periodica polytechnica

Electrical Engineering
55/3-4 (2011) 133–138
doi: 10.3311/pp.ee.2011-3-4.06
web: http://www.pp.bme.hu/ee

RESEARCH ARTICLE

Building dependency graph for slicing erlang programs

Melinda Tóth / *István* Bozó

Received 2012-07-09

Abstract

Program slicing is a well-known technique that utilizes dependency graphs and static program analysis. Our goal is to perform impact analysis of Erlang programs based on the resulted program slices, that is we want to measure the impact of any change made on the source code: especially we want to select a subset of test cases which must be rerun after the modification. However impact analyzer tools exist for object oriented languages, the used dependency graphs heavily depend on the syntax and semantics of the used programming language, thus we introduce dependency graphs for a dynamically typed functional programming language, Erlang.

Keywords

Program slicing · Erlang program · Dependency graph

Acknowledgement

Supported by TECH 08 A2-SZOMIN08, ELTE IKKK, and Ericsson Hungary.

Melinda Tóth
István Bozó
Department of Information Systems, Eötvös Loránd University, H-1117 Budapest, Pázmány Péter sétány 1/C, Hungary

1 Introduction

Program slicing [14] is the most well-known method used to perform impact analysis. Different methods are available to perform program slicing (e.g. dataflow equations, information flow relations, dependency graphs), but the most popular techniques are based on dependency graphs built form the program to be sliced [7]. These graphs include both the data and the control dependencies of the program.

There are many ways to use program slicing during the software life-cycle. It can be used in debugging, optimization, program analysis, testing or other software maintenance tasks. For example, using program slicing to detect the impact of a change on a certain point of the program could help to the developer to select the subset of the test cases which could be affected by a program code change.

Our goal is to adopt the existing methods and to develop new algorithms for program slicing of programs written in a dynamically typed functional programming language, Erlang [2]. Therefore we use three kinds of dependencies: data, behaviour and control dependency information. The first two kinds of dependencies have been studied in previous papers [9, 13], so in this paper we focus on control dependency. The control dependency graph is based on the control-flow graph of the Erlang programs.

The dependency graphs are useful to reach the mentioned goal and transform the program slicing to a graph reachability problem. We want to calculate the forward slices of the program, especially for those program parts which are changed after a refactoring [3]. Calculating the forward slices could help the programmers to reduce the number of test cases to be rerun after the transformation.

Our project's goal is to measure the impact of refactorings made by Refactor-Erl. RefactorErl [5, 6] is a refactoring tool for Erlang. It was originally designed to be a framework for source code transformation, but it is also a static analyzer tool. It has 24 implemented refactorings, features for module and function clustering, a user defined semantic query language to support code comprehension and a query language to query structural complexity metrics of Erlang programs.

The rest of this paper is structured as follows. Section 2 describes the Erlang syntax. Section 3 introduces the Erlang control-flow and dependency graph. Section 4 presents related work, and Sections 5 and 6 conclude the paper and discuss future work.

2 A partial model for Erlang programs

In the following sections we introduce formal rules to define the control flow graph for Erlang. In the presented rules we use the Erlang syntax described in Figure 1. This syntax is a subset of the Erlang syntax presented in [4]. The symbol P denotes the Erlang patterns, E denotes the Erlang (guard)expressions and F denotes the named functions.

The presented model is not complete and contains some simplifications, these are:

- Some expression types (try, if) are left out form the table, because they can be handled similarly to the presented ones.
- The attributes of the Erlang modules do not carry relevant information in the meaning of control and data dependencies, thus they are also left out from the table.
- In fact the guards in Erlang are expressions with some restrictions, but we represent the guards as simple expressions. The differences between them are that the guards can call just a few functions ("guard" built in functions or type test), the infix guard expressions are arithmetic or boolean expressions, or term comparisons and guards can contain only bound variables.
- The receive expression has an optional "after" clause that is not present in the formal description.
- $\circ$ denotes the infix expressions. "!" is a special infix expression: it denotes the message passing in Erlang.

3 Retrieving dependency information

3.1 A representation of the Erlang programs

For building the dependency graph we use the Semantic Program Graph (SPG) of RefactorErl. The SPG is a three layered graph, which stores lexical, syntactic and semantic information about the Erlang programs. The base of the graph is an abstract syntax tree and different static analyzers extend the AST with semantic information, for example the call graph of the program, the record usage, or the binding structure of the variables. Information retrieval is available through a query language, which is quicker and more efficient than traversing the abstract syntax tree of the program.

The analyzer framework of RefactorErl is asynchronous and incremental. The SPG is stored in Mnesia (built in database for Erlang), and after each syntactic transformation the analyzer framework restores the necessary semantic information in the graph and in the database, so we do not need to reanalyze the

V	::=	variables (including _, the underscore pattern)
A	::=	atoms
I	::=	integers
K	::=	$A \mid I$ | other constants (e.g. strings, floats)
P	::=	$K \mid V \mid \{P,\ldots,P\} \mid [P,\ldots,P\|P]$
E	::=	$K \mid V \mid \{E,\ldots,E\} \mid [E,\ldots,E\|E] \mid$
		$[E\|P<\text{-}E] \mid P = E \mid E \circ E \mid$
		$E!E \mid (E) \mid E(E,\ldots,E) \mid$
		`case` E `of`
		P when E -> $E,\ldots,E$;
		$\vdots$
		P when E -> $E,\ldots,E$
		`end` |
		`receive`
		P when E -> $E,\ldots,E$;
		$\vdots$
		P when E -> $E,\ldots,E$
		`end`
F	::=	$A(P,\ldots,P)$ when E -> $E,\ldots,E$;
		$\vdots$
		$A(P,\ldots,P)$ when E -> $E,\ldots,E$

Fig. 1. The used Erlang syntax subset

programs before each transformation, just an initial load is necessary. The analyzer framework guarantees the semantic consistency of the graph using efficient incremental analysis, when a subexpression is transformed (insert/remove/update/replace) only the affected expression and its necessary context will be reanalyzed. Since we do not want to rebuild the whole dependency graph after each refactoring step, we should make the used flow analysis as incremental as possible.

3.2 Dependency information

We have to consider different kinds of dependency information to perform program slicing. The following dependencies must be taken in account: data, be haviour and control dependency. In this paper our focus is on control dependency. The Dependency Graph (DG), that is used to perform program slicing, contains each kinds of dependencies. The DG contains the Control Dependency Graph (CDG) and additional data and behaviour dependency edges. The CDG is built based on the Control-Flow Graph (CFG) of the Erlang program.

The steps in creating the DG are:

- Create the CFG of the needed Erlang functions separately
- Create the intrafunctional CDG from the CFG
- Interconnect the CDG-s of the functions
- Add data and behaviour dependency edges to the resulted interfunctional control dependency graph.

The data, behaviour and control flow edges could be calculated in an incre- mental way (based on the compositional rules: Section 3.3 and [9, 13]. After a refactoring we should rebuild the intrafunctional CDG-s only for the changed functions and replace the old version in the interfunctional CDG.

3.3 Control-Flow Graph

We build the control flow graph of the Erlang program based on the formal rules defined in Figures 3 and 2 and 4. The rules correspond to the semantics of Erlang presented in [4].

The notation on the figures are: $e \in E$ is an expression, $g \in E$ is a guard expression, $p \in P$ is a pattern and $f \in F$ is a function. $e' \in E$ is a dummy node in the controlflow graph, its role is to avoid unnecessary loops in the CFG. There are summary nodes (*ret*) to represent return value in case of branching evaluation. The relation $\rightarrow$ represents a direct control flow relation between two nodes. The relations $\xrightarrow{call}$, $\xrightarrow{rec}$, $\xrightarrow{send}$ represent an auxiliary relation which indicate dependency between the nodes of different functions (for details, see Section 3.4). In the rest of this section we want to describe some of the control flow rules.

Functions The control flow model of an Erlang function is shown on Figure 2:Function. When a function is called the first matching pattern should be selected at first. If the pattern on the first function clause does not match ($p_1^1, \ldots, p_n^1 \xrightarrow{no} p_1^2, \ldots, p_n^2$), then the second clause follows. Otherwise ($p_1^1, \ldots, p_n^1 \xrightarrow{yes} g_1$) the guard expression is evaluated, and if it holds, then the control flows to the body of the function ($g_1 \xrightarrow{yes} e_1^1$). Otherwise the control flows to the second clause, etc. The control flow among the expressions in the body of the function and the last expression returns ($e_{l_i}^i \rightarrow ret\ f/n$).

Match expressions. On Figure 3:Match exp. the rule $e_0' \rightarrow e_1$ means that when the match expression gets the control the e_1 is evaluated at first, and then the control flows to e_0.

Infix expressions Figure 3:Infix exp. shows that before evaluating an infix expression the left and then the right hand side subexpression is evaluated.

Compound data structures. In the evaluation of compound data structures (Figures 3: Tuple exp. and List exp) the control flows from left to right direction.

List comprehensions. List comprehensions (Figures 3: List gen.) are like loops in the imperative languages. At first we take one element of the list $e_2(e_2 \rightarrow p)$ and then we evaluate the expression e_1. After it the control flows back to $e_2(e_1 \rightarrow e_2)$. When e_2 becomes empty then the control flows back to $e_0(e_1 \rightarrow e_0)$.

Conditional expressions. The rule of a conditional expression (Figure 3: Case exp.) is similar to the rule of the function (Figure 2: Function.), but before matching the patterns e is evaluated.

Function calls. In case of the parameters of a function call (Figure 4: Fun. call.) the control flows from left to right. Then the evaluation should pass to the called function. Therefore the $\xrightarrow{call}$ edge indicate an interfunctional dependency, which should be considered during building the control dependency graph.

Receive and send expressions. Similarly to the function calls the rules of the receive and the send expressions (Figure 4: Receive. and Send.) also contain auxiliary edges ($\xrightarrow{rec}$, $\xrightarrow{send}$) indicating that the evaluation depend on the sent/received messages.

3.4 Compositional CDG

As we want to define a dependency graph that can be maintained we follow the compositional approach described in [11].

First we build the CFG based on the formal rules described in Section 3.3. For every function in the program the CFG is built separately, thus we obtain so called intrafunctional CFG for every function. This CFG does not follow the call function calls, but denotes the fact of the function call $\xrightarrow{call}$ and this information will be used while building the post-dominator tree and the control dependency graph (CDG). This edge is called potential control-flow edge.

The next step in building the CDG is to construct the post-dominator tree (PDT). We use the algorithm presented in [8]. There are two types of edges in the postdominator tree, these are: immediate postdominator and potential postdominator. The post-dominator tree is extended with the potential postdominator arcs, that the next expression after the function call potentially postdominates the function call. If it turns out at composing the CDGs that it is not the case, the edge will be replaced corresponding to the context, or can be deleted.

We now have the CFGs and PDTs of the functions built intrafunctionally. Using the CFG and the corresponding PDT we build the intrafunctional CDG that contains the direct control dependencies and the potential control dependencies inherited from the potential post-dominators. The potential control dependency edges will be resolved at the time of composing the intrafunctional CDGs.

The next level in building the CDG for the entire program is to compose the intrafunctional CDG of the functions. In this process we change the potential control dependence edges to real control dependencies or indirect control dependence edges corresponding to the calling context of the functions.

```
calc_dg(SPG)->
  FlowGraph_List = calc_cfg(SPG),
  CDG_List =
    lists:map(fun calc_cdg/1,
       FlowGraph_List),
  Comp_CDG = compose_cdg(CDG_List),
  Intrafunc_CDG =
    resolve_potential_dep(Comp_CDG),
  _DG = add_behav_dep(add_data_dep(CDG)).
```

Fig. 5. Draft algorithm for creating the dependency graph

When we build the intrafunctional CDG we also have to resolve the potential dependency indicated by the edges $\xrightarrow{rec}$ and $\xrightarrow{send}$. The received message influences the control, thus adds dependency edges to the graph. We have to extend our data- and behaviour-flow model with message passing analysis.

3.5 Slicing

Our main goal is to select a subset of Erlang test cases which has to be rerun after some kinds of change on the source code, therefore we want to perform static froward slicing. A forward

	Expressions	CFG edge
		$f/n \to p_1^1$
		$\{p_1^1,\ldots,p_n^1\} \xrightarrow{yes} g^1$ $\{p_1^1,\ldots,p_n^1\} \xrightarrow{no} \{p_1^2,\ldots,p_n^2\}$ $\vdots$ $\{p_1^{m-1},\ldots,p_n^{m-1}\} \xrightarrow{yes} g^{m-1}$ $\{p_1^{m-1},\ldots,p_n^{m-1}\} \xrightarrow{no} \{p_1^m,\ldots,p_n^m\}$
		$\{p_1^m,\ldots,p_n^m\} \xrightarrow{yes} g^m$ $\{p_1^m,\ldots,p_n^m\} \xrightarrow{no} error$
(Function)	f/n : $f(p_1^1,\ldots,p_n^1)$ when $g^1 \to e_1^1,\ldots,e_{l_1}^1$; $\vdots$ $f(p_1^m,\ldots,p_n^m)$ when $g^m \to e_1^m,\ldots,e_{l_m}^m$	$g^1 \xrightarrow{yes} e_1^1$ $\vdots$ $g^{m-1} \xrightarrow{yes} e_1^{m-1}$, $g^{m-1} \xrightarrow{no} \{p_1^m,\ldots,p_n^m\}$,
		$g^m \xrightarrow{yes} e_1^m$ $g^m \xrightarrow{no} error$
		$e_1^1 \to e_1^1,\ldots,e_{l_1-1}^1 \to e_{l_1}^1$, $\vdots$ $e_1^m \to e_2^m,\ldots,e_{l_m-1}^m \to e_{l_m}^m$,
		$e_{l_1}^1 \to ret\ f/n$ $\vdots$ $e_{l_m}^m \to ret\ f/n$,

Fig. 2. Control-flow edges

slice contains from those expressions of the program that are dependent on the value of the modified expression.

The slicing criteria is a vertex in the graph, that represents the modified expression in the DG. It is also possible that the slicing criteria is a set of vertices, if the change affects more than one expression.

Program slicing is a graph reachability problem on the resulted Dependency Graph. We have to traverse the DG starting from the slicing criteria, and the resulted slice contains all the vertices from the DG that are reachable from the source. The resulted slice will be a non executable slices of the program. Designing the graph reaching and traversing algorithms are in progress.

4 Related work

There are some projects that work with test case selection in case of object-oriented languages. For example, the paper [1] gives a formal mapping between design changes and a classification of regression test cases (reusable, retestable, obsolete) using the Unified Modeling Language.

Using program slicing to measure the impact of a change in case of functional languages is not really widespread, but some publications are dealing with flow analysis of functional languages. Shivers' thesis [10] presented the theory of flow analysis of higher order languages, and that is applied for optimization in compilers. Different flow analysis was applied for improving the testing process in Erlang [15].

In the thesis [11] a language independent control dependency analysis was studied and applied for example to software architecture descriptions [12].

5 Conclusions

Our goal is to perform impact analysis through program slicing. Specially we want to measure the impact of a change on a set of test cases, and select a subset from it which should be retested after the source code modification.

There are many forms of program slicing, we choose the dependency graph based analysis. The Dependency Graph of the program depends on the syntax and semantics of the used language. In this paper we focused on the dynamically typed functional programming language, Erlang.

The Dependency Graph contains control, data and behaviour dependency information about the Erlang programs. In this paper we presented the controlflow graphs of Erlang programs and a method to build the interfunctional control dependency graph from it. The dependency graph contains the interfunctional con-

	Expressions	CFG edges
(Match exp.)	$e_0 : p = e_1$	$e'_0 \to e_1, e_1 \to e_0$
(Infix exp.)	$e_0 : e_1 \circ e_2$	$e'_0 \to e_1, e_1 \to e_2, e_2 \to e_0$
(Parenthesis)	$e_0 : (e_1)$	$e'_0 \to e_1, e_1 \to e_0$
(Tuple exp.)	$e_0 :$ $\{e_1, \ldots, e_n\}$	$e'_0 \to e_1$ $e_1 \to e_2, \ldots, e_{n-1} \to e_n$ $e_n \to e_0$
(List exp.)	$e_0 :$ $[e_1, \ldots, e_n \vert e_{n+1}]$	$e'_0 \to e_1$ $e_1 \to e_2, \ldots, e_n \to e_{n+1}$ $e_{n+1} \to e_0$
(List gen.)	$e_0 :$ $[e_1 \Vert p \leftarrow e_2]$	$e'_0 \to e_2, e_2 \to p, p \to e_1$ $e_1 \to e_2, e_1 \to e_0$
(Case exp.)	$e_0 :$ case e of p_1 when $g_1 \to e^1_1, \ldots, e^1_{l_1}$; $\vdots$ p_n when $g_n \to e^n_1, \ldots, e^n_{l_n}$ end	$e'_0 \xrightarrow{e}, e \to p_1,$ $p_1 \xrightarrow{yes} g_1, p_1 \xrightarrow{no} p_2,$ $\vdots$ $p_{n_1} \xrightarrow{yes} g_{n-1}, p_{n-1} \xrightarrow{no} p_n,$ $p_n \xrightarrow{yes} g_n, p_n \xrightarrow{no} error,$ $g_1 \xrightarrow{yes} e^1_1, g_1 \xrightarrow{no} p_2,$ $\vdots$ $g_{n-1} \xrightarrow{yes} e^{n-1}_1, g_{n-1} \xrightarrow{no} p_n,$ $g_n \xrightarrow{yes} e^n_1, g_n \xrightarrow{no} error,$ $e^1_1 \to e^1_2, \ldots, e^1_{l_1-1} \to e^1_{l_1},$ $\vdots$ $e^n_1 \to e^n_2, \ldots, e^n_{l_n-1} \to e^n_{l_n},$ $e^1_{l_1} \to ret\ case$ $\vdots$ $e^n_{l_n} \to ret\ case,$ $ret\ case \to e_0$

Fig. 3. Control-flow edges

trol dependency graph extended with data and behaviour dependency edges. The program slice could be calculated by traversing the dependency graph. The resulted slice is a non executable static forward slice of the program.

6 Future work

The presented DG could be improved and refined in different ways. One of them is the usage of n-th order flow analysis. The presented model based on a 0-th order data flow graph. One of the disadvantage of that graph is that we can not distinguish the different function calls and that make the graph imprecise. An other improvement on the data flow graph is an accurate message passing analysis which can also improve the control dependency graph.

Regarding the dynamic nature of the language the static analysis is not straightforward, but some kinds of extra knowledge about the library functions could help to improve the accuracy of the graph. An example could be the usage of generic servers (gen_servers) to implement client-server applications [2]. In this case the library functions hide a lot of information about the control flow, but we know that each gen server call indicate a calback function call which can be analyzed instead of the gen_server call.

References

1 **Briand L, Labiche Y, Soccar G**, *Automating impact analysis and regression test selection based on uml designs*, 18th IEEE International Conference on Software Maintenance (ICSM'02), posted on 2002, DOI 10.1109/ICSM.2002.1167775, (to appear in print).

2 **Ericsson AB**, *Erlang Reference Manual*, available at `http://www.erlang.org/doc/referencemanual/usersguide.html`.

3 **Fowler M, Beck K, Brant J, Opdyke W, Roberts D**, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

4 **Fredlund L A**, *A Framework for Reasoning about ERLANG code*, PhD thesis, Stockholm, Sweden, 2001.

5 **Horváth Z, Lövei L, Kozsik T, Kitlei R, Tóth M, Bozó I, Király R**, *Modeling semantic knowledge in Erlang for refactoring*, Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009 **34** (2009), 7–16.

6 **Horváth Z, Lövei L, Kozsik T, Kitlei R, Víg A N, Nagy T, Tóth M, Király R**, *Building a refactoring tool for erlang*, Workshop on Advanced Software Development Tools and Techniques, WASDETT 2008 (Jul, 2008).

	Expressions	CFG edge
		$e'_0 \to e_1$,
	e_0 :	$e_1 \to e_2, \ldots, e_{n-1} \to e_n$,
(Fun. call)	f($e_1, \ldots, e_n$)	$e_n \xrightarrow{call} e_0$
		$e'_0 \xrightarrow{rec} p_1$,
	e_0:	⋮
	`receive`	Similar to rule (Case exp.)
	p_1 when $g_1 \to e^1_1, \ldots, e^1_{l_1}$;	⋮
	⋮	
(Receive)	p_n when $g_n \to e^n_1, \ldots, e^1_{l_n}$	$e^1_{l_1} \to$ *ret receive*
	end	⋮
	e_0 :	$e'_0 \to e_2, e_2 \to e_1$,
(Send)	$e_1 ! e_2$	$e_1 \xrightarrow{send} e_0$

Fig. 4. Control-flow edges

7 **Horwitz S, Reps T, Binkley D**, *Interprocedural slicing using dependence graphs*, PhD thesis, Ann Arbor, MI, 1979.

8 **Lengauer T, Tarjan R E**, *A fast algorithm for finding dominators in a flowgraph*, ACM Transactions on Programming Languages and Systems (TOPLAS) **1** (1979), no. 1, 121–141.

9 **Lövei L**, *Automated module interface upgrade*, Erlang '09: Proceedings of the 8th ACM SIGPLAN workshop on Erlang (2009), 11-22.

10 **Shivers O**, *Control-Flow Analysis of Higher-Order Languages*, PhD thesis, 1991.

11 **Stafford J**, *A formal, language-independent, and compositional approach to control dependence analysis*, PhD thesis, 2000.

12 **Stafford J A, Wolf A L, Wolf E L, Caporuscio M**, *The application of dependence analysis to software architecture descriptions*, Formal Methods to Software Architects (2003), 52–62.

13 **Tóth M, Bozó I, Horváth Z, Lövei L, Tejfel M, Kozsik T**, *Impact analysis of erlang programs using behaviour dependency graphs*, Central European Functional Programming School. Third Summer School, CEFP 2009. Revised Selected Lectures (2010).

14 **Weiser M**, *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*, ACM Transactions on Programming Languages and Systems **12** (January, 1990), no. 1, 3546.

15 **Widera M**, *Flow graphs for testing sequential erlang programs*, Proceedings of the ACM SIGPLAN 2004 Erlang Workshop (2004), 48–53.