The hidden computation steps of Turbo Abstract State Machines

Nicu G. Fruja and Robert F. Stärk

Computer Science Department, ETH Zürich, CH–8092 Zürich, Switzerland
fruja@inf.ethz.ch and staerk@inf.ethz.ch

Abstract. Turbo Abstract State Machines are ASMs with parallel and sequential composition and possibly recursive submachine calls. Turbo ASMs are viewed as black-boxes that can combine arbitrary many steps of one or more submachines into one big step. The intermediate steps of a turbo ASM are not observable from outside. It is not even clear what exactly the intermediate steps are, because the semantics of turbo ASMs is usually defined inductively along the call graph of the ASM and the structure of the rule bodies. The most important application of turbo ASMs are recursive algorithms. Such algorithms can directly be simulated on turbo ASMs without transforming them into multi-agent (distributed) ASMs. In this article we analyze the hidden intermediate steps of turbo ASMs and characterize them using PAR/SEQ trees. We also address the problem of the reserve in the presence of recursion and sequential composition.

1 Introduction

Börger and Schmid view in [7] the execution of a sequential rule of an ASM or a recursive submachine call as one atomic step that produces exactly one set of parallel updates. Their black-box view of subcomputations makes it possible to combine arbitrary many computation steps of a submachine into one big step. The intermediate states of a submachine call are hidden and inaccessible from outside. Since recursive ASMs allow arbitrary speed-ups in the black-box view, we call them turbo ASMs in this article.

The black-box view of sequential composition and recursive submachine allows a direct simulation of recursive algorithms. Gurevich's ASM thesis, however, is violated, since for example a turbo ASM for the MergeSort algorithm can sort a list in *one* step. Hence the obvious question is: what are the hidden intermediate steps between two main steps $\mathfrak{A}$ and $\mathfrak{B}$ of a turbo ASM?

$$\mathfrak{A} \to \underbrace{\sigma_0 \to \sigma_1 \to \cdots \to \sigma_n}_{?} \to \mathfrak{B}$$

The first idea is to add a stack to ASMs as it is done in implementations of imperative programming languages. This works well as long as recursive calls

Technical Reports 416, ETH Zürich, Theoretical Computer Science. Extended version of [9].

occur sequentially. If many recursive calls are done in parallel, the structure of the stack becomes more complicated. It is no longer a linear list, but a PAR/SEQ tree with nodes labeled by PAR (parallel composition) and SEQ (sequential composition).

The main technical contribution of this article is the analysis of PAR/SEQ trees. A parallel node in a PAR/SEQ tree has to wait until all its children are fully evaluated. Then the union of the update sets of all the children is added to the already computed updates for the node. A sequential node has a transition rule and exactly one child. The node has to wait until its child is fully evaluated. The transition rule of the node is evaluated in a local state that is obtained by firing the updates of the child. The leaves of a PAR/SEQ tree are update sets, sequential compositions or rule calls. When a sequential composition in a leaf of a PAR/SEQ tree is evaluated, a new sequential node is created. A recursive call in a leaf of PAR/SEQ tree is simply unfolded. A single computation step of a turbo ASM consists of applying all the operations in parallel to all the leaves of the PAR/SEQ tree.

The local states in a PAR/SEQ tree can all be different. Also the environments that assign values to the free variables of transition rules can be different in different nodes. Hence, the approach of Bolognesi and Börger in [5] for the intermediate states of *Abstract State Processes* (ASPs) cannot be applied. Abstract State Processes extend ASM programs by process-algebraic behavior expressions for non-deterministic and concurrent processes. The configurations of an ASP are pairs consisting of an algebraic structure and a program. In a single computation step the ASP produces a set of updates and a new program which is called *residual* program. The update set is then applied to the structure and the residual program is executed in the new state. The intermediate steps of a sequential execution or a submachine call (process instantiation) can be traced and observed. The white-box operator for the sequential composition, however, is quite different from the black-box operator **seq** that we consider here. If two sequential processes run in parallel, then in the white-box view they can interfere, whereas in the black-box view they run completely independent. In the white-box view, the processes do not have a local state. After each step, the local changes become visible in the global state.

The plan of this article is as follows. In Sect. 2 we review the MergeSort example as a motivation for studying turbo ASMs. We then summarize in Sect. 3 syntax and semantics of turbo ASMs. In Sect. 4 we first consider the case of recursive ASMs without **seq**. In this case, the intermediate steps are sets consisting of updates and rule calls. In each step the remaining rule calls are unfolded and yield new updates and more rule calls. If the subset of rule calls becomes empty, the updates can be applied to the initial state and the result is the next main state. In Sect. 5 we consider the case of recursive ASMs with **seq** but without **par** and **forall** (= recursive while-programs). In this case, the hidden intermediate steps are given by a stack of frames. If the stack becomes empty, then the accumulated update set is applied to the initial state. In Sect. 6 we consider the general case of recursive ASMs with arbitrary nesting of **par**, **seq** and **forall**. In

the general case, the intermediate states are trees with PAR and SEQ nodes. In each case we prove the equivalence of the inductive big-step semantics of turbo ASMs with the small-step semantics.

In Sect. 7, we address the problem of the reserve in the context of recursion and sequential composition. We show how the inductive definition of the semantics of turbo ASMs can easily be extended to turbo ASMs with a reserve by simply adding additional constraints. Once the reserve is available, turbo ASMs with return values are obtained simply by syntactic sugar. Finally, in Sect. 8 we compare turbo ASMs with other approaches to recursion for ASMs.

2 The MergeSort example

Moschovakis argues in [14] that algorithms are *recursive equations* while abstract machine models are just *implementations*, a special kind of algorithms. This contradicts in some sense Gurevich's ASM thesis [3,12] that each algorithm can be simulated on its natural level of abstraction step-by-step by an ASM. As an example for the insufficiency of machine models, Moschovakis, mentions the MergeSort algorithm.

Consider the problem of sorting a finite list $x = [x_1, \ldots, x_n]$ with respect to a linear ordering $\leq$. The MergeSort algorithm can succinctly be defined by the recursive equation

$$\mathtt{msort}(x) = \begin{cases} x, & \text{if } |x| \leq 1; \\ \mathtt{merge}(\mathtt{msort}(\mathtt{l}(x)), \mathtt{msort}(\mathtt{r}(x))), & \text{otherwise.} \end{cases} \tag{1}$$

where $|x|$ denotes the length of the list x and $\mathtt{l}(x)$, $\mathtt{r}(x)$ are the first and second halves of the list x. The function $\mathtt{merge}(x, y)$ is also defined recursively by the equation

$$\mathtt{merge}(x, y) = \begin{cases} y, & \text{if } |x| = 0; \\ x, & \text{else, if } |y| = 0; \\ \mathtt{hd}(x) \cdot \mathtt{merge}(\mathtt{tl}(x), y), & \text{else, if } \mathtt{hd}(x) \leq \mathtt{hd}(y); \\ \mathtt{hd}(y) \cdot \mathtt{merge}(x, \mathtt{tl}(y)), & \text{otherwise.} \end{cases} \tag{2}$$

Here $a \cdot x$ denotes the result of prepending the element a to the list x (the `cons` of Lisp); $\mathtt{hd}(x)$ is the first element of the list x (the `car` of Lisp) and $\mathtt{tl}(x)$ is the rest of the list (the `cdr` of Lisp).

Moschovakis asks the question: 'If algorithms are machines, then which machine is the MergeSort?' — Gurevich's answer to the question in [4,13] is that recursion is implicitly distributed (multi-agent) computation. To model the recursive MergeSort algorithm one has to translate the equations (1) and (2) into a distributed ASM. The algorithm is then given by the (partially ordered) runs of the distributed ASM.

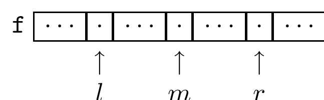
Most people, however, do not think on distributed computation when they see equations (1) and (2). Computer Science students, for example, learn that

the equations have a *declarative* (denotational) and a *procedural* (operational) semantics. The declarative semantics is given by the least fixed-point of a continuous functional associated to the equations. The operational semantics is obtained by viewing the equations as term rewrite rules and applying certain reduction strategies (call-by-value, call-by-name, leftmost-innermost, parallel-innermost, etc.). Moreover, in courses about algorithms, the data to be sorted are not elements of an immutable list but are stored in a dynamic array and are sorted in place.

We show now that the declarative as well as the operational semantics of the equations (1) and (2) can be modeled by ASMs in a natural way without using the notion of distributed ASMs. The ASMs in Fig. 1 and Fig. 2 are alternative answers to Moschovakis' question. We do not claim that they solve the problem, but they show another possible approach to recursive ASMs not mentioned in [4]. Moreover, the turbo ASMs in Fig. 2 are used as examples later.

The idea of the ASM for the declarative semantics in Fig. 1 is that the finite stages of the least fixed-point computation are the states of the ASM. The functions $\mathtt{msort}(x)$ and $\mathtt{merge}(x, y)$ are dynamic functions which are initialized with the value *undef* for all lists x, y. The ASM in Fig. 1 is monotonic in the sense that a location that is defined ($\neq$ *undef*) cannot change its contents in later steps of the run. The ASM is highly parallel. In the first step it computes $\mathtt{msort}(x)$ for all lists x of length 1. In the second step $\mathtt{merge}(x, y)$ is computed for all lists x, y of length 1. In the third step, $\mathtt{msort}(x)$ is computed for all lists of length 2, etc. The ASM runs forever. Hence, the ASM in Fig. 1 is not an implementation ASM but a pure specification ASM. One step in the run of the ASM corresponds to one step in the least fixed-point computation. (Note that the ASM thesis holds even in this rather abstract case.)

In Fig. 2 a turbo ASM is used for an implementation of the MergeSort algorithm. The ASM uses a unary dynamic function $\mathtt{f}$ that represents an array. The parameters l and r in $\textsc{Msort}(l, r)$ are the left and the right bound of the subarray $\mathtt{f}(l\,..\,r)$ that has to be sorted.



The middle point m between l and r is computed and the subarrays $\mathtt{f}(l\,..\,m)$ and $\mathtt{f}(m+1\,..\,r)$ are recursively sorted in parallel by the calls $\textsc{Msort}(l, m)$ and $\textsc{Msort}(m+1, r)$. After that, the two sorted halves are merged together using $\textsc{Merge}(l, m, r)$. Thereby a second unary function $\mathtt{g}$ is used in which a copy of $\mathtt{f}$ is stored before the merging starts. $\textsc{MergeCopy}(i, m, j, r, k)$ merges the subarrays $\mathtt{g}(i\,..\,m)$ and $\mathtt{g}(j\,..\,r)$ into $\mathtt{f}(k\,..\,r)$. The parameter i runs over the first half of $\mathtt{g}$ from l to m; the parameter j runs over the second half of $\mathtt{g}$ from $m+1$ to r and the parameter k runs over the whole array $\mathtt{f}$ from l to r.

Note that in the specification ASM in Fig. 1 the symbols $\mathtt{msort}$ and $\mathtt{merge}$ are names of dynamic functions, whereas in the implementation ASM in Fig. 2 the symbols $\textsc{Msort}$ and $\textsc{Merge}$ are names of (recursive) transition rules. In the

$\textsc{DeclarativeMergeSort} \triangleq$
forall $x \in \mathit{List}$ **do**
  if $|x| \leq 1$ **then**
    $\texttt{msort}(x) := x$
  else
    $\texttt{msort}(x) := \texttt{merge}(\texttt{msort}(\texttt{l}(x)), \texttt{msort}(\texttt{r}(x)))$

$\textsc{DeclarativeMerge} \triangleq$
forall $x, y \in \mathit{List}$ **do**
  if $|x| = 0$ **then**
    $\texttt{merge}(x, y) := y$
  else if $|y| = 0$ **then**
    $\texttt{merge}(x, y) := x$
  else if $\texttt{hd}(x) \leq \texttt{hd}(y)$ **then**
    $\texttt{merge}(x, y) := \texttt{hd}(x) \cdot \texttt{merge}(\texttt{tl}(x), y)$
  else
    $\texttt{merge}(x, y) := \texttt{hd}(y) \cdot \texttt{merge}(x, \texttt{tl}(y))$

Fig. 1. The declarative semantics of MergeSort as a parallel ASM.

$\textsc{Msort}(l, r) \triangleq$
if $l < r$ **then**
  let $m = \lfloor (l + r)/2 \rfloor$ **in**
    $(\textsc{Msort}(l, m)$ **par** $\textsc{Msort}(m + 1, r))$ **seq** $\textsc{Merge}(l, m, r)$

$\textsc{Merge}(l, m, r) \triangleq$
(**forall** i **with** $l \leq i \leq r$ **do** $\texttt{g}(i) := \texttt{f}(i)$) **seq** $\textsc{MergeCopy}(l, m, m + 1, r, l)$

$\textsc{MergeCopy}(i, m, j, r, k) \triangleq$
if $k \leq r$ **then**
  if $(i \leq m \wedge j \leq r \wedge \texttt{g}(i) \leq \texttt{g}(j)) \vee (r < j)$ **then**
    $\texttt{f}(k) := \texttt{g}(i)$ **par** $\textsc{MergeCopy}(i + 1, m, j, r, k + 1)$
  else
    $\texttt{f}(k) := \texttt{g}(j)$ **par** $\textsc{MergeCopy}(i, m, j + 1, r, k + 1)$

Fig. 2. An implementation of MergeSort as a turbo ASM.

specification ASM in Fig. 1, list operations are static functions. In the implementation ASM in Fig. 2, lists are represented by unary dynamic functions.

In the black-box view of turbo ASMs (as defined in [7]) the $\textsc{Msort}(l, r)$ in Fig. 2 sorts the subarray $\texttt{f}(l\,..\,r)$ in one step. This fact is rather surprising and the question arises what the intermediate steps of turbo ASMs exactly are. We try to answer this question in this article.

3 The semantics of recursive ASMs

Given a vocabulary Σ for predicate logic, we denote the terms of Σ by s, t and the first-order formulas of Σ by φ, ψ. The function names in Σ are declared either as static or dynamic. A turbo ASM consists of a finite set of rule declarations. A rule declaration for a rule name r is an expression $r(x) \triangleq P$, where P is a transition rule in which there are no free occurrences of variables except of x. Rule declarations can be recursive, i.e., the rule name r may appear in the body P like, for example, the rule name $\textsc{Msort}$ in Fig. 2. The transition rules P, Q are syntactic expressions generated as follows (the function arguments can be read as vectors):

1. *Skip Rule:* **skip**
   Meaning: Do nothing.
2. *Update Rule:* $f(s) := t$
   Syntactic condition: f is a dynamic function name of Σ
   Meaning: In the next state, the value of f at s is updated to t.
3. *Block Rule:* P **par** Q
   Meaning: P and Q are executed in parallel.
4. *Conditional Rule:* **if** φ **then** P **else** Q
   Meaning: If φ is true, then execute P, otherwise execute Q.
5. *Let Rule:* **let** $x = t$ **in** P
   Meaning: Assign the value of t to x and execute P.
6. *Forall Rule:* **forall** x **with** φ **do** P
   Meaning: Execute P in parallel for each x satisfying φ.
7. *Choose Rule:* **choose** x **with** φ **do** P
   Meaning: Choose an x satisfying φ and execute P.
8. *Sequence Rule:* P **seq** Q
   Meaning: P and Q are executed sequentially, first P and then Q.
9. *Call Rule:* $r(t)$
   Meaning: Call transition rule r with parameters t.

The variables x in **let**, **forall** and **choose** are logical variables (also called *read only* variables). Its values cannot be updated by a transition rule. The values of the logical variables are not stored in the state but in a finite environment (also called *variable assignment*). The scope of x in **let** is the rule P (but not the term t), whereas the scope of x in **forall** or **choose** is φ and P.

The possible states of an ASM are the first-order structures for the vocabulary Σ. The semantics of transition rules is inductively defined in Table 1 by

a predicate yields$(P, \mathfrak{A}, \zeta, U)$ with the meaning 'the transition rule P yields in state $\mathfrak{A}$ under the variables assignment ζ the update set U.' In the presence of the **choose** rule, a transition rule can yield several different update sets.

The following notations are used in Table 1: The value of a term t in a structure $\mathfrak{A}$ under a variable assignment ζ is denoted by $[\![t]\!]^{\mathfrak{A}}_{\zeta}$; the truth value of a formula φ by $[\![\varphi]\!]^{\mathfrak{A}}_{\zeta}$. The range of a formula φ with distinguished variable x in a state $\mathfrak{A}$ and environment ζ is the set of all elements of $\mathfrak{A}$ that make the formula true:

$$\text{range}(x, \varphi, \mathfrak{A}, \zeta) = \{a \in |\mathfrak{A}| : [\![\varphi]\!]^{\mathfrak{A}}_{\zeta[x \mapsto a]} = \mathit{true}\}$$

By $\zeta[x \mapsto a]$ we denote the environment that is like ζ except for the variable x to which it assigns the element a.

A location of a state $\mathfrak{A}$ is a pair $\langle f, a \rangle$, where f is a function name of Σ and a is a list of elements of $\mathfrak{A}$. The content $\mathfrak{A}(l)$ of the location l in $\mathfrak{A}$ is the value of $f(a)$ in $\mathfrak{A}$. An update for $\mathfrak{A}$ is a pair $\langle l, v \rangle$, where l is a location and v (the new content of the location) is an element of $\mathfrak{A}$. The meaning of the update is that the function f is updated to the value v at the argument a.

An update set U is a set of updates. It is consistent, if it does not contain two updates for the same location with different values. If an update set U is consistent, then its updates can be applied to a state $\mathfrak{A}$. The result is a new state $\mathfrak{A} + U$ where the locations which are not updated in U keep their old contents.

The composition $U \oplus V$ of two update sets U and V is the set of updates obtained from U by adding the updates of V and overwriting updates in U which are redefined in V. If U and V are consistent, then $U \oplus V$ is consistent, too. The composition is defined such that the following equation is true for any state $\mathfrak{A}$:

$$\mathfrak{A} + (U \oplus V) = (\mathfrak{A} + U) + V$$

The equation says that applying the update set $U \oplus V$ to state $\mathfrak{A}$ is the same as first applying U and then V.

How can it be that $\textsc{Msort}(l, r)$ in Fig. 2 sorts the subarray $\mathtt{f}(l \mathinner{.\,.} r)$ in one step? — We have to look at the rules for the sequential composition and at the rules for calls in Table 1. A sequential composition P **seq** Q is evaluated in state $\mathfrak{A}$ as follows. First, the rule P is evaluated in state $\mathfrak{A}$ and yields an update set U. If U is consistent, it is applied to $\mathfrak{A}$. The rule Q is then evaluated in the new state $\mathfrak{A} + U$ and yields an update set V. The result of the execution of P **seq** Q is the combined update set $U \oplus V$. After the completion of P **seq** Q the intermediate state $\mathfrak{A} + U$ is discarded. It is not visible any more.

A call $r(t)$ of a transition rule r with declaration $r(x) \triangleq P$ is evaluated by substituting the parameter t for x in the body P. The resulting transition rule $P\frac{t}{x}$ is evaluated in the same state. If it is defined and yields an update set, then the update set is the result of the call. A call to a recursive rule that uses sequential compositions can generate an arbitrary number of intermediate states.

The substitution (call-by-name) semantics of rule calls has the advantage that it can easily be generalized to rule schemes that contain variables for transition rules which can be instantiated at run-time. Rule schemes have been used, for

$$\frac{}{\text{yields}(\textbf{skip}, \mathfrak{A}, \zeta, \emptyset)}$$

$$\frac{}{\text{yields}(f(s) := t, \mathfrak{A}, \zeta, \{\langle l, v\rangle\})} \qquad \text{where } l = \langle f, a\rangle,\ a = [\![s]\!]_\zeta^\mathfrak{A},\ v = [\![t]\!]_\zeta^\mathfrak{A}$$

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U) \quad \text{yields}(Q, \mathfrak{A}, \zeta, V)}{\text{yields}(P \textbf{ par } Q, \mathfrak{A}, \zeta, U \cup V)}$$

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U)}{\text{yields}(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q, \mathfrak{A}, \zeta, U)} \qquad \text{if } [\![\varphi]\!]_\zeta^\mathfrak{A} = \mathit{true}$$

$$\frac{\text{yields}(Q, \mathfrak{A}, \zeta, V)}{\text{yields}(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q, \mathfrak{A}, \zeta, V)} \qquad \text{if } [\![\varphi]\!]_\zeta^\mathfrak{A} = \mathit{false}$$

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\text{yields}(\textbf{let } x = t \textbf{ in } P, \mathfrak{A}, \zeta, U)} \qquad \text{where } a = [\![t]\!]_\zeta^\mathfrak{A}$$

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U_a) \quad \text{for each } a \in I}{\text{yields}(\textbf{forall } x \textbf{ with } \varphi \textbf{ do } P, \mathfrak{A}, \zeta, \bigcup_{a \in I} U_a)} \qquad \text{where } I = \text{range}(x, \varphi, \mathfrak{A}, \zeta)$$

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\text{yields}(\textbf{choose } x \textbf{ with } \varphi \textbf{ do } P, \mathfrak{A}, \zeta, U)} \qquad \text{if } a \in \text{range}(x, \varphi, \mathfrak{A}, \zeta)$$

$$\frac{}{\text{yields}(\textbf{choose } x \textbf{ with } \varphi \textbf{ do } P, \mathfrak{A}, \zeta, \emptyset)} \qquad \text{if range}(x, \varphi, \mathfrak{A}, \zeta) = \emptyset$$

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U) \quad \text{yields}(Q, \mathfrak{A} + U, \zeta, V)}{\text{yields}(P \textbf{ seq } Q, \mathfrak{A}, \zeta, U \oplus V)} \qquad \text{if } U \text{ is consistent}$$

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta, U)}{\text{yields}(P \textbf{ seq } Q, \mathfrak{A}, \zeta, U)} \qquad \text{if } U \text{ is inconsistent}$$

$$\frac{\text{yields}(P\frac{t}{x}, \mathfrak{A}, \zeta, U)}{\text{yields}(r(t), \mathfrak{A}, \zeta, U)} \qquad \text{where } r(x) \triangleq P \text{ is a rule declaration}$$

Table 1. Inductive definition of the semantics of turbo ASM rules.

example, in [17] for the uniform specification of different versions of the Java Virtual Machine.

4 ASMs with parallel composition

In this section we consider the special case of recursive ASMs without sequential composition (**seq**). For simplicity we omit also **choose**. The ASM MergeCopy in Fig. 2 is an example of a recursive ASM without sequential composition.

The idea is that between two main states $\mathfrak{A}$ and $\mathfrak{B}$ of a turbo ASM there is a finite sequence of (hidden) intermediate states:

$$\mathfrak{A} \rightarrow \sigma_0 \rightarrow \sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \mathfrak{B}$$

The intermediate states σ_i are sets consisting of updates and frames. A *frame* is a triple $\langle P, \mathfrak{A}, \zeta\rangle$ consisting of a transition rule P, a turbo ASM, state $\mathfrak{A}$, and

$$
\begin{aligned}
&\mathrm{ev}(\mathbf{skip}, \mathfrak{A}, \zeta) = \emptyset\\
&\mathrm{ev}(f(s) := t, \mathfrak{A}, \zeta) = \{\langle\langle f, a\rangle, b\rangle\} \quad \text{where } a = [\![s]\!]_\zeta^\mathfrak{A} \text{ and } b = [\![t]\!]_\zeta^\mathfrak{A}\\
&\mathrm{ev}(P \mathbf{\ par\ } Q, \mathfrak{A}, \zeta) = \mathrm{ev}(P, \mathfrak{A}, \zeta) \cup \mathrm{ev}(Q, \mathfrak{A}, \zeta)\\
&\mathrm{ev}(\mathbf{if}\ \varphi\ \mathbf{then}\ P\ \mathbf{else}\ Q, \mathfrak{A}, \zeta) = \begin{cases} \mathrm{ev}(P, \mathfrak{A}, \zeta), & \text{if } [\![\varphi]\!]_\zeta^\mathfrak{A} = \mathit{true};\\ \mathrm{ev}(Q, \mathfrak{A}, \zeta), & \text{otherwise.}\end{cases}\\
&\mathrm{ev}(\mathbf{let}\ x = t\ \mathbf{in}\ P, \mathfrak{A}, \zeta) = \mathrm{ev}(P, \mathfrak{A}, \zeta[x \mapsto a]) \quad \text{where } a = [\![t]\!]_\zeta^\mathfrak{A}\\
&\mathrm{ev}(\mathbf{forall}\ x\ \mathbf{with}\ \varphi\ \mathbf{do}\ P, \mathfrak{A}, \zeta) = \textstyle\bigcup_{a \in \mathrm{range}(x, \varphi, \mathfrak{A}, \zeta)} \mathrm{ev}(P, \mathfrak{A}, \zeta[x \mapsto a])\\
&\mathrm{ev}(P\ \mathbf{seq}\ Q, \mathfrak{A}, \zeta) = \{\langle P\ \mathbf{seq}\ Q, \mathfrak{A}, \zeta\rangle\}\\
&\mathrm{ev}(r(t), \mathfrak{A}, \zeta) = \{\langle r(t), \mathfrak{A}, \zeta\rangle\}
\end{aligned}
$$

Table 2. Partial evaluation of ASM rules.

an environment ζ. In the absence of sequential compositions, the transition rule of a frame is always a rule call $r(t)$ consisting of a rule name r and a list of argument terms t. The state $\mathfrak{A}$ in the frame is redundant. Only in the general case in Sect. 6 it will be needed.

The first intermediate state σ_0 is the result of the partial evaluation of the main rule of the ASM in the state $\mathfrak{A}$. The result of the partial evaluation of an ASM rule P in a state $\mathfrak{A}$ and an environment ζ is denoted by $\mathrm{ev}(P, \mathfrak{A}, \zeta)$ and defined in Table 2. The partial evaluation leaves rule calls (and as we will see later also sequential compositions) unevaluated. The partial evaluation is recursively defined on the structure of ASM rules.

The next intermediate state σ_{i+1} is obtained by unfolding all frames in σ_i. To unfold a frame containing a rule call means to substitute the argument terms for the formal parameters in the corresponding rule body and to evaluate it in the environment stored in the frame:

$$
\begin{aligned}
&\mathrm{unfold}(\sigma) = \{u \in \sigma \mid u \text{ is an update}\} \cup\\
&\qquad \bigcup \{\mathrm{ev}(P\tfrac{t}{x}, \mathfrak{A}, \zeta) \mid \langle r(t), \mathfrak{A}, \zeta\rangle \in \sigma,\ r(x) \triangleq P \text{ is a rule declaration}\}
\end{aligned}
$$

The last intermediate state must be a consistent set of updates (without frames) which applied to $\mathfrak{A}$ yields the state $\mathfrak{B}$ (so that $\mathfrak{B} = \mathfrak{A} + \sigma_n$).

We will now prove that a turbo ASM (without **seq**) can make one big step from $\mathfrak{A}$ to $\mathfrak{B}$ if, and only if, there exists a finite sequence of unfoldings as described above that lead from $\mathfrak{A}$ to $\mathfrak{B}$.

Lemma 1. *Let P be a rule of an ASM without* **seq**. *If P yields U in $\mathfrak{A}$ under ζ, then there exists a finite sequence of unfoldings from the set* $\mathrm{ev}(P, \mathfrak{A}, \zeta)$ *to U.*

Proof. One shows by induction on the definition of $\mathrm{yields}(P, \mathfrak{A}, \zeta, U)$ that there exists a sequence $(\sigma_i)_{0 \le i \le n}$ such that

1. $\sigma_0 = \mathrm{ev}(P, \mathfrak{A}, \zeta)$,

2. $\sigma_n = U$,
3. $\text{unfold}(\sigma_i) = \sigma_{i+1}$ for each $i < n$.

In the case of **par** and **forall** one can use the following distribuity properties:

$$\text{unfold}(\sigma \cup \tau) = \text{unfold}(\sigma) \cup \text{unfold}(\tau) \quad \text{unfold}(U \cup \sigma) = U \cup \text{unfold}(\sigma)$$

Several parallel sequences of unfoldings can then be combined into a single sequence. □

Lemma 2. *Let P be a rule of an ASM without* **seq**. *If there exists a finite sequence of unfoldings from* $\text{ev}(P, \mathfrak{A}, \zeta)$ *to an update set U, then P yields U in $\mathfrak{A}$ under ζ.*

Proof. In a first step, the following statement is proved by induction on the size of a transition rule P:

If $\text{ev}(P, \mathfrak{A}, \zeta) = U \cup C$ (where U is a set of updates and C is a set of frames) and for each frame $\langle r(t), \mathfrak{A}, \zeta \rangle$ in C there exists an update set $V_{\langle r(t), \mathfrak{A}, \zeta \rangle}$ such that $\text{yields}(r(t), \mathfrak{A}, \zeta, V_{\langle r(t), \mathfrak{A}, \zeta \rangle})$, then $\text{yields}(P, \mathfrak{A}, \zeta, U \cup \bigcup_{i \in C} V_i)$.

In a second step, one proves by induction on n the following statement:

If $(\sigma_i)_{1 \leq i \leq n}$ is a sequence of unfoldings such that

1. $\sigma_0 = \text{ev}(P, \mathfrak{A}, \zeta)$,
2. $\text{unfold}(\sigma_i) = \sigma_{i+1}$ for each $i < n$,
3. $\sigma_n = U_n \cup C_n$ (where U_n is a set of updates and C_n is a set of frames),
4. for each frame $\langle r(t), \mathfrak{A}, \zeta \rangle$ in C_n there exists an update set $V_{\langle r(t), \mathfrak{A}, \zeta \rangle}$ such that $\text{yields}(r(t), \mathfrak{A}, \zeta, V_{\langle r(t), \mathfrak{A}, \zeta \rangle})$

then $\text{yields}(P, \mathfrak{A}, \zeta, U \cup \bigcup_{i \in C_n} V_i)$.

The lemma is then the special case where C_n is empty. □

5 ASMs with sequential composition

In this section we consider the special case of recursive ASMs without bounded parallel composition (**par**) and unbounded parallel composition (**forall**). As we see when we look at the inductive definition of the semantics of ASM rules in Table 1, the set-theoretic union is not involved in computing the update set in that case. Hence, if an ASM does not contain **par** and **forall**, its update set is always consistent. The only operation involved is the sequential composition $U \oplus V$ which is consistent, if U and V are consistent.

Recursive ASMs without **par** and **forall** are more or less classical imperative programs with recursive procedure calls. A while-loop

$$\textbf{while } \varphi(x) \textbf{ do } P(x)$$

can be reduced to the following recursive definition:

$$r(x) \triangleq \textbf{if } \varphi(x) \textbf{ then } (P(x) \textbf{ seq } r(x)) \textbf{ else skip}$$

The equation says that if the condition $\varphi(x)$ is true, then the body $P(x)$ is executed once and after that the whole while-loop is executed again. If the condition $\varphi(x)$ is false, then the while-loop is terminated.

Like in the parallel case, between two main states $\mathfrak{A}$ and $\mathfrak{B}$ of a recursive ASM (without **par** and **forall**) there is a finite sequence of intermediate states:

$$\mathfrak{A} \to \sigma_0 \to \sigma_1 \to \ \cdots \ \to \sigma_n \to \mathfrak{B}$$

An intermediate state σ_i, however, is now a pair $\langle U, S\rangle$ consisting of an update set U (the accumulated set of updates) and a stack S consisting of frames (tasks that have to be done in a sequential order). The stack S is a finite list $[\langle P_1, \zeta_1\rangle, \ldots, \langle P_k, \zeta_k\rangle]$ that grows from left to right (the states in the frames are ignored in this section).

In the state σ_0, the update set is empty and the stack consists of a single frame with the main transition rule of the ASM, the main state $\mathfrak{A}$ plus the environment in which it has to be evaluated, i.e.

$$\sigma_0 = \langle \emptyset, [\langle P_0, \zeta_0\rangle]\rangle.$$

In the last state σ_n, the stack is empty and the state $\mathfrak{B}$ is obtained from $\mathfrak{A}$ by applying the accumulated updates contained in σ_n, i.e.

$$\sigma_n = \langle U_n, []\rangle \quad \text{and} \quad \mathfrak{B} = \mathfrak{A} + U_n.$$

The transition from σ_i to σ_{i+1} is given by the following step function. Assume that in the intermediate state σ the set of updates is U, the topmost frame on the stack is $\langle P, \zeta\rangle$ and the remaining frames on the stack are the list S. Then the next intermediate state is obtained by a partial evaluation of P in the state $\mathfrak{A} + U$ and the environment ζ (see Table 2):

$$\begin{array}{l}
\mathrm{step}(\sigma, \mathfrak{A}) \triangleq \textbf{if } \sigma = \langle U, S \cdot [\langle P, \zeta\rangle]\rangle \textbf{ then} \\
\quad \textbf{case } \mathrm{ev}(P, \mathfrak{A} + U, \zeta) \textbf{ of} \\
\qquad \begin{array}{ll}
\emptyset\text{:} & \textbf{return } \langle U, S\rangle \\
\{\langle\langle f, a\rangle, b\rangle\}\text{:} & \textbf{return } \langle U \oplus \{\langle\langle f, a\rangle, b\rangle\}, S\rangle \\
\{\langle Q \textbf{ seq } R, \eta\rangle\}\text{:} & \textbf{return } \langle U, S \cdot [\langle R, \eta\rangle, \langle Q, \eta\rangle]\rangle \\
\{\langle r(t), \eta\rangle\}\text{:} & \textbf{return } \langle U, S \cdot [\langle P\frac{t}{x}, \eta\rangle]\rangle
\end{array} \\
\quad \textbf{where } r(x) \triangleq P \text{ is a rule declaration}
\end{array}$$

If the result of the partial evaluation of P in state $\mathfrak{A} + U$ is the empty set, then the topmost frame of the stack is deleted. If the result is an update set (consisting of a single update), then the set is added to U and the topmost frame on the stack is deleted. If the result is a frame with a sequential composition, then the topmost frame on the stack is replaced by two frames for the two components (in the right order). If the result is a frame with a rule call, then the topmost frame is replaced by a new frame with the body of the rule call.

As in the parallel case we now prove that a turbo ASM without **par** and **forall** can make one big step from $\mathfrak{A}$ to $\mathfrak{B}$ if, and only if, there exists a finite sequence of intermediate steps as described above.

Lemma 3. *Let P be a rule of an ASM without* **par** *and* **forall**. *If P yields U in $\mathfrak{A}$ under ζ, then there exists a finite sequence of steps from $\langle \emptyset, [\langle P, \zeta \rangle] \rangle$ to $\langle U, [] \rangle$.*

Proof. One shows the following more general statement by induction on the definition of 'yields' in Table 1:

If $\text{yields}(P, \mathfrak{A} + V, \zeta, U)$, then for each stack S there exists a finite sequence $(\sigma_i)_{1 \leq i \leq n}$ of intermediate states such that

1. $\sigma_0 = \langle V, S \cdot [\langle P, \zeta \rangle] \rangle$,
2. $\sigma_n = \langle V \oplus U, S \rangle$,
3. $\sigma_{i+1} = \text{step}(\sigma_i, \mathfrak{A})$ for each $i < n$.

The lemma is then the special case where V is the empty update set and S is the empty stack. □

Lemma 4. *Let P be a rule of an ASM without* **par** *and* **forall**. *If there exists a finite sequence of steps from $\langle \emptyset, [\langle P, \zeta \rangle] \rangle$ to $\langle U, [] \rangle$, then P yields U in $\mathfrak{A}$ under ζ.*

Proof. First we define by induction on the length of the stack S what it means that an intermediate state $\langle U, S \rangle$ yields the update set V (written $\langle U, S \rangle \rhd V$).

1. $\langle U, [] \rangle \rhd U$.
2. If $\text{yields}(P, \mathfrak{A} + U, \zeta, V)$ and $\langle U \oplus V, S \rangle \rhd W$, then $\langle U, S \cdot [\langle P, \zeta \rangle] \rangle \rhd W$.

Then we prove the following statement:

If $\text{step}(\sigma, \mathfrak{A}) = \tau$ and $\tau \rhd W$, then $\sigma \rhd W$.

Thereby, the following property is used:

If $\text{ev}(P, \mathfrak{A}, \zeta) = \{\langle Q, \eta \rangle\}$, then the following two statements are equivalent:

$$\text{yields}(P, \mathfrak{A}, \zeta, U) \iff \text{yields}(Q, \mathfrak{A}, \eta, U)$$

Finally, one proves by induction on n the following statement:

If $(\sigma_i)_{1 \leq i \leq n}$ is a sequence of intermediate steps such that

1. $\sigma_0 = \langle \emptyset, [\langle P, \zeta \rangle] \rangle$,
2. $\text{step}(\sigma_i, \mathfrak{A}) = \sigma_{i+1}$ for each $i < n$,
3. $\sigma_n \rhd W$,

then $\text{yields}(P, \mathfrak{A}, \zeta, W)$.

The lemma is then the special case where $\sigma_n = \langle U, [] \rangle$. □

6 General turbo ASMs

In this section we consider the general case of recursive ASMs with parallel and sequential composition. In the general case, several sequential ASMs can run in parallel as shown in the following example:

$$P \textbf{ seq } ((Q_1 \textbf{ seq } Q_2) \textbf{ par } (R_1 \textbf{ seq } R_2))$$

According to the semantics of rules in Table 1, the update set of this transition rule in a state $\mathfrak{A}$ is $U \oplus ((V_1 \oplus V_2) \cup (W_1 \oplus W_2))$, if

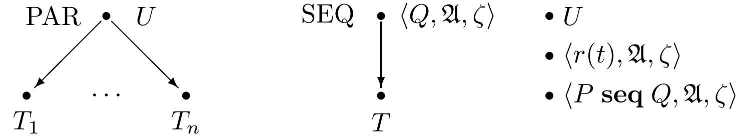
1. P yields U in $\mathfrak{A}$ and U is consistent,
2. Q_1 yields V_1 in $\mathfrak{A} + U$ and V_1 is consistent,
3. Q_2 yields V_2 in $(\mathfrak{A} + U) + V_1$,
4. R_1 yields W_1 in $\mathfrak{A} + U$ and W_1 is consistent,
5. R_2 yields W_2 in $(\mathfrak{A} + U) + W_1$.

The example shows that, in general, the update set that has to be added to the initial state $\mathfrak{A}$ in order to obtain the current state and the update set to which the result of a transition rule has to be added are not the same. For example, the transition rule Q_2 is evaluated in state $\mathfrak{A} + (U \oplus V_1)$, but the result set V_2 has to be added to V_1 and not to $U \oplus V_1$.

Like in the parallel and sequential cases, between two main states $\mathfrak{A}$ and $\mathfrak{B}$ of a recursive ASM there is a finite sequence of intermediate states:

$$\mathfrak{A} \to \sigma_0 \to \sigma_1 \to \ \cdots \ \to \sigma_n \to \mathfrak{B}$$

In the general case, the intermediate states between two main states of a turbo ASM are trees. The trees are generalized stacks. The leaves of the trees are either update sets or frames containing rule calls or sequential compositions. The internal nodes of the trees are labeled with PAR or SEQ. The trees are called PAR/SEQ trees. The nodes can be pictured as follows:



Each PAR node has an update set. The children of the PAR node are executed in parallel. The union of their result sets is added to the update set stored in the PAR node. A PAR node with update set U and children $T_1, \ldots, T_n$ is written as $\mathrm{PAR}(U, [T_1, \ldots, T_n])$. If each subtree T_i yields an update set V_i, then the PAR node yields the update set $U \oplus (V_1 \cup \ldots \cup V_n)$.

A SEQ node has a frame $\langle Q, \mathfrak{A}, \zeta \rangle$ and a single child T. It is written as $\mathrm{SEQ}(\mathfrak{A}, T, Q, \zeta)$. If the child T yields a consistent update set U, then Q is executed in the state $\mathfrak{A} + U$. The resulting update set is then added to U. Formally, PAR/SEQ trees are defined by the following grammar:

$$T ::= U \mid \langle r(t), \mathfrak{A}, \zeta \rangle \mid \langle P \ \mathbf{seq} \ Q, \mathfrak{A}, \zeta \rangle \mid \mathrm{PAR}(U, [T_1, \ldots, T_n]) \mid \mathrm{SEQ}(\mathfrak{A}, T, Q, \zeta)$$

Since in our notation ev() denotes a set and the PAR trees are defined using a list of children (given by a partial evaluation as we will see in the rewrite rules below), we introduce $\overline{\mathrm{ev}}()$ to represent the list of all elements from the corresponding ev() (updates are combined into a single set), i.e.

$$\overline{\mathrm{ev}}(P, \mathfrak{A}, \zeta) = [U, F_1, \ldots, F_k], \text{ if } \mathrm{ev}(P, \mathfrak{A}, \zeta) = U \cup \{F_1, \ldots, F_k\},$$

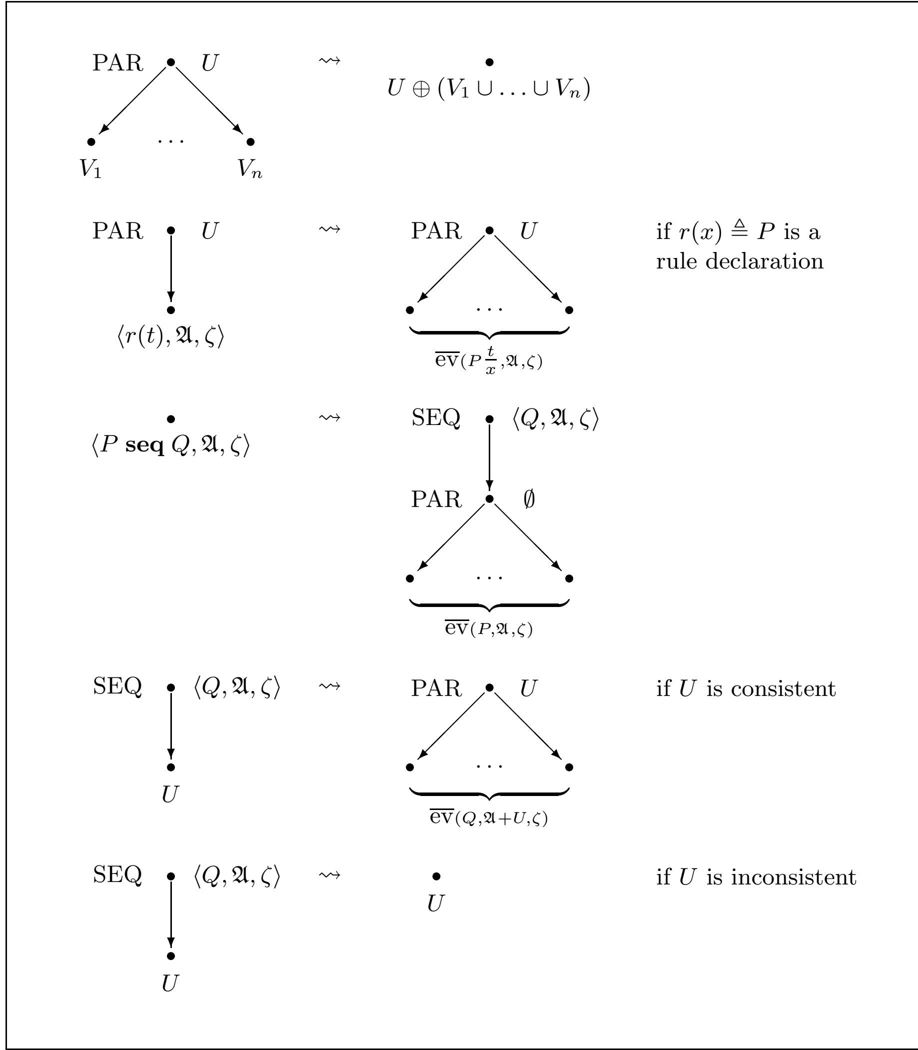


Table 3. Graphical representation for the rewrite rules.

where U is an update set and $F_1, \ldots, F_k$ are frames.

The first intermediate state σ_0 is the PAR tree consisting of an empty update set and the children given by the partial evaluation of the main rule of the ASM in the state $\mathfrak{A}$:

$$\sigma_0 = \mathrm{PAR}(\emptyset, \overline{\mathrm{ev}}(P_0, \mathfrak{A}, \zeta_0))$$

The last intermediate state σ_n is a tree consisting of a single node hosting an update set U. The next main state $\mathfrak{B}$ is obtained from $\mathfrak{A}$ by applying the

updates from U, i.e.

$$\mathfrak{B} = \mathfrak{A} + U$$

Since the intermediate states are given as trees, the transition between trees is described by means of *rewrite rules* for trees, i.e. given an intermediate state σ_i, the next intermediate state σ_{i+1} is determined by applying simultaneously the rules from Table 3 to the tree that represents σ_i.

Note that, because of the rewrite rules, all the intermediate states except the last are PAR trees. The SEQ trees may occur only as subtrees of these intermediate states. The subtrees of an intermediate PAR tree are transformed simultaneously as follows:

1. If the children of a PAR node with update set U are update sets $V_1, \ldots, V_n$, then the PAR node is replaced by the update set obtained from U by adding the union $V_1 \cup \ldots \cup V_n$.
2. If a leaf is a rule call, then it is replaced by the partial evaluation of the rule obtained by substituting the formal parameters in the corresponding rule body.
3. If a leaf is a sequential composition $\langle P \textbf{ seq } Q, \mathfrak{A}, \zeta\rangle$, then it is replaced by the SEQ tree with the frame $\langle Q, \mathfrak{A}, \zeta\rangle$ and the child given by the PAR tree with an empty update set and children given by the partial evaluation of P.
4. If a subtree is a SEQ tree with the frame $\langle Q, \mathfrak{A}, \zeta\rangle$ and its child is a consistent update set U, then it is replaced by the PAR tree with the update set U and the children given by the partial evaluation of Q in the state $\mathfrak{A} + U$ under ζ.
5. If a subtree is a SEQ tree and its child is an inconsistent update set U, then it is replaced by the tree consisting of a single node given by U.

We say that a tree S is a successor of a tree T (written $T \rightsquigarrow S$) iff the tree S is obtained by applying the rewrite rules to T. Since we do not deal only with single trees, but also with lists of trees, we inductively define what it means that a tree T rewrites to a list of trees $S_1, \ldots, S_k$ (written $T \rightsquigarrow [S_1, \ldots, S_k]$):

1. If U is an update set, then $U \rightsquigarrow [U]$.
2. If $r(x) \triangleq P$ is a rule declaration, then $\langle r(t), \mathfrak{A}, \zeta\rangle \rightsquigarrow \overline{\mathrm{ev}}(P\frac{t}{x}, \mathfrak{A}, \zeta)$.
3. $\langle P \textbf{ seq } Q, \mathfrak{A}, \zeta\rangle \rightsquigarrow [\mathrm{SEQ}(\mathfrak{A}, \mathrm{PAR}(\emptyset, \overline{\mathrm{ev}}(P, \mathfrak{A}, \zeta)), Q, \zeta)]$.
4. If $V_1, \ldots, V_n$ are update sets, then $\mathrm{PAR}(U, [V_1, \ldots, V_n]) \rightsquigarrow [U \oplus (V_1 \cup \ldots \cup V_n)]$.
5. If $L_1, \ldots, L_n$ are lists of trees such that $T_i \rightsquigarrow L_i$ for all $i = 1, \ldots, n$ and not all of $T_1, \ldots, T_n$ are update sets, then

   $$\mathrm{PAR}(U, [T_1, \ldots, T_n]) \rightsquigarrow [\mathrm{PAR}(U, L_1 \cdot \ldots \cdot L_n)]$$

   (where $L_1 \cdot \ldots \cdot L_n$ denotes the concatenation of the lists $L_1, \ldots, L_n$).
6. If S is not an update set and $S \rightsquigarrow [T]$, then

   $$\mathrm{SEQ}(\mathfrak{A}, S, Q, \zeta) \rightsquigarrow [\mathrm{SEQ}(\mathfrak{A}, T, Q, \zeta)].$$

7. If U is a consistent update set, then

   $$\mathrm{SEQ}(\mathfrak{A}, U, Q, \zeta) \rightsquigarrow [\mathrm{PAR}(U, \overline{\mathrm{ev}}(Q, \mathfrak{A} + U, \zeta))].$$

8. If U is an inconsistent update set, then $\mathrm{SEQ}(\mathfrak{A}, U, Q, \zeta) \rightsquigarrow [U]$.

According to the previous recursive definition, if T and S are trees such that $T \rightsquigarrow [S]$, then S is a successor of T. We denote by $\rightsquigarrow^*$ the transitive closure of $\rightsquigarrow$ and consider the natural extension of $\rightsquigarrow$ (and implicitly $\rightsquigarrow^*$) for lists of trees.

One direction of the main result can be proved by induction on the definition of 'yields': if a (general) turbo ASM can make one big step from $\mathfrak{A}$ to $\mathfrak{B}$ then, there exists a finite sequence of intermediate states as described above.

Lemma 5. *Let P be a rule of a general turbo ASM. If P yields U in $\mathfrak{A}$ under ζ, then there exist update sets $V_1, \ldots, V_k$ such that*

$$\overline{\mathrm{ev}}(P, \mathfrak{A}, \zeta) \rightsquigarrow^* [V_1, \ldots, V_k] \quad \textit{and} \quad U = V_1 \cup \ldots \cup V_k.$$

Proof. By induction on the definition of $\mathrm{yields}(P, \mathfrak{A}, \zeta, U)$ in Table 1. □

For the other direction we have to define what it means that a PAR/SEQ tree T yields an update set U (written $T \rhd U$). The relation $T \rhd U$ is inductively defined as follows :

1. $U \rhd U$.
2. If $\mathrm{yields}(r(t), \mathfrak{A}, \zeta, U)$, then $\langle r(t), \mathfrak{A}, \zeta \rangle \rhd U$.
3. If $\mathrm{yields}(P \textbf{ seq } Q, \mathfrak{A}, \zeta, U)$, then $\langle P \textbf{ seq } Q, \mathfrak{A}, \zeta \rangle \rhd U$.
4. If $T = \mathrm{PAR}(U, [T_1, \ldots, T_n])$ and $T_i \rhd V_i$ for all $1 \leq i \leq n$, then $T \rhd U \oplus (V_1 \cup \ldots \cup V_n)$.
5. If $S \rhd U$ with U consistent and $\mathrm{yields}(Q, \mathfrak{A} + U, \zeta, V)$, then $\mathrm{SEQ}(\mathfrak{A}, S, Q, \zeta) \rhd U \oplus V$.
6. If $S \rhd U$ with U inconsistent, then $\mathrm{SEQ}(\mathfrak{A}, S, Q, \zeta) \rhd U$.

The following result generalizes a property used in the proof of Lemma 4.

Lemma 6. *Let P be a rule of a general turbo ASM. If*

$$\mathrm{ev}(P, \mathfrak{A}, \zeta) = U \cup \{\langle P_i, \mathfrak{A}, \eta_i \rangle \mid 1 \leq i \leq k\}$$

where U is an update set and $\mathrm{yields}(P_i, \mathfrak{A}, \eta_i, V_i)$ *for each $i = 1, \ldots, k$, then the following holds:*

$$\mathrm{yields}(P, \mathfrak{A}, \zeta, U \cup \bigcup_{1 \leq i \leq k} V_i)$$

Proof. By induction on the size of P using the definition of ev() in Table 2. □

A tree that rewrites to a list of trees, yields an update set, that is the same with the union of the update sets the trees from the list yield:

Lemma 7. *If T and $S_1, \ldots, S_k$ are trees such that $T \rightsquigarrow [S_1, \ldots, S_k]$ and $S_i \rhd V_i$ for all $i \leq k$, then*

$$T \rhd \bigcup_{1 \leq i \leq k} V_i.$$

Proof. By induction on the definition of $T \rightsquigarrow [S_1, \dots, S_k]$. We use Lemma 6 for the cases when T has one of the following forms:

1. $T = \langle r(t), \mathfrak{A}, \zeta \rangle$,
2. $T = \langle P \textbf{ seq } Q, \mathfrak{A}, \zeta \rangle$,
3. $T = \text{SEQ}(\mathfrak{A}, U, Q, \zeta)$ with U consistent.

We need the result of the lemma only for the particular case:

T, S are intermediate states (PAR trees or update sets) such that $T \rightsquigarrow [S]$,

but since the proof is based on the inductive definition of $T \rightsquigarrow [S_1, \dots, S_k]$, one has to prove the lemma for all the trees. □

Now, we are able to prove the other direction of the main result: if there exists a finite sequence of rewrite steps that ends into an update set U, then a turbo ASM can make one big step from $\mathfrak{A}$ to $\mathfrak{A} + U$.

Lemma 8. *Let P be a rule of a general turbo ASM. If there exists a finite sequence of successors from* $\text{PAR}(\emptyset, \overline{\text{ev}}(P, \mathfrak{A}, \zeta))$ *to an update set U, then P yields U in $\mathfrak{A}$ under ζ.*

Proof. If $(\sigma_i)_{0 \le i \le n}$ is a sequence of trees such that

1. $\sigma_0 = \text{PAR}(\emptyset, \overline{\text{ev}}(P, \mathfrak{A}, \zeta))$,
2. $\sigma_i \rightsquigarrow \sigma_{i+1}$ for each $i < n$,
3. $\sigma_n = U$,

then, using Lemma 7 for the case $k = 1$, it can be proved by induction on $n - i$ that $\sigma_i \rhd U$ for each $i \le n$. Then, the proof is done, since $\text{PAR}(\emptyset, \overline{\text{ev}}(P, \mathfrak{A}, \zeta)) \rhd U$ implies $\text{yields}(P, \mathfrak{A}, \zeta, U)$ (Lemma 6). □

7 The reserve and return values

Algorithms often need to increase their working space. ASMs therefore have by definition (Lipari Guide [10]) an infinite reserve which is part of the base set. New elements are allocated using the construct

import x **do** P

which means 'choose an element x from the reserve, delete it from the reserve and execute P'. The reserve of a state is represented using a special dynamic predicate (boolean valued function) `Reserve` such that the reserve elements $\text{Res}(\mathfrak{A})$ of a state $\mathfrak{A}$ is the set of elements a of $\mathfrak{A}$ such that $\texttt{Reserve}(a) = \textit{true}$ in $\mathfrak{A}$.

The special dynamic function `Reserve` is avoided in [2,11]. In [11] the reserve of a state is defined to be the base set of the state minus the contents of locations and the elements (arguments) of defined ($\neq$ *undef*) locations. In the presence of sequential composition, however, the use of a special dynamic function `Reserve` seems to be unavoidable.

Rule	Condition
$\dfrac{\text{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\text{yields}(\textbf{import}\ x\ \textbf{do}\ P, \mathfrak{A}, \zeta, V)}$	if $a \in \text{Res}(\mathfrak{A}) \setminus \text{ran}(\zeta)$ and $V = U \cup \{\langle\langle \texttt{Reserve}, a\rangle, \mathit{false}\rangle\}$
$\dfrac{\text{yields}(P, \mathfrak{A}, \zeta, U) \quad \text{yields}(Q, \mathfrak{A}, \zeta, V)}{\text{yields}(P\ \textbf{par}\ Q, \mathfrak{A}, \zeta, U \cup V)}$	if $\text{Res}(\mathfrak{A}) \cap \text{El}(U) \cap \text{El}(V) \subseteq \text{ran}(\zeta)$
$\dfrac{\text{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U_a) \quad \text{for each } a \in I}{\text{yields}(\textbf{forall}\ x\ \textbf{with}\ \varphi\ \textbf{do}\ P, \mathfrak{A}, \zeta, \bigcup_{a \in I} U_a)}$	if $I = \text{range}(x, \varphi, \mathfrak{A}, \zeta)$ and for $a \neq b$ $\text{Res}(\mathfrak{A}) \cap \text{El}(U_a) \cap \text{El}(U_b) \subseteq \text{ran}(\zeta)$

Table 4. The semantics of turbo ASMs with a reserve.

Why? First, one has to ensure that, if new elements are imported in parallel, they are different. For example, after executing the rule

$$(\textbf{import}\ x\ \textbf{do}\ f(x) := 0)\ \textbf{par}\ (\textbf{import}\ x\ \textbf{do}\ f(x) := 0)$$

there exist two different elements a and b with $f(a) = 0$ and $f(b) = 0$. That different occurrences of **import** get different reserve elements can only be ensured from outside by requiring, for example, in a **par** rule that the reserve elements of the update sets of the two components are disjoint.

Now, if we execute a sequential rule P **seq** Q and we do not use the special predicate `Reserve` as it is done in [11], it can happen that an element which is not in the reserve in the initial state, is again in the reserve after executing P (for example, if P updates all locations that contain the element to *undef*). Hence, the element can be imported in Q. In the combined update set of P and Q, however, the element does no longer count as a reserve element, since it is not in the reserve of the initial state but only in the reserve of the intermediate state (which is hidden from outside). Therefore, if we execute two copies of P **seq** Q in parallel, it could be that both processes import the same element inside Q without violating the condition that in a union of update sets the reserve elements are disjoint.

For this reason we use the special dynamic function `Reserve` and say that a state $\mathfrak{A}$ satisfies the *reserve condition* with respect to an environment ζ, if the following two conditions hold for each element $a \in \text{Res}(\mathfrak{A}) \setminus \text{ran}(\zeta)$:

R1. The element a is not the contents of a location of $\mathfrak{A}$.
R2. If a is an element of a location l of $\mathfrak{A}$ which is not a location for `Reserve`, then the contents of l in $\mathfrak{A}$ is *undef*.

The new rules for **import** are listed in Table 4. Rather than using *actions* (equivalence classes of update sets modulo permutations of the reserve, see [11]) instead of update sets, we add additional constraints to the rules for **par** and **forall**. By $\text{El}(U)$ we denote the set of elements that occur in the updates of U. The elements of a location $\langle f, \langle a_1, \ldots, a_n\rangle\rangle$ are the arguments $a_1, \ldots, a_n$. The elements of an update $\langle l, v\rangle$ are the elements of the location l and the value v.

When a new element a is imported in the rule for **import** we require that a is an element of the reserve of $\mathfrak{A}$ but does not occur in the range of the variable

assignment ζ which contains possibly other new elements that are imported in the same step. The value of the variable x is redefined to a and the body P of the **import** rule is executed in the new environment. The function `Reserve` is updated at the argument a to *false* (by the system and not directly by the ASM).

The constraint for P **par** Q says that the reserve elements of the update set U for P are disjoint from the reserve elements of the update set V for Q except for reserve elements in the range of ζ. We have to exclude the range of ζ, since otherwise the execution of a rule like

$$\textbf{import } x \textbf{ do } (f(x) := 0 \textbf{ par } g(x) := 0)$$

would be impossible.

In order that the constraints in Table 4 work correctly, we have to require that in the scope of bound variable the same variable is not used again as a bound variable (in a **let**, **forall**, **choose**, or **import**). Otherwise, it could happen that a reserve element that is imported and bound to a variable in the environment is hidden and then imported again. Example:

import x **do**
  $f(x) := 0$ **par**
  let $x = 1$ **in**
    import y **do** $f(y) := x$

In this example, the same reserve element could be used for x as well as for y, since the outermost x is hidden by the **let** and not visible in the environment, when y is imported.

The range of a formula has to be redefined such that reserve elements which are not in the range of the environment are excluded in the **forall** rule and cannot be chosen in the **choose** rule:

$$\text{range}(x, \varphi, \mathfrak{A}, \zeta) = \{a \in |\mathfrak{A}| : a \notin \text{Res}(\mathfrak{A}) \setminus \text{ran}(\zeta),\ [\![\varphi]\!]^{\mathfrak{A}}_{\zeta[x\mapsto a]} = \mathit{true}\}$$

If the constraints in Table 4 are enforced and a transition rule yields a consistent update set in a state that satisfies the reserve condition, then the state also satisfies the reserve condition after firing the updates. Hence, the reserve condition is preserved in runs of ASMs.

Lemma 9 (Preservation of the reserve condition). *If a state $\mathfrak{A}$ satisfies the reserve condition wrt. ζ and P yields a consistent update set U in $\mathfrak{A}$ under ζ, then*

1. *for every element a of U which is in in the reserve of $\mathfrak{A}$ but not in the range of ζ, the update $\langle\langle \texttt{Reserve}, a\rangle, \mathit{false}\rangle$ is in U,*
2. *the sequel $\mathfrak{A} + U$ satisfies the reserve condition wrt. ζ.*

Proof. We first show that statement 1 implies statement 2. Let l be a location and v its content in $\mathfrak{A} + U$. Then either the update $\langle l, v\rangle$ is in U or v is the

content of l in $\mathfrak{A}$. We have to show that the reserve conditions R1 and R2 are satisfied in $\mathfrak{A}+U$. Let a be in $\mathrm{Res}(\mathfrak{A}+U)\setminus \mathrm{ran}(\zeta)$. Then a is also in $\mathrm{Res}(\mathfrak{A})$.

Case 1. $\langle l, v\rangle \in U$: Then a cannot be an element of $\langle l, v\rangle$, since otherwise, by 1, the update $((\texttt{Reserve}, a), \mathit{false})$ would be in U and a could not be in the reserve of $\mathfrak{A}+U$.

Case 2. $\mathfrak{A}(l) = v$: Since $\mathfrak{A}$ satisfies the reserve condition wrt. ζ, a is different from v and, if a is an element of l and l not a location for `Reserve`, then v is $undef$.

Now we prove statement 1 by induction on the definition of 'yields'. We can assume that the bound variables of P are not in the domain of the environment ζ. We consider the critical cases:

Case **let**: Assume that $a = [\![t]\!]_\zeta^{\mathfrak{A}}$ and

$$\frac{\mathrm{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\mathrm{yields}(\textbf{let } x = t \textbf{ in } P, \mathfrak{A}, \zeta, U)}$$

Since $\mathfrak{A}$ satisfies the reserve consider wrt. ζ and x is not in the domain of ζ, the state $\mathfrak{A}$ satisfies the reserve condition also with respect to the extended environment $\zeta[x \mapsto a]$. Let b be an element of U which is in $\mathrm{Res}(\mathfrak{A})\setminus \mathrm{ran}(\zeta)$. Then the element b must be different from a, since the value of the term t in $\mathfrak{A}$ under ζ is the content of a location of $\mathfrak{A}$ or an element in the range of ζ. Hence b is in $\mathrm{Res}(\mathfrak{A})\setminus \mathrm{ran}(\zeta[x \mapsto a])$ and, by the induction hypothesis, the update $\langle\langle\texttt{Reserve}, b\rangle, \mathit{false}\rangle$ is in U.

Case **import**: Assume that $a \in \mathrm{Res}(\mathfrak{A})\setminus \mathrm{ran}(\zeta)$ and

$$\frac{\mathrm{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\mathrm{yields}(\textbf{import } x \textbf{ do } P, \mathfrak{A}, \zeta, U \cup \{\langle\langle\texttt{Reserve}, a\rangle, \mathit{false}\rangle\})}$$

Let b be an element of U which is in $\mathrm{Res}(\mathfrak{A})\setminus \mathrm{ran}(\zeta)$. If $b = a$, we are done, since the update $\langle\langle\texttt{Reserve}, a\rangle, \mathit{false}\rangle$ is automatically added. Otherwise, b is in $\mathrm{Res}(\mathfrak{A})\setminus \mathrm{ran}(\zeta[x \mapsto a])$ and we can apply the induction hypothesis.

Case **seq**: Assume that U is consistent and

$$\frac{\mathrm{yields}(P, \mathfrak{A}, \zeta, U) \qquad \mathrm{yields}(Q, \mathfrak{A}+U, \zeta, V)}{\mathrm{yields}(P \textbf{ seq } Q, \mathfrak{A}, \zeta, U \oplus V)}$$

By the induction hypothesis for P and since statement 1 implies statement 2, the state $\mathfrak{A}+U$ satisfies the reserve condition wrt. ζ. Let a be an element of $U \oplus V$ which is in $\mathrm{Res}(\mathfrak{A})\setminus \mathrm{ran}(\zeta)$. If a is an element of an update of U, then by the induction hypothesis for P, the update $\langle\langle\texttt{Reserve}, a\rangle, \mathit{false}\rangle$ is in U and hence also in $U \oplus V$. Otherwise, a is an element of an update of V. If a is not in the reserve von $\mathfrak{A}+U$, then a must have been deleted from the reserve of $\mathfrak{A}$ by an update $\langle\langle\texttt{Reserve}, a\rangle, \mathit{false}\rangle$ of U. If a is in $\mathrm{Res}(\mathfrak{A}+U)$, then by the induction hypothesis for Q, the update $\langle\langle\texttt{Reserve}, a\rangle, \mathit{false}\rangle$ is in V and hence also in $U \oplus V$. □

If a structure $\mathfrak{A}$ satisfies the reserve condition wrt. ζ, then a permutation of the set $\mathrm{Res}(\mathfrak{A})\setminus\mathrm{ran}(\zeta)$ can always be extended to an automorphism of the structure $\mathfrak{A}$ which is the identity on the non-reserve elements and the elements in the range of ζ. Hence, the following lemma can be used to rename the reserve elements using a permutation of $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$.

Lemma 10 (Preservation of isomorphisms). *If α is an isomorphism from $\mathfrak{A}$ to $\mathfrak{B}$ and P yields U in $\mathfrak{A}$ under ζ, then P yields $\alpha(U)$ in $\mathfrak{B}$ under $\alpha \circ \zeta$.*

We can now prove that the update set computed by a transition rule in a state is unique modulo permutations of the reserve (if no **choose** is used). This is the justification for the constraints in Table 4. The lemma is true only for ASMs that produce finite update sets.

Lemma 11 (Independence of the choice of reserve elements). *Let P be a rule of an ASM without* **choose**. *If $\mathfrak{A}$ satisfies the reserve condition wrt. ζ, the bound variables of P are not in the domain of ζ, and P yields two finite update sets U and U' in $\mathfrak{A}$, then there exists a permutation α of* $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$ *such that $\alpha(U) = U'$.*

Proof. By induction on the definition of 'yields'. We consider the critical cases.

Case **import**: Assume that $a, a' \in \mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$ and

$$\frac{\mathrm{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\mathrm{yields}(\textbf{import } x \textbf{ do } P, \mathfrak{A}, \zeta, U \cup \{\langle\langle \texttt{Reserve}, a\rangle, \mathit{false}\rangle\})}$$

$$\frac{\mathrm{yields}(P, \mathfrak{A}, \zeta[x \mapsto a'], U')}{\mathrm{yields}(\textbf{import } x \textbf{ do } P, \mathfrak{A}, \zeta, U' \cup \{\langle\langle \texttt{Reserve}, a'\rangle, \mathit{false}\rangle\})}$$

Let α be the permutation of $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$ that transposes a and a'.

Since α is an automorphism of $\mathfrak{A}$ and $\alpha \circ (\zeta[x \mapsto a]) = \zeta[x \mapsto a']$, we can apply Lemma 10 and obtain $\mathrm{yields}(P, \mathfrak{A}, \zeta[x \mapsto a'], \alpha(U))$.

Since $x \notin \mathrm{dom}(\zeta)$, we have $\mathrm{ran}(\zeta[x \mapsto a']) = \mathrm{ran}(\zeta) \cup \{a'\}$ and therefore $\mathfrak{A}$ satisfies the reserve condition wrt. $\zeta[x \mapsto a']$. By the induction hypothesis, there exists a permutation β of $\mathrm{Res}(\mathfrak{A}) \setminus (\mathrm{ran}(\zeta) \cup \{a'\})$ such that $\beta(\alpha(U)) = U'$. The composition $\beta \circ \alpha$ is a permutation of $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$. Since $\beta(\alpha(a)) = a'$, the function $\beta \circ \alpha$ maps the update $\langle\langle \mathit{Reserve}, a\rangle, \mathit{false}\rangle$ to $\langle\langle \mathit{Reserve}, a'\rangle, \mathit{false}\rangle$.

Case **par**: Assume that

$$\frac{\mathrm{yields}(P, \mathfrak{A}, \zeta, U) \quad \mathrm{yields}(Q, \mathfrak{A}, \zeta, V)}{\mathrm{yields}(P \textbf{ par } Q, \mathfrak{A}, \zeta, U \cup V)} \qquad \frac{\mathrm{yields}(P, \mathfrak{A}, \zeta, U') \quad \mathrm{yields}(Q, \mathfrak{A}, \zeta, V')}{\mathrm{yields}(P \textbf{ par } Q, \mathfrak{A}, \zeta, U' \cup V')}$$

and $\mathrm{Res}(\mathfrak{A}) \cap \mathrm{El}(U) \cap \mathrm{El}(V) \subseteq \mathrm{ran}(\zeta)$ and $\mathrm{Res}(\mathfrak{A}) \cap \mathrm{El}(U') \cap \mathrm{El}(V') \subseteq \mathrm{ran}(\zeta)$.

By the induction hypothesis there exist permutations α and β of $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$ such that $\alpha(U) = U'$ and $\beta(V) = V'$.

Let γ be the restriction of $\alpha \cup \beta$ to $\mathrm{El}(U) \cup \mathrm{El}(V)$. The function γ is well-defined. Let a be an element in $\mathrm{El}(U) \cap \mathrm{El}(V)$. If a is not in $\mathrm{Res}(\mathfrak{A})$, then $\alpha(a) = a = \beta(a)$. If a is in $\mathrm{Res}(\mathfrak{A})$, then $a \in \mathrm{ran}(\zeta)$ and $\alpha(a) = a = \beta(a)$.

The function γ is injective. Assume that $\alpha(a) = \beta(b)$, where $a \in \mathrm{El}(U)$ and $b \in \mathrm{El}(V)$. Let $c = \alpha(a)$. If c is not in $\mathrm{Res}(\mathfrak{A})$, then $c = \alpha(c)$ and therefore $a = \alpha(a) = c = \beta(b) = b$. If c is in $\mathrm{Res}(\mathfrak{A})$, then $c \in \mathrm{Res}(\mathfrak{A}) \cap \mathrm{El}(U') \cap \mathrm{El}(V')$ and therefore $c \in \mathrm{ran}(\zeta)$ and $c = \alpha(c)$. As in the previous case, we can conclude that $a = b$.

Since $\mathrm{El}(U) \cup \mathrm{El}(V)$ is finite, there exists a permutation δ of $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$ such that $\delta(U) = \gamma(U) = \alpha(U) = U'$, $\delta(V) = \gamma(V) = \beta(V) = V'$ and thus $\delta(U \cup V) = U' \cup V'$.

Case **seq**: Assume that U and U' are consistent and

$$\frac{\mathrm{yields}(P, \mathfrak{A}, \zeta, U) \quad \mathrm{yields}(Q, \mathfrak{A} + U, \zeta, V)}{\mathrm{yields}(P \ \mathbf{seq}\ Q, \mathfrak{A}, \zeta, U \oplus V)}$$

$$\frac{\mathrm{yields}(P, \mathfrak{A}, \zeta, U') \quad \mathrm{yields}(Q, \mathfrak{A} + U', \zeta, V')}{\mathrm{yields}(P \ \mathbf{seq}\ Q, \mathfrak{A}, \zeta, U' \oplus V')}$$

By the induction hypothesis there exists a permutation α of $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$ such that $\alpha(U) = U'$. Since α is an automorphism of $\mathfrak{A}$, it is an isomorphism of $\mathfrak{A} + U$ to $\mathfrak{A} + U'$. Since $\alpha \circ \zeta = \zeta$, by Lemma 10, it follows that $\mathrm{yields}(Q, \mathfrak{A} + U', \zeta, \alpha(V))$.

By Lemma 9, it follows that $\mathfrak{A} + U'$ satisfies the reserve condition wrt. ζ and $\mathrm{Res}(\mathfrak{A} + U') \setminus \mathrm{ran}(\zeta)$ is contained in $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{El}(U')$.

By the induction hypothesis, there exists a permutation β of $\mathrm{Res}(\mathfrak{A} + U') \setminus \mathrm{ran}(\zeta)$ such that $\beta(\alpha(V)) = V'$.

Hence, $\beta \circ \alpha$ is a permutation of $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$. Since $\beta(U') = U'$, it follows that $(\beta \circ \alpha)(U \oplus V) = U' \oplus V'$. □

The lemma is in general not true for ASMs that produce infinite update sets. Let $\mathfrak{A}$ be a state that contains two copies of the set of natural numbers, one copy $0, 1, 2, \ldots$ for a subuniverse *Nat* and one copy $\hat{0}, \hat{1}, \hat{2}, \ldots$ for the `Reserve`. Consider the following transition rule:

forall $x \in \mathit{Nat}$ **do**
  import y **do** $f(x) := y$

Two possible update sets of the transition rule in state $\mathfrak{A}$ are:

$$\begin{aligned} U &= \{\langle\langle f, n\rangle, \hat{n}\rangle \mid n \in \mathbb{N}\} \cup \{\langle\langle \mathtt{Reserve}, \hat{n}\rangle, \mathit{false}\rangle \mid n \in \mathbb{N}\} \\ U' &= \{\langle\langle f, n\rangle, \widehat{n+1}\rangle \mid n \in \mathbb{N}\} \cup \{\langle\langle \mathtt{Reserve}, \hat{n}\rangle, \mathit{false}\rangle \mid n > 0\} \end{aligned}$$

In the first case, the reserve is fully exhausted after firing the updates of U. In the second case, the reserve still contains the element $\hat{0}$ after firing U'. A map α that maps U to U' has to map $\hat{n}$ to $\widehat{n+1}$. The map α, however, cannot be a permutation of the reserve, since $\hat{0}$ is not met by α.

ASMs with return values are now obtained by syntactic sugar. We use the notation of Börger and Bolognesi in [6]. The expression **return** t is considered as an abbreviation for the update `result`$(\mathit{this}) := t$, where `result` is special unary dynamic function that is used for returning values. The variable *this* is a special

```
MSORT(x) ≜
  if |x| = 1 then
    return x
  else
    let y = MSORT(l(x)), z = MSORT(r(x)) in
      let r = MERGE(y, z) in return r

MERGE(x, y) ≜
  if |x| = 0 then
    return y
  else if |y| = 0 then
    return x
  else if hd(x) ≤ hd(y) then
    let z = MERGE(tl(x), y) in return hd(x) · z
  else
    let z = MERGE(x, tl(y)) in return hd(y) · z
```

Fig. 3. MergeSort as a turbo ASM with return values.

variable which is implicit in each rule declaration as argument 0. The value of *this* is the location into which the return value has to be stored. If we have a rule declaration $r(x) \triangleq P$, we write $s.r(t)$ to indicate that the variable *this* is bound to s and x is bound to t. We write

$$\textbf{let } x_1 = r_1(t_1), \ldots, x_n = r_n(t_n) \textbf{ in } Q$$

as a syntactic abbreviation for

$$\begin{array}{l}\textbf{import } y_1, \ldots, y_n \textbf{ do} \\ \quad (y_1.r_1(t_1) \textbf{ par } \ldots \textbf{ par } y_n.r_n(t_n)) \textbf{ seq} \\ \quad \textbf{let } x_1 = \texttt{result}(y_1), \ldots, x_n = \texttt{result}(y_n) \textbf{ in } Q\end{array}$$

First, new locations are created for storing the return values. Then the transition rules with return values are called in parallel and store the results in the new locations. After that, the results of the executions are bound to the **let**-variables and are then used in the body Q.

We are now in the position to present another turbo ASM for MergeSort. The ASM in Fig. 3 uses return values and sorts an immutable list. The ASM is a direct translation of the equations (1) and (2) in Sect. 2. Hence we have shown that recursive algorithms *are* ASMs.

8 Recursion in other implementations of ASMs

The inductive semantics for turbo ASMs in Table 1 describes the semantical basis for Schmid's AsmGofer system [15]. A slightly different approach to recursion is

$$\frac{}{\text{run}(P, \mathfrak{A}, \zeta, \mathfrak{A})}$$

$$\frac{\text{run}(P, \mathfrak{A}, \zeta, \mathfrak{B}) \quad \text{yields}(P, \mathfrak{B}, \zeta, U)}{\text{run}(P, \mathfrak{A}, \zeta, \mathfrak{B} + U)} \text{ if } U \text{ is consistent}$$

$$\frac{\text{yields}(P, \mathfrak{A}, \zeta, \emptyset)}{\text{final}(P, \mathfrak{A}, \zeta)}$$

$$\frac{\text{run}(P, \mathfrak{A}, \eta, \mathfrak{B}) \quad \text{final}(P, \mathfrak{B}, \eta)}{\text{yields}(r(t), \mathfrak{A}, \zeta, \mathfrak{B} - \mathfrak{A})} \quad \text{where } r(x) \triangleq P \text{ is a rule declaration, } a = [\![t]\!]^{\mathfrak{A}}_{\zeta} \text{ and } \eta = \zeta[x \mapsto a]$$

Table 5. Inductive definition of the semantics of Xasm rule calls.

taken in Xasm [1]. In this section, we want to point out the essential differences. We focus just on recursion and omit other features of Xasm like the evaluation of terms with side effects (update sets) or the so-called external functions of Xasm. Unfortunately, it is not clear to us which kind of recursion is supported by AsmL [8]. We doubt that it is the translation into distributed ASMs of [4,13], since the partial order based concept of "distributed ASMs runs" as defined at the end of [10] (Lipari Guide) is difficult to implement.

For the definition of the semantics of recursive calls in Xasm we need in addition to the predicate $\text{yields}(P, \mathfrak{A}, \zeta, U)$ two new predicates $\text{run}(P, \mathfrak{A}, \zeta, \mathfrak{B})$ and $\text{final}(P, \mathfrak{A}, \zeta)$. The predicate $\text{run}(P, \mathfrak{A}, \zeta, \mathfrak{B})$ means that there exists a finite run of P from $\mathfrak{A}$ into state $\mathfrak{B}$ where the free variables of P are defined in the environment ζ. The predicate $\text{final}(P, \mathfrak{A}, \zeta)$ means that $\mathfrak{A}$ is a final state for P under ζ.

The predicates $\text{yields}(P, \mathfrak{A}, \zeta, U)$, $\text{run}(P, \mathfrak{A}, \zeta, \mathfrak{B})$ and $\text{final}(P, \mathfrak{A}, \zeta)$ are defined by *simultaneous* induction. The clauses for 'yields' are the same as in Table 1 except for rule calls. The new clauses are listed in Table 5. The predicates 'yields' and 'run' mutually depend on each other. In the definition of a finite 'run' the predicate 'yields' is used. Conversely, in the definition of 'yields' for rule calls, the predicate 'run' is used.

How does a (recursive) rule call $r(t)$ in Xasm work? Assume that $r(x) \triangleq P$ is a rule declaration. First the argument t is evaluated in the current state $\mathfrak{A}$ and the current environment ζ. The value of t is assigned to the variable x in a new environment η. Then a new run is started for the body P in state $\mathfrak{A}$ with the environment η. If after finitely many steps, the run terminates in a final state $\mathfrak{B}$ (where P yields an empty update set), then the rule call $r(t)$ yields the difference between the final state $\mathfrak{B}$ and the initial state $\mathfrak{A}$. The difference $\mathfrak{B} - \mathfrak{A}$ is the set of non-trivial updates that have to be applied to $\mathfrak{A}$ to get the state $\mathfrak{B}$.

$$\mathfrak{B} - \mathfrak{A} = \{\langle l, v\rangle \mid \mathfrak{B}(l) = v,\ \mathfrak{A}(l) \neq v\}$$

Hence the essential difference between a rule call in Xasm and a rule call of a turbo ASM is that in Xasm the rule call means the *repeated* execution of the

body, whereas for turbo ASMs it is just the one-time execution. In the presence of sequential composition (**seq**) the two approaches have the same expressive power, since the iterated execution of an ASM can be obtained as one step of an enclosing turbo ASM. From the logical point of view, the turbo ASM calls are simpler, since they just mean that the call has to be replaced by its body (see [16] for details).

References

1. M. Anlauff and P. Kutter. Xasm Open Source. Web pages at `http://www.xasm.org/`, 2001.
2. A. Blass and Y. Gurevich. Background, reserve, and Gandy machines. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (CSL 2000)*, pages 1–17. Springer-Verlag, Lecture Notes in Computer Science 1862, 2000.
3. A. Blass and Y. Gurevich. Abstract state machines capture parallel algorithms. *ACM Transactions on Computational Logic*, 2002. to appear.
4. A. Blass and Y. Gurevich. Algorithms vs. machines. *The Logic in Computer Science Column, Bulletin of the European Association for Theoretical Computer Science*, 77:96–118, 2002.
5. T. Bolognesi and E. Börger. Abstract State Processes. Technical report, CNR, Istituto IEI—Dipartimento di Informatica, Università di Pisa, 2002.
6. T. Bolognesi and E. Börger. Turbo ASMs for functional equations and recursion schemes. Technical report, CNR, Istituto IEI—Dipartimento di Informatica, Università di Pisa, 2002.
7. E. Börger and J. Schmid. Composition and submachine concepts. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (CSL 2000)*, pages 41–60. Springer-Verlag, Lecture Notes in Computer Science 1862, 2000.
8. Foundations of Software Engineering Group, Microsoft Research. AsmL. Web pages at `http://research.microsoft.com/foundations/AsmL/`, 2001.
9. 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 — Advances in Theory and Applications, 10th International Workshop, ASM 2003, Taormina, Italy*, pages 244–262. Springer-Verlag, Lecture Notes in Computer Science 2589, 2003.
10. Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1993.
11. Y. Gurevich. May 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, EECS Dept., University of Michigan, 1997.
12. Y. Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, 2000.
13. Y. Gurevich and M. Spielmann. Recursive Abstract State Machines. *J. of Universal Computer Science*, 3(4):233–246, 1997.
14. Y. Moschovakis. What is an algorithm? In B. Engquist and W. Schmid, editors, *Mathematics Unlimited — 2001 and beyond (Part II)*, pages 919–936. Springer-Verlag, 2001.
15. J. Schmid. Executing ASM specifications with AsmGofer. Web pages at `http://www.tydo.de/AsmGofer`.
16. R. F. Stärk and S. Nanchen. A logic for Abstract State Machines. *J. of Universal Computer Science*, 7(11):981–1006, 2001.

17. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine—Definition, Verification, Validation*. Springer-Verlag, 2001.