Universal System of Interpretation of Strongly Typed Functional Programs

Ruben Hakopian

Yerevan State University
Yerevan, Armenia
e-mail: ruben.hakopian@gmail.com

ABSTRACT

The paper is devoted to implementation of universal system of interpretation of strongly typed functional programs, which uses following seven algorithms: active, passive and algorithms based on reduction to normal form and full, parallel external, parallel internal, left external and left internal substitutions.

Keywords

Interpretation system, strongly typed, functional program, interpretation algorithm.

1. DEFINITIONS AND RESULTS USED

Let us give definitions used in the paper. Most of them can be found in [1], [2] and [3].

Let M be a partially ordered set, which has a least element $\perp$, and each element of M is comparable with itself and $\perp$ only. Let us define the set types $Types$:

1. $M \in Types$;
2. if $\beta, \alpha_1, ..., \alpha_k \in Types$ $(k > 0)$, then the set of all monotomic mappings from $\alpha_1 \times ... \times \alpha_k$ into β (denoted by $[\alpha_1 \times ... \times \alpha_k \to \beta]$) belongs to $Types$.

Let $\alpha \in Types$. The order of the type α is a natural number (defined as $ord(\alpha)$), where:

1. if $\alpha = M$, then $ord(\alpha) = 0$;
2. if $\alpha = [\alpha_1 \times ... \times \alpha_k \to \beta]$, then $ord(\alpha) = \max(ord(\alpha_1), ..., ord(\alpha_k), ord(\beta)) + 1$.

For each $\alpha \in Types$ we will have a countable set of variables V_α of type α. If the variable $x \in V_\alpha$ and constant $c \in \alpha$, then $ord(x) = ord(c) = ord(\alpha)$. Let us define the set of terms: $\Lambda = \bigcup_{\alpha \in Types} \Lambda_\alpha$, where Λ_α - is set of terms of type α, the following way:

1. if $c \in \alpha$, $\alpha \in Types$, then $c \in \Lambda_\alpha$;
2. if $x \in V_\alpha$, $\alpha \in Types$, then $x \in \Lambda_\alpha$;
3. if $t \in \Lambda_{[\alpha_1 \times .. \times \alpha_k \to \beta]}$, $t_i \in \Lambda_{\alpha_i}$, $\beta, \alpha_i \in Types$, $i = 1,...,k$ $(k > 0)$, then $t(t_1,...,t_k) \in \Lambda_\beta$ and $(t_1,...,t_k)$ will be denoted by area of influence of applicator t;
4. if $t \in \Lambda_\beta$, $x_i \in V_{\alpha_i}$, $\beta, \alpha_i \in Types$, $i \neq j \Rightarrow x_i \neq x_j$, $i, j = 1,...,k$ $(k > 0)$, then $\lambda x_1 ... x_k[t] \in \Lambda_{[\alpha_1 \times .. \times \alpha_k \to \beta]}$;

The notions of a free and bound occurrence of a variable in a term and the notation of a free variable of a term are introduced in a conventional way. The set of all free variables of a term t is denoted by $FV(t)$. Terms t_1, t_2 are said to be congruent (which is denoted by $t_1 \equiv t_2$) if one term can be obtained from the other by renaming bound variables. In what follows, congruent terms are considered identical.

Let $t \in \Lambda$, $t_i \in \Lambda_{\alpha_i}$, $x_i \in V_{\alpha_i}$, $\alpha_i \in Types$, $i \neq j \Rightarrow x_i \neq x_j$, $i, j = 1,...,k$ $(k > 0)$. The simultaneous substitution of the terms $t_1,...,t_k$ for some of free occurrences of the variables $x_1,...,x_k$ respectively, into term t will be admissible, if all considered free variables of the term being substituted remain free after substitution. Term t, where some of free occurrences of the variables $x_1,...,x_k$ are interested is denoted by $t < x_{i_1}, \ldots, x_{i_s} >$, $x_{i_j} \in \{x_1, \ldots, x_k\}$, $j = 1, \ldots, s$. The term obtained from a term $t < x_{i_1}, \ldots, x_{i_s} >$ as a result of substitution of terms $t_{i_1},...,t_{i_s}$ for occurrences of the variables $x_{i_1}, \ldots, x_{i_s}$ respectively will be denoted by $t < t_{i_1}, \ldots, t_{i_s} >$. The term obtained from a term t as a result of substitution of terms $t_1,...,t_k$ for all free occurrences of variables $x_1,..., x_k$ respectively will be denoted by $t\{t_1 / x_1, \ldots, t_k / x_k\}$.

For each term $t \in \Lambda_\alpha$ we will correspond a constant $Val_{\bar{y}_0}(t) \in \alpha, \alpha \in Types$, where $FV(t) \subset \{y_1, \ldots, y_n\}$, $y_i \in V_{\alpha_i}$, $\bar{y}_0 =< y_1^0, \ldots, y_n^0 >$, $y_i^0 \in \alpha_i$, $\alpha_i \in Types$, $i = 1, \ldots, n$ $(n \geq 0)$, the following way:

1. if $t \equiv c$, $c \in \alpha$, then $Val_{\bar{y}_0}(t) = c$;
2. if $t \equiv x$, $x \in V_\alpha$, then $Val_{\bar{y}_0}(t) = y_i^0$, where $x \equiv y_i$, $i = 1,...,n$ $(n > 0)$;
3. if $t \equiv \tau(t_1,...,t_k) \in \Lambda_\alpha$, $\tau \in \Lambda_{[\alpha_1 \times .. \times \alpha_k \to \alpha]}$, $t_i \in \Lambda_{\alpha_i}$, $\alpha_i \in Types$, $i = 1,...,k$ $(k \geq 1)$, then $Val_{\bar{y}_0}(\tau(t_1,...,t_k)) = Val_{\bar{y}_0}(\tau)(Val_{\bar{y}_0}(t_1), \ldots, Val_{\bar{y}_0}(t_k))$;
4. if $t \equiv \lambda x_1 ... x_k[\tau] \in \Lambda_{[\alpha_1 \times .. \times \alpha_k \to \beta]}$, $\tau \in \Lambda_\beta$, $x_i \in V_{\alpha_i}$, $\beta, \alpha_i \in Types$, $i = 1,...,k$ $(k \geq 1)$, then $Val_{\bar{y}_0}(\lambda x_1 ... x_k[\tau]) : \alpha_1 \times \ldots \times \alpha_k \to \beta$ and for any $\bar{x}_0 =< x_1^0, \ldots, x_k^0 >$, $x_i^0 \in \alpha_i$, $i = 1, \ldots k$ will have $Val_{\bar{y}_0}(\lambda x_1 ... x_k[\tau])(\bar{x}_0) = Val_{\bar{x}_0 \bar{y}_0'}(\tau)$, where $FV(\lambda x_1 ... x_k[\tau]) = \{y_{i_1}, \ldots y_{i_m}\}$, $\bar{y}_0' =< y_{i_1}^0, \ldots, y_{i_m}^0 >$.

Let t_1, t_2 be terms and $FV(t_1) \cup FV(t_2) = \{y_1, \ldots, y_n\}$, $y_i \in V_{\alpha_i}$, $\alpha_i \in Types$, $i = 1, \ldots n$ $(n \geq 0)$. Terms t_1, t_2 will be called equivalent (denoted by $t_1 \sim t_2$), if for any

$\bar{y}_0 =< y_1^0,\ldots, y_n^0 >$, where $y_i^0 \in \alpha_i, i = 1,\ldots,n$, it is true $Val_{\bar{y}_0}(t_1) = Val_{\bar{y}_0}(t_2)$. Let $t \in \Lambda_\alpha$, $\alpha \in Types$, $FV(t) \subseteq \{y_1,\ldots, y_n\}$, $y_i \in V_{\alpha_i}$, $\alpha_i \in Types$, $i = 1,...,n$ $(n \geq 0)$. Term t will be called constant term with $b \in \alpha$ value, if for any $\bar{y}_0 =< y_1^0,\ldots, y_n^0 >$, $y_i^0 \in \alpha_i$, $i = 1,\ldots,n$ we have $Val_{\bar{y}_0}(t) = b$. Further we will consider terms, which do not use constants with order ≥ 2.

A strongly type functional program P is a system of equations of the form

$$\begin{cases} F_1 = \tau_1 \\ \ldots \\ F_n = \tau_n \end{cases} \quad (1)$$

where $F_i \in V_{\alpha_i}$, $i \neq j \Rightarrow F_i \not\equiv F_j$, $\tau_i \in \Lambda_{\alpha_i}$, $FV(\tau_i) \subset \{F_1,\ldots,F_n\}$, $\alpha_i \in Types$, $i, j = 1,\ldots,n$ $(n \geq 1)$, all used constants have an order ≤ 1, constants of order 1 are computable mappings and $\alpha_1 = [M^k \to M]$ $(k \geq 1)$. The mapping $\Psi_P : \alpha_1 \times \ldots \times \alpha_n \to \alpha_1 \times \ldots \times \alpha_n$, where $\Psi_P(\bar{g}) =< Val_{\bar{g}}(\tau_1),\ldots, Val_{\bar{g}}(\tau_n) >$ and $\bar{g} \in \alpha_1 \times \ldots \times \alpha_n$ will be called a mapping corresponding the program (1). By $(\bar{g})_i$, $i = 1,\ldots,n$ we will denote i^{th} component of the vector $\bar{g}$. We will say that $\bar{f} \in \alpha_1 \times \ldots \times \alpha_n$ is the solution of the program (1), if $\Psi_P(\bar{f}) = \bar{f}$. In [1] it is proved that any program (1) has a least solution. The equation $F_1 = \tau_1$ is called the main equation of program P, and the function $f_P = (\bar{f})_1$ – the fixpoint semantics of the program P, where $\bar{f}$ is the least solution of the program P. The set $Fix(P)$, which corresponds to fixpoint semantics of the program P, is defined the following way: $Fix(P) = \{< \bar{m}, m > | f_P(\bar{m}) = m,\ \bar{m} \in M^k,\ m \in M\}$.

Let us remind the notion of β-reduction. $\beta = \{< \lambda x_1 \ldots x_k[t](t_1,\ldots t_k),\ t\{t_1 / x_1,\ldots,\ t_k / x_k\} > | x_i \in V_{\alpha_i}, t_i \in \Lambda_{\alpha_i}, \alpha_i \in Types, i = 1,\ldots,k\ (k \geq 1)\}$. Term $\lambda x_1 \ldots x_k[t](t_1,\ldots t_k)$ is called β-redex, and $t\{t_1 / x_1,\ldots,t_k / x_k\}$ – its bundle. We will say, that term t' is obtained from term t by one-step β-reduction $(t \to_\beta t')$, if $t \equiv t_\tau, t' \equiv t_{\tau'}$, τ is β-redex, and τ' – is its bundle, where t_τ is denoted by term t with some fixed occurrence of sub-term τ, and $t_{\tau'}$ – term, obtained from substitution of the particular occurrence of sub-term τ with term τ'. We will say, that term t' is obtained from term t by β-reduction $(t \to\to_\beta t')$, if either $t \equiv t'$, or exists a sequence of terms $t_1,\ldots,t_k$ $(k \geq 0)$ for which $t \to_\beta t_1 \to_\beta \ldots \to_\beta t_k \to_\beta t'$.

Let us remind the notion of δ-reduction. Let $\Delta = \{< f(t_1,\ldots t_k), \tau > | f$ is a constant, $\tau, t_1,\ldots t_k$ $(k > 0)$ are terms and either τ – is constant and $f(t_1,\ldots t_k)$ is constant term with value τ, or τ – is a sub-term of term $f(t_1,\ldots t_k)$ and $f(t_1,\ldots t_k) \sim \tau\}$. Any recursive subset δ of set Δ is defined as notion of δ-reduction. If $< \tau, \tau' > \in \delta$, then τ is noted to be δ-redex, and τ' – its bundle. We will say, that term t' is obtained from term t by one-step δ-reduction $(t \to_\delta t')$, if $t \equiv t_\tau, t' \equiv t_{\tau'}$ and τ is δ-redex, and τ' – its bundle. We will say, that term t' is obtained from t by δ-reduction $(t \to\to_\delta t')$, if either $t \equiv t'$, or exists a sequence of terms $t_1,\ldots,t_k$ $(k \geq 0)$ for which $t \to_\delta t_1 \to_\delta \ldots \to_\delta t_k \to_\delta t'$.

The notion of $\beta \cup \delta$-reduction will be denoted by $\beta\delta$. The $\beta\delta$-reduction will be called just reduction, $\beta\delta$-redex will be called just redex, one-step $\beta\delta$-reduction will be denoted by $\to$, and $\beta\delta$-reduction – $\to\to$. The term not containing redexes is referred as normal form. The set of all normal forms is denoted by NF.

Let us denote Δ_0 as the following subset of the set Δ: $\Delta_0 = \{< f(t_1,\ldots t_k), \tau > | < f(t_1,\ldots t_k), \tau > \in \Delta$, where τ is either a constant, or $\tau \equiv t_i\ (1 \leq i \leq k)\}$. The notion of δ-reduction will be called natural, if:

1. $\delta \subset \Delta_0$;
2. δ – is a single-valued relation, i.e. if $< t, \tau_1 > \in \delta$ and $< t, \tau_2 > \in \delta$, then $\tau_1 \equiv \tau_2$, where $t, \tau_1, \tau_2 \in \Lambda$;
3. for each constant term $f(t_1,\ldots t_k)$ with value $m \in M$, $f(t_1,\ldots t_k) \to\to m$, where f is a constant and $t_1,\ldots t_k \in \Lambda$.

In [3] is given a notion of substitutionability and inheritability of δ-reduction. It is also proved, that $t \to\to t', t \to\to t'', t', t'' \in NF \Rightarrow t' \equiv t''$ for any terms t, t' and t'', if and only if the δ-reduction is natural and featured with substitutionability and inheritability. Further, we will consider only these δ-reductions.

2. INTERPRETATION ALGORITHMS

Let us introduce the notion of an interpretation algorithm A. Having received a program P of form (1) and a term $F_1(\bar{m})$ $(where\ \bar{m} \in M^k)$ on its input, the algorithm A either terminates with the result $m \in M$ or works endlessly. The algorithm A uses following three kinds of operations:

1. substitution of terms $\tau_1,\ldots,\tau_n$ for some free occurrences of variables $F_1,\ldots,F_n$ respectively;
2. a one-step β-reduction;
3. a one-step δ-reduction.

Let A be an interpretation algorithm and P a program. The set $\mathrm{Proc}_A(P)$, which corresponds the procedural semantics which uses the interpretation algorithm A, we will define the following way: $\mathrm{Proc}_A(P) = \{< \bar{m}, m > |$ algorithm A on P and $F_1(\bar{m})$ terminates with a result $m \neq \perp$, where $\bar{m} \in M^k$, $m \in M\}$. We will say, that the procedural semantics which uses the interpretation algorithm A is consistent, if $\mathrm{Proc}_A(P) \subset Fix(P)$ is true for any program P.

In [4] is proven the following theorem (on consistency). The procedural semantics which uses any interpretation algorithm is consistent.
We will describe seven interpretation algorithms.

Algorithm ACT (active).
Input: program P, term t.
Output: term $\mathrm{ACT}(P, t)$ if ACT is defined on P and t.

1. if $t \in NF$ and $\mathrm{FV}(t) \cap \{F_1,\ldots,F_n\} = \emptyset$, then t, else go to 2;
2. let $t \equiv t < F_i >$, where F_i is the leftmost occurrence of variables $\{F_1,\ldots,F_n\}$ in term t, and this particular occurrence is on the left of the leftmost redex of the term t, then $\mathrm{A}CT(P, t < \tau_i >)$, else go to 3;
3. if $t \equiv t_{\lambda x_1 \ldots x_k [\tau](t_1,\ldots,t_k)}$ and $\lambda x_1 \ldots x_k [\tau](t_1,\ldots,t_k)$ is the leftmost redex of the term t, then $\mathrm{ACT}(P, t_{\tau\{ACT(P,t_1)/x_1,\ldots,ACT(P,t_k)/x_k\}})$, else go to 4;
4. if $t \equiv t_\tau$ and $\tau -$ is the leftmost δ-redex, then $\mathrm{ACT}(P, t_{\tau'})$, where $\tau' -$ is the bundle of the redex τ.

Algorithm PAS (passive).
Input: program P, term t.
Output: term $\mathrm{PA}S(P,t)$ if PAS is defined on P and t.

1. if $t \in NF$ and $\mathrm{FV}(t) \cap \{F_1,\ldots,F_n\} = \emptyset$, then t, else go to 2;
2. let $t \equiv t < F_i >$, where F_i is the leftmost occurrence of variables $\{F_1,\ldots,F_n\}$ in term t, and this particular occurrence is on the left of the leftmost redex of the term t, then $\mathrm{PA}S(P, t < \tau_i >)$, else go to 3;
3. if $t \equiv t_{\lambda x_1 \ldots x_k [\tau](t_1,\ldots,t_k)}$ and $\lambda x_1 \ldots x_k [\tau](t_1,\ldots,t_k)$ is the leftmost redex of the term t, then $\mathrm{PAS}(P, t_{\tau\{t_1/x_1,\ldots,t_k/x_k\}})$, else go to 4;
4. if $t \equiv t_\tau$ and $\tau -$ is the leftmost δ-redex, then $\mathrm{PAS}(P, t_{\tau'})$, where $\tau' -$ is the bundle of the redex τ.

Algorithm FS (full substitution).
Input: program P, term t.
Output: term $\mathrm{FS}(P,t)$ if FS is defined on P and t.

1. if $t \in NF$ and $\mathrm{FV}(t) \cap \{F_1,\ldots,F_n\} = \emptyset$, then t, else if $t \notin NF$, go to 2, else go to 3;
2. let $t \equiv t_\tau$ and τ is leftmost redex, then $\mathrm{FS}(P, t_{\tau'})$, where τ' is bundle of the redex τ;
3. let $t \equiv t < F_{i_1},\ldots,F_{i_s} >$, where $F_{i_1},\ldots,F_{i_s}$ are all occurrences of variables $\{F_1,\ldots,F_n\}$ in term t, then $\mathrm{FS}(P, t < \tau_{i_1},\ldots,\tau_{i_s} >)$.

The free occurrence of variable in a term will be called internal, if it is not a part of the applicator the area of influence of which includes a free occurrence of a variable. The free occurrence of a variable in a term will be called external, if it is not a part of an area of an influence of an applicator which contains a free occurrence of a variable.

Algorithm PES (parallel external substitution).
Input: program P, term t.
Output: term $\mathrm{PES}(P,t)$ if PES is defined on P and t.

1. if $t \in NF$ и $\mathrm{FV}(t) \cap \{F_1,\ldots,F_n\} = \emptyset$, then t, else if $t \notin NF$, go to 2, else go to 3;
2. let $t \equiv t_\tau$ and τ is the leftmost redex, then $\mathrm{PES}(P, t_{\tau'})$, where τ' is bundle of the redex τ;
3. let $t \equiv t < F_{i_1},\ldots,F_{i_s} >$, where $F_{i_1},\ldots,F_{i_s}$ are all external occurrences of variables $\{F_1,\ldots,F_n\}$ in term t, then $\mathrm{PES}(P, t < \tau_{i_1},\ldots,\tau_{i_s} >)$.

Algorithm PIS (parallel internal substitution).
Input: program P, term t.
Output: term $\mathrm{PIS}(P,t)$ if PIS is defined on P and t.

1. if $t \in NF$ and $\mathrm{FV}(t) \cap \{F_1,\ldots,F_n\} = \emptyset$, then t, else if $t \notin NF$, go to 2, else go to 3;
2. let $t \equiv t_\tau$ and τ is the leftmost redex, then $\mathrm{PIS}(P, t_{\tau'})$, where τ' is bundle of the redex τ;
3. let $t \equiv t < F_{i_1},\ldots,F_{i_s} >$, where $F_{i_1},\ldots,F_{i_s}$ are all internal occurrences of variables $\{F_1,\ldots,F_n\}$ in term t, then $\mathrm{PIS}(P, t < \tau_{i_1},\ldots,\tau_{i_s} >)$.

Algorithm LES (left external substitution).
Input: program P, term t.
Output: term $\mathrm{LES}(P,t)$ if LES is defined on P and t.

1. if $t \in NF$ and $\mathrm{FV}(t) \cap \{F_1,\ldots,F_n\} = \emptyset$, then t, else if $t \notin NF$, go to 2, else go to 3;
2. let $t \equiv t_\tau$ and τ is the leftmost redex, then $\mathrm{LES}(P, t_{\tau'})$, where τ' is bundle of the redex τ;
3. let $t \equiv t < F_i >$, where F_i is the leftmost external occurrence of variables $\{F_1,\ldots,F_n\}$ in term t, then $\mathrm{LES}(P, t < \tau_i >)$.

Algorithm LIS (left internal substitution).
Input: program P, term t.
Output: term $\mathrm{LIS}(P,t)$ if LIS is defined on P and t.

1. if $t \in NF$ and $\mathrm{FV}(t) \cap \{F_1,\ldots,F_n\} = \emptyset$, then t, else if $t \notin NF$, go to 2, else go to 3;
2. let $t \equiv t_\tau$ and τ is the leftmost redex, then $\mathrm{LIS}(P, t_{\tau'})$, where τ' is bundle of the redex τ;
3. let $t \equiv t < F_i >$, where F_i is the leftmost internal occurrence of variables $\{F_1,\ldots,F_n\}$ in term t, then $\mathrm{LIS}(P, t < \tau_i >)$.

3. UNIVERSAL SYSTEM OF INTERPRETATION (USI)

Here is described the implementation of universal system of interpretation of strongly type functional programs of form (1). In section 3.1 will be given the notion of semantic tree and an algorithm for generating a semantic tree for a given term. In section 3.2 will be described the implementation of USI.

3.1. Semantic Tree

Let us denote $Node(value, T_1, ..., T_k)$, $k \geq 0$ as a semantic tree, where $T_1, ..., T_k$ are its semantic sub-trees, and the *value* is one of the following:

1. $value = c$, where c is constant and $k = 0$;
2. $value = x$, where x is variable and $k = 0$;
3. $value = application$, where $k \geq 2$;
4. $value = abstraction$, where $k \geq 2$, $T_1, ..., T_{k-1}$ are semantic trees with $value = x$, where x is variable.

Given the term t, algorithm TREE returns corresponding semantic tree T.

Algorithm TREE

Input: term t.

Output: semantic tree T.

1. if $t \equiv c$, then $Node(c)$;
2. if $t \equiv x$, then $Node(x)$;
3. if $t \equiv \tau(t_1,...,t_n)$, then $Node(application, TREE(\tau), TREE(t_1), ..., TREE(t_n))$;
4. if $t \equiv \lambda x_1,...,x_n[\tau]$, then $Node(abstraction, TREE(x_1), ..., TREE(x_n), TREE(\tau))$;

Reductions and substitutions on semantic tree are defined the following way: let $t,t' \in \Lambda, t \to t'$. We will say, that tree $T' = TREE(t')$ is obtained from tree $T = TREE(t)$ using one-step reduction. The similar way is defined reduction operation. Let $t,t' \in \Lambda$, $t'\{\tau_1 / x_1,...,\tau_k / x_k\}$, $\tau_i \in \Lambda_{\alpha_i}$, $x_i \in V_{\alpha_i}$, $\alpha_i \in Types$, $i = 1,...,k, k \geq 1$. We will say, that tree $T' = TREE(t')$ is obtained from tree $T = TREE(t)$ using substitution of occurrences of variables x_i.

Given the semantic tree T, algorithm TYPE returns the type of the term t for which $T = TREE(t)$.

Algorithm TYPE

Input: semantic tree $T = Node(value, T_1,...,T_k)$.

Output: α, where $t \in \Lambda_\alpha$ and $T = TREE(t)$.

1. if $value = c$, $c \in \Lambda_\alpha$ is constant then α;
2. if $value = x$, $x \in \Lambda_\alpha$ is variable then α;
3. if $value = application$, $\alpha_i = TYPE(T_i)$, $i = 1,...,k$, and $\alpha_1 = [\alpha_2 \times ... \times \alpha_k \to \beta]$ then β;
4. if $value = abstraction$, $\alpha_i = TYPE(T_i)$, $i = 1,...,k$, then $[\alpha_1 \times ... \times \alpha_{k-1} \to \alpha_k]$.

3.2. Implementation of the System

Given the program P and terms $\overline{m}$ the system generates semantic trees for terms $F_1(\overline{m})$ and $\tau_1,...,\tau_n$. The lexical analyzer tokenizes terms $F_1(\overline{m}), \tau_1,...,\tau_n$ into lexemes and checks for the correctness of braces. These lexemes are processed by semantic analyzer and following semantic trees: $T = TREE(F_1(\overline{m}))$, $T_i = TREE(\tau_i)$, $i = 1,...,n$ are generated using the algorithm TREE. The strong type correctness of the program P is checked using algorithm TYPE by ensuring that $\alpha_i = TYPE(T_i)$, $i = 1,...,n$. The USI starts the interpretation from the tree T. The system iteratively performs reduction and substitution operations over the semantic tree based on the chosen interpretation algorithm and terminates with a result $Node(m), m \in M$ or works endlessly.

The interpretation algorithms: ACT, PAS, FS, PIS, PES, LIS and LES described in section 2 are implemented in the USI. The system provides step-by-step tracing and automatic termination of execution if maximum number of iteration is elapsed. The system also provides comparison engine for algorithms. Having a program P and chosen multiple interpretation algorithms the system simultaneously interprets the program using chosen interpretation algorithms. In the system there are implemented serialization facilities for persisting programs and with ability of loading them further.

The system Graphical User Interface (GUI) provides functionality of constructing terms, programs, program inputs and controlling the interpreter. During the interpretation the GUI outputs interpretation progress and graphically renders the semantic tree to the screen. It also provides facilities for step-by-step interpretation.

The USI was implemented on C# language based on Microsoft .NET Framework 2.0. The GUI was developed using Windows Forms 2.0 library.

REFERENCES

[1] S.A. Nigiyan, "Functional Languages", *Programming and Computer Software*, Vol. 17, pp. 290-297, 1992.

[2] S.A. Nigiyan, "On Interpretation of Functional Programming Languages", *Programming and Computer Software*, Vol. 19, N 2, pp. 71-78, 1994.

[3] L.E. Budaghyan, "On Formalizing of Notion of δ-reduction in Monotomic Models of Typed λ-calculus", *YSU: Uchenie Zapiski*, N 1, pp. 27-36, 2000 (in Russian).

[4] R.Y. Hakopian, "On Procedural Semantics of Strongly Typed Functional Programs", *YSU: Uchenie Zapiski*, N 3, pp. 59-70, 2008 (in Russian).

[5] M. Michaelis, "Essential C# 2.0 (Microsoft .NET Development Series)", *Addison-Wesley*, 2006.