

On Gurevich’s Theorem for Sequential ASM

W. Reisig

Abstract. Abstract-State Machines have been introduced as “a computation model that is more powerful and more universal than standard computation models”, by Yuri Gurevich in 1985 ([Gur85]).

ASM gained much attention as a specification method, in particular for the description of the semantics of programming languages, communication protocols, distributed algorithms, etc. Gurevich proved recently that a sequential algorithm must only meet a few, liberal requirements, to be representable as an ASM.

We re-formulate Gurevich’s requirements for sequential algorithms, as well as the semantics of ASM-programs and the proof of his main theorem. A couple of examples support and explain intuition and motivation of ASM.

1 Introduction

We illustrate the decisive aspect of Gurevich’s “Abstract-State Machines” (ASM) formalism by a simplistic program \mathcal{P} that computes the square root of the value of the variable x and assigns it to y :

$$\mathcal{P} : \text{begin } y := \text{sqr}(x) \text{ end.} \quad (1)$$

In the conventional setting, this program is applicable to a *state*. Technically, a state of (1) is a mapping

$$S : \text{var} \longrightarrow \text{val} \quad (2)$$

with a finite set *var* of *variables* (or registers) including x and y , and a set *val* of *values*. Values are assumed to be symbolically represented, and *sqr* is assumed to be effectively, mechanically executable on the symbolic representation of values.

This of course does not hold if *sqr* is conceived as the mathematical square root function, $\sqrt{\cdot}$: There exist values v such that \sqrt{v} is not finitely representable. An implementation of (1) on a concrete hardware must *approximate* \sqrt{v} . Details are usually left to the compiler and the operating system.

Conceived as an ASM program, \mathcal{P} is treated quite differently: A state

S for \mathcal{P} not only provides a values x_S for x , but also a function sqr_S for sqr . A reasonable such state is e.g. S_1 , where $x_{S_1} = 2$, y_{S_1} is any value and sqr_{S_1} is the mathematical square root, $\sqrt{\quad}$, approximated to two decimals in the mantissa. Application of \mathcal{P} to S_1 yields a state S'_1 with $x_{S'_1} = 2$, $y_{S'_1} = 1.41$ and $sqr_{S'_1} = sqr_{S_1}$.

Likewise reasonable is a state S_2 where $x_{S_2} = 2$, y_{S_2} is any value, and sqr_{S_2} is the mathematical square root function, $\sqrt{\quad}$. Application of \mathcal{P} to S_2 yields a state S'_2 with $y_{S'_2} = \sqrt{2}$. The value $\sqrt{2}$, and hence the entire state S'_2 , is not finitely representable in the decimal representation for real numbers. S'_2 is nevertheless treated as a first class citizen in the world of ASM-states.

Here a further state, S_3 where $x_{S_3} = 2$, y_{S_3} is any value, and sqr_{S_3} is the successor function for integers. Application of \mathcal{P} to S_3 yields a state S'_3 with $y_{S'_3} = 3$. This example is to emphasize that sqr is just a *symbol*, not connected to any specific function.

The final example is the state S_4 , with sqr_{S_4} defined as follows: For a Universal Turing Machine M , let $sqr_{S_4}(v) = 0$ if the Turing Machine encoded by v terminates for at least one initial tape inscription, and $sqr_{S_4}(v) = 1$ if v encodes an always terminating Turing Machine. Hence the program \mathcal{P} “solves” the Halting Problem for Turing Machines.

In general, the above ASM program \mathcal{P} is applicable to any state S that provides values x_S for x and y_S for y , and a function sqr_S for sqr . It is not required that x_S or y_S be finitely representable, or that sqr_S be computable.

More general, an ASM program \mathcal{P} is constructed from variables and function symbols of any arity. A \mathcal{P} -state assigns to each variable in \mathcal{P} a constant, and to each function symbol of arity n a function of arity n . Constants and functions are not required to be “constructive” in any sense. An ASM-program \mathcal{P} can be applied to a \mathcal{P} -state, yielding again a \mathcal{P} -state. Starting at a \mathcal{P} -state S , iterated application of \mathcal{P} yields a sequence $SS'S'' \dots$ of \mathcal{P} -states, called a *computation of \mathcal{P} with initial state S* . Together with a set *init* of \mathcal{P} -states, \mathcal{P} defines a set of computations. This set of computations can be represented by help of an *initialized transition system*, i.e. a set \mathbf{S} of states, a *next state* function $\tau : \mathbf{S} \rightarrow \mathbf{S}$, and a subset *init* $\subseteq \mathbf{S}$ of *initial* states.

In this paper we consider ASM programs that describe deterministic, non-reactive, sequential algorithms.

This is an intuitive notion; the above examples showed that such algorithms differ substantially from means to generate computable functions.

Nevertheless, an algorithm characterizes a set of behaviors that can

be represented by an initialized transition system. For the sake of a better intuitive understanding of algorithms we search for typical properties of “algorithmic” transition systems. More concretely, we compile a list of four properties and motivate in detail that without any reasonable doubt, each deterministic, non-reactive, sequential algorithm meets these properties. Three of these properties are of merely technical nature: Essentially, they require states and steps be closed under isomorphism. The decisive fourth property only requires that an algorithm must be representable in finite terms.

Gurevich’s fundamental Theorem states a surprising fact: Every initialized transition system that meets the above mentioned four properties can be generated by an ASM program.

This implies that ASM programs are a most general formalism to define algorithms.

The rest of the paper is organized as follows: Three examples of algorithms are discussed in Chapter 2. Each algorithm refers to a specific aspect of the ASM formalism.

Chapter 3 starts with initialized transition systems, a common framework for any kind of operational models of sequential behaviour, and poses the central question of this paper: Which transition systems are “algorithmic”? We discuss four requirements for “algorithmic” transition systems.

Chapter 4 describes the operational model of sequential, deterministic, non-reactive ASM programs.

Chapter 5 reformulates Gurevich’s fundamental Theorem: The four necessary requirements for algorithmic transition systems are sufficient for algorithms: To each transition system \mathcal{A} that meets the requirements, there exists an ASM program \mathcal{M} such that the runs of \mathcal{A} and \mathcal{M} coincide.

Chapter 6 will finally relate the constructs of this paper to the conventional theory of computing.

We will not discuss the impressive collection of industrial projects based on ASM. For this issue we refer to survey [Bo02] and Gurevich’s website <http://research.microsoft.com/foundations/>.

2 Examples

2.1 *The tangent algorithm*

The reader certainly remembers this algorithm from school: Given a circle C with center p , and a point q outside C . Wanted: a tangent

of C through q (cf. Fig. 1).

The well-known solution: Construct the half-way point, r , of p and

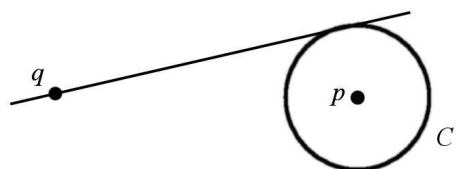


Fig. 1 The problem of the tangent algorithm

q . Construct a circle, D , with center r and p (as well as q) on its surface. C and D intersect in two points. Choose one of them, s . The wanted tangent is uniquely determined by q and s . (cf. Fig. 2).

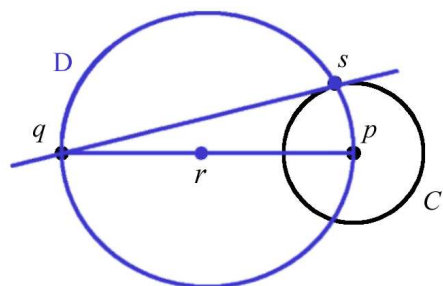


Fig. 2 The solution to the tangent algorithm

This algorithm doesn't fix representations for points, circles and lines. A concrete behavior may do with any representation. Here a choice of representations:

- *analytical geometry*: A point is a pair (x, y) of real numbers; a circle and a line is the set of all solutions of equations formed $((y + y_0) - (x + x_0))^2 = c$, and $y = ax + b$, respectively.

- *computing*: A point is a pair (x, y) of integers with $min \leq x, y \leq max$; solutions of the above equations for circles and lines must be approximated to integers.

- *school*: A point is a black dot on white paper; a circle and a line consists of grains of lead, derived from a pencil by help of rulers.

- *axiomatic geometry*: Axioms characterize points, circle and lines are characterized as sets of points that meet specific properties.

This example shows that the data of an algorithm are not necessarily uniquely or formally represented. Operations and algorithms using these data can nevertheless be specified in a unique, symbolic way.

For the tangent algorithm, we start out with symbols p, q, r, s for points, M for sets of point, C and D for circles, and l for lines. We furthermore employ symbols for functions such as *outside*, *halfway*, *circle*, *intersect*, *pick* and *makeline*. The semantics of symbols is obvious in the representation of the tangent algorithm as given in Fig. 3. This algorithm operates on initial states, S_0 , where the values of p, c

```

input(p,C,q);
if q outside C then
  r := halfway(p,q);
  D := circle(r,p);
  M := intersect(C,D);
  { |M| = 2 }
  s := pick(M);
  l := makeline(q,s);
output(l);

```

Fig. 3 The tangent algorithm

and q are the input values. The other variables' values are initially irrelevant. S_0 may include any number of additional variables, with any initial value. The program of Fig. 3 then yields a state where the value of l is the wanted tangent. This succeeds for each of the above representations of points, circles and lines. The algorithm in Fig. 3 does not follow the syntax of ASM. The algorithm nevertheless exemplifies the decisive aspect of ASM, outlined in the introduction already: A liberal notion of state is combined with a formal, operational, symbol manipulating notion of algorithm. The next chapter gives the details.

2.2 A Data Base

In an abstract setting, a state of a database can be described as a function $f : var \rightarrow val$, as suggested in the introduction already. An examples is the function

$$salary : employees \rightarrow \mathbb{N} \quad (3)$$

that assigns a salary to each employee of a company. The company may run a policy to increase a salary by 1000 for extraordinary achievements, such as authoring a paper. The function

$$increase : \mathbb{N} \rightarrow \mathbb{N} \quad (4)$$

with $increase(n) = n + 1000$ will be used for this purpose.

The company may store an author in a variable, *author*. A variable can be conceived as the special case of a 0-ary function.

The algorithm to increase the recent author's salary then reads

$$\text{salary}(\text{author}) := \text{increase}(\text{salary}(\text{author})). \quad (5)$$

In a general setting, to update a state means to update a function at a specific position.

A couple of extensions to (5) will serve as a running example throughout this paper.

2.3 Graph reachability

Let G be a directed graph, and let *root* be a node of G . Here an algorithm that computes a unary predicate R to discriminate the nodes reachable from the root:

Initially $R(x)$ if and only if $x = \text{root}$. Iterate the following step: For all arcs $x \rightarrow y$ with $R(x)$ and $\neg R(y)$, extend R by y .

The amount of work executed in one step is unbounded: The number of arcs $x \rightarrow y$ in G with $R(x)$ and $\neg R(y)$ depends on the graph G . Thus, the number of locations affected in one step is not bounded. The algorithm therefore does not belong to the most elementary class of Abstract-State Machines.

3 Some Basics of Sequential, Deterministic, Non-Reactive Algorithms

3.1 Transition Systems

A sequential, deterministic, non-reactive algorithm is abstractly given by three constituents, *states*, *init* and τ , where *states* is a set of admissible states, *init* is a subset of *initial* states and $\tau : \text{states} \rightarrow \text{states}$ is a function that assigns each state S its successor state $\tau(S)$. Neither *states* nor *init* are required to be finite. One may wonder about *terminal* states. A state S intended to be terminal may be characterized by $\tau(S) = S$; i.e. we apply a “trick” well known from UNITY, TLA, and other formalisms: A terminal state is iterated infinitely often. The three components constitute what is usually called a transition system:

Definition 1. *Let states be a set, $\text{init} \subseteq \text{states}$, and $\tau : \text{states} \rightarrow \text{states}$. Then $\mathcal{A} = (\text{states}, \text{init}, \tau)$ is an initialized transition system.*

A transition system \mathcal{A} can be conceived as a graph. Its nodes are the states; its directed arcs are defined by τ . Each node is the source of exactly one arc. Paths in this graph describe *runs* of \mathcal{A} :

Definition 2. Let $\mathcal{A} = (\text{states}, \text{init}, \tau)$ be an initialized transition system. Let $w = S_0 S_1 S_2 \dots$ be an infinite sequence of states $S_i \in \text{states}$, with $S_0 \in \text{init}$ and $S_{i+1} = \tau(S_i)$ for $i = 0, 1, 2, \dots$. Then w is a run of \mathcal{A} .

Not each initialized transition system \mathcal{A} constitutes an algorithm. A first idea to discriminate algorithms is to require a finite, symbolic representation of \mathcal{A} . We require less: To describe, in finite and symbolic terms, how a state S can operationally, effectively be transformed into its successor state, $\tau(S)$. To this end, we assume states be mathematical objects in a most general form, and the sets of states and initial states be closed under isomorphism.

3.2 States

The database example 2.2 has shown already that a state of a database can be described by a set of *mappings* f_1, \dots, f_k . Each f_i is of the form

$$f_i : \text{var}_i \rightarrow \text{val}_i \quad (6)$$

where var_i is a set of *variables* or *registers*, and val_i is a set of *values*. For the sake of simplicity we assume a common *universe*, U , to cover all domains and ranges of all f_i . We will be specific on the arity n_i for each f_i , i.e. the numbers of components of each argument: f_i is of the form

$$f_i : U^{n_i} \rightarrow U. \quad (7)$$

Each f_i of shape (6) can easily be brought into shape (7), provided U is chosen sufficiently large and includes an *undefined* symbol, serving as the value of $f_i(u_1, \dots, u_{n_i})$ in (7), whenever in (6), (u_1, \dots, u_{n_i}) is not in the domain of f_i .

Summing up, a state S will be conceived as a *homogeneous algebra*

$$S = (U, f_1, \dots, f_k), \quad (8)$$

where each function f_i has a specific arity, n_i .

We could have employed *heterogeneous* algebras instead, where each f_i has its own domain and range. This would come along with some amount of technical overhead, but without conceptual benefit.

For reasons to be explained later, every state S contains the propositional constants *true* and *false*, as well as the above mentioned constant *undefined*. The usual propositional combinators, such as NOT and AND are unary and binary combinators of S , respectively (with $\text{NOT}(u) = \text{AND}(u, v) = \text{undef}$ for $u, v \notin \{\text{true}, \text{false}\}$). We denote such states as ASM-algebras:

Definition 3. *An algebra of the form*

$$S = (U \cup \{\text{true}, \text{false}, \text{undefined}\}, \\ \text{true}, \text{false}, \text{undefined}, \text{NOT}, \text{AND}, f_1, \dots, f_n)$$

is an ASM-algebra. As a shorthand we write $S = (U, f_1, \dots, f_n)_{ASM}$.

As a running example, we extend the database of Chapter 2.1: Let *me*, the author of this paper, and *you*, its reader, be elements of a universe U , together with the natural numbers \mathbb{N} .

Let *salary* be a function that assigns to *me* and *you* an integer:

$$\text{salary}(\text{me}) = \text{salary}(\text{you}) = 5000.$$

For technical reasons, let $\text{salary}(u) = \text{undefined}$ for all $u \in U \setminus \{\text{me}, \text{you}\}$. Furthermore, let *increase* be a function that increases every integer n by 1000, i.e. $\text{increase}(n) = n + 1000$. For technical reasons, let $\text{increase}(u) = \text{undefined}$ for all $u \in U \setminus \mathbb{N}$. Then the ASM-algebra

$$S_0 = (\{\text{me}, \text{you}\} \cup \mathbb{N}, \text{me}, \text{you}, \text{salary}, \text{increase})_{ASM} \quad (9)$$

is a state.

The Tangent Algorithm of Chapter 2.1 may be applied to various different initial states. Each initial state fixes a circle C , the points p and q , as well as the operations *outside*, *halfway*, *circle*, *intersect*, *pick* and *makeline*. All these items remain untouched during a run. In addition, variables for the points r , s , the set of points M , and the line l are being updated during a run.

Different initial states may represent and initialize all mentioned items differently, as explained in Chapter 2 already. But each constant and operation symbol addresses the same *kind* of items in each initial state. In technical terms, the initial states are all algebras with the same signature.

3.3 Signatures

As explained above, each state contains the values *true*, *false* and *undefined* as constants. Hence each corresponding signature must contain corresponding symbols.

Definition 4. *A signature of the form*

$$\Sigma = (tt, ff, undef, \neg, \wedge, fct_1, \dots, fct_k, 0, 0, 0, 1, 2, n_1, \dots, n_k)$$

is called an ASM-signature. As a shorthand we write $\Sigma = (fct_1, \dots, fct_k, n_1, \dots, n_k)_{ASM}$.

In our running example, consider the signature

$$\Sigma_0 = (author, reader, fct_1, fct_2, 0, 0, 1, 1)_{ASM}. \quad (10)$$

The above algebra S_0 (as in (9)) is a Σ_0 -algebra. We construct a different Σ_0 -algebra, including two other persons: *my-boss* and *your-boss*. Additionally, take the function *boss-salary*, with $boss\text{-}salary(my\text{-}boss) = boss\text{-}salary(your\text{-}boss) = 7000$ as $boss\text{-}salary(u) = undefined$ for all $u \in U \setminus \{my\text{-}boss, your\text{-}boss\}$. Additionally, we employ the *increase*-function of S_0 . Then

$$S_1 = (\{my\text{-}boss, your\text{-}boss\} \cup \mathbb{N}, \\ my\text{-}boss, your\text{-}boss, boss\text{-}salary, increase)_{ASM} \quad (11)$$

is a Σ_0 -algebra, too. Hence both S_0 and S_1 can serve as initial states of the same algorithm.

3.4 The First Requirement: A state is an algebra

In all above examples, the next state function τ updates functions f at some argument locations (u_1, \dots, u_n) . But τ does not produce new functions, nor does τ delete existing functions. (Recursive algorithms, which very well may produce or delete existing functions, will be considered elsewhere).

This observation is generalized to the class of algorithms considered here, and we assume each state S and its successor state $\tau(S)$ have the same signature. Consequently, all states of a run have the same signature. Together with (4), this implies that all reachable states have the same signature. Each signature is an ASM-signature, of course.

For a transition system $\mathcal{A} = (states, init, \tau)$ to be an algorithm, we do not require the set *states* to contain the reachable states only: It is frequently simpler to start out with a more comprehensive set

of states that can be characterized more easily. But we require the signature of each state to be equal to the signature of the reachable states. This is a technicality that does not exclude any reasonable algorithm.

In our running examples, S_0 and S_1 may be initial states of an algorithm with signature Σ_0 .

As a variant, one may wish to extend S_0 and S_1 by an additional person, e.g. the journal manager. This would result in states that are Σ_1 -algebras, where

$$\Sigma_1 = (\text{author}, \text{reader}, \text{manager}, \text{fct}_1, \text{fct}_2, 0, 0, 0, 1, 1)_{ASM}. \quad (12)$$

Σ_1 -states and Σ_0 -states can not occur in runs of the same algorithm, however.

As a conclusion, all states S of an algorithmic transition system should be algebras with the same signature:

Definition 5. Let $\mathcal{A} = (\text{states}, \text{init}, \tau)$ be an initialized transition system. \mathcal{A} is said to be ASM-algebraic iff there exists an ASM-signature Σ such that each $S \in \text{states}$ is a Σ -algebra. Σ is the signature of \mathcal{A} .

First Requirement: An algorithmic transition system is ASM-algebraic.

3.5 The Second Requirement: A run has a universe

As we have seen above already, the signature of a state S is preserved by $\tau(S)$. Now we will motivate why τ should even more retain the universe. This is quite obvious for the tangent algorithm (2.1): This algorithm updates the variables r , s , D , M and l . Objects and operations either follow the requirements of analytical or axiomatic geometry, or employ rounding algorithms, or are just symbolic. We even considered black dots and lines on white paper. Initially chosen, the kind of objects and operations remains fixed during a run. As an example, a run $S_0 S_1 \dots$ where S_0 represents points as pairs of reals should not continue with S_1 , characterizing points axiomatically. Nevertheless, one run may represent points as pairs of reals, whereas an other run represents points as terms over a fixed alphabet.

Summing up, all states of a run employ the same domain, the *domain of the run*. Different runs may employ different domains. Hence we require the next-state function τ to be *domain preserving*:

Definition 6. Let Σ be a signature, let states be a set of Σ -algebras, and let $\tau : \text{states} \rightarrow \text{states}$. Then τ is domain preserving iff for each $S \in \text{states}$ S and $\tau(S)$ have the same domain.

Second requirement: The next-state function τ of an ASM-algebraic algorithmic transition system $\mathcal{A} = (\text{states}, \text{init}, \tau)$ is domain preserving.

For our running example, the successor state $\tau(S_0)$ of S_0 cannot be the state S_1 , because the function τ can not produce *my-boss* and *your-boss*. However, τ may increase my salary by the help of the program

$$\text{fct}_1(\text{author}) := \text{fct}_2(\text{fct}_1(\text{author})). \quad (13)$$

The resulting state, S_2 , is

$$S_2 = (\{me, you\} \cup \mathbb{N}, me, you, salary', increase)_{ASM} \quad (14)$$

with $salary'(me) = 6000$ and, $salary'(u) = salary(u)$ for all $u \in U \setminus \{me\}$. S_2 is a Σ_0 ASM - algebra. Its carrier is identical to the carrier of S_1 . The successor $S_3 := \tau(S_2)$ of S_2 may declare *you* the author and *me* the reader (of an other paper, say). This is achieved by the program

$$(\text{author}, \text{reader}) := (\text{reader}, \text{author}). \quad (15)$$

and yields the state

$$S_3 = (\{me, you\} \cup \mathbb{N}, you, me, salary', increase)_{ASM}. \quad (16)$$

Notice that S_2 and S_3 differ only in the order of the constants *me* and *you*. We may now apply program (13) to S_3 . This yields the state

$$S_4 = (\{me, you\} \cup \mathbb{N}, you, me, salary'', increase)_{ASM} \quad (17)$$

with $salary''(you) = 6000$ and $salary''(u) = salary'(u)$ for all $u \in U \setminus \{me\}$.

As the underlying signature Σ_0 is an ASM-signature, we may formulate conditional assignment statements such as

$$\begin{aligned} &\text{if } f_1(\text{author}) = f_1(\text{reader}) \text{ then} \\ &f_1(\text{author}) := f_2(f_1(\text{author})). \end{aligned} \quad (18)$$

Summing up, in our running example let $\text{states}_0 = \{S_0, \dots, S_4\}$, let $\text{init}_0 = \{S_0, S_1\}$ with $\tau_0(S_1) = S_1$, $\tau_0(S_0) = S_2$, $\tau_0(S_2) = S_3$ and $\tau_0(S_3) = \tau_0(S_4) = S_4$. Then

$$\mathcal{A}_0 = (\text{states}_0, \text{init}_0, \tau_0) \quad (19)$$

is an initialized transition system. It has two runs, $S_1S_1\dots$ and $S_0S_2S_3S_4S_4\dots$. It fulfills the first requirement because each state $S \in \text{states}$ is a Σ_0 -algebra. Inspection of all states reveals that \mathcal{A}_0 fulfills also the second requirement, because the domains of S_0 , S_2 , S_3 and S_4 are all the same.

3.6 The Third Requirement: The transition system is closed under isomorphism

Remember our ultimate goal: The discrimination of "algorithmic" transition systems $\mathcal{A} = (\text{state}, \text{init}, \tau)$, where the next-state function τ should be representable by finitely many Σ -terms (with Σ the signature of \mathcal{A}). Σ -terms cannot discriminate isomorphic Σ -algebras. Hence, \mathcal{A} would fail to be algorithmic in case the set of states includes an algebra, R , but not an algebra S isomorphic to R . Furthermore, the next-state function τ cannot discriminate isomorphic states. This is the third requirement:

Definition 7. Let $\mathcal{A} = (\text{states}, \text{init}, \tau)$ be an ASM-algebraic transition system with signature Σ , where for all pairs R, S of isomorphic Σ -algebras holds:

1. $R \in \text{states}$ iff $S \in \text{states}$ and $R \in \text{init}$ iff $S \in \text{init}$.
2. If $h : R \rightarrow S$ is an isomorphism, then $h : \tau(R) \rightarrow \tau(S)$ is an isomorphism, too.

Then \mathcal{A} is isomorphism closed.

As an example, in \mathcal{A}_0 of (19) extend states_0 and init_0 by all states isomorphic to the given states, and extend τ_0 canonically to the new states. The resulting transition system is isomorphism closed.

Third Requirement: An algorithmic transition system is isomorphism closed.

3.7 Updates and Update sets

This is the decisive requirement of the ASM approach: Finitely many Σ -terms suffice to characterize the runs of an algorithm \mathcal{A} . Notice that we do not require finitely many terms to characterize \mathcal{A} itself, i.e. the set of states, the subset of initial states, and the next-state function, τ . We require less, instead: A state S and its successor state $\tau(S)$ are identical up to a "small" number of functions f and

values $f(u_1, \dots, u_n)$. An algorithm characterizes these positions together with the updated values.

The ASM formalism employs *variable* functions: For a function f and a value u , the value $f(u)$ may differ from state to state, just like the value of a conventional variable. In our running example, the value of the term $salary(author)$ evaluates to 5000 in state S_0 , and to 6000 in states S_2 and S_4 . But observe the fundamental difference between S_2 and S_4 : The argument variable, $author$, evaluates to me in S_2 and to you in S_4 !

Hence, an update of a state S is given by three parameters: a function symbol, f , an n -tuple \mathbf{u} of arguments for f , and the new value, v , for $f(\mathbf{u})$.

Definition 8. 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 initialized transition system (19) yields Σ_0 -updates over $\{me, you\} \cup \mathbb{N}$: The update $(salary, (me), 6000)$ fixes my salary at 6000. The update $(reader, (), me)$ turns me into a reader.

The next-state function τ usually updates more than one position: The step from a state S to $\tau(S)$ is defined by an *update set*:

Definition 9. Let $\mathcal{A} = (\text{states}, \text{init}, \tau)$ be an initialized, ASM - algebraic transition system with signature Σ and let $S \in \text{states}$ with universe U .

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

For example, in the transition system (19),

$$\begin{aligned} \Delta(\tau, S_0) &= \{(f_1, (me), 6000)\}, \\ \Delta(\tau, S_2) &= \{(author, (), you), (reader, (), me)\}, \\ \Delta(\tau, S_3) &= \{(f_1, (you), 6000)\}, \text{ and} \\ \Delta(\tau, S_1) &= \Delta(\tau, S_4) = \emptyset. \end{aligned}$$

Here some technical examples, to be re-considered later: Let $\Sigma = (\text{const}, \text{fct}, 0, 1)$ be a signature with a constant symbol, const , and a unary function symbol, fct . For each $n \in \mathbb{N}$, let $S_n = (\mathbb{N}, n, \text{suc})$ be a Σ -algebra, with $\text{suc} : \mathbb{N} \rightarrow \mathbb{N}$ the successor function (i.e. $\text{suc}(i) := i + 1$). Let $\text{state} = \{S_0, S_1, \dots\}$, $\text{init} = \{S_0\}$, and $\tau : \text{state} \rightarrow \text{state}$

with $\tau(S_n) = S_{n+1}$, for all $n \in \mathbb{N}$. Then $\mathcal{A} = (\text{state}, \text{init}, \tau)$ is an initialized transition symbol, and for each $n \in \mathbb{N}$ holds:

$$\Delta(\tau, S_n) = \{(const, (), n + 1)\}.$$

As a second example, let $\Sigma = (const, fct_1, fct_2, 0, 1, 1)$ be a signature with one constant and two unary function symbols. For each $n \in \mathbb{N}$, let:

$$\begin{aligned} g_n : \mathbb{N} &\rightarrow \mathbb{N} \\ i &\mapsto i + 1 \text{ if } i \leq n \\ i &\mapsto 0 \quad \text{if } i > n \end{aligned}$$

and let $S_n = (\mathbb{N}, n, \text{succ}, g_n)$. Let $\text{states} = \{S_0, S_1, \dots\}$, $\text{init} = \{S_0\}$ and $\tau(S_n) = S_{n+1}$. Then $\mathcal{A} = (\text{states}, \text{init}, \tau)$ is an initialized transition system, and for each $n \in \mathbb{N}$ holds:

$$\Delta(\tau, S_n) = \{(const, (), n + 1), (fct_2, (n + 1), n + 2)\}.$$

3.8 The Update Lemma

The following Lemma characterizes the functions $f_{\tau(S)}$ in terms of f_S and $\Delta(\tau, S)$:

Lemma 1 (Update Lemma). *Let $\mathcal{A} = (\text{states}, \text{init}, \tau)$ be an initialized transition system, let Σ be a signature, let f be a unary function symbol, and let $S \in \text{states}$ be a Σ -algebra with carrier U .*

Then $f_{\tau(S)}$ is given by

$$\begin{aligned} f_{\tau(S)} : U^n &\rightarrow U \\ \mathbf{u} &\mapsto v \quad , \text{ if } (f, \mathbf{u}, v) \in \Delta(\tau, S) \\ \mathbf{u} &\mapsto f_S(\mathbf{u}), \text{ otherwise.} \end{aligned}$$

This Lemma follows directly from the definition of $\Delta(\tau, S)$ ■.

3.9 The Δ -Isomorphism Lemma

Isomorphic states have isomorphic update sets. More precisely: An isomorphism h between two states R and S expands to the successor states of R and S (cf. the fourth requirement) if and only if h defines a bijection on the update sets of R and S . As technicality we define the homomorphic image of an update:

Definition 10. Let U, \mathcal{V} be sets, let $\mathbf{u} = (u_1, \dots, u_n) \in U^n$, let $v \in U$ and let $h : U \rightarrow \mathcal{V}$.

1. Let $h(\mathbf{u}) =_{def} (h(u_1), \dots, h(u_n))$.
2. For an update $\delta = (f, \mathbf{u}, v)$, let $h(\delta) =_{def} (f, h(\mathbf{u}), h(v))$.

Lemma 2 (Δ -Isomorphism Lemma). Let $\mathcal{A} = (\text{states}, \text{init}, \tau)$ be an initialized transition system with signature Σ , let $R, S \in \text{states}$ and let $h : R \rightarrow S$ be an isomorphism. Then $\delta \in \Delta(\tau, R)$ iff $h(\delta) \in \Delta(\tau, S)$.

Proof. $\delta = (f, \mathbf{u}, v) \in \Delta(\tau, R)$ iff $f_R(\mathbf{u}) \neq f_{\tau(R)}(\mathbf{u}) = v$ iff $f_S(h(\mathbf{u})) \neq f_{\tau(S)}(h(\mathbf{u})) = h(v)$ iff $(f, h(\mathbf{u}), h(v)) \in \Delta(\tau, S)$ iff $h(\delta) \in \Delta(\tau, S)$.

1st equivalence: Def $\Delta(\tau, R)$, 3.8.

2nd equivalence: h is an isomorphism, by the Lemma's assumption.

3rd equivalence: Def $\Delta(\tau, S)$. 3.8.

4th equivalence: Def $h(\delta)$, above. ■

3.10 The Fourth Requirement: Finitely many terms suffice

An initialized transition system $\mathcal{A} = (\text{states}, \text{init}, \tau)$ with signature Σ is algorithmic only if there exists a finite set T of Σ -terms such that for all $S \in \text{states}$, the step from S to $\tau(S)$ can be described by $\Delta(\tau, S)$. So, we may choose T to describe $\Delta(\tau, S)$, for all $S \in \text{states}$. By no means we want to restrict how T would represent $\Delta(\tau, S)$: Here a necessary requirement: If two states R and S have different sets of τ -updates (i.e. $\Delta(\tau, R) \neq \Delta(\tau, S)$), then R and S interpret at least one term $t \in T$ differently, i.e. $t_R \neq t_S$:

Definition 11. Let $\mathcal{A} = (\text{states}, \text{init}, \tau)$ be an initialized ASM - algebraic transition system with signature Σ . Let $T \subseteq T_\Sigma$ such that for all $R, S \in \text{states}$: If for all $t \in T$ the equation $t_R = t_S$ holds, then $\Delta(\tau, R) = \Delta(\tau, S)$. In this case, T is called characteristic for τ .

τ is bounded exploration if there exists a finite characteristic set T of terms.

Fourth Requirement: An ASM-algebraic algorithmic transition system $\mathcal{A} = (\text{states}, \text{init}, \tau)$ with signature Σ is bounded exploration.

The isomorphic closure $(\mathcal{A}_0)_{ASM}$ of our running example \mathcal{A}_0 does not meet the fourth requirement. We prove this by help of the following Σ_0 -algebra,

$$S_5 = (\{me, you\} \cup \mathbb{N} \setminus \{0, \dots, 999\}, \quad (20) \\ you, me, salary'', increase)_{ASM}$$

where $salary''$ has been defined for (17) already. The function h with

$$h(me) := you, h(you) := me, \text{ and } h(n) := n + 1000 \quad (21)$$

defines an isomorphism

$$h : S_0 \rightarrow S_5. \quad (22)$$

Hence S_5 is a state of \mathcal{A}_0 , according to the third requirement. Furthermore, $\delta = (fct_1, (me), 6000) \in \Delta(\tau, S_0)$. Hence,

$$h(\delta) \in \Delta(\tau, S_5) \quad (23)$$

by the third requirement, and the Δ -isomorphism lemma, 3.9.

Now we relate S_5 and S_4 : For each term $t \in T_\Sigma$ holds: $t_{S_5} = t_{S_4}$. The fourth requirement requires $\Delta(\tau, S_5) = \Delta(\tau, S_4)$. But $\Delta(\tau, S_4) = \emptyset \neq \Delta(\tau, S_5)$ with (23).

Consequently, the behavior described by \mathcal{A} , i.e. the set of two runs $S_1S_1\dots$ and $S_0S_2S_3S_4S_4\dots$ is not algorithmic. This comes up as a surprise: Intuitively, one expects this - finite - behavior to be algorithmic, of course.

The problem is due to the limited expressiveness of the signature Σ_0 : Terms of Σ_0 cannot distinguish S_4 and some isomorphic images of S_0 , e.g. S_5 . The states S_0 and S_5 differ wrt the salary of the authors and readers. But the signature Σ_0 offers no symbols to express salaries. This defect can be repaired with the help of an additional symbol, *basic*, in the signature, denoting the basic salary:

$$\Sigma'_0 = (author, reader, basic, fct_1, fct_2, 0, 0, 0, 1, 1). \quad (24)$$

We extend the algebras S_0, \dots, S_4 of (19) correspondingly, gaining states S'_0, \dots, S'_4 , with $basic_{S'_0} = \dots = basic_{S'_4} = 5000$. For example,

$$\begin{aligned} S'_0 &= (\{me, you\} \cup \mathbb{N}, me, you, 5000, salary, increase)_{ASM}, \\ S'_4 &= (\{me, you\} \cup \mathbb{N}, you, me, 5000, salary'', increase)_{ASM}. \end{aligned}$$

These states in turn yield the transition system

$$\mathcal{A}'_0 = (states'_0, init'_0, \tau'_0). \quad (25)$$

The state S'_5 with $basic_{S'_5} = 5000$ is not isomorphic to S'_0 and hence is no state of $(\mathcal{A}'_0)_{ASM}$. We may choose $basic_{S'_5} = 6000$, in which case S'_5 is isomorphic to S'_0 and hence a state of $(\mathcal{A}'_0)_{ASM}$. But then $basic_{S'_5} = 6000 \neq 5000 = basic_{S'_4}$, hence a characteristic set T exists with $basic \in T$. In fact,

$$T = \{basic, salary(author), salary(reader)\}$$

is characteristic for τ'_0 : For all $R, S \in \{S'_0, \dots, S'_4\}$ and all $R' \simeq R$ and $S' \simeq S$ there exists a term $t \in T$ such that $t_{R'} \neq t_{S'}$. Hence T is characteristic for τ_0 . Consequently, $(\mathcal{A}'_0)_{ASM}$ is bounded exploration.

3.11 Algorithmic Transition Systems

We are now prepared to define the central notion of this paper: algorithmic transition systems. A transition system is algorithmic if and only if it meets the above four requirements:

Definition 12. *Let $\mathcal{A} = (\text{states}, \text{init}, \tau)$ be an initialized, ASM - algebraic, isomorphism closed transition system . Furthermore let τ be domain preserving and bounded exploration. Then \mathcal{A} is algorithmic.*

4 Abstract-State Machines

Here we aim at the programming language to represent algorithmic transition systems: Elementary operations on symbolic representations are sought, together with controlled consecutive execution of operations.

As we have seen above, an algorithmic transition system \mathcal{A} comes together with a signature, Σ . We will employ Σ -terms to construct a program that describes (the next-step function τ of) \mathcal{A} . This kind of programs are *abstract state machines* (ASM).

To start with, we re-consider *updates* as defined in Section 3.7. Applied to a state S , an update $\delta(f, \mathbf{u}, v)$ updates the value of the location $f_S(\mathbf{u})$ by v . With Σ -terms t_0, \dots, t_k such that $\mathbf{u} = (t_{1S}, \dots, t_{kS})$ and $v = t_{0S}$, this update can syntactically be represented by the *assignment*

$$f(t_1, \dots, t_n) := t_0.$$

An algorithmic transition system \mathcal{A} consists of infinitely many states S and steps $S \rightarrow \tau(S)$. It will turn out that finitely many assignments suffice to describe the next-state function τ of \mathcal{A} . Assignments r will be *guarded* in general, i.e. are formed

$$\text{if } \beta \text{ then } r. \tag{26}$$

An ASM program is just a set of guarded assignments. In a step, all guards are evaluated. Assignments with true guard are executed in parallel.

4.1 Assignments

We give a formal definition of assignments. A set of assignments is *consistent* or *inconsistent* in a state S :

Definition 13. Let Σ be a signature.

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

4.2 ASM programs

We are now prepared to define the central syntactical items: sequential, bounded ASM-programs:

Definition 14. Let Σ be an ASM-signature.

1. Let r be a Σ -assignment and let β be a Boolean term over Σ . Then

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

is a guarded Σ -assignment.

2. Let q_1, \dots, q_m be guarded Σ -assignments. Then

$$\text{par } q_1, \dots, q_n \text{ endpar}$$

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

Notation We frequently write r for the guarded assignment **if true then r**.

4.3 Semantics of assignments

Each set Z of (unconditional) assignments defines for each state S a set of updates, provided Z is consistent at S :

Definition 15. 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

$$f^{Z,S} : \quad U^k \rightarrow U$$

$$(t_{1S}, \dots, t_{kS}) \mapsto t_{0S} \text{ iff } f(t_1, \dots, t_k) := t_0 \in Z$$

$$\mathbf{u} \mapsto f_S(\mathbf{u}), \text{ otherwise}$$

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

4.4 Semantics of ASM-programs

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

Definition 16. Let Σ be an ASM-signature and let

$$\begin{array}{l} \mathcal{M}: \text{par if } \beta_1 \text{ then } r_1 \\ \quad \vdots \\ \quad \text{if } \beta_m \text{ then } r_m \\ \text{endpar} \end{array}$$

be an ASM-program over Σ . Then the semantic function $\text{sem}_{\mathcal{M}} : \text{Alg}_{\Sigma} \rightarrow \text{Alg}_{\Sigma}$ is defined as follows:

For $S \in \text{Alg}_{\Sigma}$ let $Z := \{r_i \mid \beta_{iS} = \text{true}, 1 \leq i \leq m\}$. Then

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

Notice that an ASM-program \mathcal{M} consists only of Σ -terms and keywords $:=$, *if*, *then*, *par*, *endpar*. This is pure syntax. \mathcal{M} is applied to a semantical construct, a Σ -algebra S , that cannot necessarily be represented in finite, symbolic terms.

Iteration of $\text{sem}_{\mathcal{M}}$ now yields the *runs* of an ASM-program \mathcal{M} :

Definition 17. Let Σ be an ASM-signature and let

$$\mathcal{M} : \text{par if } \beta_1 \text{ then } r_1, \dots, \text{if } \beta_m \text{ then } r_m \text{ endpar}$$

be an ASM-program over Σ . Let $w = S_0 S_1 \dots$ be a sequence of Σ -algebras with $S_{i+1} = \text{sem}_{\mathcal{M}}(S_i)$ for $i = 0, 1, \dots$. Then w is a run of \mathcal{M} .

This completes syntax and semantics of ASM-programs. It is almost trivial to observe that each ASM-program \mathcal{M} defines an algorithmic transition system \mathcal{A} such that the runs of \mathcal{M} and \mathcal{A} coincide. The reverse property, i.e. each algorithmic transition system defines a corresponding program, is to be shown next.

5 Gurevich's sequential ASM-Theorem

5.1 What is to be shown

Theorem. Let $\mathcal{A} = (\text{states}, \text{init}, \tau)$ be an algorithmic transition system. Then there exists an ASM-program \mathcal{M} with $\tau = \text{sem}_{\mathcal{M}}$.

Assumptions For the entire rest of this chapter we assume an algorithmic transition system

$$\mathcal{A} = (\text{states}, \text{init}, \tau)$$

with signature Σ and a finite set T of characteristic terms for τ .

Notation For each state S of \mathcal{A} let $T_S =_{\text{def}} \{t_s \mid t \in T\}$.

Definition 18. An ASM-program \mathcal{M} is characteristic for \mathcal{A} iff all terms occurring in \mathcal{M} are in T .

5.2 Updates of ASM-programs

Each ASM-program assigns to each state a set of updates. Isomorphic states yield isomorphic update sets. States that cannot be distinguished by characteristic terms yield equal update sets for characteristic programs.

First we construct an update for each assignment and each state of \mathcal{A} :

Definition 19. Let $\mathbf{q} =_{\text{def}} f(t_1, \dots, t_n) := t_0$ be a Σ -assignment, and let S be a Σ -algebra. Then we define the Σ -update \mathbf{q}_S by

$$\mathbf{q}_S =_{\text{def}} (f, (t_{1S}, \dots, t_{nS}), t_0).$$

Each ASM-program yields a set of updates for each state:

Definition 20. Let \mathcal{M} be an ASM-program over a signature Σ , let $S \in \text{states}$ and let $S' = \text{sem}_{\mathcal{M}}(S)$. Let $\text{delta}(\mathcal{M}, S) =_{\text{def}} \{(f, \mathbf{u}, v) \mid (f, \mathbf{u}, v) \text{ is a } \Sigma\text{-update with } f_S(\mathbf{u}) \neq f_{S'}(\mathbf{u}) = v\}$.

The *delta-Isomorphism-Lemma* shows that isomorphic states yield isomorphic updates:

Lemma 3 (delta-Isomorphism-Lemma). Let \mathcal{M} be an ASM-program over Σ , let $R, S \in \text{states}$ and let $h : R \rightarrow S$ be an isomorphism. Then $\delta \in \text{delta}(\mathcal{M}, R)$ iff $h(\delta) \in \text{delta}(\mathcal{M}, S)$.

Proof. Let $\delta = (f, \mathbf{u}, v)$, $R' =_{\text{def}} \text{sem}_{\mathcal{M}}(R)$ and $S' =_{\text{def}} \text{sem}_{\mathcal{M}}(S)$. Then $\delta \in \text{delta}(\mathcal{M}, R)$ iff δ is a Σ -update with $f_R(\mathbf{u}) \neq f_{R'}(\mathbf{u}) = v$ iff there exists an assignment formed $f(t_1, \dots, t_n) := t_0$ in \mathcal{M} with $(t_{1R}, \dots, t_{nR}) = \mathbf{u}$ and $t_{0R} = v$ and $f_R(\mathbf{u}) \neq f_{R'}(\mathbf{u})$ iff there exists an assignment formed $f(t_1, \dots, t_n) := t_0$ in \mathcal{M} with $(t_{1S}, \dots, t_{nS}) = h(\mathbf{u})$ and $t_{0S} = h(v)$ and $f_R(h(\mathbf{u})) \neq f_{R'}(h(\mathbf{u}))$ iff $h(\delta)$ is a Σ -update with $f_S(h(\mathbf{u})) \neq f_{S'}(h(\mathbf{u})) = h(v)$ iff $h(\delta) \in \text{delta}(\mathcal{M}, S)$.

- 1st equivalence: Def. $\text{delta}(\mathcal{M}, R)$, above.
 2nd equivalence: Def. semantics of \mathcal{M} , 4.4.
 3rd equivalence: Term Homomorphy Lemma, A4.
 4th equivalence: Def. semantics of \mathcal{M} , 4.4.
 5th equivalence: Def. $\text{delta}(\mathcal{M}, S)$. ■

The *delta-Coincidence-Lemma* relates updates to the characteristic terms: Two states generate the same updates if they interpret the characteristic term equally:

Lemma 4 (delta-Coincidence-Lemma). *Let \mathcal{M} be a characteristic program over Σ and let $R, S \in \text{states}$ with $t_R = t_S$ for all $t \in T$. Then $\text{delta}(\mathcal{M}, R) = \text{delta}(\mathcal{M}, S)$.*

Proof. in analogy to the proof of the delta-Isomorphy-Lemma:

Let $\delta = (f, \mathbf{u}, v)$, let $R' =_{\text{def}} \text{sem}_{\mathcal{M}}(R)$ and $S' =_{\text{def}} \text{sem}_{\mathcal{M}}(S)$. Then $\delta \in \text{delta}(\mathcal{M}, R)$ iff δ is a Σ -update with $f_R(\mathbf{u}) \neq f_{R'}(\mathbf{u}) = v$
 iff there exists an assignment formed $f(t_1, \dots, t_n) := t_0$ in \mathcal{M} with $(t_{1R}, \dots, t_{nR}) = \mathbf{u}$ and $t_{0R} = v$ and $f_R(\mathbf{u}) \neq f_{R'}(\mathbf{u})$
 iff there exists an assignment formed $f(t_1, \dots, t_n) := t_0$ in \mathcal{M} with $(t_{1S}, \dots, t_{nS}) = \mathbf{u}$ and $t_{0S} = v$ and $f_S(\mathbf{u}) \neq f_{S'}(\mathbf{u})$
 iff δ is a Σ -update with $f_S(\mathbf{u}) \neq f_{S'}(\mathbf{u}) = v$ iff $\delta \in \text{delta}(\mathcal{M}, S)$.

The 3rd equivalence holds because \mathcal{M} is characteristic, hence $t_0, \dots, t_n \in T$. $t_{iR} = t_{iS}$ then follows from the Lemma's assumption. All other equivalences follow as in the proof of the above delta-Isomorphy-Lemma. ■

5.3 A program for the next state

Each state S of the algorithmic transition system \mathcal{A} gives rise to a characteristic program \mathbf{S} . This program consists of consistent assignments only. It properly computes $\tau(S)$.

We start with the *characteristic elements Lemma*: The values occurring in updates of the transition system \mathcal{A} can be represented by characteristic terms:

Lemma 5 (characteristic elements Lemma). *Let S be a Σ -algebra with universe U , and let $(f, (u_1, \dots, u_n), u_0) \in \Delta(\tau, S)$. Then $u_0, \dots, u_n \in T_S$.*

Proof. Assume by contradiction there exists an index k with $u_k \notin T_S$. Let $v \notin U$, and replace in S each occurrence of u_k by v . This yields a Σ -algebra, R , with domain $U' = U \setminus \{u_k\} \cup \{v\}$. Obviously, $R \simeq S$,

hence $R \in \text{state}$.

Firstly, $u_k \notin T_S$, implies $t_R = t_S$ for all $t \in T$. This implies $\tau(R) = \tau(S)$, because τ is bounded exploration with the characteristic set T . Hence $(f, (u_1, \dots, u_n), u_0)$ is an update of R , too.

Secondly, $u_k \notin U'$ implies u_k is not in the domain of $\tau(R)$, because τ is domain preserving. Hence $(f, (u_1, \dots, u_n), u_0)$ is no update of R . Contradiction! ■

Here comes the definition of the program \mathbf{S} . This definition is sensible due to the above characteristic elements lemma:

Definition 21. *Let S be a state.*

1. Let $\delta = (f, (u_1, \dots, u_n), u_0) \in \Delta(\tau, S)$. For $i = 0, \dots, n$ let $t_i \in T$ with $t_{iS} = u_i$ (t_i exists according to the above characteristic elements Lemma).

Then $f(t_1, \dots, t_n) := t_0$ is a characteristic assignment of δ .

2. Let $\Delta(\tau, S) = \{\delta_1, \dots, \delta_m\}$ and let r_i be a characteristic assignment of δ_i ($i = 1, \dots, m$). Then $\mathbf{S} = \mathbf{par} \ r_1, \dots, r_m \ \mathbf{endpar}$ is a characteristic S -program.

The following \mathbf{S} -consistence Lemma shows that the assignments in \mathbf{S} are consistent:

Lemma 6 (S-consistence Lemma). *Let $S \in \text{states}$ and let $\mathbf{S} = \mathbf{par} \ r_1, \dots, r_m \ \mathbf{endpar}$ be a characteristic S -program. Then $\{r_1, \dots, r_m\}$ is consistent in S .*

Proof. Assume by contradiction two indices i, j with $r_i = f(t_1, \dots, t_n) := t_0$ and $r_j = f(t_1, \dots, t_n) := t'_0$ with $t_{0S} \neq t'_{0S}$. Then $(f, (t_{1S}, \dots, t_{nS}), t_{0S})$ and $(f, (t_{1S}, \dots, t_{nS}), t'_{0S})$ are both in $\Delta(\tau, S)$. Then $f_{\tau(S)} = t_{0S}$ and $f_{\tau(S)} = t'_{0S}$, hence $f_{\tau(S)}$ was no function. ■

Applied to S , the program \mathbf{S} yields $\tau(S)$, due to the following $\text{sem}_{\mathbf{S}}(S)$ -Lemma:

Lemma 7 ($\text{sem}_{\mathbf{S}}(S)$ -Lemma). *Let $S \in \text{states}$, let $S' =_{\text{def}} \text{sem}_{\mathbf{S}}(S)$, let f be an n -ary symbol, and let $\mathbf{u} \in U^n$. Then $f_{S'}(\mathbf{u}) = f_{\tau(S)}(\mathbf{u})$.*

Proof. With $\mathbf{S} = \mathbf{par} \ r_1, \dots, r_m \ \mathbf{endpar}$, let $Z =_{\text{def}} \{r_1, \dots, r_m\}$.
1st case: $(f, \mathbf{u}, v) \in \Delta(\tau, S)$, for some $v \in U$. Then there exists an index i with $r_i = f(t_1, \dots, t_n) := t_0$, $\mathbf{u} = (t_{1S}, \dots, t_{nS})$ and $v = t_{0S}$. Then $f_{S'}(\mathbf{u}) = f_{\text{sem}_{\mathbf{S}}(S)}(\mathbf{u}) = f_{\text{sem}_Z(S)}(\mathbf{u}) = f^{Z, S}(\mathbf{u}) = t_{0S} = v = f_{\tau(S)}(\mathbf{u})$.

1st equation: Definition of S' .

2nd equation: semantics of ASM-programs, 4.4, and the above \mathbf{S} -consistence Lemma.

3rd equation: semantics of assignment sets, 4.3.

4th equation: semantics of an assignment, 4.3.

5th equation: construction of r_i .

6th equation: $(f, \mathbf{u}, v) \in \Delta(\tau, S)$.

2nd case: There exists no $v \in U$ with $(f, \mathbf{u}, v) \in \Delta(\tau, S)$. Then $f_{S'}(\mathbf{u}) = f_{sem_{\mathbf{S}}(S)}(\mathbf{u}) = f_{sem_Z(S)}(\mathbf{u}) = f^{Z,S}(\mathbf{u}) = f_S(\mathbf{u}) = f_{\tau(S)}(\mathbf{u})$.

The first four equations are as in the 1st case, the 5th equation holds by construction of $\Delta(\tau, S)$. ■

5.4 Updates of characteristic programs

Given a state S of \mathcal{A} , we construct ASM programs \mathcal{M} such that $delta(\mathcal{M}, S) = \Delta(\tau, S)$. We start with characteristic programs $\mathcal{M} = \mathbf{S}$, and generalize to $\mathcal{M} = \mathbf{R}$, where R is a state that cannot be distinguished from S by characteristic terms.

We start with the *delta*(\mathbf{S}, S)-*Lemma*: Each characteristic program \mathbf{S} , as defined in 5.3, yields a set $delta(\mathbf{S}, S)$ of updates that is equal to $\Delta(\tau, S)$:

Lemma 8 (*delta*(\mathbf{S}, S)-*Lemma*). *Let $S \in \text{states}$. Then $delta(\mathbf{S}, S) = \Delta(\tau, S)$.*

Proof. Let $S' =_{def} sem_{\mathbf{S}}(S)$. Then $(f, \mathbf{u}, v) \in delta(\mathbf{S}, S)$ iff (f, \mathbf{u}, v) is a Σ -update with $f_S(\mathbf{u}) \neq f_{S'}(\mathbf{u}) = v$ iff $f_S(\mathbf{u}) \neq f_{\tau(S)}(\mathbf{u}) = v$ iff $(f, \mathbf{u}, v) \in \Delta(\tau, S)$.

1st equivalence: Def. *delta*(\mathbf{S}, S), 5.2.

2nd equivalence: *sem* $_{\mathbf{S}}$ -Lemma, 5.3.

3rd equivalence: Def. $\Delta(\tau, S)$, 3.7. ■

The following *delta*- Δ -*Coincidence-Lemma* generalizes the *delta*(\mathbf{S}, S)-*Lemma*: The program \mathbf{S} may be replaced by any program \mathbf{R} , provided R and S interpret the characteristic terms equally:

Lemma 9 (*delta*- Δ -*Coincidence-Lemma*). *Let $R, S \in \text{states}$. For each $t \in T$ let $t_R = t_S$. Then $delta(\mathbf{R}, S) = \Delta(\tau, S)$.*

Proof. $delta(\mathbf{R}, S) = delta(\mathbf{R}, R) = \Delta(\tau, R) = \Delta(\tau, S)$.

1st equation: *delta*-*Coincidence Lemma*, 5.2.

2nd equation: *delta*(\mathbf{S}, S)-*Lemma*, 5.4.

3rd equation: Def. characteristic set of terms, 3.10. ■

The *QRS-lemma* shows that the equation $\text{delta}(\mathbf{R}, S) = \text{delta}(\tau, S)$ is retained by isomorphic state:

Lemma 10 (QRS-Lemma). *Let $Q, R, S \in \text{states}$, let $\text{delta}(\mathbf{R}, Q) = \Delta(\tau, Q)$, let $Q \simeq S$. Then $\text{delta}(\mathbf{R}, S) = \Delta(\tau, S)$.*

Proof. Let $h : S \rightarrow Q$ be an isomorphism. Then $\delta \in \text{delta}(\mathbf{R}, S)$ iff $h(\delta) \in \text{delta}(\mathbf{R}, Q)$ iff $h(\delta) \in \Delta(\tau, Q)$ iff $h^{-1}(h(\delta)) \in \Delta(\tau, S)$ iff $\delta \in \Delta(\tau, S)$.

1st equivalence: delta-Isomorphy Lemma, 5.2.

2nd equivalence: the Lemma's assumption.

3rd equivalence: Δ -Isomorphism Lemma, 3.9.

4th equivalence: h is an isomorphism. ■

5.5 An ASM for equivalent states

Given a state S of \mathcal{A} , we construct ASM programs \mathcal{M} such that $\text{sem}_{\mathcal{M}}(S) = \tau(S)$. First we show that $\text{delta}(\mathcal{M}, S) = \Delta(\tau, S)$ is sufficient for this property. Then we specialize \mathcal{M} to \mathbf{R} , with states R that are equivalent, in a very sophisticated sense, to S .

First the Δ - τ -Lemma; a necessary requirement for $\text{sem}_{\mathcal{M}}(S) = \tau(S)$:

Lemma 11 (Δ - τ -Lemma). *Let \mathcal{M} be an ASM-program, let $S \in \text{states}$ with $\text{delta}(\mathcal{M}, S) = \Delta(\tau, S)$. Then $\text{sem}_{\mathcal{M}}(S) = \tau(S)$.*

Proof. Let $S' = \text{sem}_{\mathcal{M}}(S)$, let U be the universe of S , let f be an n -ary symbol in Σ and let $\mathbf{u} \in U^n$. We have to show: $f_{S'}(\mathbf{u}) = f_{\tau(S)}(\mathbf{u})$.
1st case: $f_S(\mathbf{u}) = f_{S'}(\mathbf{u})$. Then there exists no $v \in U$ with $(f, \mathbf{u}, v) \in \text{delta}(\mathcal{M}, S)$. Then there exists no $v \in U$ with $(f, \mathbf{u}, v) \in \Delta(\tau, S)$. Then $f_{\tau(S)}(\mathbf{u}) = f_{S'}(\mathbf{u})$.

1st implication: Def. of $\text{delta}(\mathcal{M}, S)$, 5.2.

2nd implication: $\text{delta}(\mathbf{S}, S)$ -Lemma, 5.4.

3rd implication: Def. $\Delta(\tau, S)$, 3.7.

2nd case: $f_S(\mathbf{u}) \neq f_{S'}(\mathbf{u}) = v$. Then $(f, \mathbf{u}, v) \in \text{delta}(\mathcal{M}, S)$. Then $(f, \mathbf{u}, v) \in \Delta(\tau, S)$. Then $f_{\tau(S)}(\mathbf{u}) = v$.

Justification of all three implications follows the arguments of the first case. ■

Every state S defines an equivalence \sim on the characteristic term set T . These equivalences, in turn, define an equivalence \approx on states:

Definition 22. *1. Each $S \in \text{states}$ defines an equivalence, \sim_S , on T , defined for $t, t' \in T$ by $t \sim_S t'$ iff $t_S = t'_S$.*

2. The equivalence \approx on states is defined for $R, S \in \text{states}$ by $R \approx S$ iff $\sim_R = \sim_S$.

As a first, obvious observation, the *equivalence Lemma* on equivalent states: Equivalent states cannot discriminate characteristic terms. And isomorphic states are equivalent:

Lemma 12 (equivalence Lemma). *Let $R, S \in \text{states}$.*

1. *If $R \approx S$ then for all $t, t' \in T$: $t_R = t'_R$ iff $t_S = t'_S$.*
2. *If $R \simeq S$ then $R \approx S$.*

The *semantic lemma* characterizes $\tau(S)$ of equivalent states:

Lemma 13 (semantic lemma). *Let $R, S \in \text{states}$, $R \approx S$. Then $\text{sem}_{\mathbf{R}}(S) = \tau(S)$.*

Proof. With the above Δ - τ -Lemma suffices to show: $\text{delta}(\mathbf{R}, S) = \Delta(\tau, S)$.

With the *QRS-Lemma* in 5.4 it suffices to construct a state $Q \simeq S$ such that $\text{delta}(\mathbf{R}, Q) = \Delta(\tau, Q)$. We distinguish two cases for the construction of Q :

1st case: The universes U_R and U_S of R and S are disjoint. Then for each characteristic term $t \in T$, replace in U_S the value t_S by t_R . This is well defined as to part 1 of the above equivalence Lemma. Let Q denote the new state. Obviously, $t_Q = t_R$ for each $t \in T$. Then the *delta- Δ -Coincidence Lemma* in 5.4 implies $\text{delta}(\mathbf{R}, Q) = \Delta(\tau, S)$.

2nd case: The universes of U_R and U_S are not disjoint. Then replace each element of $U_R \cap U_S$ by some new element in U_S . The resulting state, Q , is isomorphic to S and has a universe disjoint to U_R . The second part of the above equivalence lemma implies $Q \approx S$. This reduces the 2nd case to the 1st case. ■

5.6 Guards and the final proof

Each state S defines a boolean expression $\beta[S]$ that is true exactly in all states R equivalent to S . We first define the expression $\beta[S]$:

Definition 23. *Let S be a Σ -algebra.*

1. *For two terms $t, t' \in T_\Sigma$, the boolean term $t = t'$ is an S-guard in case $t_S = t'_S$. $t \neq t'$ is an S-guard in case $t_S \neq t'_S$.*
2. *Let $\beta[S]$ denote the conjunction of all S-guards with terms $t, t' \in T$.*

The *Guard-Lemma* shows that the conjunction of all S-guards characterizes the equivalence \approx :

Lemma 14 (Guard-Lemma). *Let $R, S \in \text{states}$. Then $\beta[S]_R = \text{true}$ iff $R \approx S$.*

Proof follows immediately from the first part of the equivalence Lemma. ■

Lemma 15 (S^* -Lemma). *Let S be Σ -algebra, and let $S^* =_{\text{def}}$ if $\beta[S]$ then **S**. Then holds for each $R \in \text{states}$: $\text{sem}_{S^*}(R) = \text{sem}_{\mathbf{S}}(R)$ if $R \approx S$, and $\text{sem}_{S^*}(R) = R$, otherwise.*

Proof follows immediately from the semantic of guarded Σ -assignments, as in 4.3. ■

We are now prepared to show the Theorem. Based on the general assumptions as given in 5.1, it reads:

Theorem. *There exists an ASM, \mathcal{M} , such that for each $S \in \text{states}$ holds: $\text{sem}_{\mathcal{M}}(S) = \tau(S)$.*

Proof. \approx has finitely many equivalence classes. Let S_1, \dots, S_n be representants of these classes. Let $\mathcal{M} = \mathbf{par} S_1^*, \dots, S_n^* \mathbf{endpar}$. Each state $s \in S$ has an index i with $S \approx S_i$. Then $\text{sem}_{\mathcal{M}}(S) = \text{sem}_{S_i^*}(S) = \text{sem}_{S_i}(S) = \tau(S)$.

1st equation: guard lemma in 5.6, and semantics of \mathcal{M} in 4.4.

2nd equation: S^* -Lemma in 5.6.

3rd equation: semantic-Lemma in 5.5. ■

6 Further aspects of ASM

We discussed the most elementary version of ASM in a particular syntactical representation. There are many variants and extensions of the formalism introduced here. Section 6.1 gives some details. Section 6.3 expands on the role of ASM as a model of computation. Section 6.4 glances ASM as a specification technique.

6.1 Variants of ASM

The term “Abstract-State Machine” refers to the fundamental idea that a state is a mathematical structure, not necessarily fixed by a symbolic, effective representation.

The next-state function τ must meet a couple of requirements discussed in Chapter 3. There is even a syntactical representation for

next-state functions, given in Chapter 4.

But the syntax of ASM is fragile: many other versions are likewise reasonable. For example homogeneous algebras may be replaced by heterogeneous algebras.

There are substantial extensions, too. The deterministic next-state function τ may be replaced by a non-deterministic next-state relation. Reactive behaviour may be modelled by steps $S_i S_{i+1}$ which are not conducted by the corresponding ASM program, but by the outside world (a similar idea has also been suggested in the FOCUS formalism [BS01]). Bounded exploration may be skipped, using \forall -quantified formulas in ASM programs.

Gurevich suggested these variants and extensions in his *Lipari Guide* [Gur95]. A distributed version of ASM has likewise been advocated, where a single run consists of a partial order of actions, in the line of Petri, Lamport and Pratt. This version of “Multi Agent ASM” also captures recursion.

6.2 Special Classes of ASM

Let $\mathcal{A} = (\text{states}, \text{init}, \tau)$ be an algorithmic transition system. According to Definition 13, *any* Σ -algebra can serve as a state. One may expect states to be confined to “reasonable” algebras. Here three such proposals:

Call an algebra *finitary* iff its carrier U is finite. As a variant, U may be infinite, but each n -ary function f may have only finitely many arguments $\mathbf{u} \in U^n$ with $f(\mathbf{u}) \neq \text{undef}$. The class of finitary algebras is closed under isomorphism. If $S \in \text{states}$ is finitary, then $\tau(S)$ is finitary, too. Hence all reachable states are finitary if the initial states are finitary.

Let Σ be the signature of \mathcal{A} . A state $S \in \text{states}$ is *generated* if to each element u of its carrier there exists a term $t \in T_\Sigma$ with $t_S = u$. The class of generated algebras is closed under isomorphism. But for a generated state S , the successor $\tau(S)$ is not necessarily generated. Finally, call an algebra S *computable* iff its carrier is countable and all its functions are computable (in the classical framework). The class of computable algebras is closed under isomorphism. If S is computable, so is $\tau(S)$.

The above classes of algebras give rise to the definition of *finitary*, *generated* and *computable* ASM. These versions of ASM are implementable (the latter ones up to finitary limitations of hardware). But the ASM formalism does not support any particularly interesting

properties, e.g. analysis techniques, for those classes.

6.3 ASM in the context of the Theory of computing

ASM have been introduced as “a computation model that is more powerful and more universal than standard computation models for theoretical computer science” [Gur99]. They are “improving on Turing’s Thesis” [Gur88] with a “new thesis” [Gur85], contrasting the “old” Church-Turing Thesis.

The central argument: An algorithmic idea is not bound to a syntactical representation. Chapter 2.1 gave an example: Elementary items are points, circles, and lines. Elementary operations construct the halfway point of two points, the circle from its center and a point on its surface, the intersection of two circles, and the line through two points. The idea to employ algebras as states has occasionally been suggested, e.g. in [FJ92]. An early source of similar ideas is [Pai80]. But ASM is the only formalism to ask for a more general version of *steps* on top of those states.

The representation free (though mathematically unique) notion of states is combined with a likewise mathematically unique characterization of “algorithmic” next-state functions. We characterized them by help of five requirements; re-shuffling three requirements of [Gur00]. The Theorem of Chapter 5 provides a syntactical version for each next-state function. A corresponding syntactical version of states cannot exist: not all reachable states are necessarily syntactically representable, as discussed in the introduction.

The first of the five requirements, i.e. “each state is an algebra” can be reversed: “Each algebra can serve as a state”. The third requirement implies: All states of an algorithm are Σ -algebras. This provides the key for the central idea of ASM: Describe the algorithm syntactically, as a program, by Σ -terms. The semantics of an ASM program is a mathematical structure, not bothering with symbolic representation. The relationship between Σ -terms and Σ -algebras, including the expressive power of finite sets of terms, has extensively been studied in Logics and Model Theory. It has been adapted to the needs of computing, under the headline of “Algebraic Specifications”, since the mid 1970ies.

The step from algorithms to programs is then not trivial: An algorithm’s states must be implemented. It is here, but not before, where symbolically represented Data Structures enter the scene.

The Abstract-State Machine formalism is not the first one to ques-

tion the role of the Church-Turing Thesis in the foundations of computer science and to suggest computation not necessarily based on the fixed set of objects and operations. The first ones include C.A Petri, who in his seminal thesis [Pet62] showed that asynchronous models describe “The Implementable” more faithfully than synchronous models. Friedman in [Fri71] suggests a theory of computation over relational structures, where the operations and relations of the structure are taken as basic computational operations. The huge field of program schemata (cf. [LPP70]) considers computation on a symbolic level, over the terms of a signature. [Gan80] suggests four principles which every computation mechanism is required to meet.

The above mentioned papers, as well as many others, are essentially bound to the manipulation of symbols. ASM provides a *semantical* framework for the notion of “algorithm”. It must be syntactically underpinned, of course, when it comes to the representation and tool-based manipulation of Abstract-State Machines.

6.4 ASM as a specification language

Formal specification methods usually suggest a set of elementary “constructive” data items. For example the Z method is based on the set of integers. Complex data structures are inductively constructed from simpler ones. This approach guarantees that every specification is implementable.

ASM starts out from *any* objects and operations.

[Bo99] emphasizes “The ability to simulate arbitrary algorithms on their natural level of abstraction, without implementing them, makes ASM appropriate for high-level system design and analysis”.

In fact, the liberal requirement for states and steps adapt easily and perfectly to any kind of algorithm.

Assuming a proper intuition of what an “algorithmic idea” might be, the ASM formalism is intended to capture each algorithmic idea “faithfully and unambiguously”. This requires

- to represent each elementary item of the algorithmic idea as an elementary formal object,
- to represent each elementary operation of the algorithmic idea as an elementary formal operation,
- to map states and steps of the algorithmic idea *bijectively* to states and steps of the corresponding formal model.

It comes without surprise that ASM have been particularly successful in describing the semantics of programming languages: Semantics

deals with a rich variety of involved mathematical structures that don't require a syntactical representation. This, exactly, is what ASM provides.

The idea to employ mathematical structures as components of states has been advocated in [CH78] already: Data spaces such as stacks, trees and all forms of data structures from Algol, Lisp and Fortran, together with corresponding operations, define *virtual machines*. ASM generalizes this to *any* kind of data spaces, via algebras; [CH78] stick to structures that are implementable in a canonical way.

[Gan81] suggests to define the state of a program \mathcal{P} as Σ -algebras, exactly as done in ASM. Ganzinger formally defines the semantics of \mathcal{P} to be a free construct, i.e. a mapping from a set of Σ -algebras to a set of Σ -algebras. [Gan83] expands on this idea; it may be likewise applied to ASM.

The “state-as-algebra” paradigm [Gan83] has been a basis for various lines of research. Categorical constructs, as employed in [Gan81] already, are likewise used in [Zuc96]. In [GZ00], the authors show that the “state-as-algebra” paradigm is useful to describe the semantics of specification languages such as VDM, Z and B. They advocate the combination of algebraic and imperative specifications, of which ASM is an example. A further example is the “Algebraic Specification with Implicit state” approach of [DG93].

Modern specification techniques such as Z, TLA and FOCUS [Lam00,BS01] follow logic based guidelines, such as “a specification is a (huge) logical expression”, “implementation (refinement) is implication”, or “composition is conjunction”. The ASM formalism has not been designed along these guidelines, nor does it contradict them. It might be useful to integrate those guidelines into ASM.

Conclusion

The idea that *every* algebra may serve as a system state is a fundamental feature of ASM. It is accompanied by a notion of “algorithmic” next-state functions. Without any reasonable doubt, the four requirements of chapter 3 are necessary for algorithmic next-state functions. That they are sufficient, is a very beautiful result. It witnesses that the approach may in fact be a reasonable starting point for a new theory of algorithms.

The liberal notions of states and next-state functions of ASM allows to incorporate any kind of analysis methods that have been developed for various specification techniques. But no *specific* analysis methods have been designed for ASM so far.

The extension of ASM, outlined in Chapter 6.1, may be accompanied by restrictions with sufficient expressive power for particular classes of algorithms, and equipped with analysis methods that would exploit particular properties of those algorithms.

All this and much more, including a number of tools, can be found in the survey paper [Bo02], and the ASM web page <http://www.eecs.umich.edu/gasm>.

Acknowledgements

I owe much to Egon Börger. He introduced me to the field of ASM. Yuri Gurevich answered my many (occasionally not too intelligent) questions promptly and in detail – thus influencing much of this paper. Peter Päppinghaus and Robert Stärk pointed me at some bugs in a preliminary version of this material. The ASM meetings in Monte Verita and Dagstuhl boosted my understanding of the field.

References

- [Bo99] E. Börger. High Level System Design and Analysis using Abstract State Machines. In *Current Trends in Applied Formal Methods*, volume 1464 of *LNCS*, pages 1–43, 1999.
- [Bo02] E. Börger. The Origin of the ASM Method for High Level System Design and Analysis. *Journal of Universal Computer Science*, Vol.8 No.1:2–74, 2002.
- [BS01] M. Broy and K. Stølen. *Specification and Development of Interactive Systems*. Springer-Verlag, 2001.
- [CH78] A.B. Cremers and T.N. Hibbard. Formal Modeling of Virtual Machines. *IEE on Software Engineering*, SE-4 No 5:426–436, September 1978.
- [DG93] P. Dauchy and M.-C. Gaudel. Implicit state in algebraic specifications. In *ISCORE'93*, volume No 01/93 of *Informatik-Berichte*. Universität Hannover, 1993.
- [FJ92] L.M.G. Feijs and H.B.M. Jonkers. Formal Specification and Design. *Cambridge Tracts in Theoretical Computer Science*, 35:188ff, 1992.
- [Fri71] H. Friedmann. Algorithmic procedurs, generalized Turing algorithms and elementary recursion theory. In *Logic Colloquium '69*. Gandy and Yates(eds), North-Holland Publ. Co, Amsterdam, 1971.
- [Gan80] R.O. Gandy. Church's Thesis and principles for mechanisms. In *The Kleene Symposium*. H.Barwise et al(eds), North-Holland Publ. Co, Amsterdam, 1980.
- [Gan81] H. Ganzinger. Programs as Transformations of Algebraic Theories. In *11. GI-Jahrestagung*, Informatikfachberichte 50, pages 32–41. Springer-Verlag, 1981.
- [Gan83] H. Ganzinger. Denotational Semantics for Languages with Modules. In D. Bjørner, editor, *Formal Description of Programming Concepts - II*, pages 3–20. North-Holland, 1983.

- [Gau83] M.-C. Gaudel. Correctness Proof of Programming Language Translations. In *IFIP-TC2 Working Conference on Formal Descriptions of Programming Concepts, Garmisch Partenkirchen, June 83*, pages pp 25–43. North-Holland, 1983.
- [Gur85] Y. Gurevich. A new thesis. *American Mathematical Society Abstracts*, page 317, August 1985.
- [Gur88] Y. Gurevich. Logic and the Challenge of Computer Science. In *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.
- [Gur95] Y. Gurevich. Evolving Algebra 1993: Lipari Guide. In *Specification and Validation Methods, E.Börger(ed)*, pages 9–36. Oxford University Press, 1995.
- [Gur99] Y. Gurevich. *The Sequential ASM Thesis Bulletin of the EATCS*, Number 67:93–124, 1999.
- [Gur00] Y. Gurevich. Sequential Abstract-State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, Vol.1 No.1:77–111, July 2000.
- [GZ00] M.-C. Gaudel and A. Zamulin. Imperative Algebraic Specifications. In *conference, PSI'99, Novosibirsk, June 1999*, volume 1755 of *LNCS*, pages 17–39. Springer-Verlag, 2000.
- [Lam00] L. Lamport. Specifying Concurrent Systems with TLA⁺. unpublished manuscript, <http://www.research.digital.com/SRC/tla>, 2000.
- [LPP70] D.C. Luckham, D.M.R. Park, and M. Paterson. Formalized computer programs. *J. Comput. System Sci.*, 4:220–249, 1970.
- [Pai80] C. Pair. Types abstrait et Sémantique Algébrique des Langues de Programmation. *Centre de Recherche en Informatique de Nancy*, TR 80-R-011:1–46, Feb/July 1980.
- [Pet62] C.A. Petri. Kommunikation mit Automaten. *Schriften des Institutes für Instrumentelle Mathematik Bonn*, 1962.
- [Zuc96] E. Zucca. From Static to Dynamic Abstract Data-Types. In A.Szalas W.Penczek, editor, *MFCS 96*, volume 1113 of *LNCS*, pages pp 579–590, 1996.

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.

A.1 Algebras and Isomorphisms

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

Definition 1. *Let U be a set.*

1. Let $f : \underbrace{U \times \dots \times U}_{n\text{-fold}} \rightarrow U$ be a function. Then n is its arity.
2. The case of arity $n = 0$ yields a constant, i.e. an element $f() \in U$, written f .
3. Let $n_1, \dots, n_k \in \mathbb{N}$. For $i = 1, \dots, k$ let f_i be a function over U with arity n_i . Then $S = (U, f_1, \dots, f_k)$ is an algebra. U is its carrier; (n_1, \dots, n_k) is its type.

Isomorphic algebras correspond bijectively:

Definition 2. Let $R = (U, f_1, \dots, f_k)$ and $S = (V, g_1, \dots, g_k)$ be two algebras, both with type (n_1, \dots, n_k) .

1. Let $h : U \rightarrow V$ such that for $i = 1, \dots, k$ and $u_1, \dots, u_{n_i} \in U$ holds: $h(f_i(u_1, \dots, u_{n_i})) = g_i(h(u_1), \dots, h(u_{n_i}))$. Then h is a homomorphism from R to S , written $h : R \rightarrow S$.
2. Let $h : R \rightarrow S$ be a bijective homomorphism. Then h is an isomorphism.
3. R and S are isomorphic, written $R \simeq S$, if there exists an isomorphism $h : R \rightarrow S$.

A.2 Signature and Terms

A signature provides names for the functions of algebras:

Definition 3. Let f_1, \dots, f_k be symbols and let $n_1, \dots, n_k \in \mathbb{N}$.

1. $\Sigma = (f_1, \dots, f_k, n_1, \dots, n_k)$ is a signature. For $1 \leq i \leq k$, the number n_i is the arity of f_i ; (n_1, \dots, n_k) is the arity of Σ .
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 4. Let Σ be a signature. The set T_Σ of Σ -ground terms is inductively defined as follows:

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

A.3 Σ -Algebras

Each signature characterizes a set of algebras:

Definition 5. Let $\Sigma = (f_1, \dots, f_k, n_1, \dots, n_k)$ be a signature. Each algebra $S = (U, g_1, \dots, g_k)$ with arity (n_1, \dots, 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.

Each ground term denotes in each interpretation a unique element:

Definition 6. Let $\Sigma = (f_1, \dots, f_k, n_1, \dots, n_k)$ be a signature and let S be a Σ -algebra with carrier U . Each ground term $t \in T_\Sigma$ denotes an element $t_s \in U$, inductively defined by

$$\begin{aligned} t = f_{iS} & \text{ iff } t = f_i \text{ and } n_i = 0 \\ t = f_{iS}(t_{1S}, \dots, t_{kS}) & \text{ iff } t = f_i(t_1, \dots, t_k). \end{aligned}$$

While a signature Σ defines a set $\text{Alg}(\Sigma)$ of algebras, an algebra has a unique signature (up to renaming):

Lemma 1. Let S be an algebra. Up to renaming, there exists a unique signature Σ such that S is a Σ -algebra. Σ is called the signature of S .

A.4 Two basic Lemmata

First we show the *Term Homomorphism Lemma*: A homomorphism extends uniquely to ground terms.

Lemma 2. Let Σ be a signature, let R and S be two Σ -algebras and let $h : R \rightarrow S$ be a homomorphism. Then holds for all terms $t \in T_\Sigma$: $h(t_R) = t_S$.

This is easily proven by induction on the structure of T_Σ .

Terms cannot discriminate isomorphic algebras, as the following *Indistinguishability Lemma* shows:

Lemma 3. Let Σ be a signature, let R and S be two isomorphic Σ -algebras. Then for all $t, u \in T_\Sigma$ holds: $t_R = u_R$ iff $t_S = u_S$.

This is easily proven by help of the above Term Homomorphism Lemma.