Free the Conqueror! Refactoring divide-and-conquer functions

Tamás Kozsik, Melinda Tóth, István Bozó

Eötvös Loránd University, Budapest, Hungary

Abstract

Divide-and-conquer algorithms appear in the solution of many computationally intensive problems, and are good candidates for parallelization. A divide-and-conquer computation can be expressed in a programming language in many ways. This paper presents a set of small, semantics-preserving code transformations, and a methodology to refactor divide-and-conquer functions in a functional programming language. By applying a sequence of transformations using a refactoring tool, many divide-and-conquer functions can be restructured into a canonical form – which then can be refactored into an instance of a parallel divide-and-conquer pattern. This methodology offers an effective and safe way to parallelize HPC applications.

Keywords: parallel patterns, Erlang, semantics-preserving, refactoring tool
2010 MSC: 68-N15

1. Introduction

Refactoring is the process of restructuring, shaping, or transforming some program code in order to improve its quality, to change its non-functional properties, or to make it suitable to add a new feature. This activity can be carried out by hand, or by using dedicated source code transformation tools. This paper investigates how a tool can be used to refactor the source code of divide-and-conquer functions written in a functional programming language. The main benefit of such refactoring tools is that they offer semantics preserving source code transformations: the software developer may apply these refactoring transformations without worries about breaking the code.

Divide-and-conquer is a principle that is at the heart of many useful algorithms in different domains, including searching, sorting, FFT, and number theory [1, 2, 3]. By their very nature, these algorithms can be implemented in a parallel way, and be efficiently executed on a parallel machine. Divide-and-conquer is therefore often perceived not only as a generic algorithm design principle, but also as a high-level parallel programming pattern [4, 5, 6].

A divide-and-conquer computation splits a problem into smaller subproblems, and recurses on these subproblems until a base case is reached. The solutions of the subproblems are combined to form the solution of the original problem. This structure can be expressed by the following pseudo-code.

```
solve(Problem) =
   if is_base_case(Problem)
      then solve_base_case(Problem)
      else SubProblems = divide(Problem)
           Solutions = map(solve, SubProblems)
           combine(Solutions)
   end
is_base_case(Problem) = ...
solve_base_case(Problem) = ...
divide(Problem) = ...
combine(Problem) = ...
```

Note how `solve` calls itself recursively on subproblems using the well-known higher-order *map* function.

Obviously, in any programming language one can express this kind of routines in numerous ways – and the above structure may very well be hidden. Our goal is to provide a systematic way to refactor such routines so that the divide-and-conquer structure is let free. In the end the routines can be rewritten as a parametrization of a generic higher-order `dc` operation.

```
solve = dc(is_base_case,solve_base_case,divide,combine)
dc(IsBase,Base,Divide,Combine) =
   let Solve(Problem) =
         if IsBase(Problem)
            then Base(Problem)
            else SubProblems = Divide(Problem)
                 Solutions = map(Solve, SubProblems)
                 Combine(Solutions)
         end
   in Solve
```

Now the software developer may anytime decide to replace the call to this sequential implementation of the divide-and-conquer pattern with a call to a parallel implementation, e.g. one which uses *parmap*, the parallel version of *map*.

```
pardc(ShouldBeSequential,IsBase,Base,Divide,Combine) =
   let Solve(Problem) =
         if ShouldBeSequential(Problem)
            then dc(IsBase,Base,Divide,Combine)(Problem)
            else SubProblems = Divide(Problem)
                 Solutions = parmap(Solve, SubProblems)
                 Combine(Solutions)
         end
   in Solve
```

The rest of the paper is structured as follows. In Section 2 a motivational example is shown. In the paper the examples will be written in the functional programming language Erlang. Section 3 provides background information about the techniques and tools we use. A discussion of related work is given in Section 3.2. Section 6 suggests a canonical form for divide-and-conquer functions. Section 4 presents the refactoring transformations that can be applied in order to change a divide-and-conquer function into the canonical form. Section 8 introduces an informal methodology which tells us how to select, and in which order to apply, the refactoring transformations in order to reach the canonical form. Finally, Section 9 concludes.

The contributions of this paper are the following.

- We have identified a set of semantics preserving refactoring transformations, together with their (sufficient, but not always necessary) side conditions, which can be useful when restructuring a divide-and-conquer function into the canonical form. Some of the proposed transformations already exist and are well-known, others are quite specific to refactoring divide-and-conquer functions, and are – to the best of our knowledge – novel (Section 4).

- We offer an implementation of these refactoring transformations for the Erlang language in our open source tool RefactorErl. RefactorErl is accessible at `https://plc.inf.elte.hu/erlang/`.

- We propose a methodology a software developer may follow to successfully apply these refactoring transformations in order to reach the canonical divide-and-conquer form (Section 8).

These contributions help software developers to safely restructure and parallelize divide-and-conquer functions in order to improve performance of computations. Note, however, that we have not (yet) proved formally the correctness of our proposed transformations. This is still ongoing research, with some promising first results [7].

2. Motivational example

A divide-and-conquer computation can be expressed in many ways in a programming language. We present here an Erlang implementation of *mergesort*, and show how it looks like after refactoring into the canonical form. The **`lists:split/2`** and **`lists:merge/2`** functions are from the standard library. The former splits a list into two at a given position, and the latter merges two sorted lists into one.

```
ms(   [] ) ->  [];
ms(  [H] ) -> [H];
ms(   L  ) -> {L1,L2} = lists:split(length(L) div 2, L),
              lists:merge( ms(L1), ms(L2) ).
```

The semantically equivalent canonical form can be achieved by performing a number of small refactoring transformations. These include the introduction of functions `is_base`, `base`, `divide` and `combine`, and the re-organization of the recursive calls to a call of the standard library function `lists:map/2`. Altogether about a dozen semantics preserving transformations are applied to arrive at the canonical divide-and-conquer form, which we define for Erlang, somewhat arbitrarily, and modulo alpha-conversion, as follows.

```
ms(Lst) ->
   case is_base(Lst) of
      true  ->
         base(Lst);
      false ->
         SubProblems = divide(Lst),
         Solutions = lists:map(fun ms/1, SubProblems),
         combine(Solutions)
   end.
```

The introduced helper functions, after applying some clean-up refactorings, may look the following.

```
is_base( []) -> true;             base( []) -> [];
is_base([_]) -> true;             base([H]) -> [H].
is_base( _ ) -> false.

divide(Lst)  ->
   {L1, L2} = lists:split(length(Lst) div 2, Lst),
   [L1, L2].

combine(Solutions) ->
   [SL1, SL2] = Solutions,
   lists:merge(SL1, SL2).
```

In Section 4 the refactoring transformations applied here, as well as other useful ones, are explained in more detail. The canonical form presented here is already suitable for introducing a call to a sequential or parallel implementation of a higher-order divide-and-conquer function. Considerations for the use of the parallel divide-and-conquer pattern are presented now, in the next section.

3. Background

There may be many advantages to make the divide-and-conquer structure explicit in a function definition, such as improved readability and maintainability. However, we emphasize here its usefulness in the introduction of parallelism in a code base. Earlier papers have already investigated the role of refactoring in general, and of tool supported refactoring in particular, in the process of pattern-based parallelization. Building on the results of the EU FP7 ParaPhrase project [8], one can express parallel computations in terms of parallel patterns, and benefit from their algorithmic skeleton based, compositional implementation, which can exploit heterogeneous hardware, and which is able to dynamically adapt to available hardware resources. According to the ParaPhrase approach, a programmer can use software development tools to find parallel pattern candidates (parallelizable code fragments) in a code base, to prioritize them based on their potential for parallel speedup, and to refactor them into applications of algorithmic skeletons. The composability of the parallel patterns ensure that even large and complex parallel computations can be defined, and easily restructured when the programmer wishes to change the parallel behaviour of the code.

Over the course of the ParaPhrase project, a dedicated refactoring tool for the Erlang programming language was developed. The ParaPhrase Refactoring Tool for Erlang (or PaRTE, for short — pronounced as *party*) is able to automatically identify and prioritize parallel pattern candidates in Erlang code, and provides refactoring transformations for the introduction of *task farm* and *pipeline* patterns into the code [9, 10]. The transformations described in this paper extend the capabilities of PaRTE for dealing with the divide-and-conquer pattern.

The PaRTE tool has been built atop of RefactorErl [11], a static source code analysis and transformation tool, and Wrangler [12], a refactoring tool for Erlang. In the current research we have been using the static analysis and transformation facilities of RefactorErl [13], and we are relying on the existing, already implemented refactorings provided by RefactorErl [14, 15] as well. Some of these existing refactorings (*Introduce Variable*, *Introduce Function*, and *Introduce lists:map/2*) had to be generalized in order to make them more effective in the context of this research.

The programming language of our choice, Erlang, is a functional style actor language, providing excellent support for concurrent and distributed programming, such as processes and message-based communication primitives. Together with its standard library, Erlang/OTP, it facilitates the development of large-scale fault-tolerant systems. The ParaPhrase project opted for C++ (imperative) and Erlang (functional) to demonstrate the feasibility of its methodology based on adaptive, heterogeneous parallel patterns. This methodology, as well as our refactoring support for divide-and-conquer patterns, can be generalized to other languages. However, to keep this paper focused, we stick to a single language, Erlang, in the discourse.

From our point of view, what is interesting is the suitability of Erlang for static source code analysis and transformation. Erlang is functional, but impure, which gives us only a limited capacity of referential transparency. The language is also dynamically typed, which makes it more difficult to develop static analyses for pattern candidate discovery, as well as for checking side conditions of a refactoring. Earlier papers of ours [9, 10] explained how analyses of Erlang code are implemented in RefactorErl. In particular, we have presented static analyses for discovering task farm and pipeline candidates [10].

In a previous paper [16], we investigated the discovery of divide-and-conquer pattern candidates. We described how to find the candidates, without explaining how to refactor them using a refactoring tool. In contrast, the current paper covers the refactoring technique, but does not provide any further information about the discovery process. The former, divide-and-conquer discovery, paper shows a wide range of code samples where the pattern can be identified. The same code samples were used in the present research: we are able to refactor these samples to the canonical divide-and-conquer form using semantics preserving transformations of a refactoring tool.

This paper asserts that a smart software developer is able to apply a number of automated transformations in a sequence in order to arrive at the canonical form. The next step would be to provide a *composite refactoring*, which is able to infer the right sequence of transformations, and apply them all at once. This appears to be an AI search problem, and is left as future work.

After refactoring the Erlang code, and freeing the divide-and-conquer structure, a refactoring transformation to introduce the parallel pattern can be applied. The divide-and-conquer parallel pattern has been defined in the ParaPhrase project as a “high-level pattern”, and implemented for Erlang in the *skel*-library [17, 18]. The example of Section 2 can be rewritten with this library as follows.

```
ms(L) ->
  (sk_hlp:dc_lim( fun is_base/1, fun base/1,
                  fun divide/1,  fun combine/1, 1024)
  )(L).
```

The `dc_lim/5` higher-order function from the `sk_hlp` module takes an integer number as fifth argument, which limits parallelism to (approximately) this amount of used Erlang processes. (Determining the optimal number of processes / threads / cores used for an occurrence of a parallel pattern is itself an interesting research topic [19, 9].)

3.1. Divide-and-Conquer pattern candidate discovery

Divide-and-Conquer pattern candidate discovery [16] can be used to identify functions which recursively call themselves multiple times. One important property of the identified candidate functions is that the recursive calls in them are independent: the parameter of one recursive call does not depend on the result of other recursive calls. This is exactly the reason why such functions can be restructured into the proposed divide-and-conquer canonical form, where recursion is factored out into the *map* higher-order function. Our approach is in accordance with the standard definition of divide-and-conquer functions; for example, Mou and Hudak [20] gave an algebraic model which characterizes a large class of divide-and-conquer algorithms as pseudomorphisms: a “divacon” is a function f that is defined as the function composition

$$c \circ h \circ (\text{map } f) \circ g \circ d$$

on "non-basic inputs" (i.e. on inputs that are intended to be further divided), where d is a divide function, c is a combine function, and g and h are "adjust functions". Note that this simple characterization considers functions like *map* or *fold* as divacons — from our perspective these examples are, in fact, degenerate cases, since we prefer to conceive *map* and *fold* as other patterns. The functions we identify as divide-and-conquer call themselves recursively multiple times *on the same execution path*.

The concept of execution path is important here: recursive definitions may have "base cases", i.e. non-recursive execution paths, and recursive cases (i.e. execution paths with recursive calls). We do not consider here functions without a base case: such stream-processing style functions are not taken as divide-and-conquer candidates. As a consequence, we may assume that there is a branching construct in the definition of the candidate function, which distinguishes between base cases and recursive cases. It might happen that this branching construct is not lexically in the definition of the candidate function, but in an another function called from the candidate. This, and some other trickier examples can be seen at `https://plc.inf.elte.hu/erlang/dnc/` (under the home page of RefactorErl), for example the case when the divide-and-conquer candidate is not directly recursive, but a member in a set of mutually recursive functions.

A generalization of divide-and-conquer algorithms can be expressed with hylomorphisms. A hylomorphism is a composition of an anamorphism ("unfolding" a set of solutions to subproblems) and a catamorphism ("folding" the set of solutions into the solution of the original problem). Pattern candidate discovery could be generalized according to this concept to be able to find even more pattern candidates [21].

3.2. Related work

Tool supported parallelization is a well researched topic. Parallelizing compilers — such as SkelML [22], the skeleton-based parallelizing compiler for ML — automatically identify certain forms of parallel skeletons and transform them to parallel equivalents at compile time. Compared to our approach, a main difference is that the programmer cannot see, or further modify, the resulting code. In our approach, parallelization takes place *before* compilation.

Support for the introduction of parallel skeletons appears in refactoring tools as well. For example, Brown, Janjic, Hammond *et al.* (2014) have proposed a small number of basic refactorings to introduce task farm and pipeline skeletons into C++ using the FastFlow library [23].

For Haskell, Brown, Loidl and Hammond (2011) introduced a limited number of parallelization refactorings in *HaRe* — the Haskell Refactorer. This work introduces parallelism using structural refactorings [24].

Ad hoc approaches for parallelizing Erlang programs can be found in the parallelization of Dialyzer [25] and a suite of Erlang benchmarks [26]. None of these applied structured parallelism or parallel skeletons. In contrast, Barwell *et al.* (2016) used program shaping to introduce parallelism in a multi-agent system based on *task farm*, *pipeline* and *feedback* algorithmic skeletons [27]. This approach is similar to ours, but more focused on the application and does not provide general methodology. We are providing general purpose refactorings that can be used not just in divide-and-conquer parallelization. Another difference is that our transformations are semantic preserving, while theirs are not.

Several refactoring frameworks exists for Erlang. We have chosen the basic infrastructure provided by RefactorErl [11, 15], however, our transformations can be implemented in the domain-specific language described by Horpácsi, Kőszegi and Thompson (2016) [7].

There has been some work on using static analysis to discover parallelism bottlenecks, and providing help to the programmer to reshape the program in accordance with such analyses. Markstrum *et al.* use static analyses to introduce parallelism in X10 programs by refactoring [28, 29], and Wloka, Sridharan and Tip (2009) use static analyses to discover parallelism bottlenecks and provide thread safety and/or reentrancy [30]. Compared to our approach, they neither consider patterns as instances of algorithmic skeletons, nor provide suggestions on which skeletons to apply.

Molitorisz (2013) describes a tool for automatic discovery of parts of a sequential application that implement some rather basic parallel patterns, and proposes supporting refactoring and performance tuning techniques [31]. Molitorisz, Schimmel and Otto (2012) present AutoFutures [32], a tool that performs static analysis on Java programs to discover portions of code showing no data dependencies, and inserts parallel constructs (Futures) directing asynchronously parallel execution.

Work by Dig *et al.* introduces concurrency in Java programs, also by targeting thread safety, aiming to increase throughput and scalability [33, 34, 35]. Dig's refactoring tool contains a minor selection of transformations that are able to rewrite Java code so that it employs generic Java Concurrency libraries to control parallel execution. Mak, Faxén, Janson and Mycroft (2010) report on promising results in automatic shaping and parallelization in C code based on dependence profiling [36].

4. Refactoring transformations

There are many well-known refactoring transformations offered by various refactoring tools, including RefactorErl, which are useful also for our current divide-and-conquer case. These transformations are usually well documented in the literature. In our research, we have used *Rename* (for renaming variables and functions), *Introduce/Eliminate Function* (for extracting a sequence of expressions into a new function definition, and for β-reducing a function call), *Introduce/Eliminate Variable* (for giving a name to an expression, and to inline the definition of a variable where it is used) and *Reorder Function Arguments* (for changing the order of parameters in the formal and actual parameter lists of a function). A somewhat less known refactoring is *Tuple Function Arguments*, which can decrease the arity of a function by packing some of the formal parameters into a tuple structure. Our definition of these transformations are given in [14, 37, 15]. (These refactorings are also known as *Extract/Inline Function*, *Extract Variable / Inline Temp*, *Reorder Parameters*, and *Introduce Parameter Object* [38].) The transformation *Eliminate Code Duplicate* is also well studied in the literature, with implementations for the Erlang language [39, 40]. Due to space limits, we omit the detailed description of these transformations. In this section we turn our attention to some novel ones instead.

We have identified a number of refactorings which are essential for restructuring divide-and-conquer functions, and also some clean-up transformations, which are not obligatory to use, but which may improve code quality a lot. Here we explain how the novel transformations behave, and provide (informally) their side conditions.

In the description of the transformations we use concrete Erlang syntax with typewriter font, and meta-variables with italics: ε (expression), p (pattern), g (guard), b (body, i.e. sequence of expressions), f (function symbol), V (variable). A comma- or semicolon-separated sequence of syntactic units σ_i with $i \in [1..n]$ is written as $\langle \sigma_i \rangle_{i\in[1..n]}$. In the abstract syntax used here, we make guards explicit in function clauses and case clauses. (Note that a guard which always evaluates to *true* can be omitted in the concrete syntax.) Hence we write the definition of a unary function given with a single clause as: $f(p)\ \texttt{when}\ g \to b$. Some of the transformations are illustrated with examples, which will appear also in later sections. The transformations are denoted with $\Rightarrow$ and $\Leftrightarrow$ (unidirectional and bidirectional), indicating the way it is useful to perform them.

Function Clauses to/from Case Clauses. Pattern matching on function clause level can be moved inside the function definition as a top-level case-expression, and vice versa: a top-level case-expression can be turned into function clauses. For simplicity, we defined these transformations only for unary functions. (Functions can be turned unary using the Tuple Function Arguments refactoring.)

$$\Big\langle f(p_i)\ \texttt{when}\ g_i \to b_i \Big\rangle_{i\in[1..n]} \qquad\qquad \overset{?}{\equiv}$$
$$f(V) \to \texttt{case}\ V\ \texttt{of}\ \Big\langle p_i\ \texttt{when}\ g_i \to b_i \Big\rangle_{i\in[1..n]}\ \texttt{end}$$

The side condition of the transformations is that the variable V should not occur in the patterns, guards, and bodies. (Another variable with the same name may occur in the bodies, i.e. introduced as a formal argument of a fun-expression though.) The transformations in both ways preserve the meaning of any semantically correct program, but the two variants raise different exceptions when violating dynamic semantical rules of the language (pattern-matching failure). Therefore it may be a good idea to implement the transformations in such a way that the refactoring tool notifies the user, or asks permission from them, before performing the transformation, or the transformation may be extended with compensations (modifying dynamically enclosing exception handlers accordingly).

```
ms([])  ->  [];
ms([H]) -> [H].
```

$\Leftrightarrow$

```
ms(List) -> case List of [] -> [];
                         [H] -> [H]
            end.
```

Group Case Branches. This refactoring allows us to partition the branches of a case-expression into two groups. In the divide-and-conquer case, this can separate the base case(s) and the recursive case(s) of a function. When applying this transformation, we can select the branches to belong to the `true`-group, and the other branches will belong to the `false`-group. (This selection is captured below with the meta-function $\beta : [1..n] \to \{\text{true}, \text{false}\}$.)

The transformation replaces the original case-expression with a combination of four case-expressions. In order to avoid warnings about unused variables, care must be taken with the patterns and guards of the first case-expression: for every binding occurrence of a variable in these patterns, a fresh name must be introduced, starting with an underscore. This also holds for "comparing" situations, when the same unbound variable occurs multiple times in the same pattern (such as `[_H,_H]`). Below, p_i' and g_i' denote the suitably modified variants of p_i and g_i.

$$\texttt{case}\ \varepsilon\ \texttt{of}\ \left\langle p_i\ \texttt{when}\ g_i \to b_i \right\rangle_{i\in[1..n]}\ \texttt{end}$$

$$\overset{?}{\equiv}$$

$$\begin{array}{l}\texttt{case}\ \left(\texttt{case}\ \varepsilon\ \texttt{of}\ \left\langle p_i'\ \texttt{when}\ g_i' \to \beta(i) \right\rangle_{i\in[1..n]}\ \texttt{end}\right)\ \texttt{of} \\ \quad \texttt{true} \to \texttt{case}\ \varepsilon\ \texttt{of}\ \left\langle p_i\ \texttt{when}\ g_i \to b_i \right\rangle_{i\in[1..n]\wedge\beta(i)=\text{true}}\ \texttt{end;} \\ \quad \texttt{false} \to \texttt{case}\ \varepsilon\ \texttt{of}\ \left\langle p_i\ \texttt{when}\ g_i \to b_i \right\rangle_{i\in[1..n]\wedge\beta(i)=\text{false}}\ \texttt{end} \\ \texttt{end}\end{array}$$

The side condition of the transformation is that the head-expression of the case, i.e. ε, is pure (it does not contain side-effects, such as IO, sending/receiving messages etc.). Since patterns and guards are always pure in Erlang, we need not include them in the side condition.

```
ms(L) ->
  case L of
      [] -> [];
     [H] -> [H];
      _  -> {L1,L2} = lists:split(length(L) div 2, L),
            lists:merge( ms(L1), ms(L2) )
  end.
```

$\Downarrow$

```
ms(L) ->
  IsBase = case L of []  -> true;
                     [_H] -> true;
                      _   -> false
           end,
  case IsBase ->
    true  ->
      case L of [] -> [];
               [H] -> [H]
      end;
    false ->
      case L of
        _  -> {L1,L2} = lists:split(length(L) div 2, L),
              lists:merge( ms(L1), ms(L2) )
      end
  end.
```

Eliminate Single Branch. When a case-expression contains only a single branch, it can be simplified to an optional match-expression followed by the branch body. We need no side condition here.

$$\texttt{case}\ \varepsilon\ \texttt{of}\ p \rightarrow b\ \texttt{end} \quad \overset{?}{\equiv} \quad \texttt{begin}\ p = \varepsilon,\ b\ \texttt{end}$$

Similarly to the *Function Clauses to/from Case Clauses* transformation, the equivalence of the two expressions is provided when the expressions are semantically correct (the pattern match succeeds), but raise different exceptions otherwise. The user can be notified, or compensation in dynamically containing exception handlers can be performed when possible.

If the transformation is carried out on a top-level expression in a body, the `begin-end` construct might be superfluous, and can be left out. Moreover, if the pattern is statically guaranteed to match and does not bind any variables (for example p is ε or `_`), the match-expression can be left out as well, leaving just the body b.

This clean-up transformation can be applied on the fourth case-expression in the previous example (no match-expression is needed), but the general form is better illustrated on the following example.

```
case L of [H] -> [H] end
```

$\Rightarrow$

```
[H] = L, [H]
```

Introduce lists:map/2. A divide-and-conquer function calls itself recursively multiple times, and in the canonical form these calls are collected in an application of `lists:map/2`. A list-expression with elements which are calls to the same (unary) function can be refactored into a call to `lists:map/2`.

$$\Big[\ \langle f(\varepsilon_i)\rangle_{i\in[1..n]}\ \Big] \quad \overset{?}{\equiv} \quad \texttt{lists:map}\Big(\texttt{fun}\ f/1,\ \Big[\ \langle \varepsilon_i\rangle_{i\in[1..n]}\ \Big]\Big)$$

The side condition is that the expressions ε_i or the body of f are pure. (Otherwise side effects could take place in different order.)

```
[ms(L1), ms(L2)]
```

$\Rightarrow$

```
lists:map(fun ms/1, [L1,L2])
```

A variant of this refactoring can transform a list comprehension with a single generator to an application of `lists:map/2` [10].

Bindings To List. To prepare for the previous transformation, it might be necessary to turn a sequence of match-expressions into a single match-expression with a list pattern and a list-valued right-hand side.

$$\langle p_i = \varepsilon_i \rangle_{i\in[1..n]} \quad \overset{?}{\equiv} \quad \Big[\ \langle p_i \rangle_{i\in[1..n]} \ \Big] = \Big[\ \langle \varepsilon_i \rangle_{i\in[1..n]} \ \Big]$$

A completely safe side condition here is that the expressions ε_i are pure. When they are not, and the program is not semantically correct (i.e. some p_j does not match the corresponding ε_j), the list-binding may execute more side-effects than the list of bindings before failing with an exception.

```
S1 = ms(L1), S2 = ms(L2)
```

$\Rightarrow$

```
[S1,S2] = [ms(L1), ms(L2)]
```

Reorder Expressions. This transformation allows us to change the order of expressions in a body (a sequence of expressions).

$$\varepsilon_1, \ \varepsilon_2 \quad \overset{?}{\equiv} \quad \varepsilon_2, \ \varepsilon_1$$

The side condition is that the second expression must not use any variables bound in the first one (since the transformation would change the binding structure of variables, if the code remained meaningful at all). Furthermore, at most one of the two expressions may contain side-effects. (If both expressions were impure, the transformation would change the order in which side effects take place.)

Move Expression Out Of Case. When the last expression of *all the branches* of a case-expression is the same, it can be moved out of the case expression.

$$\begin{aligned} &\texttt{case}\ \varepsilon\ \texttt{of}\ \Big\langle p_i\ \texttt{when}\ g_i \to b_i,\ \varepsilon' \Big\rangle_{i\in[1..n]}\ \texttt{end} \quad \overset{?}{\equiv} \\ &\texttt{case}\ \varepsilon\ \texttt{of}\ \Big\langle p_i\ \texttt{when}\ g_i \to b_i \Big\rangle_{i\in[1..n]}\ \texttt{end},\ \varepsilon' \end{aligned}$$

This transformation can be performed without checking any specific side-condition. Note, however, that we defined it in a very strict manner, requiring that the expressions in the end of the original case branches *lexically* match.

Also note that when performed on a subexpression of a top-level expression, it is necessary to put the resulting sequence of expressions inside a `begin-end` construct – similarly to the *Eliminate Single Branch* refactoring.

```
case Problem of
  {{ Node, Depth }, max} -> F = fun lists:max/1,
                            D = min,
                            work(Depth, Node, F, D);
  {{ Node, Depth }, min} -> F = fun lists:min/1,
                            D = max,
                            work(Depth, Node, F, D)
end
```

$$\Downarrow$$

```
case Problem of
  {{ Node, Depth }, max} -> F = fun lists:max/1,
                            D = min;
  {{ Node, Depth }, min} -> F = fun lists:min/1,
                            D = max
end
work(Depth, Node, F, D);
```

Move Expression Into Case. An expression preceding a case-expression can be moved inside the case-expression, by repeating it in *all of the branches*.

$$\varepsilon', \texttt{case}\ \varepsilon\ \texttt{of}\ \langle p_i\ \texttt{when}\ g_i \rightarrow b_i \rangle_{i\in[1..n]}\ \texttt{end} \quad \overset{?}{\equiv}$$
$$\texttt{case}\ \varepsilon\ \texttt{of}\ \langle p_i\ \texttt{when}\ g_i \rightarrow \varepsilon',\ b_i \rangle_{i\in[1..n]}\ \texttt{end}$$

Since this transformation changes the evaluation order of expressions, its side conditions are similar to those of *Reorder Expressions*. The variables bound in ε' should not be used in ε and in any of the patterns p_i and guards g_i. Furthermore, at least one of ε' and ε must be pure. (Patterns and guards cannot have side effects in Erlang.)

Introduce Unused Parameter. In Section 2 we have seen that the constituents of a divide-and-conquer definition, namely `is_base/1`, `base/1` and `divide/1`, expects the same parameter: a *problem*. To lift functions to the same state space, we may need a transformation which can add unused parameters to a function definition. This extension of the state space of a function does not affect the meaning of the function.

The refactoring extends the formal parameter list of the function with a parameter (since it is not used in the function definition, an unnamed parameter suffices). Moreover, as a compensation, the calls to this function should also be extended with a dummy actual parameter.

$$\Big\langle f\big(\langle p_{i,j}\rangle_{j\in[1..m]}\big)\ \texttt{when}\ g_i \rightarrow b_i \Big\rangle_{i\in[1..n]} \quad \Rightarrow$$
$$\Big\langle f\big(\langle p_{i,j}\rangle_{j\in[1..m]}\texttt{,}\ \_\big)\ \texttt{when}\ g_i \rightarrow b_i \Big\rangle_{i\in[1..n]}$$

and for all calls c of the function, an expression ε_c is chosen with

$$f\big(\langle\varepsilon_j\rangle_{j\in[1..m]}\big) \quad\Rightarrow\quad f\big(\langle\varepsilon_j\rangle_{j\in[1..m]},\ \varepsilon_c\big).$$

The side condition is that the dummy actual parameters passed to the modified function are free of side effects, and are statically guaranteed not to raise an exception.

Technically, the transformation is carried out at a call site of a function, specifying an (side-effect free) expression as actual parameter to match the novel formal. All other calls of the function in the program will be extended with the atom `undefined` as the last actual parameter.

Merge Function Definitions. A set of unary function definitions are replaced by this transformation with a single function definition, where the clauses of the new function are coming from the clauses of the original functions. In the new function an additional argument (discriminator) is introduced, and the clauses of the new function will pattern-match on this argument. The name of the new function and the (distinct) values for the discriminator are the parameters of the refactoring.

$$\big\langle f_j(p_{i,j})\ \texttt{when}\ g_{i,j} \to b_{i,j}\big\rangle_{i\in[1..n_j],j\in[1..m]} \quad\Rightarrow$$
$$\Big\langle f\big(p_{i,j},\beta(j)\big)\ \texttt{when}\ g_{i,j} \to b_{i,j}\Big\rangle_{i\in[1..n_j],j\in[1..m]}$$

and for all calls of the function f_j:

$$f_j(\varepsilon) \quad\Rightarrow\quad f\big(\varepsilon,\ \beta(j)\big).$$

The side condition is that the name of the new function must be fresh, and that the functions f_i are either all exported in the containing module, or none of them is exported. As a compensation, the new function must be exported instead of the original ones, if those were exported.

5. Transformations in action

In this section we look at the *mergesort* function, and investigate how it can be refactored to the expected canonical form. In this particular example we are able to achieve our goal by simply applying semantics-preserving transformations using a refactoring tool, without any further manual editing of the code. This is indeed the best possible workflow in this methodology: if software developers have confidence in the refactoring tool, they can restructure the code without worrying about accidentally breaking it. Therefore, they can work very effectively: they can focus on achieving the desired code structure, and need not waste time and effort on understanding every small detail in the code.

We start with the definition of `ms/1` as shown in Section 2.

```
ms(  [] ) -> [];
ms( [H] ) -> [H];
ms(  L  ) -> {L1,L2} = lists:split(length(L) div 2, L),
             lists:merge( ms(L1), ms(L2) ).
```

First of all, we apply *Function Clauses to Case Clauses*. We also *Rename* the parameter to Lst, in order to have a meaningful name.

```
ms(Lst) ->
  case Lst of
        [] -> [];
       [H] -> [H];
        L  -> {L1,L2} = lists:split(length(L) div 2, L),
              lists:merge( ms(L1), ms(L2) )
  end.
```

Next, we apply *Group Case Branches*, and select branches 1 and 2 as the true-branch.

```
ms(Lst) ->
  IsBase = case Lst of []  -> true;
                       [_H] -> true;
                        _L  -> false
           end,
  case IsBase of
    true  ->
       case Lst of [] -> [];
                  [H] -> [H]
       end;
    false ->
       case Lst of
         L -> {L1,L2} = lists:split(length(L) div 2, L),
              lists:merge( ms(L1), ms(L2) )
       end
  end.
```

Applying the *Introduce Function* refactoring twice, we can create the is_base/1 and base/1 functions, and call them in ms/1.

```
base(Lst) -> case Lst of [] -> [];
                        [H] -> [H]
             end.

is_base(Lst) -> ...       % similar

ms(Lst) ->
  IsBase = is_base(Lst),
  case IsBase of
    true  ->
```

```
        base(Lst);
    false ->
        case Lst of
          L -> {L1,L2} = lists:split(length(L) div 2, L),
               lists:merge( ms(L1), ms(L2) )
        end
  end.
```

Both base/1 and is_base/1 can be beautified by applying *Case Clauses to Function Clauses*. We can also get rid of the IsBase variable using *Eliminate Variable*. Moreover, the inner case-expression can be removed with the *Eliminate Single Branch* refactoring.

```
base( []) -> [];
base([H]) -> [H].
is_base ...              % similar

ms(Lst) ->
  case is_base(Lst) ->
    true  ->
        base(Lst);
    false ->
        L = Lst,
        {L1,L2} = lists:split(length(L) div 2, L),
        lists:merge( ms(L1), ms(L2) )
  end.
```

We can simplify the code further with *Eliminate Variable*, and replace all occurrences of L with its definition, Lst.

```
ms(Lst) ->
  case is_base(Lst) ->
    true  ->
        base(Lst);
    false ->
        {L1,L2} = lists:split(length(Lst) div 2, Lst),
        lists:merge( ms(L1), ms(L2) )
  end.
```

Now most of ms/1 looks like the canonical form for the divide-and-conquer pattern, only the false-branch needs to be transformed further. Using *Introduce Variable* twice, the variables SL1 and SL2 are introduced.

```
        {L1,L2} = lists:split(length(Lst) div 2, Lst),
        SL1 = ms(L1),
        SL2 = ms(L2),
        lists:merge( SL1, SL2 )
```

The two new bindings for SL1 and SL2 can be turned into a compound binding with *Bindings to List*.

```
        [SL1, SL2] = [ms(L1), ms(L2)],
```

The list on the right-hand side turns into an application of the `lists:map/2` function, when we apply the *Introduce `lists:map/2`* transformation.

```
        [SL1, SL2] = lists:map(fun ms/1, [L1,L2]),
```

The new variables are introduced now with the *Introduce Variable* refactoring: `SubProblems` for the second argument of `lists:map/2`, and `Solutions` for its result.

```
        {L1,L2} = lists:split(length(Lst) div 2, Lst),
        SubProblems = [L1, L2],
        Solutions = lists:map(fun ms/1, SubProblems),
        [SL1, SL2] = Solutions,
        lists:merge( SL1, SL2 )
```

The divide and combine functions can be extracted with the *Introduce Function* transformation.

```
ms(Lst) ->
  case is_base(Lst) ->
    true  ->
       base(Lst);
    false ->
       SubProblems = divide(Lst),
       Solutions = lists:map(fun ms/1, SubProblems),
       combine(Solutions)
  end.

divide(Lst) ->
  {L1, L2} = lists:split(length(Lst) div 2, Lst),
  SubProblems = [L1, L2],
  SubProblems.

combine(Solutions) ->
  [SL1, SL2] = Solutions,
  lists:merge( SL1, SL2 ).
```

We have reached the canonical form for divide-and-conquer pattern candidate; the `ms/1` function is ready for the *Introduce dnc* transformation, which replaces the body of the function with a call to the high-level pattern implementation in the `sk_hlp` module. As a final step, we can clean up the code of `divide/1` with *Eliminate Variable*, and remove `SubProblems`.

```
ms(Lst) -> (sk_hlp:dc(fun is_base/1, fun base/1,
                      fun divide/1, fun combine/1)
           )(Lst).
```

```
divide(Lst) ->
  {L1, L2} = lists:split(length(Lst) div 2, Lst),
  [L1, L2].
```

6. Generalized canonical form

After investigating several concrete examples of divide-and-conquer function definitions, we realized that it is useful to generalize the canonical form compared to the nice and simple one usually found in the literature, and presented in the previous sections. This is a straightforward generalization, allowing local bindings to be shared in the extracted *is_base*, *base*, *divide*, and *combine* functions.

```
dcgen(Problem) ->
  Bindings = bindings(Problem)
  case is_base(Bindings) of
    true  -> base(Bindings);
    false ->
       {SubProblems,Bindings2} = divide(Bindings),
       Solutions = lists:map(fun dcgen/1, SubProblems),
       combine(Solutions,Bindings2)
  end.
```

According to this scheme, computations can be performed, and their results stored in local variables (collected in a tuple `Bindings`) at the beginning of the divide-and-conquer function. These bindings are passed to `is_base/1`, `base/1`, and `divide/1`. This is very useful if the values computed by `bindings/1` are shared by some of these functions, because the overhead of recomputing the values from `Problem` multiple times can be avoided. The value describing the problem to solve, `Problem`, can also be wrapped into the `Bindings` tuple. This technique allows as to make the domain of `is_base/1`, `base/1` and `divide/1` the same.

The `divide/1` function can also introduce bindings, and return them together with the list of subproblems. This `Bindings2` tuple may contain values from `Bindings`, but some other values as well. In contrast to the elements of `SubProblems`, the bindings do not go through the recursive calls of the divide-and-conquer function, but go into `combine/2` directly.

Technically, we could avoid the generalized canonical from, and do our job with the original simple one, using for example the following two tricks. Firstly, the call to *bindings* can be moved inside *is_base*, *base* and *divide*. Secondly, the `Bindings2` tuple produced in *divide* could be zipped into each subproblem, and then passed untouched into the solutions of the subproblems by the recursive calls to the divide-and-conquer function. This is demonstrated with the following code scheme, which contains already the simple *dc* canonical form.

```
dcgen(Problem) ->
  {Solution,dummy} = dc({Problem,dummy}),
  Solution.

dc(DecoratedProblem) ->
  case is_base(DecoratedProblem) of
    true  -> base(DecoratedProblem);
    false ->
       SubProblems = divide(DecoratedProblem),
       Solutions = lists:map(fun dc/1, SubProblems),
       combine(Solutions)
  end.

base({Problem,DoNotTouch}) ->
  Bindings = bindings(Problem),
  % continue with original base body using Bindings

is_base({Problem,DoNotTouch}) ->
  Bindings = bindings(Problem), % orig. is_base body

divide({Problem,DoNotTouch}) ->
  Bindings = bindings(Problem),
  {SubProblems, Bindings2} =    % original divide body
  [{P, {DoNotTouch,Bindings2}} || P <- SubProblems].

combine( DecoratedSolutions =
         [{_,{DoNotTouch,Bindings2}}|_Tail] ) ->
  Solutions = [S || {S, _} <- DecoratedSolutions],
  Solution =    % original combine body
  {Solution, DoNotTouch}.
```

The price to pay is high due to the increased complexity of `divide/1` and `combine/1`. Even more importantly, the latter scheme incurs heavy overhead in terms of execution time and consumed memory. Recomputing values multiple times, and passing around values down the call chain and back may seriously degrade performance. (Other solutions to put the simple canonical form in place of the generalized one, such as computing *combine* as a closure in *divide*, also suffer from this problem.) This justifies the introduction of the generalized canonical form for the divide-and-conquer candidate.

Note, finally, that the generalized divide-and-conquer scheme can express the original one by choosing the identity function as *bindings*, and by using an empty tuple for `Bindings2`. However, for convenience reasons it is advantageous to keep both the generalized and the original divide-and-conquer schemes, and provide refactorings to introduce calls to higher-order divide-and-conquer functions for both variants.

6.1. Example

Now we illustrate the applicability of the generalized canonical form on another divide-and-conquer function, one which implements Karatsuba big integer multiplication. This function takes two bitstrings (representing big integer numbers), cuts them into lower and higher halves, and computes the product of the original numbers with three recursive multiplications of half-sized bitstrings.

```
karatsuba(Num1, Num2) ->
  S1 = bit_size( Num1 ),
  S2 = bit_size( Num2 ),
  case {Num1, Num2} of
    {<<0:1>>, _       }     -> <<0: S2>>;     % base case
    {_       , <<0:1>>}     -> <<0: S1>>;     % base case
    {<<1:1>>, _       }     -> Num2;          % base case
    {_       , <<1:1>>}     -> Num1;          % base case
    _                       ->                % recursive
       M = max( S1, S2 ),
       M2 = M - (M div 2),
       <<Low1 : M2/bitstring, High1/bitstring>> = Num1,
       <<Low2 : M2/bitstring, High2/bitstring>> = Num2,
       Z0 = karatsuba(Low1, Low2),
       Z1 = karatsuba(add(Low1,High1), add(Low2,High2)),
       Z2 = karatsuba(High1, High2),
       add(  add( shift(Z2, M2*2), Z0 ),
             shift( sub(Z1, add(Z2,Z0)), M2 )  )
  end.
```

We can carry out a transformation sequence on this definition very similarly to the *mergesort* example. Let us point out that in this example, as well, the application of semantics-preserving transformations using a refactoring tool can take us to the canonical form, without any manual editing of the code. Before we start the transformation sequence learnt from the *mergesort* example, the refactoring *Tuple Function Arguments* should be used to turn `karatsuba/2` into a unary function.

```
karatsuba({Num1, Num2}) ->
  ...
     Z0 = karatsuba({Low1, Low2}),
     Z1 = karatsuba({add(Low1,High1), add(Low2,High2)}),
     Z2 = karatsuba({High1, High2}),
  ...
```

Now we can really apply the previously successful transformation sequence. We can start with *Group Case Branches*, then extract *is_base* and *base* with *Introduce Function*, apply *Eliminate Variable* on the introduced `IsBase`, and simplify the code with *Eliminate Single Branch*. We can use *Bindings to List* and *Introduce `lists:map/2`* for the recursive calls, and *Introduce Variable* to add `SubProblems` and `Solutions`. Then we can extract the *divide* and *combine* functions using *Introduce Function*. Note, however, that on the one hand, *divide* returns a pair, since it binds two variables – namely `SubProblems` and `M2` – which are used outside of *divide*. (The refactoring allows us to make a decision on the order of the two elements in the result: we should choose `SubProblems` to come first.) On the other hand, *combine* depends on `Solutions` and `M2`, so it will be a binary function. We may need to apply *Reorder Function Arguments* to ensure that `Solutions` is the first argument.

```
karatsuba({Num1, Num2}) ->
  S1 = bit_size( Num1 ),
  S2 = bit_size( Num2 ),
  case is_base(Num1, Num2) of
    true  -> base(Num1, Num2, S1, S2);
    false ->
       {SubProblems, M2} = divide(Num1, Num2, S1, S2),
       Solutions=lists:map(fun karatsuba/1,SubProblems),
       combine(Solutions,M2)
  end.

divide(Num1, Num2, S1, S2) ->
    _ = {Num1, Num2},
    M = max(S1, S2),
    M2 = M - (M div 2),
    <<Low1:M2/bitstring, High1/bitstring>> = Num1,
    <<Low2:M2/bitstring, High2/bitstring>> = Num2,
    SubProblems = [ {Low1,Low2}
                  , {add(Low1,High1),add(Low2,High2)}
                  , {High1,High2} ],
    {SubProblems, M2}.

combine(Solutions, M2) ->
    [Z0, Z1, Z2] = Solutions,
    add(  add( shift(Z2, M2*2), Z0 ),
          shift( sub(Z1, add(Z2,Z0)), M2 )  ).
```

At this point we can start to create the *bindings* function. As a first step, we introduce two temporary variables for `Num1` and `Num2` by applying *Introduce Variable* twice, on any of the non-binding occurrences of `Num1` and `Num2`, respectively.

```
karatsuba({Num1, Num2}) ->
  Tmp1 = Num1,
  S1 = bit_size( Num1 ),
  Tmp2 = Num2,
  S2 = bit_size( Num2 ),
  case is_base(Tmp1, Tmp2) of
    true  -> base(Tmp1, Tmp2, S1, S2);
    false ->
       {SubProblems, M2} = divide(Tmp1, Tmp2, S1, S2),
       Solutions=lists:map(fun karatsuba/1,SubProblems),
       combine(Solutions,M2)
  end.
```

We need to ensure that the domains of *is_base*, *base*, and *divide* are identical. For this reason, we use *Introduce Unused Parameter* twice on the call to *is_base*, passing `S1` and `S2` as the unused parameters. Finally, we create a 4-tuple from the parameters of *is_base*, *base* and *divide* using *Tuple Function Arguments*.

```
karatsuba({Num1, Num2}) ->
  Tmp1 = Num1,
  S1 = bit_size( Num1 ),
  Tmp2 = Num2,
  S2 = bit_size( Num2 ),
  case is_base({Tmp1, Tmp2, S1, S2}) of
    true  -> base({Tmp1, Tmp2, S1, S2});
    false ->
        {SubProblems, M2} = divide({Tmp1, Tmp2, S1, S2}),
        Solutions=lists:map(fun karatsuba/1,SubProblems),
        combine(Solutions,M2)
  end.

is_base({Num1, Num2,  _,  _}) -> ...
base   ({Num1, Num2, S1, S2}) -> ...
divide ({Num1, Num2, S1, S2}) -> ...
```

The occurrences of the 4-tuple in `karatsuba/1` can be extracted into a variable `Bindings` with the *Introduce Variable* refactoring.

```
karatsuba({Num1, Num2}) ->
  Tmp1 = Num1,
  S1 = bit_size( Num1 ),
  Tmp2 = Num2,
  S2 = bit_size( Num2 ),
  Bindings = {Tmp1, Tmp2, S1, S2},
  case is_base(Bindings) of
    true  -> base(Bindings);
    false ->
        {SubProblems, M2} = divide(Bindings),
        Solutions=lists:map(fun karatsuba/1,SubProblems),
        combine(Solutions,M2)
  end.
```

We can extract function *bindings* with *Introduce Function*. Since it depends on two variables, `Num1` and `Num2`, it will be a binary function. Using *Tuple Function Arguments*, we can turn it into unary, and arrive at the generalized canonical form.

```
karatsuba({Num1, Num2}) ->
```

```
  Bindings = bindings({Num1,Num2}),
  case is_base(Bindings) of
    true  -> base(Bindings);
    false ->
       {SubProblems, M2} = divide(Bindings),
       Solutions=lists:map(fun karatsuba/1,SubProblems),
       combine(Solutions,M2)
  end.

bindings({Num1,Num2}) ->
  Tmp1 = Num1,
  S1 = bit_size( Num1 ),
  Tmp2 = Num2,
  S2 = bit_size( Num2 ),
  Bindings = {Tmp1, Tmp2, S1, S2},
  Bindings.
```

As the last step, we can clean up `bindings/1`, using *Eliminate Variable* on `Tmp1`, `Tmp2`, `S1`, `S2`, and `Bindings`.

```
bindings({Num1,Num2}) ->
  {Num1, Num2, bit_size( Num1 ), bit_size( Num2 )}.
```

7. Mutually recursive functions

Divide-and-conquer computations may be spread in multiple, mutually recursive functions. We should now consider how to deal with those. Without aiming at covering all possible issues, in this section we look at two, quite different, examples. The first example will be a simple variant of *mergesort*.

```
ms(   [] ) ->  [];
ms( [H] ) -> [H];
ms(   L   ) ->
  {L1, L2} = lists:split ( length(L) div 2, L ),
  sort_and_merge(L1, L2).

sort_and_merge( List1, List2 ) ->
  lists:merge( ms(List1), ms(List2) ).
```

In a set of mutually recursive functions, any function can be chosen as the divide-and-conquer candidate: indeed, the pattern candidate discovery will find and report all of them. Sometimes, however, one of the functions seems naturally the best choice; in this case it is `ms/1`. An indirect recursion can often be turned into direct recursion by applying the *Eliminate Function* refactoring, which inlines the body of a function at a call site.

```
ms(   [] ) ->  [];
ms( [H] ) -> [H];
ms(   L  ) ->
  {L1, L2} = lists:split ( length(L) div 2, L ),
  lists:merge( ms(L1), ms(L2) ).
```

After eliminating the call to `sort_and_merge/2`, we arrived at the definition addressed in Section 5.

In a more complicated case of mutual recursion the *Eliminate Function* technique may not be suitable. Consider for instance the following *minimax* algorithm. Function `mm_max/2` calls `mm_min/2` in the head of a list comprehension (hence probably multiple times in the same execution path), and vice-versa. Both of these functions are candidates for the divide-and-conquer pattern. The `Node` parameter gives the starting node of the minimax-search in the game tree, and `Depth` gives the number of levels to visit in the tree. A terminal node in the tree (`terminal/1`) is described with its value (`value/1`), and a non-terminal node has children in the tree (`children/1`).

```
mm_max( Node, Depth ) ->
  case Depth == 0 orelse terminal(Node) of
    true  ->
       value( Node );
    false ->
       lists:max([mm_min(C,Depth-1)||C<-children(Node)])
  end.

mm_min( Node, Depth ) ->
  case Depth == 0 orelse terminal(Node) of
    true  ->
       value( Node );
    false ->
       lists:min([mm_max(C,Depth-1)||C<-children(Node)])
  end.
```

We propose a different approach for resolving indirect recursion in this kind of situations. We can collapse the set of mutually recursive definitions into a single function with the *Merge Function Definitions* refactoring. For simplicity, we have defined this refactoring only on unary functions, so we need to use *Tuple Function Arguments* first. The *Merge Function Definitions* refactoring introduces a discriminator parameter, for which the actual parameters should be provided. In our case we use the atoms `max` and `min` as discriminating values.

```
mm( {Node, Depth}, max ) ->
  ...
     lists:max([mm({C,Depth-1},min)||C<-children(Node)])
  end;
mm( {Node, Depth}, min ) ->
  ...
```

```
      lists:min([mm({C,Depth-1},max)||C<-children(Node)])
  end.
```

In order to exploit the symmetry in these function clauses, we capture the differences in variables introduced for expressions `min`, `max`, `lists:max` and `lists:min` (*Introduce Variable* applied 4 times).

```
mm( {Node, Depth}, max ) ->
  D = min,
  F = fun lists:max/1,
  case Depth == 0 orelse terminal(Node) of
    true  -> value( Node );
    false -> F([mm({C,Depth-1},D)||C <- children(Node)])
  end.
mm( {Node, Depth}, min ) ->
  D = max,
  F = fun lists:min/1,
  case Depth == 0 orelse terminal(Node) of
    true  -> value( Node );
    false -> F([mm({C,Depth-1},D)||C <- children(Node)])
  end.
```

We can extract one of the case-expressions into a function definition (`work/4`) with *Introduce Function*, and use the *Eliminate Duplicated Code* to replace the syntactically equivalent other case-expression with a call to the same function.

```
mm( {Node, Depth}, max ) ->
  D = min,
  F = fun lists:max/1,
  work(Node,Depth,D,F);
mm( {Node, Depth}, min ) ->
  D = max,
  F = fun lists:min/1,
  work(Node,Depth,D,F).

work(Node,Depth,D,F) ->
  case Depth == 0 orelse terminal(Node) of
    true  -> value( Node );
    false -> F([mm({C,Depth-1},D)||C <- children(Node)])
  end.
```

Now we turn the `mm/2` function unary again with the *Tuple Function Arguments* transformation, and apply *Function Clauses to Case Clauses*.

```
mm(Problem) ->
  case Problem of
      {{Node, Depth}, max} ->
        D = min,
        F = fun lists:max/1,
        work(Node, Depth, D, F);
      {{Node, Depth}, min} ->
        D = max,
        F = fun lists:min/1,
        work(Node, Depth, D, F)
  end.
```

The next transformation to apply is *Move out from Case*, with which the call to `work/4` can be moved from the case branches outside of the case-expression.

```
mm(Problem) ->
  case Problem of
    {{Node, Depth}, max} ->
      D = min,
      F = fun lists:max/1;
    {{Node, Depth}, min} ->
      D = max,
      F = fun lists:min/1
  end,
  work(Node, Depth, D, F).
```

The structure of this code is very similar to that of the example with `ms/1` and `sort_and_merge/2`. After inlining the definition of `work/4`, we can apply the usual transformations to shape `mm/1` to the canonical form — without manual editing of the code.

```
mm(Problem) ->
    Bindings = bindings(Problem),
    case is_base(Bindings) of
        true  -> base(Bindings);
        false ->
           {SubProblems, F} = divide(Bindings),
           Solutions = lists:map(fun mm/1, SubProblems),
           combine(Solutions, F)
    end.

bindings( {{Node, Depth}, max} ) ->
  {Node, Depth, min, fun lists:max/1};
bindings( {{Node, Depth}, min} ) ->
  {Node, Depth, max, fun lists:min/1}.

is_base({Node, Depth, _, _}) ->
  Depth == 0 orelse terminal(Node).

base({Node, _, _, _}) -> value(Node).

divide({Node, Depth, D, F}) ->
    { [{{C, Depth - 1}, D} || C<-children(Node)], F}.

combine(Solutions, F) -> F(Solutions).
```

8. Methodology

Based on lessons learnt in the previous sections, we propose a workflow to follow when refactoring a divide-and-conquer pattern candidate into the (simple or generalized) canonical form (Figure 1). Our methodology relies on the extraction of the *bindings* (optional), *is_base*, *base*, *divide* and *combine* functions, and their separation from the recursive calls. We have investigated several function definitions identified by divide-and-conquer pattern discovery [16], and we have found that the methodology presented here is applicable for them. However, these examples do not contain nested case-expressions. Flattening nested case-expressions may require further refactoring transformations, and this is not covered in the current paper.

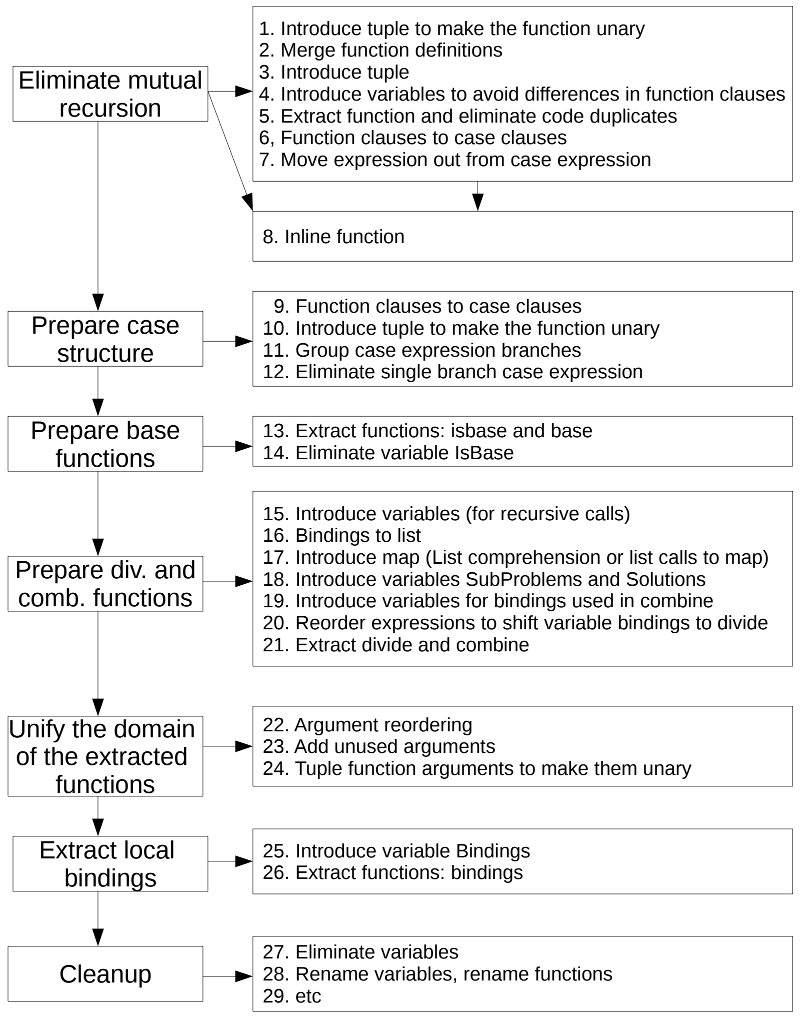
The refactoring process presented here works well for many divide-and-conquer functions, and without any manual editing of the code. However, one can always find examples where manual refactoring cannot be avoided. Extending the refactoring tool with further semantics-preserving transformations might be a solution, but of course the number of transformations the software developer can keep in mind is limited. Therefore a refactoring tool should prefer transformations which are sufficiently generic and flexible.

9. Conclusion

In this paper we present a methodology to refactor divide-and-conquer functions in a functional programming language. The refactoring is defined as a sequence of semantics-preserving code transformations, and is supposed to be performed using a refactoring tool. The aim of the refactoring is to restructure a function describing some divide-and-conquer computation into a "canonical form", which can be further refactored into the application of a generic higher-order divide-and-conquer function with a sequential or parallel implementation (i.e. an instance of a high-level parallel pattern). This process can facilitate the pattern-based parallelization of many computationally intensive software applications.

The main benefit of our approach is that software developers (if they are confident about the soundness of the refactoring tool) need not worry about breaking the code during the refactoring process. Moreover, using pattern candidate discovery (described in an earlier paper), the refactoring tool can automatically find divide-and-conquer pattern candidates: without spending too much time and effort on understanding either the whole code-base or even just the candidate, software developers are able to safely refactor, and increase the performance of, the code.

We have introduced a generalized canonical form for divide-and-conquer functions, and presented a number of semantics-preserving code transformations which are useful for this problem domain. The discussions used the Erlang programming language (and the transformations are implemented in RefactorErl, a refactoring tool for Erlang), but the results naturally apply to other functional languages as well.



Eliminate mutual recursion
1. Introduce tuple to make the function unary
2. Merge function definitions
3. Introduce tuple
4. Introduce variables to avoid differences in function clauses
5. Extract function and eliminate code duplicates
6, Function clauses to case clauses
7. Move expression out from case expression
8. Inline function
Prepare case structure
9. Function clauses to case clauses
10. Introduce tuple to make the function unary
11. Group case expression branches
12. Eliminate single branch case expression
Prepare base functions
13. Extract functions: isbase and base
14. Eliminate variable IsBase
Prepare div. and comb. functions
15. Introduce variables (for recursive calls)
16. Bindings to list
17. Introduce map (List comprehension or list calls to map)
18. Introduce variables SubProblems and Solutions
19. Introduce variables for bindings used in combine
20. Reorder expressions to shift variable bindings to divide
21. Extract divide and combine
Unify the domain of the extracted functions
22. Argument reordering
23. Add unused arguments
24. Tuple function arguments to make them unary
Extract local bindings
25. Introduce variable Bindings
26. Extract functions: bindings
Cleanup
27. Eliminate variables
28. Rename variables, rename functions
29. etc

Figure 1: Workflow for transformation to canonical form

We investigated different implementations of various divide-and-conquer algorithms (which can all be identified by pattern candidate discovery). For these code examples the tool-performed transformations were effective, without the need of manual editing of the code. We concluded that for not very complex and deeply nested function definitions the smart, human guided consecutive application of about a dozen transformations are appropriate to reach the canonical form. (For brevity, we did not include all of the invented transformations in this paper. However, the interested reader can find them in the open-source RefactorErl tool.) As future work, we shall investigate the addition of further transformations to the tool (in order to cope with more complex candidates), as well as develop an algorithm to apply the necessary sequence of transformations all at once, without human guidance.

Acknowledgement

This research has been supported by the Hungarian Government through the New National Excellence Program of the Ministry of Human Capacities.

References

[1] C.-Y. Hsu, H.-F. Wang, H.-C. Wang, K.-K. Tseng, Automatic extraction of face contours in images and videos, Future Generation Computer Systems 28 (1) (2012) 322 – 335.

[2] E. Elmroth, F. Gustavson, High-Performance Library Software for QR Factorizatio, in: Applied Parallel Computing. New Paradigms for HPC in Industry and Academia, Vol. 1947 of LNCS, 2001, pp. 53–63.

[3] W. A. Abed, K. Kucher, M. Krafczyk, M. Wittmann, T. Zeiser, G. Wellein, FETOL: A divide-and-conquer based approach for resilient HPC applications, in: Proc. 3rd Int'l Conf. on Advanced Communications and Computation, 2013, pp. 7–12.

[4] M. Cole, Algorithmic Skeletons: Structured Management of Parallel Computation, MIT Press, Cambridge, MA, USA, 1991.

[5] M. Cole, Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming, Parallel Comput. 30 (3) (2004) 389–406.

[6] S. Campa, M. Danelutto, M. Goli, H. González-Vélez, A. M. Popescu, M. Torquati, Parallel patterns for heterogeneous CPU/GPU architectures: Structured parallelism from cluster to cloud, Future Generation Computer Systems 37 (2014) 354 – 366.

[7] D. Horpácsi, J. Kőszegi, S. J. Thompson, Towards Trustworthy Refactoring in Erlang, in: Proc. 4th Int'l Workshop on Verification and Program Transformation, 2016, pp. 83–103.

[8] K. Hammond, et al., The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems, in: Formal Methods for Components and Objects, Vol. 7542 of LNCS, Springer, 2013, pp. 218–236.

[9] I. Bozó, V. Fördős, Z. Horváth, M. Tóth, D. Horpácsi, T. Kozsik, J. Kőszegi, A. Barwell, C. Brown, K. Hammond, Discovering Parallel Pattern Candidates in Erlang, in: Proc. 13th ACM SIGPLAN Workshop on Erlang, 2014, pp. 13–23.

[10] I. Bozó, V. Fördős, D. Horpácsi, Z. Horváth, T. Kozsik, J. Kőszegi, M. Tóth, Refactorings to enable parallelization, in: Trends in Functional Programming, Vol. 8843 of LNCS, Springer, 2015, pp. 104–121.

[11] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Kőszegi, M. Tejfel, M. Tóth, RefactorErl - Source Code Analysis and Refactoring in Erlang, in: Proc. 12th Symp. on Programming Languages and Software Tools, Tallin, Estonia, 2011, pp. 138–148.

[12] H. Li, S. Thompson, Tool support for refactoring functional programs, in: Proc. 2nd Workshop on Refactoring Tools, ACM, 2008, pp. 1–4.

[13] M. Tóth, I. Bozó, Static Analysis of Complex Software Systems Implemented in Erlang, in: Central European Functional Programming Summer School, Vol. 7241 of LNCS, Springer, 2012, pp. 451–514.

[14] T. Kozsik, Z. Csörnyei, Z. Horváth, R. Király, R. Kitlei, L. Lövei, T. Nagy, M. Tóth, A. Víg, Use Cases for Refactoring in Erlang, in: Central European Functional Programming School, Vol. 5161 of LNCS, Springer, 2008, pp. 250–285.

[15] Z. Horváth, L. Lövei, T. Kozsik, R. Kitlei, A. N. Víg, T. Nagy, M. Tóth, R. Király, Modeling semantic knowledge in Erlang for refactoring, Studia Universitatis Babes-Bolyai, Series Informatica 54 (2009) 7–16.

[16] T. Kozsik, M. Tóth, I. Bozó, Z. Horváth, Static Analysis for Divide-and-Conquer Pattern Discovery, Computing and Informatics 35 (4) (2016) 764–791.

[17] Skel: A Streaming Parallel Skeleton Library for Erlang, `http://skel.weebly.com/` (2016).

[18] ParaPhrase WP2, Final Pattern Definition Report, http://paraphrase-ict.eu/Deliverables (2013).

[19] V. Janjic, K. Hammond, M. Goli, J. McCall, K. Idrees, C. Glass, M. A. Wafai, Bridging the divide: A new methodology for semi-automatic programming of heterogeneous parallel machines, in: High-Level Programming for Heterogeneous and Hierarchical Parallel Systems, 2016.

[20] Z. Mou, P. Hudak, An algebraic model for divide-and-conquer and its parallelism, Journal of Supercomputing 2 (3) (1988) 257–278.

[21] D. Castro, K. Hammond, S. Sarkar, Farms, pipes, streams and reforestation: reasoning about structured parallel processes using types and hylomorphisms, in: Proc. 21st ACM SIGPLAN Int'l Conf. on Functional Programming, 2016, pp. 4–17.

[22] G. Michaelson, A. Ireland, P. King, Towards a Skeleton Based Parallelising Compiler for SML, in: Proc. 9th Int'l Workshop on Implementation of Functional Languages, 1997, pp. 539–546.

[23] C. Brown, V. Janjic, K. Hammond, H. Schner, K. Idrees, J. Gracia, C. Glass, Agricultural reform: More efficient farming using advanced parallel refactoring tools, in: Proc. 22nd Euromicro Int'l Conf. on Parallel, distributed and network-based Processing, IEEE, 2014, pp. 36–43.

[24] C. Brown, H. Loidl, K. Hammond, Paraforming: Forming Haskell Programs using Novel Refactoring Techniques, in: 12th Symp. on Trends in Functional Programming, Vol. 7193 of LNCS, Springer, 2011, pp. 82–97.

[25] S. Aronis, K. Sagonas, On Using Erlang for Parallelization: Experience from Parallelizing Dialyzer, in: Trends in Functional Programming, Vol. 7829 of LNCS, Springer, 2012, pp. 295–310.

[26] S. Aronis, N. Papaspyrou, K. Roukounaki, K. Sagonas, Y. Tsiouris, I. E. Venetis, A Scalability Benchmark Suite for Erlang/OTP, in: Proc. 11th ACM SIGPLAN Workshop on Erlang, ACM, 2012, pp. 33–42.

[27] A. D. Barwell, C. Brown, K. Hammond, W. Turek, A. Byrski, Using Program Shaping and Algorithmic Skeletons to Parallelise an Evolutionary Multi-Agent System in Erlang, Computing and Informatics 35 (4) (2016) 792–818.

[28] S. A. Markstrum, R. M. Fuhrer, Extracting Concurrency via Refactoring in X10, in: Proc. 3rd ACM Workshop on Refactoring Tools, 2009.

[29] S. A. Markstrum, R. M. Fuhrer, T. D. Millstein, Towards Concurrency Refactoring for x10, in: Proc. 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, ACM, 2009, pp. 303–304.

[30] J. Wloka, M. Sridharan, F. Tip, Refactoring for Reentrancy, in: ESEC/FSE '09, ACM, Amsterdam, 2009, pp. 173–182.

[31] K. Molitorisz, Pattern-Based Refactoring Process of Sequential Source Code, in: Proc. 17th European Conf. on Software Maintenance and Reengineering (CSMR), 2013, pp. 357–360.

[32] K. Molitorisz, J. Schimmel, F. Otto, Automatic Parallelization Using Autofutures, in: Proc. Int'l Conf. on Multicore Software Engineering, Performance, and Tools, Springer, 2012, pp. 78–81.

[33] D. Dig, A Refactoring Approach to Parallelism, IEEE Softw. 28 (2011) 17–22.

[34] D. Dig, J. Marrero, M. D. Ernst, How do programs become more concurrent: A story of program transformations, in: Proc. 4th Int'l Workshop on Multicore Software Engineering, ACM, 2011, pp. 43–50.

[35] D. Dig, J. Marrero, M. D. Ernst, Refactoring Sequential Java Code for Concurrency via Concurrent Libraries, in: Proc. 31st Int'l Conf. on Software Engineering, IEEE, 2009, pp. 397–407.

[36] J. Mak, K.-F. Faxén, S. Janson, A. Mycroft, Estimating and exploiting potential parallelism by source-level dependence profiling, in: Proc. 16th Int'l Euro-Par Conf. on Parallel Processing: Part I, Springer, 2010, pp. 26–37.

[37] I. Bozó, M. Tóth, Restructuring Erlang programs using function related refactorings, in: Proc. 11th Symp. on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering, 2009, pp. 162–176.

[38] M. Fowler, Catalog of refactorings, `https://www.refactoring.com/catalog/` (2013).

[39] H. Li, S. Thompson, Similar code detection and elimination for erlang programs, in: Practical Aspects of Declarative Languages: 12th Int'l Symp., PADL, 2010, pp. 104–118.

[40] Wiki page of RefactorErl, `http://pnyf.inf.elte.hu/trac/refactorerl` (2016).