Inheritance Profiles of Process Functional Programs

Jaroslav Porubän, Peter Václavík*
Department of Computers and Informatics,
Technical University of Košice, Letná 9, 042 00 Košice, Slovakia
E-mail: {jaroslav.poruban, peter.vaclavik}@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 creating of inheritance profiles. Inheritance profile can reduce the time spent with program profiling. 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: process functional language, execution profiler, inheritance profiles.

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 stateful 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 [1]. Donald Knuth found that less than 4 percent of a program usually accounts for more than 50 percent of its run time [7]. 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 [12]. Information 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.

* This work was supported by VEGA Grant No. 1/1065/04 - Specification and implementation of aspects in programming.

Once completed the performance can be profiled, and effort spent improving the program where it is necessary [11]. Profiling 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 [2],[10],[11]. Our previous work proved that all process functional programs can be easily transformed into pure functional programs using state transformers and monads. The paper presents our approach to profiling of process functional program, but it is simple to extend the approach to both imperative and functional language.

2. Process Functional Language

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 that is designed to fulfill the needs of easier state representation in a functional program [4], [6].

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 *f* as an example.

$f :: a\ Int \rightarrow b\ Int \rightarrow Int$
$f\ x\ y = x + y$

Applying the process *f* to arguments, for example *f* 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 *f* () 4 than the process is evaluated using the current value of the environment variable *a* and variable *b* is updated to 4.

3. Resources and execution profiling

There are two main resources that are utilized in program and systems: computation time and memory space.
The main approach to the code optimization is either to minimize the time or 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 [12]:

- 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 [5]. 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. Information about resource distribution are gathered during the program execution. The profiling cycle (shown on Fig. 1) describes the process of improving the program efficiency based on the program execution profile [12].

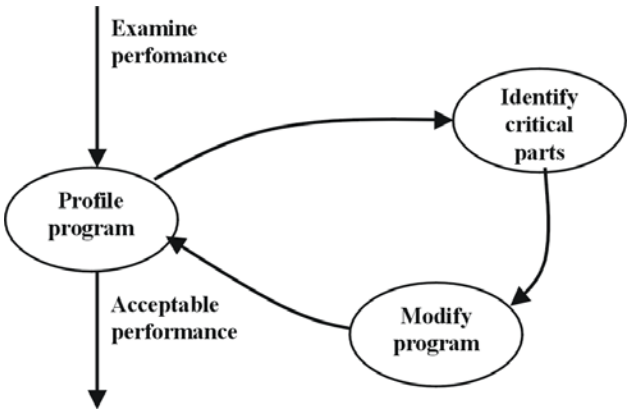

Fig. 1: The profiling cycle.

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,
- 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 label 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 label. Of course, evaluation of e_c cannot 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 the rule is checked during the static analysis in the *PFL* compiler. Resources used during the evaluation of the conditional expression e_c are attributed to the special label *profiling* representing profiling overhead costs. Labeling inside the e_c is 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 nor 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 label 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

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. Usage and meaning of inheritance profile of a program is explained on example. On Fig. 2 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.

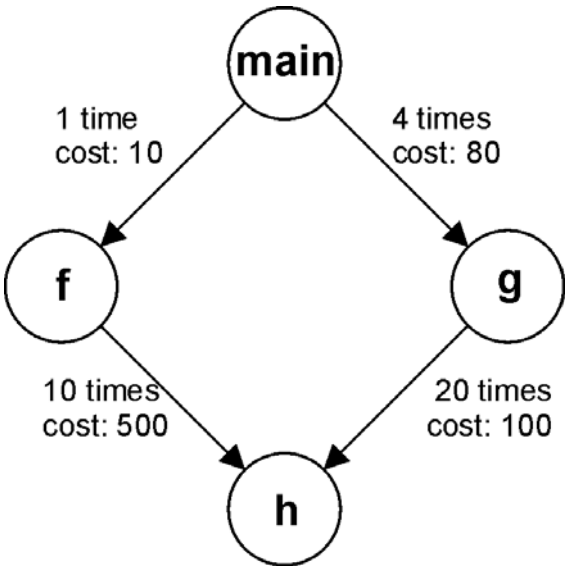

Fig. 2: Call graph example.

Simple profile for the program is in the following table on Fig. 3.

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

Fig. 3: Simple profile.

Fig. 4 presents inheritance profile for the presented call graph (Fig. 2) and function *h*. The first one is statistical profile that is generated from count and simple profile. The second one is measured accurate inheritance profile.

Parent → Function	Called/ Total	Cost Statistical	Cost Accurate
$f \rightarrow h$	10/30	200	500
$f \rightarrow g$	20/30	400	100

Fig. 4: Inheritance profiles.

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

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

Using this construction, label can be activated if parent center is same as specified. This construct 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 label is $name_c$. Otherwise, resources are attributed to the parent label.

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").

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

- frequency count profile
- time profile
- heap profile
- maximum requirements heap profile
- 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 percent depending on the concrete program and labeling. Formal semantics of the execution profiling is out of the scope of the paper and can be found in [5].

7. Conclusion

Expressing the imperative semantics by functional structure of programs in *PFL* seems to be useful for representing the resources and evaluating the behavior of programs and systems.

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.
We have already defined formal semantics of process functional program profiling. This work is based on our previous research of profiling and static evaluation of process functional programs [8].

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.

The results of this research may contribute to the field of real time and embedded systems [13], [14], system scalabity [9].

Our future plan is to extend profiling tools to object oriented *PFL* and to formal specification of a program profiling for parallel environment [3].

References

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

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

[3] Z. Horváth, Z. Hernyák, V. Zsók: *Control Language for Distributed Clean*. Acta Cybernetica 17, 2005, pp. 247-271. 2005.

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

[5] Ján Kollár, Jaroslav Porubän, Peter Václavík: *Time and Memory Profile of a Process Functional Program*. Acta Polytechnica Hungarica, Vol. 3, No. 2, 2006, pp. 27-40.

[6] Kollár Ján, Porubän Jaroslav, Václavík Peter: *From Eager PFL to Lazy Haskell*. Computing and Informatics, Vol. 25, No.1, 2006, pp. 61-80.

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

[8] Jaroslav Porubän: *Time and space profiling for process functional language.* Proceeding of the 7'th Scientific Conference with International Participation EMES '03,

Felix Spa-Oradea, May 29-31, 2003, pp. 167-172.

[9] L. Cs. Lörincz, T. Kozsik, A. Ulbert, Z. Horváth: *A method for job scheduling in Grid based on job execution status*. Multiagent and Grid Systems - An International Journal 4 (MAGS), Vol.1, No.3, 2005, pp. 197–208.

[10] Colin Runciman, David Wakeling: *Heap Profiling of Lazy Functional Programs*. Journal of Functional Programming, Vol.3, No.2, pp. 217-245, 1993.

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

[12] Patrick M. Sansom, Simon L. Peyton Jones: *Profiling lazy functional programs*. Functional Programming, Glasgow 1992, Springer Verlag, Workshops in Computing, 1992.

[13] D. Zmaranda, G. Gabor, C. Rusu: Evaluation method algorithm used to improve real-time control systems stability. Analele Universitatii din Oradea, Proc. 8-th International Conference on Engineering of Modern Electric Systems, Felix Spa-Oradea, May 26–28, University of Oradea, Romania, 2005, pp. 170–175.

[14] D. Zmaranda, G. Gabor, M. Gligor: A Framework for Modeling and Evaluating Timing Behaviour for Real-Time Systems, Proc. SINTES 12 – Int. Symposium on Systems Theory, Oct. 20–22, University of Craiova, Romania, 2005, pp. 514–520.