Toward abstract profiling

Nguyen-Minh BUI
Département d'informatique et de génie logiciel,
Université Laval, Canada
nguyen-minh.bui.1@ulaval.ca

Abstract

Profiling is a well-known technique in program analysis with many applications in compiler optimization. However, traditional profiling often requires instrumentation and execution of programs as well as many test suites. In this paper we propose a new notion: "*abstract profiling*" which is a program analysis that aims at producing similar results of traditional profiling yet with a trade-off between precision and broader applicability. Based on static analysis, this approach computes abstract program profiles without the need to modify and run the programs. Our technique proceeds in two phases. In the first phase, we use a technique in static analysis that supplies the information of control flow and types for all the expressions in the programs. In the second phase we construct a system of equations based on probability. Then we compute the abstract profile of the program by iteration. The obtained results have a similar form to that of traditional profiling: we have the abstract result and the execution frequency of each expression in the program. However, there remain several issues needed to be addressed, such as: the consistence of the system's solution, the overestimation and the disappearance of some abstract values, the effect of the initial value of variables.

1. Introduction

We usually wish to optimize our programs such as making them run faster and/or consume less memory. To optimize a program, the information about the pieces of code that we intend to optimize is very important. For example, it makes much more sense to optimize a function which is called thousands of times than one which is called once in a program. In general, profiling is a set of techniques for estimating the properties of various portions of a program at runtime, including: the amount of time spent in each function, the execution frequency of each function/piece of code, the appearance frequency of data, and so on. Moreover, these types of information can help find bugs that had otherwise been unnoticed, for instance, when we see a function executed more or less than expected or data whose appearance frequency is abnormal.

Conventionally, profiling refers to empirical measurements and so is normally performed by using dynamic analysis. A technique widely used in profiling is carried out by injecting code into a program and then executing the modified program. By recording the program behaviors and measuring the program performance, we can compute the profile of the program [5]. There are also many other well-known profilers using this technique such as gprof [3].

Static analysis and dynamic analysis are two complementary techniques to analyze a program. Static analysis does not execute the program. It examines the program's source code to find its properties that hold for all of its executions. Static analysis has many applications in software engineering such as finding potential bugs (e.g.: buffer overflow) that sometimes are impossible or very difficult to discover when using others techniques. On the other hand, dynamic analysis examines the properties of a program at runtime based on program behaviors during its execution. So the results of static analysis are usually safe, for some notion of safety, and approximate while the results of dynamic analysis are more concrete but dependent on program input [2, 1].

However, being a dynamic analysis, *dynamic* profiling has some drawbacks such as: lack of generality, dependence on test suite, etc. [6]. If profiling were to be performed statically, thanks to the nature of static analysis, we would expect the elimination or reduction of some of dynamic profiling's drawbacks.

The goal of abstract profiling is to estimate the frequency of function calls and *abstract results* for all the expressions in a program. First, one can wonder if the imprecise measures of profiling are useful and in which applications we can use them? On the other hand, there are many applications which need a more concrete analysis than traditional static analysis. Some applications can tolerate the erroneous information of profiling, such as in compiler optimization. With distorted information about the execution frequencies of functions, an optimization could be applied to rarely executed parts of a program, resulting in little or no benefit. An ill-applied optimization could also slow down a program. In either case, it does not make these programs' results wrong. We need a method that can deal with the profiling for functional languages as well as bring a broader applicability, despite less exact results. Abstract profiling is a natural idea.

This paper is organized as follows. Section 2 discusses about some related works. Section 3 presents a tiny, purely functional language used to illustrate our methodology. Section 4 contains a static analysis of control flow of functional languages that is similar to the method of Shivers [4]. It tracks not only functions but also all object types. The goal of this analysis is to get qualitative (i.e. non-numerical) results of the values that each expression in a program evaluates to and to minimize the number of the abstract values of each expression that the second phase uses. Section 5 shows our main idea of abstract profiling. We present rules to construct a system of equations that models (qualitatively) the run-time behavior of a program, we then discuss about solving the system numerically, we at last give an example and some discussions on the remaining issues needed to be addressed. Section 6 points out future work with numerous open questions. Section 7 concludes the paper.

Proceedings of the 2007 Workshop on Scheme and Functional Programming
Université Laval Technical Report DIUL-RT-0701

2. Related work

There are two properties that we wish to compute in static profiling: execution frequencies and data appearance frequencies.

The problem of computing execution frequencies for imperative languages has been studied for a long time. Ball and Larus [7] present some heuristics to predict branch direction based on program source code. Wu and Larus [8] present an algorithm to compute this property, starting with the prediction values that come from Ball and Larus's heuristics. This algorithm uses the theory of evidence in probability to calculate intra-procedural and inter-procedural block execution frequencies, local and global branch probabilities, function call and invocation frequencies.

Another approach is presented in by Wagner et. al [9]. They use the estimation of branch probabilities and Markov model of control flow to compute execution frequencies.

Pugh [10] describes a method to count the number of solutions to Presburger formulas. Using this method, we can exactly compute the execution frequency of a statement, (e.g.: the branch probability of an *if* conditional statement), within nested loops provided that all constraints are linear.

Ramalingam [11] presents a framework to compute the appearance frequencies of data for a class of data flow problem. This work, to a degree, is inspired by the lattice-theoretic framework for dataflow analysis by Kildall [12]. However, this framework requires the information of the probabilities of all edges in the control flow graph and certain conditions about the data to be computed. Our problem do not respect these conditions. In the approaches presented in [8, 9, 11], we are supposed to know some branch probabilities and the control flow of the program.

Our approach is focused on abstract profiling for functional languages. The functional languages are based on symbolic calculation, the control flow is so heavily dependent on input data and very loosely constrained that many analysis techniques for imperative languages cannot be used. Control flow and data flow in functional languages are so mutually dependent that it seems difficult for us not to compute these properties all together.

In the 0CFA [4], Shivers presents a method to compute control flows for functional languages. This approach gives safe, conservative, and qualitative results. In our approach, we would like to have quantitative results. We wish to compute not only the control flow but also the probability of each branch in the control flow, despite erroneous results. 0CFA can answer the question: Can the function F be called from a particular site? But we need to know *how often* a call site calls F. To the best of our knowledge, there is no similar solution to the problem in functional languages.

3. Language of application

We present here a language used to illustrate our methodology. The syntax of this language is similar to that of Scheme yet it is more compact, purely functional and focused on symbolic computation. In our language, a program is an expression and to run the program is to evaluate the expression.

From now on, let us suppose that we label all the expressions in a program, each syntax node corresponding to an expression has an unique label. We also drop labels when they are not necessary. We denote the set of labels by Lab.
The following is the syntax of expressions:

e	$::=$	$\#f$	constant "*false*"
	$\mid$	x	reference
	$\mid$	(λx. e)	λ - expression
	$\mid$	(e e)	function call
	$\mid$	(if e e e)	condition
	$\mid$	(μx. e)	fixed-point
	$\mid$	(cons e e)	creation of pair
	$\mid$	(car e)	extraction of the 1^{st} field of a pair
	$\mid$	(cdr e)	extraction of the 2^{nd} field of a pair
	$\mid$	(pair? e)	test whether an expression is a pair

The operational semantics of this language is of the *small-step* kind as one-step reductions are repeatedly applied to form reduction sequences. For brevity's sake, we do not present here the context of α-reduction as well as its rules. We also assume that all variables are different therefore we do not need to do α-reduction in our program. The context of β-reduction is as follows:

$$\begin{array}{rcl} C^\beta & ::= & (C^\beta\ e) \\ & \mid & (v\ C^\beta) \\ & \mid & (\text{if } C^\beta\ e\ e) \\ & \mid & (\text{cons } C^\beta\ e) \\ & \mid & (\text{cons } v\ C^\beta) \\ & \mid & (\text{car } C^\beta) \\ & \mid & (\text{cdr } C^\beta) \\ & \mid & (\text{pair? } C^\beta) \\ & \mid & [\![\cdot]\!] \end{array}$$

The following is the syntax of value v. Similarly to Scheme, our language does not have an explicit boolean value for "true". Instead, we treat all others values except the constant "false" – $\#f$ as "true".

v	$::=$	$\#f$	constant "false"
	$\mid$	(λx. e)	function
	$\mid$	(cons v v)	pair

Here are the rules of β-reduction:

$$\begin{array}{rcl} C^\beta[\![(\lambda\text{x. } e)\ v]\!] & \overset{\beta}{\mapsto} & C^\beta[\![e[\text{x} \mapsto v]]\!] \\ C^\beta[\![\text{if } \#f\ e_2\ e_3]\!] & \overset{\beta}{\mapsto} & C^\beta[\![e_3]\!] \\ C^\beta[\![\text{if } (\lambda\text{x. } e_1)\ e_2\ e_3]\!] & \overset{\beta}{\mapsto} & C^\beta[\![e_2]\!] \\ C^\beta[\![\text{if } (\text{cons } v_1\ v_2)\ e_2\ e_3]\!] & \overset{\beta}{\mapsto} & C^\beta[\![e_2]\!] \\ C^\beta[\![\mu\text{x. } e]\!] & \overset{\beta}{\mapsto} & C^\beta[\![e[\text{x} \mapsto (\mu\text{x. } e)]]\!] \\ C^\beta[\![\text{car } (\text{cons } v_1\ v_2)]\!] & \overset{\beta}{\mapsto} & C^\beta[\![v_1]\!] \\ C^\beta[\![\text{cdr } (\text{cons } v_1\ v_2)]\!] & \overset{\beta}{\mapsto} & C^\beta[\![v_2]\!] \\ C^\beta[\![\text{pair? } \#f]\!] & \overset{\beta}{\mapsto} & C^\beta[\![\#f]\!] \\ C^\beta[\![\text{pair? } (\lambda\text{x. } e)]\!] & \overset{\beta}{\mapsto} & C^\beta[\![\#f]\!] \\ C^\beta[\![\text{pair? } (\text{cons } v_1\ v_2)]\!] & \overset{\beta}{\mapsto} & C^\beta[\![\text{cons } v_1\ v_2]\!] \end{array}$$

Note that this language is strict since the arguments of a function are always evaluated completely before the function is applied. The evaluation is performed from left to right. For the sake of simplicity, we do not give here an explicit treatment of errors but it can be easily added. Our language is similar to Scheme, however, there are some differences worth noting:

- It looks strange that our language does not have an input routine. Therefore it is not suitable for use in practice. Since our goal is to do abstract profiling, we do not run the program, no input is provided to it. The interest of having a *read* function is trivial. Static profiling also does not have control of the input expressions in the program. An input expression is merely an expression with a statistical description of the possible values. We would consider to add this type of construct of input expression in the next version.

- To express infinite computations, we do not have *letrec* or *labels* forms as in Scheme, instead, we use a fixed-point operator (μ) on top of λ-calculus. More concretely, suppose that we need to evaluate the expression $e_l = (\mu_l \text{x}.\ e_{l_1})$. To evaluate e_l, we evaluate the expression e_{l_1} with the variable x is replaced by e_l. Note that, although x is replaced by e_l, we only evaluate the "later" e_l when needed. That helps the process of evaluation be able to terminate. For example, the following expression evaluates to a list of two identify functions and a $\#f$ value:

$$
\begin{array}{l}
({}_1\ (\mu_2\ \text{f}.\ (\lambda_3\ \text{L}.\ (\text{if}_4\ \text{L}_5\ (\text{cons}_6\ \ (\lambda_7\ \text{x}.\ \text{x}_8) \\
\qquad\qquad\qquad\qquad\qquad\qquad ({}_9\ \text{f}_{10}\ (\text{cdr}_{11}\ \text{L}_{12}))) \\
\qquad\qquad\qquad\qquad\quad \#f_{13}))) \\
\quad (\text{cons}_{14}\ \#f_{15}\ (\text{cons}_{16}\ \#f_{17}\ \#f_{18})))
\end{array}
$$

4. Estimation of abstract values

We need to predict the values of each expression in a program but it is inefficient to introduce variables for probability of each abstract value at each expression. Hence, it is a good idea to cut off the values that we know for sure cannot be in the results.

The goal of this phase is to get a set of the *"possible"* abstract values of each expression in a program. We construct a system of constraints that satisfy the criteria of monotonic functions in the theory of lattices, and solve the system by iteration. This is a technique in static analysis to compute static properties of a program, such as in type analysis, data flow analysis, etc. [1, 4].

The main idea of the algorithm is as follows. From the source code of a program, at each expression, according to the kind of the expression, we put constraints on the variables that hold the analysis results. Finally, we find the minimal fixed-point of the constraint system by iteration.

4.1 Definition of abstract values

First, we define the abstract values used in the results of this phase, the sub-scripts are the labels of expressions.

v	$::=$	$\#f$	
	$\mid$	λ_l	function created by expression $(\lambda_l \text{x}.\ e_{l_1})$
	$\mid$	P_l	pair created by expression $(\text{cons}_l\ e_{l_1}\ e_{l_2})$
	$\mid$	ER	when an error occurs during evaluation of the current expression.

From now on, for convenience, if we say expression e evaluates to ER that means there is an error triggered during the evaluation of e. Note that expression e also evaluates to ER when an error is triggered during the evaluation of one of its sub-expressions.

4.2 Definition of α and δ values

In the following, we define some variables used to denote the results of this phase:

- α_l is the set of the abstract values that an expression e_l could possibly evaluate to during the execution of the program.
- α_x is the set of the abstract values that a *variable x* could possibly take during the execution of a program. More concretely, α_x is the set of the values of the argument corresponding to variable x during the calls of functions or during the evaluation of μ-expressions.
- δ_l is a boolean variable indicating whether an expression e_l is evaluated or not during the execution of a program.

To facilitate the presentation, we also define the following sets:

- Val is the set of all the possible abstract values that an expression in a program can evaluate to. Val contains $\#f$, ER, the functions, and the pairs. We have $\alpha_l \subseteq Val$ for all $l \in Lab$.
- Val^v is the set of the possible abstract values that a variable can take during the execution. Val^v contains $\#f$, the functions, and the pairs. We have $\alpha_x \subseteq Val^v$ for all variables x .
- OK is the set of the abstract values indicating that the evaluation of an expression succeeds (terminates without error). OK contains $\#f$, the functions, and the pairs.
- $TRUE$ is the set of the abstract values that are treated as true by the expression *if* in our language. $TRUE$ contains the functions and the pairs.

For convenience, we say expression e evaluates to OK if e evaluates to v, and $v \in OK$, similarly for $TRUE$. The goal of the first phase is to compute all the α_l, α_x and δ_l. Note that there are many possible α_l, α_x, and δ_l for each l and x, a trivial example is $\alpha_l = Val$, $\alpha_x = Val^v$, and $\delta_l = true$, but we need the smallest possible α_l, α_x, and δ_l. (Here, for δ_l, $false$ is considered smaller than $true$)

4.3 Constraint system

We present here the rules to construct the constraint system that is used to compute the abstract results of expressions. Note that the value of ER is not passed as arguments of function calls and fixed-points. The followings are the rules to generate the constraints for different kind of expressions:

- If $e_l = \#f_l$:
  If e_l is evaluated ($\delta_l = true$), it always evaluates to $\#f$. Therefore:
  $$\delta_l \Rightarrow \alpha_l \supseteq \{\#f\}$$
- If $e_l = \text{x}_l$:
  Here the variable x is read, the value of e_l is the value of x. So we have:
  $$\delta_l \Rightarrow \alpha_l \supseteq \alpha_\text{x}$$
- If $e_l = (\lambda_l \text{x}.\ e_{l_1})$:
  - From the definition of λ_l: λ_l is the function created by the expression e_l, therefore:
    $$\delta_l \Rightarrow \alpha_l \supseteq \{\lambda_l\}$$
  - Expression e_{l_1} is executed whenever the function $(\lambda_l \text{x}.\ e_{l_1})$ is called, that is, $(\alpha_\text{x} \neq \emptyset)$. So we have the constraint:
    $$(\alpha_\text{x} \neq \emptyset) \Rightarrow \delta_{l_1}$$
- If $e_l = ({}_l e_{l_1}\ e_{l_2})$:
  - If e_l is executed then e_{l_1} and e_{l_2} are executed:
    $$\delta_l \Rightarrow \delta_{l_1}$$
    $$\delta_l \Rightarrow \delta_{l_2}$$
  - If e_{l_1} evaluates to a function, say $\lambda_{l_3} = (\lambda_{l_3} \text{x}.\ e_{l_4})$ then x will take the value of e_{l_2} as an argument to the call of the function and α_l will contain α_{l_4} as the return result of the function:
    $$\alpha_\text{x} \supseteq \alpha_{l_2}$$
    $$\alpha_l \supseteq \alpha_{l_4}$$
  - If e_{l_1} evaluates to a pair or the constant false, since we cannot make a function call on these values, the evaluation of e_l produces ER:
    $$P_i \in \alpha_{l_1} \Rightarrow ER \in \alpha_l$$
    $$\#f \in \alpha_{l_1} \Rightarrow ER \in \alpha_l$$

$$ER \in \alpha_{l_1} \Rightarrow ER \in \alpha_l$$

$$ER \in \alpha_{l_2} \Rightarrow ER \in \alpha_l$$

- If $e_l = (\mathrm{if}_l\ e_{l_1}\ e_{l_2}\ e_{l_3})$:
  - To evaluate e_l, we first evaluate e_{l_1}:

    $$\delta_l \Rightarrow \delta_{l_1}$$

  - Depending on the evaluation result of e_{l_1}, we decide to evaluate e_{l_2} or e_{l_3}. If e_{l_1} evaluates to a true value (a pair or a function) then e_{l_2} is evaluated. Otherwise if e_{l_1} evaluates to the false value ($\#f$) then e_{l_3} is evaluated.

    $$(\alpha_{l_1} - \{\#f, ER\} \neq \emptyset) \Rightarrow \delta_{l_2}$$

    $$(\#f \in \alpha_{l_1}) \Rightarrow \delta_{l_3}$$

    $$ER \in \alpha_{l_1} \Rightarrow ER \in \alpha_l$$

  - The value of the evaluation of e_l is either the value of e_{l_2} or the value of e_{l_3}:

    $$\alpha_l \supseteq \alpha_{l_2} \cup \alpha_{l_3}$$

- If $e_l = (\mu_l \mathrm{x}.\ e_{l_1})$:
  To evaluate e_l, we evaluate the expression e_{l_1} with the variable x is replaced by e_l. Note that ER cannot be passed to x. So we have:

  $$\delta_l \Rightarrow \delta_{l_1}$$
  $$\alpha_l \supseteq \alpha_{l_1}$$
  $$\alpha_{\mathrm{x}} \supseteq \{\alpha_{l_1}\} - \{ER\}$$

- If $e_l = (\mathrm{cons}_l\ e_{l_1}\ e_{l_2})$:
  If the evaluation of e_{l_1} and e_{l_2} terminates without error then e_l evaluates to a pair:

  $$\delta_l \Rightarrow \delta_{l_1}$$

  $$\delta_l \Rightarrow \delta_{l_2}$$

  $$\delta_l \Rightarrow \alpha_l \supseteq \{P_l\}$$

  $$ER \in \alpha_{l_1} \Rightarrow ER \in \alpha_l$$

  $$ER \in \alpha_{l_2} \Rightarrow ER \in \alpha_l$$

- If $e_l = (\mathrm{car}_l\ e_{l_1})$:

  $$\delta_l \Rightarrow \delta_{l_1}$$

  - If e_{l_1} evaluates to a pair then e_l evaluates to the first field of that pair:

    $\forall P_{l_2} \in \alpha_{l_1}$ such that $e_{l_2} = (\mathrm{cons}_{l_2}\ e_{l_3}\ e_{l_4})$ then $\alpha_l \supseteq \alpha_{l_3}$

    Here the errors are transferred from the *cons*-expression to the expression of extraction but it is acceptable since our analysis is still conservative.

  - If e_{l_1} evaluates to a function or the constant false then we meet an error during the evaluation of e_l:

    $$\lambda_i \in \alpha_{l_1} \Rightarrow ER \in \alpha_l$$

    $$\#f \in \alpha_{l_1} \Rightarrow ER \in \alpha_l$$

    $$ER \in \alpha_{l_1} \Rightarrow ER \in \alpha_l$$

- If $e_l = (\mathrm{cdr}_l\ e_{l_1})$:
  Similarly to the case of *car*, we have:

  $\delta_l \Rightarrow \delta_{l_1}$
  $\forall P_{l_2} \in \alpha_{l_1}$ such that $e_{l_2} = (\mathrm{cons}_{l_2}\ e_{l_3}\ e_{l_4})$ then
  $\quad \alpha_l \supseteq \alpha_{l_4}$
  $\lambda_i \in \alpha_{l_1} \Rightarrow ER \in \alpha_l$
  $\#f \in \alpha_{l_1} \Rightarrow ER \in \alpha_l$
  $ER \in \alpha_{l_1} \Rightarrow ER \in \alpha_l$

- If $e_l = (\mathrm{pair?}_l\ e_{l_1})$:

  $$\delta_l \Rightarrow \delta_{l_1}$$

  - If e_{l_1} evaluates to a pair then e_l evaluates to a true value which is the pair itself. We have no choice here as it is the operational semantic that determines how to evaluate the program and the analysis must follow.

    let $\pi = \{P_{l_2} \in \alpha_{l_1}\}$
    $\alpha_l \supseteq \pi$

  - If e_{l_1} evaluates to a function or the constant false then e_l evaluates to the false value:

    let $\pi = \{P_{l_2} \in \alpha_{l_1}\}$
    if $\alpha_{l_1} - \pi \neq \emptyset$ then $\alpha_l \supseteq \{\#f\}$
    $ER \in \alpha_{l_1} \Rightarrow ER \in \alpha_l$

- Finally, as the program starts with the evaluation of the expression e_1, we have the initial condition:

  $$\delta_1 = \mathit{true}$$

4.4 Results of the first phase

Since all the constraints are monotonic, we can use the iteration to compute the minimal fixed-point of the system. Suppose that we must compute the control flow of the following program:

$$
\begin{array}{l}
({}_1\ (\lambda_2\ \mathrm{f}.\ ({}_3\ (\lambda_4\ \mathrm{z}. \\
\qquad\qquad\qquad ({}_5\ ({}_6\ \mathrm{f}_7(\lambda_8 \mathrm{y}.\ \#f_9)) \\
\qquad\qquad\qquad\quad \#f_{10})) \\
\qquad\qquad (\mathrm{car}_{11}\ ({}_{12}\ \mathrm{f}_{13}(\mathrm{cons}_{14}\ \#f_{15}\ \#f_{16})))) \\
\quad (\lambda_{17}\mathrm{x}.\ \mathrm{x}_{18}))
\end{array}
$$

The results of the first phase (the fixed-point of the system of constraints) is presented in Figure 1.

As we observe from the results, each of the expression α_6 and α_{12} can evaluate to two abstract values, but in the concrete evaluation, they evaluate to only one value. As for expression α_{18}, it evaluates to two values during the execution but to only one value at each evaluation. Our static analysis cannot model the concept of "two distinct evaluations" of expression α_{18}. We also observe that there are many ER in the results, contrary to the reality that the errors are rare during executions of programs. This is because our analysis is rather conservative.

5. Abstract Profiling

After the first phase, we have "qualitative" results: We know the set of the abstract values that each expression or variable can produce or take, respectively, during execution but we do not know yet, among these values, which value has higher probability of appearance and which has lower one.

In the second phase, to get quantitative results, we use numerical variables to represent the measures of profiling. As mentioned above, one important thing we need to know is, for example, whether the expression e_{l_1} is evaluated more than the expression e_{l_2} during the execution of the above program? Which expression in the program is computed more than the others? Which is never computed, etc. We use the notion *execution frequency* to express that measure.

In the "traditional profiling", the profile can be considered as an *"average value"* of many executions of a program and we get the final result based on the results of a large number of test runs. We adopt a similar idea, that is, we make our programs never terminate (conceptually). We suppose that once the program terminates (normally or because of an error), it will automatically return to the beginning and restart execution.

In this phase, we model the flow of execution by probability variables. To run a functional program is to evaluate its expressions.

α_1	$\#f, ER$
α_2	λ_2
α_3	$\#f, ER$
α_4	λ_4
α_5	$\#f, ER$
α_6	λ_8, P_{14}
α_7	λ_{17}
α_8	λ_8
α_9	$\#f$
α_{10}	$\#f$
α_{11}	$\#f, ER$
α_{12}	λ_8, P_{14}
α_{13}	λ_{17}
α_{14}	P_{14}
α_{15}	$\#f$
α_{16}	$\#f$
α_{17}	λ_{17}
α_{18}	λ_8, P_{14}

δ_1	$\surd$
δ_2	$\surd$
δ_3	$\surd$
δ_4	$\surd$
δ_5	$\surd$
δ_6	$\surd$
δ_7	$\surd$
δ_8	$\surd$
δ_9	$\surd$
δ_{10}	$\surd$
δ_{11}	$\surd$
δ_{12}	$\surd$
δ_{13}	$\surd$
δ_{14}	$\surd$
δ_{15}	$\surd$
δ_{16}	$\surd$
δ_{17}	$\surd$
δ_{18}	$\surd$

α_{f}	λ_{17}
α_{z}	$\#f$
α_{y}	$\#f$
α_{x}	λ_8, P_{14}

We denote $\surd = true$.

Figure 1. The results of the first phase

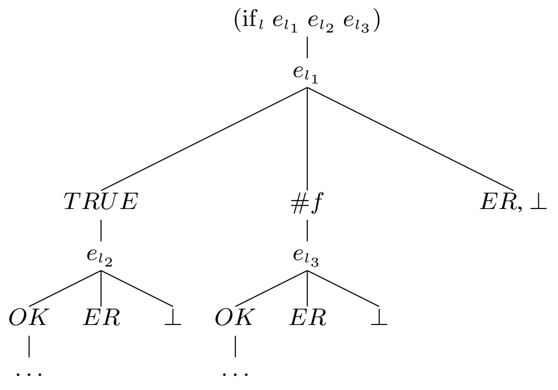

Figure 2. The evaluation of the expression $e_l = (\text{if}_l\ e_{l_1}\ e_{l_2}\ e_{l_3})$

The control flow starts from the main expression and proceeds to other expressions in the program. Supposing we are interpreting a program and we are now at the expression E_1. During the evaluation of E_1 (or just after finishing evaluating E_1, if the program does not terminate yet) there will be another expression E_2 needed to be evaluated and so forth. Therefore the process of interpreting a functional program creates a chain of expresions to be evaluated. The proportion of the appearance of an expression in the chain is its execution frequency. Since our program is cyclic (so the chain is infinite) we define that based on probability. Supposing we pick randomly an expression in the chain, then the execution frequency of an expression is the probability of the event: that expression is picked. We say it is *the next expression* to be evaluated.

In the first phase, we do not take into account the possibility that the evaluation of an expression does not terminate. In addition, in the case of the expression $e_l = (\mu_l \mathrm{x}.\ e_{l_1})$, we simply "approximate" α_x by α_{l_1} with the elimination of ER value. In the second phase, we introduce two new types of abstract value to address the above cases:

- When the evaluation of current expression e_l does not terminate, we denote the value of e_l by $\perp$.
- μ_l denotes the fixed-point created by expression $e_l = (\mu_l \mathrm{x}.\ e_{l_1})$. Due to the nature of fixed-point, variable x is now bound to μ_l.

With the introduction of new abstract types, we need to adapt the results of the first phase:

- We suppose that the evaluation of all the expressions in a program have a possibility of not terminating, therefore we add $\perp$ to all α_l.
- For the fixed-point expressions, for example $e_l = (\mu_l \mathrm{x}.\ e_{l_1})$, we set $\alpha_x = \{\mu_l\}$.

We define the following quantities:

- $\Pi_l(v)$ – the probability that e_l, when evaluated, evaluates to v ($v \in Val$).
- χ_l – the probability that e_l is the next expression to be evaluated.
- $\Pi_x(v)$ – the probability that x, when bound to a value, takes value v during a program execution, $v \in \alpha_x$.

Note that two new abstract values are now included in Val and Val^v. In the above definition of $\Pi_l(v)$, if $v = ER$ or $v = \perp$ then the event should be understood as "an error is triggered during the evaluation of the expression" and "the evaluation of the expression does not terminate", respectively.

We also define the following notational shorthands:

- $\Pi_l(S) = \sum_{v \in S} \Pi_l(v)$ where $S \subseteq Val$.
- $\Pi_x(S) = \sum_{v \in S} \Pi_x(v)$ where $S \subseteq Val^v$.

So the goal of abstract profiling is to compute all the variables $\Pi_l(v)$, $\Pi_x(v)$, and χ_l.

5.1 Equation system

Our idea here is to construct a system of equations between $\Pi_l(v)$, $\Pi_x(v)$, and χ_l then we try to find a solution of the system.

5.1.1 An example

First, we give an example to illustrate the idea. We intend to construct the constraint between our variables based on the kind of each expression in the program. To make the problem tractable, we need to make an assumption about the independence of the probability distributions. That is, the result of an expression is independent of the results of others expressions and the control flow of the program except for some particular cases. Let us consider an expression in the program, for instance:

$$e_l = (\text{if}_l\ e_{l_1}\ e_{l_2}\ e_{l_3})$$

The diagram of evaluation is presented in Figure 2. To evaluate e_l, we must first evaluate e_{l_1}. Depending on the value of e_{l_1} (constant $\#f$ or a true value), the result of e_l will be the result of e_{l_2} or e_{l_3}. From the assumption of the probability distributions, the probability $\Pi_l(v)$ ($v \in OK$) is computed based on the probabilities of the following events:

- Expression e_{l_1} evaluates to $TRUE$ and e_{l_2} evaluates to v.
- Expression e_{l_1} evaluates to $\#f$ and e_{l_3} evaluates to v.

Therefore we have the constraint that we wish to construct:

$$\Pi_l(v) = \Pi_{l_1}(TRUE)\Pi_{l_2}(v) + \Pi_{l_1}(\#f)\Pi_{l_3}(v) \text{ if } v \in OK.$$

The system we construct is neither a linear nor a monotonic system. It seems difficult to use traditional methods to find solutions.

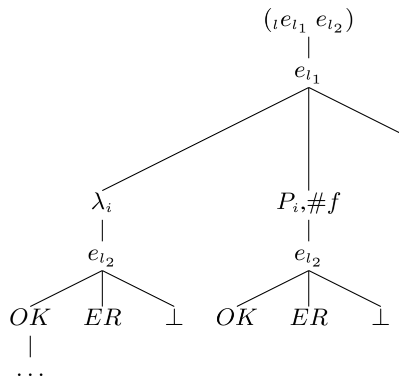

Figure 3. The evaluation of the expression $e_l = ({}_l e_{l_1}\ e_{l_2})$

We chose an iterative numerical method to find an *"approximate"* solution. In order to compute the $\Pi_l(v)$, $\Pi_x(v)$, and χ_l by iteration, we construct a system of equations that allows one to compute the values of $\Pi_l(v)$, $\Pi_x(v)$, and χ_l at step $n+1$ from their values at step n. We use the symbol ":=" (instead of "=") to imply that.

The followings are the rules to construct the system of equations from the source code of a program. For simplicity, in the following equations, the left side of the equations refers to new values (values at step $n+1$) and the right side refers to old values (values at step n):

5.1.2 Rules to compute Π_l

- For $e_l = \#f_l$:

$$\Pi_l(v) := \begin{cases} 1 & \text{if } v = \#f \\ 0 & \text{otherwise.} \end{cases} \quad \text{for } v \in \mathit{Val}.$$

  It reflects the fact that $e_l = \#f_l$ always evaluates to the value $\#f$
- For $e_l = \mathrm{x}_l$:
  Here the variable x is read. There are two different types of variables: function (λ) and fixed-point (μ). So there are two types of probability that can "contribute" to $\Pi_l(v)$:
  - One is from the functions that contributes the part $\Pi_x(v)$ to $\Pi_l(v)$.
  - The other is from the fixed-points. When x is bound to μ_l, once x is read, x will be replaced by e_l itself. So the part of the fixed points is $\sum_{\mu_i \in \mathit{Val}^v} \Pi_x(\mu_i)\Pi_i(v)$.

  And we have the equation:

$$\Pi_l(v) := \Pi_x(v) + \sum_{\mu_i \in \mathit{Val}^v} \Pi_x(\mu_i)\Pi_i(v) \quad \text{for } v \in \mathit{Val}.$$

- For $e_l = (\lambda_l \mathrm{x}.\ e_{l_1})$:
  From the definition of abstract value λ_l, we have:

$$\Pi_l(v) := \begin{cases} 1 & \text{if } v = \lambda_l \\ 0 & \text{otherwise} \end{cases} \quad \text{for } v \in \mathit{Val}.$$

- For $e_l = ({}_l e_{l_1}\ e_{l_2})$:
  The diagram illustrating the evaluation of the expression is presented in Figure 3. The evaluation of e_l is performed in three steps. First, we evaluate e_{l_1} then we evaluate e_{l_2}. If e_{l_1} evaluates to a function and e_{l_2} evaluates to OK then we continue to evaluate the body of the function. e_l evaluates to v ($v \in OK$) whenever the three conditions below are satisfied at the same time:
  - Expression e_{l_1} evaluates to a function.
  - Expression e_{l_2} evaluates to a OK value.
  - The body of the function evaluates to v.

  From the assumption of the probability distributions, the probability of the event: e_l evaluates to $v, v \in OK$ is computed as follows:

$$\Pi_l(v) := \quad \Pi_{l_2}(OK) \sum_{\lambda_{l_3} \in \mathit{Val},\ e_{l_3} = (\lambda_{l_3} \mathrm{x}.\ e_{l_4})} \Pi_{l_1}(\lambda_{l_3})\Pi_{l_4}(v)$$

  To compute $\Pi_l(ER)$, from the diagram of evaluation, we can see that there are the following cases that lead to an error in the evaluation:
  - Expression e_{l_1} evaluates to ER.
  - Expression e_{l_1} evaluates to OK and e_{l_2} evaluates to ER.
  - Expression e_{l_2} evaluates to OK and e_{l_1} evaluates to the constant $\#f$ or a pair (since we cannot make a call on these values, an error is triggered).
  - e_{l_2} evaluates to OK, e_{l_1} evaluates to a function and the body of that function evaluates to ER.

  So we have:

$$\begin{aligned}\Pi_l(ER) := {} & \Pi_{l_1}(ER) + \Pi_{l_1}(OK)\Pi_{l_2}(ER) + \\ & \Pi_{l_2}(OK)\left(\Pi_{l_1}(\#f) + \sum_{P_i \in \mathit{Val}} \Pi_{l_1}(P_i)\right) + \\ & \Pi_{l_2}(OK) \sum_{\lambda_{l_3} \in \mathit{Val},\ e_{l_3} = (\lambda_{l_3} \mathrm{x}.\ e_{l_4})} \Pi_{l_1}(\lambda_{l_3})\Pi_{l_4}(ER)\end{aligned}$$

  The probability of the event that e_l evaluates to $\perp$ is computed from the probabilities of the following events:
  - Expression e_{l_1} evaluates to $\perp$.
  - Expression e_{l_1} evaluates to OK and e_{l_2} evaluates to $\perp$.
  - Expression e_{l_2} evaluates to OK, e_{l_1} evaluates to a function and the body of that function evaluates to $\perp$.

  We have:

$$\begin{aligned}\Pi_l(\perp) := {} & \Pi_{l_1}(\perp) + \Pi_{l_1}(OK)\Pi_{l_2}(\perp) + \\ & \Pi_{l_2}(OK) \sum_{\lambda_{l_3} \in \mathit{Val},\ e_{l_3} = (\lambda_{l_3} \mathrm{x}.\ e_{l_4})} \Pi_{l_1}(\lambda_{l_3})\Pi_{l_4}(\perp)\end{aligned}$$

- For $e_l = (\mathrm{if}_l\ e_{l_1}\ e_{l_2}\ e_{l_3})$:
  As above discussed, we have the following equations:

$$\Pi_l(v) := \Pi_{l_1}(\mathit{TRUE})\Pi_{l_2}(v) + \Pi_{l_1}(\#f)\Pi_{l_3}(v) \text{ if } v \in OK.$$

$$\Pi_l(ER) := \Pi_{l_1}(ER) + \Pi_{l_1}(\mathit{TRUE})\Pi_{l_2}(ER) + \Pi_{l_1}(\#f)\Pi_{l_3}(ER)$$

$$\Pi_l(\perp) := \Pi_{l_1}(\perp) + \Pi_{l_1}(\mathit{TRUE})\Pi_{l_2}(\perp) + \Pi_{l_1}(\#f)\Pi_{l_3}(\perp)$$

- For $e_l = (\mu_l \mathrm{x}.\ e_{l_1})$:
  From the definition of fixed-point, we have:

$$\Pi_l(v) := \Pi_{l_1}(v) \text{ for } v \in \mathit{Val}$$

- For $e_l = (\mathrm{cons}_l\ e_{l_1}\ e_{l_2})$:
  The diagram of the evaluation of this expression is presented in Figure 4. Expression e_l evaluates to a pair whenever e_{l_1} and e_{l_2} are evaluated without error:

$$\Pi_l(P_l) := \Pi_{l_1}(OK)\Pi_{l_2}(OK)$$

$$\Pi_l(v) := 0 \text{ if } v \in OK \setminus \{P_l\}$$

$$\Pi_l(v) := \Pi_{l_1}(v) + \Pi_{l_2}(OK)\Pi_{l_2}(v) \text{ if } v \in \{ER, \perp\}$$

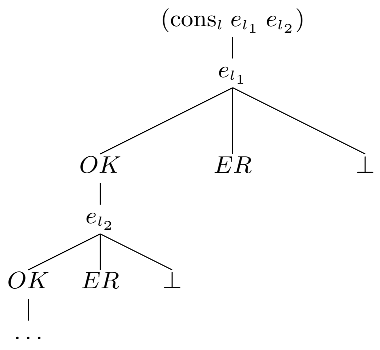

Figure 4. The evaluation of the expression $e_l = (\text{cons}_l\ e_{l_1}\ e_{l_2})$

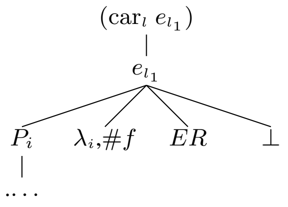

Figure 5. The evaluation of the expression $e_l = (\text{car}_l\ e_{l_1})$

- For $e_l = (\text{car}_l\ e_{l_1})$:
 The diagram of the evaluation of this expression is presented in Figure 5. In the case of *car*, this functions extracts the first field of e_{l_1} if e_{l_1} is a pair. The value of $\Pi_l(v)$, $(v \in OK)$ is computed based on probability of the following events:
  - Expression e_{l_1} evaluates to a pair, for example: $P_{l_2} \in Val,\ e_{l_2} = (\text{cons}_{l_2}\ e_{l_3}\ e_{l_4})$
  - The first field of that pair contains to v.

 The second probability is computed with the hypothesis that the expression e_{l_2} evaluates to a pair. We use the conditional probability to compute that:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

 Here:
  - A is the event: e_{l_3} evaluates to v $(v \in OK)$.
  - B is the event: e_{l_2} evaluates to a pair.
  - The probability of the event B is $\Pi_{l_3}(OK)\Pi_{l_4}(OK)$.
  - The probability of the event $A \cap B$ is $\Pi_{l_3}(v)\Pi_{l_4}(OK)$

 So we have:
 $\Pi_l(v) :=$

$$\sum_{P_{l_2} \in Val,\ e_{l_2} = (\text{cons}_{l_2}\ e_{l_3}\ e_{l_4})} \Pi_{l_1}(P_{l_2}) \frac{\Pi_{l_3}(v)\Pi_{l_4}(OK)}{\Pi_{l_3}(OK)\Pi_{l_4}(OK)}$$

 Also note that, in the above formula, if $\Pi_{l_4}(OK) = 0$, we consider that the value of the fraction equals zero. The values of $\Pi_l(ER)$ and $\Pi_l(\perp)$ are computed as follows:

$$\Pi_l(ER) := \Pi_{l_1}(ER) + \Pi_{l_1}(\#f) + \sum_{\lambda_i \in Val} \Pi_{l_1}(\lambda_i)$$

$$\Pi_l(\perp) := \Pi_{l_1}(\perp)$$

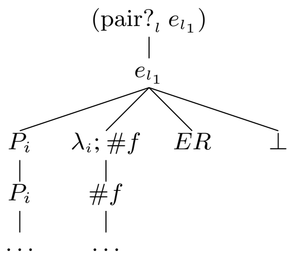

Figure 6. The evaluation of the expression $e_l = (\text{pair?}_l\ e_{l_1})$

- For $e_l = (\text{cdr}_l\ e_{l_1})$:
 Similarly to the case of *car* we have:

 $\Pi_l(v) :=$

$$\sum_{P_{l_2} \in Val,\ e_{l_2} = (\text{cons}_{l_2}\ e_{l_3}\ e_{l_4})} \Pi_{l_1}(P_{l_2}) \frac{\Pi_{l_4}(v)\Pi_{l_3}(OK)}{\Pi_{l_3}(OK)\Pi_{l_4}(OK)}$$

 if $v \in OK$

$$\Pi_l(ER) := \Pi_{l_1}(ER) + \Pi_{l_1}(\#f) + \sum_{\lambda_i \in Val} \Pi_{l_1}(\lambda_i)$$

$$\Pi_l(\perp) := \Pi_{l_1}(\perp)$$

- For $e_l = (\text{pair?}_l\ e_{l_1})$:
 The diagram of the evaluation of this expression is presented in Figure 6. Expression e_l evaluates to true value whenever e_{l_1} is a pair.

$$\Pi_l(\#f) := \Pi_{l_1}(\#f) + \sum_{\lambda_i \in Val} \Pi_{l_1}(\lambda_i)$$

$$\Pi_l(\lambda_i) := 0\ \forall i \in Lab$$

$$\Pi_l(v) := \Pi_{l_1}(v) \text{ if } v \in \{ER, \perp\} \cup \{P_i | i \in Lab\}$$

5.1.3 Rules to compute Π_x

There are two different types of variables: function (λ) and fixed-point (μ). Each type has a different rule to compute Π_x.

- For x in $(\lambda_l x.\ e_{l_1})$, we compute Π_x by examining all function calls in the program:

$$\Pi_x(v) := \frac{\displaystyle\sum_{l_2 \in Lab,\ e_{l_2} = (_{l_2} e_{l_3}\ e_{l_4})} \chi_{l_2}\Pi_{l_3}(\lambda_l)\Pi_{l_4}(v)}{\displaystyle\sum_{l_2 \in Lab,\ e_{l_2} = (_{l_2} e_{l_3}\ e_{l_4})} \chi_{l_2}\Pi_{l_3}(\lambda_l)\Pi_{l_4}(OK)}$$

 if $v \in Val^v$
 The numerator refers to the probability that x, *among all variables*, takes value v during a program execution. The denominator refers to the probability that x takes a value in the set of abstract values OK. These probabilities are computed in the probability space for all variables and all values.

- For x in $(\mu_l \text{x}.\ e_{l_1})$, from the definition of μ, we have:

$$\Pi_x(v) := \begin{cases} 1 & \text{if } v = \mu_l \\ 0 & \text{otherwise} \end{cases} \quad \text{if } v \in \mathit{Val}^v$$

5.1.4 Rules to compute χ_l

We construct the rules based on the relation of the expressions which are the next expression to be evaluated during execution of the program. For example, supposing that the expression $e_l = (\text{cons}_l\ e_{l_1}\ e_{l_2})$ is evaluated. Then we know for sure that the next expression to be evaluated is e_{l_1}, therefore:

$$\chi_{l_1} = \chi_l$$

We also know that, after the evaluation of e_{l_1}, if this evaluation succeeds, e_{l_2} will be the next expression to be evaluated, therefore:

$$\chi_{l_2} = \chi_l \Pi_{l_1}(OK)$$

The detailed rules are presented as follows:

- For $e_l = ({}_l e_{l_1}\ e_{l_2})$:
  If e_l is executed with the probability p then e_{l_1} will be executed with the same probability. Expression e_{l_2} is executed if and only if e_{l_1} is executed and the evaluation of e_{l_1} succeeds. We have:
  $$\chi_{l_1} := \chi_l$$
  $$\chi_{l_2} := \chi_l \Pi_{l_1}(OK)$$
- For $e_l = (\text{if}_l\ e_{l_1}\ e_{l_2}\ e_{l_3})$:
  Once the expression e_l is executed then e_{l_1} is executed. The result of the evaluation of e_{l_1} decides which expression will be the next to be evaluated: e_{l_2} or e_{l_3}. We have:
  $$\chi_{l_1} := \chi_l$$
  $$\chi_{l_2} := \chi_l \Pi_{l_1}(\mathit{TRUE})$$
  $$\chi_{l_3} := \chi_l \Pi_{l_1}(\#f)$$
- For $e_l = (\text{cons}_l\ e_{l_1}\ e_{l_2})$:
  Expression e_{l_2} is evaluated whenever the evaluation of e_{l_1} terminates without error. We have:
  $$\chi_{l_1} := \chi_l$$
  $$\chi_{l_2} := \chi_l \Pi_{l_1}(OK)$$
- For $e_l = (\text{car}_l\ e_{l_1})$ or $e_l = (\text{cdr}_l\ e_{l_1})$ or $e_l = (\text{car}_l\ e_{l_1})$:
  We have:
  $$\chi_{l_1} = \chi_l$$
  because e_{l_1} will be evaluated once e_l is evaluated.
- The main expression e_1 is a special case. As we assumed, once a program terminates (either normally or due to an error during execution), it will return to the beginning and restart execution, the value of χ_1 must be computed based on the values of the previous $\chi_1, \Pi_1(OK)$ as well as the probability of the event: other expressions in the program trigger ER:
  $$\begin{aligned}\chi_1 := {} & \chi_1 \Pi_1(OK) + \\ & \sum_{l \in Lab; e_l = ({}_l e_{l_1}\ e_{l_2})} \chi_l \Pi_{l_2}(OK)(\Pi_{l_1}(\#f) + \sum_{P_i \in Val} \Pi_{l_1}(P_i)) + \\ & \sum_{l \in Lab; e_l = (\text{car}_l\ e_{l_1}) \text{ or } e_l = (\text{cdr}_l\ e_{l_1})} \chi_l (\Pi_{l_1}(\#f) + \sum_{\lambda_i \in Val} \Pi_{l_1}(\lambda_i))\end{aligned}$$
  The first term is the probability that expression e_1 evaluates to OK (that is, the program terminates normally). The second is the probability that an error is triggered during function calls. The last one is the probability that an error is triggered during evaluation of *car* and *cdr* expressions.

The expression $e_l = (\lambda_l \text{x}.\ e_{l_1})$ and $e_l = (\mu_l \text{x}.\ e_{l_1})$ are other special cases where the normal flow of control ends. We must examine the entire program because the flow of control is distributed to many expressions in the program via function calls and references of fixed-point.

- For $e_l = (\lambda_l \text{x}.\ e_{l_1})$:
  e_{l_1} is executed when the function λ_l is called from some call site $e_{l_2} = ({}_{l_2} e_{l_3}\ e_{l_4})$ when the first child e_{l_3} evaluates to λ_l and the second child e_{l_4} is evaluated successfully. Therefore, to compute χ_{l_1}, we must compute this probability on all the function calls of the program and make a sum of all of these probabilities.
  $$\chi_{l_1} := \sum_{l_2 \in Lab,\ e_{l_2} = ({}_{l_2} e_{l_3}\ e_{l_4})} \chi_{l_2} \Pi_{l_3}(\lambda_l) \Pi_{l_4}(OK)$$
- For $e_l = (\mu_l \text{x}.\ e_{l_1})$:
  e_{l_1} is executed if e_l is executed or the variable x is read and evaluates to μ_l (and starts the process of the evaluation of e_l). So we have:
  $$\chi_{l_1} := \chi_l + \sum_{l_2 \in Lab; e_{l_2} = x_{l_2}} \chi_{l_2} \Pi_x(\mu_l)$$

5.2 The invariants of the system

We show here some invariants of the above system. Note that, since we construct the system based on the distribution of probability, by definition these invariants must be always satisfied. When we compute the solution by iteration, we always have:

$$\sum_{v \in Val} \Pi_l(v) = 1 \text{ for each label } l$$

$$\sum_{v \in Val^v} \Pi_x(v) = 1 \text{ for each variable } x$$

provided that when we initialize the Π_l and Π_x, the above sums are satisfied. In other words, $\sum_{v \in Val} \Pi_l(v)$ and $\sum_{v \in Val^v} \Pi_x(v)$ are unchanged over iterations.

In the case of χ, in our current model, the sum $\sum_{l \in Lab} \chi_l$ varies over iterations. Therefore, after each iteration, we must re-normalize χ so that $\sum_{l \in Lab} \chi_l$ always equals 1. So the above rules is to compute the "raw" χ_l at step $(n+1)$ from the normalized χ_l at step n. The raw χ_l then are re-normalized as follows:

$$\chi_{l,\text{normalized}} = \frac{\chi_{l,\text{raw}}}{\sum_{i \in Lab} \chi_{i,\text{raw}}}$$

5.2.1 Initialization of variables

Thanks to the results of the first phase, we can reduce a number of variables of the system. That is, all the following variables have the value of zero therefore they can be excluded from the system:

- $\Pi_l(v) = 0$ for $v \notin \alpha_l$
- $\Pi_x(v) = 0$ for $v \notin \alpha_x$
- $\chi_l = 0$ if $\delta_l = \mathit{false}$

Since we have little information about the variables, we can initialize the $\Pi_l(v)$ and χ_l with any value provided that the sum condition holds. A simple choice is to set all the variables to the same value:

$$\Pi_l(v) = \begin{cases} \dfrac{1}{\|\alpha_l\|} & \text{for } v \in \alpha_l \\ 0 & \text{for } v \in \mathit{Val} \text{ but } v \notin \alpha_l \end{cases}$$

$$\Pi_x(v) = \begin{cases} \dfrac{1}{\|\alpha_x\|} & \text{for } v \in \alpha_x \\ 0 & \text{for } v \in \mathit{Val}^v \text{ but } v \notin \alpha_{x^+} \end{cases}$$

$$\chi_l = \frac{1}{\|\{\delta_i, i \in \mathit{Lab}, \delta_i = \mathit{true}\}\|}$$

That is, all the abstract values of each expression (after first phase) have the same probability and all the expressions in the program have the same execution frequency.

We can also initialize the variables with random values. In some cases, the final results do not depend on initial values.

5.3 Example

We continue with our example, supposing that we compute the abstract profiling of following the program:

$(_1\ (\lambda_2\ \text{f}.\ (_3\ (\lambda_4\ \text{z}.$
$\quad\quad\quad\quad (_5\ (_6\ \text{f}_7(\lambda_8\text{y}.\ \#f_9))$
$\quad\quad\quad\quad\quad \#f_{10}))$
$\quad\quad (\text{car}_{11}\ (_{12}\ \text{f}_{13}(\text{cons}_{14}\ \#f_{15}\ \#f_{16}))))$
$\quad (\lambda_{17}\text{x}.\ \text{x}_{18}))$

After the first phase we have the set of the abstract values that each expression in the program possibly evaluates to during execution. We model only these objects. We choose random initial values of probability on condition that the invariants of the system are satisfied. (That is, $\sum_{v \in \mathit{Val}} \Pi_l(v) = 1$, $\sum_{v \in \mathit{Val}^v} \Pi_x(v) = 1$,$\sum_{v \in \mathit{Val}^v} \chi_v = 1$).

The initial values are presented in Figure 7. The result after 200 iterations is presented in Figure 8. The "ideal" result comes from the (concrete) interpretation of the program is presented in Figure 9, it also can be considered as the results of dynamic profiling.

Comparing the two results, we have some comments:

- Thanks to results of the first phase, we have cut off many "impossible" values. Each expression possibly evaluates to maximum three abstract values.
- Each of the expressions e_1 and e_{14} evaluates to only one value in reality but it has three values in the result of the abstract profiling. The analysis in the first phase was not able to eliminate the fake values.
- We correctly identify the expression e_{18} is most executed but give wrong result that e_9 is least executed.
- We observed in our experiments that all $\Pi(\perp)$ decrease to zero after a few iterations. It is because we do not have any expression which can contribute to the $\Pi(\perp)$ after each iteration. Another model of system of equations can address this problem.
- We also observed in our experiments that the value of ER are overestimated as our system proposes a too coarse approximation of the values and does not take the execution context into account. A finer approximation or some heuristics may improve the situation.

5.4 Discussion

The system we construct is not a linear system. That makes it hard to solve the system by simple algebra methods. Our idea is not to solve the system by a rigorous way but to model it by probability variables. We hope that, these variables, which vary during the interations but follow the rules, will converge to an exact or near exact result. It seems difficult to prove rigorously that we can always find the fixed-point of the system by using iteration.

$\Pi_1(\#f)$	0.755
$\Pi_1(ER)$	0.080
$\Pi_1(\perp)$	0.165
$\Pi_2(\lambda_2)$	1.0
$\Pi_3(\#f)$	0.310
$\Pi_3(ER)$	0.341
$\Pi_3(\perp)$	0.349
$\Pi_4(\lambda_4)$	1.0
$\Pi_5(\#f)$	0.462
$\Pi_5(ER)$	0.096
$\Pi_5(\perp)$	0.442
$\Pi_6(\lambda_8)$	0.050
$\Pi_6(P_{14})$	0.539
$\Pi_6(\perp)$	0.411
$\Pi_7(\lambda_7)$	0.613
$\Pi_7(\perp)$	0.387
$\Pi_8(\lambda_8)$	1.0
$\Pi_9(\#f)$	1.0
$\Pi_{10}(\#f)$	1.0
$\Pi_{11}(\#f)$	0.553
$\Pi_{11}(ER)$	0.431
$\Pi_{11}(\perp)$	0.016
$\Pi_{12}(\lambda_8)$	0.394
$\Pi_{12}(P_{14})$	0.540
$\Pi_{12}(\perp)$	0.066
$\Pi_{13}(\lambda_{17})$	0.167
$\Pi_{13}(\perp)$	0.833
$\Pi_{14}(P_{14})$	0.931
$\Pi_{14}(\perp)$	0.069
$\Pi_{15}(\#f)$	1.0
$\Pi_{16}(\#f)$	1.0
$\Pi_{17}(\lambda_{17})$	1.0
$\Pi_{18}(\lambda_8)$	0.064
$\Pi_{18}(P_{14})$	0.626
$\Pi_{18}(\perp)$	0.310

$\Pi_f(\lambda_9)$	1.0
$\Pi_z(\#f)$	1.0
$\Pi_y(\#f)$	1.0
$\Pi_x(\lambda_8)$	0.809
$\Pi_x(P_{14})$	0.191

χ_1	0.045
χ_2	0.036
χ_3	0.043
χ_4	0.005
χ_5	0.102
χ_6	0.019
χ_7	0.090
χ_8	0.123
χ_9	0.011
χ_{10}	0.077
χ_{11}	0.056
χ_{12}	0.029
χ_{13}	0.061
χ_{14}	0.014
χ_{15}	0.118
χ_{16}	0.012
χ_{17}	0.094
χ_{18}	0.057

Figure 7. The initial values of Π_l, Π_x, and χ

Morever, there exists some systems that have several fixed-points. Nevertheless, since all the variables are tied with the invariants of the system, it is likely that there exist two following cases in practice:

- Our system converges to a fixed-point. This occurs in a few simple programs.
- The system does not converge but it creates a "loop". That is, after certain iterations, the new χ are just a permutation of the old χ in previous iterations.

To address this problem, after the calculation of the variables at each iteration, we apply a technique in machine learning: use a *leaning rate*. The result at step $n+1$ is computed from the result at step n and the *"raw"* result at step $n+1$:

$$\chi_l^{(n+1)} = \chi_l^{(n)}(1-\phi) + \chi_{l,\text{raw}}^{(n+1)}\phi$$

ϕ is the learning rate, $0 < \phi < 1$.

So it should be noted that, in the above rules, the left sides are the "raw" χ_l at step $(n+1)$ (before applying learning rate) and the right sides refer to the χ_l at step n (after applying learning rate). The above example is computed with $\phi = 0.5$, small ϕ makes the system more stable but it takes more time to converge. However, we are not able to prove that using the learning rate can always make the system converge.

$\Pi_1(\#f)$ $\Pi_1(ER)$ $\Pi_1(\bot)$	0.236 0.764 0.0
$\Pi_2(\lambda_2)$	1.0
$\Pi_3(\#f)$ $\Pi_3(ER)$ $\Pi_3(\bot)$	0.236 0.764 0.0
$\Pi_4(\lambda_4)$	1.0
$\Pi_5(\#f)$ $\Pi_5(ER)$ $\Pi_5(\bot)$	0.382 0.618 0.0
$\Pi_6(\lambda_8)$ $\Pi_6(P_{14})$ $\Pi_6(\bot)$	0.382 0.618 0.0
$\Pi_7(\lambda_7)$ $\Pi_7(\bot)$	1.0 0.0
$\Pi_8(\lambda_8)$	1.0
$\Pi_9(\#f)$	1.0
$\Pi_{10}(\#f)$	1.0
$\Pi_{11}(\#f)$ $\Pi_{11}(ER)$ $\Pi_{11}(\bot)$	0.382 0.618 0.0
$\Pi_{12}(\lambda_8)$ $\Pi_{12}(P_{14})$ $\Pi_{12}(\bot)$	0.382 0.618 0.0
$\Pi_{13}(\lambda_{17})$ $\Pi_{13}(\bot)$	1.0 0.0
$\Pi_{14}(P_{14})$ $\Pi_{14}(\bot)$	1.0 0.0
$\Pi_{15}(\#f)$	1.0
$\Pi_{16}(\#f)$	1.0
$\Pi_{17}(\lambda_{17})$	1.0
$\Pi_{18}(\lambda_8)$ $\Pi_{18}(P_{14})$ $\Pi_{18}(\bot)$	0.382 0.618 0.0

$\Pi_f(\lambda_9)$	1.0
$\Pi_z(\#f)$	1.0
$\Pi_y(\#f)$	1.0
$\Pi_x(\lambda_8)$ $\Pi_x(P_{14})$	0.382 0.618

χ_1	0.063
χ_2	0.063
χ_3	0.063
χ_4	0.063
χ_5	0.039
χ_6	0.039
χ_7	0.039
χ_8	0.039
χ_9	0.015
χ_{10}	0.039
χ_{11}	0.063
χ_{12}	0.063
χ_{13}	0.063
χ_{14}	0.063
χ_{15}	0.063
χ_{16}	0.063
χ_{17}	0.063
χ_{18}	0.101

Figure 8. The values of Π_l, Π_x, and χ after 200 iterations

$\Pi_1(\#f)$	1.0
$\Pi_2(\lambda_2)$	1.0
$\Pi_3(\#f)$	1.0
$\Pi_4(\lambda_4)$	1.0
$\Pi_5(\#f)$	1.0
$\Pi_6(\lambda_8)$	1.0
$\Pi_7(\lambda_7)$	1.0
$\Pi_8(\lambda_8)$	1.0
$\Pi_9(\#f)$	1.0
$\Pi_{10}(\#f)$	1.0
$\Pi_{11}(\#f)$	1.0
$\Pi_{12}(P_{14})$	1.0
$\Pi_{13}(\lambda_{17})$	1.0
$\Pi_{14}(P_{14})$	1.0
$\Pi_{15}(\#f)$	1.0
$\Pi_{16}(\#f)$	1.0
$\Pi_{17}(\lambda_{17})$	1.0
$\Pi_{18}(\lambda_8)$ $\Pi_{18}(P_{14})$	0.5 0.5

$\Pi_f(\lambda_9)$	1.0
$\Pi_z(\#f)$	1.0
$\Pi_y(\#f)$	1.0
$\Pi_x(\lambda_8)$ $\Pi_x(P_{14})$	0.5 0.5

χ_1	0.053
χ_2	0.053
χ_3	0.053
χ_4	0.053
χ_5	0.053
χ_6	0.053
χ_7	0.053
χ_8	0.053
χ_9	0.053
χ_{10}	0.053
χ_{11}	0.053
χ_{12}	0.053
χ_{13}	0.053
χ_{14}	0.053
χ_{15}	0.053
χ_{16}	0.053
χ_{17}	0.053
χ_{18}	0.106

Figure 9. The "ideal" result of Π_l, Π_x, and χ

The effect of the initial values of the variables to the system is an issue that has not been solved yet. In practice, in some cases, we can initialize with random values and consistently get the same result.

As we use a numerical method, the stability of the system is also a problem. In some constraints we compute the probability by some divisions that could lead to a "division by zero" error.

6. Future work

Since our work is only in a preliminary state, there remain many open questions. Our future work will focus on the following issues:

- Clarify the consistence of the system, the connection between operational semantic, the equations of abstract profiling, and the control-flow analysis.
- As we construct our model based on the distribution of probability and the control-flow analysis, we define χ_l based on the probability of the appearance of e_l as the next expression to be evaluated in the chain of execution, we need to clarify if this probability always exists for all programs.
- We can improve the system by using the models of probabilistic inference that make fewer independence assumptions such as Bayesian inference in graphical models.
- We also can consider to change the syntax of our language so that it has a more natural link with the equations of profiling.
- In this model, we must re-normalized the χ after each iteration because the sum of χ is not preserved. In the next paper we intend to develop a solution that can preserve the sum of χ.

For the system of equations, we need to address the following issues:

- Investigate the consistency of the system of equations.
- We also need to study the accuracy and the numeric stability of the system.
- Study the effect of the initial values of variables. We can use some heuristics to predict and initialize the variables base on the semantic context so that our system converges faster and has a better result?
- In this example we just find one "approximate" solution of the system, but it is likely that the system will have more than one solution. If possible we can find and compare these solutions.
- The iterative method is not the only method that can solve the system of equations, there exist many other methods to do so. It would be interesting to measure their performances.
- Our modeling that we have proposed here is still coarse. A finer modeling of abstract values and context of evaluation can help improve the quality of profiling and maybe address the problems of overestimating the values of ER as well as the disappearance of $\Pi(\bot)$.

7. Conclusions

We have proposed a methodology to compute abstract profiling. This methodology aims at computing a "similar" results of traditional profiling yet without having to execute programs. This methodology is applied to the functional languages and can be considered as a static approach to the profiling.

In the first phase of our algorithm. we use a control flow analysis to get qualitative results and to limit the numbers of numerical variables in the second phase. In the second phase, we propose a methodology based on probability to model the control flow of programs. We use numerical variables to represent the measurements of abstract profiling. We also present a set of rules that can be used to construct an equation system that tie these variables together. Finally, we use iteration to compute an “approximate” solution of the system.

The approach used to illustrate the methodology is just one among many approaches that can be proposed and it still has many drawbacks. After all, since our work is at a very preliminary stage, there are still numerous issues that need to be addressed.

Acknowledgments

I wish to thank professor Danny Dubé for his advice when carefully reviewing this paper and the anonymous referees for their helpful comments. This work has been funded by Natural Sciences and Engineering Research Council of Canada.

References

[1] Flemming Nielson, Hanne R. Nielson, Chris Hankin. Principles of Program Analysis. Springer, 1999.

[2] Thomas Ball. The Concept of Dynamic Analysis. *Foundation of Software Engineering,* p. 216-234, 1999.

[3] gprof documentation.
`http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html`

[4] Olin Shivers. Control-flow analysis in Scheme. *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation.* 1988.

[5] Thomas Ball, James R. Larus. Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems, 16(4),* p. 1319-1360, 1994.

[6] Michael Ernst. Static and dynamic analysis: synergy and duality. *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering.* 2004.

[7] Thomas Ball , James R. Larus, Branch prediction for free. *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, p.300-313, 1993,

[8] Youfeng Wu, James R. Larus, Static branch frequency and program profile analysis. *Proceedings of the 27th annual international symposium on Microarchitecture*, p.1-11, 1994.

[9] Tim A. Wagner, Vance Maverick, Susan L. Graham, Michael A. Harrison, Accurate static estimators for program optimization. *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, p.85-96, 1994.

[10] William Pugh, Counting solutions to Presburger formulas: How and Why. *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, p.121-134, 1994.

[11] G. Ramalingam, Data flow frequency analysis. *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, p.267-277, 1996.

[12] Gary A. Kildall, A unified approach to global program optimization. *Proceedings of the ACM SIGACT-SIGPLAN 1973 Symposium on Principles of Programming Languages*, p.194-206, 1973.