PROOF FOUNDATIONS

(W) WEISER DEFINITION OF SLICING:
Given a program P, an slicing criterion C=<v,s> where v is a variable at statement s, and an slice S: If P halts on input I, then the value of v at statement s each time s is executed in P is the same in P and S. If P fails to terminate normally, s may be executed more times in S than in P, but P and S compute the same values for v each time s is executed by P.

(A) DATA DEPENDENCE:
We say there exists a data dependence between two expressions when the first expression defines the value of a variable and the second one uses this value in at least one of the possible program executions without being any other expression modifying it.
NOTE: We consider that the arguments passed in a function call and the parameters of that function are a specific case of data dependence where the expression changes its name.

(B) CONTROL DEPENDENCE:
There exists a control dependence between two expressions when the second expression cannot be evaluated without evaluating the first expression.

(C) SEQUENTIAL REDUNDANCE:
When the return expression of a block or a function (the last expression of the block in Erlang) is a variable defined in the previous expression, this can be deleted avoiding the definition of this variable and returning the result of the previous expression, taking this expression the last position of the block and being returned in consecuense.

(D) SYNTAX ERRROR:
We say there exists a syntax error in a program when the removal or modification of a chosen expression transforms the program into a non-executable state.

(E) SEMANTIC MODIFICATION:
There exists a semantic modification in an expression when the modification of one of its subexpressions modifies the behaviour of the whole expression.

(F) ABSORBING PROPERTY:
A clause of a conditional or a function statement is absorbing when its guard is always evaluated to true or its pattern always matches.

(G) FULL TEST VALIDATION:
There exists full test validation when an original program and a slice extracted from it can be executed with all possible input values of the original program and the values of the slicing criterion are the same in both executions.
NOTE: We consider in this definition also programs with slicing criteria that are independent of program inputs, where there is only one possible execution.

COLOUR LEGEND

Black: Expressions deleted by executing phase 1 (iterative slicing with the selected slicers)
Red: Expressions deleted by executing phase 2 (modified ORBS algorithm)
Green: Expressions remaining in the quasi-minimal slices
Orange: Slicing Criterion

NOTE1: We will not prove wether black expressions of the program code can be deleted or not because they have been deleted by phase 1. Phase 1 produces a complete slice of the original code, thus we can guarantee that these expressions are not part of the minimal slice.
NOTE2: Our slices keep the syntax of the original program (we are not interested in amorphous slices). However, in order to make the final slice executable, some modifications of the source code are compulsory (e.g., replacing calls to deleted functions with a constant called "undef"). Therefore, we allow for some modifications of the source code to produce executable slices. The modifications made never affect the behaviour of the source code, they just ensure that the final code is a valid Erlang program.

```
%------------------------------------------------------------------------------------------
%------------------------------------------------------------------------------------------
%-- bench8.erl
%--
%-- AUTHORS:     Anonymous
%-- DATE:        2016
%-- PUBLISHED:   Software specially developed to test tail recursion and guards.
%-- COPYRIGHT:   Bencher: The Program Slicing Benchmark Suite for Erlang
%--              (Universitat Politècnica de València)
%--              http://www.dsic.upv.es/~jsilva/slicing/bencher/
%-- DESCRIPTION
%-- The program receives a list of movements in a saving account with the format
%-- {Movement, Amount, User} and a user Who. It returns two lists, one list
%-- for the user's deposits greater than 300 and another one for the withdraws
%-- greater than 100.
%------------------------------------------------------------------------------------------
%------------------------------------------------------------------------------------------
```

```
-module(bench8).
-export([main/2]).
main(Account,Who) ->
%Given (A), Account and Who are necessary w.r.t
recount(Who,Account)

        {Deposits, Withdraws} = recount(Who, Account).
%Deposits is the SC
%{Deposits, Withdraws} cannot be replaced with _
(NOTE2) because the SC would be removed
%Given (A), Who and Account are necessary w.r.t.
recount(Person,List)
%recount(…) is the only expression that can
assign a value to the SC. If we replace it with
undef the SC would never be reached because of
a matching error

recount(Person, List) ->
%Delete the recound() function would produce (D)
w.r.t    the    recount    function    call
recount(Who,Account), in the main function
%Given (A), Person and List are necessary w.r.t.
tailrecount(Person,List,[],[])

        tailrecount(Person,List,[],[]).
%Given (L1), there is not data dependence (A)
between [] and any other expression of the slice,
so we can delete []
%Given (A), Person, List and [] are necessary
w.r.t.
tailrecount(Person,[H|T],LDeposits,LWithdraws)
%The tailrecount(…) call is necessary because it
is the only expression of the function and defines
the returned value of the recount function. If we
replace it with undef, the SC would never be
reached because of a matching error

tailrecount(Person,[],LDeposits, LWithdraws) ->
%Replace [] with _ (NOTE2) would produce (F) and
thus (2) is not satisfied because of (E)
%Given (A), LDepostis is necessary w.r.t.
{LDeposits,LWithdraws}

        {LDeposits,LWithdraws};
%{LDeposits,LWithdraws} cannot be replaced with
undef (NOTE2) because the SC would never be reached
because of a matching error
%LDeposits is the only expression that can give a
value to the SC by performing 0 or more
transformations

tailrecount(Person,[H|T],LDeposits,LWithdraws) ->
%Delete this function clause makes impossible to
satisfy (2). It would be possible to satisfy (1)
& (3) by modifying the first clause but not (2)
%Given (A), Person, H, T and LDeposits are
necessary w.r.t. the subexpressions of the case
expression: case H of,
tailrecount(Person,T,Ldeposits,LWithdraws)

    case H of
%case H of … expression is necessary because it
is the only expression of the function and
defines the returned value of the recount
function. If we replace it with undef the SC
would never be reached because of a matching
error
%H cannot be replaced with undef because it makes
impossible to satisfy (2). It would be possible
to satisfy (2) by modifying the second clause but
in this case (1) & (3) would not be satisfied

        {Mov, Amount, Person} when Mov == withdraw
                andalso Amount >100 ->
            tailrecount(Person,T,LDeposits,[H|LWithdraws]);
%Given (D1), this clause has been proven not
necessary to compute the SC value

        {Mov, Amount, Person} when Mov == deposit
                andalso Amount >300 ->
%This clause cannot be deleted because it would
not be possible to satisfy (2)
%This guard or any of its parts cannot be
replaced with true (NOTE2) because it would not
be possible to satisfy (1) & (3) because of (E)
%Delete the Mov or Amount expressions would
produce (D) because they define the parameters
needed in the guards
%Person cannot be replaced with _ (NOTE2) because
it would not be possible to satisfy (2) because of
(E)

            tailrecount(Person,T,[H|LDeposits],LWithdraws);
%tailrecount(…) is necessary because it is the
only expression of the case caluse and defines the
returned value of the recount function. If we
replace it with undef, the SC would never be
reached because of a matching error
%Person cannot be replaced with undef (NOTE2)
because it would not be possible to satisfy (2)
because of (E)
%Given (A), T and [H|LDeposits] are necessary
w.r.t.
tailrecount(Person,[],LDeposits,LWithdraws)
%LDeposits in the [H|LDeposits] expression cannot
```

```
        _ ->                                                  be deleted because it would not be possible to
                                                              satisfy (2) because of (E)
                                                              %Delete this clause implies stop satisfying (1) &
                                                              (3) because of (E) so this clause cannot be
                                                              deleted
            tailrecount(Person,T,LDeposits,LWithdraws)        %tailrecount(…) is necessary because it is the
                                                              only expression of the case clause and defines
                                                              the returned value of the recount function. If we
                                                              replace it with undef the SC would never be
                                                              reached because of a matching error
                                                              %Person, T and LDeposits are necessary because of
                                                              (A)                                          w.r.t.
                                                              tailrecount(Person,[H|T],LDeposits,LWithdraws)
                                                              and tailrecount(Person,[],LDepostis,LWithdraws)

    end.
```

%**L1:** The second and third case clauses of the tailrecount(Person,[H|T],LDeposits,LWithdraws) clause cannot directly assign a value to the Slicing Criterion, because they always return a function call to tailrecount(). The only clause of this function that can assign a value to the Slicing Criterion is tailrecount(Person,[],LDeposits,LWithdraws).
The forth parameter of the second and third clauses and the forth parameter in the clause of the function tailrecount(Person,[H|T],LDeposits,LWithdraws) -> ... (LWithdraws) is only used in the recursive tailrecount clause. This expression is used to store the value of the variable LWithdraws during the whole recursive process. However, this information is completely ignored by the slicers in the base case clause, being unnecessary to compute the Slicing Criterion value. For this reason, we can say the three LWithdraw expressions appearing in the tailrecount recursive clause are not used to calculate the Slicing Criterion.

EXECUTION RESULT:

INPUTS:

```
(1) Account = [{withdraw,390,jack}], Who = jack                SC = []
(2) Account = [{deposit,380,jack},
            {deposit,320,annie},                               SC =[{deposit,380,jack},{deposit,480,jack}]
            {deposit,480,jack}],
    Who = jack
(3) Account = [{transfer,140,annie}], Who = jack               SC = []
(4) Account = [], Who = jack                                   SC = []
```

DEMONSTRATION 1 (D1)

Involved code

```
I       tailrecount(Person,List,[],[]).
II  tailrecount(_,[],LDeposits, _) ->
III     {LDeposits,undef};
IV  tailrecount(Person,[H|T],LDeposits,LWithdraws) ->
V       case H of
VI              {Mov, Amount, Person} when Mov == withdraw andalso Amount >100 ->
VII                     tailrecount(Person,T,LDeposits,undef);
VIII            {Mov, Amount, Person} when Mov == deposit andalso Amount >300 ->
IX                      tailrecount(Person,T,[H|LDeposits],LWithdraws);
X               _ ->
XI                      tailrecount(Person,T,LDeposits,LWithdraws)
XII     end.
```

Base case

Precondition: List = []

The list is empty, so it enters into the first clause of the tailrecount function (II):
tailrecount(_,[],LDeposits,_) -> {LDeposits,undef};

Consequence: Red case clause (VI) won't be executed, so it is not part of the slice

----

I.H.

Precondition: length(List) = N > 0
Consequence: None of the N recursive calls that analyses the N elements of the list requires the execution of the red case clause (VI), so it is not part of the slice

----

I.C.

Precondition: length(List)= N+1 > 0

Because of I.H. the first N recursive calls that analyses the first N elements of the list do not require the execution of the red case clause

In the last recursive call:

Value of variable List: List = [H]

Pattern matching with the second clause of the tailrecount function:
(IV) tailrecount(Person,[H],LDeposits,LWithdraws)

MATCHING ANALYSIS

1) clause 1 (Line VI)
After extracting the H element from the list, this clause executes the expression:
(VII) tailrecount(Person,[],LDeposits,undef)

This execution calls the base case with the following values:
(III) {LDeposits,undef}

The SC before and after the execution of this clause is not modified. The parameter LDeposits is used when calling the function in its base case, so clause 1 is not modifying the SC.

2) clause 2 (Line VIII)
After extracting the H element from the list, this clause executes the expression:
(IX) tailrecount(Person,[],[H|LDeposits],LWithdraws)

This execution calls the base case returning the following values:
(III) {[H|LDeposits],undef}

The SC before and after this clause execution is modified by adding the H expression to the list LDepostis.

3) clause 3 (Line X)
After extracting the H element from the list, this clause executes the expression:
(XI) tailrecount(Person,[],LDeposits,LWithdraws)

This execution calls the base case returning the following values:
(III) {LDeposits,undef}

The SC before and after the execution of this clause is not modified. The LDeposits parameter is used when calling the function in its base case, so the clause 3 is not modifying the SC.

DEDUCED PROPERTIES:

(a) the guards of the clauses 1 and 2 are disjoint:
Mov == withdraw andalso Amount >100 ∩ Mov == deposit andalso Amount >300 = ∅

(b) the clauses 1 and 3 perform the same behaviour with respect to the SC (both return {LDeposits,undef})

(c) the clause 3 fullfils (F).

Taking **(a),(b)&(c)** in mind we can say that **the clause 1 is not necessary** to calculate the value of SC in the last iteration.

CONCLUSION: The clause 1 of the case expression was not necessary in the last iteration, then because of I.H. we can say that the clause 1 was not necessary in the previous N iterations. Because of this, we can conclude that the clause 1 of the case expression is not necessary to calculate the value of the SC and so it is not part of the minimal slice.