Computing and Informatics, Vol. 20, 2001, 1–10, V 2003-Aug-28

THE EXPRESSIVE POWER OF ABSTRACT-STATE MACHINES

Wolfgang Reisig

Institut für Informatik
Humboldt Universität zu Berlin
Unter den Linden 6
10099 Berlin, Deutschland
e-mail: `reisig@informatik.hu-berlin.de`

Abstract. Conventional computation models assume symbolic representations of states and actions. Gurevich's "Abstract-State Machine" model takes a more liberal position: *Any* mathematical structure may serve as a state. This results in "a computational model that is more powerful and more universal than standard computation models" [5].

We characterize the Abstract-State Machine model as a special class of transition systems that widely extends the class of "computable" transition systems. This characterization is based on a fundamental Theorem of Y. Gurevich.

1 DETERMINISTIC TRANSITION SYSTEMS

In the first volume of his seminal opus [8], Don Knuth introduces the notation of *algorithms*. As a framework for the semantics of algorithms, Knuth suggests *computational methods*: A computational method is what nowadays would be called an initialized, deterministic transition system, i.e. a triple

$$C = (Q, I, F)$$

where Q is a set (its elements are denoted as *states*), $I \subseteq Q$ (the *initial states*), and $F : Q \to Q$ (the *next-state function*). Furthermore, Knuth assumes a set Ω of *terminal states*, and requires $F(q) = q$ for all $q \in \Omega$: A terminal state is represented as a fixpoint of F. (Knuth denotes I, Ω and F as *input states*, *output states* and *computation rule*, respectively).

As one would expect, a *computation sequence of C* is a sequence

$$\sigma = x_0 x_1 \dots$$

of states $x_i \in Q$, with $x_0 \in I$ and $x_{i+1} = F(x_i)$ for $i = 0, 1, \dots$. The sequence σ is said to *terminate* if $x_k \in \Omega$ for some index k. Knuth denotes C as an *algorithm* if each computation sequence of C terminates.

This definition comes without the requirement of F being "effective". Quoting [8, p8]: "F might involve operations that mortal man can not always perform". Knuth defines *effective* computational methods as a special case: A computational method is effective iff it is essentially equivalent to a Turing Machine or to any other mechanism for the computable functions. Nowadays, the term "algorithm" is usually used to denote what Knuth calls an "effective computational method".

As we did above already, we will use the term "transition system" instead of "computational method", and "effective transition system" instead of "effective computational method".

Transition systems have been generalized in several directions: Non-terminating computation sequences adequately describe behaviors of reactive systems; the next-state function F has been generalized to a relation $R \subseteq Q \times Q$, with computation sequences $x_0 x_1 \dots$ where $(x_i, x_{i+1}) \in R$. This represents nondeterminism. Additionally one may require the choice of x_{i+1} to follow a stochastic distribution, or to be fair. Some system models describe a single behavior not as a sequence of states, but as a sequence of actions. The sequence orders the actions along a time axis. One may even replace the total order by a partial order, representing the cause-effect relation among actions.

All these generalizations of effective transition systems can be reduced to equivalent conventional effective transition systems, by reasonable notions of reduction and equivalence. Generalizations of this kind are intended to express algorithmic ideas more conveniently. They are not intended to challenge the established notion of effective computation.

We study non-effective transition systems in this paper. The reader may wonder whether there is anything interesting "beyond" the computable functions. In fact, there is an exciting proper subclass of all transition systems, called "Sequential Abstract-State Machines" (ASM), that in turn properly contains the effective transition systems. A Sequential ASM has a next-state function that can be represented symbolically. But its states can in general not be represented symbolically.

This paper is intended to provide quick insight into the basic principles of ASM. It should help understand why ASM in fact are more universal than standard computation models. Furthermore, the paper provides intuitive reasons for the success of ASM as a specification language, applied in many quite different areas. These applications are only glanced, as well as the various extensions of ASM and the powerful tools. The paper is not intended to present new results, so the ASM expert will not find any new result in this paper.

The rest of this paper is organized as follows: Section 2 presents a very simple example to explain the basic idea of ASM. A formal definition of a very simple kind of Sequential ASM is given in Section 3. Finally, Section 4 presents Gurevich's Theorem about this class of ASM. It characterizes the amazing expressive power of Sequential ASM.

2 A SIMPLE EXAMPLE OF AN ABSTRACT-STATE MACHINE

We begin with a somewhat overly simple example. This example is intended to show that the notion of *algorithm* is very reasonably defined also in the context of states that can not symbolically be represented.

Let *augment* be a function with two arguments: The first argument is to be any set, and the second argument is to be any item. For a set M and any item m, define

$$augment(M, m) =_{def} M \cup \{m\}. \tag{1}$$

We intend to construct an algorithm that augments two elements to a set, using the function *augment.* More precisely, we want to construct a transition system $C_0 = (Q_0, I_0, F_0)$, such that each set M together with two elements m and n constitutes an initial state $x_{M,m,n}$. A computation sequence starting with $x_{M,m,n}$ then should terminate with a state that represents the set $M \cup \{m, n\}$.

As usual in the description of algorithms and programs, we employ *variables.* We choose the variable X to vary over sets. In an initial state $x_{M,m,n}$ of C_0 the value of X is M.

The steps of C_0 can be described in the setting of an algorithm P_0:

$$\begin{array}{lll} P_0: & begin & X := augment(X, m); \\ & & X := augment(X, n) \\ & end. & \end{array} \tag{2}$$

According to (2), an initial state S_0 of C_0 is followed by a state S_1, which assigns the set $M \cup \{m\}$ to X. State S_1 is then followed by a state S_2 that assigns the set $M \cup \{m, n\}$ to X. S_2 is a terminal state of C_0.

The description (2) mixes the symbols "*X*", "*begin*", "*end*", "*;*" and "*:=*" with the concrete items m and n, and the function *augment.* We would prefer an *entirely* symbolic representation. This is achieved by two additional variables, x and y, and a binary operation symbol, g. In an initial state $x_{M,m,n}$ of C_0 the value of X, x and y is now assumed to be M, m and n, respectively, and the value of the operation g is assumed to be the function *augment.* Then the steps of C_0 can be represented as

$$\begin{array}{lll} P_1: & begin & X := g(X, x); \\ & & X := g(X, y) \\ & end. & \end{array} \tag{3}$$

The description (3) is an entirely symbolic representation indeed, i.e. a program text. All states of P_1 assign m, n and *augment* to x, y and g, respectively.

The representation of states still includes items such as M, m, n, $M \cup \{m, n\}$ and *augment*. We might wish a *symbolic* representation of states, too. This would require a symbolic representation of sets. A symbolic representation is a sequence of symbols over a finite alphabet, Σ. There are only countable many representations over Σ. There exist many more sets, however. So it is impossible to represent all sets symbolically. Consequently, it is impossible to define the assignment of values to variables symbolically. Hence it is impossible to represent the states of the transition system C_0 symbolically.

This is the essence of the above example: The next-state function F_0 of C_0 can be represented symbolically, as in (3), whereas the states of C_0 can not.

3 SEQUENTIAL ABSTRACT-STATE MACHINES

The example of Section 2 doesn't fit into the framework of conventional computation models. It is an example for a more general model. To highlight the differences, let $C = (Q, I, F)$ be an initialized, deterministic transition system, as described in Section 1.

First, assume C is an effective transition system. Each state $S \in Q$ can be assumed to be represented as a finite sequence of symbols. The successor state $F(S)$ is a sequence of symbols, in general different from S. Summing up, arguments as well as results of the next-state function F are finite sequences of symbols, and there exists a finite symbolic representation for F.

Now assume C is a transition system in the framework of the more general model. A state S of C is not symbolically represented, but a *semantical* object. It consists of a set U (in general infinite), a finite number of distinguished elements $u_1, \ldots, u_k$ in U, and a finite number $g_1, \ldots, g_l$ of functions over U, each with its arity k_i (i.e. $g_i : U^{k_i} \rightarrow U$).

We employ variables $x_1, \ldots, x_k$ and symbols $f_1, \ldots, f_l$. In a state S, the variable x_i denotes the element u_i. Hence we write x_{iS} for u_i $(i = 1, \ldots, k)$. The symbol f_j denotes the function g_j, written f_{jS} $(j = 1, \ldots, l)$.

All states of a transition system employ the same variables and function symbols, and these variables and function symbols are used to symbolically represent the next-state function F of C.

In technical terms, the variables and function symbols constitute a *signature*, Σ. Each state of C is a Σ-*Algebra* (i.e. an element of Alg_Σ). Variables, as we used them, are usually denoted as "constant symbols" or "0-ary function symbols" in the framework of Σ-algebras. We refer to the appendix for details of signatures and algebras.

We formulate the next-state function in a programming like notation. The elementary dynamic construct is the *assignment statement*, formed

$$x := t, \tag{4}$$

where x is usually called a *variable*, and t a Σ-*ground term*. Remember that x

is indeed a 0-ary constant symbol of Σ. Applied to a state S, the assignment (4) yields a successor state $F(S)$, where x is assigned the value t_S (i.e. $x_{F(S)} = t_S$).

Assignments are not only defined for 0-ary symbols, but for all ground terms $t \in T_\Sigma$. A ground term $t \in T_\Sigma$ is in general shaped $t = f(t_1, \ldots, t_n)$, where f is an n-ary function symbol, and $t_1, \ldots, t_n \in T_\Sigma$. In case of $n = 0$, t is a constant symbol, as discussed above. Applied to a state S, an assignment

$$f(t_1, \ldots, t_n) := t_0 \tag{5}$$

updates the function f_S at the argument tuple $(t_{1S}, \ldots, t_{nS})$. Hence, for the successor state $F(S)$ of S holds $f_{F(S)}(t_{1S}, \ldots, t_{nS}) = t_{0S}$. This kind of update may come as a surprise. It corresponds to an update of an n-dimensional array f in conventional programming.

We will allow a set of assignments being performed coincidentally, provided they are consistent, viz no two different assignments update the same location $f_S(t_{1S}, \ldots, t_{kS})$. Here the formal definition:

Definition 1. Let Σ be a signature.

1. Let $t, t_0 \in T_\Sigma$ with t shaped $t = f(t_1, \ldots, t_k)$. Then $f(t_1, \ldots, t_k) := t_0$ is a Σ-*assignment.*
2. Let $S \in Alg_\Sigma$. Two Σ-assignments $f(t_1, \ldots, t_k) := t_0$ and $f(t'_1, \ldots, t'_k) := t'_0$ are *consistent at S* iff $(t_{1S}, \ldots, t_{kS}) = (t'_{1S}, \ldots, t'_{kS})$ implies $t_{0S} = t'_{0S}$.
3. A set of Σ-assignments is *consistent at S* iff its elements are pairwise consistent at S.

In the sequel we do with signatures that include the symbols *true*, *false*, *undefined*, and the usual propositional logic combinators such as "$\wedge$" and "$\neg$". Such signatures will be denoted as *ASM signatures.*

We are now prepared to introduce the program notation for next-state functions:

Definition 2. Let Σ be an ASM-signature.

1. The set $guard_\Sigma$ of *guards over* Σ is the smallest set of terms such that for all $t, t' \in T_\Sigma$, $t = t' \in guard_\Sigma$, and $\beta, \beta' \in guard_\Sigma$ implies $\neg\beta \in guard_\Sigma$ and $\beta \wedge \beta' \in guard_\Sigma$.
2. Let r be a Σ-assignment and let $\beta \in guard_\Sigma$. Then

$$\mathsf{if}\ \beta\ \mathsf{then}\ \mathsf{r}$$

   is a *guarded* Σ-*assignment.*
3. Let $\mathsf{q}_1,\ldots,\mathsf{q}_m$ be guarded Σ-assignments. Then

$$\mathbf{par}\ \mathsf{q}_1, \ldots, \mathsf{q}_\mathsf{n}\ \mathbf{endpar}$$

   is a *sequential, bounded ASM-program* over Σ.

Notation We frequently write r for the guarded assignment $\mathsf{if}\ t = t\ \mathsf{then}\ \mathsf{r}$, with any term $t \in T_\Sigma$.

The semantics of an ASM program is based on the semantics of Σ-assignments: An assignment $f(t_1, \ldots, t_n) := t_0$ updates in a state S the function f_S at the argument tuple $(t_{1S}, \ldots, t_{nS})$ with the value t_{0S}. This generalizes to consistent sets Z in the obvious way.

Definition 3. Let Σ be a signature, let S be a Σ-algebra with universe U, and let Z be a set of Σ-assignments, consistent at S.

1. For a k-ary symbol $f \in \Sigma$, let

$$\begin{array}{llcl} f^{Z,S}: & U^k & \to & U \\ & (t_{1S}, \ldots, t_{kS}) & \mapsto & t_{0S} \text{ iff } f(t_1, \ldots, t_k) := t_0 \in Z \\ & \mathbf{u} & \mapsto & f_S(\mathbf{u}), \text{ otherwise} \end{array}$$

2. Let $R =_{def} \mathrm{sem}_Z(S)$ be the Σ-algebra with universe U, defined for each symbol $f \in \Sigma$ by $f_R = f^{Z,S}$.

The semantics of an ASM program $\mathcal{M}$ is now reduced to the semantics of assignments: To apply $\mathcal{M}$ to a state S, first evaluate the guards of $\mathcal{M}$ at S, and then execute all assignments with true guards:

Definition 4. Let Σ be an ASM-signature.

1. Let $\beta \in \mathrm{guard}_\Sigma$ and let $S \in \mathrm{Alg}_\Sigma$. Then $\beta_S = \mathit{true}$ iff $\beta = (t = t')$ and $t_S = t'_S$, or if $\beta = \neg\beta'$ and not $\beta'_S = \mathit{true}$, or if $\beta = \beta' \wedge \beta''$ and $\beta'_S = \beta''_S = \mathit{true}$.
2. Let

$$\begin{array}{ll} \mathcal{M}: & \textbf{par}\ \mathsf{if}\ \beta_1\ \mathsf{then}\ \mathsf{r}_1 \\ & \quad\vdots \\ & \quad \mathsf{if}\ \beta_m\ \mathsf{then}\ \mathsf{r}_m \\ & \textbf{endpar} \end{array}$$

be an ASM-program over Σ.

Then the semantic function $\mathrm{sem}_\mathcal{M} : \mathrm{Alg}_\Sigma \to \mathrm{Alg}_\Sigma$ is defined as follows:

For $S \in \mathrm{Alg}_\Sigma$ let $Z := \{\mathsf{r}_i \mid \beta_{iS} = \mathsf{true}, 1 \le i \le m\}$. Then

$$\mathrm{sem}_\mathcal{M}(S) := \begin{cases} \mathrm{sem}_Z(S), & \text{if } Z \text{ is consistent on S }, \\ S, & \text{otherwise }. \end{cases}$$

An ASM program $\mathcal{M}$ over a signature Σ can serve as the next-state function of a transition system that employs Σ-algebras as states:

Definition 5. Let Σ be an ASM signature, let $\mathcal{M}$ be an ASM program over Σ and let $\mathcal{A} = (Q, I, F)$ be a transition system with $Q \subseteq Alg_\Sigma$ and $F = sem_\mathcal{M}$. Then $\mathcal{A}$ is an *ASM transition system.*

Program P_1 in (3) of Section 2 is no ASM program at first glance: An ASM program can not express sequential composition. This deficit is easily overcome by a well-known "trick": Extend the initial state by a fresh variable, l, and valuate l by 0 in the initial state, S_0. Reformulate (3) by

$$\begin{array}{lll} P_2: & \textbf{par} & \text{if } l = 0 \text{ then } X := g(X, x); \\ & & \text{if } l = 0 \text{ then } l := 1; \\ & & \text{if } l = 1 \text{ then } X := g(X, y); \\ & & \text{if } l = 1 \text{ then } l := 2 \\ & \textbf{endpar}. & \end{array}$$

Many versions of ASM deviate from the strict syntax of Definition 4, very well allowing notations such as (3).

4 THE EXPRESSIVE POWER OF SEQUENTIAL ASM TRANSITION SYSTEMS

In the sequel we discuss the expressive power of ASM transition systems. They turn out to be amazingly expressive: Intuitively formulated, every transition system with a symbolically representable next-state function can be represented as an ASM transition system. As a technicality, all constructs go up to isomorphism.

First we describe the next-state function F for each state S by the "difference" between S and its follower state, $F(S)$. This difference is a set of *updates* $(f, \mathbf{u}, v)$, where f is a function symbol, $\mathbf{u}$ is an n-tuple of elements with n the arity of f, and v is a single element. Applied to a state S, an update $(f, \mathbf{u}, v)$ denotes $f_{F(S)}(\mathbf{u}) = v$, i.e. the function symbol f, interpreted in state $F(S)$, and applied to the argument $\mathbf{u}$, is equal to v. Here a formal definition:

Definition 6. Let Σ be a signature, let f be a function symbol in Σ with arity n, let U be a universe, let $\mathbf{u} \in U^n$ and $v \in U$. Then $(f, \mathbf{u}, v)$ is a Σ-*update over* U.

For example, the update $(X, (), M \cup \{m\})$ describes the step from S_0 to S_1 in Section 2. The next-state function F usually updates more than one function at more than one position: The step from a state S to its successor state $F(S)$ covers usually a set of updates:

Definition 7. Let $\mathcal{A} = (Q, I, F)$ be an initialized, deterministic transition system with $Q \subseteq Alg_\Sigma$ for a signature Σ. Let $S \in Q$.

1. A Σ-update $(f, \mathbf{u}, v)$ is an F-*update of* S iff $f_S(\mathbf{u}) \neq f_{F(S)}(\mathbf{u}) = v$.
2. Let $\Delta(F, S)$ denote the set of all F-updates of S.

For all S, the sets $\Delta(F,S)$ together constitute F. Hence, F may be characterized by help of the sets $\Delta(F,S)$, i.e. by a symbolic representation of $\Delta(F,S)$. Such a symbolic representation must meet the following properties: If two states R and S evolve different update sets, i.e. $\Delta(F,S) \neq \Delta(F,R)$, then at least one term $t \in T_\Sigma$ witnesses the difference: $t_R \neq t_S$. Furthermore, as a symbolic representation is always finite, i.e. finitely many witnesses must suffice for all states. F is called a *bounded exploration* in this case. Here is a formal definition:

Definition 8. Let $\mathcal{A} = (Q, I, F)$ be an initialized, deterministic transition system with $Q \subseteq Alg_\Sigma$ for a signature Σ. Let $T \subseteq T_\Sigma$ such that for all states $R, S \in Q$: If for all $t \in T$ the equation $t_R = t_S$ holds, then $\Delta(F,R) = \Delta(F,S)$. In this case, T is called *characteristic* for F. F is a *bounded exploration* if there exists a finite characteristic set T of terms for F.

The bounded exploration property is the decisive property for F to be representable by an ASM program. The additional requirement is a technicality: States and initial states are closed under isomorphism, and the next-state function F is invariant under isomorphism. These requirements are due to the well known observation that term representations can not distinguish isomorphic structures:

Definition 9. Σ be a signature, let $\mathcal{A} = (Q, I, F)$ be an initialized, deterministic transition system with $Q \subseteq Alg_\Sigma$ such that for all $R, S \in Q$ and each isomorphism $h : R \to S$ holds:

1. $R \in Q$ iff $S \in Q$ and $R \in I$ iff $S \in I$.
2. $h : F(R) \to F(S)$ is an isomorphism, too.

Then $\mathcal{A}$ is *isomorphic closed.*

The following Theorem describes the expressive power of ASM transition systems:

Theorem 1. Let Σ be an ASM signature, let $Q \subseteq Alg_\Sigma$, let $\mathcal{A} = (Q, I, F)$ be an initialized, deterministic, isomorphism closed transition system and let F be a bounded exploration. Then there exists an ASM program $\mathcal{M}$ with $F = sem_\mathcal{M}$.

This fundamental Theorem has been proven in [7]. The proof has been reformulated in [9].

5 FURTHER ASPECTS OF ASM

The term "Abstract-State Machine" refers to the fundamental idea that a state is a mathematical structure. We discussed the most elementary version of ASM in a particular syntactic representation. In general, the syntax of ASM is not entirely fixed; many other versions are likewise reasonable.

There are substantial extensions to the elementary versions of ASM as discussed in this paper: The deterministic next-state function F may be replaced by a non-deterministic next-state relation. Reactive behavior of an ASM program may be modelled by steps $x_i x_{i+1}$ which are not conducted by the program, but by the outside world. Bounded exploration may be generalized, using $\forall$-quantified formulas in ASM programs. A *distributed* version of ASM has likewise been advocated, where a simple run consists of a partial order of actions. *Turbo ASM* allow to squeeze a sequence of assignments into one super-assignment. The Lipari Guide [6] gives details on various versions of ASM and their expressive power.

Σ-algebras are the most neutral kind of mathematical structure. Σ-algebras therefore provide the most flexible means to model states of systems. This is why the ASM approach has "...the ability to simulate arbitrary algorithms on their natural level of abstraction, without implementing them" [2]. In fact, the liberal requirement for states and steps adept easily and perfectly to any kind of algorithm, and in particular to high-level system design.

It comes without surprise that ASM have been particularly successful in describing the semantics of programming languages: Semantics deals with a rich variety of mathematical structures that don't require a syntactical representation. This, exactly, is what ASM provides.

The ASM formalism has successfully been applied to virtually all areas of software. The recent text book [4] of Börger and Stärk introduces and surveys the most important aspects of ASM theory, application, and tools. A lot of applications can also be found in [2] and the excellent ASM website [1].

6 ACKNOWLEDGEMENTS

I thank Dines Bjørner for his encouragement to write this paper. Two anonymous referees suggested very valuable improvements of the text.

REFERENCES

[1] The ASM web-page: http://www.eecs.umich.edu/gasm/

[2] Börger, E.: *High Level System Design and Analysis using Abstract State Machines.* In D. Hutter et al, editor, Current Trends in Applied Formal Methods, vol 1464 of LNCS, pp 1-43, 1999.

[3] Börger, E.: *The Origin of the ASM method for High Level System Design and Analysis.* Journal of Universal Computer Science, vol 8, no 1: pp 2-74, 2002.

[4] Börger, E., Stärk, R.: *Abstract State Machines – A Method for High-Level System Design and Analysis.* Springer Verlag, 2003.

[5] Gurevich, Y.: *A new Thesis.* American Mathematical Society Abstracts, page 317, August 1985.

[6] Gurevich, Y.: *Evolving Algebra 1993: Lipari Guide.* In E. Börger, editor, Specification and Validation Methods, pp 9-36, Oxford University Press 1995.

[7] Gurevich, Y.: *Sequential Abstract-State Machines Capture Sequential Algorithms.* ACM Transactions on Computational Logic, Vol 1 No 1, pp 77-111, July 2000.

[8] Knuth, D. E.: *The Art of Computer Programming Vol. 1* Fundamental Algorithms, Addison-Wesley, 1973.

[9] Reisig, W.: *On Gurevich's Theorem on Sequential Algorithms.* Acta Informatica Vol 39, pp 273-305 (2003).

A APPENDIX: ELEMENTARY NOTIONS OF GENERAL ALGEBRA

We employ General Algebra in its most simple form, sticking to homogeneous algebras with total functions. We introduce corresponding signatures and state some elementary relations between signatures and algebras.

An algebra consists of a set and a choice of functions:

Definition A-1. Let U be a set.

1. Let $\varphi : \underbrace{U \times \ldots \times U}_{n-fold} \to U$ be a function. Then n is its arity.
2. The case of arity $n = 0$ yields a constant, i.e. an element $\varphi() \in U$, written φ.
3. Let $n_1, \ldots, n_k \in \mathbb{N}$. For $i = 1, \ldots, k$ let φ_i be a function over U with arity n_i. Then $S = (U, \varphi_1, \ldots, \varphi_k)$ is an algebra. U is its carrier; $(n_1, \ldots, n_k)$ is its type.

A signature provides names for the functions of algebras:

Definition A-2. Let $f_1, \ldots, f_k$ be symbols and let $n_1, \ldots, n_k \in \mathbb{N}$.

1. $\Sigma = (f_1, \ldots, f_k, n_1, \ldots, n_k)$ is a *signature.* For $1 \leq i \leq n_k$, the number n_i is the *arity of* f_i.
2. f_i is a *constant symbol* if $n_i = 0$. f_i is a *function symbol* otherwise.

Ground terms compose symbols according to their arity:

Definition A-3. Let Σ be a signature. The set T_Σ of Σ-*ground terms* is inductively defined as follows:

1. each constant symbol is a ground term
2. if f is a function symbol with arity n, and if $t_1, \ldots, t_n$ are ground terms, then $f(t_1, \ldots, t_n)$ is a ground term, too.

Each signature characterizes a set of algebras:

Definition A-4. Let $\Sigma = (f_1, \ldots, f_k, n_1, \ldots, n_k)$ be a signature. Each algebra $S = (U, g_1, \ldots, g_k)$ with arity $(n_1, \ldots, n_k)$ is called a Σ-*algebra.* S is often called an *interpretation of* Σ; g_i is frequently written f_{iS} and denoted as the *interpretation of* f_i *in* S.

Notation For a signature Σ, let Alg_Σ denote the set of all Σ-algebras.