Journal of Universal Computer Science, vol. 3, no. 4 (1997), 233-246
submitted: 20/12/96, accepted: 21/4/97, appeared: 28/4/97 © Springer Pub. Co.

Recursive Abstract State Machines

Yuri Gurevich[1]
(University of Michigan, USA
gurevich@umich.edu)

Marc Spielmann[2]
(RWTH Aachen, Germany
masp@informatik.rwth-aachen.de)

Abstract: According to the ASM thesis, any algorithm is essentially a Gurevich abstract state machine. The only objection to this thesis, at least in its sequential version, has been that ASMs do not capture recursion properly. To this end, we suggest recursive ASMs.

Key Words: abstract state machines, recursion, distributed computations, concurrency

Category: F.1.1, F.1.2

1 Introduction

The *abstract state machine* (formerly *evolving algebra*) thesis [Gurevich 91] asserts that abstract state machines (*ASMs*, for brevity) express algorithms on their natural level of abstraction in a direct and coding-free manner. The thesis is supported by a wide spectrum of applications [Börger 95], [Castillo 96], [Huggins 96]. However, some people have objected that ASMs are iterative in their nature, whereas many algorithms (e.g., Divide and Conquer) are naturally recursive. In many cases recursion is concise, elegant, and inherent to the algorithm. The usual stack implementation of recursion is iterative, but making the stack explicit lowers the abstraction level. There seems to be an inherent contradiction between

- the ASM idea of explicit and comprehensive states, and
- recursion with its hiding of the stack.

But let us consider recursion a little more closely. Suppose that an algorithm A calls itself. Strictly speaking it does not call itself; rather it creates a clone of itself which becomes a sort of a slave of the original. This gives us the idea of treating recursion as an implicitly distributed computation. Slave agents come and go, and the master/slave hierarchy serves as the stack.

Building upon this idea, we suggest a definition of recursive ASMs. The implicit use of distributed computing has an important side benefit: it leads naturally to concurrent recursion. In addition, we reduce recursive ASMs to distributed ASMs as described in the Lipari guide [Gurevich 95]. If desired, one can view recursive notation as mere abbreviation.

[1] Partially supported by NSF grant CCR 95-04375 and ONR grant N00014-94-1-1182.
[2] Visiting scholar at the University of Michigan, partially supported by DAAD and The University of Michigan.

The paper is organized as follows. In [Section 2], we introduce a restricted model of recursive ASMs, where the slave agents do not change global functions and thus do not interfere with each other. The syntax of ASM programs is extended with a `rec` construct allowing recursive definitions like those in common programming languages. We then describe a translation of programs with recursion into distributed programs without recursion. In [Section 3], we generalize the model by allowing slave agents to change global functions. As a result, the model becomes non-deterministic. Finally, in [Section 4] we restrict the general model of [Section 3] so that global functions can be changed but determinism is ensured by sequential execution of recursive calls.

Conventions

The paper is based on the Lipari guide [Gurevich 95] and uses some additional conventions. The executor of a one-agent ASM starts in an initial state with *Mode* = *Initial* and halts when *Mode* = *Final*. A distributed ASM of the kind we use in this paper has a module *Main*, executed by the *master agent*, and additional modules $F_1, \ldots, F_n$, executed by *slave agents*. In the case of slave agents, the *Mode* function is actually a unary function *Mode*(*Me*). (The distinction between master and slave agents is mostly didactic.) As usual, the semantics of distributed ASMs is given by the class of possible *runs* [Gurevich 95]. Notice that in general this semantics is non-deterministic; different finite runs may lead to different final states.

We say that an atomic subrule of a rule R is *enabled* in a state S, if it contributes to the a priori update set of R at S (which may be wedded as the final update set of R at S is computed.) In other words, an atomic rule is enabled at S, if all the guards leading to it are true at S. Sometimes we abbreviate $f(x)$ to $x.f$ for clarity.

2 Concurrent Recursion without Interference

We start with a restricted model of recursion where different recursive calls do not interfere with each other although their execution may be concurrent. In applications of distributed ASMs, one usually restricts the collection of admissible (or regular) runs. Because of the non-interference of recursive calls here, in the distributed presentation of a recursive program, we can leave the moves of different slave agents incomparable, so that the distributed ASM has only one regular run and is deterministic in that sense.

2.1 Syntax

Definition 2.1 (Recursive program). A *recursive (ASM) program* Π consists of

1. a one-agent (ASM) program Π_{main}, and
2. a sequence Π_{rec} of *recursive definitions* of the form

```
rec F_i(Arg_i1, ..., Arg_ik_i)
     Π_i
endrec
```

Here Π_i is a one-agent program and each F_i (respectively, Arg_{ij}) is a k_i-ary (respectively, unary) function symbol which is an external function symbol in Π (respectively, Π_i). Formally speaking, any function f updated in Π_i as well as any Arg_{ij} has *Me* as its first/only argument, so that every such function is local. (We will relax this restriction in [Section 3].) For readability *Me* may be omitted.

Optionally, one may indicate the type of any Arg_{ij} or the type of F_i. A *type* is nothing but a *universe* [Gurevich 95]. All types in Π_{rec} should be universes of Π_{main}. Notice that type errors can be checked by additional guards. □

The definition easily generalizes to the case where, instead of Π_{main}, one has a collection of such one-agent programs. However, in this paper we stick to the single-agent program Π_{main}.

Example 2.2 (ListMax). The following recursive program $\Pi = (\Pi_{\mathrm{main}}, \Pi_{\mathrm{rec}})$ determines the maximum value in a list L of numbers using the divide and conquer technique. Π_{main} is

```
if Mode = Initial then
    Output := L.ListMax
    Mode := Final
endif
```

where L is a nullary function symbol of type *list*, and Π_{rec} is the recursive definition

```
rec ListMax(List : list) : int
    if List.Length = 1 then
        Return := List.Head
    else
        Return := Max(List.FirstHalf.ListMax, List.SecondHalf.ListMax)
    endif
    Mode := Final
endrec
```

The functions *List*, *Return* and *Mode* in the *body* Π_{ListMax} of the recursive definition, are local. In other words, they have a hidden argument *Me*.

Starting at an initial state S_0, the master agent computes the next state. This involves computing the recursively defined *L.ListMax*. To this end, it creates a slave agent a, passes to a the task of computing *L.ListMax*, and then remains idle till a hands over the result. When a starts working on Π_{ListMax}, it finds *Me.Mode* = *Initial*, *Me.Return* = *undef* and *Me.List* = L. Essentially, a acts on Π_{ListMax} like the master agent on Π_{main}: if *Me.List.Length* $\neq 1$, then a creates two new slave agents b and c computing *Me.List.FirstHalf.ListMax* and *Me.List.SecondHalf.ListMax*, respectively. When eventually *Me.Mode* = *Final*, *Me.Return* contains $\max\{x \mid x \in L\}$ and a stops working. In general, we use the unary function *Me.Return* to pass the result of a slave agent to its creator. Thus in our example, after receiving a's result, the master agent moves to a final state by updating *Output* with a's result and *Mode* with *Final*, and then it stops.

Syntactically the program looks quite similar to a standard implementation in a common imperative programming language like PASCAL or C. However, its informal semantics suggests a parallel implementation: associate with each agent a task executable on a multi-processor system. Before a task handles the `else` branch of Π_{ListMax}, it has to create two new tasks which compute *List.FirstHalf.ListMax* and *List.SecondHalf.ListMax*. One or both of the new tasks may be executed on another processor in parallel.

On the other hand, using many tasks may not be intended. One may wish to enforce sequential execution. A slight modification of Π_{ListMax} ensures that in every state a slave agent will find at most one enabled recursive call and thus creates at most one new slave agent. Since every agent waits for a reply of its active slave, the agents execute one after another.

```
rec SeqListMax(List : list) : int
    if Mode = Initial then
        if List.Length = 1 then
            Return := List.Head
            Mode := Final
        else
            FirstHalfMax := List.FirstHalf.SeqListMax
            Mode := Sequential
        endif
    endif

    if Mode = Sequential then
        Return := Max(FirstHalfMax, List.SecondHalf.ListMax)
        Mode := Final
    endif
endrec
```

□

Example 2.3 (Savitch's Reachability). To prove PSPACE = NPSPACE, Walter Savitch has suggested the following recursive algorithm for the REACHABILITY decision problem, which works in space $\log^2(\mathit{GraphSize})$. Some familiarity with Savitch's solution [Savitch 70] would be hepful for the reader. (We assume that the input is an ordered graph with constants *FirstNode* and *LastNode*, and a unary node-successor function *Succ*):

```
if Mode = Initial then
    Output := Reach(StartNode, GoalNode, log(GraphSize))
    Mode := Final
endif

rec Reach(From, To : node, l : int) : bool
    if Mode = Initial then
        if l = 0 then
            if From = To or Edge(From, To) then
                Return := true
            else
                Return := false
            endif
            Mode := Final
        else
            Thru := FirstNode
```

```
            Mode := CheckingFromThru
        endif
    endif

    if Mode = CheckingFromThru then
        FromThru := Reach(From, Thru, l − 1)
        Mode := CheckingThruTo
    endif

    if Mode = CheckingThruTo then
        ThruTo := Reach(Thru, To, l − 1)
        Mode := CheckingThru
    endif

    if Mode = CheckingThru then
        if FromThru and ThruTo then
            Return := true
            Mode := Final
        elseif Thru ≠ LastNode then
            Thru := Succ(Thru)
            Mode := CheckingFromThru
        else
            Return := false
            Mode := Final
        endif
    endif
endrec
```

If we remove the second and third rules in Π_{Reach} and instead add the following rule, a parallel execution is possible (which may, however, blow up the space bound).

```
    if Mode = CheckingFromThru then
        FromThru := Reach(From, Thru, l − 1)
        ThruTo := Reach(Thru, To, l − 1)
        Mode := CheckingThru
    endif
```
□

2.2 Translation to distributed ASMs

This subsection addresses those readers who are interested in a formal definition of the semantics of recursive programs.

There are many ways to formalize the intuition behind Definition 2.1. For example, one can define a one-agent interpreter for ASMs which treats $F_1, \ldots, F_n$ in Π_{main} as external functions. Whenever such an external function F_i has to be computed, the interpreter suspends its work and starts evaluating Π_i with $Arg_{i1}, \ldots, Arg_{ik_i}$ initialized properly. When eventually $Mode = Final$ for Π_i, the interpreter reactivates Π_{main} and uses *Return* as the external value. Notice that suspension and reactivation are the main tasks of implementing recursion by iteration. Typically this is realized with a stack. The one-agent interpreter sketched above can use a stack to keep track of the calling order.

Here, we describe a translation of a recursive program Π into a distributed program Π' and in this way define the semantics of Π by the runs of Π'. Suspension and reactivation is realized with a special nullary function *RecMode*. The master/slave hierarchy serves as the stack. (A more general approach would be to add a construct for suspending and reactivating agents to the formalism of distributed ASMs. The introduction of such a construct may be addressed elsewhere.) We concentrate on a useful subclass of recursive programs, where

- no recursive call occurs in a guard or inside a `vary` rule, and
- there is no nesting of external functions (with recursively defined functions counted among external functions).

A translation of recursive programs in the sense of Definition 2.1 is possible but becomes tedious in its full generality. All recursive programs in this paper satisfy the above conditions. In fact, we made Example 2.3 a little longer than necessary in order to comply with the first condition. For instance, instead of using boolean variables *FromThru* and *ThruTo* in the last rule of Π_{Reach} one might directly call $\mathit{Reach}(\mathit{From}, \mathit{Thru}, l-1)$ and $\mathit{Reach}(\mathit{Thru}, \mathit{To}, l-1)$, respectively.

The main idea of the translation is to divide the evaluation of Π_{main} into two phases:

A. Create slave agents (suspension): At a given state S, create a separate agent for every occurrence of every term $F_i(\overline{s})$ in an atomic rule u in Π_{main} such that u should fire at S. These slave agents will compute the recursively defined values needed to fire Π_{main} at S.

B. Wait, and then execute Π_{main} (reactivation): Wait until all slave agents finish their work, and then execute one step of Π_{main} with the results of the slaves substituted for the corresponding recursive calls.

A slave agent a starts executing the module $\mathit{Mod}(a)$ right after its creation. Notice that a slave agent may or may not halt. If at least one slave agent fails to halt, Π "hangs"; it will not complete the current step.

The translation of Π is given in two stages: **I.** we translate Π_{main} into a module *Main* executed by the master agent, and **II.** we translate the body Π_i of every recursive definition in Π_{rec} into a module F_i executed by some slave agents. Thus Π' consists of module *Main* and modules F_i.

I. From Π_{main} to *Main*:

A. Create slave agents: Enumerate all occurrences of subterms $F_i(\overline{s})$, i.e., recursive calls, in Π_{main} arbitrarily. Suppose there are m recursive calls. If the j^{th} recursive call has the form $F_i(s_1, \ldots, s_{k_i})$, define the rule R_j as

$$
\begin{array}{l}
\texttt{if } g_j \texttt{ then} \\
\quad \texttt{extend } \mathit{Agents} \texttt{ with } a \\
\qquad \mathit{Mod}(a) := F_i \\
\qquad \mathit{Arg}_{i1}(a) := s_1 \\
\qquad \qquad \vdots \\
\qquad \mathit{Arg}_{ik_i}(a) := s_{k_i} \\
\qquad \mathit{Mode}(a) := \mathit{Initial}
\end{array}
$$

```
        RecMode(a) := CreatingSlaveAgents
        Child(Me, j) := a
    endextend
endif
```

where the guard g_j is true in a given state S iff the atomic rule with the j^{th} recursive call is enabled in S. We will give an inductive construction of g_j in Proposition 2.4 below. The first part of the module *Main* is the rule

```
if RecMode = CreatingSlaveAgents then
    R_1
    ⋮
    R_m
    RecMode := WaitingThenExecuting
endif
```

where $RecMode = CreatingSlaveAgents$ is assumed to be valid in the initial state of Π'.

B. Wait, and then execute Π_{main}: The second part of *Main* is the rule

```
if RecMode = WaitingThenExecuting and
    and_{j=1}^{m} (Child(Me, j) = undef or Mode(Child(Me, j)) = Final)
then
    Π'_main
    Child(Me, 1) := undef
                      ⋮
    Child(Me, m) := undef
    RecMode := CreatingSlaveAgents
endif
```

where Π'_{main} is obtained from Π_{main} by substituting for $j = 1, \ldots, m$ the j^{th} recursive call with $Return(Child(Me, j))$. Note that $Child(Me, j) = undef$ happens if the j^{th} recursive call produces no slave agent.

II. From Π_i to F_i: The translation of Π_i is similar to that of Π_{main}, except that the following functions in $g_j, s_1, \ldots, s_{k_i}$ (the guard and the argument terms in phase A) and in Π'_i (the main part of phase B) now are local, i.e., get the additional initial argument *Me*:

- *Mode*
- *RecMode*
- every dynamic function (with respect to Π_i).

This modification ensures that every slave agent uses its private dynamic functions only and thus avoids any side-effects. Call the resulting module F_i.

It remains to exhibit the guards $g_1, \ldots, g_m$. For the time being, let $R(\overline{x})$ denote a rule R with free variables in $\overline{x}$, and consider free variables as nullary function symbols. Thus, the vocabulary of $R(\overline{x})$ includes some of the variables in $\overline{x}$.

Proposition 2.4 *Let $R(\overline{x})$ be a rule and o an occurrence of an atomic rule in $R(\overline{x})$ but not inside any* `import`, `choose` *or* `vary` *rule. There is a guard $g(\overline{x})$ (constructed in the proof) such that for every state S of $R(\overline{x})$ the following are equivalent:*

1. *o is enabled in S.*
2. *$S \models g(\overline{x})$.*

PROOF. Induction on the construction of $R(\overline{x})$: The cases where $R(\overline{x})$ is atomic or a block (sequence of rules) are straightforward. Assume $R(\overline{x})$ = `if` $g_0(\overline{x})$ `then` $R'(\overline{x})$ `endif`, where o occurs in $R'(\overline{x})$. (An `if-then-else` rule can easily be replaced by a block of two `if-then` rules; as guards choose the original guard and its negation.) By induction hypothesis there is a $g'(\overline{x})$ satisfying the equivalence with respect to $R'(\overline{x})$. Thus let $g(\overline{x}) = g_0(\overline{x})$ `and` $g'(\overline{x})$. □

To obtain the guards $g_1, \ldots, g_m$, distinguish two cases: 1. Suppose Π_{main} is a rule where no recursive call occurs inside an `import` or `choose` rule. (Recall our general assumption in this subsection that no recursive call occurs inside a `vary` rule either.) Since Π_{main} has no free variables, Proposition 2.4 gives us the desired closed guard g_j, if we choose o to be the atomic rule with the j^{th} recursive call.

2. Suppose Π_{main} has some recursive calls inside some `import` or `choose` rules. We can assume that Π_{main} has the form

```
import x_1, ..., x_p
    choose y_1 ∈ U_1, ..., y_q ∈ U_q
        Π*_main(x_1, ..., x_p, y_1, ..., y_q)
    endchoose
endimport
```

where every remaining `import` and `choose` in $\Pi^*_{\text{main}}(\overline{x}, \overline{y})$ occurs in a `vary` rule, and the variables in $\overline{x}, \overline{y}$ are disjoint and do not occur bounded in $\Pi^*_{\text{main}}(\overline{x}, \overline{y})$. (This special form can be obtained, e.g., by the First Normal Form procedure of [Dexter, Doyle, Gurevich 97] extended by a second round to push out `choose` rules in the same way as `import` rules in the first round. Here `vary` rules are considered as black boxes whose interior is ignored.) In this case Proposition 2.4 yields for the j^{th} recursive call a guard $g_j(\overline{x}, \overline{y})$ satisfying the equivalence with respect to $\Pi^*_{\text{main}}(\overline{x}, \overline{y})$. With these guards we translate $\Pi^*_{\text{main}}(\overline{x}, \overline{y})$ into the module *Main** consisting of two rules, say, $R^*_A(\overline{x}, \overline{y})$ for phase A and $R^*_B(\overline{x}, \overline{y})$ for phase B. As the actual module *Main* we then take

```
import x̄
    choose ȳ ∈ Ū
        R**_A(x̄, ȳ)
        R*_B(x̄', ȳ')
    endchoose
endimport
```

where $\overline{x}', \overline{y}'$ are new nullary function symbols and the rule $R^{**}_A(\overline{x}, \overline{y})$ is identical to $R^*_A(\overline{x}, \overline{y})$ except that after the basic rule *RecMode* := *WaitingThenExecuting* we add basic rules $\overline{x}' := \overline{x}$ and $\overline{y}' := \overline{y}$. ($\overline{x}', \overline{y}'$ become local in case of translating Π_i, i.e., are unary function symbols with argument *Me*.)

Remark 2.5 It is not literally true that slave agents cause no side effects. For example, a slave may leave imported elements. However, these elements will be inaccessible later on. To all practical purposes, there will be no side effects. □

Example 2.6 (Translation of ListMax). A translation of Π in Example 2.2 is:

```
Main :
   if RecMode = CreatingSlaveAgents then
      extend Agents with a
         a.Mod := ListMax
         a.List := L
         a.Mode := Initial
         a.RecMode := CreatingSlaveAgents
         Child(Me, 1) := a
      endextend
      RecMode := WaitingThenExecuting
   endif

   if RecMode = WaitingThenExecuting and
      (Child(Me, 1) = undef or Child(Me, 1).Mode = Final)
   then
      if Mode = Initial then
         Output := Child(Me, 1).Return
         Mode := Final
      endif
      Child(Me, 1) := undef
      RecMode := CreatingSlaveAgents
   endif

ListMax :
   if Me.RecMode = CreatingSlaveAgents then
      if Me.List.Length ≠ 1 then
         extend Agents with a, b
            a.Mod := ListMax
            a.List := Me.List.FirstHalf
            a.Mode := Initial
            a.RecMode := CreatingSlaveAgents
            Child(Me, 1) := a
            b.Mod := ListMax
            b.List := Me.List.SecondHalf
            b.Mode := Initial
            b.RecMode := CreatingSlaveAgents
            Child(Me, 2) := b
         endextend
      endif
      Me.RecMode := WaitingThenExecuting
   endif

   if Me.RecMode = WaitingThenExecuting and
      (Child(Me, 1) = undef or Child(Me, 1).Mode = Final) and
```

```
    (Child(Me,2) = undef or Child(Me,2).Mode = Final)
then
    if Me.List.Length = 1 then
        Me.Return := Me.List.Head
    else
        Me.Return := Max(Child(Me,1).Return, Child(Me,2).Return)
    endif
    Me.Mode := Final
    Child(Me,1) := undef
    Child(Me,2) := undef
    Me.RecMode := CreatingSlaveAgents
endif
```
□

Note that in this section we used the powerful tool of distributed ASMs to model a restricted form of recursion. All agents created live in their own worlds, not sharing any memory or competing for any resource, e.g., updating a common location. As a result every run of Π', whether interleaved or truly concurrent, produces the same result. In general a sequential execution, in which one agent starts working after another finishes, will be more space efficient than a parallel one.

In the next section we will relax our restriction that all functions in a recursive definition are local. Specially designated global functions may be shared by the master and some slave agents, and be updated by all of them. Consequently the semantics of recursive programs becomes non-deterministic.

3 Concurrent Recursion with Interference

There are problems which naturally admit a recursive solution, but also involve concurrency and competition. It makes sense to allow slave agents to vie with one another for globally accessible functions, so that they may get in each other's way.

Example 3.1 (Parallel ListMax with bounded number of processors). Recall our simple divide and conquer example *ListMax* (Example 2.2). If we consider the job of every agent as a task executable on a multi-processor system, the number of processors depends on the length of *List*. Now, if we lower the level of abstraction and take into account that a multi-processor system only has, say, 42 processors, the following recursive program describes the new view. (In the modified recursive definition of *ListMax* the key word `global` declares the nullary function *Processors* to be shared by all agents. Furthermore, assume that *Processors* equals 42 in the initial state.)

```
if Mode = Initial then
    Output := L.ListMax
    Mode := Final
endif

rec ListMax(List : list) : int
global Processors : int
     if List.Length = 1 then
```

```
        Return := List.Head
        Mode := Final
    elseif Processors ≥ 1 then
        Processors := Processors − 1
        Mode := Parallel
    else
        FirstHalfMax := List.FirstHalf.ListMax
        Mode := Sequential
    endif

    if Mode = Parallel then
        Return := Max(List.FirstHalf.ListMax, List.LastHalf.ListMax)
        Processors := Processors + 1
        Mode := Final
    endif

    if Mode = Sequential then
        Return := Max(FirstHalfMax, List.LastHalf.ListMax)
        Mode := Final
    endif
endrec
```
□

A generalization of recursive programs in [Section 2] to recursive programs with global functions is easy. Alter the second point in Definition 2.1 as follows:

2. a sequence Π_{rec} of *recursive definitions* of the form

   ```
   rec F_i(Arg_i1, ..., Arg_ik_i)
   global f_i1, ..., f_il_i
       Π_i
   endrec
   ```

   Here f_{ij} is an arbitrary function symbol in Π which does not have *Me* as its first argument, ... and the rest is as before.

The functions $f_{i1}, \ldots, f_{il_i}$ are intended to be *global* in Π_i in the sense that the interpretation of the symbols $f_{i1}, \ldots, f_{il_i}$ in Π_i is identical to that in Π_{main}. A slight modification of our translation into distributed programs reflects the new situation:

II. From Π_i to F_i: The translation of Π_i is similar to that of Π_{main}, except that the following functions in $g_j, s_1, \ldots, s_{k_i}$ and in Π'_i, which are different from any $f_{i1}, \ldots, f_{il_i}$, now are local, ... and the rest is as before.

Note that even if a global function f is static in Π_i, f is still not local, as there may be other agents which update f. We do not worry about the distinction between global and local functions when f is static with respect to $\Pi = (\Pi_{\mathrm{main}}, \Pi_{\mathrm{rec}})$. Another example, which is purely recursive and also enjoys competition, is the task of finding the shortest path between two nodes in an infinite graph.

Example 3.2 (Shortest-Path). Consider the following discrete optimization problem: Given an infinite connected graph (e.g., the computation tree of a

PROLOG program) and nodes *Start* and *Goal*, find a shortest path from *Start* to *Goal*. Of course, an imperative program implementing breadth-first search or iterative deepening will find a shortest path, but let us sketch a parallel solution.

For simplicity assume that each node *Node* has exactly four neighbors, namely *Node.North*, *Node.East*, *Node.South* and *Node.West*. The idea is to call a slave agent with some *Node* and the cost of *Node*, that is, the length of the path from *Start* to *Node*. The slave agent checks whether the cost is still less than the length of the current best solution found by some competing slave agent. If so, it searches recursively in all four directions, until a better solution is found. The cost of this solution then is made public by storing it into a global nullary function *BestSolution*. Otherwise, the slave agent rejects *Node*. For brevity, we do not incorporate a mechanism (for instance a *ClosedNodesList*) preventing agents from examining nodes several times. The algorithm can be formalized as a recursive program with the global function *BestSolution*, which is assumed to be initialized with ∞:

```
if Mode = Initial then
    OutputPath := ShortestPath(Start, 0)
    OutputCost := BestSolution
    Mode := Final
endif

rec ShortestPath(Node : node, Cost : int) : path
global BestSolution : int
    if Mode = Initial and BestSolution ≤ Cost then
        Return := dump
        Mode := Final
    endif

    if Mode = Initial and BestSolution > Cost then
        if Node = Goal then
            BestSolution := Cost
            Return := nil
            Mode := Final
        else
            North.Child := ShortestPath(Node.North, Cost + 1)
            East.Child := ShortestPath(Node.East, Cost + 1)
            South.Child := ShortestPath(Node.South, Cost + 1)
            West.Child := ShortestPath(Node.West, Cost + 1)
            Mode := SelectBestChild
        endif
    endif

    if Mode = SelectBestChild then
        if ∃x ∈ Direction : x.Child.Length+ Cost+1 = BestSolution then
            choose x ∈ Direction
            satisfying x.Child.Length + Cost + 1 = BestSolution
                Return := Cons(Node, x.Child)
            endchoose
        else
```

```
            Return := dump
        endif
        Mode := Final
    endif
endrec
```
□

There are many recursive problems which suggest a sequential execution—and thus do not need concurrency or competition—but which naturally gain from the use of global functions, e.g., global output channels. This kind of sequential recursion using global functions is the topic of the subsequent section.

4 Sequential Recursion

Consider a recursive program with global functions where it is guaranteed (by the programmer) that at each state of the computation at most one recursive call takes place. In other words, at each state, at most one of the existing slave agents a is *working* (i.e., neither a's mode is *Final* nor a is waiting for one of its slave agents to finish). In this case a deterministic, sequential evaluation is ensured. Only one agent works, whereas all other agents wait in a hierarchical dependency.

Example 4.1 (The Towers of Hanoi). The well-known Towers of Hanoi problem [Lucas 96] is purely sequential: our task is to instruct the player how to move a pile of disks of decreasing size from one peg to another using at most 3 pegs in such a way that at no point a larger disk rests on a smaller one. The player can only move the top disk of one pile to another in a single step. The following recursive program solves the Towers of Hanoi problem. We use the global function *Output* to pass instructions to the player. (The nullary function *Dummy* eventually gets the value *undef*; its only purpose is to call the recursively defined function *Towers*.)

```
if Mode = Initial then
    Dummy := Towers(Place1, Place2, Place3, PileHeight)
    Mode := Final
endif

rec Towers(From, To, Use : place, High : int)
global Output : instructions
    if Mode = Initial then
        if High = 1 then
            Output := MoveTopDisk(From, To)
            Mode := Final
        else
            Dummy := Towers(From, Use, To, High − 1)
            Mode := MoveBottomDisk
        endif
    endif

    if Mode = MoveBottomDisk then
        Output := MoveTopDisk(From, To)
```

```
        Mode := MovePileBack
    endif

    if Mode = MovePileBack then
        Dummy := Towers(Use, To, From, High − 1)
        Mode := Final
    endif
endrec
```
□

Because of the sequential character of execution, one can avoid having slave agents change global functions: a recursive call can return a list of would-be changes, such that the master can itself perform all the changes. For instance, instead of outputting instructions in the last example, we compute a list of instructions, and pass it to the player. Unfortunately the length of the list would be exponential in the number of disks involved. Thus it makes sense to use a global output channel.

The semantic property of sequentiality can easily be guaranteed by syntactic restrictions on a recursive program Π. For example, require in Definition 2.1 that Π_{main} and each Π_i is a block of rules

if $Mode = Mode_j$ then R_j endif

where the static nullary functions $Mode_j$ have distinct values and each R_j contains at most one recursive call.

As examples with this restricted syntax we refer to the Tower of Hanoi program above and Savitch's Reachability algorithm (Example 2.3).

References

[Börger 95] E. Börger. Annotated bibliography on evolving algebras. In E. Börger, editor, *Specification and Validation Methods*, pages 37–51. Oxford University Press, 1995.

[Castillo 96] G. D. Castillo. WWW page *Evolving Algebras Europe*, http://www.uni-paderborn.de/Informatik/eas.html, May 96.

[Dexter, Doyle, Gurevich 97] S. Dexter, P. Doyle, and Y. Gurevich. Gurevich abstract state machines and schönhage storage modification machines. Technical Report CSE-TR-326-97, University of Michigan, 1997. Also in this volume.

[Gurevich 91] Y. Gurevich. Evolving Algebras: An attempt to discover semantics. *Bulletin of the EATCS*, 43:264–284, 1991. A slightly revised version in G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 266–292, World Scientific, 1993.

[Gurevich 95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[Huggins 96] J. K. Huggins. WWW page *Abstract State Machines (Evolving Algebras)*, http://www.eecs.umich.edu/ealgebras, September 1996.

[Lucas 96] E. Lucas. *Recreations mathematiques*, volume 3, pages 55–59. Gauthier-Villars et fils, Paris, 1891–1896. Reprinted by A. Blanchard, Paris, 1960.

[Savitch 70] W. J. Savitch. Relational between nondeterministic and deterministic tape complexity. *Journal of Computer and System Sciences*, 4:177–192, 1970.

Acknowledgements. Thanks to the anonymous referees.