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, so we can guarantee that these expressions are not part of the 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.

```
%-------------------------------------------------------------------------------------------
%-------------------------------------------------------------------------------------------
%-- bench12.erl
%--
%-- AUTHORS:      Tamarit-Emartinm
%-- DATE:         2013
%-- PUBLISHED:    https://github.com/tamarit/edd/blob/master/examples/ternary (2016)
%-- COPYRIGHT:    Bencher: The Program Slicing Benchmark Suite for Erlang
%--               (Universitat Politècnica de València)
%--               http://www.dsic.upv.es/~jsilva/slicing/bencher/
%-- DESCRIPTION
%-- The program performs translations and operations with balanced ternary. Its inputs are
%-- two numbers in balanced ternary representation and a decimal number. It converts the
%-- three numbers to the oposite representation and performs an operation with their
%-- balanced ternary representation. The output is a tuple containing the input numbers
%-- and the result in both representations.
%-------------------------------------------------------------------------------------------
%-------------------------------------------------------------------------------------------
```

```
-module(bench12).
-export([main/3]).

main(AS,B,CS) ->
    AT = from_string(AS), A = from_ternary(AT),
    BT = to_ternary(B),

    BS = to_string(BT),
    CT = from_string(CS), C = from_ternary(CT),
    RT = mul(AT,sub(BT,CT)),

    R = from_ternary(RT),
    RS = to_string(RT),
    [{AS,A},{BS,B},{CS,C},{RS,R}].

to_string(T) -> [to_char(X) || X <- T].

from_string(S) -> [from_char(X) || X <- S].

to_char(-1) -> $-;
to_char(0) -> $0;
to_char(1) -> $+.

from_char($-) -> -1;
from_char($0) -> 0;
from_char($+) -> 1.

to_ternary(N) when N > 0 ->

    to_ternary(N,[]);

to_ternary(N) ->

    neg(to_ternary(-N)).

to_ternary(0,Acc) ->

    Acc;

to_ternary(N,Acc) when N rem 3 == 0 ->
```

```
%Given (A), B is necessary w.r.t. BT = to_ternary(B)

%Given (A), BT is necessary w.r.t. the expression
RT = mul(AT,sub(BT,CT))
%Replace to_ternary(BT) with undef (NOTE2) would prevent
to satisfy (1)&(2)

%Delete RT = mul(AT,sub(BT,CT)) would prevent to reach the
SC because it is the only call to the sub function, where
the SC is located
%Replace sub(BT,CT) with undef would prevent to reach the
SC
%Given (A), BT is necessary w.r.t. sub(A,B). Replace it
with undef (NOTE2) would prevent to satisfy (1)&(2)

%Delete this clause would prevent to reach the SC
%Given (A), N is necessary w.r.t. to_ternary(N,[])
%Guard when N > 0 cannot be deleted because it would
prevent to satisfy (2)
%Replace N with undef (NOTE2) would make this clause
fulfill (F) and it would prevent to satisfy (2)
%Replace 0 with undef (NOTE2) would prevent to satisfy (1)
because of (E). This clause would become unreachable
%to_ternary(N,[]) cannot be deleted because it is the only
expression of the clause. Replace it with undef (NOTE2)
would prevent to satisfy (1)&(2)
%Replace N with undef (NOTE2) would prevent to reach the
SC because of a badarith error in the to_ternary/2 function
%Replace [] with undef (NOTE2) would prevent to satisfy
(1)&(2)
%Delete this clause would prevent to satisfy (2)
%Given (A), N is necessary w.r.t. neg(to_ternary(-N))
%neg(to_ternary(-N)) cannot be deleted because it is the
only expression of the clause. Replace it with undef
(NOTE2) would prevent to satisfy (2)
%Replace to_ternary(-N) with undef (NOTE2) would prevent
to reach the SC because of a bad generator error in the
neg function
%Replace -N with undef (NOTE2) would prevent to reach the
SC because of a badarith error in the to_ternary/2 function
%This clause is the base case of the to_ternary/2 recursive
function. Delete this clause would prevent to reach the SC
because of an infinite loop
%Replace 0 with _ (NOTE2) would make this clause fullfils
(F) and this would prevent to satisfy (1)&(2)
%Given (A), Acc is necessary w.r.t. Acc
%Acc is the only expression of the clause and it will be
the retuned value of the to_ternary/2 function because it
is the returned value of its base case
%Delete this clause would prevent to satisfy (1)&(2)
%Given (A), N is necessary w.r.t.
to_ternary(N div 3, [0|Acc])
%Given (A), Acc is necessary w.r.t.
to_ternary(N div 3, [0|Acc])
%Delete guard when N rem 3 == 0 would make this clause
fullfils (F) and it would prevent to satisfy (1)&(2)
%Replace N rem 3 with undef (NOTE2) would prevent to
satisfy (1)&(2) because this clause would be unreachable.
This could be avoided by replacing also 0 with undef, but
this would make this clause fullfils (F) and it would also
prevent to satisfy (1)&(2)
%Replace N or 3 in N rem 3 with undef (NOTE2) would prevent
to reach the SC because of a badarith error
```

```
                                        %Replace 0 with undef (NOTE2) would make this clause
                                        unreachable and it would prevent to satisfy (1)&(2). This
                                        could be avoided by replacing also N rem 3 with undef, but
                                        this would make this clause fullfils (F) and it would also
                                        prevent to satisfy (1)&(2)
    to_ternary(N div 3, [0|Acc]);       %to_ternary(N div 3, [0|Acc]) cannot be deleted because it
                                        is the only expression of the clause. Replace it with undef
                                        (NOTE2) would prevent to satisfy (1)&(2)
                                        %Given (A), N div 3 is necessary w.r.t. to_ternary(N,Acc).
                                        Replace N div 3 with undef (NOTE2) would prevent to reach
                                        the SC in (1)&(2)because of a badarith error
                                        %Replace N or 3 with undef (NOTE2) would prevent to reach
                                        the SC in (1)&(2) due to a badarith error
                                        %Given (A), [0|Acc] is necessary w.r.t. to_ternary(0,Acc).
                                        This clause is the base case of the function and parameter
                                        Acc in the clause is necessary to calculate the returned
                                        value
                                        %Replace 0 with undef (NOTE2) would prevent to satisfy
                                        (1)&(2)
                                        %Delete Acc would prevent to satisfy (1)&(2) because each
                                        call to this clause would replace list Acc (returned in
                                        the base case) with list [0]
to_ternary(N,Acc) when N rem 3 == 1 ->  %Delete this clause would prevent to satisfy (1)&(2)
                                        %Given (A), N is necessary w.r.t.
                                        to_ternary(N div 3, [1|Acc])
                                        %Given (A), Acc is necessary w.r.t.
                                        to_ternary(N div 3, [1|Acc])
                                        %Delete guard when N rem 3 == 1 would make this clause
                                        fullfils (F) and it would prevent to satisfy (1)&(2)
                                        %Replace N rem 3 with undef (NOTE2) would prevent to
                                        satisfy (1)&(2) because this clause would be unreachable.
                                        This could be avoided by replacing also 1 with undef, but
                                        this would make this clause fullfils (F) and it would also
                                        prevent to satisfy (1)&(2)
                                        %Replace N or 3 in N rem 3 with undef (NOTE2) would prevent
                                        to reach the SC because of a badarith error
                                        %Replace 1 with undef (NOTE2) would make this clause
                                        unreachable and it would prevent to satisfy (1)&(2). This
                                        could be avoided by replacing also N rem 3 with undef, but
                                        this would make this clause fullfils (F) and it would also
                                        prevent to satisfy (1)&(2)
    to_ternary(N div 3, [1|Acc]);       %to_ternary(N div 3, [1|Acc]) cannot be deleted because it
                                        is the only expression of the clause. Replace it with undef
                                        (NOTE2) would prevent to satisfy (1)&(2)
                                        %Given (A), N div 3 is necessary w.r.t. to_ternary(N,Acc).
                                        Replace N div 3 with undef (NOTE2) would prevent to reach
                                        the SC in (1)&(2) because of a badarith error
                                        %Replace N or 3 with undef (NOTE2) would prevent to reach
                                        the SC in (1)&(2) due to a badarith error
                                        %Given (A), [1|Acc] is necessary w.r.t. to_ternary(0,Acc).
                                        This clause is the base case of the function and parameter
                                        Acc in the clause is necessary to calculate the returned
                                        value
                                        %Replace 1 with undef (NOTE2) would prevent to satisfy
                                        (1)&(2)
                                        %Delete Acc would prevent to satisfy (1)&(2) because each
                                        call to this clause would replace list Acc (returned in
                                        the base case) with list [1]
to_ternary(N,Acc) ->                    %Delete this clause would prevent to satisfy (1)&(2)
                                        because of a matching error
                                        %Given (A), N is necessary w.r.t.
                                        to_ternary((N+1) div 3, [-1|Acc])
                                        %Given (A), Acc is necessary w.r.t.
                                        to_ternary((N+1) div 3, [-1|Acc])
    X = to_ternary((N+1) div 3, [-1|Acc]). %X = to_ternary((N+1)div 3, [-1|Acc]) cannot be deleted
                                        because it is the only expression of the clause. Replace
                                        this expression with undef (NOTE2) would prevent to
                                        satisfy (1)&(2)
                                        %Given (A), (N+1) div 3 is necessary w.r.t.
                                        to_ternary(N,Acc). Replace (N+1) div 3 with undef (NOTE2)
                                        would prevent to reach the SC in (1)&(2) because of a
                                        badarith error
                                        %Replace (N+1) or 3 with undef (NOTE2) would prevent to
                                        reach the SC due to a badarith error
                                        %Replace N or 1 in (N+1) with undef (NOTE2) would prevent
                                        to reach the SC in (1)&(2) due to a badarith error
                                        %Given (A), [-1|Acc] is necessary w.r.t.
                                        to_ternary(0,Acc). This clause is the base case of the
                                        function and parameter Acc in the clause is necessary to
                                        calculate the returned value
                                        %Replace -1 with undef (NOTE2) would prevent to satisfy
                                        (1)&(2)
```

```
                                                    %Delete Acc would prevent to satisfy (1)&(2) because each
                                                    call to this clause would replace list Acc (returned in
                                                    the base case) with list [-1]

from_ternary(T) -> from_ternary(T,0).

from_ternary([],Acc) ->
    Acc;
from_ternary([H|T],Acc) ->
    from_ternary(T,Acc*3 + H).

mul(A,B) -> mul(B,A,[]).

mul(_,[],Acc) ->
    Acc;
mul(B,[A|As],Acc) ->
    BP = case A of
             -1 -> neg(B);
             0 ->  [0];
             1 ->  B
         end,
    A1 = Acc++[0],
    A2=add(BP,A1),
    mul(B,As,A2).

neg(T) ->                                           %Given (A), T is necessary w.r.t. [-H || H <- T]
        [ -H || H <- T].                            %[ -H || H <- T] cannot be deleted because it is the only
                                                    expression of the function clause. Replace it with undef
                                                    (NOTE2) would prevent to reach the SC in execution (2) due
                                                    to a badarith error in the to_ternary/2 function
                                                    %Replace -H with undef (NOTE2) would prevent to satisfy
                                                    (2)
                                                    %Replace H with _ (NOTE2) would prevent to reach the SC
                                                    because of (D) w.r.t. [ -H || _ <- T]. This would be
                                                    avoided by replacing -H with undef, but this would prevent
                                                    to satisfy (2)
                                                    %Replace T with undef (NOTE2) would prevent to reach the
                                                    SC due to a bad generator error in the list comprehension
                                                    expression
sub(A,B) ->                                         %Given (A), A is necessary w.r.t. add(A,neg(B)) because it
                                                    defines the value to the SC
        add(A,neg(B)).                              %add(A,neg(B)) cannot be deleted because the SC would be
                                                    deleted
                                                    %A cannot be deleted because it is the SC

add(A,B) when length(A) < length(B) ->
    add(lists:duplicate(length(B)-length(A),0)++A,B);
add(A,B) when length(A) > length(B) ->
   add(B,A);
add(A,B) ->
    add(lists:reverse(A),lists:reverse(B),0,[]).

add([],[],0,Acc) ->
    Acc;
add([],[],C,Acc) ->
    [C|Acc];
add([A|As],[B|Bs],C,Acc) ->
    [C1,D] = add_util(A+B+C),
    add(As,Bs,C1,[D|Acc]).

add_util(-3) -> [-1,0];
add_util(-2) -> [-1,1];
add_util(-1) -> [0,-1];
add_util(3) -> [1,0];
add_util(2) -> [1,-1];
add_util(1) -> [0,1];
add_util(0) -> [0,0].
```

EXECUTION RESULTS:

	SLICING CRITERION
(1) B = 160	SC = [1,-1,0,0,-1,1]
(2) B = -160	SC = [-1,1,0,0,1,-1]