Remarks on Turbo ASMs for Functional Equations and Recursion Schemes

Egon Börger[1] and Tommaso Bolognesi[2]

[1] Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
boerger@di.unipi.it
[2] CNR, ISTI, Pisa bolognesi@ei.pi.cnr.it

Abstract. The question raised in [15] is answered how to naturally model widely used forms of recursion by abstract machines. We show that turbo ASMs as defined in [7] allow one to faithfully reflect the common intuitive single-agent understanding of recursion. The argument is illustrated by turbo ASMs for Mergesort and Quicksort. Using turbo ASMs for returning function values allows one to seamlessly integrate functional description and programming techniques into the high-level 'abstract programming' by state transforming ASM rules.

1 Introduction

This paper has been triggered by two recent publications, namely [15] and [3]. In the first paper Moschovakis claims that identifying algorithms with abstract machines 'does not square with our intuitions about algorithms and the way we interpret and apply results about them' and suggests that 'algorithms are *recursive definitions* while machines model *implementations*, a special kind of algorithms'. A concrete challenge is stated in the form of a question, namely 'If algorithms are machines, then which machine is the mergesort?' In the second paper Blass and Gurevich react to what they call 'a provocative article...casting doubt on the ASM thesis'. Their answer to the concrete challenge consists in describing 'the mergesort algorithm, on its natural level of abstraction, in terms of distributed abstract state machines', instantiating the more general scheme in [13] with the intention to support the ASM thesis ('Every algorithm can be expressed, on its natural level of abstraction, by an ASM').

Before going to explain the reasons why we are dissatisfied with this solution and to expose what we believe to be a simpler answer which is based upon basic mono-agent ASMs and better supports the ASM thesis, a double proviso should make clear that our explanation is a technical one, driven by the concern for a clean and transparent integration of well-established programming techniques into high-level system design by ASMs. As a consequence our explanation tries to avoid the difficult epistemological issues raised by Moschovakis, Blass and Gurevich. On the one side, what is 'the natural level of abstraction' of an algorithm, in this particular case of mergesort, is to a large extent in the eye of the beholder, but our impression (and the feedback we got from asking around) is that rarely somebody thinks of a *distributed* computation when reasoning about or trying to explain or to implement common recursive algorithms like mergesort. In fact Blass and Gurevich themselves admit in [3, Sect.6] that the runs of distributed ASMs defined in [11], which their explanation of mergesort-like recursions crucially relies upon, are 'a long story', whereas our students learn to understand and use recursion quickly and rather early in their career. Blass and Gurevich base their explanation of recursion on the conversion of recursive ASMs to distributed ASMs defined in [13], a quite complex procedure they invoke to express the standard mathematical term evaluation mechanism for recursive equations:

> Converting a recursive ASM to a distributed ASM amounts to making explicit the creation of vassals, the waiting for the vassals' return values, and the use of these return values to continue the computation.

On the other side we do not want to enter the discussion whether algorithms 'are' recursive definitions, although certainly there are many examples most of us would call algorithms which however are not reducible in a natural way to recursive definitions.

Our answer to Moschovakis' question is two-fold. Formulated in general terms it says that functional languages, e.g. Haskell, or more precisely their interpreters, are well-known abstract machines for mergesort-like algorithms, since they offer a perfect match for defining and computing algorithms by such systems of recursive equations. In that sense Moschovakis' question has been answered already a long time ago when the techniques for implementing functional programming languages were developed. However one may argue that defining such interpreters has to deal with features which are typical for *implementing* functional languages, like passing subcomputation values, spawning or deleting subtasks, etc.—routine matters for implementors of functional languages and involving much more than a mathematical user of systems of recursive equations would like to see. We take this as a legitimate argument and therefore provide here also a more concrete answer by directly defining the machinery which underlies the way how recursive equations are used by humans to compute the value of the function in question, avoiding any spurious implementation related feature. We explicitly extract here the high-level *mathematical* machinery which is tacitly used for the standard functional calculations and without which the equations would not constitute the description of an algorithm (but in the best case a specification, as Blass and Gurevich point out). This machinery is easily defined as a simple mono-agent abstract state machine, namely an abstract form of well-known mechanisms for algorithmic term evaluation procedures. It avoids to invoke multiple agents and distributed computations, and directly reflects what the average computer scientist refers to when explaining how to use recursive equations.

For concrete comparison we illustrate the argument on the mergesort algorithm discussed in [15, 3] and on quicksort, but our definitions explain in full generality how functional programming concepts and methods can be modeled faithfully by a special class of ASMs and how they can thereby be naturally integrated into the state based ASM framework. Since we want our argument to provide further concrete evidence for the above stated ASM thesis [12, 2], we take care to provide our model for the conceptual ingredients of functional programming at a more abstract level than that of the ASM engines AsmGofer [16] and AsmL [9]—which implement an ASM based framework enriched by functional and object-oriented programming features—and also of the implementation of procedure calls in XASM [1, 17].

2 Turbo ASMs

As has often been critically observed, the characteristic feature of basic ASMs—simultaneous execution of multiple atomic actions in a global state—comes at a price, namely the lack of direct support for practical composition and structuring principles. To make such features available as standard refinements of basic ASMs, in [7] a class of ASMs has been defined which offers as building blocks sequential composition **seq**, iteration **iterate**, and parameterized (possibly recursive) submachines. These so-called *Turbo ASMs* realize a black-box view of their component machines, hiding the internals of their (sub)computations by compressing them into one step (hence the name) and thus fit the synchronous parallelism of standard ASMs.

Purely iterative turbo ASMs, built by **seq** and **iterate** from function updates using as background functions only the initial functions of recursion theory, have been shown in [7, Sect.3.3] to compute arbitrary computable functions in a way which combines the advantages of Gödel-Herbrand style functional and of Turing style imperative programming. For parameterized submachines also *value returning ASMs* have been defined by extracting from the global result of a turbo submachine computation what one wants to consider as result value. A simple variation of that definition suffices to support the more general goal we want to reach here, namely to justify the smooth integration of stateless recursive functional programming into the state based synchronous parallel ASM framework by a natural subclass of value returning turbo ASMs. These machines are one-agent ASMs with multiple, synchronous parallel rules (or submachines) which express the usual procedure for evaluation of functional recursive equations and thus are our candidates for answering Moschovakis' question.

The following definition of *turbo submachine call* is taken unchanged from [7, Sect.4].

Definition 1. *Let $R(x_1, \ldots, x_n) = body$ be the declaration of a named turbo ASM rule R, let $\mathfrak{A}$ be a state. If $[\![body[a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}}$ is defined, then also $[\![R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}}$ is defined and its value is*
$$[\![R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}} = [\![body[a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}}.$$

By this definition every call $R(a)$ of a turbo submachine provides its global result in *one* step (if it terminates at all), namely by yielding the cumulative update set of its entire computation (where in the case of overwriting due to sequential execution the last write wins).

To exploit this atomic view of turbo ASMs for returning values by machines which are supposed to compute functions of their input, it suffices to project that value out of the total computational effect $[\![R(a_1,\ldots,a_n)]\!]^{\mathfrak{A}}$. For this purpose in [7, Sect.5.2] the usual notation $l \leftarrow R(a_1,\ldots,a_n)$ is adopted. We want the machine to store the result of R in location l, so that at the end of the R-computation the location l will contain the intended result value, in accordance with the expectation the programmer associates with this notation. R is a named rule as in the preceding definition in which a reserved 0-ary function **result** occurs with a placeholder role for storing the return value[1]. **result** represents the interface offered for communicating results from a rule execution to a location l which can be determined by the caller. Formally this comes up to replace **result** in the computation of R by l.

$$[\![l \leftarrow R(a_1,\ldots,a_n)]\!]^{\mathfrak{A}} = [\![body[l/\ \mathbf{result}, a_1/x_1,\ldots,a_n/x_n]]\!]^{\mathfrak{A}}$$

This definition should not be misunderstood as an invitation to consider value returning machine calls as a form of side-effect producing evaluation of 'terms' $R(t)$. On the contrary we believe that it is a precious methodological insight of Gurevich's ASM concept to only have main effects—updates which change the given state as defined by the rules—and to *abstract* from any other effect, rather than having them around as uncontrolled 'side' effects. As a consequence we believe it to be a pragmatically advisable system design principle not to blur the distinction between terms (which evaluate to objects) and rules (which evaluate to update sets triggering state changes), although the functional and λ-calculus tradition in theoretical computer science shows how one can make sense out of identifying objects and algorithms by turning everything into a function. Sticking to the convenient distinctions does not prevent us from having a mechanism to extract from a state some particular values one may wish to consider as distinguished objects coming out from subcomputations. In fact calling $R(t)$ as defined in [7] means that R is executed as a turbo ASM with parameter t and produces a complete state change, namely through its update set produced starting from the call state. The notation $l \leftarrow R(t)$ allows the designer to project out of that 'result' state the value of a specific location, namely the one named by l[2]. The turbo ASM notation $R(t)$ by itself does not provide any state hiding mechanism (though it can easily be enhanced to do so by standard modularization methods for rule import/export, (re)naming conventions for submachine signatures, local state concepts, etc., see e.g. [7]).

[1] Should a tuple of return values be desired, one would have finitely many such 0-ary functions $\mathbf{result}_j$ for $j = 1,\ldots,m$.

[2] In [14, pg.19] it is stated that "the idea of XASM is to generalize the original idea of Gurevich, resulting in a more practical specification and implementation tool" by introducing a new concept of "XASM call" that "leads to a design where every construct (including expressions and rules of Gurevich's ASMs) is denoted by both a value and an update set" (op.cit.pg.2). This throwing together of algorithms and the objects they compute, blurring the distinction between rules and terms, may have led to the thorough misunderstanding of turbo ASMs which transpires from

3 Modeling Recursion by Value Returning Turbo ASMs

There are many ways to explain the meaning of various forms of recursion. The turbo ASM submachine concept reviewed above abstractly mimics a standard imperative calling mechanism, which provides the key for laying in this section a particularly simple and rigorous foundation for the common intuitive understanding of recursion in terms of single-agent ASM computations. Furthermore by the atomicity of their black-box computations, turbo ASMs allow us to reflect exactly the machinery which underlies the common mathematical use of *functional* equations to evaluate functions defined by recursion, as we illustrate in this section for the Quicksort and Mergesort algorithms.

The update set produced by executing a turbo ASM call represents the total effect of executing the submachine in the call state (atomicity of the turbo ASM computation). Using the machine to return a value adds a form of *functional abstraction* from everything in that computation except the resulting input-output (argument-value) relation[3]. Technically we combine the turbo ASM notation for value returning machines with the **let**-construct, mimicking the use of activation records to store parameters as local variables. Since for each submachine call a dedicated placeholder is needed to record the result of a subcomputation, in the following definition we apply an external function *new* to the dynamic set FUN_0 of 0-ary dynamic functions ('write variables'). *new* is supposed to provide each time a completely fresh location, i.e. a location which has never been used before and is also not used for any other simultaneous call of *new*. This implies that if a machine is invoked within a same step simultaneously on different arguments $(a_1, ... a_n)$, *new* is assumed to provide as many result locations as there are invocations. For a detailed mathematical definition of this use of *new* see [10], [8, Ch.2.4.4][4].

the statement in op.cit.pg 2 that turbo ASMs are "excluding the essential feature of both Anlauff's and May's original call to allow returning not only update sets, but as well a value." The definition from [7] reported above makes clear that the statement is simply wrong, as is the conclusion drawn from it in op.cit. pg.2 that "This restriction makes their (i.e. Börger's and Schmid's) call useless for the modeling of recursive algorithms" (see the next section). The reader is the judge whether the above definition of turbo ASMs, and their use to extract return values from subcomputations, reflects current programming practice and does it in a transparent way. As reported in the comment on May's work in [6], it was the desire to simplify—read: to avoid the complexity of unnecessarily general logico-algebraic concepts for explaining standard programming concepts—which led Börger and Schmid to look for an *elementary* and transparently implementable definition of submachines which fits the characteristic synchronous parallelism of ASMs.

[3] The pure functional effect of course is achieved only if the submachine computation on the caller's side affects only the result location.

[4] As alternative for the use of *new* one can turn **result** into a monadic function which takes the list of parameters as arguments, so that the results of invokations with different arguments are stored in different locations.

Definition 2. *(Using return values in turbo ASMs). Let R_i, S be arbitrary turbo ASMs with formal parameter sequences x_i of R_i and parameters y_i of S . For corresponding actual parameter sequences a_i we define:*

```
**let** $\{y_1 = R_1(a_1), \ldots, y_n = R_n(a_n)\}$ **in** $S \equiv$
  **let** $l_1, \ldots, l_n = new(FUN_0)$ **in**
    **forall** $1 \leq i \leq n$ **do** $l_i \leftarrow R(a_i)$
    **seq**
    **let** $y_1 = l_1, \ldots, y_n = l_n$ **in** $S$
```

This definition allows one to explicitly capture the abstract machine which underlies the common mathematical evaluation procedure for functional expressions, including those defined by forms of recursion. The definition does not invite to mix terms (which are to be evaluated to objects) and machines (which have to be executed to obtain the desired state change). In fact for rules R and terms t the use of $R(t)$ is justified by our definition only as part of a call of R (i.e. in a place where a rule is expected) and in one of the two forms $l \leftarrow R(t)$ or **let** $x = R(t)$ **in** S. Passing (by value) the result returned by a turbo ASM captures the implicit storage of intermediate values by subterms during the evaluation of functional equations[5]. We illustrate this by the following turbo ASM definitions of Quicksort and of Mergesort which directly translate the usual recursive definition of the algorithms to provide as result a sorted version of any given list.

The computation suggested by the well-known recursive equations to quicksort L proceeds as follows: FIRST partition the *tail* of the list into the two sublists $tail(L)_{<head(L)}, tail(L)_{\geq head(L)}$ of elements $< head(L)$ respectively $\geq head(L)$ and quicksort these two sublists separately (independently of each other), THEN *concatenate* the results placing $head(L)$ between them. The fact that this description uses various auxiliary list and comparison operations is reflected by the appearance of corresponding auxiliary functions in the following turbo ASM.

```
QUICKSORT$(L)$ =
  **if** $| L | \leq 1$ **then result** := $L$ **else**
    **let**
      $x =$ QUICKSORT$(tail(L)_{<head(L)})$
      $y =$ QUICKSORT$(tail(L)_{\geq head(L)})$
    **in result** := $concatenate(x, head(L), y)$
```

In the usual verbal paraphrase reported above of the recursive equations for Quicksort there is no mention of agents, certainly no mention of multiple agents or of creation of 'vassals', not even of 'return'ing (by whom? from where?) values, but only of a certain order of some computation steps which explicitly appear as

[5] Introducing a stack discipline constitutes a further step towards an implementation. Since for each call we provide a new location, we can abstract from their ordering (e.g. in a stack one being on top of the other) and from their deletion (which remains for a refinement by a garbage collector, e.g. by a stack operation *pop*).

the 'principal' steps, namely to first compute some intermediate values for certain arguments—the result values of the two recursive subcomputations—and then 'to use...these...values to continue the computation' to produce the desired 'main' result. This is exactly what our formal ASM turbo machine definition mentions, abstracting from all the details of how these subcomputations are organized, as does the system of recursive equations.

On the other side, it seems to us that the definition of our QUICKSORT turbo ASM exactly mimics the usual recursive equations for the algorithm, and is as far from 'modeling an implementation' as the 'recursive definitions' are. Our translation of functional terms $f(s)$ using the construct **let** $x = F(s)$ applied to turbo ASM computations F directly captures the common procedure to evaluate in a certain order the subterms appearing in equations, to 'keep in mind' a certain number of them for a certain time until the moment comes where they are used simultaneously[6]. This mixture of ordering and independence is captured by the appropriate mixture of synchronous parallelism, sequentialization and turbo effect (atomic submachine view) provided by turbo ASMs.

One can argue whether computing $tail(L)_{<head(L)}, tail(L)_{\geq head(L)}$ is part of the quicksort machine computation or not. The answer to the question depends on the level of abstraction at which one wants to define the machine. One can show that the following procedural refinement of list partitioning in Quicksort is correct and provides a correct 'implementation' (to use Moschovakis' term) of the more abstract machine above, though we would consider it still as an abstract machine.

Data refinement of QUICKSORT. One can refine QUICKSORT to a control state [7]turbo ASM where the partitioning of L into $L_{<head(L)}$ and $L_{\geq head(L)}$ is computed using the following basic ASM PARTITION(l, h, p), working on the representation of lists as functions $L\colon [r, s] \rightarrow VAL$ from intervals of natural numbers to a set of values. When $r < s$, *Partition* is started with the search boundaries $l = r, h = s$ and the list head $pivot = L(r)$. It terminates when reaching $l = h$ with $L(l) = pivot$, all L-elements smaller than the pivot to the left of l, and all the others at l or to the right of l. Until reaching $l = h$, the partitioning procedure alternates between searching from above for list elements $L(h) \leq pivot$ and searching from below for list elements $L(l) \geq pivot$. When such an element is encountered and it is different from the element at the other current search boundary—one of them is the pivot—, then the boundary elements $L(l), L(h)$ are swapped and the search switches to the other boundary. When $L(h) \leq pivot \leq L(l) \leq L(h)$ before $l = h$ is encountered (namely when pivot has multiple occurrences in the list), h can be decreased by one.

[6] This virtual intermediate storage of 'keeping in mind' intermediate results computed for subterms, typical for the functional handling of equations, is reflected in the definition **let** $x = R(a)$ **in** S by providing new locations where to temporarily keep the results of the subcomputations, to be used 'later' when executing S.

[7] Control state ASMs defined in [4] constitute a subclass of basic ASMs, extending the classical Finite State Machines by allowing rules to be executed when passing from one to another internal ('control') state. They are extensively used in [8].

```
PARTITION$(l, h, pivot)$ =
  **if** $L(h) > pivot$ **then** $h := h - 1$
  **elseif** $L(l) < pivot$ **then** $l := l + 1$
  **elseif** $L(l) > L(h)$ **then**
    $L(l) := L(h)$
    $L(h) := L(l)$
  **elseif** $l < h$ **then** $h := h - 1$
```

We are going now to illustrate our argument further by the Mergesort example, which is the one discussed by Moschovakis and Blass and Gurevich. The computation suggested by the usual recursive equations to mergesort a given list L consists in FIRST splitting it into a *LeftHalf*(L) and a *RightHalf*(L) (if there is something to split) and mergesorting these two sublists separately (independently of each other), THEN to *Merge* the two results by an auxiliary elementwise *Merge* operation. This is expressed by the following turbo ASM which besides two auxiliary functions *LeftHalf*, *RightHalf* comes with an external function *Merge* defined below as a submachine.

```
MERGESORT$(L)$ =
  **if**$| L | \leq 1$ **thenresult**$:= L$ **else**
    **let**
      $x =$ MERGESORT$(LeftHalf(L))$
      $y =$ MERGESORT$(RightHalf(L))$
    **in result**$:= Merge(x, y)$
```

Usually also *Merge* is defined by a recursion, suggesting the following computation scheme which is formalized by the turbo ASM below. If both lists are non-trivial, by a case distinction the smaller one of the two list heads is determined and placed as the first element of the *result* list, concatenating it with the result of a separate and independent *Merge* operation for the two lists remaining after having removed the chosen smaller head element. The ι-operator in $\iota x(P)$ denotes the unique x with property P (if there is such an x).

```
MERGE$(L, L')$ =
  **if** $L = \emptyset$ **or** $L' = \emptyset$ **then result**$:= \iota l(l \in \{L, L'\}$ **and** $l \neq \emptyset)$
  **elseif** $head(L) \leq head(L')$ **then**
    **let** $x =$ MERGE$(tail(L), L')$ **in result**$:= concatenate(head(L), x)$
  **elseif** $head(L') \leq head(L)$ **then**
    **let** $x =$ MERGE$(L, tail(L'))$ **in result**$:= concatenate(head(L'), x)$
```

Data refinement of MERGESORT **and** MERGE. One can show that the following definition of equivalence of corresponding locations provides a rule-wise step-by-step refinement to a model where lists are represented as functions $L\colon [l, h] \to VAL$ from intervals of natural numbers to a set of values. We leave it to the reader to decide whether the result of this refinement should be considered to be an abstract machine or an implementation.

– $L = \emptyset \equiv l > h$, $\mid L \mid \leq 1 \equiv l \geq h$
– $head(L) = L(l), tail(L) = L \setminus \{(l, L(l))\}$
– $LeftHalf(L) = L \restriction [l, half(l+h)]$, $RightHalf(L) = L \restriction [half(l+h)+1, h]$
  where $half(2x) = x, half(2x+1) = x+1$ (for example)
– $concatenate(v, L) = \{(l, v)\} \cup RightShift(L, 1)$
  where $RightShift(L, n) = \lambda x.L(x - n)$

In [15] also the issue of *outputting intermediate results* is mentioned, e.g. to compute the infinite sequence of primes. This can easily be incorporated by updates of the form **result**:=**result** $*newItem$ which extend the current result by a new item. Thus if the fixpoint of a computation is infinite, **result** will become a piecewise computed infinite sequence.

4 Conclusion

Result returning turbo ASMs make two ingredients of the evaluation of functional equations explicit which are taken for granted and—from the specification point of view luckily—abstracted away by the functional notation: the implicit storage of intermediate values by subterms and the independence or ordering of their computation and use in the evaluation procedure. If one wants to consider these abstract machines as 'mathematical models of computers'[15], then one should view also systems of recursive equations that way. Through our refinement definition (Def. 2) the concrete details of the machine computation are appropriately hidden, and at the same time provide a seamless connection to further implementation steps through unfolding the machine notation. The natural and tight correspondence between sets of recursive equations and these machines contributes to the pragmatic evidence accumulated over the years (see [6]) that ASMs can indeed be tailored to describe arbitrary algorithms on their natural level of abstraction, here the level of evaluating functional equations.

We pointed out above that our answer to Moschovakis' question is quite different from the one in [13, 3] where it is argued that 'recursive computations are to be viewed as a special case of distributed computation' in which each recursive call is executed by a newly created callee agent expected to return his result to the caller. The way we define the result of the turbo ASM call implies that from the caller's view it is returned immediately, directly reflecting the functional view which only uses the result in the given evaluation process and abstracts from how and by whom the result has been obtained. As a consequence our explanation of recursion does not need to invoke multiple agents, but it is compatible with using them for a distributed implementation, which is at a more detailed level of modeling than the functional one. To view recursive computations as a special case of distributed computation implies to considerably lower the level of computational abstraction where the algorithm is analysed (or to increase the level of detailing if one prefers to look at it the other way round[8]).

A related criticism of pushing the interpretation of the ASM thesis beyond its natural limits has been formulated in [5].

Acknowledgement. We thank the following colleagues for discussing and critically commenting upon a draft of this paper: W. Ahrendt, R. Banach, H. Langmaack, P. Päppinghaus, D. Rosenzweig, R. Stärk, W. Zimmermann and an anonymous referee.

References

1. M. Anlauff. XASM — an extensible, component-based abstract state machines language. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 69–90. Springer-Verlag, 2000.
2. A. Blass and Y. Gurevich. Abstract State Machines capture parallel algorithms. *ACM Transactions on Computational Logic*, 3, 2002. .
3. A. Blass and Y. Gurevich. Algorithms vs. machines. *Bulletin EATCS*, 2002.
4. E. Börger. High level system design and analysis using abstract state machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in LNCS, pages 1–43. Springer-Verlag, 1999.
5. E. Börger. Computation and specification models. A comparative study. In P. Mosses, editor, *Proc. FLoC'02 Workshop Action Semantics and Related Semantic Frameworks*, BRICS Series. Department of Computer Science at University of Aarhus, 2002.
6. E. Börger. The origins and the development of the ASM method for high level system design and analysis. *J. of Universal Computer Science*, 8(1):2–74, 2002.
7. E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *LNCS*, pages 41–60. Springer-Verlag, 2000.
8. E. Börger and R. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
9. Foundations of Software Engineering Group, Microsoft Research. AsmL. Web pages at http://research.microsoft.com/foundations/AsmL/, 2001.
10. N. G. Fruja and R. F. Stärk. The hidden computation steps of turbo Abstract State Machines. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003*. Springer, 2003.
11. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
12. Y. Gurevich. Sequential Abstract State Machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.
13. Y. Gurevich and M. Spielmann. Recursive Abstract State Machines. *J. of Universal Computer Science*, 3(4):233–246, 1997.

[8] In fact Blass and Gurevich wonder in [3, Sect.4] whether one 'shouldn't consider a broader class of implementations' than the one apparently Moschovakis is refering to. Once a recursive call has become a computation in a distributed context, many different possibilities to organize the subcomputations show up which cannot even be formulated in terms of the functional recursive equations.

14. P. Kutter. The formal definition of Anlauff's eXtensible Abstract State Machine. TIK-Report 136, ETH Zürich, June 2002.
15. Y. N. Moschovakis. What is an algorithm? In B. Engquist and W. Schmid, editors, *Mathematics Unlimited—2001 and beyond*. Springer, 2001.
16. J. Schmid. Executing ASM specifications with AsmGofer. Web pages at http://www.tydo.de/AsmGofer.
17. X.ASM. Xasm Open Source. Web pages at http://www.xasm.org/, 2001.