Composition and Submachine Concepts for Sequential ASMs

Egon Börger[1] and Joachim Schmid[2]

[1] Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
boerger@di.unipi.it (Visiting Microsoft Research, Redmond)
[2] Siemens AG, Corporate Technology, D-81730 Munich, Germany
joachim.schmid@mchp.siemens.de

Abstract. We define three composition and structuring concepts which reflect frequently used refinements of ASMs and integrate standard structuring constructs into the global state based parallel ASM view of computations. First we provide an operator which combines the atomic update view of ASMs with *sequential* machine *execution* and naturally incorporates classical *iteration* constructs into ASMs. For structuring large machines we define their *parameterization*, leading to a notion of possibly recursive submachine calls which sticks to the bare logical minimum needed for sequential ASMs, namely consistency of simultaneous machine operations. For encapsulation and state hiding we provide ASMs with *local state*, *return values* and *error handling*.
Some of these structuring constructs have been implemented in ASM-Gofer. We provide also a proof-theoretic definition which supports the use of common structured proof principles for proving properties for complex machines in terms of properties of their components.

1 Introduction

It has often been observed that Gurevich's definition of Abstract State Machines (ASMs) [13] uses only conditional assignments and supports none of the classical control or data structures. On the one side this leaves the freedom – necessary for *high-level* system design and analysis – to introduce during the modeling process any control or data structure whatsoever which may turn out to be suitable for the application under study. On the other hand it forces the designer to specify standard structures over and over again when they are needed, at the latest when it comes to implement the specification. In this respect ASMs are similar to Abrial's Abstract Machines [1] which are expressed by non-executable pseudo-code without sequencing or loop (Abstract Machine Notation, AMN). In particular there is no notion of submachine and no calling mechanism. For both Gurevich's ASMs and Abrial's Abstract Machines, various notions of refinement have been used to introduce the classical control and data structures. See for example the definition in [15] of recursion as a distributed ASM computation (where calling a recursive procedure is modeled by creating a new instance of multiple agents executing the program for the procedure body) and the definition in [1, 12.5] of recursive AMN calls of an operation as calls to the operation of importing the implementing machine.

Operations of B-Machines [1] and of ASMs come in the form of atomic actions. The semantics of ASMs provided in [13] is defined in terms of a function *next* from states (structures) to states which reflects one step of machine execution. We extend this definition to a function describing, as one step, the result of executing an a priori unlimited number n of basic machine steps. Since n could go to ∞, this naturally leads to consider also non halting computations. We adapt this definition to the view of simultaneous atomic updates in a global state, which is characteristic for the semantics of ASMs, and avoid prescribing any specific syntactic form of encapsulation or state hiding. This allows us to integrate the classical control constructs for *sequentialization and iteration* into the global state based ASM view of computations. Moreover this can be done in a compositional way, supporting the corresponding well known structured proof principles for proving properties for complex machines in terms of properties of their components. We illustrate this by providing structured ASMs for computing arbitrary computable functions, in a way which combines the advantages of functional and of imperative programming. The atomicity of the ASM iteration constructor we define below turned out to be the key for a rigorous definition of the semantics of event triggered exiting from compound actions of UML activity and state machine diagrams, where the intended instantaneous effect of exiting has to be combined with the request to exit nested diagrams sequentially following the subdiagram order, see [5,6].

For structuring large ASMs extensive use has been made of macros as notational shorthands. We enhance this use here by defining the semantics of *named parameterized ASM rules* which include also recursive ASMs. Aiming at a foundation which supports the practitioners' procedural understanding and use of submachine calls, we follow the spirit of the basic ASM concept [13] where domain theoretic complications – arising when explaining what it means to iterate the execution of a machine "until ..." – have been avoided, namely by defining only the one-step computation relation and by relegating fixpoint ("termination") concerns to the metatheory. Therefore we define the semantics of submachine calls only for the case that the possible chain of nested calls of that machine is finite. We are thus led to a notion of calling submachines which mimics the standard imperative calling mechanism and can be used for a definition of recursion in terms of *sequential* (not distributed) ASMs. This definition suffices to justify the submachines used in [8] for a hierarchical decomposition of the Java Virtual Machine into loading, verifying and executing machines for the five principal language layers (imperative core, static classes, object oriented features, exception handling and concurrency).

The third kind of structuring mechanism for ASMs we consider in this paper is of syntactical nature, dealing essentially with name spaces. Parnas' [17] information hiding principle is strongly supported by the ASM concept of external functions which provides also a powerful interface mechanism (see [4]). A more syntax oriented form of information hiding can be naturally incorporated into

ASMs through the notion of *local machine state*, of machines with *return values* and of *error handling* machines which we introduce in Section 5.

Some of these concepts have been implemented in ASMGofer [18], allowing us to define executable versions of the machines for Java and the JVM in [8].

2 Standard ASMs

We start from the definition of basic sequential (i.e. non distributed) ASMs in [13] and survey in this section our notation.

Basic ASMs are built up from *function updates* and *skip* by *parallel composition* and constructs for *if then else*, *let* and *forall*. We consider the *choose*-construct as a special notation for using choice functions, a special class of external functions. Therefore we do not list it as an independent construct in the syntactical definition of ASMs. It appears however in the appendix because the non-deterministic selection of the *choose*-value is directly related to the non-deterministic application of the corresponding deduction rule.

The interpretation of an ASM in a given state $\mathfrak{A}$ depends on the given environment *Env*, i.e. the interpretation $\zeta \in \mathit{Env}$ of its free variables. We use the standard interpretation $[\![t]\!]_\zeta^\mathfrak{A}$ of terms t in state $\mathfrak{A}$ under variable interpretation ζ, but we often suppress mentioning the underlying interpretation of variables. The semantics of standard ASMs is defined in [13] by assigning to each rule R, given a state $\mathfrak{A}$ and a variable interpretation ζ, an update set $[\![R]\!]_\zeta^\mathfrak{A}$ which – if consistent – is fired in state $\mathfrak{A}$ and produces the next state $\mathit{next}_R(\mathfrak{A}, \zeta)$.

An update set is a set of *updates*, i.e. a set of pairs $(\mathit{loc}, \mathit{val})$ where loc is a location and val is an element in the domain of $\mathfrak{A}$ to which the location is intended to be updated. A location is n-ary function name f with a sequence of length n of elements in the domain of $\mathfrak{A}$, denoted by $f\langle a_1, \ldots, a_n\rangle$. If u is an update set then $\mathit{Locs}(u)$ denotes the set of locations occurring in elements of u ($\mathit{Locs}(u) = \{\mathit{loc} \mid \exists \mathit{val} : (\mathit{loc}, \mathit{val}) \in u\}$). An update set u is called *inconsistent* if u contains at least two pairs (loc, v_1) and (loc, v_2) with $v_1 \neq v_2$ (i.e. $|u| > |\mathit{Locs}(u)|$), otherwise it is called *consistent*.

For a consistent update set u and a state $\mathfrak{A}$, the state $\mathit{fire}_\mathfrak{A}(u)$, resulting from firing u in $\mathfrak{A}$, is defined as state $\mathfrak{A}'$ which coincides with $\mathfrak{A}$ except $f^{\mathfrak{A}'}(a) = \mathit{val}$ for each $(f\langle a\rangle, \mathit{val}) \in u$. Firing an inconsistent update set is not allowed, i.e. $\mathit{fire}_\mathfrak{A}(u)$ is not defined for inconsistent u. This definition yields the following (partial) next state function next_R which describes *one* application of R in a state with a given environment function $\zeta \in \mathit{Env}$. We often write also $\mathit{next}(R)$ instead of next_R.

$$
\begin{aligned}
&\mathit{next}_R &&: \mathit{State}(\Sigma) \times \mathit{Env} \to \mathit{State}(\Sigma) \\
&\mathit{next}_R(\mathfrak{A}, \zeta) &&= \mathit{fire}_\mathfrak{A}([\![R]\!]_\zeta^\mathfrak{A})
\end{aligned}
$$

The following definitions describe the meaning of standard ASMs. We use R and S for rules, x for variables, s and t for expressions, p for predicates (boolean expressions), and u, v for semantical values and update sets. We write $f^\mathfrak{A}$ for the interpretation of the function f in state $\mathfrak{A}$ and $\zeta' = \zeta\frac{x}{u}$ is the variable environment which coincides with ζ except for x where $\zeta'(x) = u$.

$$
\begin{array}{ll}
[\![x]\!]_\zeta^{\mathfrak{A}} & = \zeta(x) \\
[\![f(t_1,\ldots,t_n)]\!]_\zeta^{\mathfrak{A}} & = f^{\mathfrak{A}}([\![t_1]\!]_\zeta^{\mathfrak{A}},\ldots,[\![t_n]\!]_\zeta^{\mathfrak{A}}) \\
[\![\mathbf{skip}]\!]_\zeta^{\mathfrak{A}} & = \emptyset \\
[\![f(t_1,\ldots,t_n) := s]\!]_\zeta^{\mathfrak{A}} & = \{(f\langle[\![t_1]\!]_\zeta^{\mathfrak{A}},\ldots,[\![t_n]\!]_\zeta^{\mathfrak{A}}\rangle, [\![s]\!]_\zeta^{\mathfrak{A}})\} \\
[\![\{R_1,\ldots,R_n\}]\!]_\zeta^{\mathfrak{A}} & = [\![R_1]\!]_\zeta^{\mathfrak{A}} \cup \cdots \cup [\![R_n]\!]_\zeta^{\mathfrak{A}} \\
[\![\mathbf{if}\ t\ \mathbf{then}\ R\ \mathbf{else}\ S]\!]_\zeta^{\mathfrak{A}} & = \begin{cases} [\![R]\!]_\zeta^{\mathfrak{A}}, & \text{if } [\![t]\!]_\zeta^{\mathfrak{A}} = \mathrm{true}^{\mathfrak{A}} \\ [\![S]\!]_\zeta^{\mathfrak{A}}, & \textbf{otherwise} \end{cases} \\
[\![\mathbf{let}\ x = t\ \mathbf{in}\ R]\!]_\zeta^{\mathfrak{A}} & = [\![R]\!]_{\zeta\frac{x}{v}}^{\mathfrak{A}} \ \text{ where } v = [\![t]\!]_\zeta^{\mathfrak{A}} \\
[\![\mathbf{forall}\ x\ \mathbf{with}\ p\ \mathbf{do}\ R]\!]_\zeta^{\mathfrak{A}} & = \bigcup_{v\in V} [\![R]\!]_{\zeta\frac{x}{v}}^{\mathfrak{A}} \ \text{ where } V = \{v \mid [\![p]\!]_{\zeta\frac{x}{v}}^{\mathfrak{A}} = \mathrm{true}^{\mathfrak{A}}\}
\end{array}
$$

Remark: Usually the parallel composition $\{R_1,\ldots,R_n\}$ of rules R_i is denoted by displaying the R_i vertically one above the other.

For a standard ASM R, the update set $[\![R]\!]_\zeta^{\mathfrak{A}}$ is defined for any state $\mathfrak{A}$ and for any variable environment ζ, but $next_R(\mathfrak{A},\zeta)$ is undefined if $[\![R]\!]_\zeta^{\mathfrak{A}}$ is inconsistent.

3 Sequential Composition and Iteration

The basic composition of ASMs is parallel composition, and this is so for a fundamental reason explained in [14]. It is for practical purposes that in this section we incorporate into ASMs their sequential composition and their iteration, but in a way which fits the basic paradigm of parallel execution of all the rules of a given ASM. The idea is to treat the sequential execution P **seq** Q of two rules P and Q as an "atomic" action, in the same way as executing a function update $f(t_1,\ldots,t_n) := s$, and similarly for the iteration **iterate**(R) of rule R, i.e. the repeated application of sequential composition of R with itself, as long as possible. The notion of repetition yields a definition of the traditional **while** $(cond)$ R construct which is similar to its proof theoretic counterpart in [1, 9.2.1]. Whereas Abrial explicitly excludes sequencing and loop from the specification of abstract machines [1, pg. 373], we take a more pragmatic approach and define them in such a way that they can be used coherently in two ways, depending on what is needed, namely to provide black-box descriptions of abstract submachines or glass-box views of their implementation (refinement).

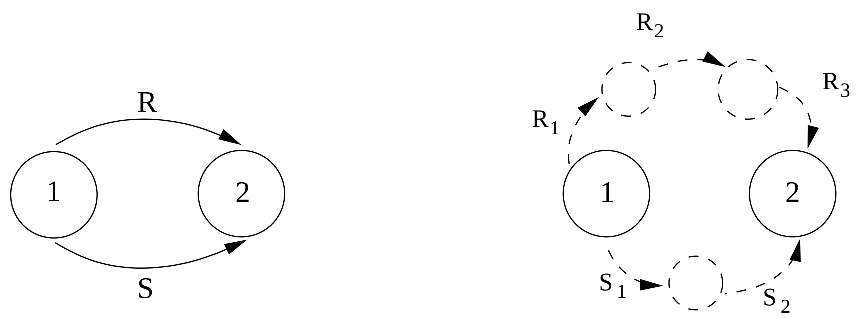
3.1 Sequence Constructor

If one wants to specify executing one standard ASM after another, this has to be explicitly programmed. Consider for example the function *pop_back* in the Standard Template Library for C++ (abstracting from concrete data structures). The function deletes the last element in a list. Assume further that we have already defined rules *move_last* and *delete* where *move_last* sets the list pointer to the last element and *delete* removes the current element. One may be tempted to program *pop_back* as follows to first execute *move_last* and then *delete*:

pop_back $\equiv$
  if *mode* $=$ *Move* **then**
    move_last
    mode $:=$ *Delete*
  if *mode* $=$ *Delete*
    delete
    mode $:=$ *Move*

This definition has the drawback that the user of *pop_back* must know that the action to be completed needs two steps, which really is an implementation feature. Moreover the dynamic function *mode*, which is used to *program* the sequential ordering, is supposed to be initialized by *Move*. Such an explicit programming of execution order quickly becomes a stumbling block for large specifications, in particular the initialization is not easily guaranteed without introducing an explicit initialization mechanism.

Another complication arises when sequentialized rules are used to refine abstract machines. In the machine on the left side of the picture below, assume that the simultaneous execution of the two rules R and S in state 1 leads to state 2. The machine on the right side is supposed to refine the machine on the left side with rules R and S refined into the sequence of rules $R_1 R_2 R_3$ and $S_1 S_2$ respectively. There is no obvious general scheme to interleave the R_i-rules and the S_j-rules, using a *mode* function as above. What should happen if rule R_2 modifies some locations which are read by S_2? In such cases R and S could not be refined independently of each other.



Therefore we introduce a sequence constructor yielding a rule P **seq** Q which can be inserted into another ASM but whose semantical effect is nevertheless the sequential execution of the two rules P and Q. If the new rule P **seq** Q has to share the same status as any other ASM rule together with which it may be executed in parallel, one can define the execution of P **seq** Q only as an atomic action. Obviously this is only a way to "view" the sequential machine from outside; its refined view reveals its internal structure and behavior, constituted by the non atomic execution, namely in two steps, of first P and then Q.

Syntactically the sequential composition P **seq** Q of two rules P and Q is defined to be a rule. The semantics is defined as first executing P, obtaining an intermediate state, followed by executing Q in the intermediate state. This is formalized by the following definition of the update set of P **seq** Q in state $\mathfrak{A}$.

Semantics: Let P and Q be rules. We define

$$[\![P \textbf{ seq } Q]\!]^{\mathfrak{A}} = [\![P]\!]^{\mathfrak{A}} \oplus [\![Q]\!]^{\mathfrak{A}'}$$

where $\mathfrak{A}' = \mathit{next}_P(\mathfrak{A})$ is the state obtained by firing the update set of P in state $\mathfrak{A}$, if this is defined; otherwise $\mathfrak{A}'$ can be chosen arbitrarily. The operator $\oplus$ denotes the merging for update sets.

The merging of two update sets u and v by the operator $\oplus$ reflects that an update in v overwrites an update in u if it is for the same location, since through a destructive assignment $s := t$ the previous value of s is lost. We merge an update set v with u (i.e. $u \oplus v$) only if u is consistent, otherwise we stick to u because then we want both $\mathit{fire}_{\mathfrak{A}}(u)$ and $\mathit{fire}_{\mathfrak{A}}(u \oplus v)$ to be undefined.

$$u \oplus v = \begin{cases} \{(\mathit{loc}, \mathit{val}) \mid (\mathit{loc}, \mathit{val}) \in u \wedge \mathit{loc} \notin \mathit{Locs}(v)\} \cup v, & \mathit{consistent}(u) \\ u, & \textbf{otherwise} \end{cases}$$

Proposition 1. (Persistence of inconsistency)

If $[\![P]\!]^{\mathfrak{A}}$ is not consistent, then $[\![P \textbf{ seq } Q]\!]^{\mathfrak{A}} = [\![P]\!]^{\mathfrak{A}}$

The next proposition shows that the above definition of the **seq** constructor captures the intended classical meaning of sequential composition of machines, if we look at them as state transforming functions[1]. Indeed we could have defined **seq** via the composition of algebra transforming functions, similarly to its axiomatically defined counterpart in Abrial's AMN [1] where **seq** comes as concatenation of generalized substitutions.

Proposition 2. (Compositionality of **seq**)

$$\mathit{next}(P \textbf{ seq } Q) = \mathit{next}(Q) \circ \mathit{next}(P)$$

This characterization illustrates that **seq** has the expected semiring properties on update sets.

Proposition 3. The ASM constructor **seq** has a left and a right neutral element and is associative, i.e. for rules P, Q, and R the following holds:

$$[\![\textbf{skip seq } R]\!]^{\mathfrak{A}} = [\![R \textbf{ seq skip}]\!]^{\mathfrak{A}} = [\![R]\!]^{\mathfrak{A}}$$

$$[\![P \textbf{ seq } (Q \textbf{ seq } R)]\!]^{\mathfrak{A}} = [\![(P \textbf{ seq } Q) \textbf{ seq } R]\!]^{\mathfrak{A}}$$

[1] We assume that $f(x)$ is undefined if x is undefined, for every function f (f is strict).

3.2 Iteration Constructor

Once a sequence operator is defined, one can apply it repeatedly to define the iteration of a rule. This provides a natural way to define for ASMs an iteration construct which encapsulates a computation with a finite but a priori not explicitly known number of iterated steps into an atomic action (one-step computation). As a by-product we obtain the classical loop and while constructs, cf. [1, 9.2].

The intention of rule iteration is to execute the given rule again and again – as long as *needed* and as long as *possible*. We define

$$R^n = \begin{cases} \textbf{skip}, & n = 0 \\ R^{n-1}\ \textbf{seq}\ R, & n > 0 \end{cases}$$

Denote by $\mathfrak{A}_n$ the state obtained by firing the update set of the rule R^n in state $\mathfrak{A}$, if defined (i.e. $\mathfrak{A}_n = next_{R^n}(\mathfrak{A})$).

There are two natural stop situations for iterated ASM rule application, namely when the update set becomes empty (the case of successful termination) and when it becomes inconsistent (the case of failure, given the persistence of inconsistency as formulated in Proposition 1).[2] Both cases provide a fixpoint $\lim_{n\to\infty} [\![R^n]\!]^{\mathfrak{A}}$ for the sequence $([\![R^n]\!]^{\mathfrak{A}})_{n>0}$ which becomes constant if a number n is found where the update set of R, in the state obtained by firing R^{n-1}, is empty or inconsistent.

Proposition 4. (Fixpoint Condition)

> $\forall m \geq n > 0$ the following holds:
> if $[\![R]\!]^{\mathfrak{A}_{n-1}}$ is not consistent or if it is empty, then $[\![R^m]\!]^{\mathfrak{A}} = [\![R^n]\!]^{\mathfrak{A}}$

Therefore we extend the syntax of ASM rules by **iterate**(R) to denote the iteration of rule R and define its semantics as follows.

Semantics: Let R be a rule. We define

$$[\![\textbf{iterate}(R)]\!]^{\mathfrak{A}} = \lim_{n\to\infty} [\![R^n]\!]^{\mathfrak{A}}, \quad \text{if } \exists\, n \geq 0 : [\![R]\!]^{\mathfrak{A}_n} = \emptyset \vee \neg consistent([\![R]\!]^{\mathfrak{A}_n})$$

The sequence $([\![R^n]\!]^{\mathfrak{A}})_{n>0}$ eventually becomes constant only upon termination or failure. Otherwise the computation diverges and the update set for the iteration is undefined. An example for a machine R which naturally produces a diverging (though in other contexts useful) computation is **iterate**$(a := a + 1)$, see [16, Exl. 2, pg. 350].

[2] We do not include here the case of an update set whose firing does not change the given state, although including this case would provide an alternative stop criterion which is also viable for implementations of ASMs.

Example 1. (Usage of **iterate** for starting the Java class initialization process)

The ASM model for Java in [9] includes the initialization of classes which in Java is done implicitly at the first use of a class. Since the Java specification requires that the superclass of a class c is initialized before c, the starting of the class initialization is iterated until an initialized class c' is encountered (i.e. satisfying $\mathit{initialized}(c')$, as eventually will happen towards the top of the class hierarchy). We define the initialization of class *class* as follows:

$$
\begin{array}{l}
\mathit{initialize} \equiv \\
\quad c := \mathit{class}\ \mathbf{seq}\ \mathbf{iterate}(\mathbf{if}\ \neg \mathit{initialized}(c)\ \mathbf{then} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{createInitFrame}(c) \\
\qquad\qquad\qquad\qquad\qquad \mathbf{if}\ \neg \mathit{initialized}(\mathit{superClass}(c))\ \mathbf{then} \\
\qquad\qquad\qquad\qquad\qquad\quad c := \mathit{superClass}(c))
\end{array}
$$

The finiteness of the acyclic class hierarchy in Java guarantees that this rule yields a well defined update set. The rule abstracts from the standard sequential implementation (where obviously the class initialization is started in a number of steps depending on how many super classes the given class has which are not yet initialized) and offers an atomic operation to push all initialization methods in the right order onto the frame stack.

The macro to create new initialization frames can be defined as follows. The current computation state, consisting of *method*, *program*, program position *pos* and *localVars*, is pushed onto the *frames* stack and is updated for starting the initialization method of the given *class* at position 0 with empty local variables set.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{createInitFrame}(c) \equiv} \\
\quad \mathit{classState}(c) & := \mathit{InProgress} \\
\quad \mathit{frames} & := \mathit{frames} \cdot (\mathit{method}, \mathit{program}, \mathit{pos}, \mathit{localVars}) \\
\quad \mathit{method} & := c/\texttt{<clinit>} \\
\quad \mathit{program} & := \mathit{body}(c/\texttt{<clinit>}) \\
\quad \mathit{pos} & := 0 \\
\quad \mathit{localVars} & := \emptyset
\end{array}
$$

While and Until. The iteration yields a natural definition of classical loop and while constructs. A *while loop* repeats the execution of the *while body* as long as a certain condition holds.

$$\mathbf{while}\ (\mathit{cond})\ R = \mathbf{iterate}(\mathbf{if}\ \mathit{cond}\ \mathbf{then}\ R)$$

This *while* loop, if started in state $\mathfrak{A}$, terminates if eventually $[\![R]\!]^{\mathfrak{A}_n}$ becomes empty or the condition *cond* becomes *false* in $\mathfrak{A}_n$ (with consistent and non empty previous update sets $[\![R]\!]^{\mathfrak{A}_i}$ and previous states $\mathfrak{A}_i$ satisfying *cond*). If the iteration of R reaches an inconsistent update set (failure) or yields an infinite sequence of consistent non empty update sets, then the state resulting from executing the while loop starting in $\mathfrak{A}$ is not defined (divergence of the while loop). Note that the function $next(\mathbf{while}\ (cond)\, R)$ is undefined in these two cases on $\mathfrak{A}$.

A *while* loop may satisfy more than one of the above conditions, like **while** (*false*) **skip**. The following examples illustrate the typical four cases:

- (success) **while** (*cond*) **skip**
- (success) **while** (*false*) R
- (failure) **while** (*true*) $a := 1$
 $a := 2$
- (divergence) **while** (*true*) $a := a$

Example 2. (Usage of **while**)

The following iterative ASM defines a while loop to compute the factorial function for given argument x and stores the result in a location *fac*. It uses multiplication as given (static) function. We will generalize this example in the next section to an ASM analogue to the Böhm-Jacopini theorem on structured programming [3].

$$\begin{aligned} compute\_fac \equiv (fac := 1)\ \mathbf{seq}\ (\mathbf{while}\ (x > 0)\ & fac := x * fac \\ & x \quad := x - 1) \end{aligned}$$

Remark: As usual one can define the *until loop* in terms of **while** and **seq** as first executing the body once and then behaving like a while loop:

$$\mathbf{do}\ R\ \mathbf{until}\ (cond) = R\ \mathbf{seq}\ (\mathbf{while}\ (\neg cond)\ R).$$

The sequencing and iteration concepts above apply in particular to the *Mealy-ASMs* defined in [4] for which they provide the sequencing and the feedback operators. The fundamental parallel composition of ASMs provides the concept of parallel composition of Mealy automata for free. These three constructs allow one to apply to Mealy-ASMs the decomposition theory which has been developed for finite state machines in [10].

3.3 Böhm-Jacopini ASMs

The sequential and iterative composition of ASMs yields a class of machines which are known from [3] to be appropriate for the computation of partial recursive functions. We illustrate in this section how these *Böhm-Jacopini-ASMs* naturally combine the advantages of the Gödel-Herbrand style functional definition of computable functions and of the Turing style imperative description of their computation.

Let us call Böhm-Jacopini-ASM any ASM which can be defined, using the sequencing and the iterator constructs, from basic ASMs whose functions are restricted as defined below to input, output, controlled functions and some simple static functions. For each Böhm-Jacopini-ASM M we allow only one external function, a 0-ary function for which we write in_M. The purpose of this function is to contain the number sequence which is given as input for the computation of the machine. Similarly we write out_M for the unique (0-ary) function which will be used to receive the output of M. Adhering to the usual practice one may also require that the M-output function appears only on the left hand side of M-updates, so that it does not influence the M-computation and is not influenced by the environment of M. As static functions we admit only the *initial* functions of recursion theory, i.e. the following functions from Cartesian products of natural numbers into the set $\mathbb{N}$ of natural numbers: $+1$, all the projection functions U_i^n, all the constant functions C_i^n and the characteristic function of the predicate $\neq 0$.

Following the standard definition we call a number theoretic function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ computable by an ASM M if for every n-tuple $x \in \mathbb{N}^n$ of arguments on which f is defined, the machine started with input x terminates with output $f(x)$. By "M started with input x" we mean that M is started in the state where all the dynamic functions different from in_M are completely undefined and where $in_M = x$. Assuming the external function in_M not to change its value during an M-computation, it is natural to say that M terminates in a state with output y, if in this state out_M gets updated for the first time, namely to y.

Proposition 5. (Structured Programming Theorem)

> Every partial recursive function can be computed by a Böhm-Jacopini-ASM.

Proof. We define by induction for each partial recursive function f a machine F computing it. Each initial function f of recursion theory is computed by the following machine F consisting of only one function update which reflects the defining equation of f.

$$F \equiv out_F := f(in_F)$$

For the inductive step it suffices to construct, for any partial recursive definition of a function f from its constituent functions f_i, a machine F which mimics the standard evaluation procedure underlying that definition. We define the following macros for using a machine F for given arguments in, possibly including to assign its output to a location out:

$$\begin{array}{ll} F(in) & \equiv in_F := in \ \mathbf{seq}\ F \\ out := F(in) & \equiv F(in)\ \mathbf{seq}\ out := out_F \end{array}$$

We start with the case of function composition. If functions $g, h_1, \ldots, h_m$ are computed by Böhm-Jacopini-ASMs $G, H_1, \ldots, H_m$, then their composition f defined by $f(x) = g(h_1(x), \ldots, h_m(x))$ is computed by the following machine[3] F:

$$F \equiv \{H_1(in_F), \ldots, H_m(in_F)\}\ \mathbf{seq}\ out_F := G(out_{H_1}, \ldots, out_{H_m})$$

[3] For reasons of simplicity but without loss of generality we assume that the submachines have pairwise disjoint signatures.

Unfolding this structured program reflects the order one *has* to follow for evaluating the subterms in the defining equation for f, an order which is implicitly assumed in the equational (functional) definition. First the input is passed to the constituent functions h_i to compute their values, whereby the input functions of H_i become controlled functions of F. The parallel composition of the submachines $H_i(in_F)$ reflects that any order is allowed here. Then the sequence of out_{H_i} is passed as input to the constituent function g. Finally g's value on this input is computed and assigned as output to out_F.

Similarly let a function f be defined from g, h by primitive recursion:

$$f(x, 0) = g(x), \quad f(x, y+1) = h(x, y, f(x, y))$$

and let Böhm-Jacopini-ASMs G, H be given which compute g, h. Then the following machine F computes f, composed as sequence of three submachines. The start submachine of F evaluates the first defining equation for f by initializing the recursor *rec* to 0 and the intermediate value *ival* to $g(x)$. The *while* submachine evaluates the second defining equation for f for increased values of the recursor as long as the input value y has not been reached. The output submachine provides the final value of *ival* as output.

$$\begin{aligned} F \equiv\ & \textbf{let } (x, y) = in_F \textbf{ in} \\ & \{ival := G(x),\ rec := 0\}\ \textbf{seq} \\ & (\textbf{while } (rec < y)\ \{ival := H(x, rec, ival),\ rec := rec + 1\})\ \ \textbf{seq} \\ & out_F := ival \end{aligned}$$

If f is defined from g by the μ-operator, i.e. $f(x) = \mu y(g(x, y) = 0)$, and if a Böhm-Jacopini-ASM G computing g is given, then the following machine F computes f. The start submachine computes $g(x, rec)$ for the initial recursor value 0, the iterating machine computes $g(x, rec)$ for increased values of the recursor until 0 shows up as computed value of g, in which case the reached recursor value is set as output.

$$\begin{aligned} F \equiv\ & \{G(in_F, 0),\ rec := 0\}\ \textbf{seq} \\ & (\textbf{while } (out_G \neq 0)\ \{G(in_F, rec + 1),\ rec := rec + 1\})\ \ \textbf{seq} \\ & out_F := rec \end{aligned}$$

Remark. The construction of Böhm-Jacopini-ASMs illustrates, through the idealized example of computing recursive functions, how ASMs allow to pragmatically reconcile the often discussed conceptual dichotomy between functional and imperative programming. In the context of discussing the "functional programming language" Gödel used to exhibit undecidable propositions in Principia Mathematica, as opposed to the "imperative programming language" developed by Turing and used in his proof of the unsolvability of the *Entscheidungsproblem* (see [7]), Martin Davis [12] states:

> "The programming languages that are mainly in use in the software industry (like C and FORTRAN) are usually described as being *imperative*. This is because the successive lines of programs written in these

> languages can be thought of as *commands* to be executed by the computer ... In the so-called *functional* programming languages (like LISP) the lines of a program are definitions of operations. Rather than telling the computer what to do, they *define* what it is that the computer is to provide."

The equations which appear in the Gödel-Herbrand type equational definition of partial recursive functions "define what it is that the computer is to provide" only within the environment for evaluation of subterms. The corresponding Böhm-Jacopini-ASMs constructed above make this context explicit, exhibiting how to evaluate the subterms when using the equations (updates), as much as needed to make the functional shorthand work correctly. We show in the next section how this use of shorthands for calling submachines, which appear here only in the limited context of structured WHILE programs, can be generalized as to make it practical without loss of rigor.

4 Parameterized Machines

For structuring large ASMs extensive use has been made of macros which, semantically speaking, are mere notational shorthands, to be substituted by the body of their definition. We enhance this use here by introducing named parameterized ASM rules which in contrast to macros also support recursive ASMs.

We provide a foundation which justifies the application of named parameterized ASMs in a way which supports the practitioners' procedural understanding. Instead of guaranteeing within the theory, typically through a fixpoint operator, that under certain conditions iterated calls of recursive rules yield as "result" a first-class mathematical "object" (namely the fixpoint), we take inspiration from the way Kleene proved his recursion theorem [16, Section 66] and leave it to the programmer to guarantee that a possibly infinite chain of recursive procedure calls is indeed well founded with respect to some partial order.

We want to allow a named parameterized rule to be used in the same way as all other rules. For example, if f is a function with arity 1 and R is a named rule expecting two parameters, then $R(f(1), 2)$ should be a legitimate rule, too. In particular we want to allow rules as parameters, like in the following example where the given dynamic function *stdout* is updated to "hello world":

rule $R(output) =$
  $output($`"hello world"`$)$

rule $output\_to\_stdout(msg)$
  $stdout := msg$

$R(output\_to\_stdout)$

Therefore we extend the inductive syntactic definition for rules by the following new clause, called a rule application with actual parameters $a_1, \ldots, a_n$:

$R(a_1, \ldots, a_n)$

and coming with a rule definition of the following form:

rule $R(x_1, \ldots, x_n) = \mathit{body}$

where *body* is a rule. R is called the rule name, $x_1, \ldots, x_n$ are the formal parameters of the rule definition. They bind the free occurrences of the variables $x_1, \ldots, x_n$ in *body*.

The basic intuition the practice of computing provides for the interpretation of a named rule is to define its semantics as the interpretation of the rule body with the formal parameters replaced by the actual arguments. In other words we unfold nested calls of a recursive rule R into a sequence $R_1, R_2, \ldots$ of rule incarnations where each R_i may trigger one more execution of the rule body, relegating the interpretation of possibly yet another call of R to the next incarnation R_{i+1}. This may produce an infinite sequence, namely if there is no ordering of the procedure calls with respect to which the sequence will decrease and reach a basis for the recursion. In this case the semantics of the call of R is undefined. If however a basis for the recursion does exist, say R_n, it yields a well defined value for the semantics of R through the chain of successive calls of R_i; namely for each $0 \leq i < n$ with $R = R_0$, R_i inherits its semantics from R_{i+1}.

Semantics: Let R be a named rule declared by **rule** $R(x_1, \ldots, x_n) = \mathit{body}$, let $\mathfrak{A}$ be a state.

If $[\![\mathit{body}[a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}}$ is defined, then
$[\![R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}} = [\![\mathit{body}[a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}}$

For the rule definition **rule** $R(x) = R(x)$ this interpretation yields no value for any $[\![R(a)]\!]^{\mathfrak{A}}$, see [16, Example 1, page 350]. In the following example the update set for $R(x)$ is defined for all $x \leq 10$, with the empty set as update set, and is not defined for any $x > 10$.

rule $R(x) =$ **if** $x < 10$ **then** $R(x+1)$
if $x = 10$ **then skip**
if $x > 10$ **then** $R(x+1)$

Example 3. (Defining **while** by a named rule)

Named rules allow us to define the *while loop* recursively instead of iteratively:

rule $\mathit{while}(\mathit{cond}, R) =$
if *cond* **then**
R **seq** $\mathit{while}(\mathit{cond}, R)$

This recursively defined **while** operator behaves differently from the iteratively defined **while** of the preceding section in that it leads to termination only if the condition *cond* will become eventually false, and not in the case that eventually the update set of R becomes empty. For example the semantics of the recursively defined $\mathit{while}(\mathit{true}, \mathbf{skip})$ is not defined.

Example 4. (Starting Java class initialization)

We can define the Java class initialization of Example 1 also in terms of a recursive named rule, avoiding the local input variable to which the actual parameter is assigned at the beginning.

rule *initialize*(c) =
  if *initialized*(*superClass*(c)) **then**
    createInitFrame(c)
  else
    createInitFrame(c) **seq** *initialize*(*superClass*(c))

Remark: Iterated execution of (sub)machines R, started in state $\mathfrak{A}$, unavoidably leads to possibly undefined update sets $[\![R]\!]^{\mathfrak{A}}$. As a consequence $[\![R]\!]^{\mathfrak{A}} = [\![S]\!]^{\mathfrak{A}}$ denotes that either both sides of the equation are undefined or both are defined and indeed have the same value. In the definitions above we adhered to an algorithmic definition of $[\![R]\!]^{\mathfrak{A}}$, namely by computing its value from the computed values $[\![S]\!]^{\mathfrak{A}}$ of the submachines S of R. In the appendix we give a deduction calculus for proving statements $[\![R]\!]^{\mathfrak{A}} = u$ meaning that $[\![R]\!]^{\mathfrak{A}}$ is defined and has value u.

5 Further Concepts

In this section we enrich named rules with a notion of local state, show how parameterized ASMs can be used as machines with return value, and introduce error handling for ASMs which is an abstraction of exception handling as found in modern programming languages.

5.1 Local State

Basic ASMs come with a notion of state in which all the dynamic functions are global. The use of only locally visible parts of the state, like variables declared in a class, can naturally be incorporated into named ASMs. It suffices to extend the definition of named rules by allowing some dynamic functions to be declared as local, meaning that each call of the rule works with its own incarnation of local dynamic functions f which are to be initialized upon rule invocation by an initialization rule $Init(f)$. Syntactically we allow definitions of named rules of the following form:

rule $name(x_1, \ldots, x_n)$ =
  local $f_1[Init_1]$
  $\vdots$
  local $f_k[Init_k]$
  body

where *body* and $Init_i$ are rules. The formal parameters $x_1, \ldots, x_n$ bind the free occurrences of the corresponding variables in *body* and $Init_i$. The functions

$f_1, \ldots, f_k$ are treated as local functions whose scope is the rule where they are introduced. They are not part of the signature of the ASM. Init_i is a rule used for the initialization of f_i. We write **local** $f := t$ for **local** $f[f := t]$.

For the semantic interpretation of a call of a rule with local dynamic functions, the updates to the local functions are collected together with all other function updates made through executing the body. This includes the updates required by the initialization rules. The restriction of the scope of the local functions to the rule definition is obtained by then removing from the update set u, which is available after the execution of the body of the call, the set $\mathit{Updates}(f_1, \ldots, f_k)$ of updates concerning the local functions $f_1, \ldots, f_k$. This leads to the following definition.

Semantics: Let R be a rule declaration with local functions as given above. If the right side of the equation is defined, we set:

$$\begin{aligned}&[\![R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}} = \\ &\quad [\![(\{\mathit{Init}_1, \ldots, \mathit{Init}_k\}\ \mathbf{seq}\ \mathit{body})[a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}} \setminus \mathit{Updates}(f_1, \ldots, f_k)\end{aligned}$$

We assume that there are no name clashes for local functions between different incarnations of the same rule (i.e. each rule incarnation has its own set of local dynamic functions).

Example 5. (Usage of local dynamic functions)

The use of local dynamic functions is illustrated by the following rule computing a function f defined by a primitive recursion from functions g and h which are used here as static functions. The rule mimics the corresponding Böhm-Jacopini machine in Proposition 5.

$$\begin{aligned}&\mathbf{rule}\ F(x, y) = \\ &\quad \mathbf{local}\ \mathit{ival} := g(x) \\ &\quad \mathbf{local}\ \mathit{rec}\ := 0 \\ &\quad (\mathbf{while}\ (\mathit{rec} < y)\ \{\mathit{ival} := h(x, \mathit{rec}, \mathit{ival}),\ \mathit{rec} := \mathit{rec} + 1\})\ \ \mathbf{seq} \\ &\quad \mathit{out} := \mathit{ival}\end{aligned}$$

5.2 ASMs with Return Value

In the preceding example, for outputting purposes the value resulting from the computation is stored in a global dynamic function out. This formulation violates good information hiding principles. To store the return value of a rule R in a location which is determined by the rule caller and is independent of R, we use the following notation for a new rule:

$$l \leftarrow R(a_1, \ldots, a_n)$$

where R is a named rule with n parameters in which a 0-ary (say reserved) function *result* does occur with the intended role to store the return value. Let

rule $R(x_1, \ldots, x_n) = body$ be the declaration for R, then the semantic of $l \leftarrow R(a_1, \ldots, a_n)$ is defined as the semantics of $R_l(a_1, \ldots, a_n)$ where R_l is defined like R with *result* replaced by l:

rule $R_l(x_1, \ldots, x_n) = body[l/result]$

In the definition of the rule R by *body*, the function name *result* plays the role of a placeholder for a location, denoting the interface which is offered for communicating results from any rule execution to its caller. One can apply simultaneously two rules $l \leftarrow R(a_1, \ldots, a_n)$ and $l' \leftarrow R(a'_1, \ldots, a'_n)$ with different return values for l and l'.
Remark: When using $l \leftarrow R(a_1, \ldots, a_n)$ with a term l of form $f(t_1, \ldots, t_n)$, a good encapsulation discipline will take care that R does not modify the values of t_i, because they contribute to determine the location where the caller expects to find the return value.

Example 6. (Using return values)

Using this notation the above Example 5 becomes $f(x, y) \leftarrow F(x, y)$ where moreover one can replace the use of the auxiliary static functions g, h by calls to submachines G, H computing them, namely $ival \leftarrow G(x)$ and $ival \leftarrow H(x, rec, ival)$.

Example 7. (Recursive machine computing the factorial function, using multiplication as static function.)

rule $Fac(n) =$
  local $x := 1$
  if $n = 1$ **then**
    $result := 1$
  else
    $(x \leftarrow Fac(n-1))$ **seq** $result := n * x$

5.3 Error Handling

Programming languages like C++ or Java support exceptions to separate error handling from "normal" execution of code. Producing an inconsistent update set is an abstract form of throwing an exception. We therefore introduce a notion of catching an inconsistent update set and of executing error code.

The semantics of **try** R **catch** $f(t_1, \ldots, t_n)$ S is the update set of R if either this update set is consistent ("normal" execution) or it is inconsistent but the location *loc* determined by $f(t_1, \ldots, t_n)$ is not updated inconsistently. Otherwise it is the update set of S.

Since the rule enclosed by the **try** block is executed either completely or not at all, there is no need for any **finally** clause to remove trash.

Semantics: Let R and S be rules, f a dynamic function with arguments $t_1, \ldots, t_n$. We define

$$[\![\textbf{try}\ R\ \textbf{catch}\ f(t_1,\ldots,t_n)\ S]\!]^{\mathfrak{A}} = \begin{cases} v, & \exists\, v_1 \neq v_2 : (loc, v_1) \in u \wedge (loc, v_2) \in u \\ u, & \textbf{otherwise} \end{cases}$$

where $u = [\![R]\!]^{\mathfrak{A}}$ and $v = [\![S]\!]^{\mathfrak{A}}$ are the update sets of R and S respectively, and loc is the location $f\langle [\![t_1]\!]^{\mathfrak{A}}, \ldots, [\![t_n]\!]^{\mathfrak{A}} \rangle$.

6 Related Work

The sequence operator defined by Zamulin in [19] differs from our concept for rules leading to inconsistent update sets where it is not associative, due to Zamulin's definition of the merge operator for update sets. For consistent update sets Zamulin's loop constructor coincides with our while definition in Example 2.

In Anlauff's XASM [2], calling an ASM is the iteration of a rule until a certain condition holds. [2] provides no formal definition of this concept, but for consistent update sets the XASM implementation seems to behave like our definition of *iterate*.

Named rules with parameters appear in the ASM Workbench [11] and in XASM [2], but with parameters restricted to terms. The ASM Workbench does not allow recursive rules. Recursive ASMs have also been proposed by Gurevich and Spielmann [15]. Their aim was to justify recursive ASMs within distributed ASMs [13]. If R is a rule executed by agent a and has two recursive calls to R, then a creates two new agents a_1 and a_2 which execute the two corresponding recursive calls. The agent a waits for termination of his slaves a_1 and a_2 and then combines the result of both computations. This is different from our definition where executing a recursive call needs only one step, from the caller's view, so that the justification remains within purely sequential ASMs without invoking concepts from distributed computing. Through our definition the distinction between suspension and reactivation tasks in the iterative implementation of recursion becomes a matter of choosing the black-box or the glass-box view for the recursion. The updates of a recursive call are collected and handed over to the calling machine as a whole to determine the state following in the black-box view the calling state. Only the glass-box view provides a refined inspection of how this collection is computed.

Acknowledgment. For critical comments on earlier versions of this paper[4] we thank Giuseppe Del Castillo, Martin Davis, Jim Huggins, Alexander Knapp, Peter Päppinghaus, Robert Stärk, Margus Vianes, and Alexandre Zamulin.

[4] presented to the IFIP Working Group 1.3 on Foundations of System Specification, Bonas (France) 13.-15.9.1999, and to the International ASM'2000 Workshop, Ascona (Switzerland) 20.-24.3.2000

A Deduction Rules for Computing Update Sets

The following rules provide a calculus for computing the semantics of standard ASMs and for the constructs introduced in this paper.

We use R, R_i, and S for rules, f for functions, x for variables, s and t for expressions, p for predicates (boolean expressions), and u and v for semantical values and update sets.

Standard ASMs

$$\frac{\forall\, i : [\![t_i]\!]_\zeta^{\mathfrak{A}} = v_i}{[\![f(t_1,\ldots,t_n)]\!]_\zeta^{\mathfrak{A}} = f^{\mathfrak{A}}(v_1,\ldots,v_n)} \qquad \frac{}{[\![x]\!]_\zeta^{\mathfrak{A}} = \zeta(x)}\ \mathit{variable}(x)$$

$$\frac{}{[\![\mathbf{skip}]\!]_\zeta^{\mathfrak{A}} = \emptyset} \qquad \frac{[\![t]\!]_\zeta^{\mathfrak{A}} = \mathrm{true}^{\mathfrak{A}},\ [\![R]\!]_\zeta^{\mathfrak{A}} = u}{[\![\mathbf{if}\ t\ \mathbf{then}\ R\ \mathbf{else}\ S]\!]_\zeta^{\mathfrak{A}} = u}$$

$$\frac{\forall\, i : [\![t_i]\!]_\zeta^{\mathfrak{A}} = v_i,\ [\![s]\!]_\zeta^{\mathfrak{A}} = u}{[\![f(t_1,\ldots,t_n) := s]\!]_\zeta^{\mathfrak{A}} = \{(f\langle v_1,\ldots,v_n\rangle, u)\}} \qquad \frac{[\![t]\!]_\zeta^{\mathfrak{A}} = \mathrm{false}^{\mathfrak{A}},\ [\![S]\!]_\zeta^{\mathfrak{A}} = u}{[\![\mathbf{if}\ t\ \mathbf{then}\ R\ \mathbf{else}\ S]\!]_\zeta^{\mathfrak{A}} = u}$$

$$\frac{\forall\, i : [\![R_i]\!]_\zeta^{\mathfrak{A}} = u_i}{[\![\{R_1,\ldots,R_n\}]\!]_\zeta^{\mathfrak{A}} = u_1 \cup \ldots \cup u_n} \qquad \frac{[\![t]\!]_\zeta^{\mathfrak{A}} = v,\ [\![R]\!]_{\zeta\frac{x}{v}}^{\mathfrak{A}} = u}{[\![\mathbf{let}\ x = t\ \mathbf{in}\ R]\!]_\zeta^{\mathfrak{A}} = u}$$

$$\frac{V = \{v_1,\ldots,v_n\},\ \forall\, i : [\![R]\!]_{\zeta\frac{x}{v_i}}^{\mathfrak{A}} = u_i}{[\![\mathbf{forall}\ x\ \mathbf{with}\ p\ \mathbf{do}\ R]\!]_\zeta^{\mathfrak{A}} = u_1 \cup \ldots \cup u_n}\ V = \{v \mid [\![p]\!]_{\zeta\frac{x}{v}}^{\mathfrak{A}} = \mathrm{true}^{\mathfrak{A}}\}$$

$$\frac{[\![p]\!]_{\zeta\frac{x}{v}}^{\mathfrak{A}} = \mathrm{true}^{\mathfrak{A}},\ [\![R]\!]_{\zeta\frac{x}{v}}^{\mathfrak{A}} = u}{[\![\mathbf{choose}\ x\ \mathbf{with}\ p\ \mathbf{do}\ R]\!]_\zeta^{\mathfrak{A}} = u}$$

$$\frac{}{[\![\mathbf{choose}\ x\ \mathbf{with}\ p\ \mathbf{do}\ R]\!]_\zeta^{\mathfrak{A}} = \emptyset}\ \nexists\, v : [\![p]\!]_{\zeta\frac{x}{v}}^{\mathfrak{A}} = \mathrm{true}^{\mathfrak{A}}$$

Sequential composition

$$\frac{[\![R]\!]_\zeta^{\mathfrak{A}} = u,\ [\![S]\!]_\zeta^{\mathit{fire}_{\mathfrak{A}}(u)} = v}{[\![R\ \mathbf{seq}\ S]\!]_\zeta^{\mathfrak{A}} = u \oplus v}\ \mathit{consistent}(u) \qquad \frac{[\![R]\!]_\zeta^{\mathfrak{A}} = u}{[\![R\ \mathbf{seq}\ S]\!]_\zeta^{\mathfrak{A}} = u}\ \mathit{inconsistent}(u)$$

Iteration

$$\frac{[\![R^n]\!]_\zeta^{\mathfrak{A}} = u}{[\![\mathbf{iterate}(R)]\!]_\zeta^{\mathfrak{A}} = u}\ n \geq 0, \mathit{inconsistent}(u)$$

$$\frac{[\![R^n]\!]_\zeta^{\mathfrak{A}} = u,\ [\![R]\!]_\zeta^{\mathit{fire}_{\mathfrak{A}}(u)} = \emptyset}{[\![\mathbf{iterate}(R)]\!]_\zeta^{\mathfrak{A}} = u}\ n \geq 0, \mathit{consistent}(u)$$

Parameterized Rules with local state
Let R be a named rule as in Section 5.1.

$$\frac{[\![(\{\mathit{Init}_1, \ldots, \mathit{Init}_k\} \ \mathbf{seq} \ \mathit{body})[a_1/x_1, \ldots, a_n/x_n]]\!]_\zeta^{\mathfrak{A}} = u}{[\![R(a_1, \ldots, a_n)]\!]_\zeta^{\mathfrak{A}} = u \setminus \mathit{Updates}(f_1, \ldots, f_k)}$$

Error Handling

$$\frac{[\![R]\!]_\zeta^{\mathfrak{A}} = u}{[\![\mathbf{try} \ R \ \mathbf{catch} \ f(t_1, \ldots, t_n) \ S]\!]_\zeta^{\mathfrak{A}} = u} \quad \begin{array}{l} \nexists \, v_1 \neq v_2 : (\mathit{loc}, v_1) \in u \wedge (\mathit{loc}, v_2) \in u \\ \mathit{where} \ \ \mathit{loc} = f\langle [\![t_1]\!]_\zeta^{\mathfrak{A}}, \ldots, [\![t_n]\!]_\zeta^{\mathfrak{A}} \rangle \end{array}$$

$$\frac{[\![R]\!]_\zeta^{\mathfrak{A}} = u, \ [\![S]\!]_\zeta^{\mathfrak{A}} = v}{[\![\mathbf{try} \ R \ \mathbf{catch} \ f(t_1, \ldots, t_n) \ S]\!]_\zeta^{\mathfrak{A}} = v} \quad \begin{array}{l} \exists \, v_1 \neq v_2 : (\mathit{loc}, v_1) \in u \wedge (\mathit{loc}, v_2) \in u \\ \mathit{where} \ \ \mathit{loc} = f\langle [\![t_1]\!]_\zeta^{\mathfrak{A}}, \ldots, [\![t_n]\!]_\zeta^{\mathfrak{A}} \rangle \end{array}$$

Remark: The second rule for *choose* reflects the decision in [13] that an ASM does nothing when there is no choice. Obviously also other decisions could be formalized in this manner, e.g. yielding instead of the empty set an update set which contains an error report.

Remark: The rule for *forall* is formulated as finitary rule, i.e. it can be applied only for quantifying over finite sets. The set theoretic formulation in Section 2 is more general and can be formalized by an infinitary rule. It would be quite interesting to study different classes of ASMs, corresponding to different finitary or infinitary versions of the *forall* construct.

References

1. J. R. Abrial. *The B-Book. Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. M. Anlauff. XASM – An extensible, component-based Abstract State Machines language. In Y. Gurevich, M. Odersky, and L. Thiele, editors, *Proc. ASM 2000*, Lecture Notes in Computer Science. Springer-Verlag, 2000. to appear.
3. C. Böhm and G. Jacopini. Flow diagrams, Turing Machines, and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.
4. E. Börger. High level system design and analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in Lecture Notes in Computer Science, pages 1–43. Springer-Verlag, 1999.
5. E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML Activity Diagrams. In T. Rust, editor, *Proc. AMAST 2000*, Lecture Notes in Computer Science. Springer-Verlag, 2000.
6. E. Börger, A. Cavarra, and E. Riccobene. A simple formal model for UML State Machines. In Y. Gurevich, M. Odersky, and L. Thiele, editors, *Proc. ASM 2000*, Lecture Notes in Computer Science. Springer-Verlag, 2000. to appear.
7. E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem.* Perspectives in Mathematical Logic. Springer-Verlag, 1997.
8. E. Börger, J. Schmid, W. Schulte, and R. Stärk. *Java and the Java Virtual Machine.* Lecture Notes in Computer Science. Springer-Verlag, 2000. to appear.
9. E. Börger and W. Schulte. Modular Design for the Java Virtual Machine Architecture. In E. Börger, editor, *Architecture Design and Validation Methods*, pages 297–357. Springer-Verlag, 2000.
10. A. Brüggemann, L. Priese, D. Rödding, and R. Schätz. Modular decomposition of automata. In E. Börger, G. Hasenjäger, and D. Rödding, editors, *Logic and Machines: Decision Problems and Complexity*, number 171 in Lecture Notes in Computer Science, pages 198–236. Springer-Verlag, 1984.
11. G. D. Castillo. *ASM-SL, a Specification Language based on Gurevich's Abstract State Machines*, 1999.
12. M. Davis. *The Universal Computer: The Road from Leibniz to Turing.* W.W. Norton, New York, 2000. to appear.
13. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
14. Y. Gurevich. Sequential Abstract State Machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1), 2000.
15. Y. Gurevich and M. Spielmann. Recursive abstract state machines. *Journal of Universal Computer Science*, 3(4):233–246, 1997.
16. S. C. Kleene. *Introduction to Metamathematics.* D. van Nostrand, Princeton, New Jersey, 1952.
17. D. L. Parnas. Information distribution aspects of design methodology. In *Information Processing 71*, pages 339–344. North Holland Publishing Company, 1972.
18. J. Schmid. Executing ASM specifications with AsmGofer. Web pages at: http://-www.tydo.de/AsmGofer, 1999.
19. A. Zamulin. Object-oriented Abstract State Machines. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science.* Technical Report, Magdeburg University, 1998.