On a Method of Verification of Functional Programs

Andrew M. Mironov

Moscow State University

amironov66@gmail.com

Abstract. In this paper the problem of verification of functional programs (FPs) over strings is considered, where specifications of properties of FPs are defined by other FPs, and a FP Σ_1 meets a specification defined by another FP Σ_2 iff a composition of functions defined by the FPs Σ_1 and Σ_2 is equal to the constant 1. We introduce a concept of a state diagram of a FP, and reduce the verification problem to the problem of an analysis of the state diagrams of FPs. The proposed approach is illustrated by the example of verification of a sorting program.

Keywords: functional program, state diagram, verification

1 Introduction

The problem of program verification is one of the main problems of theoretical computer science. For various classes of programs there are used various verification methods. For example, for a verification of sequential programs there are used Floyd's inductive assertions method [1], Hoare logic [2], etc. For verification of parallel and distributed programs there are used methods based on a calculus of communicating systems (CCS) and π-calculus [3], [4], a theory of communicating sequential processes (CSP) and its generalizations [5], [6], temporal logic and model checking [7], process algebra [8], Petri nets [9], etc.

Main methods of verification of functional programs (FPs) are computational induction and structural induction [10]. Disadvantages of these methods are related to difficulties to construct formal proofs of program correctness. Among other methods of verification of FPs it should be noted a method based on reasoning with datatypes and abstract interpretation through type inference [12], a model checking method to verify FPs [13], [14], methods based on flow analysis [11] methods based on the concept of a multiparametric tree transducer [15].

In this article we consider FPs as systems of algebraic equations over strings. We introduce a concept of a state diagram for such FPs and present the verification method based on the state diagrams. The main advantages of our approach in comparison with all the above approaches to verification of FPs are related to the fact that our approach allows to present proofs of correctness of FPs in the form of simple properties of their state diagrams.

The basic idea of our approach is the following. We assume that a specification of properties of a FP under verification Σ_1 is defined by another FP Σ_2, whose input is equal to the output of Σ_1, i.e. we consider FP $\Sigma_1 \circ \Sigma_2$, which is a composition Σ_1 and Σ_2. We say that a FP Σ_1 is correct with respect to the specification Σ_2 iff the input-output map $f_{\Sigma_1 \circ \Sigma_2}$, which corresponds to the FP $\Sigma_1 \circ \Sigma_2$ (i.e. $f_{\Sigma_1 \circ \Sigma_2}$ is a composition of the input-output maps corresponded to Σ_1 and Σ_2) has an output value 1 on all its input values. We reduce the problem of a proving the statement $f_{\Sigma_1 \circ \Sigma_2} = 1$ to the problem of an analysis of a state diagram for the FP $\Sigma_1 \circ \Sigma_2$.

The proposed method of verification of FPs is illustrated by an example of verification of a sorting FP. At first, we present a complete proof of correctness of this FP by structural induction. This is done for a comparison of the complexity of a manual verification of the FP on the base of the structural induction method, and the complexity of the proposed method of automatic verification of FPs. At second, we present a correctness proof of the FP by the method based on constructing its state diagram. The proof by the second method is significantly shorter, and moreover, it can be generated automatically. This demonstrates the benefits of the proposed method of verification of FPs in comparison with the manual verification based on the structural induction method.

2 Main concepts

2.1 Terms

We assume that there are given sets

- $\mathcal{D}$ of **values**, which is the union $\mathcal{D}_{\mathbf{C}} \cup \mathcal{D}_{\mathbf{S}}$, where
  - elements of $\mathcal{D}_{\mathbf{C}}$ are called **symbols**, and
  - elements of $\mathcal{D}_{\mathbf{S}}$ are called **symbolic strings** (or briefly **strings**), and each string from $\mathcal{D}_{\mathbf{S}}$ is a finite (maybe empty) sequence of elements of $\mathcal{D}_{\mathbf{C}}$,
- $\mathcal{X}$ of **data variables** (or briefly **variables**)
- $\mathcal{C}$ of **constants**,
- $\mathcal{F}$ of **functional symbols (FSs)**, and
- Φ of **functional variables**

where each element m of any of the above sets is associated with a **type** designated by the notation $type(m)$, and

- if $m \in \mathcal{D} \cup \mathcal{X} \cup \mathcal{C}$, then $type(m) \in \{\mathbf{C}, \mathbf{S}\}$,
- if $m \in \mathcal{F} \cup \Phi$, then $type(m)$ is a notation of the form $t_1 \times \ldots \times t_n \to t$, where $t_1, \ldots, t_n, t \in \{\mathbf{C}, \mathbf{S}\}$.

If $d \in \mathcal{D}_{\mathbf{C}}$, then $type(d) = \mathbf{C}$, and if $d \in \mathcal{D}_{\mathbf{S}}$, then $type(d) = \mathbf{S}$.

Each constant $c \in \mathcal{C}$ corresponds to an element of $\mathcal{D}_{type(c)}$, called a **value** of this constant. The notation ε denotes a constant of the type $\mathbf{S}$, whose value is an empty string. We assume that ε is the only constant of the type $\mathbf{S}$.

Each FS $f \in \mathcal{F}$ corresponds to a partial function of the form $\mathcal{D}_{t_1} \times \ldots \times \mathcal{D}_{t_n} \to \mathcal{D}_t$, where

$$type(f) = t_1 \times \ldots \times t_n \to t.$$

This function is denoted by the same symbol f.

Below we list some of the FSs which belong to $\mathcal{F}$. Beside each FS we point out (with a colon) its type.

1. $head : \mathbf{S} \to \mathbf{C}$. The function *head* is defined for non-empty string, it maps each non-empty string to its first element.
2. $tail : \mathbf{S} \to \mathbf{S}$. The function *tail* is defined for non-empty string, it maps each non-empty string u to a string (called a **tail** of u) derived from u by removal of its first element.
3. $conc : \mathbf{C} \times \mathbf{S} \to \mathbf{S}$. For each pair $(a, u) \in \mathcal{D}_{\mathbf{C}} \times \mathcal{D}_{\mathbf{S}}$ the string $conc(a, u)$ is obtained from u by adding the symbol a before.
4. $empty : \mathbf{S} \to \mathbf{C}$. Function *empty* maps empty string to the symbol 1, and each non-empty string to the symbol 0.
5. $=: \mathbf{C} \times \mathbf{C} \to \mathbf{C}$. The value of the function $=$ on the pair (u, v) is equal to 1 if $u = v$, and 0 otherwise.
6. $\leq: \mathbf{C} \times \mathbf{C} \to \mathbf{C}$. We assume that $\mathcal{D}_{\mathbf{C}}$ is linearly ordered set, and the value of the function $\leq$ on the pair (u, v) is equal to 1 if $u \leq v$, and 0 otherwise.
7. Boolean FSs: $\neg : \mathbf{C} \to \mathbf{C}$, $\wedge : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$, etc., corresponding functions are standard boolean functions on the arguments 0 and 1 (i.e. $\neg(1) = 0$,, etc.) and are not defined on other arguments.
8. *if_then_else* $: \mathbf{C} \times t \times t \to t$, where $t = \mathbf{C}$ or $\mathbf{S}$ (i.e. the notation *if_then_else* denotes two FSs), and functions corresponding to both FSs are defined by the same way:

$$\textit{if\_then\_else}\ (a, u, v) \stackrel{\text{def}}{=} \begin{cases} u, \text{ if } a = 1 \\ v, \text{ otherwise.} \end{cases}$$

A concept of a **term** is defined inductively. Each term e is associated with a certain type $type(e) \in \{\mathbf{C}, \mathbf{S}\}$. Each data variable and each constant is a term, a type of which is the same as the type of this variable or constant. If $e_1, \ldots, e_n$ is a list of terms and g is a FS or a functional variable such that

$$type(g) = \ type(e_1) \times \ldots \times type(e_n) \to t$$

then the notation $g(e_1, \ldots, e_n)$ is a term of the type t.

We shall denote terms

$$\begin{array}{l} head(e),\ tail(e),\ conc(e_1, e_2),\ empty(e), \\ = (e_1, e_2),\ \leq (e_1, e_2),\ \textit{if\_then\_else}\ (e_1, e_2, e_3) \end{array}$$

in the form

$$e_h,\ e_t,\ e_1 e_2,\ [\![e = \varepsilon]\!],\ [\![e_1 = e_2]\!],\ [\![e_1 \leq e_2]\!],\ [\![e_1]\!]\, e_2 : e_3$$

respectively. Terms containing boolean FSs will be denoted as in mathematical texts (i.e. in the form $e_1 \wedge e_2$, etc.). Terms of the form $e_1 \wedge \ldots \wedge e_n$ can also be denoted as $[\![e_1, \ldots, e_n]\!]$.

2.2 A concept of a functional program over strings

A **functional program over strings** (referred below as a **functional program (FP)**) is a set Σ of functional equations of the form

$$\begin{cases} \varphi_1(x_{11}, \ldots, x_{1n_1}) = e_1 \\ \ldots \\ \varphi_m(x_{m1}, \ldots, x_{mn_m}) = e_m \end{cases} \tag{1}$$

where $\varphi_1, \ldots, \varphi_m$ are distinct functional variables, and for each $i = 1, \ldots, m$ $\varphi_i(x_{i1}, \ldots, x_{in_i})$ and e_i are terms of the same type, such that

$$X_{e_i} = \{x_{i1}, \ldots, x_{in_i}\}, \quad \Phi_{e_i} \subseteq \{\varphi_1, \ldots, \varphi_m\}$$

(where X_e and Φ_e are sets of all data variables and functional variables respectively occurred in the term e). We shall use the notation Φ_Σ for the set of all functional variables occurred in Σ.

FP (1) specifies a list

$$(f_{\varphi_1}, \ldots, f_{\varphi_m}) \tag{2}$$

of functions corresponding to the functional variables from Φ_Σ, which is the least (in the sense of an order on lists of partial functions, described in [10]) solution of (1) (this list is called a **least fixed point (LFP)** of the FP (1), all details related to the concept of a LFP can be found in chapter 5 of the book [10]). Values of these functions can be calculated by a standard recursion. We assume that for each FP under consideration all components of its LFP are total functions. First function in the list (2) (i.e. f_{φ_1}) is denoted by f_Σ, and is called a **function corresponding to** Σ. If Σ has the form (1), then $type(\Sigma)$ denotes the type $type(e_1)$.

3 Example of specification and verification of a FP

3.1 Example of a FP

Consider the following FP:

$$\begin{aligned} \mathbf{sort}(x) &= [\![x = \varepsilon]\!]\, \varepsilon : \mathbf{insert}(x_h, \mathbf{sort}(x_t)) \\ \mathbf{insert}(a, y) &= [\![y = \varepsilon]\!]\, a\varepsilon \\ &\qquad : [\![a \leq y_h]\!]\, ay \\ &\qquad\qquad : y_h\ \mathbf{insert}(a, y_t) \end{aligned} \tag{3}$$

This FP defines a function of string sorting. The FP consists of two equations, which define the following functions:

- **sort** : $\mathbf{S} \to \mathbf{S}$ is a main function, and
- **insert** : $\mathbf{C} \times \mathbf{S} \to \mathbf{S}$ is an auxiliary function, which maps a pair $(a, y) \in \mathbf{C} \times \mathbf{S}$ to the string derived by an insertion of the symbol a to the string y, with the following property: if the string y is ordered, then the string $\mathbf{insert}(a, y)$ also is ordered.
  (we say that a string is ordered, if its components form a nondecreasing sequence).

3.2 Example of a specification of a FP

One of correctness properties of FP (3) it the following: $\forall\, x \in \mathbf{S}$ the string $\mathbf{sort}(x)$ is ordered. This property can be described formally as follows. Consider a FP defining a function **ord** of string ordering checking:

$$\begin{aligned}\mathbf{ord}(x) = [\![x = \varepsilon]\!]\ 1 \\ : [\![x_t = \varepsilon]\!]\ 1 \\ : [\![x_h \le (x_t)_h]\!]\ \mathbf{ord}(x_t) : 0\end{aligned} \tag{4}$$

The function **ord** allows to describe the above property of correctness as the following mathematical statement:

$$\forall\, x \in \mathbf{S} \quad \mathbf{ord}(\mathbf{sort}(x)) = 1 \tag{5}$$

3.3 Example of a verification of a FP

The problem of verification of the correctness property of FP (3) consists of a formal proof of (5). This proposition can be proved like an ordinary mathematical theorem, for example using the method of mathematical induction. For example, a proof of this proposition can be the following.

If $x = \varepsilon$, then, according to first equation of system (3), the equality $\mathbf{sort}(x) = \varepsilon$ holds, and therefore

$$\mathbf{ord}(\mathbf{sort}(x)) = \mathbf{ord}(\varepsilon) = 1.$$

Let $x \neq \varepsilon$. We prove (5) for this case by induction. Assume that for each string y, which is shorter than x, the equality

$$\mathbf{ord}(\mathbf{sort}(y)) = 1$$

holds. Prove that this implies the equality

$$\mathbf{ord}(\mathbf{sort}(x)) = 1. \tag{6}$$

(6) is equivalent to the equality

$$\mathbf{ord}(\ \mathbf{insert}(x_h, \mathbf{sort}(x_t))) = 1 \tag{7}$$

By the induction hypothesis, the equality

$$\mathbf{ord}(\mathbf{sort}(x_t)) = 1$$

holds, and this implies (7) on the reason of the following lemma.

Lemma.
The following implication holds:

$$\mathbf{ord}(y) = 1 \quad \Rightarrow \quad \mathbf{ord}(\mathbf{insert}(a, y)) = 1 \tag{8}$$

Proof.

We prove the lemma by induction on the length of y.

If $y = \varepsilon$, then the right side of(8) has the form

$$\mathbf{ord}(a\varepsilon) = 1$$

which is true by definition **ord**.

Let $y \neq \varepsilon$, and for each string z, which is shorter than y, the following implication holds:

$$\mathbf{ord}(z) = 1 \quad \Rightarrow \quad \mathbf{ord}(\mathbf{insert}(a, z)) = 1 \tag{9}$$

Let $c \stackrel{\text{def}}{=} y_h$, $d \stackrel{\text{def}}{=} y_t$.

(8) has the form

$$\mathbf{ord}(cd) = 1 \quad \Rightarrow \quad \mathbf{ord}(\mathbf{insert}(a, cd)) = 1 \tag{10}$$

To prove the implication (10) it is necessary to prove that if $\mathbf{ord}(cd) = 1$, then the following implications hold:

(a) $a \leq c \quad \Rightarrow \quad \mathbf{ord}(a(cd)) = 1$,
(b) $c < a \quad \Rightarrow \quad \mathbf{ord}(c\,\mathbf{insert}(a, d)) = 1$.

(a) holds because $a \leq c$ implies

$$\mathbf{ord}(a(cd)) = \mathbf{ord}(cd) = 1.$$

Let us prove (b).

- $d = \varepsilon$. In this case, right side of (b) has the form

$$\mathbf{ord}(c(a\varepsilon)) = 1 \tag{11}$$

(11) follows from $c < a$.

- $d \neq \varepsilon$. Let $p \stackrel{\text{def}}{=} d_h$, $q \stackrel{\text{def}}{=} d_t$.

In this case, it is necessary to prove that if $c < a$, then

$$\mathbf{ord}(c\,\mathbf{insert}(a, pq)) = 1 \tag{12}$$

1. if $a \leq p$, then (12) has the form

$$\mathbf{ord}(c(a(pq))) = 1 \tag{13}$$

Since $c < a \leq p$, then (13) follows from the equalities

$$\mathbf{ord}(c(a(pq))) = \mathbf{ord}(a(pq)) = \mathbf{ord}(pq) =$$
$$= \mathbf{ord}(c(pq)) = \mathbf{ord}(cd) = 1$$

2. if $p < a$, then (12) has the form

$$\mathbf{ord}(c(p\ \mathbf{insert}(a, q))) = 1 \tag{14}$$

Since, by assumption,

$$\mathbf{ord}(cd) = \mathbf{ord}(c(pq)) = 1$$

then $c \leq p$, and therefore (14) can be rewritten as

$$\mathbf{ord}(p\ \mathbf{insert}(a, q)) = 1 \tag{15}$$

If $p < a$, then

$$\mathbf{insert}(a, d) = \mathbf{insert}(a, pq) = p\ \mathbf{insert}(a, q)$$

therefore (15) can be rewritten as

$$\mathbf{ord}(\mathbf{insert}(a, d)) = 1 \tag{16}$$

(16) follows by the induction hypothesis for the Lemma (i.e., from the implication (9), where $z \stackrel{\text{def}}{=} d$) from the equality

$$\mathbf{ord}(d) = 1$$

which is justified by the chain of equalities

$$\begin{aligned} &1 = \mathbf{ord}(cd) = \mathbf{ord}(c(pq)) = \quad (\text{since } c \leq p) \\ &= \mathbf{ord}(pq) = \mathbf{ord}(d). \quad \blacksquare \end{aligned}$$

From the above example we can see that even for the simplest FP, which consists of several lines, a proof of its correctness is not trivial mathematical reasoning, it is difficult to check it and much more difficult to construct it.

Below we present a radically different method for verification of FPs based on a construction of state diagrams for FPs, and illustrate it by a proof of (5) on the base of this method. This proof can be generated automatically, that is an evidence of advantages of the method for verification of FPs based on state diagrams.

4 State diagrams of functional programs

4.1 Concepts and notations related to terms

The following notations and concepts will be used below.

- $\mathcal{E}$ is a set of all terms.
- $\mathcal{E}_0$ is a set of all terms not containing functional variables.
- $\mathcal{E}_{conc}$ is a set of terms $e \in \mathcal{E}_0$, such that each FS occurring in e is *conc*.

- If Σ is a FP, then $\mathcal{E}_\Sigma$ is a set of terms, each of which is either a variable or has the form $\varphi(u_1, \ldots, u_n)$, where $\varphi \in \Phi_\Sigma$ and $u_1, \ldots, u_n \in \mathcal{E}_{conc}$.
- If $e \in \mathcal{E}$, $x_1, \ldots, x_n$ is a list of the different variables, and $e_1, \ldots, e_n$ are terms such that $\forall\, i = 1, \ldots, n$ $type(e_i) = type(x_i)$, then the notation

$$e(e_1/x_1, \ldots, e_n/x_n) \tag{17}$$

  denotes a term derived from e by replacement $\forall\, i \in \{1, \ldots, n\}$ of all occurrences of x_i in e with the term e_i.
- If e and e' are terms, then for each term e'', such that $type(e'') = type(e')$, the notation $e(e''/e')$ denotes a term derived from e by a replacement of all occurrences of e' in e with the term e''.
- An **assignment** is a notation of the form

$$u := e \tag{18}$$

  where $u \in \mathcal{E}_{conc}$, $e \in \mathcal{E}_\Sigma$, $type(u) = type(e)$.
- If $X \subseteq \mathcal{X}$, then an **evaluation** of variables occurring in X is a function ξ, which maps each variable $x \in X$ to a value $x^\xi \in \mathcal{D}_{type(x)}$. The set of all evaluations of variables occurring in X will be denoted by $X^\bullet$.
- For each $e \in \mathcal{E}_0$, each $X \supseteq X_e$ and each $\xi \in X^\bullet$ the notation e^ξ denotes an object called a **value** of e on ξ and defined by a standard way (i.e. if $e \in \mathcal{C}$, then e^ξ is equal to the value of the constant e, if $e \in \mathcal{X}$, then e^ξ is equal to the value of the evaluation ξ on the variable e, and if $e = f(e_1, \ldots, e_n)$, then $e^\xi = f(e_1^\xi, \ldots, e_n^\xi)$).
- We shall consider terms $e_1, e_2 \in \mathcal{E}_0$ as equal iff for each $\xi \in (X_{e_1} \cup X_{e_2})^\bullet$ the equality $e_1^\xi = e_2^\xi$ holds. We understand this equality in the following sense: values e_1^ξ and e_2^ξ either both undefined, or both defined and coincide.
- A term $e \in \mathcal{E}_0$ is called a **formula**, if all variables from X_e are of the type $\mathbf{C}$, and $\forall\, \xi \in X_e^\bullet \quad e^\xi \in \{0, 1\}$. The symbol $\mathcal{B}$ denotes the set of all formulas. The symbols $\top$ and $\bot$ denote formulas taking the values 1 and 0 respectively on each evaluation of their variables.

4.2 A concept of a state of a FP

Let Σ be a FP.

A **state** of Σ is a notation s of the form

$$[\![b]\!]\, u\, (\theta_1, \ldots, \theta_m) \tag{19}$$

components of which are the following:

- b is a formula from $\mathcal{B}$, called a **condition** of s,
- u is a term from $\mathcal{E}_{conc}$, called a **term related to** s, and
- $\theta_1, \ldots, \theta_m$ are assignments.

We shall use the following notations.

- S_Σ is the set of all states of Σ.
- If a state $s \in S_\Sigma$ is of the form (19), then we shall denote by b_s, u_s, Θ_s and $type(s)$ a formula b, a term u, a sequence of assignments (which can be empty) in (19), and a type $type(u)$, respectively.
  If $b_s = \top$, then the formula b in (19) will be omitted.
- If $s \in S_\Sigma$, then
  - X_s is a set of all data variables occurring in s,
  - each variable from X_s, occurring in the left side of an assignment from Θ_s, is called an **internal variable** of s, all other variables from X_s are called **input variables** of s,
  - $s^\bullet$ is a set of all $\xi \in X_s^\bullet$, such that $b_s^\xi = 1$, and $\forall\,(u_i := e_i) \in \Theta_s$
    - if $e_i \in \mathcal{E}_{conc}$, then $u_i^\xi = e_i^\xi$, and
    - if $e_i = \varphi(v_1, \ldots, v_n)$, then

$$u_i^\xi = f_\varphi(v_1^\xi, \ldots, v_n^\xi),$$

where f_φ is a corresponding component of a LFP of Σ.

A state $s \in S_\Sigma$ is said to be **terminal**, if Θ_s does not contain functional variables.

Given a pair of states $s_1, s_2 \in S_\Sigma$. We denote by the notation $s_1 \subseteq s_2$ the following statement: sets of input variables s_1 and s_2 are equal, and

$$\forall\,\xi_1 \in s_1^\bullet\ \exists\,\xi_2 \in s_2^\bullet : u_{s_1}^{\xi_1} = u_{s_2}^{\xi_2}.$$

Along with the states of FPs, we shall consider also **pseudo-states**, which differ from states only that their assignments have the form $u := e$, where $u \in \mathcal{E}_{conc}$, $e \in \mathcal{E}$. For each pseudo-state s the notations b_s, u_s and Θ_s have the same meaning as for states.

4.3 Unfoldinig of states

Let Σ be a FP, $s \in S_\Sigma$ be a state, $\theta \in \Theta_s$ be an assignment of the form

$$u := \varphi(v_1, \ldots, v_n)$$

and an equation in Σ that corresponds to φ has the form $\varphi(x_1, \ldots, x_n) = e_\varphi$.

Denote by s^θ a set, called an **unfolding** of the state s with respect to θ, and defined by the procedure of its construction, which consists of the steps listed below.

Step 1.

s^θ is assumed to be a singleton, which consists of a pseudo-state, derived from s by a replacement of θ with the assignment

$$u := e_\varphi(v_1/x_1, \ldots, v_n/x_n).$$

Step 2.

(This step can be performed several times until there is the possibility to perform it.)

If all the elements of the set s^θ are states from S_Σ, then the performance of this step ends, otherwise s^θ is modified in the following way.

We choose an arbitrary element $s' \in s^\theta$, which is not a state of S_Σ, and denote by θ' the first of the assignments, occurring in $\Theta_{s'}$, which has the form $u := e$, where $e \notin \mathcal{E}_\Sigma$. Consider all possible variants of the form of the term e, and for each of these variants, we present a rule of a modification of the set s^θ, according to this variant. Below, the phrase "a new variable" means "a variable that has no occurrences in the pseudo-state under consideration".

- $e \in \mathcal{C}$, in this case
  - if $u = e$, then remove θ' from s',
  - if $u \in \mathcal{X}$, then replace all occurrences of u in s' on e, and remove θ' from s',
  - otherwise remove s' from s^θ.
- $e = e'_h$, in this case replace θ' on the assignment
  - $u := e_1$, if e' has the form $e_1 e_2$,
  - $ux := e'$, where x is a new variable, otherwise.
- $e = e'_t$, in this case replace θ' on the assignment
  - $u := e_2$, if e' has the form $e_1 e_2$,
  - $xu := e'$, where x is a new variable, otherwise.
- $e = e_1 e_2$, in this case
  - if $u = u_1 u_2$, then replace θ' on a couple of assignments $u_1 := e_1$, $u_2 := e_2$,
  - if $u \in \mathcal{X}$, then replace all occurrences of u in s' on the term xy (where x and y are new variables), and θ' on the couple of assignments $x := e_1$, $y := e_2$,
  - otherwise remove s' from s^θ.
- $e = [\![e_1 = \varepsilon]\!]$, in this case
  - add to s^θ a copy of the state s' (denote it by s''),
  - replace
    - θ' in s' on the couple $u := 1$, $\varepsilon := e_1$, and
    - θ' in s'' on the couple $u := 0$, $xy := e_1$, where x and y are new variables.
- $e = [\![e_1 = e_2]\!]$, $e = [\![e_1 \le e_2]\!]$, $e = [\![e_1 \wedge e_2]\!]$ etc., in this case
  - replace θ' on the couple $x_1 := e_1$, $x_2 := e_2$, where x_1, x_2 are new variables, and
  - add to $b_{s'}$ the conjunctive member $u = e'$, where e' is derived from e by a replacement of e_i with x_i $(i = 1, 2)$.
- $e = [\![e_1]\!]\, e_2 : e_3$, in this case add to s^θ a copy of s' (denote it by s''), and replace all occurrences
  - θ' in s' on the couple $1 := e_1$, $u := e_2$,
  - θ' in s'' on the couple $0 := e_1$, $u := e_3$.
- $e = \varphi(e_1, \ldots, e_k)$, $\exists\, i : e_i \notin \mathcal{E}_{conc}$, in this case, replace e_i in θ' on the new variable x, and add $x := e_i$ before θ'.

Step 3.

For each $s' \in s^\theta$

- if $\Theta_{s'}$ has a pair of the form $u := x$, $v := x$, where $x \in \mathcal{X}$, and u, v are of the form $u_1 \dots u_n$, $v_1 \dots v_m$ respectively, then there is executed an algorithm which consists of the following steps:
  (as a result of each of the these steps it is changed a form of these assignments, but we will denote the changed assignments by the same notation as original assignments):
  - if $n < m$, then in the case $u_n \in \mathcal{X}$ each occurrence of the variable u_n in s' is replaced on the term $v_n \dots v_m$, and in the case $u_n = \varepsilon$ we remove s' from s^θ,
  - analogously in the case $m < n$,
  - $\forall\, i = 1, \dots, n$:
    - if $u_i \in \mathcal{X}$, then replace all occurrences u_i in s' on v_i, and if $u_i \notin \mathcal{X}$, but $v_i \in \mathcal{X}$, then replace all occurrences v_i in s' on u_i,
    - if $u_i \neq v_i$, then remove s' from s^θ,
  - delete one of the considered assignments,
- if $b_{s'} = [\![b', x = u]\!]$, where $x \in \mathcal{X}$, $u \in \mathcal{X} \cup \mathcal{C}$, then $b_{s'}$ is replaced on b', and all occurrences x in s' are replaced on u,
- $b_{s'}$ is simplified by
  - a replacement of subterms without variables with corresponding constants, and
  - simplifying transformations related to boolean identities and properties of equality and linear order relations,
- if $b_{s'} = \bot$, then s' is removed from s^θ.

Theorem 1.
The above procedure for constructing of the set S^θ is always terminated. ■

A state $s \in S_\Sigma$ is **inconsistent**, if it is not terminal, and $\exists\, \theta \in \Theta_s$: either $s^\theta = \emptyset$, or all states in s^θ are inconsistent.

4.4 Substitution of states in terms

Let Σ be a FP, e be a term, $x_1, \dots, x_n$ be a list of different variables from $\mathcal{X}$, and $s_1, \dots, s_n$ be a list of states from S_Σ, such that $\forall\, i = 1, \dots, n \;\; type(s_i) = type(x_i)$. The notation

$$e(s_1/x_1, \dots, s_n/x_n) \tag{20}$$

denotes a state $s_e \in S_\Sigma$, defined by induction on the structure of e:

- if $e = x_i \in \{x_1, \dots, x_n\}$, then $s_e \stackrel{\text{def}}{=} s_i$,
- if $e \in \mathcal{X} \setminus \{x_1, \dots, x_n\}$ or $e \in \mathcal{C}$, then $s_e \stackrel{\text{def}}{=} e\,(\,)$,
- if $e = g(e_1, \dots, e_k)$, where $g \in \mathcal{F} \cup \Phi$, and the states $s_{e_1}, \dots, s_{e_k}$ of the form (20), which are corresponded to terms $e_1, \dots, e_k$, are already defined, then s_e is defined as follows:

- internal variables of the states s_{e_i} are replaced on new variables by a standard way, so that all the internal variables of these states will be different, let $[\![b_i]\!]\, u_i(\Theta_i)$ $(i = 1, \ldots, k)$, be the resulting states,
- s_e is a result of an application of actions 2 and 3 from section (4.3) to the state

$$[\![b_1, \ldots, b_k]\!]\, (u_1, \ldots, u_k)\, (\Theta_1, \ldots, \Theta_k).$$

Term (20) will be denoted by the notation $e(s_1, \ldots, s_n)$, in that case, when the list of the variables $x_1, \ldots, x_n$ is clear from the context.

4.5 A concept of a state diagram of a FP

Let Σ be a FP, and left side of first equation in Σ has the form $\varphi(x_1, \ldots, x_n)$.

A **state diagram (SD)** of the FP Σ is a graph G with distinguished node n_0 (called an **initial node**) satisfying the following conditions.

– Each node n of the graph G is labelled by a state $s_n \in S_\Sigma$, and s_{n_0} has the form

$$y\,(y := \varphi(x_1, \ldots, x_n)), \quad \text{where } y \notin \{x_1, \ldots, x_n\}.$$

– For each node n of the graph G one of the following statements holds.

1. There is no an edge outgoing from n, and s_n is terminal.
2. There are two edges outgoing from n, and states s', s'' corresponding to the ends of these edges have the following property: $\exists\, x \in X_{s_n}$: $type(x) = \mathbf{S}$, there are no assignments of the form $u := x$ in Θ_{s_n}, and s', s'' are obtained from s_n by
   - a replacement of all occurrences of x with the constant ε and with the term yz respectively (where y and z are variables not occurring in X_{s_n}), and
   - if x is not occurring in the left side of any assignment from Θ_{s_n}, then – by adding assignments $\varepsilon := x$ and $yz := x$ to $\Theta_{s'}$ and $\Theta_{s''}$ respectively.
3. $\exists\, \theta \in \Theta_{s_n}$: a set of states corresponding to ends of edges outgoing from n, is equal to the set of all consistent states from s_n^θ.
4. u_{s_n} has the form $u_1 u_2$, and there is one edge outgoing from n labeled by $tail$, and the end n' of this edge satisfies the condition: $tail(s_n) \subseteq s_{n'}$.
5. There is an edge outgoing from n labelled by $<$, the end n' of which satisfies the condition:
   - $\exists\, n_1, n_2$: G contains an edge from n_1 to n_2 labelled by $tail$, and
   - $\exists\, e \in \mathcal{E}_\Sigma$, $\exists\, x \in X_e$:

$$s_n \subseteq e(tail(s_1)/x), \quad e(s_2/x) \subseteq s_{n'}.$$

We describe an informal sense of the concept of a SD. Each state s can be considered as a description of a process of a calculation of the value of the term u_s on concrete values of input variables of this state (by an execution of assignments from Θ_s, checking the condition b_s and a calculation of the value of the term

u_s on the calculated values of the variables occurring in this term). If all edges outgoing from the state n are unlabeled, then ends of these edges correspond to possible options for calculating the value of u_{s_n} (by detailization of a structure of a value of some variable from X_{s_n}, or by an equivalent transformation of any assignment from Θ_{s_n}). If there is an edge from n to n' labeled by *tail*, then this edge expresses a reduction of the problem of calculating of the tail of the value u_{s_n} to the problem of calculating the value of $u_{s_{n'}}$. If there is an edge from n to n' labeled by $<$, then this edge expresses a reduction of the problem of calculating the value u_{s_n} to the problem of calculating the value $u_{s_{n'}}$ on arguments on the smaller size.

We say that FP Σ **has a finite SD**, if there is a SD of Σ with finite set of nodes.

Theorem 2.

Let Σ_1 and Σ_2 have finite SDs, $\Phi_{\Sigma_1} \cap \Phi_{\Sigma_2} = \emptyset$, and left sides of first equations in Σ_1 and Σ_2 have the form $\varphi_1(x_1, \ldots, x_n)$ and $\varphi_2(y_1, \ldots, y_m)$ respectively, where $type(\Sigma_1) = type(y_1)$.

Then FP Σ such that

- its first equation has the form

$$\varphi(x_1, \ldots, x_n, y_2, \ldots, y_m) = \\ = \varphi_2(\varphi_1(x_1, \ldots, x_n), y_2, \ldots, y_m)$$

- and a set of other equations is $\Sigma_1 \cup \Sigma_2$

has a finite SD. ■

We do not give a description of the algorithm for the construction of a finite SD for Σ due to limitations on the size of the article. We note only that the SD is a union of a SD for Σ_1, a SD for Σ_2, and a SD, which is a Cartesian product of two previous SDs.

Theorem 3.

Let FP Σ has a finite SD, where terms, related to states corresponding to terminal nodes of this SD, which are reachable from an initial state, are equal to 1. Then f_Σ has value 1 on all its arguments. ■

The above theorems are theoretical foundation of a method of verification of FPs. This method consists in a constructing finite SDs

- for a FP Σ_1 under verification, and
- for a FP Σ_2 which represents some property of Σ_1.

If there are finite SDs for Σ_1 and Σ_2, then, according to Theorem 2, there is a finite SD for a superposition of Σ_1 and Σ_2. If this SD has the property indicated in Theorem 3, then the superposition of functions corresponding to Σ_1 and Σ_2, has the value 1 on all its arguments.

In the next section we present an example of this method.

For a constructing of SDs it is used a method of justification of statements of the form $s_1 \subseteq s_2$, which we did not set out here due to limitations on the size of the article. We only note that this method uses the concept of an unification of terms.

We shall use the following convention for graphical presentation of SDs: if a state s associated with a node of a SD has the form $[\![b]\!]\, u(\theta_1, \ldots, \theta_n)$, then this node is designated by an oval, over which it is drawn a notation $b.u$ (or u, if $b = \top$), and components of the list Θ_s are depicted inside the oval. An identifier of the node can be depicted from the left of the oval.

5 An example of verification of a FP by constructing of a state diagram

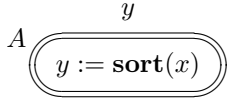
In this section we illustrate the verification method outlined above by an example of verification of FP of sorting, in this case $\Sigma_1 = (3)$ and $\Sigma_2 = (4)$.

We shall use the following convention: if nodes n_1 and n_2 of a SD are such that n_2 can be derived from n_1 by a performing of actions 2 and 3 from the definition of a SD, then we draw an unlabeled edge from n_1 to n_2 (i.e. unlabeled edges in a new understanding of a SD correspond to paths consisting of unlabeled edges in original understanding of a SD).

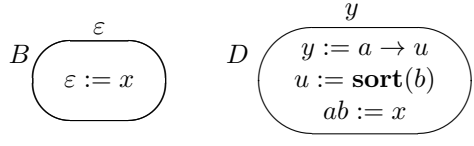
5.1 A state diagram for the FP of sorting

In this section we describe the process of building of a SD for FP (3). Terms of the form $\mathbf{insert}(a, y)$ we denote by $a \to y$.

An initial node of the SD for FP (3) (this node will be denoted by the symbol A) has the form



Two unlabeled edges can be drawn (corresponding to replacement of x with ε and with ab, and to an unfolding of one assignment) from this node to the nodes



Also it is possible to draw two unlabeled edges (corresponding to replacement of the variable u with constant ε and with the term cd) from D to nodes with labels

$$y\,(y := a \to cd, cd := \mathbf{sort}(b), ab := x), \tag{21}$$

and

$$y\,(y := a\varepsilon, \varepsilon := \mathbf{sort}(b), ab := x). \tag{22}$$

Also it is possible to draw two edges from the node labeled by (21) to nodes labeled by

$$\begin{array}{ll} C: & [\![a \le c]\!]\, acd\,(cd := \mathbf{sort}(b), ab := x), \\ G: & [\![c < a]\!]\, cz\,(z := a \to d, cd := \mathbf{sort}(b), ab := x) \end{array}$$

(by an unfolding of the first assignment).

It is possible to draw an edge labeled by *tail* from C to the initial node (the existence of such an edge is seen directly).

It is possible to draw two edges from the node labeled by (22) (replacing b to ε and to pq) to nodes, one of which is terminal and has the form

$$E: \quad a\varepsilon\,(a\varepsilon := x),$$

and the second node is inconsistent (that can be determined by additional unfoldings, which we do not present here).

It is possible to draw two unlabeled edges from G (corresponding to the replacement of b with ε and with pq) to nodes, one of which is inconsistent, and the second node is labeled by

$$[\![c < a]\!]\; cz \begin{pmatrix} z := a \to d, \\ cd := p \to w, \\ w := \mathbf{sort}(q), \\ apq := x \end{pmatrix}. \tag{23}$$

It is possible to draw two unlabeled edges from (23) (corresponding to the replacement of w with ε and ij):

– from the end of the first of these edges it can be drawn several unlabeled edges, but among ends of all these edges there is a unique consistent node labeled by

$$[\![c < a]\!]\, cz \begin{pmatrix} z := a \to \varepsilon, \\ c := p, \\ d := \varepsilon, \\ ap\varepsilon := x \end{pmatrix},$$

and there is a unique unlabeled edge from this node to a terminal node

$$H: \quad [\![c < a]\!]\, ca\varepsilon\,(ac\varepsilon := x),$$

– the end of second edge has a label

$$[\![c < a]\!]\; cz \begin{pmatrix} z := a \to d, \\ cd := p \to ij, \\ ij := \mathbf{sort}(q), \\ apq := x \end{pmatrix}. \tag{24}$$

It can be drawn a couple of edges from (24), the ends of which have labels

$$F: \quad [\![c < a, c \le i]\!]\; cz \begin{pmatrix} z := a \to ij, \\ ij := \mathbf{sort}(q), \\ acq := x \end{pmatrix},$$

$$I: \quad [\![c < a, c < p]\!]\ cz \begin{pmatrix} z := a \to d, \\ d := p \to j, \\ cj := \mathbf{sort}(q), \\ apq := x \end{pmatrix}.$$

It can be drawn an edge labeled by *tail* from F to the initial node (the existence of such an edge is seen directly).

A pair of nodes (D, I) is related to the pair of nodes (A, G) by the following relations:

$$tail(I) = e(tail(G)/h), \quad D = e(A/h) \tag{25}$$

where $e = a \to h$. In other words, labels of nodes I, D can be obtained from labels of nodes G, A by adding an assignment to the top. This fact can be used to justify an existence of an edge from G to A with label *tail*. We do not set out the detailed justification of an existence of such an edge, we describe only a scheme of such a justification. Let $\rho(x)$ be a partial function with the following property: if ρ is defined on a value α of the variable x, then it maps α to a string β, which has the property

$$u^{x \mapsto \alpha}_{tail(G)} = u^{x \mapsto \beta}_{A}.$$

The formula (25) directly implies the following property of the function ρ:

$$x \neq \varepsilon \ \Rightarrow\ \rho(x) \sqsupseteq x_h \rho(x_t) \tag{26}$$

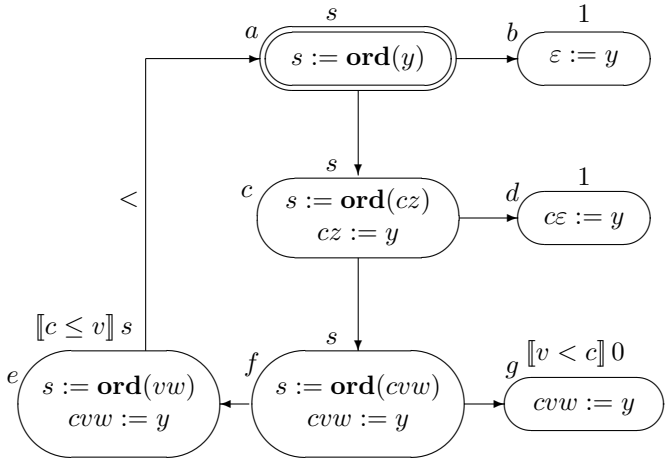
where the inequality $\sqsupseteq$ is understood as an order relation on the set of partial functions: if for some value of the variable x the right side of (26) is defined, then the left side also is defined for this value of x, and values of both parts are the same.

A property of totality of the function ρ is justified by the inequality (26) and by an analysis of a fragment of SD for (3) which is already built. Note that this justification can be generated automatically. A proof of correctness of this justification is based on the concept of unification of state pairs, it has a large volume, and we omit it.

The constructed SD for FP (3) is shown in Fig. 1. it can be simplified to the SD in Fig. 2 (we do not present here the detailed algorithm of this simplification).

5.2 A state diagram for the FP of cheking of string ordering

A fragment of a SD for FP Σ_2 (see (4)) (consisting of nodes reachable from the initial state) has the form



5.3 A state diagram for a superposition of the sorting FP and the FP of ordering checking

There is an algorithm based on Theorem 3, which can be applied to SDs for the FPs (3) and (4), which results the SD shown in Fig. 3. This SD has two terminal nodes, and labels of both these nodes have a value of 1. According to Theorem 3, this implies that the function **ord** ∘ **sort** has the value 1 on all its arguments.

In conclusion we note, that despite on the complexity of all of the above transformations and reasonings, all of them can be generated automatically. An attempt to justify an existence of edges with labels *tail* and $<$ can be executed automatically for each pair of nodes arising in the process of building of the SD. It can be seen from this example that the process of a construction of a SD is terminated fast enough.

6 Conclusion

We have proposed the concept of a state diagram (SD) for functional programs (FPs) and a verification method based on the concept of a SD. One of the problems for further research related to the concept of a SD has the following form: find a sufficient condition φ (as strong as possible) on a FP Σ such that if Σ meets φ then Σ has a finite SD.

References

1. R.W. Floyd: Assigning meanings to programs. In J.T. Schwartz, editor, Proceedings Symposium in Applied Mathematics, Mathematical Aspects of Computer Science, pages 19-32. AMS, 1967.

2. C. A. R. Hoare: An axiomatic basis for computer programming. Communications of the ACM, 12(10): 576580, 583, October 1969.
3. R. Milner: A Calculus of Communicating Systems. Number 92 in Lecture Notes in Computer Science. Springer Verlag, 1980.
4. R. Milner: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press, 1999.
5. Hoare, C. A. R.: Communicating sequential processes. Communications of the ACM 21 (8): 666677, 1978.
6. Separation Logic: A Logic for Shared Mutable Data Structures. John C. Reynolds. LICS 2002.
7. Clarke, E.M., Grumberg, O., and Peled, D.: Model Checking. MIT Press, 1999.
8. J.A. Bergstra, A. Ponse, and S.A. Smolka, editors: Handbook of Process Algebra. North-Holland, Amsterdam, 2001.
9. C.A. Petri: Introduction to general net theory. In W. Brauer, editor, Proc. Advanced Course on General Net Theory, Processes and Systems, number 84 in LNCS, Springer Verlag, 1980.
10. Z. Manna: Mathematical Theory of Computation. McGraw-Hill Series in Computer Science,1974.
11. N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. Theoretical Computer Science, 375:120136, 2007.
12. Ranjit Jhala, Rupak Majumdar, Andrey Rybalchenko: HMC: Verifying Functional Programs Using Abstract Interpreters, `http://arxiv.org/abs/1004.2884`
13. N. Kobayashi and C.-H. L. Ong. A type theory equivalent to the modal mu- calculus model checking of higher-order recursion schemes. In Proceedings of LICS 2009. IEEE Computer Society, 2009.
14. C.-H. L. Ong. On model- checking trees generated by higher order recursion schemes. In Proceedings 21st Annual IEEE Symposium on Logic in Computer Science, Seattle, pages 8190. Computer Society Press, 2006.
15. N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multiparameter tree transducers and recursion schemes for program verification. In POPL, pages 495508, 2010.

Fig. 1:

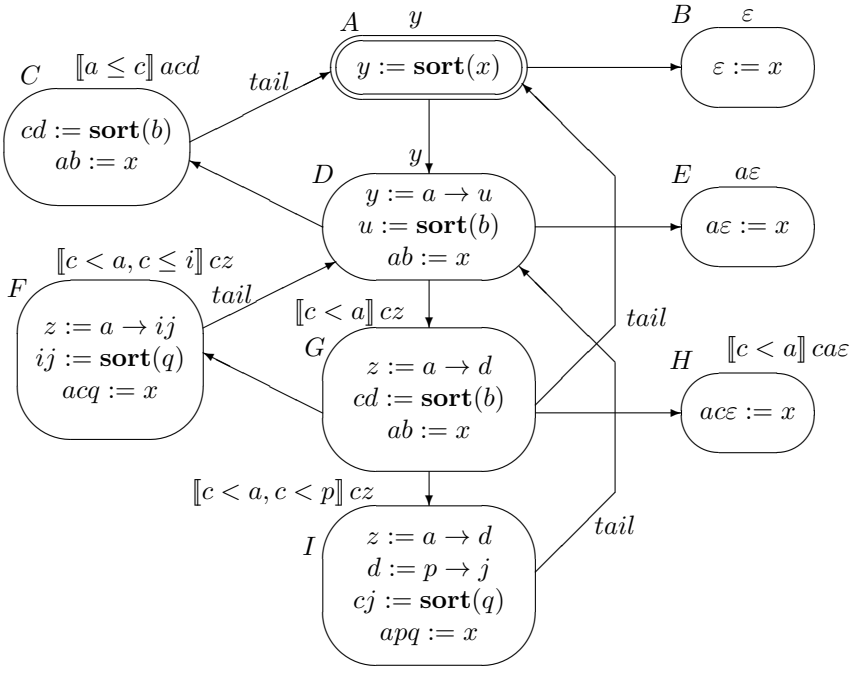

A
y
y := sort(x)
B
ε
ε := x
C
[[a ≤ c]] acd
cd := sort(b)
ab := x
tail
y
D
y := a → u
u := sort(b)
ab := x
E
aε
aε := x
[[c < a, c ≤ i]] cz
F
z := a → ij
ij := sort(q)
acq := x
tail
[[c < a]] cz
tail
G
z := a → d
cd := sort(b)
ab := x
H
[[c < a]] caε
acε := x
[[c < a, c < p]] cz
I
z := a → d
d := p → j
cj := sort(q)
apq := x
tail

Fig. 2:

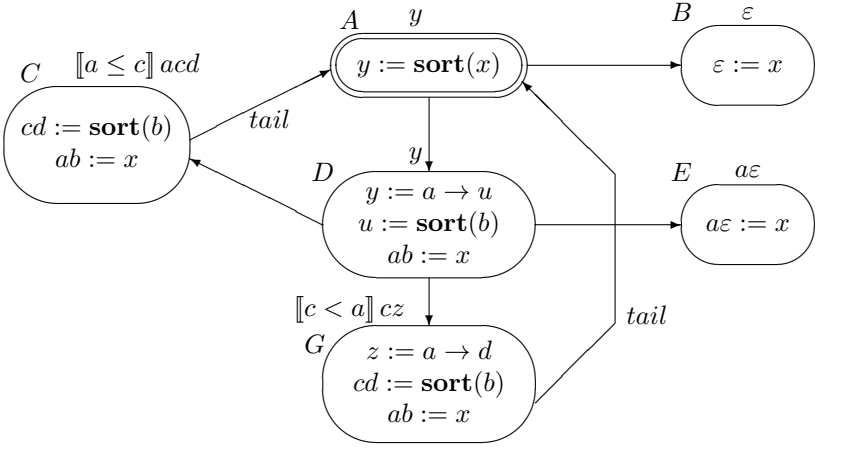
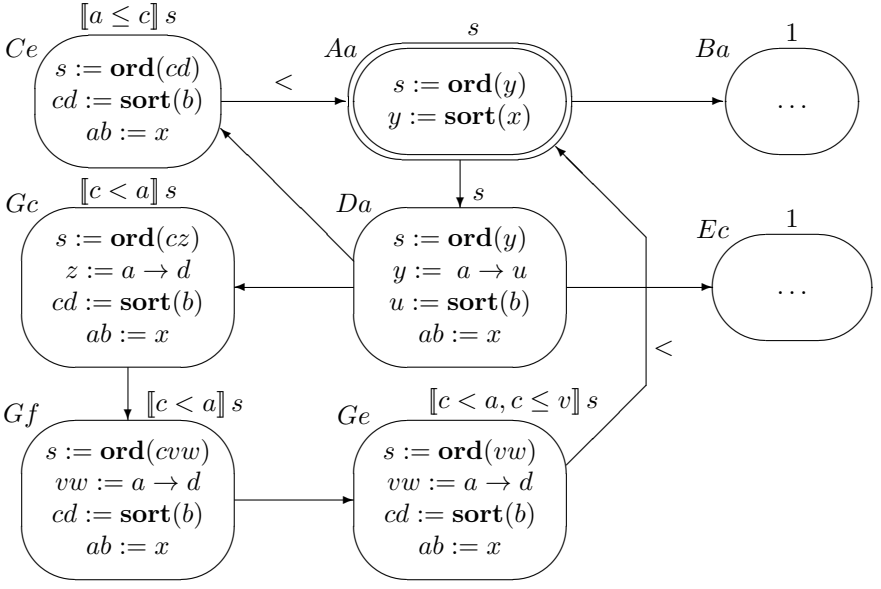

A
y
y := sort(x)
B
ε
ε := x
C
[[a ≤ c]] acd
cd := sort(b)
ab := x
tail
y
D
y := a → u
u := sort(b)
ab := x
E
aε
aε := x
[[c < a]] cz
G
z := a → d
cd := sort(b)
ab := x
tail

Fig. 3:


Ce
⟦a ≤ c⟧ s
s := ord(cd)
cd := sort(b)
ab := x
<
Aa
s
s := ord(y)
y := sort(x)
Ba
1
...
Gc
⟦c < a⟧ s
s := ord(cz)
z := a → d
cd := sort(b)
ab := x
Da
s
s := ord(y)
y := a → u
u := sort(b)
ab := x
Ec
1
...
<
Gf
⟦c < a⟧ s
s := ord(cvw)
vw := a → d
cd := sort(b)
ab := x
Ge
⟦c < a, c ≤ v⟧ s
s := ord(vw)
vw := a → d
cd := sort(b)
ab := x