Joint NCC & IIS Bull., Comp. Science, 7 (1997), 95–121

Typed Gurevich machines revisited*

A.V. Zamulin

An approach to combining type-structured algebraic specifications with Gurevich Machines (evolving algebras) is proposed. A type-structured algebraic specification, in its simplest form, consists of data type specifications and independent function (detached operation) specifications. Concrete and generic specification components (data types and functions) are distinguished in a more developed case. A type-structured algebraic specification is augmented by a number of transition rules of a conventional Gurevich Machine indicating in which way an algebra of a given signature evolves to another algebra of the same signature. A function update is a primitive algebra transformation. Two classes of functions are distinguished, static functions which do not change when an algebra evolves, and dynamic functions which do change when an algebra evolves. Data type operations are static. The semantics of data types and static functions is given by axioms, the semantics of dynamic functions is given by transition rules.

Keywords: evolving algebras, Gurevich Machines, Abstract State Machines, algebraic specifications, implicit state.

1. Introduction

Evolving algebras proposed by Gurevich [1] have been intensively used for formal definition of various algorithms and programming language semantics [2–9]. The success of the approach can be attributed to two reasons: (1) sound mathematical background and (2) imperative specification style. In contrast to conventional algebraic specification languages providing the specification technique resembling functional programming, evolving algebras provide several transition rules resembling imperative programming statements. As a result, a specification looks like an imperative program, it is easier to understand, and it is executable. This imperative nature of evolving algebras has led to the introduction of a new term for them, Gurevich Machines (the terms Abstract State Machines and Gurevich Abstract State Machines are also in use).

Unfortunately, Gurevich Machines in their present form are quite low-level. They are based on the notion of a universal algebraic structure consisting of a set, a number of functions, and a number of relations. There

*Partially supported by the Russian Foundation for Basic Research under Grant 95-01-00878.

are a number of transition rules indicating in which way an algebra can be converted into another algebra of the same signature. Normally, this is done by a slight change of a function. For this reason, functions can be either *static* or *dynamic*. A static function never changes, a change of a dynamic function produces a new algebra. Another means of algebra modification is changing the number of elements in the underlying set (importing new elements or discarding existing ones).

The underlying set is called a *superuniverse* and can be subdivided into *universes* by means of unary relations. A universe serves to model a data type. When writing a specification, one can write the signature of any function operating with values of one or more universes. One cannot, however, define formally the semantics of a static function (the behavior of a dynamic function is defined by transition rules) or the set of values of a particular universe. It is assumed that the behavior of all static functions is either well-known or defined by some external tools; in the majority of cases, the same refers to universes (one can make sure of this, looking at the definition of C [6] where almost all static functions and universes are defined in plain words). As a result, one cannot construct arbitrary data types and functions with a well-defined semantics and either one has to use a small number of well-known data types like Boolean, Integer, etc. or one has to define informally needed data types and functions.

The purpose of this work is to propose a specification mechanism incorporating the advantages of both many-sorted algebraic specifications and Gurevich Machines. It is assumed that universes should be replaced with data types for which the semantics is formally defined by means of algebraic equations. The mechanism should provide means for defining both concrete data types and type constructors (generic, or parameterized data types). Some popular data types and type constructors should be built-in. A facility for defining independent static functions (i.e., functions not attributed to particular data types) should also be provided.

The mechanism of data type specifications proposed in this paper is a refined version of that of the specification language Ruslan [10, 11] (one-level specification versus multi-level specification) which better fits the evolving algebras environment. The main idea behind the choice of basic specification constructs has been to use the notions most familiar to the programming society. Such notions are mainly functions, data types, generic (parameterized) functions and data types, and type constructors (in contrast to such strange notions as "universe" [1], "concept" [12], "trait" [13], "sort" [14], etc.). The author believes that the closer the notions used in a specification language to the notions used in a programming language, the more chances exist that a programmer will ever pay attention to a specification. Another task has been avoidance of any other logic except the first-order many-sorted logic which is most familiar to the computer science specialists.

The set of transition rules proposed in the paper is mainly based on the set of basic rules of [1] with the addition of a tagcase constructor resembling a tagcase statement of some programming languages. The current proposal ignores algebra evolving by means of importing/discarding algebra elements as a strange mechanism to the programming community. The author believes that a proper choice of data types needed in a specification can solve this problem.

The previous paper [15] on the subject has presented an approach to combining type-structured algebraic specifications and Gurevich Machines. The present version of the approach takes into account some features proposed by the work of P. Dauchy and M. C. Gaudel on algebraic specifications with implicit state [16]. The approach is tested by formal definition of an Oberon compiler [17].

The rest of the paper is organized in the following way. Type-structured signatures and underlying algebras are defined in Section 2. The construction and interpretation of well-formed terms are described in Section 3. Type-structured specifications are introduced in Section 4. Specification classes serving to "make ad-hoc polymorphism less ad-hoc" [18] are presented in Section 5. Generic data types are defined in Section 6, and some built-in type constructors are described in Section 7. A mechanism for specifying generic functions is proposed in Section 8. A set of transition rules is defined in Section 9 and the general form of a specification is presented in Section 10. Two specification examples are given in Section 11. Some related work is discussed in Section 12, and some conclusions are drawn and directions of further work are outlined in Section 13.

2. Type-structured signatures and algebras

A type-structured signature is based on an ordinary many-sorted signature and is defined in the following way:

Definition. Let

$TYPE$ be a set of names called *type names*;

an *operation type* be either a type name or $T_1, \ldots, T_n \longrightarrow T$, where T, T_i, $i = 1, \ldots, n$, are type names; in an operation type $T_1, \ldots, T_n \longrightarrow T$, T_i is called a *domain unit* and T is called a *codomain unit*;

an *operation signature* be a pair $op : O$, where op is an operator (function symbol) and O is an operation type;

a *data type signature* be a set of operation signatures constructed as above by extending the set $TYPE$ with the symbol "@" meaning "myself" [20] and used at least once as domain and/or codomain unit in each operation signature,

then a *type-structured signature* Σ is a tuple $\langle TYPE, \Phi, \Delta, int^t \rangle$, where Φ is a set of data type signatures, Δ is a set of (detached) operation signatures, and int^t is a function mapping $TYPE$ into the set of data type signatures Φ. For any $T \in TYPE$ and $\phi \in \Phi$, we say that ϕ is marked with T if $int^t(T) = \phi$.

An operation signature *op* : @ is called a *constant signature*, *op* is called a *constant name*.

The conversion of a type-structured signature into an ordinary many-sorted signature is straightforward: in each data type signature the symbol "@" is replaced with the data type name marking the signature and the data type name is converted into a sort name.

A signature constructed in this way has the following benefits:

1. A data type signature groups all operations of a particular data type, and a type name is associated with it; this directly corresponds to the notion of a type interface in modern programming languages.
2. Operators can be overloaded in different signatures (and even in the same signature).
3. A data type name is not tightly coupled with a data type signature; the use of a special symbol meaning "myself" in a data type signature permits us to introduce easily specification classes (Section 5) and generic type specifications (Section 6);
4. Independent functions (detached operations) are allowed in addition to data types.

Notation. We generally construct the function int^t as a set of pairs ⟨*type-name, data-type-signature*⟩. A data type signature is introduced with a keyword **type** and enclosed in square brackets, a detached operation signature is introduced with a keyword **function** or **const**.

Example.

```
type Boolean =
 [true, false: @;
  "¬": @ ⟶ @;
  "|", "&": @, @ ⟶ @;
  "=", "<>": @, @ ⟶ @];

type Nat =
 [zero: @;
  succ: @ ⟶ @;
  "+", "-": @, @ ⟶ @;
  "=", "<>", "<", "<=", ">", ">=": @, @ ⟶ Boolean];
```

type SeqOfNat =
[empty: @;
append: Nat, @ ⟶ @;
head, length: @ ⟶ Nat;
tail: @ ⟶ @;
has: @, Nat ⟶ Boolean;
is_empty: @ ⟶ Boolean;
"=", "<>": @, @ ⟶ Boolean];

type RecNatSeqOfNat =
[create_rec: Nat, SeqOfNat ⟶ @;
p1: @ ⟶ Nat;
p2: @ ⟶ SeqOfNat];

function if: Boolean, Nat, Nat ⟶ Nat;

const count: Nat.

The data type *Boolean* with the above signature and conventional interpretation of its operations is built-in.

In all definitions which follow, it is assumed that the symbol "@" in a data type signature is replaced with the type name marking the signature.

An algebra A of a signature $\Sigma = \langle TYPE, \Phi, \Delta, int^t \rangle$ is built by assigning:

1) a set of elements to each type name from $TYPE$;
2) an element from the set $|A|_T$ to each operator *op* with the signature $op : T$, where $|A|_T$ is the set associated with the type name T in A;
3) a (partial) function $|A|_{T_1} \times \ldots \times |A|_{T_n} \longrightarrow |A|_T$ to an operator *op* with the signature $op : T_1, \ldots, T_n \longrightarrow T$, where $|A|_T, |A|_{T_i}, i = 1, \ldots, n$ are sets associated with the type names T, T_i.

The set of elements assigned in A to a type name $T \in TYPE$ is called the *set of values* of the (data) type T. The *carrier* of an algebra of a type-structured signature is the family of sets of data type values. We let $|A|$ denote the carrier of A, $|A|_T$ denote the set of values of the data type T in A, and op^A denote the element of the carrier of the algebra A assigned to *op*, if *op* is a constant, and the function assigned to *op* in A if it is a function symbol. An $a \in |A|$ is called an *element* of A. If Σ_T is the data type signature marked with the type name T, then the set assigned to T and the carrier elements and functions assigned to operators from Σ_T in a given algebra A are called *implementation* of the data type T in algebra A.

3. Term algebra

The term algebra $W_\Sigma(X)$ of a given type-structured signature Σ is constructed as follows. Let X_T be a set of variables indexed with a type name T. Then elements of this algebra, which are called *(data) terms*, are the following:

1) if x is a variable indexed with a type name T, then x is a term of type T;
2) if $c : T$ is a constant signature, then c is a term of type T;
3) if $op : T_1, \ldots, T_n \longrightarrow T$ is an operation signature and $e_1, \ldots, e_n$ are terms of types $T_1, \ldots, T_n$, respectively, then $op(e_1, \ldots, e_n)$ is a term of type T.

Operations of this algebra create larger terms from smaller ones.

Remark. Sometimes, we qualify an operator with a type name to avoid ambiguity, if there is a danger of it; thus, if t is a term of type T, then $T't$ is a term of type T.

A term without variables is called a *ground* term. We denote the set of all terms of type T by $|W_\Sigma(X)|_T$ and the set of ground terms of type T by $|W_\Sigma|_T$, the sets of all terms and ground terms are denoted by $|W_\Sigma(X)|$ and $|W_\Sigma|$, respectively.

For any Σ-algebra A and a set of typed variables X, given a substitution function $v : X \longrightarrow |W_\Sigma|$, an interpretation function

$$eval^A : |W_\Sigma(X)| \longrightarrow |A|$$

is defined as follows:

1) if $x \in X$, then $eval^A(x) = eval^A(v(x))$;
2) if c is a constant name, then $eval^A(c) = c^A$;
3) if $op : T_1, \ldots, T_n \longrightarrow T$ is an operation signature and $e_1, \ldots, e_n$ are terms of types $T_1, \ldots, T_n$, respectively, then
   - $eval^A(op(e_1, \ldots, e_n)) = op^A(eval^A(e_1), \ldots, eval^A(e_n))$, if $eval^A$ is defined for each term $e_i, i = 1, \ldots, n$, and op^A is defined for $(eval^A(e_1), \ldots, eval^A(e_n))$;
   - $eval^A(op(e_1, \ldots, e_n))$ is undefined, otherwise.

The interpretation function thus defined permits us to disregard whether algebras of a given signature are term-generated (reachable) or not.

4. Type-structured specifications

Definition. The notion of an *axiom* is defined in the following way:

1) if t_1 and t_2 are terms of type T, then $t_1 == t_2$ is an equation or axiom;
2) if A is an axiom, then **forall** $x_1 : T_1, \ldots, x_n : T_n.A$, where x_i is a variable and T_i is either a type name or the symbol "@", is an axiom;
3) if A is an axiom, then **exist** $x_1 : T_1, \ldots, x_n : T_n.A$, where x_i is a variable and T_i is either a type name or the symbol "@", is an axiom;
4) an axiom does not contain other variables except those bounded by quantifiers **forall** and **exist** (i.e., in an axiom $t_1 == t_2$, t_1 and t_2 are both ground terms);
5) an axiom $t_1 == t_2$ evaluates to *true* in a given algebra only if both t_1 and t_2 are defined and evaluate to the same element of the corresponding set for all valuations of any variable bounded by quantifier **forall** and at least for one valuation of any variable bounded by quantifier **exist**.

Remark. The symbol "@" can be used only in a data type specification.

Notation. We list all bounded variables at the beginning of the set of axioms.

Definition. Let Σ be a type-structured signature and E^ϕ and E^δ be sets of axioms associated with each $\phi \in \Phi$ and $\delta \in \Delta$, respectively, so that each axiom in E^ϕ has at least one operator from ϕ, and each axiom in E^δ uses the operator defined in δ. Then we get a *specification.* A pair $\langle \phi, E^\phi \rangle$ is called a *data type specification*, and a pair $\langle \delta, E^\delta \rangle$ is called a *detached operation specification.*

Thus, a type-structured specification is generally a set of data type specifications marked with type names and a set of detached operation specifications. Note that at the level of specification the function int^t binds type names to data type specifications. An algebra A is an algebra of a given specification if each axiom of this specification evaluates in A to true. This means that a data type implementation must satisfy all the axioms of the corresponding data type specification.

To express the definedness of a term t of a type-structured specification, we use a special semantic predicate D, such that $D(t)$ holds in an algebra A iff $eval^A(t)$ produces some object in A [19]. It is assumed that a detached constant can also be partial, i.e., it can be undefined in some algebras of a given signature[1].

[1]The idea of partial constants is suggested to the author by Felix Cornelius.

Notation. When constructing a specification, we build the function int^t as a set of pairs $\langle$*type-term, data-type-specification*$\rangle$. Data type signatures are enclosed in square brackets and sets of axioms are enclosed in curly brackets. The clause **dom**$f(x_1, \ldots, x_n) : p(x_1, \ldots, x_m)$, where f is an operator being specified, p is a predicate, and $(x_1, \ldots, x_m, \ldots, x_n)$ are universally quantified variables[2], such that $x_1, \ldots, x_m \subseteq x_1, \ldots, x_n$, defines the domain of a partial operation: $D(f(x_1, \ldots, x_n))$ holds only if $p(x_1, \ldots, x_m)$ evaluates to *true*; operations without explicitly indicated domain clause are considered total.

Example.

type Nat = **spec**
[zero: @;
succ: @ ⟶ @;
"+", "−": @, @ ⟶ @;
"=", "<>", "<", "<=", ">", ">=": @, @ ⟶ Boolean]
{**forall** x, y: @. **dom** x − y: y <= x;
zero < succ(x) == true; succ(x) < zero == false;
succ(x) < succ(y) == x < y; zero = succ(x) == false;
succ(x) = zero == false; succ(x) = succ(y) == x = y;
x <= y == x < y | x = y; x > y == y < x;
x >= y == y =< x; x <> y == ¬(x = y);
x + zero == x; x + succ(y) == succ(x + y);
x − zero == x; x − x == zero;
succ(x) − y == succ(x − y)};

type SeqOfNat = **spec**
[empty: @;
append: Nat, @ ⟶ @;
head, length: @ ⟶ Nat;
tail: @ ⟶ @;
has: @, Nat ⟶ Boolean;
is_empty: @ ⟶ Boolean;
"=", "<>": @, @ ⟶ Boolean]
{**forall** x, y: Nat, s, s1: @. **dom** head(s): ¬is_empty(s);
is_empty(empty) == true; is_empty(append(x, s)) == false;
length(empty) == 0; length(append(x, s)) == length(s) + 1;
head(append(x, s)) == x;
has(empty, x) == false; has(append(y, s), x) == x = y | has(s, x);
tail(empty) == empty; tail(append(x, s)) == s;
append(x, s) = empty == false; empty = append(x, s) == false;

[2] the predicate can also contain existentially quantified variables, see an example in Section 7.

s = s1 == head(s) = head(s1) & tail(s) = tail(s1);
s <> s1 == ¬(s = s1)};

type RecordNatSeqOfNat =
[create_rec: Nat, SeqOfNat ⟶ @;
p1: @ ⟶ Nat;
p2: @ ⟶ SeqOfNat]
{**forall** x, x1: Nat, s, s1: SeqOfNat, r: @.
p1(create_rec(x, s)) == x; p2(create_rec(x, s)) == s};

function if: Boolean, Nat, Nat ⟶ Nat
{**forall** x, y: Nat, p: Boolean.
if(true, x, y) == x; if(false, x, y) == y}.

5. Specification classes

Some data type specifications can share the same subset of operation signatures and the same subset of axioms. The notion of a *specification class* is introduced to make use of this feature. This notion corresponds to the notion of a *type class* which has become widely known due to [18] and which was originally proposed under the name *sype* in [20].

Let Σ be a signature, $\underline{C}$ be a set of names (of specification classes), C be a name from $\underline{C}$, and $\theta E = \langle \theta, E^\theta \rangle$ be a specification constructed like a data type specification (Section 4). For each pair $\langle C, \theta E \rangle$, we say that θE is a *specification class* marked with the class name C if no algebra of Σ assigns a set to C and a function to any operation signature from θ.

If T is a type name marking a data type specification $\langle \phi, E^\phi \rangle$ and C is a name marking a specification class $\langle \theta, E^\theta \rangle$, we say that type name T[3] *belongs to* class C (or T *is of* class C) and write $T \in C$ if $\theta \subseteq \phi$ and $E^\theta \subseteq E^\phi$. Thus, the specification of a data type belonging to a certain class contains all operation signatures and all axioms of that class and, possibly, some other operation signatures and axioms. Semantically, we can view a specification class as a set of implementations of the data types belonging to the class.

Example. Let us have the following specification class (introduced with the keyword **class**):

class EQUAL = **spec**
["=", "<>": @, @ ⟶ Boolean]
{**forall** x, y: @, **exist** z: @.
x = x == true; x = y == y = x; x = z & z = y == x = y;
x <> y == ¬(x = y)}.

[3]Sometimes we say "data type T".

Any data type possessing the operations " = " and " <> " specified as above belongs to this class.

For two specification classes $\langle C, \theta E\rangle$ and $\langle C1, \theta E1\rangle$, where $\theta E = \langle \theta, E^{\theta}\rangle$ and $\theta E1 = \langle \theta 1, E1^{\theta 1}\rangle$, we say $C1$ is a *subclass* of C ($C1 \subseteq C$), if $\theta \subseteq \theta 1$ and $E^{\theta} \subseteq E1^{\theta 1}$.

Notation. Specifying a subclass, we usually inherit the specification of its superclass.

Example.

class ORDERED = **spec** EQUAL – *specification of EQUAL is inherited*
["<=", ">=", "<", ">": @, @ $\longrightarrow$ Boolean] .
{**forall** x, y: @, **exist** u: @.
x <=x == true; x <= u & u <= y == x <= y;
x <= y & y <= x == x <= y; x < y == x <= y & $\neg$(x = y);
x >= y == y <= x; x > y == y < x}.

The class ORDERED inherits the class EQUAL, and, therefore, any data type belonging to this class must possess all operations of the class EQUAL and extra operations introduced in the specification of ORDERED. We also use inheritance when specifying data types, i.e., a data type specification indexed with a class name inherits the specification of the class indicated.

Since any type name is a member of the set $TYPE$, we consider in the sequel that $TYPE$ is a class name marking an empty specification class (any data type is a data type of this class).

6. Generic data types

We propose the following way of constructing the names in $TYPE$ (which will be called *type terms* from now on), using two nonintersecting sets of names, $\underline{S}$ and $\underline{R}$: if $S \in \underline{S}$, then S is a type term; if $T_1, \ldots, T_n$ are type terms and $R \in \underline{R}$, then $R(T_1, \ldots, T_n)$ is a type term.

Let now

$$int^t(R(T_{11}, \ldots, T_{1n})) = Spec1 \text{ and } int^t(R(T_{21}, \ldots, T_{2n})) = Spec2,$$

where $R(T_{11}, \ldots, T_{1n})$ and $R(T_{21}, \ldots, t_{2n})$ are type terms and $Spec1$ and $Spec2$ are data type specifications. We say the data type $R(T_{11}, \ldots, T_{1n})$ is a *sibling* of the data type $R(T_{21}, \ldots, T_{2n})$, if the replacement of each T_{1i} with T_{2i}, $i = 1, \ldots, n$, in $Spec1$ converts it into $Spec2$. We can propose a special way of constructing a part of the function int^t for a family of data type siblings.

Let $q_1 : C_1, \ldots, q_k : C_k$ be names (of type parameters) indexed with class names. A pair $\langle R(q_1 : C_1, \ldots, q_k : C_k), Spec\rangle$ (where $R \in \underline{R}$ and $Spec$ is a data type specification additionally using $q_1, \ldots, q_k$ as type names in operation signatures and operators from $C_1, \ldots, C_k$ in axioms) is part of the function int^t, such that for any type term T_i of class C_i, $i = 1, \ldots, k$, $int^t(R(T_1, \ldots, T_k)) = Spec[q_1/T_1, \ldots, q_k/T_k]$, where $Spec[q_1/T_1, \ldots, q_k/T_k]$ is a data type specification produced by replacing each q_i in $Spec$ with T_i.

A pair $\langle R(q_1 : C_1, \ldots, q_k : C_k), Spec\rangle$ is normally called a *generic type specification*, and R is called a *generic type* or *type constructor*. The replacement of type parameters with type terms in both parts of a generic type specification is called a *generic type instantiation*. Note that due to the use of the function int^t, we do not need to introduce a special semantics for generic data types. A generic type specification in this approach is just a way of defining this function. This corresponds one to one to the practice of modern programming languages regarding generic data types as templates.

Two type terms $R(T_{11}, \ldots, T_{1n})$ and $R(T_{21}, \ldots, T_{2n})$ are equivalent if T_{1i} and T_{2i}, $i = 1, \ldots, n$, are the same type name or if they are equivalent.

Example.

type Seq(T: EQUAL) = **spec** EQUAL
[empty: @;
append: T, @ ⟶ @;
head: @ ⟶ T;
length: @ ⟶ Nat;
tail: @ ⟶ @;
has: @, T ⟶ Boolean;
is_empty: @ ⟶ Boolean;
"=", "<>": @, @ ⟶ Boolean]
{**forall** x, y: T, s, s1: @. **dom** head(s): ¬is_empty(s);
is_empty(empty) == true; is_empty(append(x, s)) == false;
length(empty) == 0; length(append(x, s)) == length(s) + 1;
head(append(x, s)) == x;
has(empty, x) == false; has(append(y, s), x) == x = y | has(s, x);
tail(empty) == empty; tail(append(x, s)) == s;
append(x, s) = empty == false;
s = s1 == head(s) = head(s1) & tail(s) = tail(s1);
s <> s1 == ¬(s = s1)}.

An instantiation $Seq(Nat)$ produces a data type specification exemplified in Section 4.

7. Built-in type constructors

Some data type constructors are built-in, i.e., they follow special rules of type term creation. These are enumeration, record, and union types.

A type term $(p1, p2, \ldots, pm, pn)$, where $pi, i = 1, \ldots, m, n$, is a name, denotes the following *enumeration type* specification:

spec ORDERED – *inherits the specification of the class ORDERED*
[p1, p2, ..., pm, pn: @;
first, last: @;
succ, pred: @ ⟶@]
{**forall** x, y: @. **dom** pred(x): x>p1; **dom** succ(x): x<pn;
first == p1; last == pn;
succ(p1) == p2; ...; succ(pm) == pn; pred(succ(x)) == x;
succ(x) = x == false; x < succ(x) == true}.

A type term $Record(p1 : T1; p2 : T2; \ldots; pn : Tn)$, where $pi, i = 1, \ldots, n$, is a name (of a projection function) and Ti is a type term, denotes the following *record type* specification:

type Record(T1, T2, ..., Tn: TYPE) = **spec**
[create_rec: T1, T2, ..., Tn ⟶ @;
p1: @ ⟶ T1;
p2: @ ⟶ T2;
...
pn: @ ⟶ Tn;
{**forall** x1: T1, x2: T2, ..., xn : Tn [End of sentence needs a space after it.].
p1(create_rec(x1, x2, ..., xn)) == x1;
p2(create_rec(x1, x2, ..., xn)) == x2;
...
pn(create_rec(x1, x2, ..., xn)) == xn}.

An instantiation $Record(Nat, Seq(Nat))$ produces a data type specification similar to that of $RecordNatSeqOfNat$ exemplified in Section 4.

Notation. If r is a record and p is a projection function name, we normally write $r.p$ for $p(r)$.

A constructor (type term) *Union(T1, T2, ..., Tn)*, where $Ti, i = 1, \ldots, n$, is a type term, denotes the following *union type* specification:

type Union(T1, T2, ..., Tn: TYPE) = **spec**
[T1: T1 ⟶@;
T2: T2 ⟶@;
...
Tn: Tn ⟶@;

get_T1: @ $\longrightarrow$T1;
get_T2: @ $\longrightarrow$T2;
...
get_Tn: @ $\longrightarrow$Tn]
{**forall** x1: T1, x2: T2, ..., xn: Tn, u: @, **exist** y1: T1, y2: T2, ..., yn : Tn.
dom get_T1(u): u = T1(y1); **dom** get_T2(u): u = T2(y2);
...
dom get_Tn(u): u = T2(yn);
get_T1(T1(x1)) == x1; get_T2(T2(x2)) == x2;
...
get_Tn(Tn(xn)) == xn}.

A special case of a union type is the data type Any which is the union of all data types of a given specification.

For any algebra A of a given signature with a union type

$$U = Union(T1, T2, \ldots, Tn),$$

a typing function $\tau : |A|_U \longrightarrow TYPE$ is defined as follows: if $u \in |A|_U$, $T \in \{T1, T2, \ldots, Tn\}$, and $a \in |A|_T$, then $\tau(u) = T$ iff $u = T(a)$.

8. Generic functions

Definition.

1) a *Generic type term* is a pair $\langle (q1 : C1, \ldots, qk : Ck), oT^q \rangle$, where $q1 : C1, \ldots, qk : Ck$ are names (of type parameters) indexed with class names and oT^q is an operation type constructed by extending the set of type terms of classes $C1, \ldots, Ck$ with $q1, \ldots, qk$, respectively;
2) a *Generic function signature* is a pair $op : goT$, where op is an operator and goT is a generic type term.

We now allow the set Δ to contain generic function signatures.

Notation. We put type parameters in the brackets **gen** ... **op.**

Example.

if: **gen** T: TYPE **op** Boolean, T, T $\longrightarrow$ T.

If $op^q : \langle (q1 : C1, \ldots, qk : Ck), oT^q \rangle$ is a generic function signature and $T1, \ldots .Tk$ are type terms such that each $Ti, i = 1, \ldots, k$ belongs to the class Ci, then $op^q(T1, \ldots, Tk) : oT$ is an instantiated function signature, where oT is an operation type obtained from oT^q by replacing each qi with Ti; $op^q(T1, \ldots, Tk)$ is called an *instantiated operator.* Instantiated operators

are used for producing data terms in the same way as ordinary operators do (Section 3, paragraph 3).

Example.

$$if(Nat)(p, x, y).$$

According to the extension of the independent function set with generic function signatures, an algebra A of a given signature is extended with a set of functions map^{Aq}, one for each generic function signature $op^q : \langle(q1 : C1, \ldots, qk : Ck), oT^q\rangle$; such a function binds an instantiated operator $op^q(T1, \ldots, Tk)$ to a function as it is described in Section 2.

The definition of the interpretation function $eval^A$ is extended in the following way: for an instantiated operator $op^q(T11, \ldots, T1k) : T1, \ldots, Tn \rightarrow T$,

$$eval^A(op^q(T11, \ldots, T1k)(t1, \ldots, tn)) = map^{Aq}(op^q(T11, \ldots, T1k)(eval^A(t1), \ldots, eval^A(tn)).$$

Notation. Instead of $op^q(T11, \ldots, T1k)(t1, \ldots, tn)$, we write $op^q(t1, \ldots, tn)$, where it seems appropriate.

A generic function specification consists of a generic function signature and a set of axioms.

9. Algebra modification

9.1. Function updates

Let A be a Σ-algebra and let a function signature

$$f : T_1, \ldots, T_n \longrightarrow T$$

be mapped into a function f^A. We say that the algebra A *evolves* to an algebra B by a function update if:

1) all data type signatures and detached operation signatures of Σ, except f, have in B the same interpretations as in A;
2) there are some $a_1 \in |A|_{T_1}, \ldots, a_n \in |A|_{T_n}$ such that $f^B(a_1, \ldots, a_n)$ produces a result different from $f^A(a_1, \ldots, a_n)$ and f^B is the same as f^A elsewhere.

If f is a constant name, this means that f gets a new value in algebra B.

Example. Assume that we had a function signature

$$f : Nat \longrightarrow Nat$$

mapped in algebra A into a function f^A satisfying the equation

$$\textbf{forall}\, x : Nat.f(x) == 0.$$

If now we construct an algebra B with a function f^B satisfying the equations

$$\textbf{forall}\, x : Nat.f(zero) == 0; \quad f(succ(x)) == 1;$$

we can say that A evolves to B.

The modification of a function is done by one or more function *updates*. Let A be a Σ-algebra, and let a function signature

$$f : T_1, \ldots, T_n \longrightarrow T$$

be mapped into a function f^A. A *primitive update*, α, of f^A in A is a pair $(\langle a_1, \ldots, a_n \rangle, a)$, where $a \in |A|_T, a_i \in |A|_{T_i}, i = 1, \ldots, n$. To fire α in A, transform A into a new algebra B, such that $f^B(a_1, \ldots, a_n) = a$ and f^B is the same as f^A elsewhere. Such a primitive update of an algebra A is denoted by $\mu 1(A, f^A, \alpha)$.

An *update*, β, of f^A in A is a set of primitive updates of f^A. If f^A has the signature $f : T_1, \ldots, T_n \longrightarrow T$, then β is *consistent* if there are no $\alpha 1 \in \beta$ and $\alpha 2 \in \beta$, such that $\alpha 1 = (\langle a_1, \ldots, a_n \rangle, a)$ and $\alpha 2 = (\langle a_1, \ldots, a_n \rangle, a')$, where $a \neq a'$. To fire a consistent β at A, fire simultaneously $\mu 1(A, f^A, \alpha)$ for all $\alpha \in \beta$. Such an update of f^A in an algebra A is denoted by $\mu 2(A, f^A, \beta)$. To execute an inconsistent β at the given algebra A, do nothing; the new algebra B is the same as A.

An *update set* in A is a set of updates. To execute an update set, execute each member of the set simultaneously. Note that an update set is inconsistent if at least one of its members is inconsistent.

9.2. Algebra invariant

In the sequel, we consider that independent functions (constants) are classified in two sets: *static functions* which do not change when the algebra evolves and *dynamic functions* which do change when the algebra evolves. Data type implementations do not change. In this way, we come to the notions of a static algebra, instant algebra, and algebra invariant. We denote the sets of signatures of static and dynamic functions by Δ and Δ', respectively.

Definition. Let $\Sigma = \langle TYPE, \Phi, \Delta, int^t \rangle$ and A be a Σ-algebra called a *static algebra*. An (*instant*) algebra[4], IA, of the signature $I\Sigma =$

[4]sometimes also called a *state*

$\langle TYPE, \Phi, I\Delta, int^t \rangle$, where $I\Delta = \Delta \cup \Delta'$ is built by extending A with a set of dynamic functions with signatures in Δ'. A set, I, of $I\Sigma$-algebras, such that all algebras in I have the same static algebra, is called an $I\Sigma - invariant$.

Since all algebras in an $I\Sigma$-invariant I have the same carrier, we let $|I|$ denote the carrier of an algebra belonging to I and $|I|_T$ denote the set of elements associated with the data type name T.

An initial state for a specification $\langle I\Sigma, E \rangle$ is given by an instant algebra of this specification. A dynamic function (constant) supplied with an axiom (axioms) is considered to be initialized (static functions must be initialized). A noninitialized dynamic function (constant) is considered to be totally undefined in this algebra.

9.3. Transition rules

Algebra updates are specified by means of special *transition rules*. A transition rule is a special element of the $I\Sigma$ term algebra called a *transition term*. The evaluation function $eval^I : |I| \times |W_{I\Sigma}(X)| \longrightarrow |I|$ is used to transform one instant algebra into another according to a transition rule.

9.3.1. Basic transition rules

Definition. Let $f : T_1, \ldots, T_n \longrightarrow T$ be a dynamic function signature, t_i, $i = 1, \ldots, n$, be a ground term of type T_i and t be a ground term of type T. Then

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

is a transition rule called a *primitive update instruction*.

Semantics. If A is an algebra of an $I\Sigma$-invariant I, then

$$eval^I(A, f(t_1, \ldots, t_n) := t) = \mu 1(A, f^A, \alpha),$$

where $\alpha = (\langle eval^A(t_1), \ldots, eval^A(t_n) \rangle, eval^A(t))$.

Definition. Let $f : T_1, \ldots, T_n \longrightarrow T$ be a dynamic function signature, then

$$\textbf{forall}\, x_1 : T_1, \ldots, x_k : T_k . f(t_1, \ldots, t_n) := t,$$

where t_i, $i = 1, \ldots, n$, is a term of type T_i using no other variables except those from the set $\{x_1, \ldots, x_k\}$, and t is a term of type T using no other variable except those used in t_i, is a transition rule called an *update instruction*.

Semantics.

$$eval^I(A, \textbf{forall}\, x_1 : T_1, \ldots, x_k : T_k.\, f(t_1, \ldots, t_n) := t) = \mu 2(A, f^A, \beta),$$

where $(\langle a_1, \ldots, a_n \rangle, a) \in \beta$ if there exists a substitution $\sigma : \{x_1, \ldots, x_k\} \to |W_{O\Sigma}|$, such that $a_1 = eval^A(\sigma t_1), \ldots, a_n = eval^A(\sigma t_n), a = eval^A(\sigma t)$ (σ is identity in case of ground terms).

Example. A transition rule

$$x := x + 1$$

will transform an algebra A into an algebra B so that

$$x^B = x^A + 1.$$

A transition rule

$$\textbf{forall}\, x : Nat.\, f(x) := f(x) + 1$$

will change the function in such a way that for each $a \in |A|_{Nat}$,

$$f^B(a) = f^A(a) + 1.$$

An update instruction $f(t_1, \ldots, t_n) := undef$ means $D(f(t_1, \ldots, t_n)) := false$, i.e., $f(t_1, \ldots, t_n)$ becomes undefined (note that $undef$ is not a term denoting a special value, this is just a special word providing some kind of syntactic sugar). Respectively, an expression $f(t_1, \ldots, t_n) = undef$ means $D(f(t_1, \ldots, t_n))$.

9.3.2. Three rule constructors

More complex transition rules (or simply *rules* in the sequel) are constructed recursively from basic transition rules by means of three rule constructors: *block constructor*, *conditional constructor*, and *tagcase constructor*.

The block constructor. If $R_1, \ldots, R_n$ are rules, then **begin** $R_1, \ldots, R_n$ **end** is a rule called a *block*.

Semantics.

$$eval^I(A, \textbf{begin}\ R_1, \ldots, R_n\ \textbf{end}) = \{eval^I(A, R_1), \ldots, eval^I(A, R_n)\}.$$

In other words, to execute a block of rules, execute all of them simultaneously.

Example. Let x, y, z be dynamic constants of type *Nat* and f be a function from *Nat* to *Nat*. Then the execution of a block

$$\textbf{begin } f(x) := y, y := x, x := z \textbf{ end}$$

will produce:

$$f^B(x) = y^A \quad y^B = x^A \quad x^B = z^A$$

The conditional constructor. If k is a natural number, $g_0, \ldots, g_k$ are Boolean terms, and $R_0, \ldots, R_k$ are rules, then the following expression is a rule called a *conditional transition rule*:

if g_0 **then** R_0
elseif g_1 **then** R_1
.
.
.
elseif g_k **then** R_k
endif

If the term g_k is the Boolean constant *true*, then the last **elseif** clause can be replaced with "**else** R_k".

Semantics. If R is a conditional transition rule, then

$$eval^I(A, R) = eval^I(A, R_i)$$

if g_i holds in A, but every g_j with $j < i$ fails in A. If every g_i fails in A, then $eval^I(A, R) = \oslash$.

The tagcase constructor. If u is a term of type $Union(T_1, T_2, \ldots, T_n)$ or of type Any, $R_1, R_2, \ldots, R_k$ are rules and $k <= n$, then the following expression is a rule called a *tagcase transition rule*:

tagcase u **of**
$T_1 : R_1$,
$T_2 : R_2$,
...
$T_k : R_k$
endtag.

Semantics. Let R be a tagcase transition rule, then each u in R_i is converted to type T_i and $eval^I(A, R) = eval^I(A, R_i)$ if $\tau(u) = T_i$ holds in A, but every $\tau(u) = T_j$ with $j < i$ fails in A. If every $\tau(u) = T_i$ fails in A, $eval^I(A, R) = \oslash$. Thus, the tagcase constructor permits us to regard a union type value as a value of the type needed, this facility is not provided by the conditional constructor.

Remark. The last $T_k : R_k$ can be replaced with **else** R_k; then u is evaluated as a union type term in R_k.

9.3.3. Guarded update

A guarded update instruction is a rule of the form **if** g **then** R **endif**, where R is a rule.

Semantics. Let $R1 =$ **if** g **then** R **endif**. Then $eval^I(A, R1) = eval^I(A, R)$ if $g = true$; $eval^I(A, R1) = \oslash$ otherwise.

9.3.4. Record field update

If r is a dynamic constant of type $Record(p_1 : T_1; \ldots p_i : T_i; \ldots; p_n : T_n)$, then an update instruction

$$r.p_i := t_i$$

means $r := create\_rec(r.p_1, \ldots, t_i, \ldots, r.p_n)$[5].

9.4. Dependent functions

A static function cannot be updated by means of a transition rule. However, if a function specification uses a dynamic function symbol, the function should be updated according to the new value of the dynamic function used in the specification each time the corresponding dynamic function is updated.

Example 1. Let us have the following definitions:

dynamic const s: Seq(Int);
depend const num_of_elem: Nat; {num_of_elem == length(s)}.

Each time the constant s gets a new value, the constant num_of_elem gets a new value, too. We call such a function (constant) a *dependent* function (constant) and introduce it with a special keyword **depend**. A dependent function corresponds to a function in programming languages which uses a global variable and to a non-elementary access function of [16].

With the use of dependent functions, an algebra update proceeds in two stages:

1. At stage 1 a transition rule is executed.
2. At stage 2 all dependent functions touched by the transition rule are reimplemented.

Remark. We do not touch here the problem whether a dynamic function is really reimplemented each time the corresponding dynamic function is updated or a result is computed each time the function is called.

[5]The idea is suggested to the author by Giuseppe Del Castillo.

9.5. Temporary updates

An update can be temporary, i.e., be valid only in evaluation of a certain term. Thus, let R be a rule, t be a term of type T, and $t1$ denote a term **temp** R **in** t. Then, for any algebra A, $t1^A = t^B$, where B is the algebra produced by updating A according to R.

The rule has proved to be useful in defining the formal semantics of the Oberon WITH statement [17].

9.6. Procedures

A procedure generally serves for the execution of the same transition rule for different terms. If $p, n_1, \ldots, n_k$ are names, $T_1, \ldots, T_k$ are type terms, and R is a transition rule using the names $n_1, \ldots, n_k$, then

proc $p : T_1, \ldots, T_k; p(n_1, \ldots, n_k) =$ **begin** R **end**

is a procedure definition. The names $n_1, \ldots, n_k$ are called *formal parameters.*

A new transition rule, a procedure call, is introduced for procedure invoking. If

proc $p : T_1, \ldots, T_k; p(n_1, \ldots, n_k) =$ **begin** R **end**

is a procedure definition and $t_1, \ldots, t_k$ are ground terms of types $T_1, \ldots, T_k$, respectively, then $p(t_1, \ldots, t_k)$ is a transition rule called a *procedure call.* The terms $t_1, \ldots, t_k$ are called *actual parameters.*

Semantics. To execute $p(t_1, \ldots, t_k)$, replace in R each $n_i, i = 1, \ldots, k$, with t_i and fire R.

Remark 1. If an actual parameter replaces a formal parameter used as the name of a dynamic constant modified in R, it must be either the name of a dynamic constant or a dynamic function application.

Remark 2. A procedure can be called recursively.

Remark 3. Procedures are counterparts of modifiers introduced in [16].

10. General form of a specification

In the most general case, a specification can consist of the following parts:

1) specifications of data types and static functions;
2) specifications of dynamic functions (the absence of an axiom part means a totally undefined function);
3) specifications of dependent functions;
4) definitions of procedures;

5) a transition rule.

Only the first part is obligatory, all the others are optional. If we have only the first part, we have a traditional algebraic specification of a static state of a system. A specification consisting of parts 1, 2, and 5 defines an algorithm in the manner close to that of the traditional Gurevich Machine. A specification without the last part defines the behavior of a dynamic system in the manner close to that described in [16]. A specification including all the parts describes a particular algorithm in terms of the behavior of a dynamic system.

11. Sample examples

11.1. Specification of a stack machine

This is an example from [21] rewritten with the use of data types. The stack machine computes expressions given in reverse Polish notation, or PRN. It is supposed that the PRN expression is given in the form of a list where each entry denotes a natural number or an operation. The stack machine reads one entry of the list at a time. If the entry denotes a number, it is pushed onto the stack. If the entry denotes an operation, the machine pops two items from the stack, applies the operation and pushes the result onto the stack. At the beginning, the stack is empty.

type Oper = ('+','−','∗','/');
type Doper = $Union(Nat, Oper)$;

type Stack = **spec**
[empty: @;
push: Nat, @ ⟶ @;
pop: @ ⟶ @;
top: @ ⟶ Nat];
{− *axioms are conventional*}

type List = Seq(Doper);

dynamic const S: Stack = empty;
dynamic const Arg1, Arg2: Nat; − *initially undefined constants*
dynamic const F: List; − *initialized by a demon*

tagcase head(F) **of**
Nat: **begin** S := push(head(F), S), F := tail(F) **end**,
Oper:
  if Arg1 = undef **then**

```
    begin Arg1 := Nat(top(S)), - Arg1 is defined now
          S := pop(S)
    end
  elseif Arg2 = undef then
    begin Arg2 := Nat(top(S)), - Arg2 is defined now
          S := pop(S)
    end
  else
    begin S := push(apply(head(F), Arg1, Arg2), S),
          F := tail(F),
          Arg1 := undef, - Arg1 is undefined now
          Arg2 := undef - Arg2 is undefined now
    end
  endif
endtag
```

Note that all the operations used in the example are now formally defined in contrast to [21].

11.2. Identifier table manipulation

The identifier table stores some data for each identifier definition. It can be block-structured according to block nesting. Typical operations are creation of an empty identifier table, insertion of identifier data in the current block, checking whether an identifier is defined in the current block, checking whether an identifier is defined in the program, fetching identifier data, and deletion of all identifier definitions of the current block.

Let *Name* be the type of identifiers and *Defdata* be the type of the identifier definition data, the concrete structure of this type is unimportant in this example. Then we can define the identifier table as a dynamic function

dynamic function $id\_table : Name, Nat \longrightarrow Defdata$;

Taking into account a block structure of a modern programming language, we define a constant

dynamic const $cur\_level : Nat = 0$

to indicate the current level of identifier definition.

According to the initialization conventions, *id_table* is initialized with a totally undefined function and *cur_level* is initialized with 0.

The insertion of a new entry into the table is done by the following procedure

proc insert_entry: Name, Defdata;
insert_entry(id, d) = **begin** id_table(id, cur_level) := d **end**;

To check whether an identifier is defined in the current block, we define the following dependent function:

depend function defined_current: Name ⟶ Boolean;
{**forall** id: Name. defined_current(id) == D(id_table(id, cur_level))};
The dependent function
depend function name_defined: Name, Nat ⟶ Boolean;
{**forall** id: Name, k: Nat.
name_defined(id, 0) == false;
name_defined(id, k) == D(id_table(id, k)) | name_defined(id, k-1)};
will check whether an identifier is defined in the program. An identifier definition data is fetched from the table by the dependent function
depend function find: Name, Nat ⟶ Defdata;
{**forall** id: Name, k: Nat.
dom find(id, k): k > 0;
find(id, k) == **if** D(id_table(id, k)) **then** id_table(id, k) **else** find (id, k-1)};

Finally, the deletion of all identifier definitions of the current block is done by the following procedure:

proc delete_level;
delete_level =
begin forall id: Name. id_table(id, cur_level) := undef;
cur_level := cur_level -1
end.

12. Some related work

A concept of an algebraic specification with implicit state is introduced in [16]. The main idea is to represent states of a system by algebras and dynamic operations by transformations between them. Transformations are defined by means of so-called modifiers which are counterparts of transition rules of Gurevich Machines. A state can be analyzed by a number of simple and complex analyzers. The approach is mainly directed on system specification rather than on algorithm specification, while our approach is directed on both and has an imperative style of specification. However, complex modifiers gave rise to procedures, and complex analyzers gave rise to dependent functions presented in this paper.

The same idea of an implicit state in terms of a new mathematical structure, d-oid (dynamic object identity), is given in [22, 23]. A d-oid is a set of instant structures (e.g., algebras) and a set of dynamic operations (transformations of instant structures with a possible result of a definite sort). Here transition rules of Gurevich machines and modifiers of [16] are replaced with dynamic operations. The approach is highly associated with object-oriented data representation. With this purpose, algebra elements are considered as

objects supplied with unique identifiers preserving object identities in the process of algebra transformations. The approach deals with models, e.g., it assumes that sorts (like in a conventional Gurevich machine) can grow and shrink, and do not address the issue of specifying the class of such behaviors, which is (like in [16]) our aim. We also doubt that a dynamic operation producing a side-effect (i.e., an operation both transforming an instant structure and yielding a result) and resembling a side-effect function in programming languages should be part of a specification language.

A multimodal logic MLCM (Modal Logic of Creation and Modification), which is a variant of dynamic logic, is used in [24] to formalize reasoning about evolving algebras. Another formalism for the formal definition of Gurevich Machines (Evolving Algebras in the paper) based on Di-algebras is suggested in [25]. These papers are just two other formalizations of the Gurevich's idea of evolving algebras.

Dynamic abstract types are introduced in [26, 27]. In [26], a dynamic abstract type consists of an abstract data type and a collection of dynamic operations; four levels of its specification are proposed: value type specification, instant structure specification, dynamic operation specification, and higher-level specification. The first two levels mainly correspond to algebraic specifications of abstract data types with fixed and loose semantics, respectively. The specification of the third level is a suitable extension of algebraic specification to define transformations between instant structures. The specification of the fourth level includes higher-level dynamic generators and dynamic operations for update, composition and communication of different instances of dynamic abstract types. In [27], no direct definition of a dynamic abstract type is given. Instead of this, formal definitions of a static framework and of a dynamic framework over a static framework are given. These papers are closely related in the sense that while the first one gives an informal proposal, the second one proposes concrete mechanisms of static and dynamic frameworks according to the approach stated in [22, 23]. In contrast to this approach, we are still satisfied with static data types which taken together with dynamic functions provide a simple and powerful mechanism for the formal definition of dynamic systems.

13. Conclusion and further work

An approach to combining static type-structured algebraic specifications, Gurevich Machines and algebraic specifications with implicit state is proposed in the paper. It has resulted in Typed Gurevich Machines using fully specified data types and static functions in the process of creation and use of dynamic functions. The facilities include concrete and generic data types, type classes, independent concrete and generic functions, procedures, and

transition rules. The use of well-specified data types has allowed us to propose the exclusion of such a "strange" transition rule as import constructor [1] serving to extend a universe with a new element. At the same time it is proposed to include a tagcase transition rule which is highly useful in operations with objects of union types. The proposal does not prohibit the inclusion of some other transition rules like **Elect**, **Collect**, and **Choose** [28].

The structure of a specification resembles the structure of a program written in a modern programming language: static part, where specifications of data types and static functions are counterparts of data type and function definitions, and dynamic part, where transition rules are counterparts of imperative statements. This gives good chances for the specification language (which can be called an *abstract imperative language*) to be accepted by the system designing/programming community. A careful choice of a set of transition rules and specification constructs is still needed before a concrete specification language can be proposed. A move to it is demonstrated by the extension of the basic set of transition rules given in [1] with sequential updates [24] and a kind of for-loop [29].

The approach presented is based on static signatures, which means that all algebra transformations are done within the same signature. One of the possible directions of the future work is the introduction of signature transforming operations with corresponding algebra extension (addition of new data types and/or functions) or reduction (deletion of existing components).

Acknowledgements. The author thanks Giuseppe Del Castillo, Philippe Kutter, and Felix Cornelius for lengthy discussions of the subject and helpful comments on drafts of earlier versions of the paper.

References

[1] Y. Gurevich, *Evolving Algebras 1993: Lipary Guide*, Specification and Validation Methods, Oxford University Press, 1994.

[2] E. Börger, E. Riccobene, *A formal specification of Parlog*, Semantics of Programming Languages and Model Theory, Gordon and Breach, 1993, 1–42.

[3] E. Börger, D. Rosenzweig, *A matematical definition of full Prolog*, Science of Computer Programming, 1994.

[4] E. Börger, Y. Gurevich, D. Rosenzweig, *The bakery algorithm: Yet another specification and verification*, Specification and Validation Methods, Oxford University Press, 1994.

[5] E. Börger, I. Durdanovich, D. Rosenzweig, *Occam: Specification and Compiler Correctness. Part 1: Simple mathematical interpreters*, Proc. PROCOMET'94

(IFIP Working Conference on Programming Concepts, Methods and Calculi), North-Holland, 1994, 489–508.

[6] Y. Gurevich, J. Huggins, *The semantics of the C programming language*, Computer Science Logic, Lect. Notes in Comp. Sci., **702**, 1993, 274–309.

[7] P. Glavan, D. Rosenzweig, *Evolving Algebra Model of Programming Language Semantics*, IFIP 13th World Computer Congress, 1994, Vol. 1: Technology/Foundations, Elsevier, Amsterdam, 416–422.

[8] C. Wallace, *The semantics of the C++ Programming Language*, Specification and Validation Methods, Oxford University Press, 1994.

[9] P.W.Kutter, *Dynamic Semantics of the Oberon Programming Language*, Term thesis, ETH Zurich, 1996.

[10] A.V. Zamulin, *The Database Specification Language RUSLAN*, Siberian Division of the Russian Academy of Sciences, Institute of Informatics Systems, Preprints No. 28, 29, Novosibirsk, 1994, 35 p. (ftp:// *math.tulane.edu*, directory "pub/zamulin", files "Ruslan1.ps.Z", "Ruslan2.ps.Z").

[11] A.V. Zamulin, *The Database Specification Language RUSLAN: Main Features*, East-West Database Workshop (Proc. 2nd International East-West Database Workshop, Klagenfurt, Austria, September 25–28, 1994), Springer (Workshops in Computing), 1994, 315–327.

[12] D. Kapur, D.R. Musser, *Tecton: a framework for specifying and verifying generic system components*, Rensselaer Polytechnic Institute, Computer Science Technical Report 92–20, July, 1992.

[13] J. V. Guttag, J. J. Horning, *Larch: Languages and Tools for Formal Specification*, Springer, 1993.

[14] M. Broy, C. Facchi, R. Grosu, et al, *The Requirement and Design Specification, Language Spectrum, An Informal Introduction, Version 1.0.* , Technische Universitaet Muenchen, Instituet fuer Informatik, April 1993.

[15] A.V. Zamulin, *Typed Gurevich Machines*, Institute of Informatics Systems, Preprint No. 36, Novosibirsk, (ftp://xsite.iis.nsk.su/pub/articles/tgm.ps.gz).

[16] P. Dauchy, M.C. Gaudel, *Implicit State in Algebraic Specifications*, International Workshop on Information Systems – Correctness and Reusability (IS-CORE'93), Informatik-Berichte, No. 01/93, 1993.

[17] A.V. Zamulin, *Specification of an Oberon Compiler by means of a Typed Gurevich Machine*, Institute of Informatics Systems of the Siberian Division of the Russian Academy of Sciences, Report No. 589.3945009.00007-01, Novosibirsk, 1997.

[18] P. Wadler, S.Blott, *How to make ad-hoc polymorphism less ad-hoc*, Conf. Record of the 16th ACM Annual Symp. on Principles of Progr. Lang., Austin, Texas, January 1989.

[19] M. Broy, M. Wirsing, *Partial Abstract Data Types*, Acta Informatica, **18**, 1982, 47–64.

[20] R. Nakajima, M. Honda, H. Nakahara, *Hierarchical Program Specification: a Many-sorted Logical Approach*, Acta Informatica, **14**, No. 2, 1980, 135–155.

[21] Y. Gurevich, *Evolving Algebras: An Attempt to Discover Semantics*, Current Trends in Theoretical Computer Science, World Scientific, 1993, 266–292.

[22] E. Asteziano, E. Zucca, *A Semantic Model for Dynamic Systems*, Modeling Database Dynamics, Volkse 1992, Workshops in Computing, Springer Verlag, 1993, 63–83.

[23] E. Asteziano, E. Zucca, *D-oids: a Model for Dynamic Data Types*, Mathematical Structures in Computer Science, **5(2)**, June 1995, 257–282.

[24] R. Groenboom, R. Renardel de Lavalette, *Reasoning about Dynamic Features in Specification Languages*, Workshop in Semantics of Specification Languages, Springer Verlag, 1994, 340–355.

[25] H. Reichel, *Unifying ADT and Evolving Algebra Specifications*, Bull. of EATCS, No. 59, 1996, 112–126.

[26] H. Ehrig, F. Orejas, *Dynamic Abstract Types: An Informal Proposal*, Bull of EATCS, No. 53, 1994, 162–169.

[27] E. Zucca, *From Static to Dynamic Data Types*, Mathematical Foundations of Computer Science 1996, Lect. Notes in Comp. Sci., **1113**, 1996, 579–590.

[28] Y. Gurevich, *ASM Guide 1997*, University of Michigan, 1997.

[29] E. Boerger, D. Rosenzweig, *The WAM-Definition and Compiler Correctness*, Logic Programming: Formal Methods and Practical Applications, North-Holland Series in Computer Science and Artificial Intelligence, 1994.