Formalization of the OOP Paradigm: INHERITANCE OF ABSTRACT AUTOMATA

A. G. Piskunov

April 11, 2010

ABSTRACT

In the paper some important terms of object - oriented programming (abstract data type, class, object, process, type, inheritance) are mathematically defined through cartesian products, relations, functions and automata. Proposed formalization of OOP notions significally differs from the formalization of Luca Cardelli, which could be considered as generally accepted.

Contents

1 INTRODUCTION

Following B. Mayer,

- by an abstract data type, we mean the quadruple: (type; function signatures; axioms connected the functions; preconditions of function execution) (see [1, Abstract data types. Formalizing the specification]);
- by a class, we mean an abstract data type (in what follows, ADT) equipped with a possibly partial implementation (see [1, The static structure: classes]), i.e., implemented in a programming language.

Since the definition of a class implicitly contains the notion of programming language, it should be formulated more rigorously in the following form: By a Λ-class, we mean an ADT equipped with a possibly partial implementation in the programming language Λ.

If, as the implementation language Λ, we take the language of formal specifications, which admits an implicit description of functions (i.e., with the help of signatures, axioms, and preconditions), then we should agree that the notion of ADT cannot be distinguished from the notion of Λ-class. For example, as such an language, we may use Z [3] (a method of mathematical notation adapted for requirements of programmers) or, furthermore, RSL [6] , which allows any style of programm writing from functional to imperative one. In [8] one can find an examples of algebraic design of ADT via RAISE developer method. In this method, one uses the implicit description of functions of ADT on early stages of designing; then, one passes to the imperative style of function writing in a subset of the RSL language such that it admits an automated translation of the RSL-code into C++ (or ADA). This implies that the notion of RSL-class is equivalent to the notion of C++-class (ADA-class).

Thus, we assume that ADT, as well as a class, are synonyms of the pair (type, set of functions). Then, in the paper, we pass from the pair (type, set of functions) to the triplet (set of states, next-state function, transition function), i.e., to an abstract automaton and specify inheritance relations on the sets of automata.

The inheritance condition for output function of the Mealy automaton, which is added in this paper, allows us to generalize in a natural way the definition of polymorphism introduced in [2] for the Moore automaton.

Iy is possible to look at formalization of Luca Cardelli in ([11] or http://lucacardelli.name/).

2 MAIN DEFINITIONS AND RESULTS

2.1 Main definitions

Semicomputable function

(see [12]) A function f is called semicomputable if there exists a program such that:

- it returns a value $f(x)$ if a value x from its definition domain dom f is given to the input;
- it never stops if a value which does not belong to its definition domain $x \notin$dom f is given to the input.

Mealy automaton

Assume that Q is a set of states, X is a set of input signals, Y is a set of output signals.

Then the triplet $(Q, \delta : Q \times X \to Q, \lambda : Q \times X \to Y)$ where δ is a next-state function (transition function) and λ is an output function, is referred to as an abstract Mealy automaton.

Diagonal product of mappings

Let mappings $g_1 : X \to Y_1$, ..., $g_k : X \to Y_k$ be given. Then, the mapping $(g_1$,..., $g_k)$: $X \to Y_1 \times ... \times Y_k$, specified by the condition $(g_1$,..., $g_k)(x) = (g_1(x)$, ..., $g_k(x))$ for any $x \in X$, is referred to as the diagonal product of mappings g_1, ..., g_k. The mappings g_1, ..., g_k are components of the diagonal product.

Projection

The mapping $\pi_{X_j} : X_1 \times ... \times X_k \to X_j$ specified by π_{X_j} $(x_1$, ..., $x_k) = x_j$, where $1 \leq j \leq k$ и $x_j \in X_j$ will be referred to as a projection.

2.2 Proposition

Let T, X, and Y be some sets. Then, any function $f : T \to X \times Y$ can be represented as the diagonal product of compositions of function f and its corresponding projections of the factors of $X \times Y$.

Proof. Suppose that f satisfies the condition $f(t) = (x, y)$, $t \in T$, $x \in X$, $y \in Y$. Then, by the definition of projection, we have $(x, y) = (\pi_X(x, y), \pi_Y(x, y)) = (\pi_X(f(t)), \pi_Y(f(t))) = (\pi_X \circ f(t), \pi_Y \circ f(t)) = (\pi_X \circ f, \pi_Y \circ f)(t) = f(t)$. Thus, f is the diagonal product of the functions $\pi_X \circ f$ и $\pi_Y \circ f$.

2.3 Corollary

Any (semicomputable) function h: $T \to Q \times Y$ may be derived from two (semicomputable) functions g and f, where g: $T \to Q$ and $f : T \to Y$.

Discussion. In the proof of Proposition 2.2 , we have used the simplest functions, namely, projections and elementary operations over the function, namely, composition and diagonal product (see [12, Partial recursive functions]). Therefore, if a function f is partially recursive, then the functions-components $f_s = \pi_Q \circ f$ и $f_o = \pi_Y \circ f$ are also partially recursive and, hence, semicomputable. Thus, we may assume that function f is derived from simpler functions-components. We will consider the semicomputability of a function as a synonym of the possibility to write this function in some programming language.

Remark

In the Rogers paper [9, Example: primitive recursive functions] there is primitive operation (iv) for the constructing of primitive recursive functions. This operation substitutes operations of composition and diagonal product used by Manin ([12, Example: primitive recursive functions]). This operation maps some functions f: $Y_1 \times ... \times Y_k \to Z$ and g_1: $T \to Y_1$, ..., g_k: $T \to Y_k$ to the function $\lambda\, x$: $T \bullet f(g_1(x), ... g_k(x))$.

As we can see:

- It in one partial case ($k = 1$) it is equivalent to definition of composition;

- In other partial case (f is identity function, $f(x) = x$) it is equivalent to definition of diagonal product.

The symbol λ here is symbol of λ -expression.

Next-state function and и output function

Let for some sets Q, X, Y (such that Y cannot be represented as the Cartesian product of Q with any other set), function $g : Q \times X \to Q$ will be referred to as a next-state function and function $f : Q \times X \to Y$ will be referred to as the Mealy output function (function $f : Q \to Y$ will be referred to as the Moore output function).

Remark

It seems that, in work [7, pp. 47, 95] a next-state function is called a generator, an output function is called an observer, and set Q – a type of interests.

2.4 Proposition

Let Q be a set,
I be a set of indices $\{1, ..., n\}$. Then, any set of functions
$\{ h_i : Q \times X_i \to Q \times Y_i | \ i \in I \}$ specifies a Mealy automaton.

Proof. First, in accordance with 2.3 , we construct two sets of functions
$G = \{ g_i : Q \times X_i \to Q \mid i \in I \}$
$F = \{ f_i : Q \times X_i \to Y_i | \ i \in I \}$
Then, by the first set of functions, we specify the next-state function
$\delta : Q \times I \times X_1 \times ... \times X_n \to Q$ as follows:

$\delta (q, i, x_1, ... , x_n) \equiv$
case i of
$1 \to g_1(q, x_1)$,
$2 \to g_2(q, x_2)$,
...
$n \to g_n(q, x_n)$
end

By the second set, we specify the output function $\lambda : Q \times I \times X_1 \times ... \times X_n \to Y_1 \times ... \times Y_n$ as follows:
λ (q, i, x_1, x_2, ..., x_n) is ($f_1(q, x_1)$, $f_2(q, x_2)$, ..., $f_n(q, x_n)$)

Denoting $I \times X_1 \times ... \times X_n$ by X, and $Y_1 \times ... \times Y_m$ – by Y,
we may see that, by construction, the obtained triplet (Q, δ, λ) satisfies the definition of the Mealy automaton.

Class, type

For any Mealy automaton (Q, δ, λ), the set of its states Q will be referred as the type. We consider the term an automaton as a synonym of the class term.
For an automaton A, we denote its type by $Q(A)$.

By virtue of Proposition 2.4 , by a class, we may also call the triplet (Q, $\{ g_i : Q \times X_i \to Q \mid i \in I \}$, $\{ f_j : Q \times X_j \to Y_j \mid j \in J \}$) where I, J are set of indices.

Object, process

Below, for any class $A = (Q, F_s, F_o)$

- we may use the notation of class A instead of the type of class $Q(A)$ in the signature of some function f, in particular, from sets $f \in F_s$ or $f \in F_o$; For example, we may write $f : A \times X \to Y$ instead $f : Q \times X \to Y$;
- the notation $a \in A$, instead of $a \in Q(A)$ is also accepted. This means that value a being a tuple as any element of $Q(A)$, may be used in no expressions either than functions with class A (anticipating, with a successor of class A) in the signature.

The concepts of object (process) are meaningless if the program is written in the applicative style. Maybe, it is meaningful to refer to a tuple a as $a \in A$ as to an object. If, in the set of next-state functions F_s, there is at least one function with the empty second factor, i.e., of the form $g : A \to A$, then the tuple a may be referred to as a process.

In the imperative style of a programming, for an abstract automaton A, by an object (process), we mean the variable declared by this automaton.

Note that, since the notion of type $Q(A)$ is different from that of class A, so we can speak about an object of class A and an object of type $Q(A)$.

Associativity of the Cartesian product

Note that (see, for example, [4, Natural join, p. 153] the operation of joining and, hence, Cartesian product, is associative, that is
$(A \times B) \times C = A \times (B \times C) = A \times B \times C$
The associativity of the Cartesian product means that, for $X = A \times B$, function $f : X \to T$, may be treated as a function with the signature f: $A \times B \to T$.

Lambda expressions

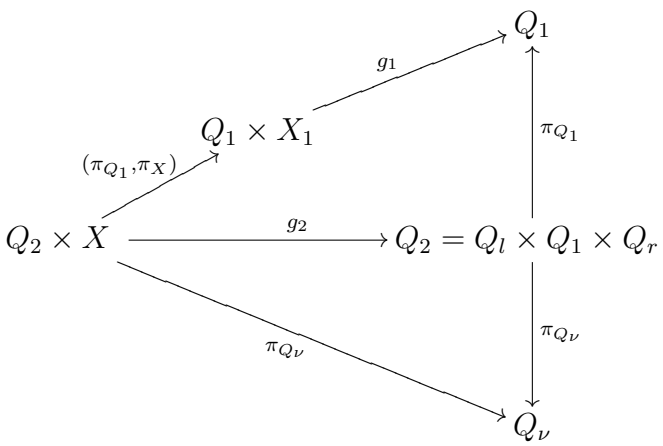
Below, we use the symbol λ to write a function expression. An expression
$\lambda\ val_1$: T_1,..., val_n : T_n • expression(val_1, ..., val_n)
representing a function $T_1 \times .... \times T_n \to T$, where T is the type of expression(val_1, ..., val_n).
For example, the expression $\lambda\ x$: Int, y : Int • $x + y$ determines the function of addition of two integers with the signature Int × Int → Int.

3 CONDITIONS OF FUNCTION INHERITANCE

Suppose that some sets Q_l, Q_1, Q_r, Q_2, connected by the relation $Q_2 = Q_l \times Q_1 \times Q_r$. Let $q_1 \in Q_1$, $q_l \in Q_l$, $q_r \in Q_r$.

Inheritance of next-state functions

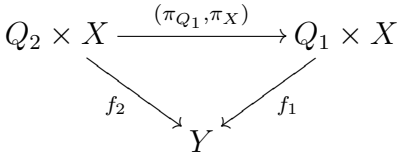
A function $g_2 : Q_2 \times X \to Q_2$ is in the inheritance relation with a function $g_1 : Q_1 \times X \to Q_1$ if
$g_2 = \lambda\ (q_l, q_1, q_r, x) : Q_2 \times X$ • $(q_l, g_1(q_1,x), q_r)$, where $x \in X$,
i.e., function g_2 is the diagonal product of functions of the form
$g_2 = (\pi_{Q_l}, g_1 \circ (\pi_{Q_1}, \pi_X), \pi_{Q_r})$



inheritance of next-state functions, $\nu \in \{l, r\}$

Inheritance of the Mealy output functions

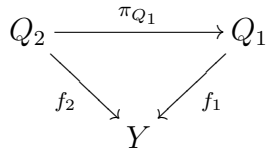
A function $f_2 : Q_2 \times X \to Y$ is in the inheritance relation with a function $f_1 : Q_1 \times X \to Y$ if
$f_2 = \lambda\ (q_l, q_1, q_r, x) : Q_2 \times X \bullet f_1(q_1,x)$, where $x \in X$,
i.e., function f_2 is a composition of functions of the form $f_2 = f_1 \circ (\ \pi_{Q_1}, \pi_X\ )$



inheritance of the Mealy output functions

Inheritance of the Moore output functions

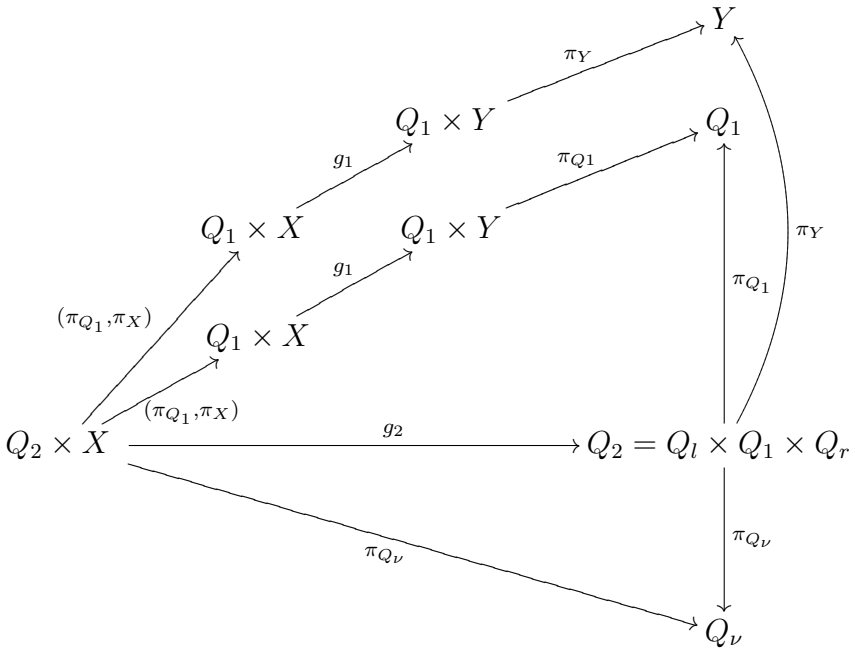
A function $f_2 : Q_2 \to Y$ is in the inheritance relation with a function $f_1 : Q_1 \to Y$ if
$f_2 = \lambda\ (q_l, q_1, q_r) : Q_2 \bullet f_1(q_1)$,
i.e., function f_2 is a composition of functions of the form $f_2 = f_1 \circ \pi_{Q_1}$



Inheritance of the Moore output functions

Inheritance of functions of the general form

A function $g_2 : Q_2 \times X \to Q_2 \times Y$ is in the inheritance relation with a function $g_1 : Q_1 \times X \to Q_1 \times Y$ if $g_2 =$
$\lambda\ (q_l, q_1, q_r, x) : Q_2 \times X \bullet (q_l, \pi_{Q_1}(g_1(q_1,x)), q_r, \pi_Y(g_1(q_1,x)))$,
where $x \in X$, i.e., function g_2 is the diagonal product of functions of the form
$g_2 = (\pi_{Q_l}, \pi_{Q_1} \circ g_1 \circ (\ \pi_{Q_1}, \pi_X\ ), \pi_{Q_r}, \pi_Y \circ g_1 \circ (\ \pi_{Q_1}, \pi_X\ )\ )$



inheritance of functions of the general form, $\nu \in \{l, r\}$

Remark

In the definition of inheritance of functions of the general form, we use different projection functions $\pi_{Q_1} : Q_1 \times X \to Q_1$ and $\pi_{Q_1} : Q_1 \times Y \to Q_1$

Remark 2

The discussion of the inheritance of functions of the general form is beyond the scope of this paper due to Proposition 2.2 . We gave the definition in order to show that a general definition of inheritance exists.

Visible and true signatures of functions

Let an object q and a function f belong to an automaton A. By virtue of programmers' tradition to record expression $f(q, x)$ as $q.f(x)$ the signature $f\colon Q \times X \to Q \times Y$ we'll be referred to as to a true signature. The signature of the same function written in the brief form $f\colon X \to Y$ will be referred to as a visible signature.

4 INHERITANCE OF ABSTRACT MEALY AUTOMATA

Suppose that we have an automaton $B =$
$(Q_2, \{ g_{2,i} : Q_2 \times X_{2,i} \to Q_2 \mid i \leq n_2 \}, \{ f_{2,j} : Q_2 \to Y_{2,j} \mid j \leq m_2 \})$
and an automaton $A =$
$(Q_1, \{ g_{1,i} : Q_1 \times X_{1,i} \to Q_1 \mid i \leq n_1 \}, \{ f_{1,j} : Q_1 \to Y_{1,j} \mid j \leq m_1 \})$.

We say that automaton B is in the inheritance relation with automaton A (inherits automaton A) (Class B inherits class A), if

- Q_2 is a Cartesian product $Q_l \times Q_1 \times Q_r$, where Q_l and/or Q_r may be empty sets;
- there exists a partial injection $\tau : \{i \mid i \leq n_1\} \rightsquigarrow \{i \mid i \leq n_2 \}$ such that, for any $i \leq n_1$, the next-state functions $g_{2,\tau(i)}$ and $g_{1,i}$ are in the inheritance relation;

- there exists a partial injection $\rho : \{i \mid \ i \leq m_1\} \rightsquigarrow \{i \mid \ i \leq m_2 \}$ such that the output functions $f_{2,\rho(i)}$ и $f_{1,i}$, $i \leq m_1$, are in the inheritance relation.

Remark

Function $f : X \to Y$ is injection if f maps different elements of the domain X to different elements of the rang Y, that is $f(x_1) = f(x_2)$ implies $x_1 = x_2$.

Automaton A be referred to as a parent and automaton B will be called child.

Then, for each pair of the parent automaton A with type Q_1 and child automaton B with type Q_2, the inheritance conditions for next-state and output functions allow one to specify the inheritance operators (ς similar to S and θ similar to O):

- for any next-state function g from A, by $\varsigma_{B:A}(g)$, we denote the function $(\pi_{Q_l}, g \circ (\pi_{Q_1}, \pi_X), \pi_{Q_r})$;
- for any function g of the general form from A, by $\varsigma_{B:A}(g)$ we denote the function $(\pi_{Q_l}, \pi_{Q_1} \circ g \circ (\pi_{Q_1}, \pi_X), \pi_{Q_r}, \pi_Y \circ g \circ (\pi_{Q_1}, \pi_X))$;
- similarly, for any output function f from A, $\theta_{B:A}(f) = f \circ (\pi_{Q_1}, \pi_X)$ (the Mealy output function), $\theta_{B:A}(f) = f \circ \pi_{Q_1}$ (the Moore output function).

We will omit one subscript or both subscripts $B{:}A$ if it is clear what automata are discussed.

5 CONSTRUCTION OF THE CHILD AUTOMATON FOR A SINGLE INHERITANCE

Single inheritance (see [10]) occurs when a subclass (child class) inherits attributes from only one class. We define the following procedure for constructing the child automaton $B = \{ Q_2, G_2, F_2 \}$ by the parent automaton $A = \{ Q_1, G_1, F_1\}$:

- We specify a factor Q_r (maybe empty) by enumerating (as programmers use to do) types of its components to the right from automaton A;
- Let for any next - state function g of automaton A there is its image $\varsigma_{B:A}(g)$ in the set of next-state functions of automaton B:
  $G_2 \supseteq \{\ B.A.g \mid B.A.g : Q_2 \times X_g \to Q_2 \bullet B.A.g = \varsigma_{B:A}(g) \wedge g \in G_1\ \}$
- Similarly, let for any output function f of automaton A there is its image $\theta_{B:A}(f)$ in the set of output functions of B:
  $F_2 \supseteq \{\ B.A.f \mid B.A.f : Q_2 \times X_f \to Y_f \bullet B.A.f = \theta_{B:A}(f) \wedge f \in F_1\ \}$.

If, for a function denoted by $B.A.h$, sets G_2 or F_2 contain no other function denoted by h with the same signature, then we may omit references to automata $B.A$ in the notation $B.A.h$ or explicitly give a synonym as one uses to do in SQL. For example, $\theta_{B:A}\ (f)$ as f_1.

The automaton B constructed in this way is, by construction, in the inheritance relation with automaton A. Note that this procedure allows one to construct only partial cases of pairs of automata in the inheritance relation.

5.1 Remarks about the Syntax

For access to an attribute x of the class, i.e., in our terms, to an element of a type T of some proper tuple, the next-state function x: $Q \times T \to Q$ and the output function x: $Q \to T$ are used. In the notation system from [5, Variant Definition], these functions may be written in the form $x : T \leftrightarrow x$. The first occurrence of x (namely, x: T) denotes the output function $x : Q \to T$, the second occurrence ($T \leftrightarrow x$) denotes the next-state $x : Q \times T \to Q$. In that case it is not necessary to mention functions such as x in the next - state fuctions set and the output functions set. The other functions of the automaton must be explicitly defined indicating the visible signature.

Consider an example of a class in terms of the C++ language:

```
class A { public: int a; public void g (int b) { a*=b;}  };
```

This class may be written, for example, as follows:

$A = (\ a$: Int $\leftrightarrow a, \{\ g = \lambda\ b$: Int $\bullet\ a(\text{this}, a(\text{this})\ *\ b)\ \}, \emptyset\ )$

where the full form of the λ expression defined function g must be

$g = \lambda$ this: Int, b: Int $\bullet$ a(this, a(this) * b).

In the following example,

```
class B : A { public: float x;}
```

the inheritance operator may look like this: $B = A \times (x$:Float $\leftrightarrow x, \emptyset, \emptyset)$.

6 MULTIPLE INHERITANCE

Suppose that we have automata $A = (Q(A), G_A, F_A)$, $B = (Q(B), G_B, F_B)$, C, such that automata C and A are in the inheritance relation and automata C and B are in the inheritance relation. Then, automata C and A, B are in the relation of multiple inheritance if

- the sets of states $Q(A)$ и $Q(B)$ are direct factors of the set $Q(C)$;
- for any pair $g_A \in G_A$ and $g_B \in G_B$ holds $\varsigma_{C:A}(g_A) \neq \varsigma_{C:B}(g_B)$;
- for any pair $f_A \in F_A$ and $f_B \in F_B$ holds $\theta_{C:A}(f_A) \neq \theta_{C:B}(f_B)$.

It is rather simple to generalize the definition of multiple inheritance from two parents to an arbitrary number of them.

6.1 Remarks about the Syntax

Consider an example of multiple inheritance in terms of the C++ language.

```
class A        { public: int a; };
class B        { public: int a; };
class C : public A, public B {        } ;
```

In our terms, it is written as follows:

$A =$ (a: Int $\leftrightarrow a, \emptyset , \emptyset$)

$B =$ (a: Int $\leftrightarrow a, \emptyset , \emptyset$)

$C = A \times B$

The notation a $((x,y), z)$ for some x, y, z :Int means usage of the next - state function a: (Int $\times$ Int) $\times$ Int $\rightarrow$ (Int $\times$ Int) inherited from class A (as the

first one met from the left) is applied. So, we will assume that it is the brief form of the $C.A.a((x, y), z)$ notation. To apply the second function, we must either specify it explicitly $C.B.a((x, y), z)$ or declare a synonym, for instance, in the form

$C = A \times B$ (a as $a1$)

It also seems rational to introduce synonyms for classes. Then, for the C++ text

```
class A        { public: int a; };
class C : public A, public A {        } ;
```

an analogue could be the following:
$A =$ (a: Int $\leftrightarrow a$, $\emptyset$, $\emptyset$)
$C = A \times A$ as B (a as $a1$).

Index

References

[1] Bertrand Meyer, Object-Oriented Software Construction, Second Edition, Prentice Hall Professional Technical Reference, ISE Inc., Santa Barbara, USA, 1997.

[2] A.G. Piskunov. Formalization of the object-oriented programming paradigm (In Russian), 2007. http://www.realcoding.net/article/view/4570.

[3] Jonathan Bowen. Formal Specification and Documentation using Z: A Case Study Approach, 2003. http://www.jpbowen.com/pub/zbook.pdf.

[4] C.J.Date. An Introduction to Database Systems, Sixth edition, 1995. Addison-Wesley Publishing Company, New York, 1995.

[5] Chris George. Introduction to RAISE, 2002. http://users.iptelecom.net.ua/~agp1/arts/RAISE4.pdf.

[6] Chris George. Introduction to RAISE. UNU-IIST report No. 249, 2002. ftp://www.iist.unu.edu/pub/techreports/report249.pdf.

[7] Chris George. Introduction to RAISE. UNU-IIST report No. 249, 2002. http://users.iptelecom.net.ua/~agp1/arts/report249.pdf.

[8] The RAISE Method Group. The RAISE Development Method, 1999. ftp://www.iist.unu.edu/pub/RAISE/method_book/book.zip.

[9] H. Rogers, Jr. Theory of recursive functions and effective computability. (McGraw-Hill Book Company, 1967).

[10] Jamie Shield. Towards an Object-Oriented Refinement Calculus, 2001. Thesis.

[11] Luca Cardelli. A Semantics of Multiple Inheritance, 1988. Information and Computation 76, 138-164,1988.

[12] Yu.I. Manin. Computable and incomputable (in russian), 1980. http://agp1.nm.ru/arts/manin2.html.