Static analysis for divide-and-conquer pattern discovery

Tamás Kozsik, Melinda Tóth, István Bozó, Zoltán Horváth

Eötvös Loránd University and ELTE-Soft Nonprofit Ltd.
(Budapest, Hungary)

What is this talk about?

Given the source code of some program. . .

. . . find a routine . . .

. . . which implements a divide-and-conquer algorithm

What is this talk about?

Given the source code of some **Erlang** program. . .

. . . find a **function** . . .

. . . which implements a divide-and-conquer algorithm

Divide-and-Conquer algorithm?

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

merge( ... ) -> ...
```

Or more generally. . .

Mou & Hudak (1988):

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

```
mergesort( []  ) -> [];
mergesort( [H] ) -> [H];
mergesort(  L  ) ->
      {L1, L2} = lists:split( length(L) div 2, L ),
      [S1, S2] = lists:map( fun mergesort/1, [L1,L2] ),
      merge( S1, S2 ).
```

Why is it important?

Possibility for parallelisation

```
mergesort(  [] ) -> [];
mergesort( [H] ) -> [H];
mergesort(  L  ) ->
      {L1, L2} = lists:split( length(L) div 2, L ),
      spawn(?MODULE, mergesort_proc, [self(),L1]),
      spawn(?MODULE, mergesort_proc, [self(),L2]),
      receive
          S1 -> receive
                     S2 -> merge( S1, S2 )
                end
      end.

mergesort_proc(Pid,List) -> Pid ! mergesort(List).
```

Trivialized...

Motivation

If a tool finds divide-and-conquer functions

- Give parallelisation hints to programmers
- Tool supported refactoring
- ParaPhrase Refactoring Tool for Erlang (PaRTE)
- Pattern-based parallelism

Pattern-based parallelism

- Parallel patterns (task farm, pipeline, ... d&c ...)
- Algorithmic skeleton library (`skel` and `sk_hlp`)

```
mergesort(List) ->
    IsBase   = fun(L) -> length(L) < 2 end,
    BaseCase = fun(L) -> L end,
    Divider  =
        fun(L) ->
            {L1, L2} = lists:split( length(L) div 2, L ),
            [L1, L2]
        end,
    Combiner = fun([L1, L2]) -> merge(L1,L2) end,
    (sk_hlp:dc(IsBase,BaseCase,Divider,Combiner))(List).
```

D&C pattern candidate

- Mou & Hudak (1988) is not good enough
  - too restrictive
  - too general
- Our definition:

  function triggers multiple independent recursive calls

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

Indirectly recursive function

```
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.
```

Recursive call is in a recursive function

```
bucketsort(  [],    _   ) ->  [];
bucketsort( [V],    _   ) -> [V];
bucketsort(List, Level) ->
    conquer(divide(List,Level),Level).

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

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

Recursive call is in the head of a list comprehension

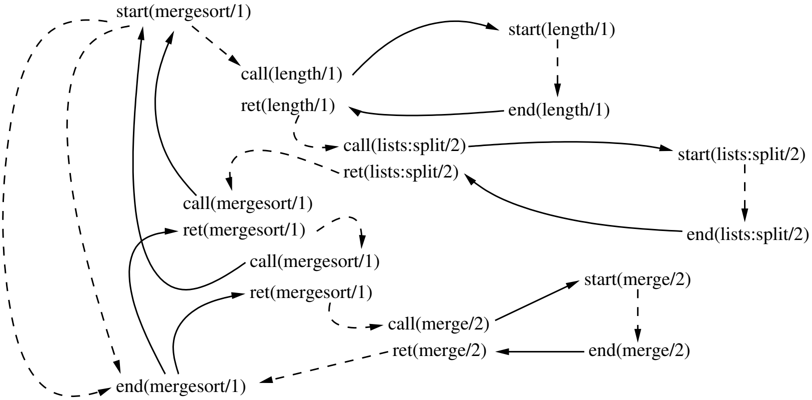
```
bucketsort(  [],    _  ) ->  [];
bucketsort( [V],    _  ) -> [V];
bucketsort(List, Level) ->
  lists:append(
     [bucketsort(B,Level+1) || B<-divide(List,Level)]
  ).
```

How to find D&C pattern candidates?

Static source code analysis

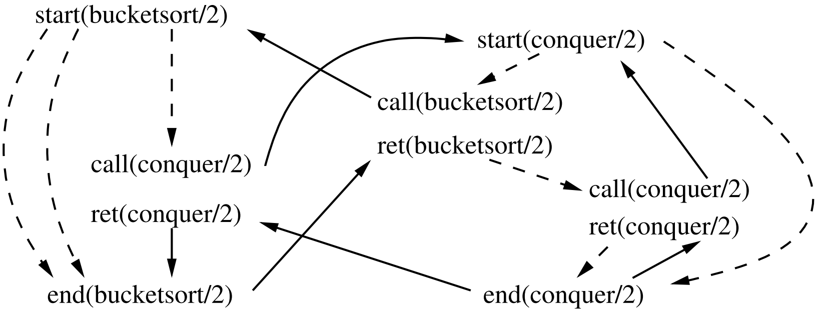
- Control flow, Data flow, Function call, Dependency, Reaching
  - Decorated AST (nodes are expressions)
  - Inter-procedural analyses
  - Higher-order analyses require multiple passes
- **Analysis of recursive calls**
- Extras for parallelism (e.g. side effects)

Execution paths




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

Execution paths




```
bucketsort(  [],    _   ) ->  [];
bucketsort( [V],    _   ) -> [V];
bucketsort(List, Level) ->
    conquer(divide(List,Level),Level).

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

Formal rules (1)

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

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

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

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

- f must have multiple recursive calls in its body, for example
  - it may contain an execution path that contains at least two independent recursive calls

$$\exists c_1, c_2, \exists p \in EP(ret_f^{c_1}) \text{ such that}$$

$$call_f^{c_2} \in p \ \wedge\ \forall a \in \mathrm{ARG}(c_2) :\ \neg\left(a \overset{\textbf{dep}}{\rightsquigarrow} ret_f^{c_1}\right)$$

Formal rules (2)

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

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

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

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

- f must have multiple recursive calls in its body, for example
  - it may have an execution path containing a list comprehension with head expression h, which calls f directly or indirectly, that is:

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

Formal rules (3)

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

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

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

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

- f must have multiple recursive calls in its body, for example
  - it may (directly or indirectly) call a recursive function g, which in turn calls f in its every recursive execution path.

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

$$call_g^{c_1} \in p \wedge \forall q \in EP(start_g) : \left(\exists c_2 :\ call_g^{c_2} \in q\right) \rightarrow \left(\exists c_3 :\ call_f^{c_3} \in q\right)$$

Fast approximation rule

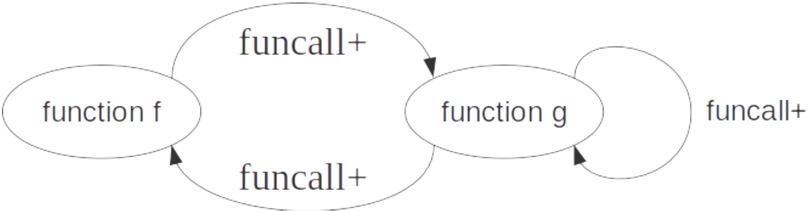


Figure 1: Function Call Graph fragment

Results in “artificial” and real-world code

- Case study with quicksort, mergesort, bucketsort, minimax, karatsuba etc. variants
- Mnesia:
  - distributed database management system
  - 1,693 function definitions in 31 files, and consists of 22,653 ELOC
  - 57 d&c candidates
- RefactorErl
  - static program analysis and transformation systems
  - `referl_core` component contains 1,534 function definitions in 53 files, and consists of 19,694 ELOC.
  - 31 d&c candidates

Example candidate (RefactorErl)

```
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.
```

Example candidate (RefactorErl)

```
realtoken_neighbour(Node, DirFun, DownFun) ->
  ...
  realtoken_neighbour_(Parents, DownFun(Node), DirFun, DownFun)
  ...

% 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.
```

Summary

- PaRTE can find many d&c pattern candidates
- Hints programmers where to introduce parallelism
- Analyses recursion
  - Static source code analysis
  - "A function that triggers multiple independent recursive calls"
  - Execution paths: slow – FunCall graph: fast approximation
- Extra analysis for side effets
- Approach generalizable to other languages / paradigms
  - Interprocedural CFG and DFG, FunCall, Dependency