System Dependence Graphs in Sequential Erlang[⋆]

Josep Silva, Salvador Tamarit and César Tomás

Universitat Politècnica de València, Camino de Vera s/n, E-46022 Valencia, Spain.
{jsilva,stamarit,ctomas}@dsic.upv.es

Abstract. The system dependence graph (SDG) is a data structure used in the imperative paradigm for different static analysis, and particularly, for program slicing. Program slicing allows us to determine the part of a program (called slice) that influences a given variable of interest. Thanks to the SDG, we can produce precise slices for interprocedural programs. Unfortunately, the SDG cannot be used in the functional paradigm due to important features that are not considered in this formalism (e.g., pattern matching, higher-order, composite expressions, etc.). In this work we propose the first adaptation of the SDG to a functional language facing these problems. We take Erlang as the host language and we adapt the algorithms used to slice the SDG to produce precise slices of Erlang interprocedural programs. As a proof-of-concept, we have implemented a program slicer for Erlang based on our SDGs.

1 Introduction

Program slicing is a general technique of program analysis and transformation whose main aim is to extract the part of a program (the so-called *slice*) that influences or is influenced by a given point of interest (called *slicing criterion*) [18, 15]. Program slicing can be dynamic (if we only consider one particular execution of the program) or static (if we consider all possible executions). While the dynamic version is based on a data structure representing the particular execution (a trace) [7, 1], the static version has been traditionally based on a data structure called *program dependence graph* (PDG) [4] that represents all statements in a program with nodes and their control and data dependencies with edges. Once the PDG is computed, slicing is reduced to a graph reachability problem, and slices can be computed in linear time.

Unfortunately, the PDG is imprecise when we use it to slice interprocedural programs, and an improved version called *system dependence graph* (SDG) [6] has been defined. The SDG has the advantage that it records the calling context of each function call and can distinguish between different calls. This allows us to define algorithms that are more precise in the interprocedural case.

[⋆] This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02 and by the *Generalitat Valenciana* under grant PROMETEO/2011/052. Salvador Tamarit was partially supported by the Spanish MICINN under FPI grant BES-2009-015019.

```
(1)  main() ->
(2)    Sum = 0,
(3)    I = 1,
(4)    {Result,_} = while(Sum,I,11),
(5)    Result.

(6)  while(Sum,I,Top) ->
(7)    if
(8)     I /= Top -> {NSum,NI} = a(Sum,I),
(9)                 while(NSum,NI,Top-1)
(10)    I == Top -> {Sum,Top};
(11)   end.

(12) a(X,Y) ->
(13)  {add(X,Y),
(14)  fun(Z)->add(Z,1) end(Y)}.

(15) add(A,0) -> A;
(16) add(A,B) -> A+B.
```

Fig. 1: Original Program

```
(1) main() ->
(2)
(3)  I = 1,
(4)  {_,_} = while(undef,I,11).
(5)

(6) while(_,I,Top) ->
(7)    if
(8)     I /= Top -> {_,NI} = a(undef,I),
(9)                 while(undef,NI,Top)
(10)
(11)   end.

(12) a(_,Y)->
(13)   {undef,
(14)   fun(Z)->add(Z,1) end(Y)}.

(15)
(16) add(A,B) -> A+B.
```

Fig. 2: Sliced Program

In this paper we adapt the SDG to the functional language Erlang. This adaptation is interesting because it is the first adaptation of the SDG to a functional language. Functional languages pose new difficulties in the SDG, and in the definition of algorithms to produce precise slices. For instance, Erlang does not contain loop commands such as `while`, `repeat` or `for`. All loops are made through recursion. In Erlang, variables can only be assigned once, and pattern matching is used to select one of the clauses of a given function. In addition, we can use higher-order functions and other syntactical constructs not present in imperative programs. All these features make the traditional SDG definition useless for Erlang, and a non-trivial redefinition is needed.

Example 1. The interprocedural Erlang program[1] of Figure 1 is an Erlang translation of an example in [6]. We take as the slicing criterion the expression `add(Z,1)` in line `(14)`. This means that we are interested in those parts of the code that might affect the value produced by the expression `add(Z,1)`. A precise slice w.r.t. this slicing criterion would discard lines `(2)`, `(5)`, `(10)` and `(15)`, and also replace some parameters by anonymous variables (represented by underscore), and some expressions by a representation of an undefined value (atom `undef`). This is exactly the result computed by the slicing algorithm described in this paper and shown in Figure 2. Note that the resulting program is still executable.

[1] We refer those readers non familiar with Erlang syntax to Section 3 where we provide a brief introduction to Erlang.

The structure of the paper is as follows. Section 2 presents the related work. Section 3 introduces some preliminaries. The Erlang Dependence Graph is introduced in Section 4, and the slicing algorithm is presented in Section 5. Finally, Section 6 presents some future work and concludes.

2 Related work

Program slicing has been traditionally associated with the imperative paradigm. Practically all slicing-based techniques have been defined in the context of imperative programs and very few works exist for functional languages (notable exceptions are [5, 13, 12]). However, the SDG has been adapted to other paradigms such as the object-oriented paradigm [8, 9, 17] or the aspect-oriented paradigm [21].

There have been previous attempts to define a PDG-like data structure for functional languages. The first attempt to adapt the PDG to the functional paradigm was [14] where they introduced the *functional dependence graph* (FDG). Unfortunately, the FDGs are useful at a high abstraction level (i.e., they can slice modules or functions), but they cannot slice expressions and thus they are insufficient for Erlang. Another approach is based on the *term dependence graphs* (TDG) [3]. However, these graphs only consider term rewriting systems with function calls and data constructors (i.e., no complex structures such as if-expressions, case-expressions, etc. are considered). Moreover, they are not able to work with higher-order programs. Finally, another use of program slicing has been done in [2] for Haskell. But in this case, no new data structure was defined and the abstract syntax tree of Haskell was used with extra annotations about data dependencies.

In [19, 20] the authors propose a flow graph for the sequential component of Erlang programs. This graph has been used for testing, because it allows us to determine a set of different flow paths that test cases should cover. Unfortunately, this graph is not based on the SDG and it does not contain the information needed to perform precise program slicing. For instance, it does not contain summary edges, and it does not decompose expressions, thus in some cases it is not possible to select single variables as the slicing criterion. However, this graph solve the problem of flow dependence and thus it is subsumed by our graphs. Another related approach is based on the *behavior dependency graphs* (BDG) [16] that has been also defined for Erlang. Even though the BDG is able to handle pattern matching, composite expressions and all constructs present in Erlang, it has the same problem as previous approaches: the lack of the summary edges [6] used in the SDG implies a loss of precision.

All these works have been designed for intra-procedural slicing, but they lose precision in the inter-procedural case. This problem can be solved with the use of a SDG. From the best of our knowledge, this is the first adaptation of the SDG to a functional language.

3 Preliminaries

In this section we introduce some preliminary definitions used in the rest of the paper. For the sake of concreteness, we will consider the following subset of the Erlang language:

$$
\begin{array}{lcll}
pr & ::= & \overline{f_n} & \text{(Program)} \\
f & ::= & \overline{atom\ fc_n} & \text{(Function Definition)} \\
fc & ::= & (\overline{p_m}) \to \overline{e_n} \mid (\overline{p_m})\ \texttt{when}\ \overline{g_o} \to \overline{e_n} & \text{(Function Clause)} \\
p & ::= & l \mid V \mid \langle \overline{p_n} \rangle \mid [\overline{p_n}] \mid p_1 = p_2 & \text{(Pattern)} \\
g & ::= & l \mid V \mid \langle \overline{g_n} \rangle \mid [\overline{g_n}] \mid g_1\ op\ g_2 \mid op\ g & \text{(Guard)} \\
e & ::= & l \mid V \mid \langle \overline{e_n} \rangle \mid [\overline{e_n}] \mid \texttt{begin}\ \overline{e_n}\ \texttt{end} & \text{(Expression)} \\
 & \mid & e_1\ op\ e_2 \mid op\ e \mid e(\overline{e_n}) \mid p = e & \\
 & \mid & [e \mid\mid \overline{gf_n}] \mid \texttt{if}\ \overline{ic_n}\ \texttt{end} \mid \texttt{case}\ e\ \texttt{of}\ \overline{cc_n}\ \texttt{end} & \\
 & \mid & \texttt{fun}\ atom/number \mid \texttt{fun}\ \overline{fc_n}\ \texttt{end} & \\
l & ::= & number \mid string \mid atom & \text{(Literal)} \\
gf & ::= & p \leftarrow e \mid e & \text{(Generator | Filter)} \\
ic & ::= & \overline{g_m} \to \overline{e_n} & \text{(If Clause)} \\
cc & ::= & p \to \overline{e_n} \mid p\ \texttt{when}\ \overline{g_m} \to \overline{e_n} & \text{(Case Clause)} \\
op & ::= & + \mid - \mid * \mid / \mid div \mid rem \mid ++ \mid -- & \text{(Operation)} \\
 & \mid & not \mid and \mid or \mid xor \mid == \mid /= & \\
 & \mid & =< \mid < \mid >= \mid > \mid =:= \mid =/= &
\end{array}
$$

An Erlang program is a collection of function definitions. Note that we use the notation $\overline{f_n}$ to represent the sequence $f_1 \ldots f_n$. Each function definition is formed in turn by a sequence of n pairs $atom\ fc$ where $atom$ is the name of the function with arity n and fc is a function clause. Function clauses are formed by a sequence of patterns enclosed in parentheses followed optionally by a sequence of guards, and then an arrow and a sequence of expressions (e.g., $f(X,\ Y,\ Z)$ `when` $X > 0; Y > 1; Y < 5\ \to\ X+Y, Z.$). A pattern can be a literal (a number, a string, or an atom), a variable, a compound pattern or a tuple or list of other patterns. Guards are similar to patterns, but they must evaluate to represent a boolean value, and they do not allow compound patterns. Expressions can be literals, variables, tuples, lists, blocks composed of sequences of expressions, operations, applications, pattern matching, list comprehensions, if-expressions and case-expressions, function identifiers and declarations of anonymous functions, which are formed by a sequence of function clauses as in function definitions. In Erlang, when a call to a function is evaluated, the compiler tries to do pattern matching with the first clause of the associated function definition and it continues with the others until one succeeds. When pattern matching succeeds with a clause then its body is evaluated and the rest of clauses are ignored. If no clause succeeds then an error is raised.

In the following we will assume that each syntactic construct of a program (e.g., patterns, guards, expressions, etc.) can be identified with *program positions.* Program positions are used to uniquely identify each element of a program. In

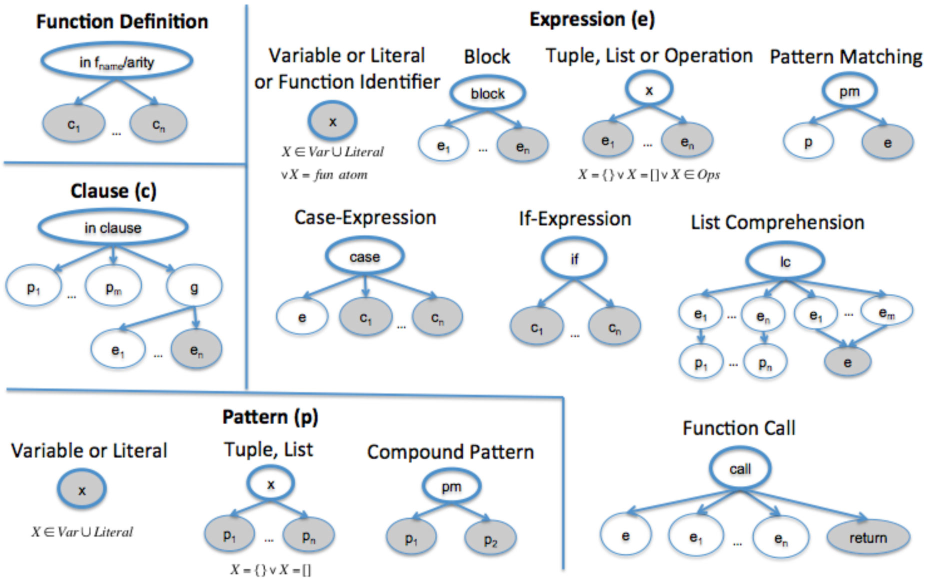

Fig. 3: Graph representation of Erlang programs

particular, the program position of an element identifies the row and column where it starts, and the row and column where it ends. We also assume the existence of a function *elem* that returns the element associated to a given program position. Additionally, we use the finite sets *Vars*, *Literal*, *Ops* and $\mathcal{P}$ that respectively contain all variables, literals, operators and positions in a program.

4 Erlang Dependence Graphs

In this section we adapt the SDG to Erlang. We call this adaptation Erlang dependence graph. Its definition is based on a graph representation of the components of a program that is depicted in Figure 3.

Figure 3 is divided into four sections: function definitions, clauses, expressions and patterns. Each graph in the figure represents a syntactical construct, and they all can be composited to build complex syntactical definitions. The composition is done by replacing some nodes by a particular graph. In particular, nodes labeled with `c` must be replaced by a clause graph. Nodes labeled with `e` must be replaced by one of the graphs representing expressions or function definitions (for anonymous functions). And nodes labeled with `p` must be replaced by one of the graphs representing patterns. In order to replace one node by a graph, we connect all input arcs of the node to the initial node of the graph that is represented with a bold line; and we connect all output arcs of the node to the final nodes of the graph that are the dark nodes. Note that in the case that a final node is replaced by a graph, then the final nodes become recursively the dark nodes of this graph. We explain each graph separately:

Function Definition: The initial node includes information about the function name and its arity. The value of f_{name} is $\perp$ for all anonymous functions. Clauses are represented with $\mathtt{c_1} \ldots \mathtt{c_n}$ and there must be at least one.

Clause: They are used by functions and by case- and if-expressions. In function clauses each clause contains zero or more patterns ($\mathtt{p_1} \ldots \mathtt{p_m}$) that represent the arguments of the function. In case-expressions each clause contains exactly one pattern and in if-expressions no pattern exists. Node `g` represents all the (zero or more) guards in the clause. If the clause does not have guards it contains the empty list []. There is one graph for each expression ($\mathtt{e_1} \ldots \mathtt{e_n}$) in the body of the clause.

Variable/Literal: They can be used either as patterns or as expressions, and they are represented by a single (both initial and final) node.

Function Identifier: It is used for higher order calls. It identifies a function with its name and its arity and it is represented by a single (both initial and final) node.

Pattern Matching/Compound Pattern: It can be used either as a pattern or as an expression. The only difference is that if it is a pattern, then the final nodes of both subpatterns are the final nodes. In contrast, if it is an expression, then only the final nodes of the subexpression are the final nodes.

Block: It contains a number of expressions ($\mathtt{e_1} \ldots \mathtt{e_n}$), being the final nodes the last nodes of the last expression ($\mathtt{e_n}$).

Tuple/List/Operation: Tuples and lists can be patterns or expressions. Operations can only be expressions. The initial node is the tuple ({}), list ([]) or operator (+, $*$, etc.) and the final nodes are the final nodes of all participating expressions ($\mathtt{e_1} \ldots \mathtt{e_n}$).

Case-Expression: The evaluated expression is represented by `e`, and the last nodes of its clauses are its final nodes.

If-Expression: Similar to case-expressions but missing the evaluated expression.

Function Call: The function is represented by `e`, the arguments are $\mathtt{e_1} \ldots \mathtt{e_n}$ and the final node is the `return` node that represents the output of the function call.

List Comprehension: A list comprehension contains n generators formed by an expression and a pattern; m filters ($\mathtt{e_1} \ldots \mathtt{e_m}$) and the final expression (`e`).

Definition 1 (Graph Representation). *The graph representation of an Erlang program is a labelled graph* $(\mathcal{N}, \mathcal{C})$ *where* $\mathcal{N}$ *are the nodes and* $\mathcal{C}$ *are the edges. Additionally, the following functions are associated to the graph:*

$type : \mathcal{N} \to \mathcal{T}$
$pos : \mathcal{N} \to \mathcal{P}$
$function : \mathcal{N} \to (atom, number)$
$child : (\mathcal{N}, number) \to \mathcal{N}$

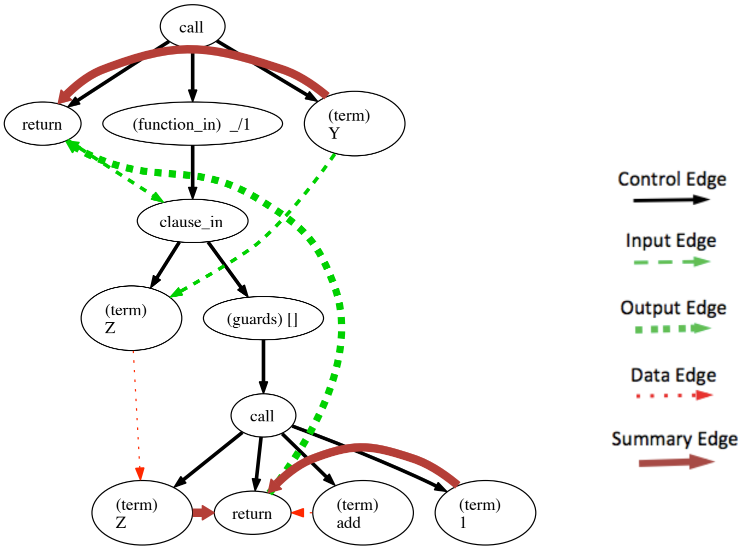

Fig. 4: EDG associated to expression `fun(Z)->add(Z,1) end(Y)` of Example 1.

$children : \mathcal{N} \to \{\mathcal{N}\}$
$lasts : \mathcal{N} \to \{\mathcal{N}\}$
$rootLasts : \mathcal{N} \to \{\mathcal{N}\}$

For each function of the program there is a function definition graph that is compositionally constructed according to the cases of Figure 3.

Total function *type* returns the type of a node. $\mathcal{T}$ is the set of node types: `function_in`, `clause_in`, `pm`, `guards`, `fid` (function identifier), `var`, `lit`, `block`, `case`, `if`, `tuple`, `list`, `op`, `call`, `lc`, and `return`. The total function *pos* returns the program position associated to a node. Partial function *function* is defined for nodes of types `function_in`, and it returns a tuple containing the function name and its arity. Function *child* returns the child that is in the position specified by the inputed number of a given node. Function *children* returns all the children of a given node. Given a node in the EDG, function *lasts* returns the final nodes associated to this node (observe that these nodes will always be leafs). Finally, given a node in an EDG that is associated to one of the graphs in Figure 3, function *rootLast* returns for each final node of this graph, (1) the initial node of the graph that must replace this node (in the case the node is gray), or (2) the node itself (in the case the node is white). This function is useful to collect the initial nodes of all arguments of a function clause.

Example 2. The graph in Figure 4 has been automatically generated by our implementation, and it illustrates the composition of some graphs associated to the code in Example 1. This graph corresponds to the function call `fun(Z)-> add(Z,1) end(Y)`. For the time being, the reader can ignore all dashed, dotted and bold edges. In this graph, the final nodes of the `call` nodes are their respective `return` nodes. Also, the result produced by function *rootLast* taking the `clause_in` node as input is the `call` node that is its descendant.

4.1 Control Edges

The graph representation of a program implicitly defines the control dependence between the components of the program.

Definition 2 (Control Dependence). *Given the graph representation of an Erlang program* $(\mathcal{N}, \mathcal{C})$ *and two nodes* $n, n' \in \mathcal{N}$*, we say that* n' *is* control dependent *on* n *if and only if* $(n \to n') \in \mathcal{C}$*.*

In Figure 4, there are control edges, e.g., between nodes $clause_{in}$ and *tuple*.

4.2 Data Edges

The definition of data dependence in Erlang is more complicated than in the imperative paradigm mainly due to pattern matching. Data dependence is used in four cases: (i) to represent the flow dependence between the definition of a variable and its later use (as in the imperative paradigm), (ii) to represent the matches in pattern matching, (iii) to represent the implicit restrictions imposed by patterns in clauses, and (iv) to relate the name of a function with the result produced by this function in a function call. Let us explain and define each case separately.

Dependence produced by flow relations. In the imperative paradigm data dependence relations are due to flow dependences. These relations also happen in Erlang. As usual it is based on the sets $Def(n)$ and $Use(n)$ [15] that in Erlang contain the (single) variable defined (respectively used) in node $n \in \mathcal{N}$.

Given two nodes $n, n' \in \mathcal{N}$, we say that n' is *flow dependent* on n if and only if $Def(n) = Use(n')$ and n' is in the scope of n. We define the set $\mathcal{D}_f$ as the set containing all data dependencies of this kind, i.e., $\mathcal{D}_f = \{(n, n') \mid n'$ is *flow dependent* on $n\}$.

As an example, there is a data dependence of type $\mathcal{D}_f$ between the pairs of nodes containing variables `Z` in Figure 4, and between variables `A` in Figure 5.

Dependence produced by pattern matching. In this section, when we talk about pattern matching, we refer to the matching of an expression against a pattern. For instance, the graph of `{X,Y}` matches the graph of `{Z,42}` with three matching nodes: {} with {}, `X` with `Z` and `Y` with 42. Also, the graph of the expression `if X>1 -> true; _ -> false end` matches the graph of the pattern `Y` with two matching nodes: `Y` with `true` and `Y` with `false`. Pattern matching is

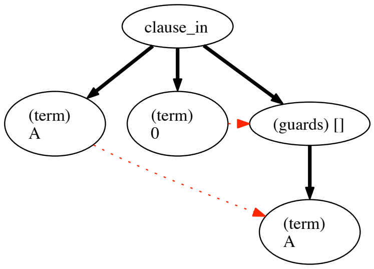

Fig. 5: EDG associated to clause `add(A,0) -> A` of Example 1.

used in three situations, namely, (i) in case-expressions to match the expression against each of the patterns, (ii) in pattern-matching-expressions, and (iii) in function calls to match each of the parameters to the arguments of the called function. Here we only consider the first two items because the third one is represented with another kind of edge that will be discussed in Section 4.3. Given the initial node of a pattern (say n_p) and the initial node of an expression (say n_e) we can compute all matching pairs in the graph with function *match* that is recursively defined as:

$$
\begin{array}{l}
\mathit{match}(n_p, n_e) = \\
\{(n_e, n_p) \mid \mathit{type}(n_e) = \texttt{var} \lor \\
\qquad\qquad (\ \mathit{type}(n_p), \mathit{type}(n_e) \in \{\texttt{lit}, \texttt{fid}\} \\
\qquad\qquad\quad \land \ \mathit{elem}(\mathit{pos}(n_p)) = \mathit{elem}(\mathit{pos}(n_e)) \ )\} \cup \\
\{(\mathit{last}_e, n_p) \mid (\ \mathit{type}(n_p) = \texttt{var} \lor \\
\qquad\qquad\qquad (\mathit{type}(n_p) = \texttt{lit} \ \land \ \mathit{type}(n_e) \in \{\texttt{op}, \texttt{call}\}) \lor \\
\qquad\qquad\qquad (\mathit{type}(n_p) = \texttt{tuple} \ \land \ \mathit{type}(n_e) = \texttt{call}) \lor \\
\qquad\qquad\qquad (\mathit{type}(n_p) = \texttt{list} \ \land \ \mathit{type}(n_e) \in \{\texttt{op}, \texttt{call}, \texttt{lc}\}) \ ) \\
\qquad\qquad\quad \land \ \mathit{last}_e \in \mathit{lasts}(n_e)\} \cup \\
\{\mathit{edge} \mid (\ (\mathit{type}(n_p) \in \{\texttt{lit}, \texttt{tuple}, \texttt{list}\} \ \land \ \mathit{type}(n_e) \in \{\texttt{case}, \texttt{if}, \texttt{pm}, \texttt{block}\}) \\
\qquad\quad \land \ \mathit{edge} \in \bigcup_{n'_e \in \mathit{rootLasts}(n_e)} \mathit{match}(n_p, n'_e) \ ) \lor \\
\qquad (\ \mathit{type}(n_p) = \texttt{pm} \ \land \ \mathit{edge} \in \bigcup_{n'_p \in \mathit{rootLasts}(n_p)} \mathit{match}(n'_p, n_e) \ )\} \cup \\
\{\mathit{edge} \mid \mathit{type}(n_p) \in \{\texttt{tuple}, \texttt{list}\} \ \land \ \mathit{type}(n_e) = \mathit{type}(n_p) \\
\qquad \land \ |\{n' \mid (n_e \rightarrow n') \in \mathcal{C}\}| = |\{n' \mid (n_p \rightarrow n') \in \mathcal{C}\}| \\
\qquad \land \ \bigwedge_{i \in 1 \ldots |\mathit{children}(n_e)|} \mathit{match}(\mathit{child}(n_p, i), \mathit{child}(n_e, i)) \neq \emptyset \\
\qquad \land \ \mathit{edge} \in (\ (n_e, n_p) \ \cup \ (\bigcup_{i \in 1 \ldots |\mathit{children}(n_e)|} \mathit{match}(\mathit{child}(n_p, i), \mathit{child}(n_e, i)) \ )\}
\end{array}
$$

The set of all pattern matching edges in a graph is denoted with $\mathcal{D}_{pm}$.

Dependence produced by restrictions imposed by patterns. The patterns that appear in clauses can impose restrictions to the possible values of the expressions that can match these patterns. For instance, the patterns used in the function definition `foo(X,X,0,Y) -> Y` impose two restrictions that must be fulfilled in order to return the value `Y`: (1) The first two arguments must be equal, and (2) the third argument must be `0`.

These restrictions can be represented with an arc from the pattern that imposes a restriction to the guards node of the clause; meaning that, in order to reach the guards, the restrictions of the nodes that point to the guards must be fulfilled. The set of all restrictions in a graph is denoted with $\mathcal{D}_r$, and it can be easily computed with function *constraints* that takes the initial node of a pattern and the set of repeated variables in the parameters of the clause associated to the pattern, and it returns all nodes in the pattern that impose restrictions.

$$
\mathit{constraints}(n, \mathit{RVars}) = \begin{cases} \{n\} & \mathit{type}(n) = \texttt{lit} \ \lor \\ & (\mathit{type}(n) = \texttt{var} \ \land \ \mathit{elem}(\mathit{pos}(n)) \in \mathit{RVars}) \\ \{n\} \cup \bigcup\limits_{n' \in \mathit{children}(n)} \mathit{constraints}(n', \mathit{RVars}) & \mathit{type}(n) \in \{\texttt{list}, \texttt{tuple}\} \\ \bigcup\limits_{n' \in \mathit{children}(n)} \mathit{constraints}(n', \mathit{RVars}) & \mathit{type}(n) = \texttt{pm} \\ \emptyset & \mathit{otherwise} \end{cases}
$$

As an example, there is a data dependence of type $\mathcal{D}_r$ between the term `0` and the guard node in Figure 5.

Dependence produced in function calls. The returned value of a function call always depends on the function that has been called. In order to represent this kind of dependence, we add an edge from any node that can represent the name of the function that is being called to the return node of the function call. Note that the name of the function is always represented by a node of type `atom`, `variable` or `fid`. We represent the set containing all dependences of this kind with $\mathcal{D}_{fc}$.

As an example, there is a data dependence of type $\mathcal{D}_{fc}$ between the node containing the literal `add` and the `return` node in Figure 4.

We are now in a position to define a notion of data dependence in Erlang.

Definition 3 (Data Dependence). *Given the graph representation of an Erlang program $(\mathcal{N}, \mathcal{C})$ and two nodes $n, n' \in \mathcal{N}$, we say that n' is* data dependent *on n if and only if $(n, n') \in (\mathcal{D}_f \cup \mathcal{D}_{pm} \cup \mathcal{D}_r \cup \mathcal{D}_{fc})$.*

4.3 Input/Output Edges

Input and output edges represent the information flow in function calls. One of the problems of functional languages such as Erlang is that higher-order calls can hide the name of the function that is being called. And even if we know the name of the function, it is not always possible to know the actual clause that will match the function call.

Example 3. In the following program, it is impossible to statically know what clause will match the function call `g(X)` and thus we need to connect the function call to all possible clauses that could make pattern matching at execution time.

```
-export(f/1).

f(X) -> g(X).

g(1)-> a;
g(X)-> b.
```

Determining all possible clauses that can pattern match a call is an undecidable problem because a call can depend on the termination of a function call, and proving termination is undecidable. Therefore, we are facing a fundamental problem regarding the precision of the graphs. Conceptually, we can assume the existence of a function `clauses`(`call`) that returns all clauses that match a given call. In practice, some static analysis must be used to approximate the clauses. In our implementation we use Typer [10] that uses the type inference system of Dialyzer [11] producing a complete approximation.

Given a graph $(\mathcal{N}, \mathcal{C})$ we define the set $\mathcal{I}$ of input edges as a set of directed edges. For each function call graph *call*, we make graph matching between each parameter subgraph in the call to each argument subgraph in the clauses belonging to *clauses*(*call*). There is an edge in $\mathcal{I}$ for each pair of nodes matching. Moreover, there is an edge from the `return` node of the call to the $\texttt{clause}_{\texttt{in}}$ node of the clause. As an example, in Figure 4, there are input edges from node with variable `Y` to node with variable `Z` and from the `return` node to the `clause_in` node.

Given a graph $(\mathcal{N}, \mathcal{C})$ we define the set $\mathcal{O}$ of output edges as a set of directed edges. For each function call graph *call* and each clause belonging to `clauses`(`call`). There is an edge in $\mathcal{O}$ for each final node of the clause graph to the `return` node of the call. As an example, in Figure 4, there is an output edge between the two `return` nodes.

4.4 Summary Edges

Summary edges are used to precisely capture inter-function dependences. Basically, they say what arguments of a function do influence the result of this function (see [6] for a deep explanation about summary edges). Given a graph $(\mathcal{N}, \mathcal{C})$, we define the set $\mathcal{S}$ of summary edges as a set of directed edges. As in the imperative paradigm, they can be computed once all the other dependencies have been computed. We have a summary edge between two nodes n, n' of the graph if n belongs to the graph representing (a part of) an argument of a function call, n' is the return-node of the function call, and there is an input edge starting at n. In Figure 4, the summary edges are all bold edges.

We are now in a position to formally introduce the Erlang Dependence Graphs.

Definition 4 (Erlang Dependence Graph). *Given an Erlang program $\mathcal{P}$, its associated* Erlang Dependence Graph (EDG) *is the directed labelled graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ where $\mathcal{N}$ are the nodes and $\mathcal{E} = (\mathcal{C}, \mathcal{D}, \mathcal{I}, \mathcal{O}, \mathcal{S})$ are the edges.*

Example 4. The EDG corresponding to the expression `fun(Z)-> add(Z,1) end(Y)` in line `(14)` of Figure 1 is shown in Figure 4.

5 Slicing Erlang Specifications

The EDG is a powerful tool to perform different static analysis and it is particularly useful for program slicing.

In this section we show that our adaptation of the SDG to Erlang keeps the most important property of the SDG: computing a slice from the EDG has a cost linear with the size of the EDG. This means that we can compute slices with a single traversal of the EDG. However, the algorithm used to traverse the EDG is not the standard one. We only need to make one small modification that allows us to improve precision.

One important advantage of the EDG with respect to the SDG is that it minimizes the granularity level. In the EDG all syntactical constructs are decomposed to the maximum (i.e, literals, variables, etc.). Contrarily, in the imperative paradigm, each node represents a complete line in the source code. Therefore, we can produce slices that allows us to know what parts of the program affect a given (sub)expression at any point.

Definition 5. *Given an EDG $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, a slicing criterion for $\mathcal{G}$ is a node $n \in \mathcal{N}$.*

In practice, the EDG is hidden to the programmer, and the slicing criterion is selected in the source code. In our implementation this is done by just highlighting an expression in the code. This action is automatically translated to a position that in turn is the identifier of one node in the EDG. This node is the input of Algorithm 1 that allows us to extract slices from an EDG. Essentially, Algorithm 1 first collects all nodes that are reachable from the slicing criterion following backwards all edges in $\mathcal{C} \cup \mathcal{D} \cup \mathcal{I}$. And then it collects from these nodes, all nodes that are reachable following backwards all edges in $\mathcal{C} \cup \mathcal{D} \cup \mathcal{O}$. In both phases, the nodes that are reachable following backwards edges in $\mathcal{S}$ are also collected, but only if they are connected to a node that belongs to the slice through an input edge.

The behavior of the algorithm is similar to the standard one except for the treatment of summary edges. In the SDG, summary edges go from the input parameters to the output parameters of the function and they are always traversed. Moreover, each of the parameters cannot be decomposed. In contrast,

Algorithm 1 Slicing interprocedural programs

```
Input: An EDG $\mathcal{G} = (\mathcal{N}, \mathcal{E} = (\mathcal{C}, \mathcal{D}, \mathcal{I}, \mathcal{O}, \mathcal{S}))$ and a slicing criterion $\mathcal{SC}$
Output: A collection of nodes $Slice \in \mathcal{N}$
return traverse(traverse($\{\mathcal{SC}\}, \mathcal{I}$), $\mathcal{O}$)

function traverse($Slice, X$)
  repeat
    $Slice = Slice \cup \{n' \mid (n' \to n) \in (\mathcal{C} \cup \mathcal{D} \cup X) \text{ with } n \in Slice\}$
                $\cup \{n_2 \mid (n_2 \to n_1) \in \mathcal{S} \wedge (n_2 \to n_3) \in \mathcal{I} \text{ with } n_1, n_3 \in Slice\}$
  until a fix point is reached
return $Slice$
```

in Erlang, the arguments of a function can be composite data structures, and thus, it is possible that only a part of this data structure influences the slicing criterion. Therefore, in function calls, we only traverse the summary edges if they come from nodes that are actually needed. The way to know that they are actually needed is to observe their outgoing input edge and know if the node pointed does belong to the slice. Of course this can only be known after having analyzed the function that is called.

Once we have collected the nodes that belong to the slice, it is easy to map the slice into the source code. For a program $\mathcal{P}$, the exact collection of positions (lines and columns) that belong to the slice is $\{pos(n) \mid n \in Slice(\mathcal{P})\}$ where function *Slice* implements Algorithm 1. In order to ensure that the final transformed program is executable, we also have to replace those expressions that are not in the slice by the (fresh) atom `undef` and those unused patterns by an anonymous variable. The result of our algorithm with respect to the program in Figure 1 is shown in Figure 2.

6 Conclusions and future work

This work adapts the SDG to be used with Erlang programs. Based on this adaptation, we introduce a program slicing technique that precisely produces slices of interprocedural Erlang programs. This is the first adaptation of the SDG for a functional language. Even though we implemented it for Erlang, we think that it can be easily adapted to other functional languages with slight modifications.

The slices produced by our technique are executable. This means that other analysis and tools could use our technique as a preprocessing transformation stage simplifying the initial program and producing a more accurate and reduced one that will predictably speed up the subsequent transformations. We have implemented a slicer for Erlang that generates EDGs, this tool is called `Slicerl` and it is publicly available at:

`http://kaz.dsic.upv.es/slicerl`

The current implementation of `Slicerl` accepts more syntactical constructs that those described in this paper. It is able to produce slices of its own code. Even though the use of summary edges together with the algorithm proposed provides a solution to the interprocedural loss of precision, there is still a loss of precision that is not faced by our solution. This loss of precision is produced by the expansion and compression of data structures.

Example 5. Consider the Erlang program at the left and the slicing criterion `Y` in line `(4)`:

```
(1)  main() ->               (1)  main() ->                (1)  main() ->
(2)     X={1,2},             (2)     X={1,2},              (2)     X={1,_},
(3)     {Y,Z}=X,             (3)     {Y,_}=X,              (3)     {Y,_}=X,
(4)     Y.                   (4)     Y.                    (4)     Y.
```

Our slicing algorithm produces the slice shown in the center. It is not able to produce the more accurate slice shown at the right because it losses precision.

The loss of precision shown in Example 5 is due to the fact that the EDG does not provide any mechanism to trace an expression when it is part of a data structure that is collapsed into a variable and then expanded again. In the example, there is a dependence between variable `Y` and variable `X` in line `(3)`, This dependence means "*The value of* `Y` *depends on the value of* `X`". Unfortunately, this is only partially true. The real meaning should be "*The value of* `Y` *depends on a part of the value of* `X`". We are currently defining a new dependence called *partial-dependence* to solve this problem. A solution to this problem has already been defined in [16]. Its implementation will be available soon in the webpage of `Slicerl`.

References

1. H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Programming Language Design and Implementation (PLDI)*, pages 246–256, 1990.
2. C. Brown. *Tool Support for Refactoring Haskell Programs.* PhD thesis, School of Computing, University of Kent, Canterbury, Kent, UK, 2008.
3. Diego Cheda, Josep Silva, and Germán Vidal. Static slicing of rewrite systems. *Electron. Notes Theor. Comput. Sci.*, 177:123–136, June 2007.
4. J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.
5. John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 379–392, New York, NY, USA, 1995. ACM.
6. Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions Programming Languages and Systems*, 12(1):26–60, 1990.
7. B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 29(3):155–163, 1988.

8. Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *Proceedings of the 18th international conference on Software engineering*, ICSE '96, pages 495–505, Washington, DC, USA, 1996. IEEE Computer Society.
9. D. Liang and M. J. Harrold. Slicing objects using system dependence graphs. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 358–, Washington, DC, USA, 1998. IEEE Computer Society.
10. Tobias Lindahl and Konstantinos F. Sagonas. Typer: a type annotator of erlang code. In Konstantinos F. Sagonas and Joe Armstrong, editors, *Erlang Workshop*, pages 17–25. ACM, 2005.
11. Tobias Lindahl and Konstantinos F. Sagonas. Practical type inference based on success typings. In Annalisa Bossi and Michael J. Maher, editors, *PPDP*, pages 167–178. ACM, 2006.
12. Claudio Ochoa, Josep Silva, and Germán Vidal. Dynamic slicing based on redex trails. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '04, pages 123–134, New York, NY, USA, 2004. ACM.
13. Thomas Reps and Todd Turnidge. Program specialization via program slicing. In *Proceedings of the Dagstuhl Seminar on Partial Evaluation, volume 1110 of Lecture Notes in Computer Science*, pages 409–429. Springer-Verlag, 1996.
14. Nuno F. Rodrigues and Luis S. Barbosa. Component identification through program slicing. In *In Proc. of Formal Aspects of Component Software (FACS 2005). Elsevier ENTCS*, pages 291–304. Elsevier, 2005.
15. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.
16. Melinda Tóth, István Bozó, Zoltán Horváth, László Lövei, Máté Tejfel, and Tamás Kozsik. Impact analysis of erlang programs using behaviour dependency graphs. In *Proceedings of the Third summer school conference on Central European functional programming school*, CEFP'09, pages 372–390, Berlin, Heidelberg, 2010. Springer-Verlag.
17. Neil Walkinshaw, Marc Roper, Murray Wood, and Neil Walkinshaw Marc Roper. The java system dependence graph. In *In Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 5–5, 2003.
18. M. Weiser. Program Slicing. In *Proceedings of the 5th international conference on software engineering*, pages 439–449. IEEE Press, 1981.
19. Manfred Widera. Flow graphs for testing sequential erlang programs. In *Proceedings of the 2004 ACM SIGPLAN workshop on Erlang*, ERLANG '04, pages 48–53, New York, NY, USA, 2004. ACM.
20. Manfred Widera and Fachbereich Informatik. Concurrent erlang flow graphs. In *In Proceedings of the Erlang/OTP User Conference 2005*, 2005.
21. Jianjun Zhao. Slicing aspect-oriented software. In *Proceedings of the 10th International Workshop on Program Comprehension*, IWPC '02, pages 251–260, Washington, DC, USA, 2002. IEEE Computer Society.