LambdaDays, Krakow, 2017

Free the Conqueror!

Tamás Kozsik Melinda Tóth István Bozó
Eötvös Loránd University

LambdaDays, Krakow, 2017

Free the Conqueror!

Refactoring divide-and-conquer functions

Tamás Kozsik Melinda Tóth István Bozó

Eötvös Loránd University

Content

- Static source code analysis
- Tool-based refactoring
  - Preserve semantics while changing the code
  - Avoid *copy-and-paste* and *replace-all* errors
  - Human guided, automatized transformations
  - Faster, easier and more reliable!
  - Bounded expressiveness

Divide-and-Conquer

Split problem into smaller ones and recurse!

- Sorting (Quicksort, Mergesort, Bucketsort...)
- Mini-max search
- Karatsuba-multiplication
- ...

Many possible applications in HPC!

special hylomorphism

Divide-and-Conquer structure

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

Divide-and-Conquer HOF

```
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)
              Sols = map(Solve, SubProblems)
              Combine(Sols)
         end
  in Solve
```

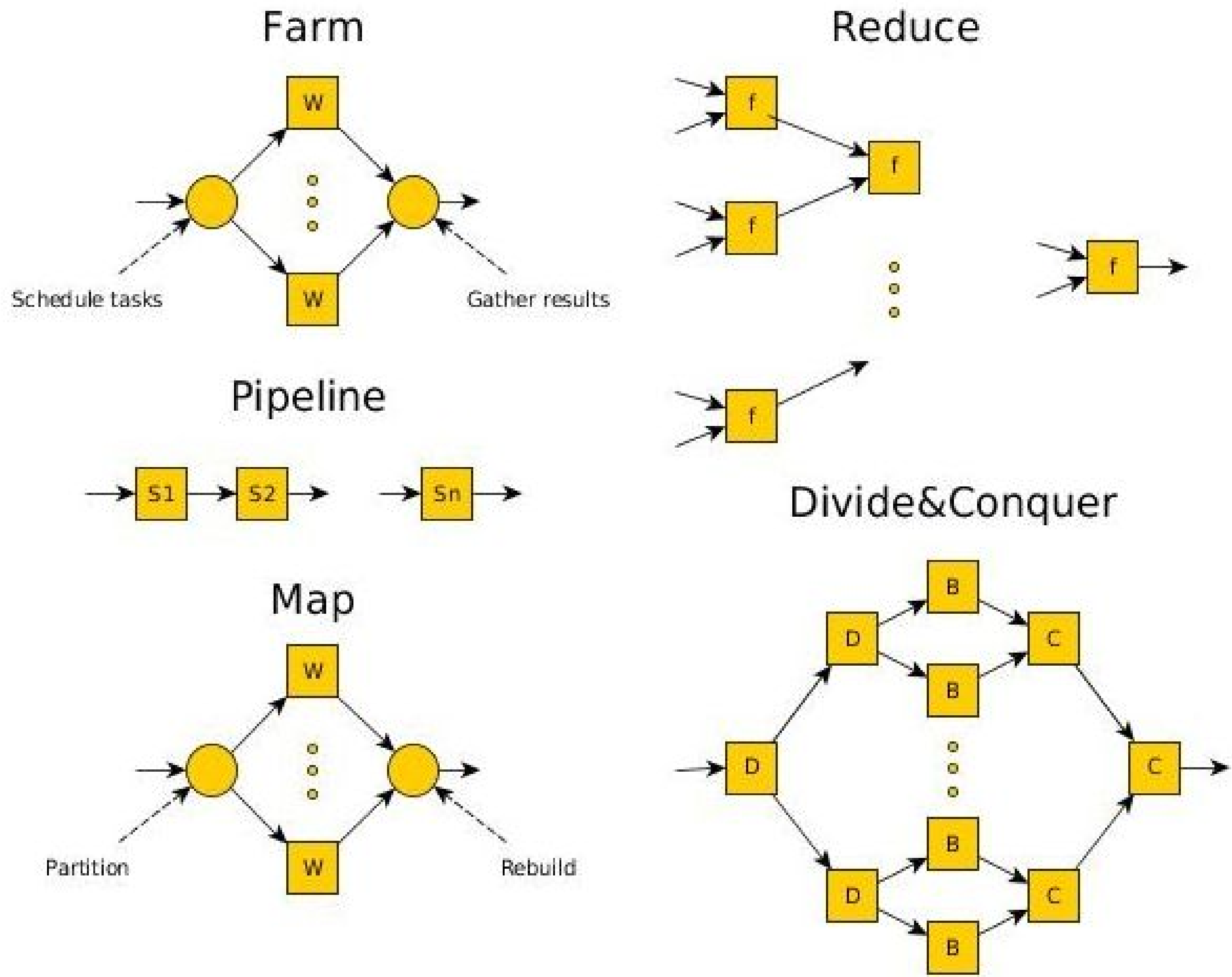
Motivation

- Highly heterogeneous mega-core computers
- Pattern-based parallelism

Parallel Patterns for
Adaptive Heterogeneous Multicore Systems

PARAPHRASE

SEVENTH FRAMEWORK PROGRAMME



Kozsik et al.: Refactoring divide-and-conquer functions

Mergesort

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

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

ms( [H|T]=L )   ->
    {LL,LR} = lists:split(length(L) div 2, L),
    merge( ms(LL), ms(LR) ).


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

Mergesort with d&c pattern

```
ms(List) ->
 (skel_hlp:dc(
    fun(L) -> length(L) < 2 end, % base case?
    fun(L) -> L end,             % base case
    fun(L) ->                    % divide
     {LL,LR}=lists:split(length(L) div 2, L),
     [LL,LR] end,
    fun([L1,L2]) ->              % combine
     merge(L1,L2) end
 ))(List).
```

Divide-and-Conquer (seq)

```
dc(IsBase,Base,Divide,Combine) ->
  Knot =
     fun(Self,Problem) ->
        case IsBase(Problem) of
           true  ->
             Base(Problem);
           false ->
             PS = Divide(Problem),
             SS = [Self(Self,P) || P <- PS]
             Combine(SS)
        end
     end,
  fun(Problem) -> Knot(Knot,Problem) end.
```

Divide-and-Conquer (par)

```
dc(IsBase,Base,Divide,Combine)

dc(IsBase,Base,Divide,Combine,NrProc)
% at most NrProc processes

dc(IsSeq,IsBase,Base,Divide,Combine)
% switch to sequential at IsSeq
```

We provide tooling to...

- **F**ind divide-and-conquer pattern candidates
- **A**ssist in making parallelization decisions
- **R**efactor candidates to the pattern

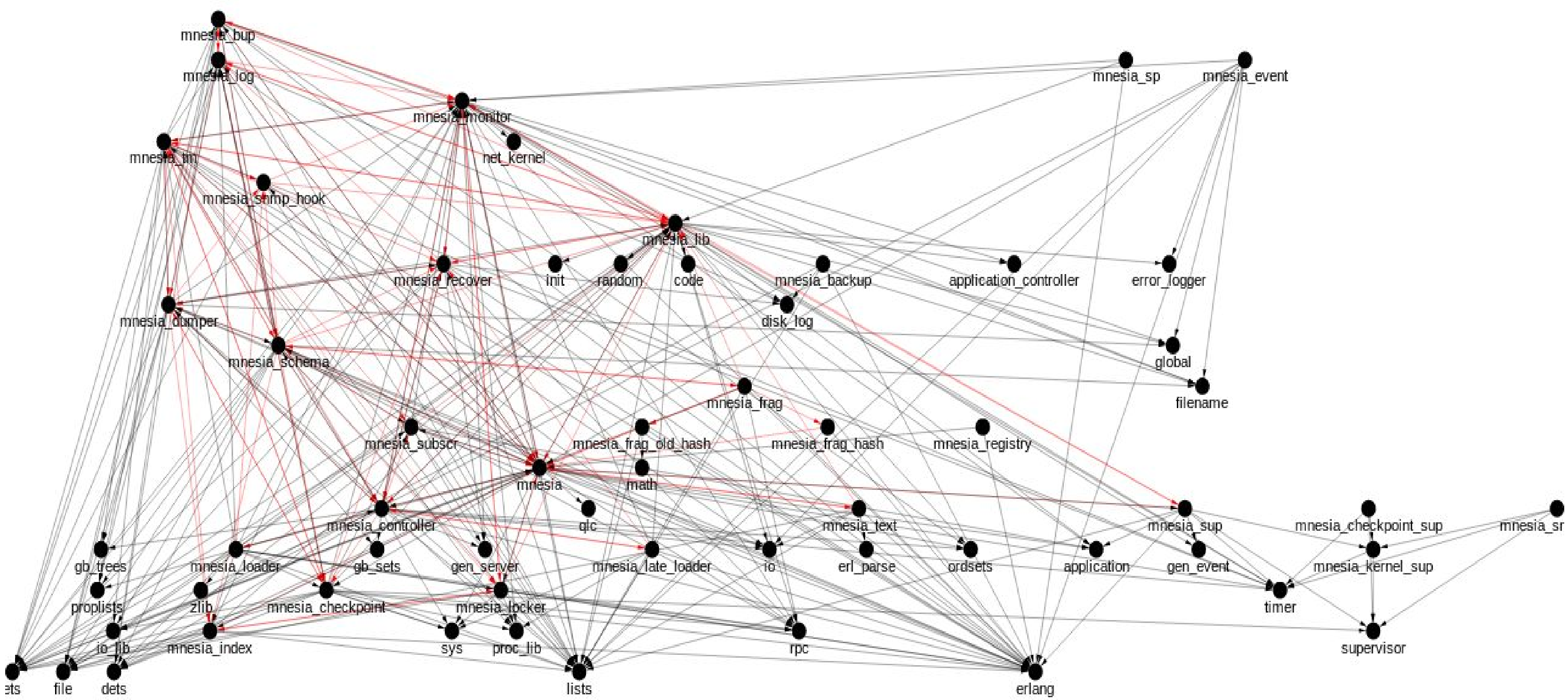
PaRTE

ParaPhrase Refactoring Tool for Erlang

(Powered by RefactorErl)

RefactorErl

Static source code analyzer and transformer: http://plc.inf.elte.hu/erlang/



Kozsik et al.: Refactoring divide-and-conquer functions

Standard Refactorings

- Rename/Move definitions
- Introduce/Eliminate function/variable
- Generalize function
- Reorder/Tuple function parameters

plus many Erlang-specific transformations,

altogether 24 refactorings

Intro Divide-and-Conquer

- Find functions which are d&c (candidates)
- Transform them step-by-step into *canonical* form

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

- Replace canonical form with call to skeleton

D&C candidate

- Our definition:

a function activates itself
multiple times on the same execution path,
with parameters not depending on recursive calls

- Many syntactical possibilities!
  - explicit recursive calls
  - lists:map or list comprehensions
  - mutual recursion

Karatsuba

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

Minimax

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


mm_min( Node, Depth ) -> ... mm_max ...
```

Novel refactorings

- Function clauses to case expression
- Group case branches
- Bindings to list
- Introduce lists:map/2
- Merge function definitions
- Move in/out expression to/from case
- Case on list comprehension
- Eliminate single branch in case expression

Function clauses to case expr.

```
ms(  []      )   ->  [];
ms( [H]      )   -> [H];
ms( [H|T]=L )   ->
    {LL,LR} = lists:split(length(L) div 2, L),
    merge( ms(LL), ms(LR) ).

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

Function clauses to case expr.

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

Group case branches

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

```
ms( L ) ->
  IsBase = case L of
    []    -> true;
    [H]   -> true;
    [H|T] -> false
  end,
  case IsBase of
    true  -> case L of
                 []    ->  [];
                 [H]   -> [H]
             end;
    false -> case L of
                 [H|T] -> {LL,LR} = lists:split(
                                        length(L) div 2, L),
                          merge( ms(LL), ms(LR) )
  end        end.
```

Introduce function

```
ms( L ) ->
  IsBase = case L of
    []    -> true;
    [H]   -> true;
    [H|T] -> false
  end,
  case IsBase of
    true  -> case L of
                  []    ->  [];
                  [H]   -> [H]
                end;
    false -> case L of
                  [H|T] -> {LL,LR} = lists:split(
                                    length(L) div 2, L),
                           merge( ms(LL), ms(LR) )
  end         end.
```

Introduce function

```
                isBase( L ) ->   case L of
                                   []    -> true;
ms( L ) ->                         [H]   -> true;
  IsBase = isBase(L),              [H|T] -> false
  case IsBase of                 end.
    true  -> case L of
               []    ->  [];
               [H]   -> [H]
             end;
    false -> case L of
               [H|T] -> {LL,LR} = lists:split(
                                    length(L) div 2, L),
                        merge( ms(LL), ms(LR) )
             end
  end.
```

Eliminate variable

```
isBase( L ) ->   case L of
                   []    -> true;
                   [H]   -> true;
                   [H|T] -> false
                 end.
```

```
ms( L ) ->
  IsBase = isBase(L),
  case IsBase of
    true  -> case L of
               []    ->  [];
               [H]   -> [H]
             end;
    false -> case L of
               [H|T] -> {LL,LR} = lists:split(
                                    length(L) div 2, L),
                        merge( ms(LL), ms(LR) )
             end
  end.
```

Eliminate variable

```
isBase( L ) ->   case L of
                   []    -> true;
                   [H]   -> true;
                   [H|T] -> false
                 end.
```

```
ms( L ) ->
  case isBase(L) of
    true  -> case L of
               []    ->  [];
               [H]   -> [H]
             end;
    false -> case L of
               [H|T] -> {LL,LR} = lists:split(
                                     length(L) div 2, L),
                        merge( ms(LL), ms(LR) )
             end
  end.
```

Canonical form

```
ms( L ) ->
  case isBase(L) of
    true  -> solve(L);
    false -> Problems = divide(L),
             Solutions = lists:map(fun ms/1, Problems),
             combine(Solutions)
  end.

combine( ListOfLists ) -> ...
solve ( L ) -> ...
isBase( L ) -> ...
divide( L ) -> ...
```

Divide-and-Conquer pattern

```
ms( L ) ->
   (skel_hlp:dc( fun isBase/1,
                 fun solve/1,
                 fun divide/1,
                 fun combine/1 ))(L).



combine( ListOfLists ) -> ...
solve ( L ) -> ...
isBase( L ) -> ...
divide( L ) -> ...
```

Merge function definitions

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


mm_min( {Node, Depth} ) ->
                     ... mm_max ...
```

Merge function definitions

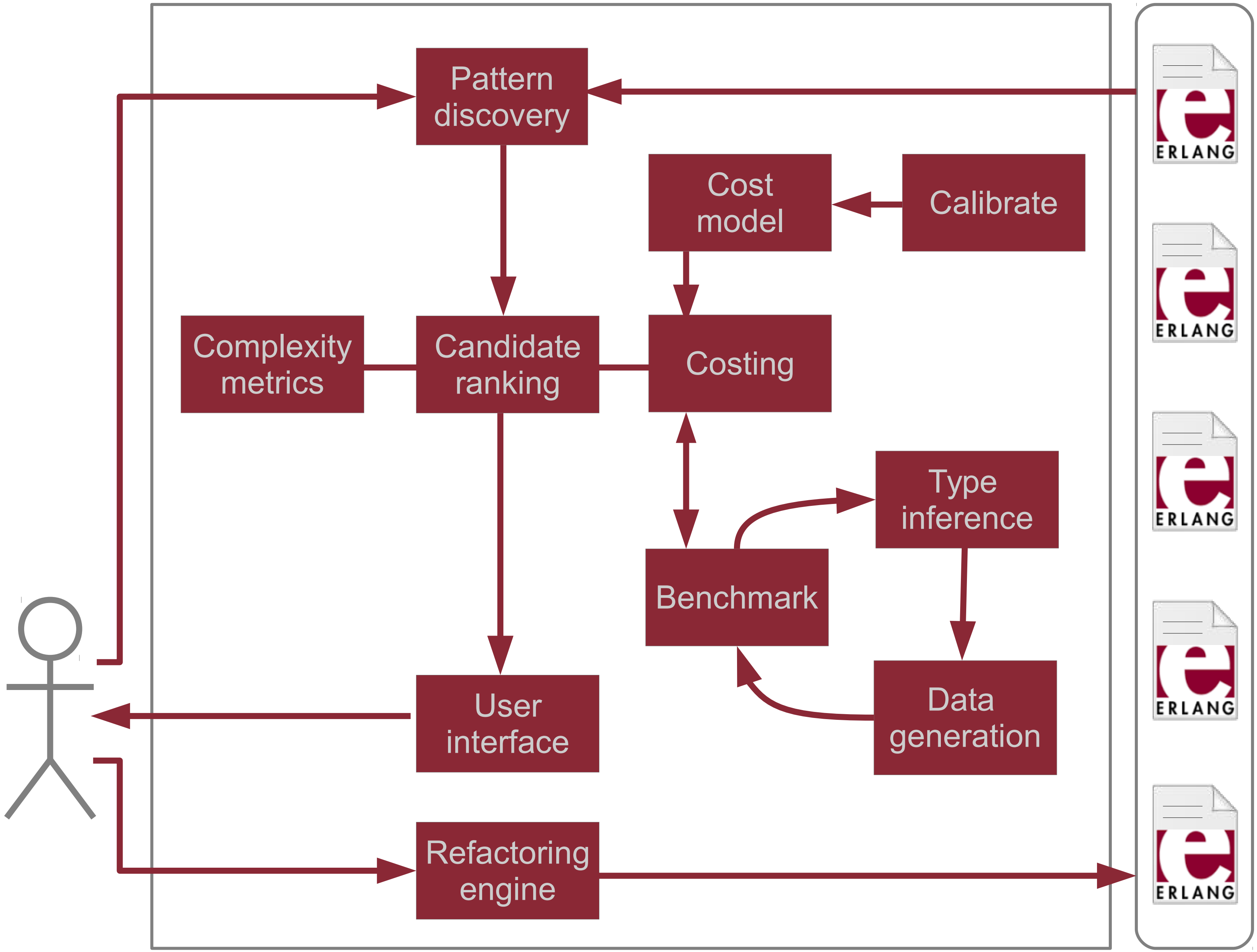
```
mm( {Node, Depth}, max ) ->
  case Depth == 0 orelse terminal(Node)
    true  -> value ( Node ) ;
    false -> lists:max([ mm({C, Depth-1},min)
                       || C <- children(Node)
                       ])
  end;

mm( {Node, Depth}, min ) ->
                     ... mm(... , max) ...
```

Conclusions

- Pattern-based parallelism
- Pattern discovery and refactoring tool
- Step-by-step refactoring of D&C functions
  - Programmer guided, performed with tool
  - Preserving semantics (proof?)
  - Small but meaningful transformations
    - Generic or d&c-specific
  - Compound transformation?
  - Manual refactoring can come between

Mergesort with processes

```
ms( [H|T]=L )   ->
    {LL,LR} = lists:split(length(L) div 2, L),
    Parent = self(),
    spawn( fun() -> Parent ! ms(LL) end ),
    spawn( fun() -> Parent ! ms(LR) end ),
    receive L1 -> ok end,
    receive L2 -> ok end,
    merge( L1, L2 ).
```



Pattern discovery
Cost model
Calibrate
Complexity metrics
Candidate ranking
Costing
Type inference
Benchmark
Data generation
User interface
Refactoring engine
ERLANG
ERLANG
ERLANG
ERLANG
ERLANG

Refactoring Mergesort

```
ms(  []       )   ->  [];
ms( [H]       )   -> [H];
ms( [H|T]=L )   ->
    {LL,LR} = lists:split(length(L) div 2, L),
    merge( ms(LL), ms(LR) ).

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

Function clauses to case expr.

```
ms(  []      )   ->  [];
ms( [H]      )   -> [H];
ms( [H|T]=L )   ->
   {LL,LR} = lists:split(length(L) div 2, L),
   merge( ms(LL), ms(LR) ).

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

Function clauses to case expr.

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

Group case branches

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

```
ms( L ) ->
  IsBase = case L of
    []    -> true;
    [H]   -> true;
    [H|T] -> false
  end,
  case IsBase of
    true  -> case L of
                 []    ->  [];
                 [H]   -> [H]
               end;
    false -> case L of
                 [H|T] -> {LL,LR} = lists:split(
                                    length(L) div 2, L),
                          merge( ms(LL), ms(LR) )
  end          end.
```

Introduce function

```
ms( L ) ->
  IsBase = case L of
    []    -> true;
    [H]   -> true;
    [H|T] -> false
  end,
  case IsBase of
    true  -> case L of
                [] ->  [];
                [H]   -> [H]
              end;
    false -> case L of
                [H|T] -> {LL,LR} = lists:split(
                                   length(L) div 2, L),
                         merge( ms(LL), ms(LR) )
  end         end.
```

Introduce function

```
                   isBase( L ) ->   case L of
                                      []     -> true;
ms( L ) ->                            [H]    -> true;
  IsBase = isBase(L),                 [H|T] -> false
  case IsBase of                    end.
    true  -> case L of
               []    ->  [];
               [H]   -> [H]
             end;
    false -> case L of
               [H|T] -> {LL,LR} = lists:split(
                                    length(L) div 2, L),
                        merge( ms(LL), ms(LR) )
             end
  end.
```

Eliminate variable

```
ms( L ) ->
  IsBase = isBase(L),
  case IsBase of
    true  -> case L of
                 []    ->  [];
                 [H]   -> [H]
              end;
    false -> case L of
                 [H|T] -> {LL,LR} = lists:split(
                                    length(L) div 2, L),
                          merge( ms(LL), ms(LR) )
              end
  end.
```

Eliminate variable

```
ms( L ) ->
  case isBase(L) of
    true  -> case L of
               []    ->  [];
               [H]   -> [H]
             end;
    false -> case L of
               [H|T] -> {LL,LR} = lists:split(
                                   length(L) div 2, L),
                        merge( ms(LL), ms(LR) )
             end
  end.
```

Introduce function

```
ms( L ) ->
  case isBase(L) of
    true  -> case L of
               []    ->  [];
               [H]   -> [H]
             end;
    false -> case L of
               [H|T] -> {LL,LR} = lists:split(
                                    length(L) div 2, L),
                        merge( ms(LL), ms(LR) )
             end
  end.
```

Introduce function

```
ms( L ) ->
  case isBase(L) of
    true  -> base(L);
    false -> case L of
               [H|T] -> {LL,LR} = lists:split(
                                    length(L) div 2, L),
                        merge( ms(LL), ms(LR) )
             end
  end.

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

Eliminate case expression

```
ms( L ) ->
  case isBase(L) of
    true  -> base(L);
    false -> case L of
               [H|T] -> {LL,LR} = lists:split(
                                   length(L) div 2, L),
                        merge( ms(LL), ms(LR) )
             end
  end.
```

Eliminate case expression

```
ms( L ) ->
  case isBase(L) of
    true  -> base(L);
    false -> [H|T] = L,
             {LL,LR} = lists:split(length(L) div 2, L),
             merge( ms(LL), ms(LR) )
  end.
```

Introduce function

```
ms( L ) ->
  case isBase(L) of
    true  -> base(L);
    false -> [H|T] = L,
             {LL,LR} = lists:split(length(L) div 2, L),
             merge( ms(LL), ms(LR) )
  end.
```

Introduce function

```
ms( L ) ->
  case isBase(L) of
    true  -> base(L);
    false -> {LL,LR} = divide(L),
             merge( ms(LL), ms(LR) )
  end.

divide( L ) ->
  [H|T] = L,
  {LL,LR} = lists:split(length(L) div 2, L),
  {LL,LR}.
```

Introduce variables

```
ms( L ) ->
  case isBase(L) of
    true  -> base(L);
    false -> {LL,LR} = divide(L),
             merge( ms(LL), ms(LR) )
  end.
```

Introduce variables

```
ms( L ) ->
  case isBase(L) of
    true  -> base(L);
    false -> {LL,LR} = divide(L),
             L1 = ms(LL),
             L2 = ms(LR),
             merge( L1, L2 )
  end.
```

Bindings to list

```
ms( L ) ->
  case isBase(L) of
    true  -> base(L);
    false -> {LL,LR} = divide(L),
             L1 = ms(LL),
             L2 = ms(LR),
             merge( L1, L2 )
  end.
```

Bindings to list

```
ms( L ) ->
  case isBase(L) of
    true  -> base(L);
    false -> {LL,LR} = divide(L),
             [L1, L2] = [ms(LL), ms(LR)],
             merge( L1, L2 )
  end.
```

Introduce lists:map/2

```
ms( L ) ->
  case isBase(L) of
    true  -> base(L);
    false -> {LL,LR} = divide(L),
             [L1, L2] = [ms(LL), ms(LR)],
             merge( L1, L2 )
  end.
```

Introduce lists:map/2

```
ms( L ) ->
  case isBase(L) of
    true  -> base(L);
    false -> {LL,LR} = divide(L),
             [L1, L2] = lists:map(fun ms/1, [LL,LR]),
             merge( L1, L2 )
  end.
```

Introduce variable

```
ms( L ) ->
  case isBase(L) of
    true  -> base(L);
    false -> {LL,LR} = divide(L),
             [L1, L2] = lists:map(fun ms/1, [LL,LR]),
             merge( L1, L2 )
  end.
```

Introduce variable

```
ms( L ) ->
  case isBase(L) of
    true  -> base(L);
    false -> {LL,LR} = divide(L),
             ListOfLists = lists:map(fun ms/1,[LL,LR]),
             [L1, L2] = ListOfLists,
             merge( L1, L2 )
  end.
```

Introduce function

```
ms( L ) ->
  case isBase(L) of
    true  -> base(L);
    false -> {LL,LR} = divide(L),
             ListOfLists = lists:map(fun ms/1,[LL,LR]),
             [L1, L2] = ListOfLists,
             merge( L1, L2 )
  end.
```

Introduce function

```
ms( L ) ->
  case isBase(L) of
    true  -> base(L);
    false -> {LL,LR} = divide(L),
             ListOfLists = lists:map(fun ms/1,[LL,LR]),
             combine(ListOfLists)
  end.

combine( ListOfLists ) ->
             [L1, L2] = ListOfLists,
             merge( L1, L2 ).
```

Canonical form

```
ms( L ) ->
  case isBase(L) of
    true  -> base(L);
    false -> {LL,LR} = divide(L),
             ListOfLists = lists:map(fun ms/1,[LL,LR]),
             combine(ListOfLists)
  end.

combine( ListOfLists ) -> ...
solve ( L ) -> ...
isBase( L ) -> ...
divide( L ) -> ...
```

Divide-and-Conquer pattern

```
ms( L ) ->
  (skel_hlp:dc( fun isBase/1,
                fun solve/1,
                fun divide/1,
                fun combine/1 ))(L).



combine( ListOfLists ) -> ...
solve ( L ) -> ...
isBase( L ) -> ...
divide( L ) -> ...
```