Time and Memory Profile of a Process Functional Program

Ján Kollár, Jaroslav Porubän, Peter Václavík

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
Jan.Kollar@tuke.sk

Abstract: An execution profiling attempts to provide feedback by reporting to the programmer information about inefficiencies within the program Instead of writing whole code highly optimized, the programmer can initially write simple maintainable code without much concern for efficiency. Profiling is an effective tool for finding hot spots in a program or sections of code that consumes most of the computing time and space. The paper presents already implemented execution profiler for process functional program. From the viewpoint of implementation, process functional language is between an impure eager functional language and a monadic lazy pure functional language. The key problem of execution profiling is to relate gathered information about an execution of the program back to the source code in well defined manner. The paper defines language constructs for monitoring resource utilization during the program execution. In our solution programmer can associate label with each expression in a program. All resources used during the evaluation of a labeled expression are mapped to the specified label. The paper is concerned with formal profiling model. Research results are presented on sample program illustrating different types of time and space profiles generated by already implemented profiler for process functional programs.

Keywords. Functional programming, program profiling, process functional language, formal profiling model

1 Introduction

A purely functional language is concise, composable and extensible. The reasoning about the pure functional programs defined in terms of expressions and evaluated without side effects is simpler than the reasoning about the imperative programs describing the tasteful systems. From the viewpoint of systems design, it seems more appropriate (at least to most of programmers) to describe the systems using an imperative language, expressing the state explicitly by variables as memory cells. Although the reliability of an imperative approach may be increased using object oriented paradigm, it solves neither the problem of reasoning about the functional correctness of fine grains of computation, since they are still affected by subsequent updating the cells in a sequence of assignments, nor the problem of profiling the program to obtain the execution satisfying the time requirements of a user.

Using the today compilers, code generators and tools, programmer can define functionality of a program on a higher abstract level than anytime before. Many programmers write their programs without knowledge of resource utilization during the program execution what leads to inefficiencies within the code. Barry Boehm reports that he has measured that 20 percent of the routines consume 80 percent of the execution time 0. Donald Knuth found that less than 4 percent of a program usually accounts for more than 50 percent of its run time 0. That is why the code optimization is so important. An execution profiling attempts to provide feedback by reporting to the programmer information about inefficiencies within the program 0. Informations about resource utilization are collected during the program execution. Instead of writing whole code highly optimized, the programmer can initially write simple, maintainable code without much concern for efficiency. Once completed the performance can be profiled, and effort spent improving the program where it is necessary 0. Profiling 00 is an effective tool for finding hot spots in a program, the functions or sections of code that consume most of the computing time. Profiles should be interpreted with care, however. Given the sophistication of compilers and the complexity of caching and memory effects, as well as the fact that profiling a program affects its performance, the statistics in a profile can be only approximate.

Many of ideas for process functional program profiling come out a pure functional program profiling because of the same functional basis 00,000,0,0. Our previous work proved that all process functional programs can be easily transformed into pure functional programs 0 using state transformers and monads. The paper presents our approach to profiling of process functional program and formal model of process functional program profiling. It is simple to extend the approach to both imperative and functional language.

2 Process Functional Language PFL

From the viewpoint of implementation, PFL is between an impure eager functional language and a monadic lazy pure functional language. The main difference between a process functional language and a pure functional language is variable environment which is designed to fulfill the needs of easier state representation in a functional program 0.

Variable environment in PFL is a mapping from variable to its value. The variable environment are updated and accessed during the runtime implicitly applying the process to values. The process in the process functional language differs from a function in a purely functional language only by its type definition. Let us define process *p* as an example.

$$p :: a\ Int \rightarrow b\ Int \rightarrow Int$$

$$p\ x\ y = x + y$$

Applying the process *p* to arguments, for example *p* 2 3, expression evaluates to 5, environment variable *a* is updated to value 2 and environment variable *b* is update to value 3. If the process is applied to a control value (), for example *p* () 4 than the process is evaluated using the current value of the environment variable *a* and variable *b* is updated to 4.

3 Execution Profiling

There are two main resources that are utilized in program and systems: computation time and memory space. Although it would be better to minimize both time and space, it is well understood that these two requirements are contradictory and it is impossible to fulfill both at the same time. Before being able to improve the efficiency of a program, a programmer must be able to 0:

- Identify execution bottlenecks of the program - parts of a program where much of time and space is used.
- Identify the cause of these bottlenecks

The potential benefits of execution profiling were first highlighted by Knuth 0. A profiler must conform two main criteria:

- must measure the distribution of the key program resources,
- measurement data must be related to the source code of a program in understandable manner.

Execution profile describes resource distribution during the program execution. Informations about resource distribution are gathered during the program execution. The profiling cycle describes the process of improving the program efficiency based on the program execution profile. The key problem of execution profiling is to relate gathered information about an execution of the program back to the source code in well defined manner. This is difficult when functional program is profiled since it provides higher level of abstractions than imperative one. Some features of a functional language, which makes program profiling more difficult than profiling an imperative program are: program transformation during compilation, polymorphism, higher-order function, lazy evaluation, a lot of simple functions within code.

4 Simple Program Profiling

Since our aim is "to compute" resource utilization at any point of computation, we define special constructs to monitor the resource utilization during the PFL program execution. In our solution programmer can associate *label* with each expression in a program as follows:

label *name e*

All resources used during the evaluation of an expression *e* are mapped to the center specified with label *name*. Using this construct programmer can concentrate on a specific part (or parts) of a program. Expression **label** *name e* is evaluated to value of *e*. Construct *label* is useful for the profiling purposes only. Of course, it is necessary to preserve the semantics of the expressions labeling during the transformation of the program when it is compiled. To be more precise, constructs for conditional profiling were incorporated into the process functional language. The first one is as follows.

label *name* e **when** e_c

If expression e_c is evaluated to value *True* of the *Bool* type, then all resources used during the evaluation of *e* are associated with label *name*. Otherwise, all used resources are attributed to the parent center. Of course, evaluation of e_c can not update the variable environment, because it is necessary to evaluate the program to the same value during the program profiling as during the program execution. On the other side, variable environment can be accessed during the evaluation of expression e_c. Fulfillment of this is rule checked during the static analysis in the compiler. Resources used during the evaluation of the conditional expression e_c are attributed to the special label *profiling* representing profiling overhead costs. All labeling inside the e_c are ignored.

It is clear, that conditional labeling is not the same as

if e_c **then label** *name e* **else** *e*

because of two main reasons:

- expression e_c is evaluated only during the profiling not the program execution,
- all resources used during the evaluation of e_c are attributed to the center with label *profiling* regardless of labeling in e_c.

Conditional profiling can enormously extend the time of profiling depending on the complexity of expressions e_c. Using conditional profiling labeled center can be dynamically activated based on decision during the execution of a program. Next example presents conditional labeling.

```
label "test" is_prime n when n > 100
```

5 Inheritance Profiles

Inheritance profiling can reduce the time spent with program profiling concentrating on smaller grains (pats of a program) than in simple profiling. Programmer can profile a part of program/function/expression depending on arguments and context. The usage of inheritance profile is explained on example. Usually the cost of function evaluation depends on arguments to which are function applied. Sometimes it is useful to consider the context of function in which it is called - parent. That is why the inheritance profiles are created.

On Figure 1 call graph of a simple program is depicted. Function *h* is called from function *f* 10 times with total cost 500 and from function *g* 20 times with total cost 100. Simple profile for the program is on Figure 2. Figure 3 presents inheritance profile for the presented call graph and function *h*. The first one is statistical profile which is generated from count and simple profile. The second one is measured accurate inheritance profile.

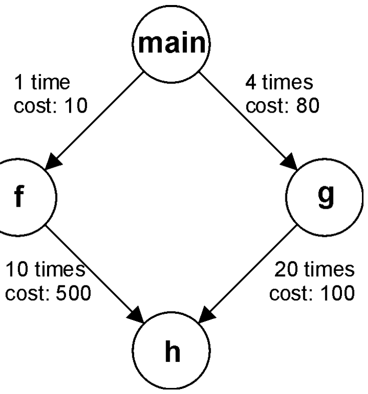

Figure 1
Call graph example

Function	Called	Cost
f	1	10
g	4	80
h	30	600

Figure 2
Simple profile

Parent→ Function	**Called Total**	**Cost Statistical Inheritance**	**Cost Accurate Inheritance**
f→h	10/30	200	500
g→h	20/30	400	100

Figure 3
Inheritance profile

In our profiler a few constructs for inheritance profile support have been implemented. The first one construct defined for conditional labeling with regard to parent context.

label *name e* **when enclosed** $name_c$

Using this construction, labeled center can be activated if parent center is same as specified. This construction can be used to create inheritance profiles. Resources used during the evaluation of expression *e* are attributed to the center with label *name* only if parent center is $name_c$. Otherwise, resources are attributed to the parent center. Next example presents usage of conditional enclosed labeling.

```
f = label "f" h 500

g = label "g" h 100

h n = label "f-h" (label "f-g" (p n)

    when enclosed "f") when enclosed "f"
```

For more flexible inheritance profiling two other constructions were defined.

label *name* **inherits** *e*

label *name* **inherits** *e* **when** e_c

Parent context are automatically added to the current labeled center. Inheritance profiling is not limited only to two levels parent/child.

Next example presents labeling for inheritance profiling of a simple process functional program.

```
f = label "f" h 500

g = label "g" h 100

h n= label "h" inherits (p n)
```

Function h can be evaluated from function f or h. Using the inheritance profiling label "h" is always connected with context of evaluation (function "f" or "g").

6 Formally Based Program Profiling

This section presents formal model of process functional program profiling. Our approach is presented on subset of PFL with constructs for profiling - simplified PFL. All PFL constructions are transformed to simplified PFL during the program compile time. This approach can be extended to all PFL language constructs. The meta-variables and categories for simplified PFL language are as follows:

$Prg \in \text{Program} \quad Def \in \text{Definition} \quad e \in \text{Expression} \quad x \in \text{V}ar$

$f \in \text{FncName} \quad \oplus \in \text{Primitive} \quad y \in \text{EnvVar} \quad C \in \text{Constructor}$

$name, label \in \text{Label}$

The meta-variables can be primed or subscripted. The syntactic category Primitive defines strict primitive operations like elementary arithmetic operations. The syntactic category Var represents variable identifiers and syntactic category EnvVar represents environment variable identifiers. The syntactic category Constructor comprises constructors of algebraic types. Primitive types, such as Int and Real, are included in the syntactic category Constructor as zero arity constructors. Syntactic category Label comprises label names. Program in simplified PFL consists of processes, functions and main expression which are evaluated during the program execution. Abstract syntax of simplified PFL is as follows.

$e ::= x$	Variable
$\mid\ f$	Function
$\mid\ e_1 \oplus e_2$	Primitive
$\mid\ y\,()$	Access
$\mid\ y\ e$	Update
$\mid\ C\ e_1 \ldots e_2$	Constructor
$\mid\ e_1\ e_2$	Aplication
$\mid\ \text{case } e \text{ of } \{C_i\ x_i \ldots x_{m_i} \rightarrow e_i\}_{i=1}^{n}$	Case
$\mid\ \text{label } name\ e$	Label
$\mid\ \text{label } name\ e \text{ when } e_c$	CondLabel
$\mid\ \text{label } name\ e \text{ when enclosed } name_c$	EncLabel
$\mid\ \text{label } name\ e \text{ inherits } e$	InhLabel
$\mid\ \text{label } name\ e \text{ when } e \text{ when } e_c$	InhCondLabel

The value v of an expression is either a lambda abstraction or a value of an algebraic type

$$
\begin{aligned}
v ::= &\ C\ v_1 \ldots v_n \\
&\mid \lambda x.e
\end{aligned}
$$

where $n \geq 0$.

The runtime state is defined by environments env_v, env_e. Environment env_f is created during the compilation. Environment env_v represents the heap for the values of lambda variables and env_e is a set of memory cells for storing values of environment variables.

$$env_f \in \mathrm{Env_f} = \mathrm{FncName} \to \mathrm{Expr}$$
$$env_v \in \mathrm{Env_v} = \mathrm{Var} \to \mathrm{Value}$$
$$env_e \in \mathrm{Env_e} = \mathrm{EnvVar} \to \mathrm{Value}$$
$$(env_v, env_e) = s \in \mathrm{State} = \mathrm{Env_v} \times \mathrm{Env_e}$$

The semantic rules for simplified PFL expressions are defined on Figure 4. Figure Figure 5 presents semantic rules for label expressions. All rules are named corresponding to abstract syntax rule names. The predicate *matches* for pattern matching and operator *extract* for extracting *i*-th item value of the structure constructed by $C\ v_1\ ...\ v_i\ ...\ v_n$ are defined as follows.

$$matches\ v\ \ C\ x_1 \ldots x_i \ldots x_n \Leftrightarrow v = C\ v_1 \ldots v_i \ldots v_n$$
$$extract\ C\ v_1 \ldots v_i \ldots v_n\ \ i = v_i, \text{where}\, 1 \leq i \leq n$$

The notation

$$env_f, label : \langle e, s \rangle \to_\mu (v, s')$$

defines that expression e is evaluated in environment env_f considering the state s and current label *label* and produces the value v, new state s' and resource environment μ. Resource environment maps label to resources used during the evaluation of labeled expression with specified label.

$$l_i \in \mathrm{Label}, \rho_i \in \mathrm{Resources}, 1 \leq i \leq n$$
$$\mu \in \mathrm{ResourceMap} = \mathrm{Label} \to \mathrm{Resources}$$
$$\mu = [l_1 \to \rho_1] \ldots [l_n \to \rho_n]$$
$$(\mu_1 \cup \mu_2)\, l = \mu_1\, l + \mu_2\, l$$

The costs of elementary operation such as variable access or function application are defined as follows.

V	cost of variable access
F	cost of closure creation
P	cost of primitive operation evaluation
E_a	cost of environment variable access
E_u	cost of environment variable access
C	cost of constructor creation
A	cost of function application

D	cost of case evaluation
L	cost of lambda abstraction evaluation

7 Implementation

Implemented process functional program profiler nowadays supports five types of profiles:

1 frequency count profile

2 time profile

3 heap profile

4 maximum requirements heap profile

5 variable access/update profile

Program profile is created during the execution using the sampling method. Execution is interrupted in specified time intervals (predefined value is 10 milliseconds) and information about used resources are collected and attributed to the current labeled center. Program profiling increases execution time approximately from 5 to 10% depending on the concrete program and labeling. Formal semantics of the execution profiling is out of the scope of this paper and can be found in 0.

$$env_f, label : \langle x, (env_v, env_e) \rangle \rightarrow_{[label \mapsto \mathrm{V}]} (env_v\, x, (env_v, env_e)) \qquad \text{Variable}$$

$$\frac{env_f, label : \langle env_f\, f, s \rangle \rightarrow_{\mu} (v, s')}{env_f, label : \langle f, s \rangle \rightarrow_{\mu \cup [label \mapsto F]} (v, s')} \qquad \text{Function}$$

$$\frac{\begin{array}{l} env_f, label : \langle e_1, s \rangle \rightarrow_{\mu_1} (v_1, s_1) \\ env_f, label : \langle e_2, s_1 \rangle \rightarrow_{\mu_2} (v_2, s_2) \end{array}}{env_f, label : \langle e_1 \oplus e_2, s \rangle \rightarrow_{\mu_1 \cup \mu_2 \cup [label \mapsto P_{\oplus}]} (v_1 \oplus v_2, s_2)} \qquad \text{Primitive}$$

$$env_f, label : \langle y\,(), (env_v, env_e) \rangle \rightarrow_{[label \mapsto \mathrm{E}_a]} (env_e\, y, (env_v, env_e)) \qquad \text{Access}$$

$$\frac{env_f, label : \langle e, (env_v, env_e) \rangle \rightarrow_{\mu} (v, (env_v', env_e'))}{env_f, label : \langle y\ e, (env_v, env_e) \rangle \rightarrow_{\mu \cup [label \mapsto \mathrm{E}_u]} (v, (env_v', env_e'[y \mapsto v]))} \qquad \text{Update}$$

$$\frac{\begin{array}{l} env_f, label : \langle e_1, s \rangle \rightarrow_{\mu_1} (v_1, s_1) \\ \ldots \\ env_f, label : \langle e_i, s_{i-1} \rangle \rightarrow_{\mu_i} (v_i, s_i) \\ \ldots \\ env_f, label : \langle e_n, s_{n-1} \rangle \rightarrow_{\mu_n} (v_n, s_n) \end{array}}{env_f, label : \langle C\, e_1 \ldots e_i \ldots e_n, s \rangle \rightarrow_{\mu_1 \cup \ldots \cup \mu_i \cup \ldots \cup \mu_n \cup [label \mapsto \mathrm{C}]} (C\, v_1 \ldots v_i \ldots v_n, s_n)} \qquad \text{Constructor}$$

$$\frac{\begin{array}{l} env_f, label : \langle e_1, s \rangle \rightarrow_{\mu_1} (\lambda x.e, s') \\ env_f, label : \langle e_2, s' \rangle \rightarrow_{\mu_2} (v_2, (env_v, env_e)) \\ env_f, label : \langle e[\overline{x}/x], (env_v[\overline{x} \mapsto v_2], env_e) \rangle \rightarrow_{\mu_3} (v, s'') \end{array}}{env_f, label : \langle e_1\, e_2, s \rangle \rightarrow_{\mu_1 \cup \mu_2 \cup \mu_3 \cup [label \mapsto \mathrm{A}]} (v, s'')} \qquad \text{Application}$$

$$\frac{\begin{array}{l} env_f, label : \langle e, s \rangle \rightarrow_{\mu} (v', (env_v', env_e')) \\ matches\ v'\ (C_j\, x_1 \ldots x_{m_j}) \\ env_f, label : \langle e_j, (env_v'[\overline{x_1} \mapsto extract\ v'\, 1] \ldots \\ \qquad \ldots [\overline{x_{m_j}} \mapsto extract\ v'\, m_j], env_e' \rightarrow_{\mu_j} (v'', s'') \end{array}}{env_f, label : \langle \mathrm{case}\ e\ \mathrm{of}\ \{C_i\, x_1 \ldots x_{m_i} \rightarrow e_i\}_{i=1}^{n}, s \rangle \rightarrow_{\mu \cup \mu_j \cup [label \mapsto \mathrm{D}]} (v'', s'')} \qquad \text{Case}$$

$$env_f, label : \langle \lambda x.e, s \rangle \rightarrow_{[label \mapsto \mathrm{L}]} (\lambda x.e, s) \qquad \text{Lambda}$$

Figure 4
The semantics of expressions

$$\frac{env_f, name : \langle e, s\rangle \rightarrow_\mu (v, s')}{env_f, label : \langle \text{label } name\ e, s\rangle \rightarrow_\mu (v, s')} \qquad \text{Label}$$

$$\frac{\begin{array}{l} env_f, \text{profiling} : \langle e_c, s\rangle \rightarrow_{\mu_c} (v_c, s_c) \\ matches\ v_c\ True \\ env_f, name : \langle e, s\rangle \rightarrow_\mu (v, s') \end{array}}{env_f, label : \langle \text{label } name\ e \text{ when } e_c, s\rangle \rightarrow_\mu (v, s')} \qquad \text{CondLabel}^{\text{tt}}$$

$$\frac{\begin{array}{l} env_f, \text{profiling} : \langle e_c, s\rangle \rightarrow_{\mu_c} (v_c, s_c) \\ matches\ v_c\ False \\ env_f, label : \langle e, s\rangle \rightarrow_\mu (v, s') \end{array}}{env_f, label : \langle \text{label } name\ e \text{ when } e_c, s\rangle \rightarrow_\mu (v, s')} \qquad \text{CondLabel}^{\text{ff}}$$

$$\frac{\begin{array}{l} name_c = label \\ env_f, name : \langle e, s\rangle \rightarrow_\mu (v, s') \end{array}}{env_f, label : \langle \text{label } name\ e \text{ when enclosed } name_c, s\rangle \rightarrow_\mu (v, s')} \qquad \text{EncLabel}^{\text{tt}}$$

$$\frac{\begin{array}{l} name_c \neq label \\ env_f, label : \langle e, s\rangle \rightarrow_\mu (v, s') \end{array}}{env_f, label : \langle \text{label } name\ e \text{ when enclosed } name_c, s\rangle \rightarrow_\mu (v, s')} \qquad \text{EncLabel}^{\text{ff}}$$

$$\frac{env_f, context(label, name) : \langle e, s\rangle \rightarrow_\mu (v, s')}{env_f, label : \langle \text{label } name \text{ inherits } e, s\rangle \rightarrow_\mu (v, s')} \qquad \text{InhLabel}$$

$$\frac{\begin{array}{l} env_f, \text{profiling} : \langle e_c, s\rangle \rightarrow_{\mu_c} (v_c, s_c) \\ matches\ v_c\ True \\ env_f, context(label, name) : \langle e, s\rangle \rightarrow_\mu (v, s') \end{array}}{env_f, label : \langle \text{label } name \text{ inherits } e \text{ when } e_c, s\rangle \rightarrow_\mu (v, s')} \qquad \text{InhCondLabel}^{\text{tt}}$$

$$\frac{\begin{array}{l} env_f, \text{profiling} : \langle e_c, s\rangle \rightarrow_{\mu_c} (v_c, s_c) \\ matches\ v_c\ False \\ env_f, label : \langle e, s\rangle \rightarrow_\mu (v, s') \end{array}}{env_f, label : \langle \text{label } name \text{ inherits } e \text{ when } e_c, s\rangle \rightarrow_\mu (v, s')} \qquad \text{InhCondLabel}^{\text{ff}}$$

Figure 5
The semantics of label expressions

The next section presents simple example with profiling outputs from the implemented profiler for process functional language. The problem solved by the program is to

1 generate prime numbers from 1 to 100 (label *prime*),

2 sum prime number from 1 to 100 (label *sum*),

3 calculate dividers of the sum (label *dividers*),

4 generate list of values from 1 to 1000 (without any label).

PFL program source code for the problem with profile labeling is as follows.

```
gen_list m n = if m > n then [] else m : gen_list (m + 1) n
can_be_divided n m = (n % m) == 0
is_prime_number n = not (foldl (or) False (map (can_be_divided n)
                         (gen_list 2 (n - 1))))
prime_numbers from to = filter (is_prime_number) (gen_list from to)
dividers n = filter (can_be_divided n) (gen_list 1 n)
main = (toUnit (label "dividers" dividers (
         label "sum" sum
           (label "prime" primeNumbers 1 100))))
  `bl` (toUnit (generateIntegerList 1 1000))
```

Time and memory profile for example program produced by the profiler is on Figure 6.

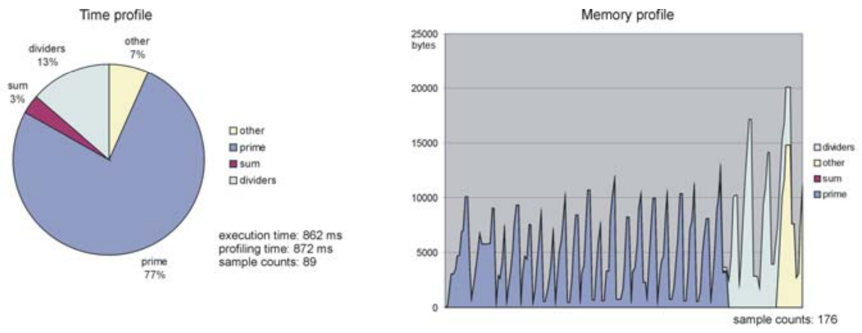

Figure 6
Time and memory profile

Conclusions

Using the execution profile of a program a programmer had to answer next two questions:

- How are resources distributed during the program execution?
- What is the effect of a particular modification of a program?

Our solution to process functional program execution profiling was presented in this paper. Using our method every expression in the PFL program can be separately profiled. The definition of profiling grains is up to the programmer. Suggested formal model can be used for reasoning about program profiling.

This work is based on our previous research of profiling and static evaluation of process functional programs 0. As a result, the static evaluation method is strongly associated with the source specification. This may help to a programmer while program development considering not just the function but also the behavior, represented by resources used. Combining execution profiling with static analysis look very promising in gathering information about resource utilization during program execution.

In the past, we have PFL-to-Java and PFL-to-Haskell generators developed. The subject of our current research is integrating aspect and process functional paradigm of programming. Our future plan is to extend profiling tools to object oriented PFL and to formal specification of a program profiling for parallel environment.

References

[1] B. W. Boehm: Improving Software Productivity. IEEE Computer 20, Vol. 9, 1987, pp. 43-57

[2] Ch. D. Clack, S. Clayman, D. Parrott: Lexical Profiling: Theory and Practice. Journal of Functional Programming Vol. 5, No. 2, 1993, pp. 225-277

[3] D. Hamlet: On subdomains: Testing, profiles, and components, Proceedings of the International Symposium on Software Testing and Analysis, Portland, Oregon, United States, August 21-24, 2000, pp.71-76

[4] J. Kollár: Partial Monadic Approach in Process Functional Language. Acta Electrotechnica et Informatica No. 1, Vol. 3, Košice, Slovak Republic, 2003, pp. 36-42

[5] J. Kollár, J. Porubän, P. Václavík, M. Vidiščák: Lazy State Evaluation of Process Functional Program. Proceding of 5th International Conference ISM 2002, Rožnov pod Radhoštem, Czech Republic, April 22-24, 2002

[6] D. E. Knuth: An Empirical Study of FORTRAN Programs. Software - Practice and Experience 1, 1971, pp. 105-133

[7] J. Porubän: Time and space profiling for process functional language. Proceeding of the 7th Scientific Conference with International Participation EMES '03, Felix Spa-Oradea, May 29-31, 2003, pp. 167-172

[8] N. Rojemo: nhc - Nearly a Haskell compiler, in Proceedings of La Wintermote, Dept of Computer Science, Chlamers University, Sweden, January 1994

[9] C. Runciman, D. Wakeling: Heap Profiling of Lazy Functional Programs. Journal of Functional Programming, Vol. 3, No. 2, pp. 217-245, 1993

[10] P. M. Sansom: Execution profiling for non-strict functional languages. Research Report FP-1994-09, Dept. of Computing Science, University of Glasgow, September 1994

[11] P. M. Sansom, S. L. Peyton Jones: Profiling lazy functional programs. Functional Programming, Glasgow 1992, Springer Verlag, Workshops in Computing, 1992

[12] P. M. Sansom, S. L. Peyton Jones: Time and space profiling for non-strict, higher-order functional languages, Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, San Francisco, California, United States, January 23-25, 1995, pp. 355-366

[13] P. M. Sansom , S. L. Peyton Jones: Formally based profiling for higher-order functional languages, ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 19, No. 2, March 1997, pp. 334-385

[14] S. Rubin, R. Bodík , T. Chilimbi: An efficient profile-analysis framework for data-layout optimizations, Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Portland, Oregon, January 16-18, 2002, pp. 140-153

[15] P. Wadler, P. Thiemann: The marriage of effects and monads. ACM Transactions on Computational Logic, Volume 4, Issue 1, 2003, pp. 1-32