2. Abstract State Machines

The notion of *Abstract State Machines* (ASMs), defined in [20], captures in mathematically rigorous yet transparent form some fundamental operational intuitions of computing, and the notation is familiar from programming practice and mathematical standards. This allows the practitioner to work with ASMs without any further explanation, viewing them as 'pseudocode over abstract data' which comes with a well defined semantics supporting the intuitive understanding. We therefore suggest to skip this chapter and to come back to it only should the need be felt upon further reading.

For the sake of a definite reference, we nevertheless provide in this chapter a survey of the notation, including some extensions of the definition in [20] which are introduced in [7] for structuring complex machines and for reusing machine components. For the reader who is interested in more details, we also provide a mathematical definition of the syntax and semantics of ASMs. This definition helps understanding how the ASMs in this book have been made executable, despite of their abstract nature; it will also help the more mathematically inclined reader to check the proofs in this book. We stick to non distributed (also called sequential) ASMs because they suffice for modeling Java and the JVM.

2.1 ASMs in a nutshell

ASMs are systems of finitely many *transition rules* of form

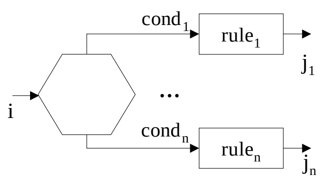
if *Condition* **then** *Updates*

which transform abstract states. (Two more forms are introduced below.) The *Condition* (so called guard) under which a rule is applied is an arbitrary first-order formula without free variables. *Updates* is a finite set of function updates (containing only variable free terms) of form

$f(t_1, \ldots, t_n) := t$

whose execution is to be understood as *changing* (or defining, if there was none) the value of the (location represented by the) function f at the given parameters.

Fig. 2.1 Control state ASM diagrams



means

```
if cond$_1$ & ctl_state = i
then ctl_state := j$_1$
     rule$_1$
  .
  .
  .
if cond$_n$ & ctl_state = i
then ctl_state := j$_n$
     rule$_n$
```

Assume disjoint cond$_i$. Usually the "control states" are notationally suppressed.

The global JVM structure is given by so called control state ASMs [3] which have finitely many control states $ctl\_state \in \{1, \ldots, m\}$, resembling the internal states of classical Finite State Machines. They are defined and pictorially depicted as shown in Fig. 2.1. Note that in a given control state i, these machines do nothing when no condition $cond_j$ is satisfied.

The notion of ASM *states* is the classical notion of mathematical *structures* where data come as abstract objects, i.e., as elements of sets (domains, *universes*, one for each category of data) which are equipped with basic operations (partial *functions*) and predicates (attributes or relations). Without loss of generality one can treat predicates as characteristic functions.

The notion of ASM *run* is the classical notion of computation of transition systems. An ASM computation step in a given state consists in executing *simultaneously* all updates of all transition rules whose guard is true in the state, if these updates are consistent. For the evaluation of terms and formulae in an ASM state, the standard interpretation of function symbols by the corresponding functions in that state is used.

Simultaneous execution provides a convenient way to abstract from irrelevant sequentiality and to make use of synchronous parallelism. This mechanism is enhanced by the following concise notation for the simultaneous execution of an ASM rule R for each x satisfying a given condition φ:

forall x **with** φ **do** R

A priori no restriction is imposed neither on the abstraction level nor on the complexity nor on the means of definition of the functions used to compute the arguments t_i and the new value t in function updates. The major distinction made in this connection for a given ASM M is between *static* functions—which never change during any run of M—and *dynamic* ones which typically do change as a consequence of updates by M or by the environment (i.e., by some other agent than M). The dynamic functions are further divided into four subclasses. *Controlled* functions (for M) are dynamic functions which are directly updatable by and only by the rules of M, i.e., functions f which appear in a rule of M as leftmost function (namely in an update $f(s) := t$ for some s, t) and are not updatable by the environment. *Monitored* functions are dynamic functions which are directly updatable by and only by the environment, i.e., which are updatable but do not appear as leftmost function in updates of M. *Interaction* functions are dynamic functions which are directly updatable by rules of M and by the environment. *Derived* functions are dynamic functions which are not directly updatable neither by M nor by the environment but are nevertheless dynamic because defined (for example by an explicit or by an inductive definition) in terms of static *and* dynamic functions.

We will use functions of all these types in this book, their use supports the principles of separation of concerns, information hiding, modularization and stepwise refinement in system design. A frequently encountered kind of static or monitored functions are choice functions, used to abstract from details of static or dynamic scheduling strategies. ASMs support the following concise notation for an abstract specification of such strategies:

choose x **with** φ **do** R

meaning to execute rule R with an arbitrary x chosen among those satisfying the selection property φ. If there exists no such x, nothing is done. For **choose** and **forall** rules we also use graphical notations of the following form:

choose x **with** φ
R

forall x **with** φ
R

We freely use as abbreviations combinations of **where**, **let**, **if then else**, **case** and similar standard notations which are easily reducible to the above basic definitions. We usually use the table like **case** notation with pattern matching and try out the cases in the order of writing, from top to bottom. We also use rule schemes, namely rules with variables and named parametrized rules, but only as an abbreviational device to enhance the readability or as macro allowing us to reuse machines and to display the global machine structure. For example

if $\ldots a = (X, Y) \ldots$
then $\ldots X \ldots Y \ldots$

abbreviates

if $\ldots \mathrm{ispair}(a) \ldots$
then $\ldots \mathrm{fst}(a) \ldots \mathrm{snd}(a) \ldots,$

sparing us the need to write explicitly the recognizers and the selectors. Similarly, an occurrence of

$r(x_1, \dots, x_n)$

where a rule is expected stands for the corresponding rule R (which is supposed to be defined somewhere, say by $r(x_1, \dots, x_n) = R$). Such a "rule call" $r(x_1, \dots, x_n)$ is used only when the parameters are instantiated by legal values (objects, functions, rules, whatever) so that the resulting rule has a well defined semantical meaning on the basis of the explanations given above.

2.2 Mathematical definition of ASMs

In this section we provide a detailed mathematical definition for the syntax and semantics of ASMs. This definition is the basis of the AsmGofer implementation of the ASMs for Java/JVM in this book.

2.2.1 Abstract states

In an ASM state, data come as abstract elements of domains (also called universes, one for each category of data) which are equipped with basic operations represented by functions. Without loss of generality we treat relations as boolean valued functions and view domains as characteristic functions, defined on the superuniverse which represents the union of all domains. Thus the states of ASMs are *algebraic structures*, also called simply *algebras*, as introduced in standard logic or universal algebra textbooks.

Definition 2.2.1 (Vocabulary). A *vocabulary* Σ is a finite collection of function names. Each function name f has an *arity*, a non-negative integer. The arity of a function name is the number of arguments the function takes. Function names can be *static* or *dynamic*. Nullary function names are often called *constants*; but be aware that, as we will see below, the interpretation of dynamic nullary functions can change from one state to the next, so that they correspond to the variables of programming. Every ASM vocabulary is assumed to contain the static constants *undef*, *True*, *False*.

Example 2.2.1. The vocabulary Σ_{bool} of Boolean algebras contains two constants 0 and 1, a unary function name '$-$' and two binary function names '$+$' and '$*$'. The vocabulary Σ_{scm} of the programming language Scheme contains a constant *nil*, two unary function names *car* and *cdr* and a binary function name *cons*, etc.

Definition 2.2.2 (State). A *state* $\mathfrak{A}$ of the vocabulary Σ is a non-empty set X, the *superuniverse* of $\mathfrak{A}$, together with *interpretations* of the function names of Σ. If f is an n-ary function name of Σ, then its interpretation $f^{\mathfrak{A}}$ is a function from X^n into X; if c is a constant of Σ, then its interpretation $c^{\mathfrak{A}}$ is an element of X. The superuniverse X of the state $\mathfrak{A}$ is denoted by $|\mathfrak{A}|$.

Example 2.2.2. Two states $\mathfrak{A}$ and $\mathfrak{B}$ for the vocabulary Σ_{bool} of Example 2.2.1: The superuniverse of the state $\mathfrak{A}$ is the set $\{0, 1\}$. The functions are interpreted as follows, where a, b are 0 or 1:

$0^{\mathfrak{A}}$	$:=$	0	(zero)
$1^{\mathfrak{A}}$	$:=$	1	(one)
$-^{\mathfrak{A}} a$	$:=$	$1 - a$	(logical complement)
$a +^{\mathfrak{A}} b$	$:=$	$\max(a, b)$	(logical or)
$a *^{\mathfrak{A}} b$	$:=$	$\min(a, b)$	(logical and)

The superuniverse of the state $\mathfrak{B}$ is the power set of the set of non-negative integers $\mathbb{N}$. The functions are interpreted as follows, where a, b are subsets of $\mathbb{N}$:

$0^{\mathfrak{B}}$	$:=$	$\emptyset$	(empty set)
$1^{\mathfrak{B}}$	$:=$	$\mathbb{N}$	(full set)
$-^{\mathfrak{B}} a$	$:=$	$\mathbb{N} \setminus a$	(set of all $n \in \mathbb{N}$ such that $n \notin a$)
$a +^{\mathfrak{B}} b$	$:=$	$a \cup b$	(set of all $n \in \mathbb{N}$ such that $n \in a$ or $n \in b$)
$a *^{\mathfrak{B}} b$	$:=$	$a \cap b$	(set of all $n \in \mathbb{N}$ such that $n \in a$ and $n \in b$)

Both states, $\mathfrak{A}$ and $\mathfrak{B}$, are so-called Boolean algebras.

Other examples of algebraic structures are: groups, rings, lattices, etc.

Remark 2.2.1. Formally, function names are interpreted in states as total functions. We view them, however, as being partial and define the *domain* of an n-ary function name f in $\mathfrak{A}$ to be the set of all n-tuples $(a_1, \dots, a_n) \in |\mathfrak{A}|^n$ such that $f^{\mathfrak{A}}(a_1, \dots, a_n) \neq \mathit{undef}^{\mathfrak{A}}$.

Example 2.2.3. In states for the vocabulary Σ_{scm} of Example 2.2.1, we usually have: $\mathit{car}^{\mathfrak{A}}(\mathit{nil}^{\mathfrak{A}}) = \mathit{undef}^{\mathfrak{A}}$, $\mathit{cdr}^{\mathfrak{A}}(\mathit{nil}^{\mathfrak{A}}) = \mathit{undef}^{\mathfrak{A}}$.

The constant *undef* represents an undetermined object, the default value of the superuniverse. It is also used to model heterogeneous domains. In applications, the superuniverse A of a state $\mathfrak{A}$ is usually divided into smaller *universes*, modeled by their characteristic functions. The universe represented by f is the set of all elements t for which $f(t) \neq \mathit{undef}$. If a unary function f represents a universe, then we simply write $t \in f$ as an abbreviation for the formula $f(t) \neq \mathit{undef}$.

Definition 2.2.3 (Term). The terms of Σ are syntactic expressions generated as follows:

1. Variables v_0, v_1, v_2, ... are terms.
2. Constants c of Σ are terms.
3. If f is an n-ary function name of Σ and $t_1, \dots, t_n$ are terms, then $f(t_1, \dots, t_n)$ is a term.

Terms are denoted by r, s, t; variables are denoted by x, y, z. A term which does not contain variables is called *closed*.

Example 2.2.4. The following are terms of the vocabulary Σ_{bool}:

$$+(v_0, v_1), \quad +(1, *(v_7, 0))$$

The are usually written as $v_0 + v_1$ and $1 + (v_7 * 0)$.

Since terms are syntactic objects, they do not have a meaning. A term can be evaluated in a state, if elements of the superuniverse are assigned to the variables of the term.

Definition 2.2.4 (Variable assignment). Let $\mathfrak{A}$ be a state. A *variable assignment* for $\mathfrak{A}$ is a function ζ which assigns to each variable v_i an element $\zeta(v_i) \in |\mathfrak{A}|$. We write $\zeta\frac{a}{x}$ for the variable assignment which coincides with ζ except that it assigns the element a to the variable x. So we have:

$$\zeta\tfrac{a}{x}(v_i) = \begin{cases} a, & \text{if } v_i = x; \\ \zeta(v_i), & \text{otherwise.} \end{cases}$$

Given a variable assignment a term can be interpreted in a state.

Definition 2.2.5 (Interpretation of terms). Let $\mathfrak{A}$ be a state of Σ, ζ be a variable assignment for $\mathfrak{A}$ and t be a term of Σ. By induction on the length of t, a value $[\![t]\!]^{\mathfrak{A}}_{\zeta} \in |\mathfrak{A}|$ is defined as follows:

1. $[\![v_i]\!]^{\mathfrak{A}}_{\zeta} := \zeta(v_i)$,
2. $[\![c]\!]^{\mathfrak{A}}_{\zeta} := c^{\mathfrak{A}}$,
3. $[\![f(t_1, \ldots, t_n)]\!]^{\mathfrak{A}}_{\zeta} := f^{\mathfrak{A}}([\![t_1]\!]^{\mathfrak{A}}_{\zeta}, \ldots, [\![t_n]\!]^{\mathfrak{A}}_{\zeta})$.

The interpretation of t depends on the values of ζ on the variables of t only: if $\zeta(x) = \xi(x)$ for all variables x of t, then $[\![t]\!]^{\mathfrak{A}}_{\zeta} = [\![t]\!]^{\mathfrak{A}}_{\xi}$ (Coincidence Lemma).

Example 2.2.5. Consider the state $\mathfrak{A}$ for Σ_{bool} of Example 2.2.2. Let ζ be a variable assignment with $\zeta(v_0) = 0$, $\zeta(v_1) = 1$ and $\zeta(v_2) = 1$. Then we have:

$$[\![(v_0 + v_1) * v_2]\!]^{\mathfrak{A}}_{\zeta} = 1.$$

The same term can be interpreted in the state $\mathfrak{B}$ of Example 2.2.2. Let $\xi(v_0) = \{2, 3, 5\}$, $\xi(v_1) = \{2, 7\}$ and $\xi(v_2) = \{3, 7, 11\}$. Then we have:

$$[\![(v_0 + v_1) * v_2]\!]^{\mathfrak{B}}_{\xi} = \{3, 7\}.$$

In the first case, the value of the term is a non-negative integer, whereas in the second case the value of the term is a set of non-negative integers.

Definition 2.2.6 (Formula). Let Σ be a vocabulary. The formulas of Σ are generated as follows:

1. If s and t are terms of Σ, then $s = t$ is a formula.
2. If φ is a formula, then $\neg\, \varphi$ is a formula.
3. If φ and ψ are formulas, then $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ and $(\varphi \rightarrow \psi)$ are formulas.
4. If φ is a formula and x a variable, then $(\forall x\, \varphi)$ and $(\exists x\, \varphi)$ are formulas.

The logical connectives and quantifiers have the standard meaning:

symbol	name	meaning
$\neg$	negation	not
$\wedge$	conjunction	and
$\vee$	disjunction	or (inclusive)
$\rightarrow$	implication	if-then
$\forall$	universal quantification	for all
$\exists$	existential quantification	there is

A formula $s = t$ is called an *equation*. The expression $s \neq t$ is an abbreviation for the formula $\neg\,(s = t)$.

In order to increase the readability of formulas parentheses are often omitted. For example, the following conventions are used:

$\varphi \wedge \psi \wedge \chi$	stands for	$((\varphi \wedge \psi) \wedge \chi)$,
$\varphi \vee \psi \vee \chi$	stands for	$((\varphi \vee \psi) \vee \chi)$,
$\varphi \wedge \psi \rightarrow \chi$	stands for	$((\varphi \wedge \psi) \rightarrow \chi)$, etc.

Formulas can be interpreted in a state with respect to a variable assignment. Formulas are either true or false in a state. The truth value of a formula in a state is computed recursively. The classical truth tables for the logical connectives and the classical interpretation of quantifiers are used. The equality sign is interpreted as identity.

Definition 2.2.7 (Interpretation of formulas). Let $\mathfrak{A}$ be a state of Σ, φ be a formula of Σ and ζ be a variable assignment in $\mathfrak{A}$. By induction on the length of φ, a truth value $[\![\varphi]\!]^{\mathfrak{A}}_{\zeta} \in \{\mathit{True}, \mathit{False}\}$ is defined as follows:

$$[\![s = t]\!]^{\mathfrak{A}}_{\zeta} := \begin{cases} \mathit{True}, & \text{if } [\![s]\!]^{\mathfrak{A}}_{\zeta} = [\![t]\!]^{\mathfrak{A}}_{\zeta}; \\ \mathit{False}, & \text{otherwise.} \end{cases}$$

$$[\![\neg\,\varphi]\!]^{\mathfrak{A}}_{\zeta} := \begin{cases} \mathit{True}, & \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{False}; \\ \mathit{False}, & \text{otherwise.} \end{cases}$$

$$[\![\varphi \wedge \psi]\!]^{\mathfrak{A}}_{\zeta} := \begin{cases} \mathit{True}, & \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{True} \text{ and } [\![\psi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{True}; \\ \mathit{False}, & \text{otherwise.} \end{cases}$$

$$[\![\varphi \vee \psi]\!]^{\mathfrak{A}}_{\zeta} := \begin{cases} \mathit{True}, & \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{True} \text{ or } [\![\psi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{True}; \\ \mathit{False}, & \text{otherwise.} \end{cases}$$

$$[\![\varphi \rightarrow \psi]\!]^{\mathfrak{A}}_{\zeta} := \begin{cases} \mathit{True}, & \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{False} \text{ or } [\![\psi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{True}; \\ \mathit{False}, & \text{otherwise.} \end{cases}$$

$$[\![\forall x\, \varphi]\!]^{\mathfrak{A}}_{\zeta} := \begin{cases} \mathit{True}, & \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta\frac{a}{x}} = \mathit{True} \text{ for all } a \in |\mathfrak{A}|; \\ \mathit{False}, & \text{otherwise.} \end{cases}$$

$$[\![\exists x\, \varphi]\!]^{\mathfrak{A}}_{\zeta} := \begin{cases} \mathit{True}, & \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta\frac{a}{x}} = \mathit{True} \text{ for some } a \in |\mathfrak{A}|; \\ \mathit{False}, & \text{otherwise.} \end{cases}$$

We say that a state $\mathfrak{A}$ is a *model* of φ, if $[\![\varphi]\!]_\zeta^{\mathfrak{A}} = \mathit{True}$ for all variable assignments ζ.

Example 2.2.6. The states $\mathfrak{A}$ and $\mathfrak{B}$ of Example 2.2.2 are models of the following equations:

$$\begin{array}{ll}
(x+y)+z = x+(y+z), & (x*y)*z = x*(y*z), \\
x+y = y+x, & x*y = y*x, \\
x+(x*y) = x, & x*(x+y) = x, \\
x+(y*z) = (x+y)*(x+z), & x*(y+z) = (x*y)+(x*z), \\
x+(-x) = 1, & x*(-x) = 0.
\end{array}$$

These formulas are called axioms of a Boolean algebra.

2.2.2 Transition rules and runs

In mathematics, states like Boolean algebras are static. They do not change over time. In computer science, states are dynamic. They evolve by being updated during computations. Updating abstract states means to change the interpretation of (some of) the functions in the underlying signature. The way ASMs update states is described by transitions rules of the following form which define the syntax of ASM programs.

Definition 2.2.8 (Transition rules). Let Σ be a vocabulary. The transition rules R, S of an ASM are syntactic expressions generated as follows:

1. *Skip Rule:*

   skip

   Meaning: Do nothing.

2. *Update Rule:*

   $f(t_1, \ldots, t_n) := s$

   Syntactic conditions:
   – f is an n-ary, dynamic function name of Σ
   – $t_1, \ldots, t_n$ and s are terms of Σ
   Meaning: In the next state, the value of the function f at the arguments $t_1, \ldots, t_n$ is updated to s. It is allowed that f is a 0-ary function, i.e., a constant. In this case, the update has the form $c := s$.

3. *Block Rule:*

   $R\ S$

   Meaning: R and S are executed in parallel.

4. *Conditional Rule:*

   if φ **then** R **else** S

   Meaning: If φ is true, then execute R, otherwise execute S.

5. *Let Rule:*

   let $x = t$ **in** R

   Meaning: Assign the value of t to x and execute R.

6. *Forall Rule:*

   forall x **with** φ **do** R

   Meaning: Execute R in parallel for each x satisfying φ.

7. *Call Rule:*

   $r(t_1, \dots, t_n)$

   Meaning: Call r with parameters $t_1, \dots, t_n$.

A *rule definition* for a rule name r of arity n is an expression

$$r(x_1, \dots, x_n) = R,$$

where R is a transition rule. In a rule call $r(t_1, \dots, t_n)$ the variables x_i in the body R of the rule definition are replaced by the parameters t_i.

Definition 2.2.9 (ASM). An *abstract state machine* M consists of a vocabulary Σ, an initial state $\mathfrak{A}$ for Σ, a rule definition for each rule name, and a distinguished rule name of arity zero called the *main rule name* of the machine.

The semantics of transition rules is given by sets of updates. Since due to the parallelism (in the Block and the Forall rules), a transition rule may prescribe to update the same function at the same arguments several times, we require such updates to be consistent. The concept of consistent update sets is made more precise by the following definitions.

Definition 2.2.10 (Update). An *update* for $\mathfrak{A}$ is a triple $(f, (a_1, \dots, a_n), b)$, where f is an n-ary dynamic function name, and $a_1, \dots, a_n$ and b are elements of $|\mathfrak{A}|$.

The meaning of the update is that the interpretation of the function f in $\mathfrak{A}$ has to be changed at the arguments $a_1, \dots, a_n$ to the value b. The pair of the first two components of an update is called a *location*. An update specifies how the function table of a dynamic function has to be updated at the corresponding location. An *update set* is a set of updates.

In a given state, a transition rule of an ASM produces for each variable assignment an update set. Since the rule can contain recursive calls to other rules, it is also possible that it has no semantics at all. The semantics of a transition rule is therefore defined by a calculus in Fig. 2.2.

Definition 2.2.11 (Semantics of transition rules). The semantics of a transition rule R of a given ASM in a state $\mathfrak{A}$ with respect to a variable assignment ζ is defined if and only there exists an update set U such that $[\![R]\!]_{\zeta}^{\mathfrak{A}} \ \triangleright \ U$ can be derived in the calculus in Fig. 2.2. In that case $[\![R]\!]_{\zeta}^{\mathfrak{A}}$ is identified with U.

It can happen that the update set $[\![R]\!]_{\zeta}^{\mathfrak{A}}$ contains several updates for the same function name f. In this case, the updates have to be consistent, otherwise the execution stops.

Definition 2.2.12 (Consistent update set). An update set U is called *consistent*, if it satisfies the following property:

$$\text{If } (f,(a_1,\ldots,a_n),b) \in U \text{ and } (f,(a_1,\ldots,a_n),c) \in U, \text{ then } b = c.$$

This means that a consistent update set contains for each function and each argument tuple at most one value.

If an update set U is consistent, it can be fired in a given state. The result is a new state in which the interpretations of dynamic function names are changed according to U. The interpretations of static function names are the same as in the old state. The interpretation of monitored functions is given by the environment and can therefore change in an arbitrary way.

Definition 2.2.13 (Firing of updates). The result of firing a consistent update set U in a state $\mathfrak{A}$ is a new state $\mathfrak{B}$ with the same superuniverse as $\mathfrak{A}$ satisfying the following two conditions for the interpretations of function names f of Σ:

1. If $(f,(a_1,\ldots,a_n),b) \in U$, then $f^{\mathfrak{B}}(a_1,\ldots,a_n) = b$.
2. If there is no b with $(f,(a_1,\ldots,a_n),b) \in U$ and f is not a monitored function, then $f^{\mathfrak{B}}(a_1,\ldots,a_n) = f^{\mathfrak{A}}(a_1,\ldots,a_n)$.

Since U is consistent, for static and controlled functions the state $\mathfrak{B}$ is determined in a unique way. Notice that only those locations can have a new value in state $\mathfrak{B}$ with respect to state $\mathfrak{A}$ for which there is an update in U. (In this way ASMs avoid the so called frame problem.)

Definition 2.2.14 (Run of an ASM). Let M be an ASM with vocabulary Σ, initial state $\mathfrak{A}$ and main rule name r. Let ζ be a variable assignment. A *run* of M is a finite or infinite sequence $\mathfrak{B}_0, \mathfrak{B}_1, \ldots$ of states for Σ such that the following conditions are satisfied:

1. $\mathfrak{B}_0 = \mathfrak{A}$.
2. If $[\![r]\!]_{\zeta}^{\mathfrak{B}_n}$ is not defined or inconsistent, then $\mathfrak{B}_n$ is the last state in the sequence.
3. Otherwise, $\mathfrak{B}_{n+1}$ is the result of firing $[\![r]\!]_{\zeta}^{\mathfrak{B}_n}$ in $\mathfrak{B}_n$.

If we assume that for each rule definition $r(x_1,\ldots,x_n) = R$ of the machine M the free variables of R are among $x_1,\ldots,x_n$, then a run is independent of the variable assignment ζ (see Exercise 2.2.2).

Fig. 2.2 The semantics of ASM rules

$$\frac{}{\llbracket \textbf{skip} \rrbracket^{\mathfrak{A}}_{\zeta} \ \rhd\ \emptyset}$$

$$\frac{}{\llbracket f(t) := s \rrbracket^{\mathfrak{A}}_{\zeta} \ \rhd\ \{(f, a, b)\}} \qquad \text{if } a = \llbracket t \rrbracket^{\mathfrak{A}}_{\zeta} \text{ and } b = \llbracket s \rrbracket^{\mathfrak{A}}_{\zeta}$$

$$\frac{\llbracket R \rrbracket^{\mathfrak{A}}_{\zeta} \ \rhd\ U \qquad \llbracket S \rrbracket^{\mathfrak{A}}_{\zeta} \ \rhd\ V}{\llbracket R\ S \rrbracket^{\mathfrak{A}}_{\zeta} \ \rhd\ U \cup V}$$

$$\frac{\llbracket R \rrbracket^{\mathfrak{A}}_{\zeta} \ \rhd\ U}{\llbracket \textbf{if } \varphi \textbf{ then } R \textbf{ else } S \rrbracket^{\mathfrak{A}}_{\zeta} \ \rhd\ U} \qquad \text{if } \llbracket \varphi \rrbracket^{\mathfrak{A}}_{\zeta} = \mathit{True}$$

$$\frac{\llbracket S \rrbracket^{\mathfrak{A}}_{\zeta} \ \rhd\ U}{\llbracket \textbf{if } \varphi \textbf{ then } R \textbf{ else } S \rrbracket^{\mathfrak{A}}_{\zeta} \ \rhd\ U} \qquad \text{if } \llbracket \varphi \rrbracket^{\mathfrak{A}}_{\zeta} = \mathit{False}$$

$$\frac{\llbracket R \rrbracket^{\mathfrak{A}}_{\zeta\frac{a}{x}} \ \rhd\ U}{\llbracket \textbf{let } x = t \textbf{ in } R \rrbracket^{\mathfrak{A}}_{\zeta} \ \rhd\ U} \qquad \text{if } a = \llbracket t \rrbracket^{\mathfrak{A}}_{\zeta}$$

$$\frac{\llbracket R \rrbracket^{\mathfrak{A}}_{\zeta\frac{a}{x}} \ \rhd\ U_a \quad \text{for each } a \in I}{\llbracket \textbf{forall } x \textbf{ with } \varphi \textbf{ do } R \rrbracket^{\mathfrak{A}}_{\zeta} \ \rhd\ \bigcup_{a \in I} U_a} \qquad \text{if } I = \{a \in |\mathfrak{A}| : \llbracket \varphi \rrbracket^{\mathfrak{A}}_{\zeta\frac{a}{x}} = \mathit{True}\}$$

$$\frac{\llbracket R \rrbracket^{\mathfrak{A}}_{\zeta\frac{a}{x}} \ \rhd\ U}{\llbracket r(t) \rrbracket^{\mathfrak{A}}_{\zeta} \ \rhd\ U} \qquad \begin{array}{l}\text{if } r(x) = R \text{ is a rule definition} \\ \text{and } a = \llbracket t \rrbracket^{\mathfrak{A}}_{\zeta}\end{array}$$

2.2.3 Syntactic sugar

To extend a subuniverse of the superuniverse by new elements we use the following notation:

create x **do** R

The meaning of this construct is

let $x = f_{new}(\ldots)$ **in** R

where $f_{new}(\ldots)$ is a monitored function (possibly with parameters) which returns a new element of the superuniverse which does not belong to any of the subuniverses.

As mentioned above we sometimes use the following notation as syntactic sugar for monitored choice functions:

choose x **with** φ **do** R

We understand this notation as an abbreviation for the rule

let $x = f_{\varphi}(\ldots)$ **in** R,

where $f_{\varphi}(\ldots)$ is an monitored choice function updated by the environment which returns elements satisfying the selection condition φ. Of course different occurrences of **choose** have to be replaced by different choice functions (possibly with parameters) to guarantee the independence of selection.

Another approach would be to add **choose** as a basic construct to the syntax and to extend the calculus in Fig. 2.2 in the following way:

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\zeta\frac{a}{x}} \;\rhd\; U}{[\![\textbf{choose } x \textbf{ with } \varphi \textbf{ do } R]\!]^{\mathfrak{A}}_{\zeta} \;\rhd\; U} \quad \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta\frac{a}{x}} = \mathit{True}$$

This approach has the disadvantage that the semantics of a transition rule R is no longer unique, because there can be different update sets U such that $[\![R]\!]^{\mathfrak{A}}_{\zeta} \;\rhd\; U$ is derivable in the calculus.

2.2.4 Exercises

Exercise 2.2.1. Prove the following equation:

$$[\![(\textbf{if } \varphi \textbf{ then } R_1 \textbf{ else } R_2)\ S]\!]^{\mathfrak{A}}_{\zeta} = [\![\textbf{if } \varphi \textbf{ then } (R_1\ S) \textbf{ else } (R_2\ S)]\!]^{\mathfrak{A}}_{\zeta}$$

Is the following equation true?

$$[\![S\ (\textbf{if } \varphi \textbf{ then } R_1 \textbf{ else } R_2)]\!]^{\mathfrak{A}}_{\zeta} = [\![\textbf{if } \varphi \textbf{ then } (S\ R_1) \textbf{ else } (S\ R_2)]\!]^{\mathfrak{A}}_{\zeta}$$

If yes, why? If not, give a counter example.

Exercise 2.2.2. The set of free variables of a term t is defined as follows:

1. $\mathrm{FV}(v_i) := \{v_i\}$
2. $\mathrm{FV}(c) := \emptyset$