Computing and Informatics, Vol. 35, 2016, 764–791

STATIC ANALYSIS FOR DIVIDE-AND-CONQUER PATTERN DISCOVERY

Tamás Kozsik

Eötvös Loránd University, Budapest, Hungary
e-mail: `kto@elte.hu`

Melinda Tóth, István Bozó, Zoltán Horváth

ELTE-Soft Nonprofit Ltd., Budapest, Hungary
e-mail: {`tothmelinda, bozoistvan, hz`}`@elte.hu`

Abstract. Routines implementing divide-and-conquer algorithms are good candidates for parallelization. Their identifying property is that such a routine divides its input into "smaller" chunks, calls itself recursively on these smaller chunks, and combines the outputs into one. We set up conditions which characterize a wide range of d & c routine definitions. These conditions can be verified by static program analysis. This way d & c routines can be found automatically in existing program texts, and their parallelization based on semi-automatic refactoring can be facilitated. We work out the details in the context of the Erlang programming language.

Keywords: Erlang, divide-and-conquer, pattern, parallelization

Mathematics Subject Classification 2010: 68-N19

1 INTRODUCTION

Divide-and-conquer is a principle that is in the heart of many useful algorithms in different domains, including searching, sorting, FFT and number theory. 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 as a high-level parallel programming pattern [5].

This fact is recognized in the EU FP7 ParaPhrase project (Parallel Patterns for Adaptive Heterogeneous Multicore Systems), which "aims to produce a new structured design and implementation process for heterogeneous parallel architectures, where developers exploit a variety of parallel patterns to develop component-based applications that can be mapped to the available hardware resources, and which may then be dynamically remapped to meet application needs and hardware availability" [27]. In this project, software tools to facilitate parallel programming are designed, which allow programmers to identify parallelizable components, and to refactor them into occurrences of parallel patterns. These patterns can often be expressed in terms of algorithmic skeletons [5]. The implementation of such skeletons may support the dynamic mapping (and re-mapping) of algorithm components onto available hardware resources, and thus support efficient and adaptive resource usage on a heterogeneous many-core machine [8]. Software developers, even when they are not concurrency experts, can make a very good use of these skeleton implementations, if they are collected in a reusable program library, such as the `skel` library [24] for the Erlang programming language [11].

In this paper we focus on the identification of parallelizable components, viz. *pattern discovery*, in particular the identification of divide-and-conquer structures (or d & c for short). Pattern discovery is a technique to use static program analyses to find those fragments of a possibly large code body which exhibit the same behaviour as a pattern; such code fragments are called *pattern candidates*. The pattern discovery technique can be exploited in a software development tool, which may help software developers make good programming decisions. If the tool finds a pattern candidate, it may notify the programmer, who may decide to refactor the code to make the pattern explicit, possibly implementing the pattern with some predefined algorithmic skeletons. The tool may even help to perform the refactoring by applying automated program transformations on the source code under the supervision of the software developer. The refactoring technology to introduce `skel` skeletons is described e.g. in [4, 1].

The ParaPhrase Refactoring Tool for Erlang (a.k.a. PaRTE) offers exactly the above services [2]: software developers can find pattern candidates in existing Erlang code, and refactor them into applications of `skel` skeletons. Moreover, PaRTE is able to estimate the speedup that can be achieved by transforming a candidate to a skeleton-based parallel implementation, and hence prioritize candidates, in order to propose the most promising ones to its user.

The main contribution of this paper is the technique of d & c candidate discovery, implemented in a static analysis framework for Erlang (as part of PaRTE). Additionally, the paper characterizes typical occurrences of the pattern using small examples of well-known algorithms, and proves the applicability of the technique on real-world source code bodies. The presentation of d & c pattern discovery is concretized here for a nice and relatively simple language, Erlang, but the technique can be generalized to other languages, or even paradigms.

Although the identification of a d & c algorithm in some source code is an interesting problem by itself, its main benefit lies in its capability to facilitate the parallelization of the code. Once we know where to introduce parallelism, we can refactor the code either using a tool, or manually, to turn it into a parallel d & c. In this scenario, some additional side conditions must be verified, which ensure that the parallel components of the computation do not interfere with each other. Our ultimate focus on parallelization has an effect also on how we define d & c: "degenerated" cases of d & c algorithms, i.e. those that should not be turned into a parallel d & c implementation, will be exluded in our approach.

At the time of writing, support for the d & c pattern is limited in PaRTE. The tool is already able to identify d & c pattern candidates using the ideas presented in this paper. However, the introduction of the skeleton-based implementation of the pattern is to be carried out manually, because the completely automated transformations are not available yet.

The rest of the paper is structured as follows. Section 2 elaborates on the concept of pattern discovery. Section 3 points out the characteristics of d & c algorithms using a sequence of examples. Section 4 reveals the (Erlang-specific) static analyses required to discover d & c candidates. Section 5 describes the evaluation of the presented method. Section 6 discusses related work, and Section 7 concludes the paper.

2 PATTERN DISCOVERY

Pattern discovery is the act of finding pattern candidates in some source code by applying static program analyses. Albeit its applicability is broader, we focus on discovering *parallel* pattern candidates in general, and, in this paper, the d & c pattern in particular. Pattern candidates are typically characterized with a *syntactic description* and a set of properties that can be verified with some *semantic analyses*. Therefore, discovery starts with locating certain program structures (e.g. list comprehensions, function compositions, `receive`-expressions in Erlang), and then it filters out irrelevant ones using a combination of different standard analyses (such as control-flow, data-flow and data-dependence analyses), and specific ones (such as side-effect analysis). Finally, a ranking of candidates found can be computed, if a measure of "candidate goodness" is available. For example, in PaRTE a special technique to estimate parallel speedup is used to rank found parallel pattern candidates. In this paper, however, we will not consider this final step, we will focus on the former two.

PaRTE is based upon RefactorErl [3], a static program analysis and transformation framework for Erlang. RefactorErl implements the standard control-flow and data-flow analyses in an Erlang-specific way, taking into account the specificities of a concurrent, dynamically typed, impure functional language (see Section 4.1). RefactorErl also provides a rich representation of Erlang programs on which transformations, e.g. refactorings, are easy to implement.

In a language like Erlang, the main sources of uncertainty in static analyses are higher-order functions and the message-passing style inter-process communication. In the presence of such constructs, higher-order analyses [23] may be necessary to increase the precision of analysis results. However, for our purposes, first-order analyses will be sufficient in most of the cases. Firstly, the code fragments that should be parallelized are likely to contain only computations, and no inter-process communication (such as Erlang/OTP behaviours[1], callbacks, etc.). Secondly, higher-order functions and lambdas are less frequently written, or even used, in Erlang than in other functional languages; for instance, list comprehensions appear in code 5 to 10 times more often than the higher-order list processing functions of the Erlang/OTP `lists` module [20].

It is important to understand the ultimate goal of pattern discovery. It will not be built into a compiler in order to form the basis of a completely automatic program transformation (compiler optimization). In contrast, pattern discovery will be built into an integrated development environment (IDE): a software development tool to help its users make good programming decisions. The user of the tool, a software developer, should make those decisions; pattern discovery merely collects information that can be used – or discarded – by the software developer. False positives and false negatives are therefore tolerable. It is clear that in order to be useful, pattern discovery should be as precise as possible, but in an interactive tool there is always a trade-off between preciseness and response time.

The interactive nature of pattern discovery opens up further possibilities. Software developers may pass options to the discovery tool which can be used to filter the search results, and they may re-run the analyses multiple times, with different options, until satisfactory results are obtained. For instance, if the software developer believes that the discovery returns too many false positives for a given Erlang module, he or she can increase the order of the analyses for that particular module, and allow the tool a bit more time – a few minutes, or even hours – to work.

2.1 Refactoring a Pattern Candidate

One possible option which can be passed to pattern discovery would exclude candidates that cannot be automatically refactored by the tool. Assume that a software developer only wants to find those candidates that can be trivially transformed into an application of a parallel skeleton library.

For example, consider the following expression, which may appear in the code of a parser for a programming language. It reads, tokenizes and parses a list of source code modules – which can be parallelized in a straightforward manner.

```
[ parse(scan(read(Module))) || Module <- Modules ]
```

[1] OTP – a bit misleadingly – stands for Open Telecom Platform, which is “simultaneously a framework, a set of libraries, and a methodology for structuring applications” [11] for Erlang.

This candidate, a list comprehension with a function composition in the head, can be recognized as a pipeline pattern, and easily parallelized using a parallel skeleton library. A pipeline applies n transformations on each element of an input list, where n is the number of processes assigned to this task. Each process takes an item from the previous process in the pipeline, applies its transformation ("stage"), and passes the resulting item on to the next process in the pipeline. This is a parallel programming pattern, since each of the processes can work on different items at the same time. The list comprehension can be transformed into an application of the `skel` skeleton library in the following way.

```
STAGE1   = { seq,  fun read/1                 },
STAGE2   = { seq,  fun scan/1                 },
STAGE3   = { seq,  fun parse/1                },
PIPELINE = { pipe, [STAGE1, STAGE2, STAGE3] },
skel:do( [ PIPELINE ], Modules )
```

Skeletons in `skel` are expressed as tuples tagged with specific atoms, such as `seq` and `pipe`. The stages of the pipeline, the three functions, are "sequential components" denoted with `seq` – such sequential components are the basic building blocks of any skeletal descriptions. The skeleton is executed by calling the `skel:do/2` function on the skeleton description and the input list, here `Modules`.

In order to further increase parallelism, one could wrap the pipeline in a task farm, and process e.g. 5 instances of the pipeline in parallel.

```
skel:do( [ { farm, [ PIPELINE ], 5 } ], Modules )
```

Note that these transformations are really easy to carry out by a tool. However, many pattern candidates are more hidden, or tangled with other code fragments. Certain candidates can be automatically transformed at once, other candidates need to be reshaped first with a sequence of small refactorings. We presented a methodology of this programmer guided, semi-automatic transformation technique in [1]. In the most complicated cases, manual refactoring of the code might be necessary.

2.2 Examples of Parallel Patterns

The previous section gave examples of two well-known parallel patterns: the pipeline and the task farm (written in `skel` as `pipe` and `farm`, respectively). Parallel pattern libraries in general, and `skel` in particular, offer some further patterns as well.

Pipeline is used when a function composition is applied on a (possibly long) sequence of data. The functions that make up the composition are deployed in different processes, and hence form the stages of the pipeline.

```
{ pipe, [ Stage | Stages ] }
```

Task farm is suitable to perform an operation on a large data set using a number of processes. Each process takes an element of the input data set, applies the operation on it, puts the result in the output data set, and recurses until the input is empty. If the input data set is a sequence, care must be taken to produce the output in the same order. A task farm can also operate on a stream of inputs.

```
{ farm, Task, NumOfWorkers }
```

Parallel map is also applied on a large data set. A process decomposes the data set into smaller chunks, further processes perform the necessary operation on these chunks, and the last process collects results and recomposes the output data set.

```
{ map, Task, Decomp, Recomp }
```

Parallel d & c is the parallelized execution of a divide-and-conquer algorithm. A divide-and-conquer algorithm is to be applied on a (large) problem, which can be decomposed (divided) into smaller ones. This decomposition can be recursive, until a small-enough subproblem (base case) is reached. An operation (base function) is applied on the base cases, and the results are recombined (following the same recursive structure as for decomposition).

It is important to note that a subproblem can be processed in parallel with the processing of other (e.g. sibling) subproblems, where processing means either applying the base function, or recursively calling the divide-and-conquer operation. Hence we have a parallel pattern of the following structure.

```
{ dc, IsBase, BaseFun, Divide, Combine }
```

Note that the d & c pattern can be expressed as a recursive parallel map, as illustrated by the following pseudo-code (based on [19]).

```
dc ( IsBase , BaseFun , Divide , Combine ) ->
 fun ( Data ) ->
  case IsBase ( Data ) of
   true  -> BaseFun ( Data ) ;
   false -> (parmap( dc (IsBase,BaseFun,Divide,Combine),
                     Divide,
                     Combine
                   )
             )(Data)
  end
 end.
```

Practical implementations of the d & c parallel pattern may provide ways to bound parallelism. For example, an upper bound of created processes could be given,

```
{ dc, IsBase, BaseFun, Divide, Combine, MaxProcesses }
```

or subproblems satisfying a given property (e.g. subproblems of a given size) could be processed sequentially (i.e. with a sequential divide-and-conquer operation) rather than recursively with the parallel divide-and-conquer operation.

```
SEQ_DC = { seq_dc, IsBase, BaseFun, Divide, Combine },
PAR_DC = { dc, IsSeq, SEQ_DC, Divide, Combine }
```

From all these patterns, we focus on the parallel d & c pattern in this paper. Moreover, we are interested here mainly on d & c pattern *discovery*.

3 CHARACTERIZATION OF D & C

Divide-and-conquer algorithms recursively divide a problem into subproblems, solve those subproblems independently, and combine the results. Mou and Hudak gave an algebraic model in [15], which characterizes a large class of d & c 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". The last constituent of a divacon is the "base function", which is applied on basic inputs. Without going into details on further requirements on divacons, we emphasize that f is applied multiple times on independent inputs within its own definition, as implied by "map f". 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 recognize them as other patterns.

Let us remember the essence of the above structural description, but now we should change the viewpoint. Rather than considering the algebraic characterization, which is a canonical form of the pattern, one could examine a range of interesting syntactic occurrences of the pattern, which can appear quite naturally, or maybe rather unnaturally, in a functional program. We will go through some code examples, and see what we can learn from them – in order to develop a pattern discovery technique that works well enough in practice.

Quicksort. The most obvious, syntactically recognizable, form of a d & c algorithm is when we find multiple recursive calls in the body of a function.

```
1 qs(List)->
2   case List of
3     [] -> [];
4     [H | T] ->
5         {SubList1, SubList2} =
6                 lists:partition(fun(X)-> X < H end, T),
7         qs(SubList1) ++ [H] ++ qs(SubList2)
8   end.
```

The functional-style quicksort algorithm divides a list to be sorted into two sublists based on a pivot element (line 6), recursively sorts both lists, and concatenates the results (line 7).

Mergesort. In its most straightforward form, mergesort is written very similarly to quicksort – the main difference between the two is that in quicksort the divide-phase is the computationally more expensive phase, while in mergesort it is the combine-phase. Another, maybe less natural phrasing of mergesort shows that a case with two recursive calls is just a special case of the `map`-form.

```
ms(L) ->
  case L of
    [] -> [];
    [H] -> [H];
    [H | T] ->
      {SubList1, SubList2}=
        lists:split((length(L) div 2), L),
      [L1, L2] =
        lists:map(fun ms/1, [SubList1, Sublist2]),
      merge(L1, L2)
  end.

merge([], L2)-> L2;
merge(L1, []) -> L1;
merge([H1|T1], [H2|T2] = L2) when H1 < H2->
    [H1 | merge(T1, L2)];
merge([H1|T1] = L1, [H2|T2]) ->
    [H2 | merge(L1, T2)].
```

What happens, if we extract the recursive calls to `ms/1` and the call to `merge/2` into a separate function?

```
ms(L) ->
  case L of
    [] -> [];
    [H] -> [H];
    [H | T] ->
      {SubList1, SubList2}=
        lists:split((length(L) div 2), L),
      sort_and_merge(SubList1, SubList2)
  end.

sort_and_merge(L1,L2) ->
  merge( ms(L1), ms(L2) ).
```

When pattern discovery tries to identify recursive calls, it must take into account that recursion may happen indirectly, through other functions, as well.

Karatsuba. Our next d & c example is the well-known fast multiplication method for large integers. The integer numbers are represented as bit-strings. When a problem is divided into subproblems, smaller bit-strings are constructed (half the size of the original bit-strings). This example demonstrates a case when the original problem is divided into more than two subproblems: the `karatsuba/2` function calls itself three times in its body (lines 12–14). Moreover, the problem is represented with two, and not only one formal argument.

For brevity, we skip the definition of the `add/2`, `sub/2` and `shift/2` functions, which add, subtract and left-shift integers represented as bit-strings.

```
karatsuba(Num1, Num2) ->
  S1 = bit_size(Num1),
  S2 = bit_size(Num2),
  case {Num1, Num2} of
    {<<0:1>>, _} -> <<0:S2>>;
    {_, <<0:1>>} -> <<0:S1>>;
    {<<1:1>>, _} -> Num2;
    {_, <<1:1>>} -> Num1;
    _            ->
      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.
```

Radix sort. The common property of the before mentioned algorithms is that all of them contain multiple, but a small number of, recursive calls. However, a function can call itself many times, and this may not be conveniently expressed with a sequence of recursive calls in the code of the function – the canonical `map`-form discussed above seems more appropriate. The following `sort/2` function definition nicely exhibits this canonical form. Note that `sort/2` does not call itself directly, but rather through the higher-order `lists:map/2` function. This kind of implicit recursion should be also handled properly by the pattern discovery analyses. One way to achieve this is to build special knowledge about the functions of the `lists` module into the analyses.

```
sort(  [],    _   ) ->  [];
sort( [V],    _   ) -> [V];
sort(List, Level) ->
  Buckets = divide(List,Level),
  SortedLists = lists:map( fun(B) -> sort(B,Level+1) end,
                           Buckets ),
  lists:append(SortedLists).
```

Erlang programmers prefer list comprehensions to `lists:map/2`. The third clause of `sort/2` can be written considerably shorter with this language construct. A recursive call in the head of a list comprehension is the sign of a d & c function with very high probability.

```
sort(List, Level) ->
  lists:append([sort(B,Level+1) || B<-divide(List,Level)]).
```

Minimax. The depth-limited minimax algorithm is our next example. The presented implementation uses two mutually recursive functions, where both functions call the other one multiple times. Moreover, it can be the case that the number of children is not fixed for all nodes, and hence the functions call each other in varying number of times at different levels of the algorithm. Note that both of the mutually recursive functions can be regarded as d & c functions.

```
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.
```

3.1 Non-Trivial Occurrences of the Pattern

So far we have seen some natural program code structures, which implement d & c algorithms. Our real-life examples revealed some classes of d & c pattern candidates. We have learnt that a function may call itself explicitly multiple times within its own body, within a called function, or as part of a mutually recursive set of functions. The multiple recursive calls may appear lexically in the code, but they may take place due to an iterative structure, such as a list comprehension, or to a special higher-order function, such as `lists:map/2` as well.

However, pattern discovery should be able to cope with trickier examples, too. In some real-world code the programmer may write a function, which has the same recursion structure as `lists:map/2`, but maybe tangled with some other functionality. In general, it is possible to rewrite these functions with `map`, and hence make them more elegant, but pattern discovery should be able to find d & c functions in inelegant code as well.

```
sort(List, Level) ->
    lists:append(conquer(divide(List,Level),Level)).

conquer([],Level) ->
    [];
conquer([B|Bs], Level) ->
    [sort(B,Level+1) | conquer(Bs, Level)].
```

In [1] we have defined an analysis to discover "map-like functions" – those that basically work as `lists:map/2`, just like the above `conquer/2` function. As we have already noted, discovery of d & c candidates must investigate `lists:map/2` calls, but it must be able to recognize map-like functions as well.

The next complication arises when the `mapping` of the recursive calls and the combine phase get tangled into a "fold-like function", like, for instance, in the next code fragment. If pattern discovery can identify fold-like functions, this knowledge can be exploited in d & c discovery as well.

```
sort(List, Level) ->
    conquer(divide(List,Level),Level).

conquer([],Level) ->
    [];
conquer([B|Bs], Level) ->
    sort(B,Level+1) ++ conquer(Bs, Level).
```

One can construct even more contrived d & c functions, based on the ideas from the following example.

```
1  x(P) ->
2    ...
3    r(fun x/1, partition(P))
4    ...
5
6  r(F,Q) ->
7    A = some_part_of(Q),
8    B = some_other_part_of(Q),
9    ...
10   C = F(A),     % A does not depend on D
11   D = r(F,B),   % B does not depend on C
12   ...
```

Function `x/1` calls `r/2` (line 3); `r/2` is recursive (line 11), and contains a call to `x/1` (line 10). Now both `x/1` and `r/2` are d & c candidates (if some data independence side-conditions hold). Here `r/2` is responsible for an iterative call of `x/1`, and can be regarded as a generalization of map-like functions.

3.2 Non-Trivial Occurrences of the Non-Pattern

Strictly speaking, every recursive function can be considered as d & c. The structure of the simplest recursive definitions is the following.

```
1  f(Problem) ->
2    case base_case(Problem) of
3      true  -> basic_function(Problem);
4      false -> SubProblem = divide(Problem),
5               SubSolved = f(SubProblem),
6               combine(SubSolved)
7    end.
```

This corresponds to the characterization of d & c definitions, especially if line 5 is replaced with the equivalent

```
              [SubSolved] = lists:map(fun f/1, [SubProblem]),
```

line. However, we should regard similar definitions as degenerated cases of d & c. In contrast, we want d & c discovery to focus only on the really profitable candidates, ones that can benefit from pattern-based parallelization. Moreover, we expect d & c discovery to choose the best pattern for a candidate. If a function is map-like, pattern discovery should propose a task farm or a map pattern, and if it is fold-like, a reduce pattern is completely suitable – although both map-like and fold-like functions can be considered as special cases of d & c (indeed, [15] uses imbalanced reduce as an example of divacon functions).

```
maplike(List) ->
  case isempty(List) of
   true  -> [];
   false ->
      [SubList] = [tail(List)],
      [SubSolution] = lists:map(fun maplike/1, [SubList]),
      [someunaryoperation(head(List)) | head([SubSolution])
  end.

foldlike(List) ->
  case isempty(List) of
   true  -> defaultvalue;
   false ->
     [SubList] = [tail(List)],
     [SubSolution] = lists:map( fun foldlike/1, [SubList]),
     somebinaryoperation(head(List), head([SubSolution]))
  end.
```

Intuitively, pattern discovery should identify a function as d & c candidate, if it calls itself more than once during the execution of a single instance of its body – but we shall make this precise in Section 4.

Another important aspect in the analysis of d & c candidates is how we deal with execution paths. If a function calls itself on different execution paths, as in the following definition of `binsearch/4`, we should not consider it as d & c.

```
binsearch(Array,Pattern) ->
    binsearch(Array,0,array:size(Array)-1,Pattern).
binsearch(Array,Lower,Upper,Pattern) when Lower =< Upper ->
    H = (Lower+Upper) div 2,
    Val = array:get(H,Array),
    if
      Val < Pattern -> binsearch(Array,H+1,Upper,Pattern);
      Val > Pattern -> binsearch(Array,Lower,H-1,Pattern);
      true          -> true
    end;
binsearch(_,_,_,_) -> false.
```

Moreover, depending on the capabilities of the control flow analysis, d & c discovery may be able to tell the difference between the following two definitions:

a strangely written `binsearch/4` function (which is non-d & c) and the quicksort on non-empty lists (which is).

```
binsearch(Array,Lower,Upper,Pattern) when Lower =< Upper ->
  H = (Lower+Upper) div 2,
  Val = array:get(H,Array),
  (Val == Pattern)
   orelse
  (Val<Pattern andalso binsearch(Array,H+1,Upper,Pattern))
   orelse
  (Val>Pattern andalso binsearch(Array,Lower,H-1,Pattern));
binsearch(_,_,_,_) -> false.

qs( [H|T] ) ->
    {List1, List2} = lists:partition(fun(X)-> X<H end, T),
    Left  = if length(List1) > 1 -> qs(List1);
               true              -> List1
            end,
    Right = if length(List2) > 1 -> qs(List2);
               true              -> List2
            end,
    Left ++ [H] ++ Right
  end.
```

In our final example, we reimplement the `qs/1` function again. Note that this new implementation can be slightly faster, because `qs/1` is now defined in terms of a tail-recursive helper function, `qs/2`.

```
qs(List) -> lists:reverse(qs([],[List])).

qs(Result, [])              ->
        Result;
qs(Result, [[] | Lists])    ->
        qs(Result,Lists);
qs(Result, [[H] | Lists])   ->
        qs([H|Result],Lists);
qs(Result, [[H|T] | Lists]) ->
        {SubList1, SubList2} =
             lists:partition(fun(X)-> X < H end, T),
        qs(Result,[SubList1, [H], SubList2 | Lists]).
```

Theoretically, this is the same computation: the two recursive calls are simply replaced with an accumulator in the second argument of `qs/2`, which emulates the call stack. However, we shall not consider this definition d & c anymore. The recursion structure is completely changed: the `qs/2` function does not call itself "iteratively", as `qs/1` did in the original implementation. This kind of semantic equivalence is out of the scope and the capabilities of our d & c candidate discovery.

4 CANDIDATE DISCOVERY FOR D & C

In order to find d & c candidates, function definitions in the program source code shall be analysed. This section describes the required analyses: some of them are standard, general analyses, such as the construction of a control-flow graph, others are specific to d & c discovery. The description assumes that an abstract syntax tree (AST) is already available for the program text to be analysed. In the context of the Erlang language, this AST contains "forms" (such as `module` declarations, `import` and `export` clauses, function definitions, etc.), function and expression clauses (e.g. `case` and `receive` clauses), and expressions.

Section 4.1 summarizes how standard analyses map to a functional language like Erlang. Then, Section 4.2 presents rules for the identification of d & c candidates. These rules rely on the results of the standard analyses. Since these rules may be computationally expensive, in practice an approximation of these rules may be very useful. This idea is elaborated in Section 4.3.

4.1 Standard Static Analyses Customized for Erlang

Now we present a brief overview of some widely applied analyses, and show how these can be interpreted for Erlang programs. More details on these analyses, including a formal discussion, can be found in e.g. [28].

The "zeroth-order" analyses can be performed on an AST. It is well-known that the results of one analysis can be used to refine the input for another, which in return will provide more precise analysis results. Therefore, one can execute an analysis sequence repeatidly – theoretically, until a fixed point is reached. This way higher order analysis results are obtained. Since the iterative execution of all analyses is definitely expensive for large programs, in practice first-order analyses are already considered good enough. However, we are not restricted to lower order analyses. As explained in Section 2, we envision an interactive software analysis tool, in which the order of the analysis can be a customizable parameter, and the user can analyse "seemingly interesting" parts of the source code more deeply and precisely.

4.1.1 Control-Flow Analysis

In a functional language, the execution of a program is the evaluation of its expressions. The control-flow analysis should therefore discover the evaluation order of expressions. In a non-lazy language, like Erlang, this evaluation order can be computed more easily that in a lazy one.

The control-flow analysis starts from an entry point of a program. In Erlang, this is a call of an arbitrary exported function: a function that is visible from outside of its containing module.

Based on the above observations, the *inter-procedural control-flow graph* (CFG) of an Erlang program from an exported function f is denoted as $G_{CF}(f)$. This is a directed graph with labeled edges. Its nodes are the nodes in the AST of the program, plus some auxiliary nodes. Moreover, its edges correspond to the evaluation order of expressions. Edge labels are conditions, as seen below.

For each function g which is called during the evaluation of the main entry point (i.e. f), auxiliary graph nodes are created, with the following intended meaning.

- start_g represents the start of the evaluation of g,
- end_g represents the end of the evaluation of g,
- call_g^c represents the calling of g at a call site c,
- ret_g^c represents the returning from g at a call site c.

$G_{CF}(f)$ is constructed as a union of the intra-procedural control-flow graphs of the called functions, connected through the above auxiliary graph nodes. Roughly, an intra-procedural control-flow graph is constructed by visiting the AST of its body expressions in post-order, respecting the natural "subexpressions first" evaluation order. At branching expressions the control-flow graph contains branches as well, and the appropriate conditions are added as edge labels. (All other edge labels are considered *true*.) The intra-procedural control-flow graph of a function g contains the auxiliary nodes start_g and end_g: there is an edge from start_g to the first pattern of the first function clause, and there are edges from each return expression of g to end_g. (If we regard the intra-procedural CFG as a partial order over the constituting expressions, start_g is the bottom, and end_g is the top value of the partial order.)

If a call to a function g occurs in any of the intra-procedural CFGs (including that of g), the graph node c representing the call is replaced with two new nodes: call_g^c and ret_g^c. All incoming edges (a, c) (for some other node a) are replaced with edges (a, call_g^c), and all outgoing edges (c, a) are replaced with edges (ret_g^c, a). Moreover, the edges $(\mathit{call}_g^c, \mathit{start}_g)$ and $(\mathit{end}_g, \mathit{ret}_g^c)$ are also present in $G_{CF}(f)$.

There is one more important concept related to the control-flow graph, namely execution paths (EP). An execution path is a path in $G_{CF}(f)$, which may visit an edge multiple times. A finite execution path terminates with the end_f node, but infinite execution paths are also possible. Given a $G_{CF}(f)$, the set of execution paths starting from a node v will be denoted by $\mathit{EP}(v)$.

4.1.2 Data-Flow Analysis

Similarly to the inter-procedural control-flow graph, the *data-flow graph* G_{DF} of an Erlang program can be defined as a directed graph with labeled edges. Again, the nodes are the nodes of the AST – representing the (sub)expressions of the analyzed program.

The graph edges describe the flow of data. An edge from u to v represents the fact that there may be an execution of the program where the value of the expression u flows into expression v. Labels on the edges characterize the different types of data flow.

- If the value of u provides the value for v, the *flow* label is used. For example, the actual parameter of a function *flow*s into the formal parameter, or the right-hand side of a match expression *flow*s into the left-hand side expression.

$$\{A, B\} = \{X + Y, Y\}$$

  In this expression $\{X + Y, Y\}$ *flow*s into $\{A, B\}$.

- If v is a tuple, list, etc. expression, and u provides the value of a substructure (i.e. an element of a tuple/list, the tail of a list, etc.), the label is *construct* – more specifically, it is $\textit{construct}_i$, $\textit{construct}_{tail}$, etc. For the match expression above, for example, there exists a $\textit{construct}_1$ edge from $X+Y$ to $\{X+Y, Y\}$.
- If u is a tuple, list, etc. expression, and v extracts a component, then the label *select* is used, e.g. $\textit{select}_i$, $\textit{select}_{tail}$, etc. For the match expression above, for example, there exists a $\textit{select}_1$ edge from $\{A, B\}$ to A.
- Other data flow dependences between expressions are labeled with *dep*, like the edge from X to $X+Y$ in the example above.

From the *flow*, *construct* and *select* edges of G_{DF}, the *data-flow reaching* relation can be computed. This data-flow reaching computation pairs *construct* and *select* edges. For example, in the match expression above the expression A is reachable from $X+Y$, because of the $\textit{construct}_1$–*flow*–$\textit{select}_1$ path in the G_{DF}.

4.1.3 Derived Information

Function call graph. Being a functional language, Erlang supports higher-order functions. Therefore, the *function call* analysis depends on the result of the data-flow analysis (and vice-versa). As mentioned above, an approximation can be achieved by applying a fixed (but customizable) number of iterations. The result of the function call analysis is the function call graph, which will be denoted as G_{FC}.

Dependence graph. The control dependence graph (CDG) can be computed from the control flow graph as described in e.g. [16, 17]. Then the CDG is extended with the *dep*-edges of the G_{DF} and the data-flow reaching relation to form $G_D(f)$, the *dependence graph* of the program with entry point f. When a *dep*-edge or a data-flow reaching edge is to be added to the $G_D(f)$ from, or to, a function call expression, then outgoing edges will start from the respective *ret* node of the CFG, and incoming edges will arrive at the *call* node.

A path from u to v in $G_D(f)$ will be denoted by $u \overset{\textbf{dep}}{\rightsquigarrow} v$.

4.2 Identifying Rules for D & C

The divide and conquer pattern describes a computation where a problem is recursively *divided* into sub-problems (until a given condition), and after solving the sub-problems the sub-solutions are *combined* to produce the final solution.

As we have seen in Section 3, there are many syntactic forms which express a recursive d & c-like function. From semantical point of view, a canonical form can be given as follows.

```
cdc(Problem) ->
  case isbase(Problem) of
    true -> solve(Problem);
    false ->
      SubProblems = divide(Problem),
      SubSolutions  = lists:map(fun cdc/1, SubProblems),
      combine(SubSolutions)
  end.
```

Therefore, we are looking for a function, which

- has at least one parameter, defining the problem to solve; and
- has, or triggers, multiple recursive calls in its body;

furthermore, these recursive calls satisfy the following properties:

- their actual parameters do not depend on the result of (other) recursive calls;
- their actual parameters depend on the formal parameters of the function definition,
- the return value of the function depends on the result of the recursive calls.

For pure computations, the above rules are sufficient. However, Erlang functions are often impure [18]. Therefore, the concept of component hygiene should be introduced [1]. Here the recursive calls must be possible to run in parallel, and hence their side effects, if there are any, must not be conflicting. For simplicity, we disregard this issue in this paper. Information on hygiene analysis can be found in [1].

To formally define the rules for d & c identification, we consider a function f, as well as the control-flow graph $G_{CF}(f)$ and dependence graph $G_D(f)$ built from f as a starting point. If the following conditions hold, f will be identified as a d & c candidate.

1. f must be recursive: it has an execution path which contains a call to itself;

$$\exists p \in EP(start_f),\ \exists c \text{ such that } call_f^c \in p$$

2. f must have a base case: it has an execution path which does not contain a call to itself;

$$\exists p \in EP(start_f) \text{ such that } (\nexists c : call_f^c \in p) \wedge (end_f \in p)$$

3. f must have multiple recursive calls in its body, as described by the following three possibilities.

   - It may contain an execution path that contains at least two independent recursive calls[2]:

$$\exists c_1, c_2, \exists p \in EP(ret_f^{c_1}) \text{ such that } call_f^{c_2} \in p \wedge \forall a \in \mathrm{ARG}(c_2) :\ \neg(a \overset{\mathbf{dep}}{\rightsquigarrow} ret_f^{c_1})$$

[2] Obviously, there may be more than two recursive calls, but only the independent recursive calls result in exploitable parallelism.

where ARG is the set of nodes representing the arguments of a function call.

- It may have an execution path containing a list comprehension with head expression h, which calls f directly or indirectly, that is:

$$\exists p \in EP(h),\ \exists c \text{ such that } call_f^c \in p.$$

- It may (directly or indirectly) call a *farm candidate* (see below) function g, which in turn calls f in its every recursive execution paths.

$$\exists p \in EP(start_f), \exists c_1, \exists g \text{ recursive function such that } call_g^{c_1} \in p \wedge$$

$$\forall q \in EP(start_g) : (\exists c_2 :\ call_g^{c_2} \in q) \rightarrow (\exists c_3 :\ call_f^{c_3} \in q).$$

Farm candidates. The above rules refer to another important class of parallelizable functions: those functions, which can be turned into a parallel task farm. Such farm candidates operate on a collection (set, list or stream) of data by applying a computation on each data item independently of the others. A formal characterization of *map-like functions* has been published in [1]. According to this characterization, a function f is map-like, if it satisfies the following conditions. It has a list parameter L, and returns a list. The head of the returned list may depend on the head of L, but may not depend on the tail of L. Similarly, the tail of the returned list may not depend on the head of L: it should simply be the result of a recursive call to f on the tail of f. Moreover, f may have further parameters, but all these parameters must be passed to the recursive call unchanged.

Note that the standard *map* function in Erlang, `lists:map/2`, satisfies these conditions; it is indeed a map-like function.

The characterization of map-like functions can be generalized to cover tail-recursive implementations of the same behaviour. Another generalization can cover the case when the input is a stream of data items (generated or `receive`d concurrently to the processing of those items). These generalizations are described in [22]. Further generalizations can also be made, and hence even more farm candidates can be identified – this way the scope of the d & c identification analysis also extends.

The rules given in this section are based on the studies presented in Section 4, and are therefore suitable to identify the d & c pattern candidates, and discard the non-patterns, as explained in that section.

4.3 Efficient Approximation of D & C Identification

The above rules are all based on the concept of EP, the execution paths in the control-flow graph. The computation of all execution paths is extremely time-consuming, which can turn d & c analysis impractical for larger code bodies. Therefore less precise, but more efficiently computable conditions have to be applied for such cases. Execution paths are used by our analysis for finding certain function calls. One can substitute this analysis with a similar one working on the functional call graph, G_{FC}. This is clearly an approximation, since G_{FC} only records which functions call which other functions, and does not allow us differentiate calls on different execution paths. Let the edges of G_{FC} be denoted with the *funcall* label, and a path in G_{FC} with *funcall+* (which is an edge in the transitive closure of G_{FC}).

Our d & c discovery analyses G_{FC}, and searches for patterns expressing iterative evaluation of a function. Figure 1 shows such a call graph pattern. If a function f calls (directly or indirectly) a recursive function g (i.e., g calls itself directly or indirectly), and g calls f, then our analysis suspects that f is called in every recursive step, so it is called multiple times by g.

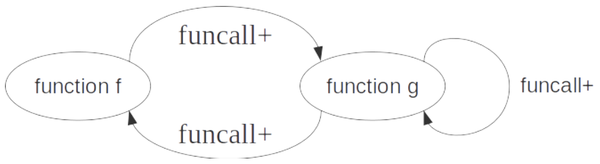

Figure 1. A fragment in a function call graph typical for d & c

After collecting such suspicious f-g pairs, a more precise analysis can be applied to determine whether f is really a d & c candidate.

5 APPLICABILITY OF D & C DISCOVERY

This section reports on our findings on the applicability of d & c pattern candidate discovery. The analyses presented in this paper have been implemented within the ParaPhrase Refactoring Tool for Erlang. In order to investigate the effectiveness of our approach, we have analysed some small examples, some use cases developed within the ParaPhrase project, and some open source projects as well. Our findings were reported in ParaPhrase project deliverable D6.6 [21]. Here we demonstrate two applications in which several d & c candidates were found.

We verified manually (some of) the pattern candidates found by the tool, to see whether they really possess the properties that qualify them for the identified pattern. Most candidates (list comprehensions and applications of predefined higher-order functions) are trivially appropriate, hence we checked them only by random sampling. Map-like functions and divide-and-conquer algorithms are more sophisticated to characterize and harder to find, and so we paid more attention to them. We checked the dozen map-like functions the tool found, but used only random sampling for the about one hundred divide-and-conquer candidates. Two interesting d & c candidates are presented below in Section 5.2.

5.1 Discovery Statistics

First of all, we have analysed the source code of the distributed database management system Mnesia [25], which is part of the standard Erlang/OTP library. The analyzed code body contains 1 693 function definitions in 31 files, and consists of 22 653 effective lines of code. We could find 57 d & c candidates (Table 1).

Candidate	**Number of Occurrences**	**Kind of Pattern**
various library calls	72	farm
various library calls	36	reduce
list comprehension	58	farm
map-like function	5	farm
d & c-like function	57	divide and conquer

Table 1. Mnesia

Then we have analyzed some components of the RefactorErl tool [26] as well. The analyzed `referl_core` component contains 1 534 function definitions in 53 files, and consists of 19 694 effective lines of code. We could find 31 d & c candidates (Table 2).

Candidate	**Number of Occurrences**	**Kind of Pattern**
various library calls	139	farm
various library calls	55	reduce
list comprehension	347	farm
map-like function	3	farm
d & c-like function	31	divide and conquer

Table 2. RefactorErl: referl_core

5.2 Examples of Interesting Candidates

During the validation of pattern candidate discovery, we have encountered really nice instances of map-like and divide-and-conquer definitions. Here we point out two demonstrative cases.

The first example (Figure 2) shows a beautiful instance of the divide-and-conquer pattern: the `refcore_callanal:listcons_length/2` function almost completely follows the "canonical form" of d & c. It operates on a list; it splits the list in the divide-phase using `lists:partition/2`, it applies (through `listcons_length/1`) itself iteratively with `lists:map/2` in the non-base case, and finally combines the results explicitly (with `lists:append/1`). The particularity of this example is that the d & c function is not directly recursive: it calls itself indirectly through another function.

```
listcons_length(ListExpr) ->
  listcons_length(ListExpr, ?Graph:data(ListExpr)).
...
listcons_length(N, #expr{}) ->
  Ns = ?Dataflow:?reach([N], [back], true),
  L1 = [N2 || N2 <- Ns, N2 /= N,
              ?Graph:class(N2) == expr],
  {L2, L3} = lists:partition(fun is_cons_expr/1, L1),
  if L2 == [] orelse L3 /= [] ->
       incalculable;
     true  ->
       lists:append(lists:map(fun listcons_length/1, L2))
  end;
...
```

Figure 2. Nice d & c in RefactorErl

The second example (Figure 3) is again from the code of RefactorErl, namely the `refcore_pp:realtoken_neighbour/3` function. At a first sight, this function is a simple recursive function calling itself in line 16. If we further investigate this function, we find another execution path that calls `realtoken_neighbour_/4`. This function is a recursive function that calls `realtoken_neighbour/3` in its every recursive execution path. This is exactly the pattern in the function call graph which can be detected by our faster d & c candidate discovery algorithm.

Our discovery analysis identifies `realtoken_neighbour/3` as a divide-and-conquer definition, because it calls `realtoken_neighbour_/4`, which iterates through the `Parents` list, and calls `realtoken_neighbour/3` in each step. Note that the identification of the base condition (and the corresponding base function) of this d & c candidate is not straightforward, since the condition is scattered over several case expression headers and patterns. Hence the automatic transformation of this definition into a d & c pattern is indeed a challenge.

6 RELATED AND FUTURE WORK

Various approaches have been published related to parallel pattern identification. Some of these methods use purely static information, but others monitor the dynamic behaviour of the system as well.

In [1] a formal definition of "map-like functions" is given in order to automatically discover list-based elementwise computations. The paper also describes a variety of program shaping transformations to refactor the map-like pattern candidates in a syntactic form that can be then transformed into an application of the *farm* skeleton.

In [2] the ParaPhrase Refactoring Tool for Erlang was introduced. This tool provides pattern discovery, candidate ranking based on performance measurements and estimates, as well as semi-automatic pattern introduction by refactorings. The cost models used by the tool were defined in [4]. Our presented d & c candidate discov-

```
realtoken_neighbour(Node, DirFun, DownFun) ->
  case lists:member(?Graph:class(Node),
                    [clause,expr,form,typexp,lex]) of
    false -> no;
    _ ->
      case ?Syn:parent(Node) of
        [] -> no;
        [{_,Parent}] ->
          case lists:dropwhile(
                      fun({_T,N}) -> N/=Node end,
                      DirFun(?Syn:children(Parent))
              ) of
            [{_,Node},{_,NextNode}|_] ->
              DownFun(NextNode);
            _ ->
              realtoken_neighbour(Parent, DirFun, DownFun)
          end;
        Parents ->
          realtoken_neighbour_(Parents, DownFun(Node),
                               DirFun, DownFun)
      end
  end.

% Implementation helper function for realtoken_neighbour/3
realtoken_neighbour_([], _FirstLeaf,_DirFun,_DownFun) ->
  no;
realtoken_neighbour_([{_,Parent}|Parents],
                     FirstLeaf, DirFun, DownFun) ->
  case realtoken_neighbour(Parent, DirFun, DownFun) of
    FirstLeaf ->
      realtoken_neighbour_(Parents, FirstLeaf,
                           DirFun, DownFun);
    NextLeaf ->
      NextLeaf
  end.
```

Figure 3. Really complex d & c in RefactorErl

ery method extends this framework – although candidate ranking based on speedup estimates is not implemented yet for d & c (pattern discovery reports currently all d & c candidates without evaluating their parallel speedup potential).

The `skel` library [24] provides a set of reusable algorithmic skeleton implementations for Erlang. In [19] some high-level pattern implementations are presented as an extension to `skel`, including d & c. As future work, we plan to implement d & c transformations as well. To achieve our goal, we can rely on this high-level skeleton library. Two useful features of this library are that we can limit the number of started parallel processes, and force sequential evaluation where beneficial. Although `skel` does not support distributed skeletons such as D-Clean [29], it is possible to extend it based on the Erlang concepts.

The Eden skeleton library [12] offers even more advanced d & c pattern implementations, for instance, the one based on the distributed expansion scheme. The three main properties of this implementation are: (1) a branching degree is given by a parameter representing the number of subproblems; (2) the process creation and allocation is controlled by a ticket list; and (3) every process keeps a subproblem for local evaluation. We are not aware of any pattern discovery tools for Eden, but it is definitely possible to implement one based on the approach of this paper. Moreover, it is also possible to re-implement Eden's sophisticated d & c patterns for Erlang.

Several other researches focus on efficient d & c implementations, including for example, Herrmann [9]. Note that the refactoring-centric approach of ParaPhrase (and hence PaRTE) fosters experimentation with the various implementations and parameter values, including thresholds controlling the parallel-sequential balance.

Although discovering d & c candidates is an interesting problem by itself, it becomes the most beneficial when a tool introduces parallelism (semi)automatically to the source code by replacing the sequential d & c with its parallel equivalent. In this case two problems have to be addressed: the efficiency of the parallel d & c (and hence the usefulness of parallelizetion) has to be investigated, and a transformation framework has to be capable of introducing the parallel d & c implementation.

In [6] an automated transformation framework was introduced to transform sequential d & c algorithms to parallel equivalents. The transformation uses a few annotations (which are required to be provided by the programmer) to identify the places where parallelism should be introduced.

The automatic transformation of d & c candidates motivated our research, but it is not covered in this paper. We set out this research topic for future work, based on the above mentioned papers.

Optimizing compilers are able to identify parallelizable code fragments, and also to parallelize them automatically. Of course, such transformations are, and should be, ultra-conservative, not allowing to change the semantics of the code. Our approach, on the contrary, can be more flexible and more effective, since we have a human in the loop: the final decision on parallelization is always made by our tool user. To mention but one parallelizing compiler, SkelML [13] is a parallel skeleton-based compiler for SML. It can automatically identify applications of certain higher-order functions as pattern candidates and transform them to applications of equivalent parallel skeletons. For instance, SkelML can transform *fold* function applications to application of d & c skeletons. However, the pattern discovery technique applied by SkelML is less generic than ours. Our tool is able identify various syntactic forms of pattern candidates which are not necessarily just applications of special higher order functions. Indeed, the strength of our tool is the analysis of recursive function calls.

Similar parallelization techniques have been developed for non-declarative languages as well. The tool AutoFutures [14], for example, uses static analysis of Java programs to discover source code fragments that have no data dependences, and can be candidates for parallelization.

The static analysis approach that we also follow can be replaced (or combined) with dynamic analyses as well. We can examine execution traces, like the tool [7] based on JavaSlicer, which uses dynamic dependence graphs to identify independent program paths in Java code, and recommends code fragments with a high potential for parallelization. We can also integrate a static analysis based approach into one using run-time monitoring of executions, which we plan for our future work.

7 CONCLUSION

This paper investigated the concept of divide-and-conquer pattern discovery, a static program analysis technique to automatically identify parallelizable code fragments. This analysis can cope with the many syntactically different occurrencies of d & c behaviour. Depending on how conservative the analysis is, the technique can be exploited in an optimizing compiler, or in an integrated software development environment, which supports refactoring. We focused on the latter use case. This allows our analysis to find d & c candidates which are too hard to transform automatically into a skeleton-based parallel pattern implementation – in an IDE the user may indeed be able to carry out the transformation manually, or semi-automatically. Since the final decisions are made by the user, this approach tolerates false positives, and hence the analysis can be parametrized (and the discovery tool be dynamically configured) by the level of conservativeness and that of aggressivity. This allows the user to make the necessary practical trade-offs between effectiveness and safety, and between effectiveness and latency.

Our analysis is built on top of control-flow, data-flow, reaching, function-call, and control-dependence analyses. Therefore, the presented d & c candidate discovery is applicable for a wide range of programming languages. However, we have worked out the details of the analysis for the Erlang language, and for this the paper summarized the Erlang-specific details of the CFG and DFG construction. We have implemented the analysis in a software development tool, the ParaPhrase Refactoring Tool for Erlang.

The numerous examples shown in the paper are also written in Erlang. As a final evaluation of the approach, we applied the analysis on real-world open-source code bases.

The lessons learned from the presented research are the following. Divide-and-conquer algorithms are often implemented using recursion: indeed, we have defined d & c as a function which calls itself recursively multiple times. Therefore, a static analysis of (direct and indirect) recursive calls is able to detect (many) occurrences of d & c. Such an analysis can be defined in the presence of an interprocedural control-flow and data-flow analysis, and hence it is language and paradigm independent. In an imperative language, for instance, a d & c function may include a loop construct with a body containing a recursive call. Of course, it is possible to implement a d & c algorithm without recursion, for instance using only iteration (loops in an imperative language) and a stack data structure – such an implementation will not be found by the proposed analysis.

The analysis of recursive calls can be defined using the concept of execution paths. It turned out that this analysis is rather time-consuming. However, a faster, though less precise, analysis based on the function call graph also performs very well in practice.

Acknowledgement

The research presented in this paper was supported by the Seventh Framework Programme (FP7) under the contract number 288570.

REFERENCES

[1] Bozó, I.—Fördős, V.—Horpácsi, D.—Horváth, Z.—Kozsik, T.—Kőszegi, J.—Tóth, M.: Refactorings to Enable Parallelization. Trends in Functional Programming, 15th International Symposium (TFP 2014). Springer International Publishing, Lecture Notes in Computer Science, Vol. 8843, 2015, pp. 104–121.

[2] Bozó, I.—Fördős, V.—Horváth, Z.—Tóth, M.—Horpácsi, D.—Kozsik, T.—Kőszegi, J.—Barwell, A.—Brown, C.—Hammond, K.: Discovering Parallel Pattern Candidates in Erlang. Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang, ACM, New York, NY, USA, 2014, pp. 13–23.

[3] Bozó, I.—Horpácsi, D.—Horváth, Z.—Kitlei, R.—Kőszegi, J.—Tejfel, M.—Tóth, M.: RefactorErl – Source Code Analysis and Refactoring in Erlang. Proceedings of the 12th Symposium on Programming Languages and Software Tools, 2011, pp. 138–148.

[4] Brown, C.—Danelutto, M.—Hammond, K.—Kilpatrick, P.—Elliott, A.: Cost-Directed Refactoring for Parallel Erlang Programs. International Journal of Parallel Programming, Vol. 42, 2014, No. 4, pp. 564–582.

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

[6] Freisleben, B.—Kielmann, T.: Automated Transformation of Sequential Divide-and-Conquer Algorithms into Parallel Programs. Computing and Informatics, Vol. 14, 1995, No. 6, pp. 579–596.

[7] Hammacher, M.—Streit, K.—Hack, S.—Zeller, A.: Profiling Java Programs for Parallelism. Proceedings of ICSE Workshop on Multicore Software Engineering (IWMSE '09), 2009, pp. 49–55.

[8] Hammond, K.—Aldinucci, M.—Brown, C.—Cesarini, F.—Danelutto, M.—González-Vélez, H.—Kilpatrick, P.—Keller, R.—Rossbory, M.—Shainer, G.: The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. Formal Methods for Components and Objects, Springer Berlin Heidelberg, Lecture Notes in Computer Science, Vol. 7542, 2013, pp. 218–236.

[9] Herrmann, C. A.: The Skeleton Based Parallelization of Divide and Conquer Recursions. Logos-Verlag, 2001. ISBN 9783897225565.

[10] Horpácsi, D.—Kőszegi, J.: Static Analysis of Function Calls in Erlang. e-Informatica Software Engineering Journal, Vol. 7, 2013, pp. 65–76.

[11] Logan, M.—Merritt, E.—Carlsson, R.: Erlang and OTP in Action. Manning Publications Co., 2010. ISBN 9781933988788.

[12] Loogen, R.: Eden – Parallel Functional Programming with Haskell. Central European Functional Programming School, Springer Berlin Heidelberg, Lecture Notes in Computer Science, Vol. 7241, 2012, pp. 142–206.

[13] Michaelson, G.—Ireland, A.—King, P.: Towards a Skeleton Based Parallelising Compiler for SML. Proceedings of 9th International Workshop on Implementation of Functional Languages, 1997, pp. 539–546.

[14] Molitorisz, K.—Schimmel, J.—Otto, F.: Automatic Parallelization Using Autofutures. Proceedings of 2012 International Conference on Multicore Software Engineering, Performance, and Tools, Springer Berlin Heidelberg, Lecture Notes in Computer Science, Vol. 7303, 2012, pp. 78–81.

[15] Mou, Z. G.—Hudak, P.: An Algebraic Model for Divide-and-Conquer and Its Parallelism. Journal of Supercomputing, Vol. 2, 1988, No. 3.

[16] Muchnick, S. S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, Inc., 1997.

[17] Nielson, F.—Nielson, H. R.—Hankin, C.: Principles of Program Analysis. Springer, 1999, corrected 2005.

[18] Pitidis, M.—Sagonas, K.: Purity in Erlang. Proceedings of 22nd International Conference on Implementation and Application of Functional Languages, Springer Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 6647, 2011, pp. 137–152.

[19] ParaPhrase Project WP2: Initial Implementation of Application-Specific Patterns. Technical report, University of Pisa, September 2013.

[20] ParaPhrase Project WP4: Specification of Pattern Candidate Refactoring Rules. Technical report, ELTE-Soft Nonprofit Ltd., July 2014.

[21] ParaPhrase Project WP6: Final Report on Experimental Evaluation. Technical report, University of Stuttgart, March 2015.

[22] ParaPhrase Project WP2: Final Pattern Discovery. Technical report, ELTE-Soft Nonprofit Ltd., March 2015.

[23] Shivers, O.: Control Flow Analysis in Scheme. Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation, 1988, pp. 164–174.

[24] Skel Tutorial. Available on: `http://chrisb.host.cs.st-andrews.ac.uk/skel-test-master/tutorial/bin/tutorial.html`, 2014.

[25] Source code of Mnesia. `https://github.com/erlang/otp/tree/maint/lib/mnesia/src`.

[26] Source code of RefactorErl. `http://plc.inf.elte.hu/erlang/dl/refactorerl-0.9.14.09.zip`.

[27] The ParaPhrase Project. `http://www.paraphrase-ict.eu`, 2014.

[28] Tóth, M.—Bozó, I.: Static Analysis of Complex Software Systems Implemented in Erlang. Central European Functional Programming School, Springer, Lecture Notes in Computer Science, Vol. 7241, 2012, pp. 440–498.

[29] Zsók, V.—Hernyák, Z.—Horváth, Z.: Designing Distributed Computational Skeletons in D-Clean and D-Box. Central European Functional Programming School, Springer, Lecture Notes in Computer Science, Vol. 4164, 2006, pp. 223–256.

Tamás Kozsik is Associate Professor at Eötvös Loránd University (Budapest, Hungary), where he is Vice Dean for projects and innovation at the Faculty of Informatics. He teaches programming paradigms and languages. His research is focused on language technologies including type systems, domain specific languages, source code analysis and transformations, parallel programming, and formal methods. Recently he was Principal Investigator in the Parallel Patterns for Adaptive Heterogeneous Multicore Systems (ParaPhrase) EU FP7 project.

Melinda Tóth works as a researcher at ELTE-Soft Nonprofit Ltd. (Budapest, Hungary), leading the ELTE-Ericsson Software Technology Lab. She is also Assistant Lecturer at Eötvös Loránd University, teaching distributed systems and Erlang/OTP technology. Melinda Tóth is the chief architect of RefactorErl, a static source code analysis and transformation system for Erlang. She co-chairs ACM SIGPLAN Erlang Workshop in 2015 and 2016.

István Bozó is a researcher at ELTE-Soft Nonprofit Ltd. (Budapest, Hungary), and Assistant Lecturer at Eötvös Loránd University. His main research topic is impact analysis of functional programs based on control-dependence graphs. He is working on static program analysis and refactoring in the RefactorErl and the ParaPhrase projects. He teaches functional programming as well as formal methods for distributed systems.

Zoltán Horváth is Project Manager at ELTE-Soft Nonprofit Ltd. (Budapest, Hungary), and Full Professor at Eötvös Loránd University, heading the Programming Languages and Compilers Department. He is also Dean of the Faculty of Informatics, and Director of the EIT Digital Budapest Associate Partner Group. He is teaching and researching functional programming and formal methods for distributed systems. He supervised numerous national and international projects, among others he was Principal Investigator in the Parallel Patterns for Adaptive Heterogeneous Multicore Systems (ParaPhrase) EU FP7 project.