Project no. 288570

PARAPHRASE

Strategic Research Partnership (STREP)
PARALLEL **P**ATTERNS FOR **A**DAPTIVE **H**ETEROGENEOUS **M**ULTICORE **S**YSTEMS

Side Condition Analysis
D2.12

Due date of deliverable: 31st March 2014

Start date of project: October 1^{st}, 2011

Type: Deliverable
WP number: WP2
Task number: T2.4

Responsible institution: ELTE
Editor and editor's address: D. Horpácsi and T. Kozsik, ELTE

Project co-funded by the European Commission within the Seventh Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Executive Summary

Task 2.4 investigates comprehensive and reliable pattern discovery in Erlang programs. The aim of pattern discovery is to identify code fragments that can be effectively parallelized. In D2.10 [13], ELTE and ELTE-Soft provided an extensive collection of parallel pattern exemplars, whilst with the same deadline, in D2.11 [12], ELTE-Soft reported about their first results of automatic pattern discovery. Now in this current phase, based mainly on these two reports, we have identified and examined the analyses which are necessary to perform more advanced and reliable pattern discovery.

The analyses focus on

- which operations are performed element-wise on multiple (possibly large amount of) data items;
- what type of data these operations are applied on; and
- whether these operations may be executed simultaneously (in parallel) without changing the outcome of the computation.

Positioning of Deliverable D2.12

The positioning of this deliverable with respect to other deliverables in the work package can be seen in Figure 1.

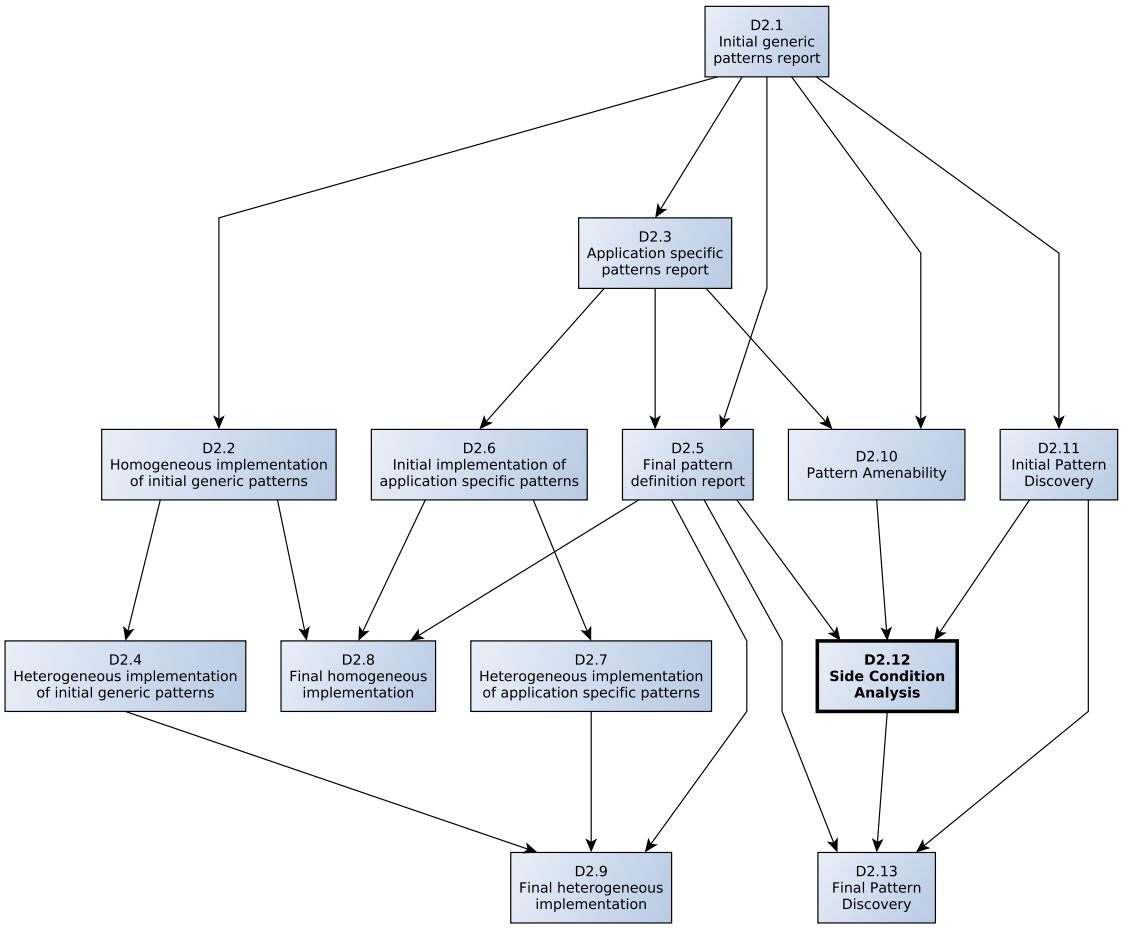

Figure 1: Positioning of deliverable D2.12 w.r.t. other WP2 deliverables

The positioning of this deliverable with respect to other deliverables in other packages can be seen in Figure 2.

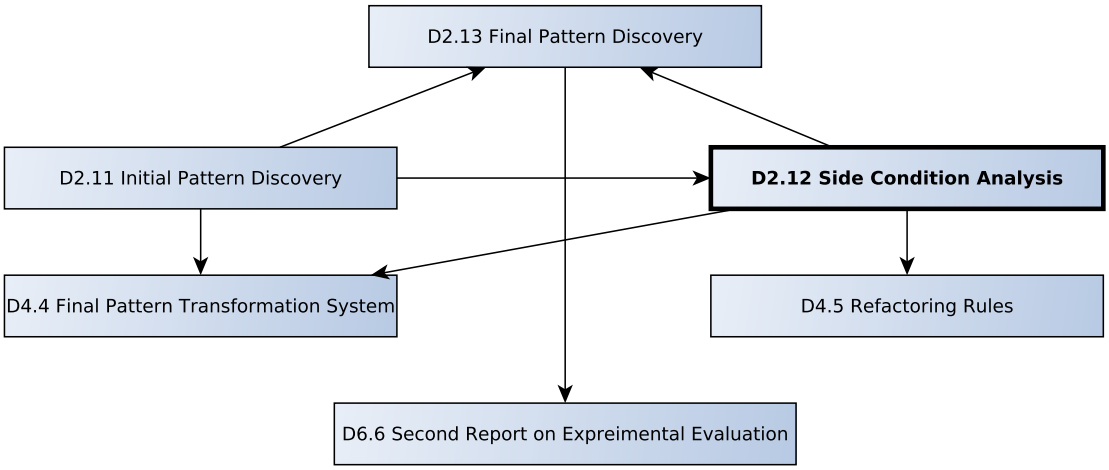

Figure 2: Positioning of deliverable D2.12 w.r.t. other deliverables

Contents

Chapter 1

Introduction

In this report we summarise our findings related to the static analyses that we have determined to be necessary for implementing comprehensive pattern discovery in Erlang programs. The aim of pattern discovery is to identify code fragments which are amenable to parallelization, and can be refactored (possibly after some program shaping) into applications of skeletons, i.e. calls to the functions of the *skel* library [7]. We discover pattern candidates by finding certain syntactic structures, and by verifying that they possess certain semantic properties (side conditions). Side conditions guarantee the validity of the transformations, i.e. that the transformations preserve the meaning of the code. Furthermore, they describe which transformations are worth to perform, i.e. which transformations yield significant performance gains. The discovery phase may result in many pattern candidates, but we must prioritize, and select those that offer the best parallel speedup in a given execution environment. In order to support the ranking of candidates, and help select the "best" options, we benchmark the (sequential) code, and estimate parallel speedup based on some cost model.

In the course of Task 2.1, as reported in D2.10, we have provided exemplars, which showed us which structures can be identified as pattern candidates, and how they should be transformed into parallel patterns. In the meantime, within Task 2.4, we have designed and implemented an initial version of pattern discovery (reported in D2.11). We could build upon these results in determining which analyses are required for an advanced and reliable pattern discovery.

What kind of analyses are required for pattern discovery? We can classify the analyses based on their use in the pattern discovery process.

Pattern spotting. Pattern discovery starts by finding occurrences of certain syntactic structures, such as list comprehensions and function definitions having some specific structure. These syntactic structures need to be further analyzed to see whether it is possible to transform them. Properties to inves-

tigate are: appropriate binding structure for variables, functions to be defined over lists and be recursive etc.

Safety check. Transformations should only be applied when they are guaranteed to preserve the meaning of the program. Side-effect analysis, for instance, is required to establish such guarantees.

Performance. Certain analyses can be used to make clever decisions on which transformations are worth to apply. Information on the type and the size of data, as well as input to heuristic decision making should be provided by these analyses.

The above goals can be achieved by a number of well-defined analyses, which will be explained in more details in the forthcoming chapter. Briefly, the analysis process is as follows. Firstly, we convert the code into an abstract syntax tree, on which we perform scope analysis as well as data-flow and control-flow analysis. On top of these, we build a call graph involving static and dynamic function applications and, based on data and control dependencies, we determine the set of expressions having side-effects. Moreover, we can infer the types of expressions based on the scope analysis and the function call graph.

By building on the results of the detailed static analysis, we are able to identify and assess pattern candidates in Erlang programs. The analysis steps explained in this document are implemented in the RefactorErl analysis framework; some of them had already been present in some form, but have been refined or improved in the scope of the ParaPhrase project.

Chapter 2

Support analyses to enable pattern discovery

The goal of the analyses is to support effective and reliable discovery and assessment of pattern candidates. In the previous chapter, we have pointed out three subgoals: firstly, we need to find those syntactic structures that can be transformed; secondly, we need to verify that the meaning of the program is not affected by the transformation; and thirdly, we need to prioritize pattern candidates. The analyses described in this chapter contribute to one or more of these subgoals.

We have found that the analyses on Figure 2.1 are required. These analyses logically depend on each other, and they can be implemented atop of each other as well. Indeed, we have implemented the analyses withing the RefactorErl analysis framework mostly in accordance with the shown logical dependencies. The only exception is type analysis, for which a Core Erlang [1] representation proved to be more useful.

Now let us summarize how the different analyses contribute to the achievement of the three subgoals.

Pattern spotting. When selecting transformable structures, certain syntactical elements (e.g. list comprehensions) are identified, on which further analyses must be performed to check required (non-syntactical) properties of the structure. We need *variable scope analysis* here to check the binding structure of used variables, to compute the set of free variables of an expression etc. *Data-flow analysis* and type information provided by the *type analysis* are used to determine whether a function in question is called on lists. Type analysis is also useful to infer information on the associativity and commutativity of operations, which properties are important in the case of parallel folding/reducing.

Safety. When we want to guarantee that the transformation of a structure does

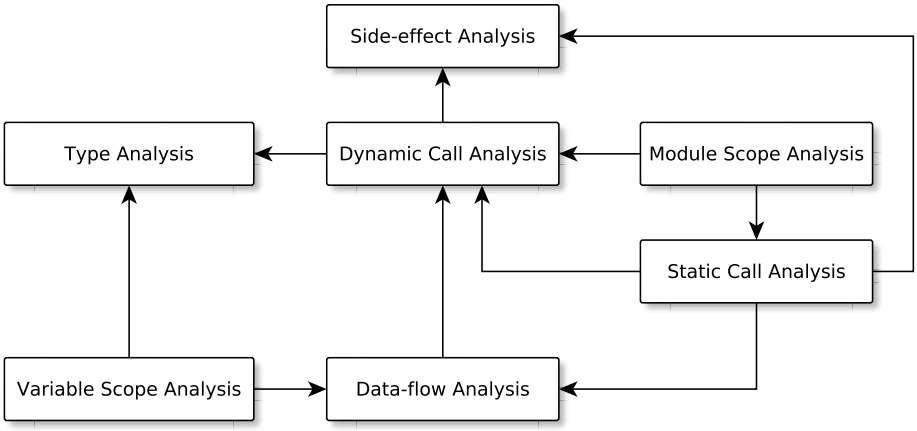

Figure 2.1: Support analysis to enable pattern discovery

not change the meaning of the program, the most important analysis is the *side-effect analysis*. The reason is that the key effect of the pattern transformations is parallelization, which means that the evaluation order changes. Expressions that were evaluated in a given order before, will be evaluated in a different order, or simultaneously. A change in the order of side-effects might prohibit a transformation; however, in certain cases, such as the order of log messages, it may also be tolerable. *Dynamic and static call analyses* are also needed for this subgoal, i.e. when we want to ensure that a transformation of a function definition impacts only those uses of the functions, where we want parallelization, and does not have undesired impact on other parts of the code.

Performance. When assessing whether a transformation is worth to apply, we need information on the type and the size of data, as well as on the computational complexity of operations. We also make use of parallel speedup estimates, obtained by benchmarking expressions and applying a cost model. For benchmarking randomly generated data of the appropriate type shall be used. All these demands are supported by the *type analysis*. Furthermore, *data-flow analysis* can be applied to predict the size of data sets in certain cases. Both this analysis and *function call analyses* are valuable for further heuristics, e.g. for finding multiple occurrences of a large data set (one that has already turned out to be suitable for data parallel processing), or for finding applications of a function that has already proved to be costly enough (and hence eligible for parallel evaluation).

Note that the *module scope analysis* is only needed indirectly for pattern discovery.

The rest of the chapter will go through these analyses in a logical order, respecting the dependencies presented on Figure 2.1. We start with the scope analyses (mod-

ule scope, static function call, variable scope). Then we turn to flow analysis, after that to dynamic call analysis, and then to side-effect analysis. Finally we describe type analysis. Before we discuss these semantic analyses, however, we make some notes on the first phase, syntactic analysis.

2.1 Syntactic analysis and program representation

Possible pattern candidates, at the very first approach, are spotted based on syntactic schemes: if a program fragment matches any of the schemes defined for the algorithmic skeletons, then it is considered eligible for further semantic analyses and checks.

In order to be able to match the program sections with syntactic patterns, we need to derive the syntactic structure (i.e. syntax tree) of the program first. This process is called parsing and typically yields an Abstract Syntax Tree (AST). Figure 2.2 shows an AST for an Erlang file, as represented in RefactorErl. The details of the labels attached to nodes and edges are not relevant for this document.

Further analysis and matching are performed on the AST (or on extensions of the AST) of the code. In RefactorErl, we apply a yecc[1]-generated [16] syntactic analyser for parsing, where the yecc-grammar is automatically generated from an enhanced EBNF [11] context-free description defining the syntactic rules as well as some key attributes to syntactic elements.

Nevertheless, in considerably large number of cases, syntactic patterns are not powerful enough to determine whether a code fragment indeed implements an instance of an algorithmic skeleton. In order to make more reliable decisions on pattern candidate selections, we perform semantic analyses to uncover context-dependent relations among program parts. The information acquired by semantic analyses is represented as an extension to the AST; that is, we enrich the syntax tree with *semantic nodes and edges* representing structural and semantic relations among syntax elements located arbitrarily far from each other in the tree. This process yields a graph that we call *Semantic Program Graph* (SPG). The SPG contains all the information required to reliably identify and assess pattern candidates. In the following sections, we give an overview of the analyses that can build the SPG for an Erlang program.

2.2 Scope analysis

The static semantic rules of a programming language describe validity requirements enforced by the compiler. One such static semantic question is how to re-

[1] yecc is a yacc-like parser generator implemented in Erlang

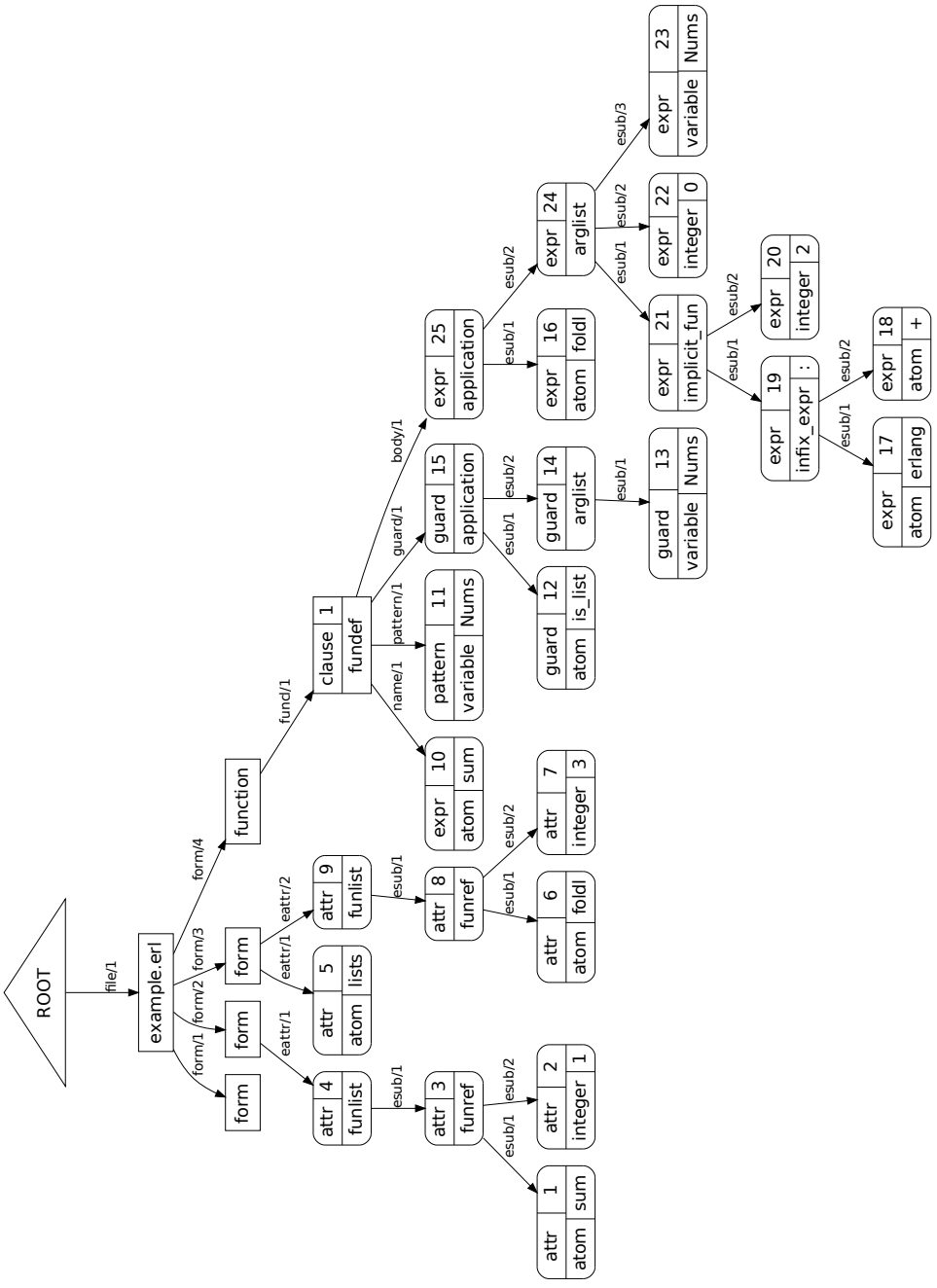

Figure 2.2: AST of the code listed in Figure 2.4

solve names. We need to introduce analyses to collect information on the uses of module, function and variable names in an Erlang program, according to the language rules – these analyses are similar to what the Erlang compiler does. In RefactorErl, we connect bindings and references of names by introducing semantic nodes. Both bindings and references are connected to these semantic nodes, for sharing information about a given program entity, let it be either a module, a function, a record or a variable.

2.2.1 Modules

The scope of modules is not restricted in Erlang, and in addition, module names are not resolved in compile-time at all. Thus, it is possible that, even if a whole application is loaded into the run-time system, there are undefined modules referred to. Our analysis strategy completely reflects this behaviour: regardless of whether a module is referred to or becomes defined in a software, we attach a semantic module node to the program location in question. Both definitions and references can introduce names, and it does not matter which comes first: even if the reference is discovered prior to the definition, they will refer to, and will be connected to, the same semantic node.

The following semantic edges are created for a module:

- The root node of the AST is linked to the module node with a *module* tag.
- When the module definition is found, the module is linked with a *moddef* tag to the node that represents the defining form.
- Every expression that references the module is linked to the module node with a *modref* tag.

Figure 2.3 demonstrates the representation on a simple code snippet. In particular, we present three semantic nodes, representing the modules `example` (which is defined here), `lists` (from the standard library) and `erlang` (which is not explicitly mentioned in the code, but referenced implicitly through the built-in function `is_list`; this module is auto-imported in all Erlang programs). The first *form* (i.e. definition) in the file introduces a new module, the second form imports the `lists:foldl/3` function (i.e. makes it visible with an unqualified name), the third form declares that function `sum/1` is exported (i.e. made visible from other modules), and the last two forms define two functions (`sum/1` and `add/2`). Function `sum/1` only accepts lists, and adds the elements of this list to `0` by folding the list with the `add/2` function. Observe how syntactic elements such as module definitions and module references are connected to the respective semantic node.

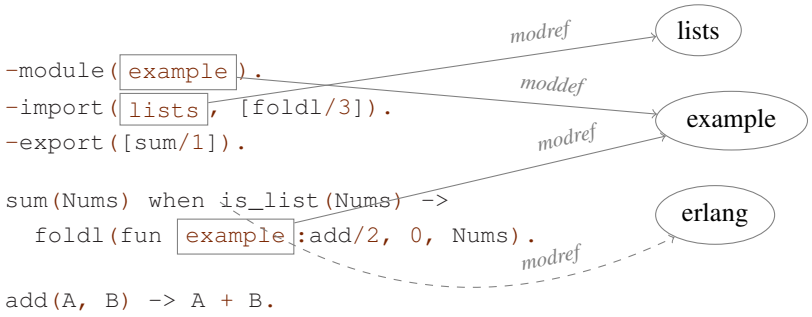

Figure 2.3: Module references

2.2.2 Functions

By default, the scope of a function is the module it is defined in. However, it is possible to export functions from a module: when a function is exported, its scope – similarly to that of its containing module – will be unrestricted. In this latter case, just like in the case of modules, names are not resolved by the compiler.

For functions, we follow the very same strategy shown with modules: both definitions and references can introduce function names, and it does not matter which comes first during the analysis process. In the case of module-local function names, our analysis, according to the static semantics of Erlang, does respect the restricted scope and only identifies references from inside the module. Function applications without a module-qualifier are regarded as calls to module-local or (auto)imported functions.

Representation Function definitions and explicit references create semantic function nodes in the semantic program graph. Such nodes capture the name and arity of the functions, and they carry information about the side-effect properties (to be discussed later). The following semantic edges are created for function nodes:

- The module node that contains the function definition is linked to the function node with a *func* tag.
- When the definition of a function is found, the function node is linked to the defining top level form with a *fundef* tag.
- When a module exports a function, the module node is linked to the corresponding function node with a *funexp* tag.
- When a module imports a function, the module node is linked to the corresponding function node with a *funimp* tag.
- Every expression that explicitly refers to the function is linked to the function

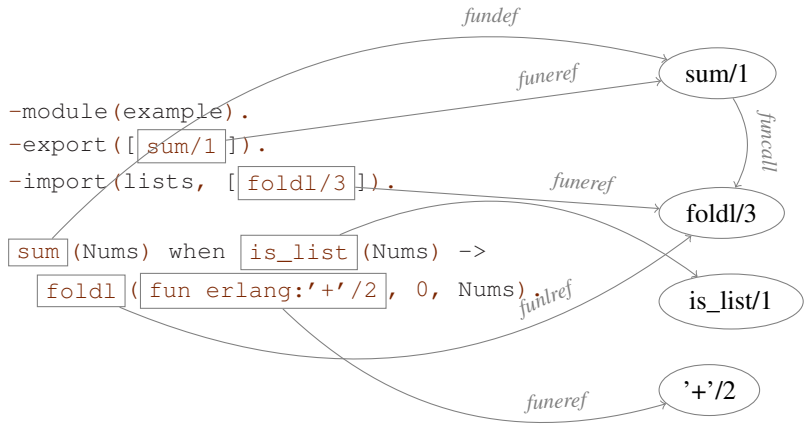

Figure 2.4: Function references

object with a *funlref* or *funeref* tag (local and external references).

Function call-graph Using the result of the function scope analysis (the expression-to-function references), we can also build the function call graph of the program. If there is a reference to function B in the body of function A, then we connect the semantic function nodes with a semantic edge labelled with *funcall*.

Figure 2.4 shows a program similar to the one used in Figure 2.3. In this example, we demonstrate the representation of function references. In particular, note that the export directive as well as the definition of function *sum* is linked to the same semantic function entity; similar conclusion could be drawn for *foldl*. Also, it is shown that these two functions are in a 'funcall' relation.

2.2.3 Variables

Unlike the simple rules for module and function scoping, variable scope and visibility is much more complex in Erlang. Variables are always bound with pattern matching, and their scope basically lasts until the end of the defining clause (either function or expression clause). Variables declared in branches of if-, case-, try- and receive-expressions are local to the defining branch (or more precisely, unsafe in global), unless they are defined in all branches (then their scope becomes that of the parent expression). On the other hand, generators in list and binary comprehensions as well as anonymous function arguments may hide names, as they open their own inner scope, and their patterns may shadow outer variables.

Structural edges In the syntactic model, the role of clauses is to group expression lists together. In the semantical structure there are two roles: clauses may limit

the visibility of variables, and some clauses may introduce new variables. Clauses that fall into the latter category are called scope clauses (as they define the scope of variables).

Besides the syntactic edges that belong to the abstract syntax tree, the following structural (semantic) edges are created.

- Every scope clause has an edge to the outermost scope clause with a *functx* tag (the outermost clauses are always function clauses).
- Every clause has an edge to its direct containing scope clause with a *scope* tag. This includes scope clauses as well, which therefore have a *scope* edge to themselves.
- Every expression is linked to its topmost super-expression that is accessible through *sub* edges with a *sup* tag.
- Every compound expression has an edge to its direct clauses (in proper order) with a *clause* tag.
- Every clause has an edge to any of its direct subexpressions (in proper order) with a *visib* tag. Note that these are the expressions that are accessible from subexpressions with a *sup* tag.

The last two kinds of edges may seem redundant, as they duplicate direct syntactic links. Their primary purpose is to make syntax tree traversal easier by defining a simple expression structure. The *visib* edges also facilitate the handling of variable visibility areas.

Representation A (semantic) *variable* node is created for every variable that occurs in the program. Such a node has an attribute *name*, which holds the variable name.

The scope and visibility rules for variables are much more complex than for functions, and the semantic edges show this complexity. The following structure is described by semantic edges:

- Every variable has a *scope*: a program part that is affected by the presence of the variable. The scope of a variable is defined by a scope clause. Scope clauses are function clauses, fun expression clauses, and list and binary generators in comprehensions. Scope clauses have the unique ability to shadow variables.
- A variable is *defined* by a scope when there is a binding directly in the scope (bindings in embedded scopes do not count here). Bindings are patterns where the variable is not visible.

- The subtree of a variable scope contains bodies (sequentially executed expression lists, modelled with clause nodes). A variable is *visible* in such a clause when there is an expression in the list that introduces the variable (binds it a value) or when the variable visibility is inherited from an outer clause. In the former case, the variable is visible from the expression after the introduction until the last expression, and in the latter case, it is visible in every expression (except when shadowing occurs in a scope clause).

This structure is described by the following edges.

- The scope clause node is linked to every variable node that is defined in it with a *vardef* tag.
- Every clause node is linked to every variable node that is visible in it with a *varvis* tag.
- The variable node is linked to every top level expression that introduces the variable to a clause with a *varintro* tag.
- Every variable expression that can bind the variable to a value is linked to the variable node with a *varbind* tag.
- Every variable expression that reads the value of a variable is linked to the variable node with a *varref* tag.

2.3 Data-flow and control-flow analysis

Data-flow analysis allows us to compute (a static approximation of) all possible values that an expression can take up during the executions of a given program. It can typically builds on the results of the control-flow analysis. Among others, we can use data-flow analysis to trace the propagation of list values in the program, and identify parallelizable computations on those lists.

Our framework is capable of performing data-flow and control-flow analysis on Erlang programs. The theoretical background of this process has already been documented in [9]. As a result of these analyses, the AST is extended with various semantic edges between expressions, which show data-flow and control-flow relations. We also implemented the transitive-reflexive closures of these relations, this way we can calculate *data-flow reaching* (both in 0^{th} and 1^{st} order, where the latter decreases over-approximation of the static analysis by taking callee contexts into account for value propagation through actual arguments of functions, trading efficiency for preciseness).

2.4 Dynamic call analysis

In Erlang, there are two syntactic ways to invoke a function: function call expressions and apply-calls (higher-order calls). In both cases, it is possible to specify the function to be applied in terms of Erlang *terms*; in these cases, the referred function is only determined at run-time. Function applications of this kind are called dynamic (in contrast to static calls). In order to analyse such applications statically, at compile-time, we utilise the results of the previously introduced data-flow analysis: we backtrack the data that flows into the terms specifying the function to be called. Also, in the case of apply-calls, the parameter count might be opaque as well, but we have a method to estimate the number of parameters passed to the function. The theoretical background of the whole process and the representation used are detailed in [4].

2.5 Side-effect analysis

An expression is defined to be side-effected (or impure) if it has an effect on, or is dependent on, the global state (or execution context) of the program. In general, we say that an expression is impure, if it reads or writes a shared variable, or accesses the file system or the standard IO; furthermore, message passing and VM manipulations are also forms of side-effect in Erlang. An expression having an impure subexpression or invoking an impure function is itself considered to be impure. On the other hand, a function is said to be impure if any of its expressions has side-effects. If an expression/function is free of side-effects, it is said to be pure, and it is referentially transparent.

Erlang is an impure functional programming language, pure and side-effected expressions are not separated by any means. In fact, it is a common programming practice to use side-effects in Erlang (indeed, many widely used libraries are based on message-passing). We need to accept the fact that most Erlang functions are not pure: in [8], the authors present statistics indicating only 5.3% of standard library functions being pure, i.e. completely side-effect free.

Our goal with static analysis is to uncover whether an arbitrary expression might have side-effects or not. In order to allow paraphrasing a reasonably wide range of programs, we do not deny side-effects, we only restrict them. After all, we are intended to determine whether the order of side-effects is changeable without altering the program behaviour – this statement is essential to enable parallel execution, since scheduling is not guaranteed to be deterministic.

2.5.1 Components

If we denied paraphrasing side-effected expressions, we would potentially exclude the most promising candidates. Therefore, we follow a more liberal strategy: do not disallow all kinds of side-effects, but only some combinations of them.

Pattern discovery identifies those code fragments that are fed into skeletons; these belong together and are regarded as "components" [14]. These are the series of expressions that are run in parallel on lots of input data.

In our concept, sub-expressions of components are not strictly required to be free of side-effects; rather, they may depend on the global state, and if they modify a shared variable, the effect has to be encapsulated, i.e. local to the component. If such a property holds, we say that the component is *sane*. Sanity of components will guarantee that the reordering of component executions will not change program behaviour.

2.5.2 White, gray and black entities

If an expression or function neither affects, nor depends on, the state of the containing program, it is safe to evaluate it in parallel with any other computation; in our terminology, such entities are said to be white (or completely pure). Note that in Erlang, very few code fragments are completely free of side-effects.

In many cases, side-effects are coming from the dependence on some sort of global variables. As the description suggests, this is not an effect in fact, only a dependence on the context. Expressions and functions with such dependencies are said to be gray, since they are not pure, but in our concept, they are not impure either. Entities actually modifying the state are said to be black.

Propagation of side-effect properties changes a bit according to the level of purity. Namely, if an expression refers to a function that is gray, the expression becomes gray, while references to black functions make expressions black. Functions containing at least one gray expression are gray, while a single black expression turns them black.

According to these levels of impurity, components are sane if and only if they only contain white and gray expressions, or the effect of any contained black expression is only observable within the component.

2.5.3 Forms of impurity in Erlang

Standard and file IO As for the standard IO, we cannot reason about rearrangeability of events; these operations are clearly impure, such expressions are black.

Only the programmer can decide and guide the tool to allow, for example, reordering some log messages. Similar ideas and consequences can be said about file IO operations.

Shared state What about shared variables? Erlang does not allow programs to have shared, global variables. However, Erlang has support for natively implemented functions, which are arbitrary C or C++ subprograms; thus, in the end, we might have some shared data structures implemented in C that we need to deal with. Also, if processes encapsulate a shared state in their call stack (which very often happens in Erlang programs), that can be seen as shared data modifiable by calls to the process, which we need to take into account.

For instance, in the case of ETS (Erlang Term Storage [2]) we manipulate hash-tables implemented in C. What we need to notice here is that ETS read operations do not modify the outside world at all. Therefore, provided that all write operations are finished, the order of read operations is freely changeable. This is why we consider ETS read operations to be gray, and the write operations to be black. Similar conclusions could be drawn for DETS tables (Disk based ETS [3]) and Mnesia tables (distributed DBMS of telecommunications applications [10]).

2.5.4 Side-effects of functions in the Open Telecom Platform (OTP)

Most Erlang applications intensively use library functions defined in the Open Telecom Platform, which includes the Erlang Standard Library. To decide whether a library-using function is side-effected or not, we need to know whether the library functions have side-effects. Analysing huge libraries in each and every case the information is needed is infeasible; therefore, we execute the analysis only once, when a new version of the library is released. Then, during the analysis of user code, we rely on our pre-built index of side-effects of OTP functions.

2.5.5 Side-effects of natively implemented functions (NIFs)

Erlang allows native implementation of functions in C or C++. NIF libraries are dynamically linked to the emulator process in run-time, providing the fastest way to call C code from Erlang. Programmers can supply both the Erlang and the C versions of the same function, but they are not forced to do so; therefore, it is typical that the Erlang version of a natively implemented function is just a stub (usually throwing an exception).

In the case of many natively implemented standard library functions we do have both an Erlang and a C implementation, so that we do analyse the body of the Erlang function as we suppose it depicts the semantics of the C/C++ function.

Consequently, if the Erlang code is side-effected, we consider the C/C++ code to be side-effected as well.

Side-effects of built-in functions (BIFs) Unfortunately, some widely used OTP functions are not implemented in Erlang, only in C. We know however the semantics of these functions and we built an index summarising our knowledge about the side-effects of built-in functions.

2.6 Type inference

Erlang is a dynamically typed language; that is, types are inferred and checked only at run-time, by the virtual machine. The language syntax was designed so that it makes it quite complicated to infer function types statically. Nevertheless, both in pattern discovery and in pattern assessment we rely on types of expressions and functions. During the identification of pattern candidates, list expressions play an important role (at the moment the *skel* library mostly contains data-parallel skeletons over lists). Side-condition analysis also relies on type information, especially when checking properties of binary operations. Last but not least, cost estimations and ranking are the most dependent on the process of type inference, since test data generation can only work with accurate information on function argument types. Note that we use benchmarking of computations on randomly generated input data in order to make parallel speedup estimates with some cost model – this approach allows us to prioritize pattern candidates, e.g. drop candidates which do not seem to offer reasonable speedup.

During the history of the Erlang language, many type inference systems have been created. The most successful is *success typing*, implemented in TypEr [6], which is part of the Erlang/OTP function collection. In our first approach to argument generation, we applied TypEr for type inference. However, it turned out that success typing infers too general types in most of the cases, and these types are not practically usable for our particular purpose.

Success typing, the algorithm implemented in TypEr, addresses the problem of creating a reasonable, user-friendly type system for Erlang. Due to the dynamic nature of the language, it is impossible to specify a type system that provides both precise and human readable types at the same time. Success typing resolves this contradiction by sacrificing accuracy. In contrast to conservative typing, a success type is good for finding bugs, but will not ensure the "well-typed programs cannot go wrong" principle. If a function is called with an argument that is not covered by the success type of the function, the function will *surely fail* with an exception. But the function *might also fail* if the argument is a value of the success type. As a consequence, types inferred by TypEr are readable, but not accurate enough to

```
decode(Label) ->
  case Label of
    function -> 0;
    module   -> 1;
    variable -> 2;
    expr     -> 3;
    value    -> 4;
  end.
```

Figure 2.5: Example function for type inference

generate test data as input for the cost measurement: we'd better generate test data that is guaranteed to not break the code. Note that we need not generate random input of all possible types for an argument of a computation, when we benchmark the computation, but we expect the generated input values to be safe to pass to the benchmarked computation.

Let us consider the example in Figure 2.5. The type inferred by TypEr for this function is $(atom()) \rightarrow byte()$. Although this type is sufficient for documenting the intention of the function and it is correct according to success typing, when used for input generation, the generated input probably contains atoms that are not present in the alternatives of the `case` expression. Test values generated based on such a general type result in run-time exceptions that hinder benchmarking.

We identified the main weakness of success typing as the mechanism of substituting an upper bound type every time a type expression becomes too complex. Although the complexity threshold could be fine tuned (which could fix some issues), still, in plenty of cases, we would get over-generalised types.

As a solution to this problem, we introduced a refined version of success typing. In our approach, no type expressions are considered to be too complex. This design decision eliminates the problems like the one related to Figure 2.5. As for the implementation, it would be practical to modify TypEr accordingly, but it is not straightforward: type substitution/simplification is a key criteria in the termination of its inference algorithm. Therefore, we decided to implement our own type inference, based on the Core Erlang equivalent of the code to be typed.

The compiler can compile the Erlang source code to *Core Erlang*, which is a functional sub-language of Erlang. Our type inference algorithm is implemented on this sub-language, which hides the complexity of Erlang while it is still semantically equivalent to it. This enables us to create an algorithm that can produce more precise types for input generation.

We introduced a syntax-based type inferring algorithm, which parses the Core Erlang syntax tree, and calculates types for the function parameters and return types. It keeps a dictionary for the variables of the function and associates the current type of the variable to it. If the analysis reaches a function call, it fetches the type

```
rec(0)             -> [];
rec(N) when N > 0 -> [rec(N-1)].
```

Figure 2.6: Example function for recursive type

of the function from a global storage. If the type is not in the global storage, it will fetch the $(any()) \rightarrow any()$ type, and mark its type as *incomplete*. To eliminate *incomplete* function types we re-analyze the functions for a predefined number of times.

This type system will not produce as compact types as success typing, but the calculated types will be accurate enough for the test data generator. To eliminate as many *incomplete* types as possible, we applied an optimization: in the first run of the analysis we build a function call graph, and run the following iterations according to the reverse order of this graph.

There are three key areas that should be addressed by this new type inference system:

1. recursion and recursive types,
2. built-in functions (BIFs) and
3. opaque types[2].

Our algorithm solves the problem of recursion by using a finite algorithm. Instead of calculating the most precise type for the function, we calculate only a subtype of its *success type*. For instance, for the function in Fig. 2.5, we calculate the union type of the five atoms. The finite nature of the algorithm will produce a finite subtype of a recursive type. E.g. the function `rec` in Fig. 2.6 will produce an N-deep list of empty lists. Success typing generates the over-general $list()$ type, i.e. "list of any". Our type system will produce a finite m-deep list of empty lists as a type. Here m represents the number of iterations performed by the algorithm.

Our type system can only type functions for which the source is available. This will not hold for built-in functions. Our implementation provides types for all the BIFs, deriving from the implementation of the HiPE [5] library from Erlang/OTP.

There are two solutions for the opaque type problem. One is to go beyond the opaque type, and make the type transparent. In order to do this, we have to analyze all the sources of the library providing the opaque type. In contrary, by leaving the opaque type opaque, there is no need for extra analysis of external libraries. But then the test data generator must be able to use the constructor of the opaque type. We have chosen the latter solution.

[2] In Erlang there are no user defined types, but there is a possibility to indicate types with annotations. If a type is annotated as opaque, then the type representation is hidden from the user and the type system.

Although we have lost the possibility to use a standard library tool to calculate types for functions, we gained more precise type information with our approach, and this can improve reliability and accuracy of benchmarking, and hence the heuristics for pattern candidate prioritization.

Chapter 3

Conclusion

We reported on different analyses that are required for pattern discovery and assessment in Erlang programs, and hence enable parallelization refactoring. Some of the analyses build upon the results of others, and they may serve different purposes, including the selection of transformable structures, the verification of side conditions and the prioritization of pattern candidates. We proposed, and briefly summarized the operation of, the following analyses:

- scope analyses for modules, functions and variables,
- static and dynamic function call analyses,
- data-flow analysis,
- side-effect analysis and
- type analysis.

Some of the analyses considered to be essential had already been present in the static analysis framework of RefactorErl (like function and variable scope analysis or data-flow analysis), some ideas were completely new (like four-valued side-effect analysis), while in a couple of cases extensions have been made to refine or improve the current framework.

The results of this work will be used to design and implement the final version of pattern discovery (to be delivered in D2.13), and the transformation advising feature of the refactoring user interface (T4.1). The analyses delivered here are implemented with the RefactorErl static program analysis and transformation system, which is integrated [15] with Wrangler, providing the Erlang refactoring tool for ParaPhrase.

Bibliography

[1] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. Core Erlang 1.0 language specification. Technical Report 2000-030, Department of Information Technology, Uppsala University, November 2000.

[2] Ericsson AB. Built-in term storage. http://erlang.org/doc/man/ets.html. Erlang STDLIB Reference Manual.

[3] Ericsson AB. A disk based term storage. http://erlang.org/doc/man/dets.html. Erlang STDLIB Reference Manual.

[4] Dániel Horpácsi and Judit Kőszegi. Static analysis of function calls in Erlang. *e-Informatica Software Engineering Journal*, 7:65–76, 2013.

[5] Erik Johansson, Mikael Pettersson, and Konstantinos Sagonas. A high performance Erlang system. In *Proceedings of the 2Nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '00, pages 32–43, New York, NY, USA, 2000. ACM.

[6] Tobias Lindahl and Konstantinos Sagonas. Typer: A type annotator of Erlang code. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang*, ERLANG '05, pages 17–25, New York, NY, USA, 2005. ACM.

[7] University of St Andrews. A Streaming Process-based Skeleton Library for Erlang. https://github.com/ParaPhrase/skel, December 2013.

[8] Mihalis Pitidis and Konstantinos Sagonas. Purity in Erlang. In *Proceedings of the 22Nd International Conference on Implementation and Application of Functional Languages*, IFL'10, pages 137–152, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] Melinda Tóth and István Bozó. Static Analysis of Complex Software Systems Implemented in Erlang. In *Central European Functional Programming Summer School - Fourth Summer School, CEFP 2011, Revisited Selected Lectures*, volume 7241 of *LNCS*, pages 451–514. Springer-Verlag, 2012.

[10] C. Wikström and H. Nilsson. Mnesia — an industrial database with transactions, distribution and a logical query language. In *Proc. Intl. Symp. on Cooperative Database Systems for Advanced Applications*, 1996.

[11] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Comm ACM*, 20(11):822, 1977.

[12] WP2. Initial Pattern Discovery, D2.11. Technical report, ES, December 2013.

[13] WP2. Pattern Amenability, D2.10. Technical report, ELTE, December 2013.

[14] WP3. Software/Hardware Virtualisation Interfaces, D3.1. Technical report, RGU, June 2012.

[15] WP4. Refactoring User Interfaces, D4.3. Technical report, USTAN, September 2013.

[16] Yecc: LALR-1 Parser Generator. `http://www.erlang.org/doc/man/yecc.html`, 2014.