e-Informatica Software Engineering Journal, Volume 7, Issue 1, 2013, pages: 65–76, DOI 10.5277/e-Inf130107

Static analysis of function calls in Erlang

Refining the static function call graph with dynamic call information
by using data-flow analysis

Dániel Horpácsi*, Judit Kőszegi*

*Department of Programming Languages and Compilers, Eötvös Loránd University, Budapest, Hungary

daniel_h@inf.elte.hu, kojqaai@inf.elte.hu

Abstract

Functions and their relations can affect numerous properties and metrics of a functional program. To identify and represent the functions and their calling connections, software analysers commonly apply semantic function analysis, which derives the static call graph of the program, based on its source code. Function calls however may be dynamic and complex, making it difficult to statically identify the callee. Dynamic calls are determined just at run-time, static analysis therefore cannot be expected to fully identify every call.

Nevertheless, by utilising the results of a properly performed data-flow analysis as well as taking ambiguous references into account, numerous dynamic calls are discoverable and representable. We consider cases where the identifiers of the callee are statically determined, but they flow into the call expression from a different program point, and also, we manage to handle function calls whose identifiers are not fully identifiable at compile-time. By utilising the improved reference analysis, we extend the static call graph with various information about dynamic function calls. We investigate such a function call analysis in the programming language Erlang.

1. Introduction

To overview the components of the software, to identify relations and dependencies, and to find out properties, we can apply static source code analysis. The analysis can be followed by semi-automatic code transformations correcting design weaknesses, based on the analysis results. Both software transformation tools and reverse engineering techniques operate on programs, and involve many sorts of static code analysis.

Consider the case of refactoring tools [1, 2], where code transformations never should change the semantics of the program being refactored. Despite the fact that such automatic code transformation tools often involve user interaction, the code modifications made are directed mostly by the information gathered from the source code. Consequently, the correctness of the transformations highly depends on the accuracy of the static analysis carried out before the actual transformation steps.

In this paper we focus on the analysis of inter-procedural relationships. In different programming languages there are different call constructs, including dynamic ones that may be unidentifiable at compile-time and therefore usually are omitted by static analysers. However, the data bindings that determine dynamic calls may be looked up by use of static data-flow analysis, and if the function identifiers are statically given in the code, we can successfully locate them and identify the referred function. We concentrate on the analysis of call constructs present in Erlang [3, 4], a dynamically typed functional programming language. The presented approach aims to refine function call graphs [5] by means of static analysis of dynamic function calls. By utilising the more sound call graph we can improve different sorts of static analysis as well as refactoring code transformations.

In the next sections we introduce the main types of function call constructs – including dynamic ones – and also we precisely define the connection between call expressions and function entities in terms of formal semantic rules. In addition, concepts of ambiguous dynamic calls and opaque functions are introduced in order to represent partially unidentifiable function references as well. Finally, we define formal relationships between the function entities, merging all the call information into the call graph and show a simple case study.

2. Call constructs

We suppose that the functions of a program are grouped into modules and are dynamically typed (like in Erlang). With this, a function may be identified by a 3-tuple (so-called *MFA*) that includes the name of the module the function is located in, the name of the function, and the number of its formal parameters (called 'function arity'). This function descriptor can be written in the form of *module:function/arity.*

	MFA-call	Apply-call
identifiers as literals	static	dynamic
identifiers as expressions	dynamic	dynamic

Figure 1. Static and dynamic calls

Call constructs are sorted in order to ease their analysis; Figure 1 shows the main groups. Syntactically, we consider two types of function call: MFA-calls and apply-calls. The former one is the common way to invoke a subroutine, while apply-calls may be regarded as symbolic calls so that they refer to a special function named 'apply' which then results in another call. For this use, both the function to be called and its arguments are specified within the arguments of the apply-call. When called, it executes the named function on the specified arguments and then returns its result. Basically, apply-calls behave very similarly to MFA-calls, however, the main difference in use lies in the way the parameters are constructed and then passed to the callee.

Beside the syntactic grouping, we make a distinction between static and dynamic call methods. Function calls may affect the data-flow within the program, and interestingly, data-flow may also take effect on function calls so that function calls may be constructed by means of run-time data. Programming languages usually support meta-programming techniques like handling the program itself as data or creating statements at run-time ('eval' methods). A special case of the latter technique is the run-time construction of function calls, where the identifier(s) of the called routine may be determined just at execution-time, similarly to the construction of actual function parameters. There are many programming languages that support meta-programming (and dynamic call constructs), including script languages (like JavaScript, Ruby, Python) as well as functional languages, such as Erlang and Scheme.

2.1. MFA-calls

In Erlang, MFA function calls provide the standard way of executing a function from inside another one. These call expressions can be written using the following syntax:

module_name:*function_name*(*arg1*, ..., *argN*)

Each of *module_name* and *function_name* must be either an identifier (atom literal) or an Erlang expression that evaluates to an identifier determining the name of the callee. Observe that the arity is fully defined at compile-time, with the number of parameters actually enumerated between the parentheses. When a call is written in the above syntax and both function identifiers are given as atom literals, we say that the function call is a static MFA-call. It is said to be static because the identifiers of the function are given statically, at the location of the call. Most static analyser software can successfully observe such call constructs and can build the corresponding call graph, however, they simply omit dynamic call methods.

In dynamic MFA-calls, either the module name or the function name is given with non-literal expressions (for instance, with variables). Static analysis of such calls requires some kind of data-flow analysis which uncovers the origin of the data (for example, the values bound to the variables).

2.2. Apply-calls

As we already mentioned, there is another call construct in Erlang, the so-called apply-call, which is based on a built-in higher-order function called `apply`. (We note that similar call constructs exist in various programming languages.) This construct is dynamic by nature, since the identifiers of the called function are given by means of the arguments of another routine, which is evaluated certainly just at run-time. The apply-calling expressions can be written using the following syntax:

$$\mathsf{apply}(\mathit{module\_name}, \mathit{function\_name}, \mathit{arg\_list})$$
$$\text{where} \qquad \mathit{arg\_list} \equiv [\mathit{arg1}, ..., \mathit{argN}]$$

The call primarily refers to a built-in function called `apply` that is located in the module `erlang` (one may refer to it as *erlang:apply/3*). Its parameters determine the secondarily referred function that is being called at run-time. Each of *module_name* and *function_name* must be either an identifier (atom literal) or an Erlang expression that evaluates to an identifier, while *arg_list* should evaluate to an Erlang list whose length precisely determines the arity of the callee at run-time.

Listing 1 shows a simple apply-call. Since the identifiers are given as atom literals, one may regard this simple case as a static apply-call. However, as we already stated, apply-calls are dynamic by their nature, so in this paper we will treat and analyse apply-calls as fully dynamic constructs. In Listing 2 a more complex example shows a function call referring to the same function whilst demonstrating possible difficulties of static analysis implemented on dynamic function calls. Note that we already came to the need of data-flow analysis and additional static analysis methods, which motivates us to examine our possibilities on static analysis of dynamic function calls.

Listing 1. Simple apply-call

```
apply(io, format, ["hello", []]).
```

Listing 2. More complex apply-call

```
f() -> {format, io}.
g() -> {F, M} = f(),
       Rest = [[]],
       Args = ["hello" | Rest],
       apply(M, F, Args).
```

3. Static analysis of function calls

In the context of static, context-insensitive call analysis, “function” stands for an identifier (or a signature) of a routine, that is a (possibly minimal) set of data that can unambiguously identify the routine. Basically, static function analysis aims to extract these function descriptors and their calling connections into a sound function call graph, which can be used within further analysis or program transformations. We assume that each routine of the program under analysis has such a function descriptor (also referred to as semantic function entity).

In the following sections we define the connection between expressions of the source code and semantic function entities involved in the program, in terms of formal semantic rules. While describing the semantics, we suppose that the program code is represented by an abstract syntax tree, so semantic rules instance syntax elements as subtrees of the semantic program graph (3-layered, labelled extension of the abstract syntax tree, including the static semantics of the program). Most of the concepts described in the paper have been implemented in RefactorErl [2], a source code analyser and transformer tool, wherein many other kinds of static semantic analysis can be carried out on Erlang programs [6].

Data-flow analysis. The analyser framework of RefactorErl includes, in addition to many kind of analysis, a data-flow analyser, which is able to carry out 0^{th} order and 1^{st} order data-flow analysis [7]. Its backward data-flow reaching relation returns all the expressions affecting the queried one. However, in the case of dynamic call analysis we only need the ends of the reaching paths, that is, those nodes that may potentially uncover the possible values of expressions. We introduce the concept of compact data-flow reaching, which performs pure closure on the data-flow relation and consequently returns only the nodes in which the reaching terminates. If such an expression is a literal, we found a possible value of the expression. Within the dynamic call analysis, we use the 0^{th} order compact backward data-flow reaching relation.

Auxiliary definitions. Previously we shortly introduced the syntax of Erlang function call expressions. In this paper we only use a small subset of the language syntax, on which we build in the course of defining semantic rules, so the syntax of the whole language is not specified by formal means. Basically, Erlang programs consist of forms (mostly functions) grouped into modules, where each function embodies a sequence of expressions. In Erlang programs, atoms (named constants) are used to identify entities, e.g. modules and functions.

In the rest of the paper,

- E_{Atom} denotes the set of Erlang atom literal expressions
- E_{List} is the set of all expressions that construct list values
- E denotes the set of all Erlang expressions (including E_{Atom} and E_{List} as well).

We define some sets of semantic values (domains):

- $Atom$ = set of possible atom values
- $Atom' = Atom \cup \{\bot\}$
- $\mathbb{N}^{\geq} = \{n^{\geq} | n \in \mathbb{N}\}$ where $n^{\geq} \equiv [n..\infty)$
- $\mathbb{N}' = \mathbb{N} \cup \mathbb{N}^{\geq} \cup \{\bot\}$

And also, we define a total ordering on the elements of $\mathbb{N}^{\geq}$ $(n, m \in \mathbb{N})$:

$$n^{\geq} \leq m^{\geq} \qquad \text{if } n \leq m$$

Now, the following functions are defined over the syntactic elements and map onto the semantic domains, giving the bridge between syntax and semantics.

$Val : E_{\boldsymbol{Atom}} \mapsto Atom$

$Val(e)$ returns the value given by the evaluation of the atom expression e.

$Length : E_{\boldsymbol{List}} \mapsto \mathbb{N}'$

$Length(e)$ gives the length of the Erlang list value represented by the expression e. Note that it maps to $\mathbb{N}'$ and therefore may return either a concrete number, a lower bound, or the $\bot$ symbol.

If the list length is only partially analysable and thus a lower bound is calculable, then $Length(e) \in \mathbb{N}^{\geq}$. Also, if we cannot calculate the list length at all, then $Length(e) = \bot$.

$\overset{0f_{cb}}{\rightsquigarrow} \subseteq E \times E$

$e_1 \overset{0f_{cb}}{\rightsquigarrow} e_2$ means that the value of e_1 flows into e_2 in 0^{th} order using compact reaching [7].

Finally, we define the set of semantic function entities and define a function that returns such entities based on their 3-tuple descriptor. Note that each of the function identifiers may be undefined ($\bot$).

$SemFun$

The semantic function entities involved in the analysed program

$Function : Atom' \times Atom' \times \mathbb{N}' \mapsto SemFun$

$Function(m, f, a)$ returns the function entity identified by its module name, function name and arity.

In the following, $e_1 \xrightarrow{L} e_2$ denotes a binary relation between e_1 and e_2, which is a directed graph edge (being labelled by L) between the two graph nodes (e.g. expression occurrences) on the implementation level.

3.1. Semantics of MFA-calls

In order to define the syntax of MFA-calls, we use the previously introduced sets of language elements. The abstract syntax of an MFA-call is the following.

$$e_{MFA} \equiv e_{MN}{:}e_{FN}(e_1, \ldots, e_n)$$

In the above line, e_{MFA} is a node belonging to an MFA-call expression referring to a function with exactly n parameters. We only assume that the module name (e_{MN}), the function name (e_{FN}), and the actual parameters ($e_1, \ldots, e_n$) are given as Erlang expressions.

$$e_{MFA}, e_{MN}, e_{FN}, e_1, \ldots, e_n \in E$$

The following semantic rules define relations between syntactic elements and semantic entities. We have already seen that the parameters of an MFA-call are explicitly enumerated within the call, thus the arity of the callee is easy to calculate. Consequently, the potential difficulties may arise during the analysis of module and function names, since they may flow into the expressions e_{MN} and e_{FN} from an arbitrarily far point of the program (e.g. from another module or another application). Our goal is to uncover the possible values of these expressions by use of data-flow analysis in order to refine the call graph with the dynamic call relations.

Static calls. First of all, we define the rule of static MFA-calls, shown in rule MFA1. It is pretty straightforward, but apparently a necessary part of the model. In this case both the module name and the function name are given as atom literals, so the identifiers are given just at the point of the call. One can see that the values of the atom expressions together with the parameter count exactly identify the function being referred. The call expression is linked to the function entity labelled by *funref* (static function reference).

Fully identifiable dynamic calls. When the module name or the function name is not explicitly given as an atom literal, however, by applying data-flow reaching we can successfully find out some possible values of the expression(s), we can identify some possible callee. Such references are said to be dynamic and unambiguous.

The call expression is linked to all the possible functions, labelled by *dynref* (dynamic function reference). Listing 3 shows a dynamic MFA-call in which the identification of the module name requires data-flow analysis. The name comes from another function call, the outer call is analysed by using the rule MFA2.

Listing 3. Fully identifiable dynamic MFA-call

```
iomodule() -> io.
f() -> (iomodule()):format("hello",[]).
```

Ambiguous function calls. Now let us define a previously not detailed kind of call reference. Namely, in the case if an element of the function descriptor cannot be determined by using data-flow analysis either, we are not able to fully identify the potentially referred functions. The reason why we are not able to calculate, for example, a function name, is that the data-flow path ends not in an atom literal but in another kind of expression from which the reaching cannot be continued. In order to be able to consider such cases, we introduce the concept of ambiguous function references, where one identifier of the callee is unable to be precisely calculated. Listing 4 demonstrates a call whose module reference is unknown. Even in this case we note a function reference, however, not to a fully defined function. Instead, we define opaque functions and create references to these special function entities.

We note that we do not deal with function calls not specifying at least two of the three main identifiers of a function (module name, function name, and arity). Opaque functions consequently only have exactly one undefined field in their descriptor. In addition, there is a special case that may appear during the analysis of apply calls, namely, when the argument list is only partially present, and based on it we can calculate a lower bound of the function arity. This issue will be detailed in the section of apply-call analysis. An overview of dynamic/ambiguous calls and their analysis is present in Figure 2.

Partially identifiable MFA-calls. As we mentioned, in the case of ambiguous references one of the three main function identifiers is unable to be determined by means of static analysis. Since the arity is exactly given by syntax of MFA-calls, the uncertainty may come from the identification of the names.

Suppose that the function name is determinable. If there is a case in which we cannot determine the name of the referred module, we cannot completely identify the potentially referred functions. In order to indicate this, we create a reference that points to an opaque function whose module name is unknown ($\perp$). See rule MFA3.

If the module name is determinable and the function name is not, we analogously get to the rule for MFA-calls with unidentifiable function names (see rule MFA4).

Listing 4 demonstrates a function call where the function name comes from a case expression and can be either “foo” or an arbitrary atom read from the standard input. Observe that while analysing this example we should apply each of rule MFA2 and rule MFA4, since the first case clause gives a fully identified name, in contrast with the second one, which refers to a value that is unknown at compile-time. Consequently, the call expression is related to two different functions: with *dynref* to a fully defined function, and with *ambref* to an opaque function entity.

Listing 4. Ambiguous MFA-call

```
Fun = case read_int() == 0 of
        false -> foo;
        true  -> read_atom()
      end,
module:Fun(ok, 0)
```

In Listing 5 we show a call that is skipped during the function analysis in order to avoid storing calling relations that are not specified enough and would excessively expand the call graph.

Listing 5. Unanalysed MFA-call

```
{Mod, Fun} = {read_atom(), read_atom()},
Mod:Fun(0)
```

3.2. Semantics of apply-calls

An apply-call may be regarded as a meta-call that primarily refers to the *erlang:apply/3* built-in function and secondly refers to the function specified in the call parameters. The abstract syntax of apply-calls is the following.

$$e_{APP} \equiv \mathsf{apply}(e_{MN},e_{FN},e_{Args})$$

In the above line, e_{APP} is a node belonging to an apply-call expression. We only assume that the module name (e_{MN}), the function name (e_{FN}), and the actual list of parameters (e_{Args}) are given as legal Erlang expressions (thus the argument list does not have to be a list expression actually).

$$e_{APP}, e_{MN}, e_{FN}, e_{Args} \in E$$

The following semantic rules define dynamic call relations between call expressions and function entities.

Fully identifiable apply-calls. In case of apply-calls, the module and function names as well as the argument lists may be constructed arbitrarily far from the call point. Provided that applying data-flow reaching we can successfully find out the possible value of the name expression(s) as well as the length of the argument list, we can fully identify the callee. Such cases are called to be dynamic, unambiguous references. The call expression is linked to all the possibly referred functions, labelled by *dynref.* Listing 6 shows an apply-call that requires data-flow analysis, but it is still possible identify the callee by using rule APP1.

Listing 6. Fully identified apply-call

```
MN   = io,
FN   = format,
Args = ["hello"],
apply(MN, FN, Args)
```

Partially identifiable apply-calls. Similarly to MFA-calls, apply-calls may be ambiguous, which means we cannot certainly identify every callee. Any component of the 3-tuple identifying

	MFA-call	*Apply-call*
all identifiers are calculable	dynamic	dynamic
one of the identifiers is incalculable	ambiguous	ambiguous
module and function names plus a lower bound of the arity are calculable	—	ambiguous
at least two key identifiers are incalculable	skipped	skipped

Figure 2. Dynamic call types in detail

$$\frac{}{e_{MFA} \xrightarrow{\mathit{funref}} \mathit{Function}(Val(e_{MN}), Val(e_{FN}), n)}\; e_{MN} \in E_{Atom},\; e_{FN} \in E_{Atom} \tag{MFA1}$$

$$\frac{e_x \overset{0f_{cb}}{\rightsquigarrow} e_{MN} \qquad e_y \overset{0f_{cb}}{\rightsquigarrow} e_{FN}}{e_{MFA} \xrightarrow{\mathit{dynref}} \mathit{Function}(Val(e_x), Val(e_y), n)}\; e_{MN} \notin E_{Atom} \vee e_{FN} \notin E_{Atom},\; e_x \in E_{Atom},\; e_y \in E_{Atom} \tag{MFA2}$$

$$\frac{e_z \overset{0f_{cb}}{\rightsquigarrow} e_{MN} \qquad e_x \overset{0f_{cb}}{\rightsquigarrow} e_{FN}}{e_{MFA} \xrightarrow{\mathit{ambref}} \mathit{Function}(\bot, Val(e_x), n)}\; e_x \in E_{Atom},\; e_z \in E \setminus E_{Atom} \tag{MFA3}$$

the function may be incalculable at compile-time, resulting in uncertain function references. Interestingly, due to the way the arguments are passed to the function, there may appear situations where only a lower bound of the function arity is calculable.

Suppose that the function name along with the arity are determinable. If there is a case in which the name of the referred module cannot be calculated by use of data-flow analysis either, we cannot fully identify the potentially referred functions. In order to indicate this, the call expression is linked to an opaque function entity whose module name is undefined (see rule APP2). The rest of the identifiers are read out from the code, while the relation is labelled by *ambref* (ambiguous function reference).

We analogously construct a rule for ambiguous apply-calls with an incalculable function name (see rule APP3). The expression e_{APP} is linked to an opaque function whose containing module name and arity can be read out from the code, however, its name is set to $\bot$ (undefined). The reference is labelled by *ambref.*

Listing 7. Ambiguous apply-call

```
MN = io,
FN = if read_int() == 1 -> format;
        true            -> read_stdin()
     end,
apply(MN, FN, ["hello"])
```

Listing 7 shows an example in which the function name comes from a conditional statement. The analysis uncovers that it may have the value `format`, but on the other hand, it may come from the standard input as well. This results in two relations, based on the rules APP1 and APP3: a dynamic reference goes to *io:format/1* and an ambiguous one to *io:*$\bot$/1.

Now suppose that the module name and the function name are determinable. If we cannot gather any information about the arity of the function by use of data-flow analysis either, a reference points to an opaque function whose arity component is unknown ($\bot$), indicating that the function reference is ambiguous in the arity (see rule APP4).

Sometimes, even if we cannot determine the arity, we still have a chance to calculate a lower bound of the parameter count (it happens if the length of the tail of an argument list is incalculable). If we can uncover such a bound, it will be indicated beside the fact that the function reference is ambiguous. To avoid creating lots of opaque functions, we do not associate separate functions to each different lower bound of the arity the call may refer to. Instead, we identify the greatest lower bound and we link only one opaque function to the call, using the minimum of the lower bounds as arity. Let us define the following predicate.

$$AL(e_{List}) = e_{List} \overset{0f_{cb}}{\rightsquigarrow} e_{Args} \wedge Length(e_{List}) \in \mathbb{N}^{\geq}$$

So $AL(e)$ is true if and only if e flows into the argument list of the call and its length is not fully known, but its lower bound can be calculated with static analysis.

By utilising the AL predicate, we can give the rule for the calls with lower-bounded arities (see APP5). The rule shows that even if many lower-bounds of the arity are calculable, we unify them into a single one, which is actually the

$$\frac{e_x \overset{0f_{cb}}{\rightsquigarrow} e_{MN} \qquad e_z \overset{0f_{cb}}{\rightsquigarrow} e_{FN}}{e_{MFA} \xrightarrow{ambref} Function(Val(e_x), \bot, n)}\ e_x \in E_{Atom},\ e_z \in E \setminus E_{Atom} \tag{MFA4}$$

$$\frac{e_x \overset{0f_{cb}}{\rightsquigarrow} e_{MN} \qquad e_y \overset{0f_{cb}}{\rightsquigarrow} e_{FN} \qquad e_L \overset{f0_{cb}}{\rightsquigarrow} e_{Args}}{e_{APP} \xrightarrow{dynref} Function(Val(e_x), Val(e_y), Length(e_L))}\ e_x, e_y \in E_{Atom},\ e_L \in E_{List},\ Length(e_L) \in \mathbb{N} \tag{APP1}$$

$$\frac{e_z \overset{0f_{cb}}{\rightsquigarrow} e_{MN} \qquad e_x \overset{0f_{cb}}{\rightsquigarrow} e_{FN} \qquad e_L \overset{f0_{cb}}{\rightsquigarrow} e_{Args}}{e_{APP} \xrightarrow{ambref} Function(\bot, Val(e_x), Length(e_L))}\ e_x \in E_{Atom},\ e_z \in E \setminus E_{Atom},\ e_L \in E_{List},\ Length(e_L) \in \mathbb{N} \tag{APP2}$$

greatest lower bound of the arity. As $\mathbb{N}^{\geq}$ is a totally ordered set, the function *minimum* can be used to get the minimal element.

Listing 8. Lower bounded arity in an apply-call

```
msg() -> if is_young(user()) ->
               [$h,$i,$ |read_stdin()];
             true ->
               [$h,$e,$l,$l,$o,$ |read_stdin()]
          end.
f() -> apply(io, format, msg()).
```

In listing 8 we demonstrate the use of rule APP5. In this example there are two different lower-bounded argument lists belonging to the call: one with length $3^{\geq}$, and another one having $6^{\geq}$ elements. Consequently, the greatest lower bound is 3, and thus the arity of the ambiguously referred opaque function is $3^{\geq}$.

The "may be" relation

Let us define

$$mfa : SemFun \mapsto Atom' \times Atom' \times Int'$$

where $mfa(f)$ results in the 3-tuple function descriptor identifying f. We claim that

$$mfa(Function(m, f, a)) = (m, f, a)$$

In the previous sections we have introduced dynamic and ambiguous function references, which relate expressions to function entities. Also, we presented the use of opaque functions, giving the opportunity to precisely represent ambiguous references. However, these opaque functions are special, they may not be regarded as legal elements of the function call graph.

To integrate opaque functions into the call graph, the first step we do is associating these functions to fully defined ones. This relation is called *may_be*, and it connects opaque functions to non-opaque ones that are potential targets of ambiguous calls. Namely, if the unambiguous and the ambiguous functions are identical in the two identifier components defined in the opaque function, the concrete function corresponds to the opaque one. In other words, the concrete function may only differ in the one identifier that is undefined in the opaque function. This relation is defined by rule MAY1.

A special kind of opaque function has a lower bound of its arity. While looking for possible *may_be* connections, one has to take into account not only the names but also the lower bound of the opaque function. A concrete function entity may be associated with a fully defined one only if their names are equal and the concrete arity complies with the lower bound (see rule MAY2).

With the above relation we successfully included the information about ambiguous references and opaque functions into the model of semantic functions by associating opaque entities with concrete ones. Thus we do not have to consider and directly handle opaque functions during further analyses.

$$\frac{e_x \overset{0f_{cb}}{\rightsquigarrow} e_{MN} \qquad e_z \overset{0f_{cb}}{\rightsquigarrow} e_{FN} \qquad e_L \overset{f0_{cb}}{\rightsquigarrow} e_{Args}}{e_{APP} \xrightarrow{ambref} Function(Val(e_x), \bot, Length(e_L))}\; e_x \in E_{Atom},\ e_z \in E \setminus E_{Atom},\ e_L \in E_{List},\ Length(e_L) \in \mathbb{N} \tag{APP3}$$

$$\frac{e_x \overset{0f_{cb}}{\rightsquigarrow} e_{MN} \qquad e_y \overset{0f_{cb}}{\rightsquigarrow} e_{FN} \qquad e_L \overset{f0_{cb}}{\rightsquigarrow} e_{Args}}{e_{APP} \xrightarrow{ambref} Function(Val(e_x), Val(e_y), \bot))}\; e_x \in E_{Atom},\ e_y \in E_{Atom},\ e_L \in E_{List},\ Length(e_L) = \bot \tag{APP4}$$

$$\frac{e_x \overset{0f_{cb}}{\rightsquigarrow} e_{MN} \qquad e_y \overset{0f_{cb}}{\rightsquigarrow} e_{FN} \qquad e_{List} \overset{f0_{cb}}{\rightsquigarrow} e_{Args}}{e_{APP} \xrightarrow{ambref} Function(Val(e_x), Val(e_y), Arity)}\; e_x, e_y \in E_{Atom},\ e_{List} \in E_{List},\ Length(e_{List}) \in \mathbb{N}^{\geq} \tag{APP5}$$

$$\text{where} \quad Arity = min\{Length(e_{List}) \mid e_{List} \in E_{List} \wedge AL(e_{List})\}$$

Extending the call graph

Let us define the function

$$Body : SemFun \mapsto \mathcal{P}\,(E)$$

where $Body(f)$ contains all expressions that are inside the function body of f (if f is actually defined in the code). If f has no definition, then $Body(f)$ is an empty set.

Consider a function call expression e and the function entity that contains this expression (that is, $f \in SemFun$ and $e \in Body(f)$). The basic part of the call graph is built upon the information gathered about static MFA-calls. Namely, if the expression inside f refers to the function f', the call graph contains an edge from f to f'.

$$\frac{e \xrightarrow{funref} f'}{f \xrightarrow{funcall} f'}\; e \in Body(f)$$

However, in our representation there are other kinds of function calls registered, so to be able to distinguish the different call types, we label the edges of the call graph. Static calls are labelled by *funcall*.

Another group of function calls that we can successfully identify is the unambiguous dynamic call. In such calls the identifiers of the callee may be defined not at the call but at another program part, provided that they are fully calculable with data-flow reaching.

$$\frac{e \xrightarrow{dynref} f'}{f \xrightarrow{dyncall} f'}\; e \in Body(f)$$

In our call graph the unambiguous dynamic calls are labelled by *dyncall*. These references are as certain as static ones are, that is, neither approximation nor heuristics are applied during analysis.

The third kind of analysed references are ambiguous references. A such function reference always points to an opaque function entity, which is not fully defined. We do not include opaque functions into the call graph, instead, we associate such functions with fully defined ones. The latter relation is called *may_be*. Consequently, the combination of the ambiguous reference and the *may_be* relation determines the ambiguous function calls.

$$\frac{e \xrightarrow{ambref} f' \quad f' \xrightarrow{may\_be} f''}{f \xrightarrow{ambcall} f''}\; e \in Body(f)$$

By applying this rule, a function with an ambiguous call expression will be linked to all the functions the call may refer to. Apparently, a call may only refer to exactly one function, however, static analysis cannot determine which of the ambiguously called functions will be actually called at run-time.

$$\frac{\mathit{mfa}(f) = (\bot, n, a) \lor \mathit{mfa}(f) = (m, \bot, a) \lor \mathit{mfa}(f) = (m, n, \bot) \qquad \mathit{mfa}(f') = (m, n, a)}{f \xrightarrow{may\_be} f'} \; m, n \in \mathit{Atom}, a \;\in \mathbb{N} \tag{MAY1}$$

$$\frac{\mathit{mfa}(f) = (m, n, i^{\geq}) \qquad \mathit{mfa}(f') = (m, n, j)}{f \xrightarrow{may\_be} f'} \; m, n \in \mathit{Atom}, i, j \in \mathbb{N}, j \geq i \tag{MAY2}$$

4. Use cases in the RefactorErl refactoring tool

By refining the call graph with dynamic invocations, we get a deeper and more accurate insight into the inter-procedural relationships of the system under analysis. While performing different code analysis, refactoring transformations, or cyclic dependency examination, we can utilise the refined function call information. Basically, the refined call relation influences the preciseness of almost every function-related refactoring steps and code analysis (also including code clustering, whose result highly depends on call relations).

Side effect analysis. So far, in the case of expressions containing dynamic calls, we could not decide whether they have side effects or not, since we did not know which function is being called within the expression (potentially having side effects). By default, the Refactorerl tool regards every dynamic call as side effected until any analysis has successfully proved the contrary.

With the new call analysis results we are able to refine the static analysis of side effects. When an unambiguous dynamic call is recognised, we can identify the callee and propagate its side effects related properties. In the example below we have a function named *f* which calls the function *foo* via an apply-call. If *foo* is provably side effect free, then *f* is certainly side effect free as well.

```
f(S) -> apply(m, foo, [S]).
```

Refactoring. The result of the side effect analysis obviously has impact on code refactoring transformations, since they are based on the static code analysis. For instance, when we reorder the arguments of a function, the actual parameters in the calls to this function should also be reordered. However, if any of the parameters might have side effects, we do not allow the transformation, as it might violate the behaviour preservation principle of refactoring (note that in Erlang, the arguments of a call are evaluated strictly from left to right). Now with the help of dynamic function call analysis we can reduce the number of the expressions with indeterminate dirtiness, thus we can allow much more transformations to perform.

As an other example, in case of renaming a function we should replace all occurrences of the old function name with the new one. If we did not recognise the dynamic references of a function, we would not be able to change the function name inside those expressions and consequently we would completely modify the meaning of the program. Listing 9 shows a module whose function *plus* is called dynamically. We rename *plus* to *add* (Listing 10). If we did not have dynamic call analysis, no references would be renamed, resulting in undefined function calls and run-time errors.

5. Conclusions

In this paper we presented that the function analysis and the data-flow analysis can be effectively combined and we also defined how static function call graphs of Erlang programs can be extended with dynamic call references uncovered by use of data-flow analysis. We successfully classified the dynamic call constructs of the language both on the syntax level and based on the way the function identifiers and call arguments are constructed and passed to the call. We defined static and dynamic calls, as well as MFA-calls, apply-calls, whose syntax definition was formally discussed. Also, we formally defined the connection between dynamic call expressions and their callee, and we introduced the concept

Listing 9. Original

```
-module(m).

plus(A,B) -> A + B.

f(A,B) -> apply(m,plus,[A,B]),
          Fun = plus,
          m:Fun(A,B).
```

Listing 10. Renaming plus to add

```
-module(m).

add(A,B) -> A + B.

f(A,B) -> apply(m,add,[A,B]),
          Fun = add,
          m:Fun(A,B).
```

of ambiguous calls. Ambiguous references are not calculable at compile-time and therefore are not fully identified by static analysis, but we presented a method that represents ambiguous calls by opaque function entities and *may_be* relations in order to include them into the call graph. Finally, we precisely formalised how the newly defined references can refine the static call graph.

6. Related results

As far as we are aware of existing static analyser tools for the Erlang programming language, only TypEr (Type Annotator of Erlang Code [8]) extends its call graph — built from static function calls — with a subset of possible dynamic calls. Unlike other Erlang type analyses (e.g. soft-typing by Nystrom [9] and sub-typing by Marlow and Wadler [10]), the success typing method of TypEr is present in the Erlang/OTP environment. In case of dynamic MFA-calls it tries to use the result of a kind of data-flow analysis to gather out the module and function names. In contrast to our analysis method, it extends the call graph with a dynamic call only if both the module and function names are clearly deducible, while apply-calls and ambiguous calls are completely ignored.

Other dynamically typed functional (and scripting) languages provide similar dynamic function call constructs. There are a large number of papers investigating static analysis and typing of dynamically typed languages. For JavaScript [11] as well as for Scheme [12] well-defined typing, data- and control-flow analysis methods have been developed. Nevertheless, to our best knowledge, none of these has detailed the consideration of dynamic call constructs.

In imperative languages, a commonly applied method to construct dynamic calls is using function pointers (variables that point to the address of functions). Building call graphs in the face of pointers requires points-to analysis to provide accurate results, which means we have to estimate the contents of pointer variables by propagating pointer assignments, copies, and arithmetic across program data flows. There is a large body of theoretical work on various pointer analyses [13] and their application for call-graph construction with different degrees of cost and precision [14, 15].

7. Future work

We presented how data-flow analysis can help to refine the function call graph, however, data-flow relations can also be adjusted according to the new information about dynamic calls. Consequently, the data-flow analysis and the function analysis mutually influence each other. It would be interesting to define an iterative algorithm that produces more and more precise data-flow information and call graph, to examine the cost and result of each iteration step, and to give a reasonable termination condition.

When function references cannot be identified with the use of data-flow analysis either, we may apply analyses taking run-time details into account. Possibilities include symbolic evaluation as well as dynamic analysis.

Further development steps could deal with *eval* expressions, which are applicable for evaluating any Erlang code stored in a string. Such constructs could result in additional dynamic function calls, as the evaluated string may contain function calls to be analysed in some way. The static analysis of *eval* constructs present in Erlang is an open question.

Acknowledgement

We would like to thank Zoltán Horváth, Róbert Kitlei and Melinda Tóth for their help and support given during our research. The project was supported by TECH_08_A2-SZOMIN08.

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, July 1999.

[2] "RefactorErl Home Page," 2011, http://plc.inf.elte.hu/erlang.

[3] F. Cesarini and S. Thompson, *ERLANG Programming*, 1st ed. O'Reilly Media, Inc., 2009.

[4] "Open Source Erlang," 2011, http://www.erlang.org.

[5] B. G. Ryder, "Constructing the Call Graph of a Program," *IEEE Trans. Softw. Eng.*, Vol. 5, No. 3, 1979, pp. 216–226.

[6] Z. Horváth *et al.*, "Modeling semantic knowledge in Erlang for refactoring," in *International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009, Selected papers*, ser. Studia Universitatis Babeş-Bolyai, Series Informatica, Vol. 54(2009) Sp. Issue, Cluj-Napoca, Romania, Jul 2009, pp. 7–16.

[7] M. Tóth, I. Bozó, Z. Horváth, and M. Tejfel, "1st order flow analysis for Erlang," in *8th Joint Conference on Mathematics and Computer Science, MACS 2010*, 2010.

[8] T. Lindahl and K. Sagonas, "Typer: a type annotator of erlang code," in *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, ser. ERLANG '05. New York, NY, USA: ACM, 2005, pp. 17–25. [Online]. http://doi.acm.org/10.1145/1088361.1088366

[9] S.-O. Nyström, "A soft-typing system for Erlang," in *Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, ser. ERLANG '03. New York, NY, USA: ACM, 2003, pp. 56–71. [Online]. http://doi.acm.org/10.1145/940880.940888

[10] S. Marlow and P. Wadler, "A practical subtyping system for Erlang," *SIGPLAN Not.*, Vol. 32, August 1997, pp. 136–149. [Online]. http://doi.acm.org/10.1145/258949.258962

[11] S. H. Jensen, A. Møller, and P. Thiemann, "Type Analysis for JavaScript," in *Proc. 16th International Static Analysis Symposium, SAS '09*, ser. LNCS, Vol. 5673. Springer-Verlag, August 2009.

[12] O. Shivers, "Control-Flow Analysis in Scheme," in *PLDI*, 1988, pp. 164–174.

[13] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '96. New York, NY, USA: ACM, 1996, pp. 32–41. [Online]. http://doi.acm.org/10.1145/237721.237727

[14] E. Horváth, I. Forgács, Ákos Kiss, J. Jász, and T. Gyimóthy, "General flow-sensitive pointer analysis and call graph," in *Proceedings of the Estonian Academy of Sciences, Engineering*, Vol. 11, December 2005, pp. 286–295.

[15] A. Milanova, A. Rountev, and B. G. Ryder, "Precise call graph construction in the presence of function pointers," In Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation, Tech. Rep., 2001.