly among the graphs on $\{n + 1, \dots, 2n\}$ isomorphic to G. In both cases, all candidate graphs are presumed to be equally probable. In this event, almost surely, there is a unique isomorphism from G to H, and $\mathcal{A}$ finds it. It finds the isomorphism in the following sense: each part in $\mathcal{A}(G + H)$ has one vertex from G and one vertex from H, and the resulting map from G to H is an isomorphism. To verify this claim, note that, by the choice of G, almost surely, distinct vertices of G will be in distinct parts. But, since G is isomorphic to H, there is a non-trivial automorphism θ of $G + H$. It follows that, for every vertex u of G, the part of u contains $\theta(u)$ but does not contain any other vertices of G.

6. Concluding remarks

So can abstract state machines be helpful to language theorists? We believe so, and we gave some reasons for our position. The answer to the question rests ultimately with language theorists themselves. They have the knowledge of the latest advances and the understanding of current problems needed to judge. We hope that abstract state machines will facilitate further advances in language theory.

Acknowledgment

We thank Andreas Blass for helpful discussions.

References

[1] Abstract State Machines, http://www.eecs.umich.edu/groups/gasm/.

[2] Abstract State Machine Language (AsmL), http://research.microsoft.com/fse/asml/.

[3] Thomas Bäck, Introduction to evolutionary algorithms, in [4], pp. 59–63.

[4] Thomas Bäck, David B. Fogel, Zbigniew Michaelewicz, Evolutionary Computation 1: Basic Algorithms and Operators, Institute of Physics Publishing, 2000.

[5] Andreas Blass, Yuri Gurevich, Algorithms: A quest for absolute definitions, Bulletin of the European Association for Theoretical Computer Science 81 (2003) 195–225. Reprinted in Current Trends in Theoretical Computer Science, World Scientific (2004) 283–311, and in Church's Thesis After 70 Years, Ontos Verlag (2006) 24–57.

[6] Andreas Blass, Yuri Gurevich, Parallel abstract state machines capture parallel algorithms, ACM Transactions on Computational Logic 4 (4) (2003) 578–651.

[7] Andreas Blass, Yuri Gurevich, Saharon Shelah, Choiceless polynomial time, Annals of Pure and Applied Logic 100 (1999) 141–187.

[8] Andreas Blass, Yuri Gurevich, Saharon Shelah, On polynomial time computation over unordered structures, Journal of Symbolic Logic 67 (3) (2002) 1093–1125.

[9] Egon Börger, Abstract state machines: A unifying view of models of computation and of system design frameworks, Annals of Pure and Applied Logic 133 (2005) 149–171.

[10] Derek G. Corneil, Calvin C. Gotlieb, An efficient algorithm for graph isomorphism, Journal of ACM 17 (1) (1970) 51–64.

[11] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, Mario Vento, A (sub)graph isomorphism algorithm of matching large graphs, IEEE Transactions on Pattern Analysis and Machine Intelligence 26 (2004) 1367–1372.

[12] DLT 2006 website, http://dlt2006.cs.ucsb.edu/.

[13] Erich Grädel, Antje Nowack, Quantum computing and abstract state machines, in: E. Börger (Ed.), Abstract State Machines: Advances in Theory and Applications, in: Lecture Notes in Computer Science, vol. 2589, Springer-Verlag, 2003, pp. 309–323.

[14] Yuri Gurevich, Evolving algebras 1993: Lipari guide, in: E. Börger (Ed.), Specification and Validation Methods, Oxford University Press, 1995, pp. 9–36.

[15] Yuri Gurevich, Sequential abstract state machines capture sequential algorithms, ACM Transactions on Computational Logic 1 (1) (2000) 77–111.

[16] Yuri Gurevich, Interactive algorithms 2005 with added appendix, in: D. Goldin, S.A. Smolka, P. Wegner (Eds.), Interactive Computation: The New Paradigm, Springer, 2006, pp. 165–182.

[17] Yuri Gurevich, Benjamin Rossman, Wolfram Schulte, Semantic essence of AsmL, Theoretical Computer Science 343 (3) (2005) 370–412.

[18] Dexter C. Kozen, Automata and Computability, Springer, 1997.

[19] Mariusz Nowostawski, Riccardo Poli, Parallel genetic algorithm taxonomy, in: Proc. Third International Conference on Knowledge-based Intelligent Information Engineering Systems, KES, 1999, pp. 88–92.

[20] Grzegorz Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1 — Foundations, World Scientific, 1997.

[21] Spec Explorer, http://research.microsoft.com/specexplorer/.

[22] Alan Turing, On computable numbers, with an application to the Entscheidungsproblem, Proceedings of London Mathematical Society (Series 2) 42 (1936–1937) 230–265;
Alan Turing, On computable numbers, with an application to the Entscheidungsproblem, Proceedings of London Mathematical Society (Series 2) 43 (1936–1937) 544–546 (Correction).