Project no. 288570

PARAPHRASE

Strategic Research Partnership (STREP)
Parallel Patterns for Adaptive Heterogeneous Multicore Systems

Initial Pattern Discovery

D2.11

Due date of deliverable: 31st December 2013

Start date of project: October 1st, 2011

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

Responsible institution: ES
Editor and editor’s address: Melinda Tóth, István Bozó, ES
Executive Summary

In this deliverable we describe the initial pattern discovery analysis. In particular, we show the calls to specific library functions and the language constructs where a skeleton can be introduced. We also describe how we identify these pattern candidates. In addition to this, we show the prototype identification of map-like and pipeline-like recursive functions.
Positioning of Deliverable D2.11

Figure 1: Positioning of Deliverable D2.11: w.r.t. other WP2 deliverables (top) and w.r.t. the forthcoming deliverables from other WPs (bottom).
Chapter 1

Introduction

In this deliverable we describe the initial pattern discovery analysis, that aims to find sources of parallelism in Erlang programs, and displays the appropriate skeletons to be introduced.

Chapter 2 shows specific language constructs (generators and library calls) to be identified as pattern candidates. We have examined several library modules and we have summarised the found candidates in this chapter. In Section 2.3 we describe the technical details and the used tools.

In Chapter 3 we show the prototype identification of map-like and pipeline-like recursive functions.
Chapter 2

Initial Pattern Discovery Analysis

In D2.2 [4] several skeletons have been defined for parallel evaluation of different iterations on lists and streams. In this chapter we introduce the basic language constructs and libraries that can be transformed to use the skel library. The transformations can be performed manually by the programmer, but it can be applied in a (semi-)automatic way with the help of a pattern discovery algorithm.

For this we have defined static analysis methods to discover the candidates for the transformations.

2.1 Identifying Language Constructs

2.1.1 List Comprehensions

The list comprehension is the most commonly used language construct in Erlang to apply some function on the elements or filter some elements of a list, or even both.

\[
[\text{OutputExpr} \mid\mid \text{IteratorsAndPredicates}]
\]

The \text{IteratorsAndPredicates} part contains iterators on input lists and predicates for filtering the proper elements corresponding to predicates.

The \text{OutputExpr} can be:

- an expression that is applied on elements of the input list(s), that satisfy the filtering predicates
- an expression/constant expression that disregards the variables bound in the \text{IteratorsAndPredicates} part

In the first stage of the pattern discovery we identify list comprehension in the source code. The list comprehension can be categorised as map or pipeline, depending on the \text{OutputExpr}. The next step is the categorisation.

A list comprehension is categorised
• as map candidate when the OutputExpr is a function application

• as pipeline candidate when the OutputExpr is a composition of functions (implicit or explicit)

The condition for pipeline candidates is stricter than the condition for map candidates, thus every list comprehension that is included in the set of pipeline candidates is also included in the set of map candidates. Later the candidates will be submitted to further analysis that determines which is more appropriate.

We identify two types of composition:

• explicit – a function that is applied in the argument list of other function.

• implicit – a function that is applied as a continuation of another function, that can be a function that is applied as a last expression in the body of another function.

2.2 Identifying Library Calls

2.2.1 Module lists

The module lists defines many functions operating on lists. There are some functions that can be transformed easily to operate the same way as the original implementation, but perform the operation in parallel on the elements, or chunks of elements.

In the discovery analysis we are looking for applications of functions from the library lists. For selecting the appropriate functions we use built in knowledge that decides which function is relevant for further analysis. These functions are:

The following function applications are included in the set of map candidates:

• lists:map/2

• lists:filter/2

• lists:foreach/2

• lists:zipwith/3

• lists:zipwith3/4

• lists:flatmap/2

The following function applications are included in the set of reduce candidates:

• lists:foldl/3

• lists:foldr/3
2.2.2 Other Library Modules

We have studied the data related library modules of Erlang/OTP and examined the calls of the library functions where skeletons can be introduced.

We have grouped the modules based on the similarities in representation or functionality:

**Data represented by lists.** These modules (proplists, ordsets, ordict) have some special constraints on the manipulated data (e.g. type of the elements, ordering). Their data are represented by lists. Therefore some of the functions defined in these modules can be transformed to use a parallel skeleton (e.g. map, reduce).

- **proplists** module— supports functions for property lists. The functions get*, lookup* and substitute* can be represented as filers and maps on a list, thus they can be transformed to run in parallel. We have to note here that this operation is sometimes to simple to run in parallel, thus an accurate cost model could help to filter out the unwanted possibilities.

- **ordsets** module – supports functions for manipulating sets as ordered lists. The functions fold and filter can be transformed similarly to the functions in module lists. The functions update* can be represented as maps on a list, thus they can be transformed to run in parallel as well.

- **orddict** – supports functions for manipulating key-value dictionary as ordered lists. The functions fold and map can be transformed similarly to the functions in module lists.

**Data to be converted to lists.** The representation of the data of these modules (array, sets, dict) are hidden from the user. They have some functions which iterate on the represented data, thus they can be run in parallel. Since the representation is not defined we have to transform the data to an appropriate representation at first. The general algorithm to handle these functions is:

- transform the data to lists
- apply the skeleton
• transform back the result to a the original representation

The following functions can be transformed:

• array module – supports functions to manipulate functional, extendible arrays. The functions map, foldr, foldl, sparse_map, sparse_foldr and sparse_foldl can be transformed similarly to the functions in module lists. The data converter functions are to_list and from_list.

• sets module – supports functions to manipulate sets. The functions fold and filter can be transformed similarly to the functions in module lists. The data converter functions are to_list and from_list.

• dict module – supports functions to manipulate key-value dictionaries. The functions fold, map and update* can be transformed similarly to the functions in module lists. The data converter functions are to_list and from_list.

Data to be converted to lists 2. The data in these modules (gb_trees and gb_sets) are represented with nested tuples. Therefore it is possible to introduce data specific parallelism with the extensions of skeletons. However the previously introduced method can be also applied. We can transform the tuples to lists and apply the already existing skeletons.

The following functions can be transformed:

• gb_trees module – supports functions to manipulate balanced trees. The function map can be transformed similarly to the functions in module lists. The data converter functions are to_list and from_ordict.

• gb_sets module – supports functions to manipulate ordered sets. The function fold can be transformed similarly to the functions in module lists. The data converter functions are to_list and from_ordset.

Data manipulation with side-effect. The modules ets, dets and mnesia support data storage and manipulations. Calling functions from these modules have side-effect, thus parallelisation is not straightforward. Although identifying the calls from these modules is similar to the discovery of other calls, an appropriate side-condition analysis is required to check the applicability of skeletons for these cases (It will be discussed in Delivery 2.12).

The following functions can be transformed:

• ets = foldr, foldl, match*, select*, update, lookup*

• dets = foldr, foldl, match*, select*, lookup

• mnesia = foldr, foldl, match*, select*, transform_table
Manipulates the data mainly as a single unit. The following modules manipulates the data as a single unit thus we can not introduce parallelism when calling the functions defined in these modules.

- string
- binary
- queue
- digraph

2.3 Technical Details

During the initial pattern discovery analysis we try to find some expressions (or sequence of expressions) or other language constructs which fit to some syntactic and semantic constraints. To find these language elements an appropriate intermediate source code representation is required. During the analysis we use this intermediate representation to identify the language constructs and to check the side conditions as well, therefore it must contain syntactic and semantic information about the source code.

2.3.1 Program Representation in RefactorErl

RefactorErl [1] is source code analysis and transformation tool for Erlang. In order to use the features of the tool, the source code has to be analysed at first. This initial analysis builds the appropriate representation of the source code and stores the calculated information.

RefactorErl represents the source code in a three layered graph. We call it Semantic Program Graph (SPG [2]). The graph contains lexical, syntactic and semantic information about the source code. All of the advanced analysis can be built upon this representation and the querying interface of the graph. During the initial analysis the tool builds the SPG from the Erlang source files and stores it in a database. The “Pattern Discovery Analysis” assumes that we have already loaded our source files to the database of RefactorErl and we are able to query the SPG.

2.3.1.1 Semantic Program Graph

The SPG is created by the lexical, the syntactic and the semantic analyser modules. At first the lexical analyser scans the source file and creates the token sequence from it. After that the syntactic analyser builds the abstract syntax tree (AST) from the tokens. The syntactic and lexical analysis are based on an extensible language description, and both the lexer and the parser are generated based on this description.
RefactorErl has an incremental asynchronous semantic analyser framework to manage the semantic analyser modules. The framework automatically calls a set of semantic analyser modules for each form (functions, attributes, records, etc.) in the source in a separate process. Each analyser process uses its local cache to store the calculated information and a global synchronisation process is used to manage the concurrent operations. The basic set of analyser modules are:

- context analyser
- variable analyser
- module analyser
- function analyser
- record analyser
- data-flow analyser
- layout analyser
- metric analyser

The initial analysis may take long for industrial scale application, but we have to perform it only once, before the first use of the tool. The framework makes possible to incrementally update the graph based on the modified source code parts. It usually takes only a few seconds.

Some of the static analyses do not fit into the operation of the incremental analyser framework, because they use global information that must be presented in the graph. Therefore we have implemented a set of “post-analysers”, like the dynamic function call analyser. It analyses the dynamic calls (“apply”-s and “MFA”-s) and adds the detected references to the graph.

The currently implemented “post analysers” are using the “data-flow reaching”. The data-flow graph [3] can be built incrementally, because it contains only the direct data-flow between expressions. The data-flow reaching traverses the whole data-flow graph, thus it can not be calculated incrementally. It returns the direct and indirect data-flow between expressions: it lists the points where a certain value can flow or the points where certain values can be originated.

During the pattern discovery we are using both the syntactic and semantic information presented in the SPG and the information calculated by the data-flow reaching.

### 2.3.2 Implementation Details

The pattern discovery analysis has been implemented in two phases. The first phase contains a set of search functions. These functions are responsible to find the points where a single skeleton can be applied. The nodes, representing these program
Figure 2.1: Finding candidates

points, are the so-called candidates. The different types of skeleton searchers are implemented separately (e.g. to identify maps in list comprehensions or pipelines in list comprehension, to identify maps in library calls) and can be easily extended with more skeleton identifiers. The result of this phase can be reached through interface functions: find/0, find_all/0, find_candidates/0 etc (Figure 2.1).

The second phase of this analysis creates chains of candidate from the candidate list. We calculate the chains by traversing the SPG based on the control flow. A chain represent a possible sequence of transformations that can be applied on the source code to make it parallel. The result of this phase can be reached through the interface function get_candidate_chain_combinations/0 (Figure 2.2).

An additional phase will be implemented in the final pattern discovery component. The last part of this analysis will be the application of the skeleton rewriting rules in an Erlang specific way.

The result of this analysis is the input of the Pattern Candidate Browser (introduced in Delivery 4.3 [6]) which provides a user friendly web interface to show the result (Figure 2.3).
Figure 2.2: Chains of candidates

![Chain of candidates](image1)

Figure 2.3: Pattern Candidate Browser

![Pattern Candidate Browser](image2)
Chapter 3

Advanced Pattern Discovery Analysis

Beside the identification of used library calls we started the work on the identification of those recursive functions which implement the same behaviour as the library calls, thus after the program shaping a skeleton can be applied to introduce parallelism.

3.1 Identifying Recursive Functions

We can identify two kinds of recursive structures as pattern candidates: map-like and pipeline-like functions. Both of the analyses are based on a syntax derived structure identification enriched with semantic information (e.g. data-flow information).

3.1.1 Map-like Functions

The map-like functions are recursive functions that iterate on the elements of a list, perform some modifications and return the modified elements.

A function is identified as map pattern candidate if fulfils the following requirements:

- it has a formal parameter which type is list
- its return value is a list
- for each recursive branch (recursive clause of the function or a clause of a branching expression containing the recursion) the return value is a list that consists of an application of a function and a recursive function call
  - the application is either a lambda function or a named function and the function is applied on the element (or a transformed variation) of the list parameter of the recursive function (mainly on the head of the list)
the function is applied on the subset of the list parameter of the recursive function (mainly on the tail of the list)

During the analysis we use data-flow reaching [3] to determine values of certain expressions and to deduce their type. We use it both forward and backward direction. The backward data-flow reaching returns those expressions from the source code which value can flow into a certain expression, it means that their values are equal. The forward data-flow reaching calculates every expressions where a certain value can flow.

For example we use data-flow reaching to calculate the return value of the function. The last expression of the function body does not always describe the return value properly. For example it can be a variable, and we bind a value to the variable that is calculated in another part of the program.

### 3.1.2 Pipeline-like Functions

The pipeline-like functions are recursive functions that iterate on the elements of a list, perform two or more modifications sequentially and return the modified elements.

A function is identified as pipeline pattern candidate if fulfills the following requirements:

- it has a formal parameter which type is list
- its return value is a list
- for each recursive branch (recursive clause of the function or a clause of a branching expression containing the recursion) the return value is a list that consists of an application and a recursive function call
  - the application is an explicit or implicit composition (see 2.1.1) and the function is applied on the element (or a transformed variation) of the list parameter of the recursive function (mainly on the head of the list)
  - the recursive call is applied on the subset of the list parameter of the recursive function (mainly on the tail of the list)

During the pattern discovery analysis of pipeline-like functions we also use data-flow reaching described in 3.1.1.

### 3.2 Future Directions

Beside the previously mentioned language constructs we are focusing on the extensions of the presented discovery techniques. We want to make it more precise and filter out false positive hits and extend it to more complex language constructs by defining a behaviour equivalence relations based on the syntax and semantics of Erlang programs.
During the implementation of the pattern discovery we will focus on the identification of patterns described in D2.10 [5].
Chapter 4

Conclusion

Our main goal is to define and implement several new static analyses for Erlang that can be used to find promising parallel pattern candidates in (unpatterned) Erlang programs.

In this delivery a simple analysis was presented that allows target Erlang code fragments to be identified: specific language structures (e.g. list comprehensions) and specific function calls with known properties (e.g. map or fold operations). In addition to this, a prototype advanced pattern discovery analysis was presented to show the identification of recursive functions.

We have described the technical details and introduced the used representation and tool to analyse the source code.

The result of this analysis will be provided to the refactoring user interface of T4.2 as potential pattern candidates using an appropriate calling interface, and corresponding refactoring rules defined in T4.3.
Bibliography


