Project no. 288570

PARAPHRASE

Strategic Research Partnership (STREP)
PARALLEL PATTERNS FOR ADAPTIVE HETEROGENEOUS MULTICORE SYSTEMS

Refactoring Rules
D4.5

Due date of deliverable: 31st July 2014

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

Type: Deliverable
WP number: WP4
Task number: T4.3

Responsible institution: ELTE-Soft
Editor and editor’s address: Zoltán Horváth, ELTE-Soft

Version 0.1 (Revision)

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th>PU</th>
<th>PP</th>
<th>RE</th>
<th>CO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Restricted to other programme participants (including the Commission Services)</td>
<td></td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Restricted to a group specified by the consortium (including the Commission Services)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Confidential, only for members of the consortium (including the Commission Services)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Executive Summary

Work package 4 aims to develop the methodology as well as the tool support for a complete refactoring transformation system, which can rewrite sequential realisations of the ParaPhrase patterns into parallel equivalents semi-automatically (i.e. under programmer’s control). The final code is a result of a complex process composed of pattern discovery, candidate assessment and refactoring transformation, and it should be a semantically equivalent, but structured parallel program. In Task 4.3, ELTE and ELTE-Soft have worked on i) finding the proper abstraction levels and suitable notations for defining refactoring transformation rules formally, and ii) defining the refactoring transformations needed for transforming typical sequential implementations of patterns into instances of the parallel skeletons. A key aspect of this task was to identify and formalise the pre- and post-conditions under which the transformations are valid and yield semantically equivalent code.

We started our work by giving an exhaustive set of code examples as a specification for required shaping transformations (summed up by the [2] technical report). Then we examined these transformations and we defined a formalism with which we were able to provide the formal rules of the transformations (for further detail see the [1] technical report).

At a very early stage, we have found that there can be many different sequential realisations of a pattern. For example, in the case of the farm pattern, so-called map-like computations are realisable with language-level constructs as well as by higher-order functions on lists and other data structures. Therefore, for each and every pattern, we defined a canonical sequential implementation to which we reshape any other variants, and then skeleton introduction is defined on the canonical form only. This idea induces the concept of shaping transformations. Shaping transformations aim at transforming pattern-like computations into one of the canonical forms. As a result, in our approach, the paraphrasing process is embodied as a composition of a number of shaping transformations, a skeleton introduction step, and some clean-up transformations. This decomposition of the paraphrasing transformation simplifies the implementation, and may facilitate formal verification.

Positioning of Deliverable D4.5

The positioning of this deliverable with respect to other deliverables can be seen in Figure 1.
Figure 1: Positioning of deliverable D4.5
# Contents

Executive Summary .............................................. 1

1 Introduction ........................................... 4

2 Taxonomy of ParaPhrase Refactoring Transformations .............. 5
  2.1 Overview ........................................... 5
     2.1.1 Pattern candidate versus refactoring candidate ....... 6
  2.2 Classification of shaping transformations .................... 6
     2.2.1 Source ........................................... 7
     2.2.2 Target ........................................... 7
     2.2.3 Canonical forms ................................... 7
  2.3 Categories in detail ...................................... 8
     2.3.1 Task farm — map-like computations .................... 8
     2.3.2 Pipeline — pipe-like computations ..................... 9

3 Methodology of Formalising Refactoring Transformations ........ 11
  3.1 Informal specification – an example ......................... 12
  3.2 Formal notation for transformation rules .................... 13
  3.3 Formal rules – an example ................................ 14

4 Transformation rules in action ................................ 16
  4.1 Shaping ............................................. 16
  4.2 Skeleton introduction ................................... 17
  4.3 Cleanup ............................................. 18
Chapter 1

Introduction

We aim to develop the methodology as well as the tool support for a complete refactoring transformation system, which can rewrite sequential realisations of the Paraphrase patterns into parallel equivalents semi-automatically (i.e. under programmer’s control). Our approach combines pattern discovery, candidate assessment and refactoring transformation. In Task 4.3, we have worked out, and formalized, refactoring transformations. We have worked on i) finding the proper abstraction levels and suitable notations for defining refactoring transformation rules formally, and ii) defining the refactoring transformations needed for transforming typical sequential implementations of patterns into instances of the parallel skeletons. A key aspect of this task was to identify and formalise the pre- and post-conditions under which the transformations are valid and yield semantically equivalent code.

By paraphrasing, we mean the step-wise transformation of a sequential program into a structured parallel equivalent. This transformation has to be a refactoring: it reshapes the code, changes the appearance and some non-functional properties, but does not alter the meaning (the semantics) of the program. Paraphrasing (and refactoring in general) can be quite challenging in Erlang: candidates can be of various syntactic forms, and also, behaviour-preservation is not obviously guaranteed due to the complex semantics of the language.

In order to ease the specification and implementation of the paraphrasing process, we decompose the complex transformations into, and define them in terms of, smaller, more obvious, reusable and verifiable steps. We have defined a number of such basic refactoring transformations, which have been identified as fundamental steps to paraphrasing a sequential Erlang program. Note that all these transformations are refactorings themselves, as well as their compositions.

In this document, we summarize a complete system of transformations. In order to better oversee the system, we show how to classify these transformations; in fact, we provide different dimensions of classification in Chapter 2. The notations used to formalize the rules are introduced in Section 3. An example that illustrates the paraphrasing process and the use of the formal rules is provided in Section 4.
Chapter 2

Taxonomy of ParaPhrase Refactoring Transformations

In this chapter the different kinds of refactoring transformations that are needed for the ParaPhrase Refactoring Tool for Erlang (PaRTE) are described. We show how to classify these transformations – the classification allows us to understand the structure in the system of transformations. Subsequent chapters explain how the rules for the transformations can be formalized. The transformations presented here may be applied one after the other in order to turn a sequential piece of code into a nice and readable parallel equivalent.

2.1 Overview

Transformations fall into one of the following three main categories:

Shaping transformations. Sequential code fragments may need to be reshaped (rephrased) to a form that is easily and straightforwardly rewritable into a parallel skeleton (we will call this form the canonical form). Shaping transformations are responsible for simplifying/normalizing the code until it contains canonical forms. In section 2.2 we further classify the different shaping transformations.

Skeleton introductions. A canonical form can be directly rewritten into an instance of a parallel skeleton. These skeletons are implemented in the skel library [4]. Usually, the canonical form determines which skeleton is to be introduced, but as new domain-specific patterns may be defined, it might happen that multiple skeletons can be applied on a single canonical form. Skeleton introduction replaces a piece of sequential Erlang code with a call to the skel library.

Cleanup transformations. A shaping transformation is general enough to be applicable on many concrete occurrences of a given syntactic form. Due to
this genericity, the transformation may introduce auxiliary variables, functions or conditions that are not necessary in every possible concrete situation. Cleanup transformations eliminate the syntactic noise: they get rid of unnecessary entities and structures in order to make the code simpler and nicer.

### 2.1.1 Pattern candidate versus refactoring candidate

If a sequential piece of code embodies a computation that can be expressed with any of the structured parallel patterns, it is said to be a *pattern candidate*. This means that it can be rewritten into a parallel skeleton (or a combination thereof) without substantially changing the semantics of the program. Note, however, that not all the pattern candidates are *refactoring candidates*, and vice versa.

- Not all occurrences of a pattern candidate can be transformed with a refactoring tool: the precondition of the refactoring transformation may be strict, and disallow the application of the transformation in a situation where a manual refactoring is still possible.

- The shaping and cleanup transformations described in this document are not only applicable on pattern candidates, and with the purpose of introducing parallel patterns. They may be used for various other purposes as well.

Deliverable 2.13 will give a detailed description of the conditions for a piece of code to be a proper pattern candidate. In this deliverable, we provide pre- and postconditions for the refactoring transformations. The pattern candidate conditions may overlap with, but need not be equal to, the refactoring pre-conditions.

### 2.2 Classification of shaping transformations

Within the system of the ParaPhrase refactoring transformations, shaping transformations can be further classified. There are two dimensions in which shaping transformations can be positioned, as shown in the following table.

**Source:** what language construct the shaping aims to transform towards the canonical form (horizontal axis);

**Target:** which parallel pattern is to be introduced after applying the given shaping transformation (vertical axis).

<table>
<thead>
<tr>
<th>Language constructs</th>
<th>Library calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Farm</td>
<td>✓</td>
</tr>
<tr>
<td>Pipeline</td>
<td>✓</td>
</tr>
<tr>
<td>Reduce</td>
<td>–</td>
</tr>
</tbody>
</table>
2.2.1 Source

There are many ways to express a given (sequential) computation – and hence, there are many syntactic forms that can be transformed into a given parallel pattern. Just to mention an example, a sequential computation that can be transformed into a task farm can be expressed with a list comprehension, with a call to a recursive function, as well as with stream processing. Moreover, such computations are defined as standard library functions, which makes calls to these library functions proper pattern candidates. On the other hand, skeleton introduction, by concept, is only defined on a single syntactic form: a canonical form that belongs to the pattern. Thus, for any other syntactic forms, we do not draw skeleton introduction rules, but we design shaping transformations that rewrite these other forms into the canonical form. As a consequence, for a given sequential implementation of a pattern we either have a shaping rule, or a skeleton introduction rule. The columns of the table correspond to the various syntactic forms which can be the source of the shaping transformations: this classification basically determines the pre-condition of the refactoring.

2.2.2 Target

The rows of the table specify the pattern to be introduced after applying a given shaping transformation. Basically, this classification determines the post-condition of the refactoring. Note that it might happen that one shaping belongs to two or more target patterns at the same time, i.e. when there are some transformation steps that happen to be common in the introduction of different parallel patterns.

2.2.3 Canonical forms

For each pattern, we define a canonical form. Canonical forms define a simple and unique sequential implementation for the parallel patterns. In most cases, canonical forms are not fully semantically equivalent to the parallel implementation, but they can be, under some further conditions.

In principle, one could find and select many code shapes that are simple sequential implementations of farm-like or pipeline-like computations. However, in our approach we decided to select only one, the abstraction level of which is very close to that of the parallel pattern (that is, typically they have the same parametrisation), which makes it straightforward to rewrite it into the skeleton. Thanks to this approach, skeleton introductions are straightforward, whilst the more complex rewriting is achieved by the shaping transformations.

Note that we do not have an obvious canonical form for every pattern. For instance, in the case of reduce and divide&conquer, we have to create our own “library functions” that mimic the semantics of the patterns.
2.3 Categories in detail

Let us investigate some categories of shaping transformations in more detail. In this description, we consider the task farm and the pipeline patterns, we show examples of their canonical forms, and we also present some non-canonical shapes so that we can forecast the necessary shaping transformations. Similar descriptions for the rest of the patterns/categories can be found in our technical report [1].

2.3.1 Task farm — map-like computations

Computations that process a data structure element-wise are amenable to the introduction of the task farm skeleton. Such computations are often expressed with the well-known map function over lists, which applies a given function on each list element. Therefore, we will refer to such computations as “map-like” computations. In Erlang, such computations can be expressed in many different ways.

Canonical form

We decided to use the application of the standard library function lists:map/2 as the canonical form of the task farm skeleton, because its dynamic semantics is very similar to that of the task farm skeleton itself.

\[
\text{lists:map}(F, L)
\]

The difference between the application of lists:map/2 and the task farm skeleton is that the latter does not preserve the order of the elements (and the computations on list elements happen in arbitrary order), while the former preserves the order. Note that if the computation to be parallelized is pure, we can derive some equivalence properties.

\[
sort(\text{FARM}(F, L)) \equiv sort(\text{map}(F, L)) \text{ if pure}(F)
\]

and

\[
\text{ORD}(\text{FARM}(F, L)) \equiv \text{map}(F, L) \text{ if pure}(F)
\]

The first property tells us that sorting the output list makes the result of the farm and that of map the same. The second property states that the combination of the farm pattern and the ord pattern ensures that the order of list elements is preserved.

Other forms

We can provide rules to transform the following map-like computations into the canonical form.
**List comprehension.** A list comprehension produces a list by computing each element according to the “head-expression” of the list comprehension, where this head-expression depends on values drawn from “generators”, and filtered by “filters”. Under some restrictions, list comprehensions can be reshaped into the canonical form. In some cases, element generation and filtering has to be separated (and made sequential), and only the head of the list comprehension can be computed in parallel.

**Recursive functions.** Programmers may define recursive functions that work in a map-like manner. It is not straightforward to decide whether a function is map-like or not. The pre-condition of the relevant shaping rule is quite complex, involving control-flow and data-flow relations to express this map-like property. The shaping of such functions extracts the so-called kernel of the computation, and uses it to rewrite the whole function into a canonical form.

**Recursive processes.** Processes that work on streams may also exhibit a map-like behaviour. The corresponding shaping transformation synthesises a call-back module, which can be consumed by the skel library. Note that in this case, the canonical form is different, abstracting over the operation and the callback module (rather than the operation and the input list).

**Library functions.** Not only lists can be processed element-wise. Many other „data types” in Erlang have map-like operations, applying a function to the elements one by one. The canonical form processes lists, so the shaping transformation has to map the data type onto lists.

### 2.3.2 Pipeline — pipe-like computations

Pipe-like computations apply a sequence of functions on every element of a data structure. In other words, they are like map-like computations, but the function to be applied can be decomposed into a series of functions. Just like in the case of task farms, such computations can be expressed in many ways.

**Canonical form**

The canonical form of pipe-like sequential computations is very similar to that of map-like ones; however, in this case, the function to be applied on the data items is given as a composition of multiple functions: these will be the stages of the pipeline skeleton.

\[
\text{lists:map}(\text{fun}(X) \rightarrow F1(F2(\ldots(FN(X))\ldots))\end{0}, L)
\]

This sequential pipe-like computation is semantically equivalent to the parallel pipeline if every element of the main computation is pure.
\[
PIPE([F_1, F_2, ..., F_N], L) \equiv map(fun(X) \rightarrow F_1(F_2(...(F_N(X))...)) end, L)
\]

if \( \forall i \in [1..N] : pure(F_i) \)

**Other forms**

**List comprehensions, recursive functions as well as library functions** may implement pipe-like behaviour. Element-wise application of a composite function can be expressed with any of these constructs, just like we have seen with task farms. Thus, the shaping transformations to be applied here are the same as those introduced in the case of the farm skeleton. However, in pipelines, the kernel has to be expressible in terms of a special function application composed of consecutive function applications.

**Composite functions** The shaping transformations that particularly belong to the pipeline skeleton are those that can decompose the kernel function of a map-like computation into the future stages of the pipeline. This requires the computation be split up into smaller functions, which call each other in their return expressions (i.e. the first one calls the second, the second calls the third, and so on), while the last component simply returns the value of the whole computation: this will be the output of the pipeline.

General-purpose refactoring transformations (like “extract function”) can assist this process of decomposition. After having the function chain, “tuple function arguments” can be utilised to group the parameters to the functions, and finally, the functions can be rewritten to stages by reshaping the tail calls into their arguments. When having such stage functions, we can compose their consecutive call, which serves as the kernel to the \texttt{lists:map/2} function, giving the canonical form of the pipeline.
Chapter 3

Methodology of Formalising Refactoring Transformations

In this chapter we explain the methodology applied to specify the refactoring transformations that were classified in the previous chapter. The specification of the transformations were developed in two steps: first an informal specification was given, and then formal rules were invented. This chapter describes the notations used in the formal rules as well.

The implementation of the transformations is carried out in T4.1, and will be reported about in D4.4. In this task (T4.3) we have provided the transformation rules in textual format, collected in two technical reports [1, 2]. The current document reports on the achievements that have been made to produce the two technical reports.

The introduced formal notation is based on the Semantic Program Graph [3] (SPG, an abstract model of Erlang programs based on a syntax tree augmented with semantic information) of Erlang programs, as defined by the RefactorErl program analysis and transformation tool. The notation relies on semantic predicates that are already implemented in RefactorErl, or that can be implemented easily over the SPG. A calculus to formally reason about the semantics preserving property of refactoring transformations described with the formal rules still has to be developed, as future research. However, note that the correctness of any implementation of the transformations will depend heavily on the correctness of RefactorErl: the construction of the SPG and the implementation of the semantic predicates.

The notation for formalization presented here has been designed to facilitate implementation as well. Therefore, we strived for a fairly readable notation. In order to achieve that, we employ graphical elements (e.g. font color) and implicit effects (e.g. with respect to begin-end, described later) to make the rules easy to read and understand.

We present our methodology by first giving an excerpt of [2] to illustrate how the informal specification of a transformation was provided. Then, we explain the notation used in the formal rules. Finally, we present the formal rule, as it appears
in [1], that corresponds to the example informal specification.

### 3.1 Informal specification – an example

The canonical form for the introduction of a task farm skeleton is a call to the `lists:map/2` function. This canonical form can be produced easily from a list comprehension with the following syntactic shape.

```erlang
[ some_function(Param) || Param <- List ]
```

If the list comprehension has a single generator, and does not have a filter, and its head applies a unary function to the pattern that was drawn from the generator, we can transform it into the following canonical form.

```erlang
lists:map(fun some_function/1,List)
```

In fact, `Param` and `List` need not even be variables in this example, they could be replaced with any pattern / expression (respectively). If the head of the list comprehension applies a variable referring to a function, or a fun-expression, on a single argument (rather than an atom which is equal to a function name in the current scope), the transformation is just a little bit different.

However, if the list comprehension contains more than one generators, or it contains filters, a more sophisticated transformation is required. (This is also the case when the head of the comprehension is not the application of a unary function – but let us put this problem away for a while.) This more sophisticated transformation will be called “Pre-generate list”, and can be captured by the following informal specification [2].

**Transforming a list comprehension that has an improper generator part**

**Scenario**

Pattern Discovery has found a list comprehension (denoted by `origlc`) to be parallelised with the following characteristics.

- It contains more than one generators, or has at least one filter.
- The head expression of the list comprehension (denoted by `oheadexpr`) uses variables bound by any of the generators.

**Example**

```erlang
f() ->
   [some_function({X,Y}) || X<-Xs, filter(X), Y<-Ys].
```
Required transformation sequence

1. A match expression needs to be introduced and also to be inserted before origlc, in the same scope. The new match expression binds a new variable (denoted by \( n v \)) to a newly introduced list comprehension (denoted by newlc) with the following properties.

   - Its generators produce the same values, in the same order, as written in origlc.
   - The filters of origlc and the filters of newlc are equivalent.
   - The head expression of newlc (denoted by nheadexpr) is a tuple, whose elements are the variables used to evaluate oheadexpr and bound inside the scope of origlg.

\[
\text{f()} \rightarrow \\
\text{NewVar} = \{ \{ X,Y \mid X<-Xs, filter(X), Y<-Ys \}, \\
\{ \text{some_function}(\{ X,Y \}) \mid X<-Xs, filter(X), Y<-Ys \} \}.
\]

2. Both the generators and the filters of origlc are replaced by a generator for which the following statements hold.

   - The source of the generator is \( nv \).
   - The pattern of the generator is a tuple, whose structure is equivalent to nheadexpr.
   - The order of the bound variables must be the same in nheadexpr and in the pattern of the new generator.

\[
\text{Result of applying the transformations}
\]

\[
\text{f()} \rightarrow \\
\text{NewVar} = \{ \{ X,Y \mid X<-Xs, filter(X), Y<-Ys \}, \\
\{ \text{some_function}(\{ X,Y \}) \mid \{ X,Y \} <- \text{NewVar} \}.
\]

3.2 Formal notation for transformation rules

A refactoring transformations is specified in terms of side-conditions (or preconditions) and the syntactic rewriting it performs, where the former guarantee that the rewriting preserves the semantics of the code. Formally, transformations are defined as term rewriting rules. Side-conditions are given by using a set of syntactic and semantic predicates. Let us introduce the following notation:

\[
\frac{A}{B} \quad \text{WHEN} \quad C
\]

Here, \( A \) is called the (syntactic) pattern, and \( B \) is called the replacement. They are both syntactic schemes. Namely, \( A \) and \( B \) may contain metavariables (denoted
with blue font), and also there are special metavariables (e.g. \( E \)) that can match and record multiple syntactic elements in a row (such as expression sequences, parameter lists), embodied as siblings in the syntax tree. In addition, the formalism supports the concept of fresh variables (unused names in the scope, denoted with green font), which can be used to introduce new function and variable names. Finally, a rule can have parameters (denoted with red font), which have to be provided each time the rule is applied.

The notation above means that \( A \) can be rewritten to \( B \) if the condition \( C \) holds. \( A \) as well as \( B \) can be either expressions or functions, and the replacement term is allowed to be composed of multiple terms. If the replacement is given as an expression sequence, it will be enclosed by a begin-end block in order to guarantee syntax-validity (the fact that this effect is implicit, makes the rule more readable). If the replacement is done in the place of a top-level expression, this begin-end block can be eliminated by a cleanup transformation.

The WHEN clause, which is defining the context-dependent side-conditions, is a list of first-order logic formulae, each needed to hold for the rule to be applicable. The formulae typically rely on semantic predicates, context-dependent relations of the program entities mentioned in the syntactic schemes, including scopes, references, control-flow and data-flow. Note that the formulae may also refer to metavariables bound in the replacement term: namely, the synthesised code fragment can be influenced by the condition. For this reason, the WHEN condition may also contribute to the post-condition of the refactoring rule.

### 3.3 Formal rules – an example

Using the notations introduced in the previous section, we can devise a formal rule [1] for the informal specification of Section 3.1. In fact, the rule presented below is even more general than the informal specification, since it covers the case when the head of the list comprehension is not the application of a unary function as well.

\[
\begin{align*}
\text{List} & = [ \{ \text{Vars} \} \mid \text{GFs} ] , \\
\text{Fun} & = \text{fun} ( \{ \text{Vars} \} ) \rightarrow \text{Head} \quad \text{end}, \\
\text{lists} : \text{map} ( \text{Fun}, \text{List} ) \\
\text{WHEN} & \quad \text{Vars} = \text{boundVars} ( \text{GFs} ) \\
& \quad \cap \text{freeVars} ( \text{Head} )
\end{align*}
\]

This is the most general shaping rule for list comprehensions: we can have more generators and filters in \( \text{GFs} \). This is the reason why we collect all of the variables that are bound by the generators and are needed for evaluating the head expression, and use them in the head expression of a new list comprehension, which pre-generates a list with the original generators and filters. This intermediate data structure changes significantly the computation (i.e. it will increase its memory...
consumption), but the result of the original list comprehension and that of the new expression sequence are the same.
Chapter 4

Transformation rules in action

In this chapter we go through a complete parallelization process (shaping, skeleton introduction and cleanup) applied on an example. We explain the needed formal rules, and show how they are used.

Let us consider a small piece of Erlang code implementing sparse matrix-vector multiplication. The matrix $\text{\textit{Rows}}$ is represented as a list of pairs, where a pair $\{I, \text{\textit{Row}}\}$ means that the $I$th row of the matrix is the vector $\text{\textit{Row}}$. Similarly, a vector is a list of pairs, where a pair $\{I, \text{\textit{Val}}\}$ means that the $I$th element of the vector is $\text{\textit{Val}}$. This way it is enough to store the non-zero values in a vector, and the non-zero rows of a matrix.

1. \texttt{vxv(\text{\textit{Row}}, \text{\textit{Col}}) -> ...} $\text{\textit{\$ sparse vector-vector mult.}}$
2. \texttt{mxv(\text{\textit{Rows}}, \text{\textit{Col}}) ->}
3. \texttt{Product = [{I, vxv(\text{\textit{Row}}, \text{\textit{Col}})} || \{I, \text{\textit{Row}}\} <- \text{\textit{Rows}}],}
4. \texttt{lists:filter(fun({\_\_, \text{\textit{V}}}) -> \text{\textit{V}} /= 0 end, Product).}$

In this code snippet we can identify two expressions that could be transformed into an application of the task farm skeleton: the list comprehension and the call to the \texttt{lists:filter/2} function. The filtering operation is quite simple, but the head expression of the list comprehension applies a computation intensive function (the sparse vector-vector multiplication), so it is more profitable to parallelize the list comprehension.

4.1 Shaping

In order to turn the list comprehension that executes vector-vector multiplication on each pair drawn from \text{\textit{Rows}} into an instance of a task farm skeleton, we need to shape it into the appropriate canonical form: a call to \texttt{lists:map/2}. The shaping replaces the list comprehension with an expression sequence.

The following formal rule from Section 3.3 can be applied to the list comprehension expression. Remember that the green and blue symbols in the rule, e.g. $\text{\textit{List}}$ and $\text{\textit{GFs}}$, are fresh program variables and meta-variables, respectively.
\[
\text{List} = \begin{bmatrix} \text{Vars} | \overline{\text{GFs}} \end{bmatrix},
\quad \text{WHEN} \quad \text{Vars} = \text{boundVars}(\overline{\text{GFs}}) \cap \text{freeVars}(\text{Head})
\]

\[\text{Fun} = \text{fun} \left( \{\text{Vars}\} \right) \rightarrow \text{Head} \text{ end},\]

\text{lists : map}(\text{Fun}, \text{List})

The shaped \(\text{mxv}\) function is the following. Assume that the refactoring tool generated the \(L\) and \(F\) names for the fresh program variables denoted by \(\text{List}\) and \(F\) in the rule. Remember that an expression sequence in the target component of a rule implies an implicit begin-end that wraps the expression sequence: this is the case with the above shaping rule as well.

\begin{verbatim}
mxv(Rows, Col) ->
Product =
    begin
        L = [(I, Row) || {I, Row} <- Rows],
        F = fun((I, Row)) -> {I, vxv(Row, Col)} end,
        lists:map(F, L)
    end,
    lists:filter(fun({_, V}) -> V /= 0 end, Product).
\end{verbatim}

\[\text{lists : map}(F, \text{List})\]

\[\text{skel : do}([\{\text{farm}, [\{\text{seq}, F\}], Nw\}], \text{List})\]

\[\text{WHEN pure}(F)\]

4.2 Skeleton introduction

We will introduce the task farm skeleton using the rule below. The canonical form is equivalent\(^1\) to the parallel skeleton only if the computation is pure. Therefore, side-effects should be disallowed by the conditions in order to guarantee behaviour-preservation [5]. Let us now introduce the farm skeleton with 16 workers (\(Nw\) here is the parameter of the rule, as indicated by the red color).

\begin{verbatim}
mxv(Rows, Col) ->
Product =
    begin
        L = [(I, Row) || {I, Row} <- Rows],
        F = fun((I, Row)) -> {I, vxv(Row, Col)} end,
        skel:do([\{farm, [\{seq, F\}], 16\}], L)
    end,
    lists:filter(fun({_, V}) -> V /= 0 end, Product).
\end{verbatim}

\(^1\)modulo the order of the list elements
4.3 Cleanup

The shaping transformation and the skeleton introduction yielded a parallelized computation. However, the code does not look pretty enough, and in fact it is unnecessarily complex. Cleanup transformations can be used to eliminate the syntactic noise caused by the application of previous transformations.

The used shaping rule, for instance, enclosed the introduced expression sequence within a begin-end block, which can be eliminated here quite simply. In general, this cleanup transformation is an iterative process: we enlarge the block until it becomes a top-level expression (or it contains only a single expression), and then remove it.

Two cleanup rules for begin-end elimination are needed for our example. The first rule makes use of the fact that the value of the begin-end block is that of its last expression. If the begin-end block is assigned to a variable, this assignment can be moved into the block. (Note that in Erlang, a begin-end block does not open a nested scope.) The second (elimination) rule is only applicable, when the begin-end block is a top-level expression in a clause, i.e. when it is not a proper part of an enclosing expression sequence.

\[
\begin{align*}
X &= \text{begin } E, F \text{ end} \\
\text{begin } E, X &= F \text{ end}
\end{align*}
\]

\[
\begin{align*}
\text{begin } E \text{ end} &\quad \text{WHEN toplevel(beg}in E \text{ end)} \\
E &\quad \text{WHEN } L \equiv [X || X \leftarrow L]
\end{align*}
\]

After applying the above rules, we arrive at the following code.

```
1 mxv(Rows, Col) ->
2   L = [{I, Row} || {I, Row} <- Rows],
3   F = fun({I, Row}) -> {I, vxv(Row, Col)} end,
4   Product = skel:do([farm, {seq, F}, 16], L),
5   lists:filter(fun(_, V) -> V /= 0 end, Product).
```

In this case, we can also remove the list comprehension composing the list of the elements to be processed. If the original list comprehension only contained a generator, and all of the elements in the list matches the pattern of the generator, we can freely eliminate the variable and the extra list comprehension. This can be achieved by the cleanup rule

\[
[X || X \leftarrow L] \quad \text{WHEN } L \equiv [X || X \leftarrow L]
\]

followed by an Eliminate Variable refactoring on \( L \). These two transformations will put the original list into the \texttt{skel:do/2} call.
This code is nice enough, so we had better stop the transformation process here. Although it is still possible to eliminate variables F and Product, and hence turn \( \text{mxv}/2 \) into a one-liner, this one line would be too complex and deeply nested, and hence rather unreadable.
5. Conclusion

The main goal of work package 4 is to develop methodology and tool to support parallel skeletons introduction by refactorings.

This document closes Task 4.3 (Pattern Candidate Refactoring Rules). Here we described how we had identified the refactoring rules that are needed to transform pattern candidates into concrete parallel implementations of those patterns, including the necessary pre- and post-conditions under which those transformations are valid, and any key effects that are induced as a result of those transformations.

We have classified three main kinds of transformations: shaping, skeleton introduction and cleanup. We have further classified shaping transformations based on the syntactic forms to transform, as well as based on the skeleton to be introduced after shaping.

Shaping transformations allow us to transform many different syntactic forms into canonical forms. Then, skeletons can be introduced by transforming canonical forms into calls to the functions of the skel library. Finally, the code can be beautified using the cleanup transformations, that eliminate unnecessary syntactic structures (such as superfluous begin-end blocks) and entities (e.g. some variables).

We have developed high-level, but precise formalisms to describe transformation rules. A formal rule consists of three components:

- **Syntactic pattern** captures the syntactic form of the transformable code fragment (and, optionally, its syntactic context). It can also bind meta-variables that abstract over syntactic units.
- **Replacement** describes the syntactic form of the code after the transformation. It can refer to the bound meta-variables.
- **Condition** introduces semantic information about the validity of the transformation. It may refer to the binding structure of variables, the call graph, the control flow and data flow graph of the code etc. It may also bind and refer to meta-variables.

The formalization increases our confidence in the correctness of the rules, and could ultimately be used to yield verifiably correct implementations. Note that in [1], we define an extensive collection of refactoring rules, which includes formalisation of more complex cases as well.

This work feeds into work in T4.1, which will implement the corresponding refactoring rules as part of the ParaPhrase refactoring transformation system.
Bibliography


