Algebraic Structures of Programs: First Steps to Algebraic Programming

Nikolai N. Nepeivoda

Program System Institute of RAS
Pereslavl–Zalessky, Russia

Abstract. Basic algebraic notions are introduced which can be used to describe and to transform program-data complexes for traditional languages, restricted computation, non-numeric computation and modeling. General algebraic program systems (GAPS) are free of assumptions what computing (programming) systems have as their particular constructs. First of all we argue why current technique is sometimes not adequate. Then we show how to interpret notions of abstract algebra (groupoid structure) as super-von Neumann computational structure and how to express many useful structures and notions in an algebra. Then GAPS are introduced and basic mathematical results are stated including the precise criterion when a given system of actions over programs can be added to given programming language. Various examples of GAPS are given. And at last we show possible primitives of 'structured algebraic modeling' (programming).

Keywords: program algebras, reversive programs, near-reversivity, algebraic computing

The general sructure of paper

Our introduction gives an informal insight why so abstract algebraic formalism is reasonable for some problems of programming, what are its motivations and main difference from other types of program algebras.

First of all we state basic analogies between program and computation structures and algebraic notions in groupoid (a structure with non-associative binary operation). The main result here is that elements of groupoids can be viewed as well as data, actions, actions over actions and so on. Thus we get a 'functional' programming on entities which are not necessary functions. The side results are that many control and data structures can be expressed purely algebraically without any reference to particular control structures.

Then we define GAPS (Generic algebraic program structure) — a very abstract algebraic description of programs.

Basic properties of GAPS are studied and it is proved that many traditional languages can be represented as GAPS because they model the λ-calculus. There are also example of dully non-traditional computing structures modeled by

GAPS. Two precise criteria when a given language can be enriched or extended by the given system of program transformations are stated.

And at last we define structures which allow to construct near-reversive computations: reversive data types, mirrors, interruptors, crystals, wheels. An example how to 'program' and to design a scheme by these structures is given.

1 Introduction

Reversible computing was the big bang inspiring this investigation. More precisely, it was a cognitive dissonance between brilliant ideas of von Neumann, Landauer, Toffoli, Feynman, Merkle and more than 30 years of stagnation in "applied research". Sorry, it is not a stagnation, but intensive running in a squirrel cage wheel (maybe different for different 'schools'). Usually such effect is induced by some assumption which is so common that it becomes almost invisible. But really it stands on the way, pollutes this way but nobody is brave enough to point to this obstacle (a *sacred cow*, as it is called in [5]).

Here this cow is that computations are binary. Remember that mathematical model of invertible functions is a *group*. Works (including mines) in this direction were based on groups till 2012.

Bennett pointed out that reversivity can beat Landauer limit only if the control system is also based on invertible transformations. By control system we mean here a system of entities organizing the execution of elementary actions as elements of a computing system. These ones can be the statements and structures of programming language, the connectors in a physically realized scheme and so on.

It is necessary to remember the difference between two notions. **Reversive** computing, reversive actions are fully invertible. a^{-1} can be used both before and after a, they are in some sense bijective. **Reversible** ones are retractable ones, they are injective, this action can be undone but not prevented [8]. The store for all intermediate results is a way to provide reversibility but not reversivity. The multiplication of integers by 2 is reversible function but not reversive one.

There are important and fundamental invertible commands which cannot be represented as functions and cannot be embedded into group structure. First of all this is the *absolute mirror* or inversion of a program segment.

We use the list postfix notation for function or action application: $(a\ F)$ where F is an action applied to a.

Definition 1. ***Absolute mirror*** *is the action transforming a preceding action into its inversion:* $(f\ M) = f^{-1}$. *UNDO is the action undoing the last block of actions:* $\{a_1;\ \ldots\ a_n\}\,;\mathit{UNDO} = \mathit{EMPTY}$.

Example 1. M and UNDO are non-associative entities and almost never can be embedded into structure of (semi)group:

$$((a \circ b) \circ M) = (b^{-1} \circ a^{-1}) \qquad (a \circ (b \circ M)) = (a \circ b^{-1})$$

$$\{\mathsf{a};\ \mathsf{b}\};\ \mathsf{UNDO} \neq \mathsf{a};\ \mathsf{b};\ \mathsf{UNDO}.$$

There is another aspect of a problem. Program and algorithmic algebras is a classic branch of computer science. They take start at 60'ths. Glushkov [1] defined algorithmic algebras using operators corresponding to base constructions of structured programming, Maurer [2] introduced and studied abstract algebras of computations (program generic algebras, PGA) more like to computer commands, based on semigroups and on operators representing **goto**s. Various kinds of algorithmic and dynamic algebras follow these two basic ideas.

For main results of Glushkov approach we refer to [3, 4]. For current investigations in PGA a good summary are [6, 7]. There are more than hundred works on algebras of programs cited in [11] where a comprehensive survey up to 1996 is done. Almost all these works are devoted to Turing complete systems. But even reversible systems cannot be Turing complete [13, 14].

Full reversivity restricts class of problems more severely. Factorial, multiplication and division of integer numbers are irreversible. All arithmetic operations on standard representation of real numbers are not reversive. If elementary actions are invertible a problem itself can be not invertible.

Example 2. Sorting is irreversive because we forget initial state and it cannot be reconstructed from sorted array. Assembling of Rubik cube is irreversive due to similar reasons.

Even if we go beyond the problem of heat pollution during computations reversivity arises due to development of computer element base. Quantum computations are reversive. Molecular computations are reversive. Superconductor computations often are reversive. Nanocrystal computations are reversive. So studying of computations where majority of operations are reversive is necessary.

Concrete functions are often defined through common λ-notation. This does not mean that all our constructs are based on λ-calculus.

2 Algebraic structures from the point of view of programming

Definition 2. ***Signature*** *σ is a list of symbols: functions (binary functions can be used as infix operations), constants and predicates. There is a metaoperator arity(s) giving for each function or predicate symbol its number of arguments. If there is the predicate $=$ it is interpreted as equality. $s \in \sigma$ where sigma is a signature means that symbol s is in σ.*

Algebraic system **S** *of signature σ is a model of this signature in the sense of classical logic. We have a data type (or nonempty set) S called* ***carrier*** *and second order function ζ such that for function symbols $\zeta(f) \in S^{\mathrm{arity}(f)} \to S$, $\zeta(f) \in S^{\mathrm{arity}(f)} \to \{\mathrm{false}, \mathrm{true}\}$ for predicates, $\zeta(c) \in S$ for constants.*

Definition 3. ***Groupoid*** *is an algebraic system of signature $\langle \circledast, = \rangle$, where* $\mathrm{arity}(\circledast) = 2$. *Its carrier is often denoted G, symbol for its binary operation can be various for different groupoids. Element e is an* ***identity*** *(or* ***neutral***

element**) if* $\forall x\, x \circledast e = e \circledast x = x$. *Element* 0 *is* ***zero *if* $\forall x\, x \circledast 0 = 0 \circledast x = 0$. ***Left identity*** *is such* a *that* $\forall x\, a \circledast x = x$. ***Left zero*** *is such* a *that* $\forall x\, x \circledast a = a$. *Analogously for right identity and zero.* ***Idempotent*** *is such* a *that* $a \circledast a = a$. *Groupoid is a* ***semigroup*** *if its operation is associative. It is commutative if its operation is commutative. It is* ***injective*** *if*

$$\forall x, y, f\, x \circledast f = y \circledast f \Rightarrow x = y.$$

Semigroup is a ***monoid*** *if it has an identity element. Monoid is a* ***group*** *if there is identity element and for each* x *exists* y *such that* $f \circledast g = g \circledast f = e$. *Bigroupoid is an algebraic system with two binary operations.*

Morphism of algebraic structures is a map of their carriers preserving all functions and constants, and truth (but not necessary falsity) for all predicates. **Isomorphism** is a bijective morphism preserving falsity.

Now we comment this notions from informatic point of view.

Non-associative groupoid gives a set of expressions isomorphic to ordered directed binary trees or Lisp lists.

Example 3. $((a \circledast (b \circledast c)) \circledast (a \circledast d))$ is an expression for Lisp ((a b c) a d) id $\circledast$ is CONS. It defines the binary tree on fig. 1.

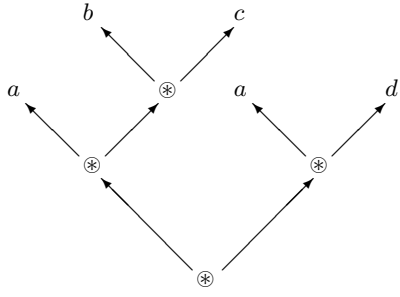

Fig. 1. $((a \circledast (b \circledast c)) \circledast (a \circledast d))$ as a tree

Abstract algebra is extremely valuable for program architects and analysts because each isomorphism gives another representation for data, each morphism gives a structure which can be viewed as approximation for full data or valuable analogy.

If operation is commutative tree becomes inordered and direct analogies with lists vanish. If it is associative list becomes linear and tree becomes a one-dimensional array. Any associative operation can be interpreted as function composition. It follows from classical theorem

Theorem 1. *Each semigroup is isomorphic to semigroup of functions with composition as operation. [10]*

This theorem gives a full criterion when a space of actions can be viewed as a space of functions: associativity. Thus Example 1 shows that *actions not always can be modeled as functions*. Because a semigroup operation always can be viewed as function composition it is usually denoted $\circ$.

The following example shows central role of semigroups in algebraic informatics.

Example 4. Each collection of functions or relations closed under composition is a semigroup. Thus we can treat functions definable by programs as the single semigroup even our language is strongly typized. Programs are simply functions undefined on data not belonging to types of their arguments. Each finite automate can be treated as a finite semigroup and vice versa any finite semigroup can be viewed as a finite automate. Actions in each program language with sequential composition (usually denoted by semicolon) also form a *syntactic* semigroup even their effect cannot be treat as a function.

The last sentence is the second keystone for very abstract notion of program algebra independent from assumptions on concrete operators of language and of functionality of actions.

One example of a commutative non-associative algebra of actions will be used below and is sufficiently simple and expressive to show many peculiarities.

Example 5. A commutative groupoid formalizing a simple game. Its carrier is the set of three elements {well, scissors, paper} (denoted {w,s,p}). Our operation gives for each pair the winner.

$$w \circledast s = w; \quad w \circledast p = p; \quad s \circledast p = s; \quad x \circledast x = x.$$

$(w \circledast s) \circledast p = w \circledast p = p$, $w \circledast (s \circledast p) = w \circledast s = w$. So simultaneous (or independent) actions of two players effect in commutativity of the operation.

So commutativity of a semigroup means that our actions are functional and independent. From the point of view of computations they can be executed in various ways (sequentially in any order; (partially) parallel; and so on). From the logical point of view our actions can be viewed as spending of a money-like resource (Girard's linear logic [9]). We take into account only a total amount of resources in the 'account'. Each operation spends them independently.

3 Groupoid as a computing structure

A groupoid can be viewed as a 'functional'[1] computing structure of 'super-von-Neumann' kind. Each element a of groupoid can be viewed also as the action with the effect $\lambda x.\,(x \circledast a)$. Furthermore it is actions on actions and so on. Data and commands are the same.

So now we look on many interesting elements and properties from the point of view of 'algebraic computer'.

[1] But remember that actions not always are functions!

1. Associativity means that our actions are without 'side effects' and can be viewed as functions.
2. Unity e means 'do nothing'. Moreover each right unity $((x \circledast e) = x$ is 'no operation' command.
3. 0 is the fatal error. Mathematically 0 in semigroups of relations is the empty relations (function which is never defined). If groupoid has more than one element that 0 is not (left, right) identity.
4. Left zero $(z \circledast x) = z$ is an output, the final result of a computation. More precisely if there is the zero then output can be described as $\forall x\,(x \neq 0 \supset (z \circledast x) = z)$ (**left near-zero**).
5. Right zero $(x \circledast z) = z$ is at the same time so called 'quine' (program giving itself) and an input, the initial value overriding all earlier. More precisely if there is the zero then input can be described as $\forall x\,(x \neq 0 \supset (x \circledast z) = z)$ (**right near-zero**).
6. Idempotent $(z \circledast z) = z$ is a pause.
7. Right contraction $(a \circledast f) = (a \circledast g) \supset f = g$ is practically almost useless property: each program acts differently on each elements than any others. But there is an important exception. If our operation is associative and each element has the inversion $a \circ a^{-1} = e;\ a^{-1} \circ a = e$ then our semigroup becomes a group. Each space of bijective functions can be viewed as a group and vice versa. Elementary fully invertible actions on some data type form a group. In a group we can 'prevent' an action not only to **undo** it.
8. If $(x \circledast a) \circledast \tilde{a} = x$ then $\tilde{a}$ is a **weak right inverse** for a. It grants undoing of a.
9. A **one-way pipe** p is such element that

$$(x \circledast p) = y \supset (y \circledast m) = 0.$$

10. A subgroupoid can be viewed as a block of program or construction.
11. Direct product of groupoids means that computation can be decomposed into independent branches corresponding components of direct product.

So we see

Algebraic programming is functional and super-von-Neumann by its essence.

Each groupoid generates the semigroup of actions

Definition 4. ***Action*** *of groupoid element sequence* $f_1, \ldots, f_n$ *is a function*

$$\varphi_{f_1;\ldots;f_n} = \lambda x.\,(\ldots((x \circledast f_1) \circledast f_2)\ldots) \circledast f_n).$$

This semigroup not always fully describes program actions.

Lemma 1. *Actions of groupoid form a semigroup.*

Proof. Composition of $\varphi_{f_1;\ldots;f_n}$ and $\varphi_{g_1;\ldots;g_k}$ is the action corresponding to

$$\varphi_{f_1;\ldots;f_n;g_1;\ldots;g_k}.$$

To make algebraic computation more structured we will partially decompose our groupoid into subsystems. Three main kinds of subsystems are:

1. block;
2. connector;
3. computation-control pair

Block is a subgroupoid. All actions with block elements do not lead out of block. In the simplest case block has inside all its outputs and often also inputs as corresponding algebraic values. To make our algebraic computation structured we demand

If x and y are from different blocks then $(x \circledast y) = 0$.

There is an important transformation of groupoid. *Dual groupoid* $\mathbf{G}'$ to $\mathbf{G}$ is the groupoid with the same carrier and operation $(x * y) = (y \circledast x)$. In both program and technique realizations dual groupoid is implemented by the same structure where arguments (signals) are exchanged. Dual to associative system is associative.

$$(a * (b * c)) = (c \circ b) \circ a; \quad ((a * b) * c) = c \circ (b \circ a).$$

Nevertheless when computing system is modeled as a groupoid dual system to a subsystem is to be represented by another block. In this case we denote the element of dual corresponding to a as a'. $'$ is not an internal operation. In realization a and a' usually will be the same value, element or signal.

Connectors transfer information and control between blocks. They are outside connecting blocks. We demand for connectors c that if $(x \circledast c) = y$ then $(x \circledast y) = 0$. Two most valuable kinds of connectors are mirrors and one-way pipes.

A **mirror** m is such an element that

$$(x \circledast m) = y \equiv (\tilde{y}' \circledast m) = \tilde{x}'.$$

It seems now that mirrors and pipes are to be the only connectors admitted in structured algebraic computations.

A **computation-control pair** is a groupoid decomposed into direct product $G1 \times G2$ where $G1$ has no inputs and outputs. Realizing this pair we are to grant that any output value computed in $G2$ will interrupt process in $G1$.

Example 6. Let a clever, brave and polite black cat is creeping up to mouse in a black room. $G1$ here means a crawling process and $G2$ means a sensor interrupting crawling and initializing attack when the mouse detects the cat. We have the following diagram Crawling and attack easily are described by semigroups (see [18]). Interruption and pipe are non-associative operators. Sensor is almost associative and reversive: its actions form a group in which one element is replaced by output.

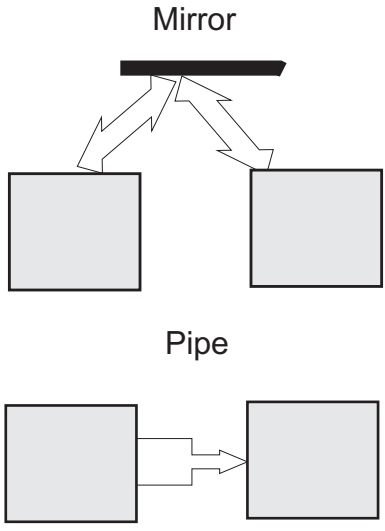

Fig. 2. Connectors

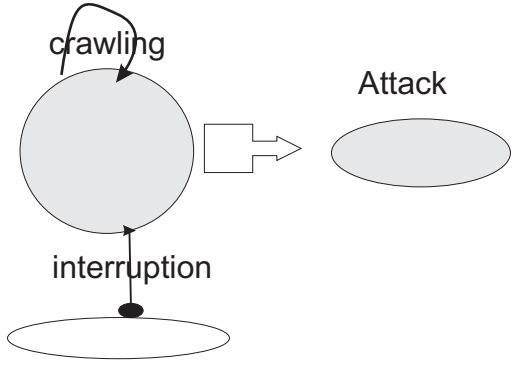

Fig. 3. Black polite cat

Let us consider another interesting possibility arising because high order essences arise in algebraic computing from the very beginning, on level of elementary actions.

It is known that current program systems are like to dinosaurs burdened by gigatons of code. Usually explosive increasing of size of program systems is considered as objective inevitable factor. But there is a side way.

Example 7. High order transformations can shorten programs and sometimes computations in tower of exponents. For example compare two sequences of definitions below

$$\begin{gathered}\Phi_1 = \lambda f.\, \lambda x.\, ((x\ f)\ f);\\ \Phi_{n+1} = \lambda \Psi_n.\, \lambda \Psi_{n-1}.\, ((\Psi_{n-1}\ \Psi_n)\ \Psi_n).\end{gathered} \tag{1}$$

$$(x\ (f\ (\Phi_1 \dots (\Phi_{n-2}\ (\Phi_{n-1}\ \Phi_n^k)) \dots))) = (x\ \ f^{2^{2^{\cdots 2}}}\ (k \text{ times})) \tag{2}$$

So groupoid induces at least two additional structures: finite order functionals and the semigroup of actions. Higher order functionals need no extra support here. To describe algebraic system effectively it suffices to add an operations converting a system of elements into a single block.

4 General algebraic program structure (GAPS)

Considerations above lead to the formal notion. Let us denote an operation of applying f to x by $x \star f$, an operation composing two elements into the single block $a \circ b$.

Definition 5. ***General algebraic program structure*** *(GAPS) is a bigroupoid where the following holds:*

$$((x \circ y) \circ z) = (x \circ (y \circ z)) \tag{3}$$
$$((x \star f) \star g) = (x \star (f \circ g)) \tag{4}$$

If GAPS has unity and (or) zero they are to satisfy equations

$$(0 \star x) = (x \star 0) = 0 \tag{5}$$
$$(x \star e) = x \tag{6}$$
$$x \circ 0 = 0 \circ x = 0 \tag{7}$$
$$e \circ x = x \circ e = x. \tag{8}$$

So application is not necessary associative (and in fact we don't need composition if it is associative) and composition is associative and gives us possibility to encapsulate a number of elements into the single one. Unity is a right unity.

Lemma 2. *Each groupoid* **G** *can be extended up to GAPS.*

Proof. Consider the following Horn theory. It has constants for all elements of **G**. It includes the diagram of **G** (i. e. the set of true closed elementary formulas), a new constant **B** and the axioms

$$((x \star f) \star g) = (x \star (f \star (g \star \mathbf{B}))) \tag{9}$$

$$(f \star ((g \star (h \star \mathbf{B})) \star \mathbf{B})) = ((f \star (g \star \mathbf{B})) \star (h \star \mathbf{B})) \tag{10}$$

Its initial model is a desired GAPS.

We are sure that GAPS is in algebraic sense a minimal system describing every imaginable collection of elementary actions and programs (constructs) composed from them.

The second operation allows us to formulate many properties more expressively and effectively.

Definition 6. *An element $x \neq 0$ is a divisor of zero if there is such $y \neq 0$ that $x \circ y = 0$. An element y is a right inverse for x if $x \circ y = e$. An element x^{-1} is the inverse of x if $x \circ x^{-1} = x^{-1} \circ x = e$. Two elements are mutually inverse if $a \circ b \circ a = a,\ b \circ a \circ b = b$.*

So each of mutually inverse elements grants undoing (prevention) for another on codomain (domain) of the last one (if they are partial functions).

The next simple theorem shows that every system of functions (every semigroups) can be enriched by every system of total program transformations, so it is called **theorem on abstract metacomputations**. It needs some preliminary discussion.

Remember the notion of enrichment for algebraic systems (its direct analogy for classes in programming is called specialization). Let a signature σ_1 extends a signature σ. An algebra $\mathbb{A}_1$ in σ_1 is enrichment of $\mathbb{A}$ in σ if all carriers, functions, constants and predicates from σ are untouched.

Let there is a morphism $\alpha : \mathbb{G} \to (G \to G)$ of the semigroup $\mathbb{G}$ into the semigroup of maps of its carrier such that $(e\ \alpha) = \lambda x.\, x$, $(0\ \alpha) = \lambda x.\, 0$. From the programmer's point of view it gives interpreter, translator, compiler or supercompiler giving an executable module for a high order program.

An example of such morphism. $(x\ \alpha) = \lambda x.\, 0$ for all divisors of zero. $(x\ \alpha) = \lambda x.\, x$ for others. $x \star y = x \circ y$.

Theorem 2. *Each semigroup $\mathbb{G}$ can be enriched to GAPS such that $(x \star f) = (x\ (f\ \alpha))$.*

Proof. Laws for GAPS hold. Non-trivial is only (10).

$$((x \star f) \star g) = ((x\ (f\ \alpha))\ (g\ \alpha)) = (x\ ((f\ \alpha) \circ (g\ \alpha))) =$$
$$(x\ (f \circ g\ \alpha)) = (x \star (f \circ g))$$

□

Lemma 3. *For each action φ of groupoid there is an element α such that $(x \star \alpha) = (x\ \varphi)$.*

Proof. Let $f_1, \ldots, f_n$ be an arbitrary action. Applying (4) and associativity of $\circ$ get

$$(\ldots((x \circledast f_1) \circledast f_2)\ldots) \circledast f_n) = (x \circledast f_1 \circ f_2 \cdots \circ f_n).$$

□

Example 8. Some 'natural' assumptions destroy GAPS. For example if $\forall f\,(e \star f) = f$ two operations coincide.

$$(x \star y) = ((e \star x) \star y) = (e \star (x \circ y)) = x \circ y.$$

Consider this phenomenon. 0 is the fatal error and we cannot do with it inside of system. e can be interpreted as an empty program but program transformer can generate non-empty code starting from empty data. See example 14 below.

Example 9. Direct application of process to enrich groupoid to GAPS almost always leads to infinite structure. Often it is possible to remove superfluous constructs and get the finite GAPS.

Consider groupoid from the example 5.First of all we write down all different actions of groupoid in the form `shorter result: longer sequence of elements`.

w: wwp, s: wss, p: psp, ws: wws, sw: www, ps: sss, sp: pss, wp: ppp, pw: pwp,wsp: pps, spw: pww, pws: sws.

Semigroup of functions of actions consists of twelve elements and is not commutative though initial groupoid is commutative. Often the groupoid operation $*$ can be extended to GAPS by different ways. For example here there are at least two extensions:

(ws * sw) = wssw = wsw = ws;
(ws * sw) = (ws)*(s*w) = (ws)w = (w*w)(s*w) = ww = w.

Now consider a problem when and how we can join some system of program transformations with a given program system (= a semigroup of program) and to get the single language of metaprogramming. There are three natural subproblems.

1. How to join a language with a system of transformation not changing the language (the semigroup)?
2. How to preserve some sub-language (sub-semigroup) maybe converting other constructions into transformations?
3. How to extend a language to a metalanguage not changing notions inside of the given language?

Definition 7. *Let some system of transformations* $\mathbb{A}$ *is given as system of functions on carrier of a semigroup* $\mathbb{G}$ *described by theory* Th *and desired properties of transformations are written down as an axiomatic theory* Th_1*. It is* ***conservative*** *if there is a GAPS enrichment* $\mathbb{AG}$ *of* $\mathbb{G}$ *such that every* $\varphi \in \mathbb{A}$ *is represented as an action of groupoid* $(x \star \alpha) = (x\ \varphi)$ *for some* α*. It is* conservative over the subgroup $\mathbb{G}_0$ *if it is conservative and in* $\mathbb{AG}$ $a \star b = a \circ b$ *for all elements of* $\mathbb{G}_0$*. It is* ***admissible*** *if there is a semigroup* $\mathbb{G}_1$ *such that* $\mathbb{G} \subseteq \mathbb{G}_1$ *and* $\mathbb{A}$ *is conservative for* $\mathbb{G}_1$*.*

Let Th_P *is a theory* Th *in which all quantifiers are restricted by unary predicate* P*:* $\forall x\, A(x)$ *is replaced by* $\forall x\, (P(x) \supset A(x))$*;* $\exists x\, A(x)$ *is replaced by* $\exists x\, (P(x) \& A(x))$*.*

$\mathbb{A}$ ***strongly admissible*** *if it is admissible and for resulting algebra theories* Th_1 *and* Th *remain valid.*

Lemma 4. *Collection of actions* $\mathbb{A}$ *is conservative iff its closure is isomorphic to subsemigroup of* $\mathbb{G}$*.*

Proof. By 3 if enrichment is successful then each element of the semigroup generated by $\mathbb{A}$ represents action of some element of $\mathbb{G}$. Thus closure of $\mathbb{A}$ is embedded into $\mathbb{G}$.

Vice versa, if the closure of $\mathbb{A}$ can be embedded into $\mathbb{G}$ then each action from $\mathbb{A}$ can be represented by its image by this embedding.
□

Example 10. Let there be only one action: inversion of programs $\mathbf{M}$ such that $((a\ \mathbf{M})\ \mathbf{M}) = a$. To enrich a semigroup od programs by this action is possible iff there is an element of order 2 in this semigroup: $f \neq e \& f \circ f = e$. No matter how this f acts as function.

Lemma 5. *Collection of actions* $\mathbb{A}$ *is conservative over* $\mathbb{G}_0$ *iff there is monomorphism* ψ *of its closure into* $\mathbb{G}$ *such that for each* f *such that* $(f\ \psi) \in \mathbb{G}_0$ $(a\ f) = (a \circ (f\ \psi))$ *holds.*

Example 11. Using this criterion we can test possibility of enrichment up to language of metacomputations considering strings as programs and remain untouched the sublanguage of numerical computations.

The following theorem is proved in [18]. Its proof requires model-theoretic technique, is long and resulting construction is not always algorithmic one.

Theorem 3. *(2012–2014) Let a system of actions* $\mathbb{A}$ *is described by a theory* Th_1*, and* Th *is a theory of semigroup* $\mathbb{G}$ *and* P *is a new unary predicate.*
Then $\mathbb{A}$ *is strongly admissible over* $\mathbb{G}$ *iff there is a partial surjection* $\psi : \mathbb{G} \to \mathbb{A}$ *such that* $(g_1 \circ g_2\ \psi) = (g_1\ \psi) \circ (g_2\ \psi)$ *if all results are defined and the theory* $\mathrm{Th}_1 \cup \mathrm{Th}_P$ *is consistent.*

There is an important consequence of this theorem.

Proposition 1. *Every set of actions $\mathbb{A}$ is admissible over $\mathbb{G}$ if both theories consist only from facts (true formulas of the form $[\neg](a\{\circ,*\}b) = c$).*

Example 12. Consider an additive group $\mathbb{Z}_3$ and add actions of its objects $\{0,1,2\}$ as well, scissors and paper from example 5. Their actions can be described as functions on $\mathbb{Z}_3$ with values 002, 011, 212. Now we extend wsp-groupoid to twelve elements semigroup as in example 9. To conform with it is necessary to add identity which don't belongs to this closure and denote this monoid $\mathbb{C}$. Consider a direct product of $\mathbb{C} \times \mathbb{Z}_3$ and define actions as follows.

$$(\langle c, x\rangle \star \langle d, y\rangle) = \langle c \circ d, (x + y\ d)\rangle\,.$$

This GAPS contains $\mathbb{Z}_3$. Elements of $\mathbb{C}$ can be considered as commands and elements of $\mathbb{Z}_3$ as data. $(x\ d)$ is application of sequence of actions to an element coded by x and coding of the result. Commands transform as a semigroups but their effects as groupoid.

There is another way to define GAPS on the same carrier.

$$(\langle c, x\rangle \star \langle d, y\rangle) = \langle (c \star d), (x + y\ (c \star d))\rangle\,.$$

Thus the problem is when this extension is computable. It can be infinite and non-computable even for finite theories, finite semigroup and collection of actions. Though this theorem is pure one it gives a valuable negative criterion.

A practical consequence. *If a system of program transformations destroys properties of program or forced to make different programs equal it is incorrect.*

There is a particular case when our extension is semi-computable.

Definition 8. ***Horn formula*** *(quasi-identity) is a formula*

$$\forall x_1, \ldots, x_k\,(Q_1 \& \cdots \& Q_n \supset P),$$

where x_i all its variables, and all Q_i, P are predicates.

If our theories consist of Horn axioms then GAPS can be constructed as the factorization of the free GAPS according to provable identity of terms (the initial model).

Example 13. Because λ-calculus lies in foundations of the modern mathematical theory of Turing-complete program systems (see [11]) it suffices to construct a model of λ-calculus as a GAPS. To do this we take an equivalent representation of λ-calculus as combinatory logic and take its basis $\{\mathbf{I}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$ [12] described in our terms as

$$\begin{gathered}(x \star \mathbf{I}) = x\\ (x \star (f \star (g \star \mathbf{B}))) = ((x \star f) \star g)\\ (x \star (f \star (g \star \mathbf{C}))) = ((x \star g) \star f)\\ (x \star (y \star (z \star \mathbf{S})) = ((x \star y) \star (x \star z)).\end{gathered}$$

Adding the axiom of associativity of composition **B** (10) and the definition

$$f \circ g = (f \star (g \star \mathbf{B}))$$

we get a model of λ-calculus and using this one we get models for all standard and almost all non-standard programming languages.

To get a model for typed λ-calculus it suffices to add zero 0 and to redefine $\star$ as zero when types are not conforming. To get the identity it suffices to add the following axioms:

$$(\mathbf{I} \star (f \star \mathbf{B})) = f \qquad (f \star (\mathbf{I} \star \mathbf{B})) = f.$$

Example 14. One more example how to turn program system into GAPS. Let we have an algorithmic language in which symbols are operators and concatenation of strings is composition of programs (e.g. Brainfuck Brainfuck). Then empty string is program doing nothing. It can be naturally represented as GAPS. $a \circ f$ is simply concatenation. $a \star f$ will be defined as follows. If f is a correct program then its value on a is $a \star f$. If f yields an error or it is syntactically incorrect then our value is 0.

We see that the result of action over an empty program can be arbitrary.

Example 15. In 1972 one research stopped one little step before GAPS [16].

Consider the alphabet $\{K, S, (,)\}$. Its symbols translate into combinators as

$$(K\ \varphi) = \mathbf{K} \quad (S\ \varphi) = \mathbf{S} \quad (\text{`('}\ \varphi) = \mathbf{B} \quad (\text{`)'}\ \varphi) = \mathbf{I}$$

The result of string translation is defined recursively: (a is a symbol, σ is a string):

$$(a\sigma\ \varphi) = ((\sigma\ \varphi) \star (a\ \varphi)).$$

This interpretation turns the combinatory logic into a Brainfuck-like programming language. But resulting GAPS is not a GAPS for the combinatory logic. It also includes syntactically incorrect constructs like))))**SK**((.

Example 16. If our semigroup is a monoid then universal function in GAPS is trivial $\mathbf{U} = e$:

$$(x \star (f \star \mathbf{U})) = (x \star f),$$

A partial evaluator is not trivial:

$$(f \star (x \star \mathbf{PE})) = (x \star f),$$

A fixed point operator

$$((f \star \mathbf{Y}) \star f) = (f \star \mathbf{Y}),$$

is trivial if there is 0: $\mathbf{Y} = 0$.

GAPS can easily express some functional restrictions on the programs. For example the papers [17, 18] are mathematically describing and investigating GAPS for reversive, reversible and completely non-invertible programs are des.

5 Some tools to compose algebraic programs

Algebraic programming is to be a collection of tools to compose and decompose algebraic substructures of GAPS. Some of these tools were outlined for groupoids in the section 3.

The important tool of (de)composition is the construct used in the example 12. Elements of $\mathbb{Z}_3$ can be considered as data and elements of the semigroup as commands. Now we formulate a general case for this construction.

Definition 9. ***Semidirect product*** *of GAPS. Let there are two GAPS: a GAPS of commands C and a GAPS of data D. Let $\varphi : \mathbf{C} \to \mathrm{Hom}(\mathbf{D}, \mathbf{D})$ is a morphism of the semigroup of commands into the semigroup of morphisms of data semigroup. Then semidirect product $C \rtimes D$ is $C \times D$ with the following operations:*

$$\langle c_1, d_1\rangle \circ \langle c_2, d_2\rangle = \langle c_1 \circ c_2, d_1 \circ (d_2\ (c_2\ \varphi))\rangle$$
$$\langle c_1, d_1\rangle \star \langle c_2, d_2\rangle = \langle c_1 \star c_2, d_1 \star (d_2\ (c_2\ \varphi))\rangle\,.$$

This is a generalization of the semidirect product for semigroups.

Now we introduce data types and their connectors taking into account that each data is also an action and that there will be no direct information or control flow from one type to other type.

Definition 10. ***Type systems*** *on GAPS is a system of subGAPSes T_i such that if $i \neq j$, $a \in T_i$, $b \in T_j$ then $(a \star b) = 0$ and $T_i \cap T_j \supseteq \{0\}$.*

Connector *(between T_i and T_J) is an element c not belonging to any type such that if $(a \star c) = b \neq 0$ then there are $i \neq j$ such that $a \in T_i$, $b \in T_j$.*

Dual *T_i' to type T_i is a GAPS for which there is a bijection to T_i $x \leftrightarrow x^p rime$ such that if $(x \star y) = z$ then $(z' \star y^p rime)e = x'$. Dual is* ***perfect*** *if $(x \circ y)' = (y' \circ x')$.*

Mirror *is a connector m such that if $(a \star m) = b$, $a \in T_i$, $b \in T_j$ then $(b' \star m) = a'$. Mirror is* ***perfect*** *if it is connector between T_i and its perfect dual T_i'.*

Reversive type *is a subGAPS R for which there exists the ideal mirror M such that for each $x, y \in R$*

$$((x \star y)' \star (y \star M)) = x' \quad ((x' \star (y \star M))' \star y) = x.$$

Crystal *is a subGAPS where $\star$ forms a group.*

Pipe *is a connector p such that $(x \star p) = y \neq 0 \supset (y \star p) = 0$.*

Result element *of type T_i is a left zero of T_i according to both operations.*

Interruption structure *is a type with a carrier $T \times \{1, 2\}$ where T is GAPS, there are no results in T except maybe 0 and operations are defined as follows:*

$$(\langle a, 1\rangle \star \langle b, x\rangle) = \langle (a \star b), x\rangle\,; \tag{11}$$
$$(\langle a, 2\rangle \star \langle b, y\rangle) = \langle a, 2\rangle \ \text{ if } b \neq 0 \tag{12}$$

Wheel *is an interruption structure based on group $\mathbb{Z}_n$.*

Interruption controlled type *is a system of type T_i, interruption structure S and the pipe p between T_i and $\{1, 2\}$ such that $(x \star p) = 1$ if x is not a result and $(x \star p) = 2$ if x is a result.*

Interruption controlled type can be easily represented through semidirect product but in this representation the pipe (which is necessary for effective program or physical realization) is hidden and the number of elements grows essentially. Each type can be transformed into a type with a given subset of results not disturbing the actions giving other than a result. Using these elements we sometimes can reduce a very complex GAPS to a composition of types, mirrors, pipes, semidirect products and interruption controls. We need no loops and conditional statements here.

An example below shows how to compute a very large power of an element of a given complex algebraic construct through a precomputed representation of power in Fibonacci system. Almost all actions in this program (system) are completely invertible (reversive) if given algebra of data is reversive. Only the initialization of the system and the final interruption are non-invertible.

Example 17. Let us try to apply the same action a large number of times. This corresponds to computing $a \circ b^{\omega}$ in a group. Then ω is represented in Fibonacci system. This can be easily made by usual computer. Let k be the number of bits in the representation of ω. The two predicates are computed and transferred to a reversive program: `(i fib_odd)`, `(i fib_even)`. The first one is 1 iff i is odd and the corresponding digit is equal to 1. `(i fib_even)` is the same for even indexes.

Type loop is resulted from the additive group of integers making integer 0 its output value. Pipe sends its interrupt to externally implemented (maybe physically) group `tp` which is controlled by two boolean commands exchanging commutation of values during compositions, implemented by mirrors and described as conditional operators. These mirrors use two precomputed arrays of booleans to commute inputs of the next operations.

```
PROGRAM Fibonacci_power
DEFINITIONS
int atom n
GROUP tn: external nooutputs
tp atom var a,b,d
tp atom e
(tp,tp) var c is (a,b)
constant e=E
INTERRUPTOR int loop output (0);
loop atom k
int atom var i [0..k] guarded
PIPE(interruption) p: (k,tn); boolean atom l;
predicate [i] fib_odd, fib_even
END DEFINITIONS

INPUT
read a, k
b← a
```

```
i ← 1
l ← TRUE
d ← E
read fib_odd, fib_even
END INPUT

{i;1},
{l; ⊕ true}
par sync (k,tp,i)
{
tp{
   {c; ∘ if l then (e,a) else (b,e) fi};
   {d; ∘ if (i fib_odd) then
       a else if (i fib_odd) then b else e
   fi fi};
}
k{+ (-1)};
i{+ (1)};
}

OUTPUT
write d
END OUTPUT
```

Rough description of implementation of this program is given on fig. 4

6 Conclusion

The main mathematical result of this paper is Theorem 3. It states that many kinds of programs and other computing schemes can be viewed as GAPS and fully states conditions when a (partially described) system of transformations can be correct for the given system. For example we need no descriptions of usual programming languages here because they were described by the λ-calculus which is GAPS.

The last section shows that complex algebras often can be reduced to structures formed with simpler ones in a way like to analog computers and structured programming. Here is a big amount of open problems because (say) a systematic mathematical theory of finite approximations for infinite algebraic systems do not exists now (even for groups).

Maybe the most valuable property of GAPS is that they can be easily adopted to describe functionally restricted classes of programs and computations. Some advanced results in this direction (for reversive, reversible, completely non-invertible programs and dynamic systems) are presented in [17, 18]. It appears now that algebraic programming and computing is the most general existing concept and it is very flexible.

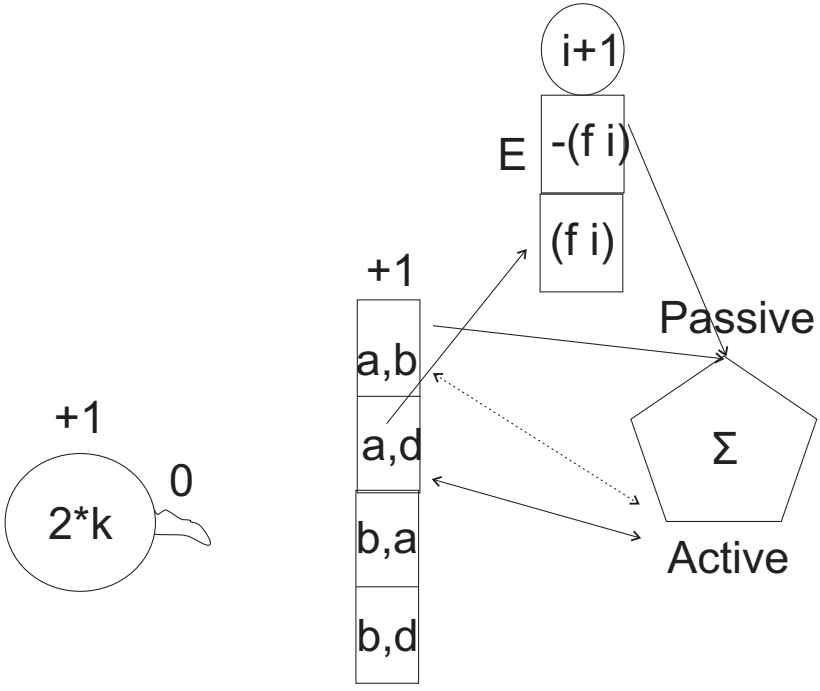

Fig. 4. Program for Fibonacci-based computation

Unfortunately using algebraic models and concepts demands deep knowledge of abstract algebra and category theory and completely another way of thinking then usual Turing-based algorithms and even then Lisp, Refal or Prolog.

Author wishes to thank prof. R. Glück for discussions during which the algebraic concept arose, PSI RAS for support an extremely non-conformist research, his followers A. Nepejvoda and V. Atamanov for their valuable developments in this completely unexplored domain.

References

1. Glushkov, V. M. Abstract Theory of Automata // Cleaver-Hume Press Ltd., 1963.
2. Maurer, W. D. A theory of computer instructions. Journal of the ACM, vol. 13, No 2 (1966) pp. 226–235.
3. Gluschkow W. M., Zeitlin G. E., Justchenko J. L. — Algebra. Sprachen. Programmierung. — Akademie-Verlag, Berlin 1980. — 340 p.
4. Ivanov, P. M. Algebraic modelling of complex systems. — Moscow, 1996. — 274 p.
5. Nepejvoda, N. N. Three-headed Dragon. https://docs.google.com/document /d/1hGzUB3p3j2zYcksoUnxtv7QamHzcgYVYr66iJJiMOhY
6. Alban Ponse and Mark B. van der Zwaag. An Introduction to Program and Thread Algebra. LNCS 3988, 2006, pp 445–488.
7. Bergstra J. A., Bethke I, Ponse A. Program Algebra and Thread Algebra. Amsterdam, 2006, 114p.
8. Nepejvoda N. N. *Reversivity, reversibility and retractability.* Third international Valentin Turchin workshop on metacomputation. Pereslavl: 2012. pp 203–215.
9. Girard J.-Y. *Linear Logic.* Theoretical Computer Science **50**, 1987, 102 pp.
10. Malcev, A.I. Algebraic Systems. Springer-Verlag, 1973, ISBN 0-387-05792-7.
11. Mitchell, J. C. Foundation for programming languages, MIT, 1996.
12. Barenfregt, H. The lambda-calculus. Its syntax and semantics. Elsevier 1984, ISBN 0-444-87508-5.
13. Nepejvoda, N. N. Reversive constructive logics. Logical Investigations, **15**, 150–169 (2008).
14. Axelsen, H. G., Glück, R. What do reversible programs compute? FOCSSACS 2011, LNCS 6604, pp. 42–56, 2011
15. http://www.muppetlabs.com/ breadbox/bf/
16. Böhm, C., Dezani-Ciancaglini, M. Can syntax be ignored during translation? In: Nivat M. (ed.) Automata, languages and programming. North-Holland, Amsterdam, 1972. p.197–207.
17. Nepejvoda, N. N. Abstract algebras of different classes of programs. Proceedings of the 3rd international conference on applicative computation systems (ACS'2012), 103–128.
18. Nepejvoda, N. N. Algebraic approach to control. Control Sciences, 2013, N 6, 2-14.