CS-Note* 32

Bill J. Ellis
October 22, 2002

Learning by Example - Exception Freedom in Table Utilities (Delete)

Contents

*Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the NuSPADE project (EPSRC GR/R24081).

1 Introduction

The learn by example strategy, briefly outlined in CS-note 31, is continued for another example.

1.1 Table Utilities

The table utilities example has been considered in detail by Praxis for both partial correctness [4] and exception freedom [3]. A similar cut down version of this example, Array Utilities, is presented in [1] and was considered early in the NuSPADE project. See Appendix A for the code of this example and B for full proof details on the Delete procedure.

2 Thinking outwards from VC 3

2.1 Failure analysis 1

2.1.1 Situation

#	From	To	Proved In: vcg	siv	plg	TO DO
1	start	rtc check @ 108		YES		
2	start	rtc check @ 108		YES		
3	start	rtc check @ 113				YES
4	start	rtc check @ 113				YES
5	start	assert @ 113		YES		
6	113	assert @ 113		YES		
7	113	rtc check @ 114				YES
8	start	rtc check @ 116		YES		
9	113	rtc check @ 116		YES		
10	start	assert @ finish	YES			
11	start	assert @ finish	YES			
12	113	assert @ finish	YES			

2.1.2 Describing VC 3

Location information

```
For path(s) from start to run-time check associated with
statement of line 113:

procedure_delete_3.
```

Code snippet

```
107    begin
108        I := Find(Table, Value);
109        if I = 0 then
110            Success := false;
111        else
112            Success := true;
113            for J in Index_Type range I .. Table.Size - 1 loop
114                Table.Elem(J) := Table.Elem(J+1);
```

Before simplification

```
H1:    true .
H2:     for_all (i___1: base_index_type, ((i___1 >=
           index_type__first) and (i___1 <= index_type__last)) -> ((element(
           fld_elem(table), [i___1]) >= contents_type__first) and (element(
           fld_elem(table), [i___1]) <= contents_type__last))) .
H3:    fld_size(table) >= base_index_type__first .
H4:    fld_size(table) <= base_index_type__last .
H5:    value >= contents_type__first .
H6:    value <= contents_type__last .
H7:    value >= contents_type__first .
H8:    value <= contents_type__last .
H9:    find(table, value) >= base_index_type__first .
H10:   find(table, value) <= base_index_type__last .
H11:   find(table, value) >= base_index_type__first .
H12:   find(table, value) <= base_index_type__last .
H13:   not (find(table, value) = 0) .
        ->
C1:    (find(table, value) <= fld_size(table) - 1) -> ((fld_size(
           table) - 1 >= index_type__first) and (fld_size(
           table) - 1 <= index_type__last)) .
C2:    (find(table, value) <= fld_size(table) - 1) -> ((
           find(table, value) >= index_type__first) and (
           find(table, value) <= index_type__last)) .
C3:    fld_size(table) - 1 >= base_index_type__first .
C4:    fld_size(table) - 1 <= base_index_type__last .
C5:    find(table, value) >= base_index_type__first .
C6:    find(table, value) <= base_index_type__last .
```

After simplification

```
H1:    for_all(i___1 : base_index_type, 1 <= i___1 and i___1 <= 100 -> - 1000
          <= element(fld_elem(table), [i___1]) and element(fld_elem(table), [
          i___1]) <= 1000) .
H2:    fld_size(table) >= 0 .
H3:    fld_size(table) <= 100 .
H4:    value >= - 1000 .
H5:    value <= 1000 .
H6:    0 < find(table, value) .
H7:    find(table, value) <= 100 .
       ->
C1:    find(table, value) <= fld_size(table) - 1 -> fld_size(table) >= 2 and
          fld_size(table) <= 101 .
C2:    fld_size(table) >= 1 .
```

This VC is concerned with the expression J in Index_Type range I .. Table.Size - 1. Null ranges, for example 2 .. 1 are allowed, thus conclusions `unsimp-C1` and `unsimp-C2` have conditions that I is greater than or equal to Table.Size - 1 to describe the non-null range situation. Where the range is non-null it is checked that I and Table.Size - 1 are in the range of Index_Type, in `unsimp-C2` and `unsimp-C1` respectively. It is also checked that I and Table.Size - 1 are in the range of Base_Index_Type, the type from which Index_Type is a subtype, in `unsimp-C5`, `unsimp-C6` and `unsimp-C3`, `unsimp-C4` respectively.

Note some interesting SPARK semantics here. Where the range is not null, it must be the case that the expressions specifying the range, here I and Table.Size - 1, are within the type of the range type, here Index_Type, and its base type, here Base_Index_Type. However, if the range is null, then the values need only be within the range of the base type of the range type. This is relied upon in the case where there is a single entry in the table. The range is I .. Table.Size - 1, as I is in the table it must be 1, and the table size is 1, so the range is 1 .. 0. The value 0 is not inside Index_Type and thus would raise an exception, where it not that the range 1 .. 0 is null. If this range was reversed to 0 .. 1 this would have been an exception.

2.1.3 Proving `simp-C1`

Conclusion `simp-C1` is tackled. The key details of the problem to be proved:

```
[H1a: fld_size(table) <= 100,
 H1b: fld_size(table) >= 0
 H1c: not (find(table, value) = 0)
 ...]
->
C1a: find(table, value) <= fld_size(table) - 1 ->
     fld_size(table) >= 2 and
     fld_size(table) <= 101
```

The first step is to move the left side of implication in `Ca` into the hypotheses. The conclusion conjunction is split into its two conjuncts.

```
[H2a: fld_size(table) <= 100
 H2b: fld_size(table) => 0
 H2c: not (find(table, value) = 0)
 H2d: find(table, value) <= fld_size(table) - 1
 ...]
->
C2b: fld_size(table) => 2
C2c: fld_size(table) <= 101
```

Hypothesis H2d is from the left side of the implication in conclusion C1a, describing the range. If this is null, or *false* in the proof context, then the conclusions can be shown directly. Although null ranges are valid SPARK, finding a null range in the general case indicates unreachable code, which might be useful information to present to the programmer. Trying to prove that this is null might be performed as a proactive non-theorem proof step. In this case such a proof can not readily be attempted due to having little information on *find(table, value)* or *fld_size(table)*.

Looking at C2c

Conclusion C2c is true given hypotheses H2a. The problem is achieving this proof in the planner. Non strict inequalities are syntactic sugar for a disjunction of a strict inequality and equality. A case split on hypotheses H2a can be performed, leading to two straight forward subgoals:

```
[H3a: fld_size(table) = 100
 ...]
->
C3a: fld_size(table) <= 101
```

This can be proved in the planner using H3a to replace all occurrences of *fld_size(table)* with the constant *100*. Such an operation is common in SPARK proofs. The remaining conclusion *100<=101* is trivially true.

```
[H4a: fld_size(table) < 100
 ...]
->
C4a: fld_size(table) <= 101
```

Here the proof will rely on the rules available. It may be more straight forward to reduce C4a to its disjunction *fld_size(table) = 101 or fld_size(table) < 101* and use a rule to show that the second disjunct follows from H4a. Alternatively, a rule which allows C4a to be rewritten into H4a might exist.

Looking at C2b

While the subtraction of one made C2c outside the legal range, and thus straight forward to prove, it pushes C2b deeper within the legal range.

```
[H5a: fld_size(table) <= 100
 H5b: fld_size(table) => 0
 H5c: find(table, value) >= 0
 H5d: find(table, value) <= 100
 H5e: not (find(table, value) = 0)
 H5f: find(table, value) <= fld_size(table) - 1
 H5g: 0 < find(table, value)
 ...]
->
C5b: fld_size(table) => 2
```

Combining H5c and H5e gives the hypothesis H5g. The SIMPLIFIER discovered this, and would be available to NuSPADE. However, in general, this strengthening of non strict inequalities using negation of the equality may be a frequent and useful operation.

Similar to proving C2c a case split is used. The hypothesis H5f gives two subgoals:

```
[H6a: find(table, value) = fld_size(table) - 1
 H6b: find(table, value) >= 0
 H6c: find(table, value) <= 100
 H6d: 0 < find(table, value)
 H6e: find(table, value) + 1 = fld_size(table)
 ...]
->
C6a: fld_size(table) => 2
```

From H6a the extra hypothesis H6e can be found. This can then be used as a rewrite rule on conclusion C6a to get *find(table, value) + 1 => 2* which can be reduced to *find(table, value) => 1* or *find(table, value) > 0* which matches with H6d.

```
[H7a: find(table, value) < fld_size(table) - 1
 H7b: find(table, value) >= 0
 H7c: find(table, value) <= 100
 H7d: 0 < find(table, value)
 H7e: 0 < fld_size(table) - 1
 ...]
->
C7a: fld_size(table) => 2
```

Combining H7a and H7d it can be found that H7e holds. This can be reduced to *1 < fld_size(table)* or *2 <= fld_size(table)* which matches with the conclusion.

Observations

• **Is provable, but not obvious -** In the process of tackling this proof, the opinion of its truth changed twice. It is not particularly difficult, however its true nature is far from obvious.

• SIMPLIFIER **did not get this -** The VC is not automatically discharged by the SIMPLIFIER. Although the majority of the proof effort is concerned with inequalities, there are some case splits that may be beyond the power of the SIMPLIFIER. This is a proof that NuSPADE will tackle.

• **Require inequality reasoning -** A large amount of inequality reasoning is used during the proof, especially reasoning about the extreme ends of such inequalities.

• **Non-strict inequalities are syntactic sugar -** Non strict inequalities are syntactic sugar for a disjunction of strict inequalities with equality. Working with the true form may reduce the number of rules required.

• **Forward and backward proof planning -** This is not an induction proof, so rippling is not a suitable reduction ordering strategy. Another reduction ordering will be required. Further, a strictly backwards proof seems unlikely to be sufficient here. In combining forwards and backwards reasoning, it may become necessary to more thoroughly consider the polarity of expressions.

• **Choosing the case split -** Much of the inequality reasoning in the proof effort is already implemented in the SIMPLIFIER and should be directly available as part of the NuSPADE planner. The tricky part of this proof is selecting the case splits. In both `C2c` and `C2b` the case splits are on inequalities that repeat the variable in the conclusion. Further, the case split inequalities match the conclusion inequalities where rearranging to make the common variable on the same side. Considering inequalities and looking for common variables and syntax seem a reasonable part of a strategy for finding case splits.

2.1.4 Proving `simp-C2`

The remaining conclusion `simp-C2` is considered.

```
[H8a: fld_size(table) >= 0
 H8b: fld_size(table) <= 100
 H8c: find(table, value) >= 0
 H8d: find(table, value) <= 100
 H8e: not (find(table, value) = 0)
 ...]
->
C8a: fld_size(table) >= 1 .
```

It becomes apparent that information is missing. All that is known about *`fld_size(table)`* is that it is in the range *`0`* to *`100`* inclusive. It could be discovered that the information that would lead to proof is *`not (fld_size(table) = 0)`*.

It is known by the programmer that if a value was found in *`table`*, it must be the case that the number of items currently in *`table`* is not *`0`*. In the SPARK discussion of this example, this is classified as *design information* of the function Find. A return annotation is added to this function to make this information available. Note that from the function perspective it is not known whether not not a value was found in *`table`*, thus the return annotation is conditional:

```
--#return K => K/=0 -> Table.Size /= 0;
```

Finding such a good postcondition likely to require programmer intervention. However, it is known that the information *`not (fld_size(table) = 0)`* is needed prior to this check inside *`Delete`*. Examining the code it can be seen that there are only two points where this might be provided, as a precondition to *`Delete`* or as a postcondition of *`Find`*. It is possible to present the programmer with a message of the form:

```
Require information: not (fld_size(table) = 0)
Prior to Delete run time check at line 113

This can be added through:
+Delete precondition
+Find postcondition
```

This is not ideal. In general SPARK code subjected to exception freedom proofs will have no annotation. There may be a number of occasions where design information is not available such that programmer intervention is necessary. An alternative strategy is to automatically describe the operation of programs to try and capture the information in the design annotations. Although it is not expected to generate the concise design annotations seen above, sufficient information may be gathered that allows this to be inferred.

Observations

• **Type information is design information -** It is noted that adding type information in assertions is essentially adding weak design information. The design information is only available as it has been encoded into relevant SPARK types.

• **Strengthen specifications for calling proof -** Proving Find may reveal the true nature of its operation, and suggest the extra design information required. As a general rule it may be worthwhile to strength code beyond the minimum necessary for proof. Adding preconditions and postconditions beyond the needs of proving the local procedure to help prove procedures that call this procedure.

• **Knowing what is missing -** It was assumed that the knowledge of what information is missing (*`not (fld_size(table) = 0)`*) could be extracted from the VC:

```
[H9a: fld_size(table) >= 0
 H9b: fld_size(table) <= 100
 ...]
->
C9a: fld_size(table) >= 1 .
```

This is not obvious in general. A number of different strategies might be considered to find this information.

Regular proof planning - Following the proof patching approach, the proof is attempted, and the failure should identify the missing information. This will require some detailed inequality reasoning.

Constraint solver - An alternative is to use a constraint solver. Conditional constraints present difficulties in constraint solvers. These would be in this example if the full information were made available, in particular making the proved conclusions available as hypothesis. It might be possible to determine that the above is the essential details. However, this would require ruling out the deceptively useful:

```
find(table, value) <= fld_size(table) - 1 ->
    fld_size(table) >= 2 and
    fld_size(table) <= 101 .
```

If this is not ruled out, then more unnecessary information, for example describing *fld_size(table)*, might be introduced.

Adding the entailment request for C9a presents more problems. This will require a sufficiently advanced constraint solver. It seems that the SICSTUS module for constraint logic programming over finite domains, `library(clpfd)`, supports this through a process called reification. The inequalities support reification, though with some distinctions between the domain and and intervals that may require a clever presentation of the problem to gain the results required. In general, constraint solver seem more useful in dealing with special cases, such as reasoning about indicies in arrays or very constrained numerical reasoning. This particular example can be successfully analysed using the SICSTUS constraint solver:

To begin with the domain range is determined. It is more efficient, and practical, to determine the domain range of variables prior to the entailment test:

```
| ?- SIZE #>=0 #/\ SIZE #=<100.

SIZE in 0..100 ?

yes
```

This range is then fed to the constraint solver with the entailment request:

```
| ?- SIZE in 0..100,
     SIZE#>=1 #<=>B.

SIZE in 0..100,
B in 0..1 ?

yes
```

It is not the case that B is always true (1). The entailment is negated to find the cases where B is 0.

```
| ?- SIZE in 0..100,
     (#\ SIZE#>=1).

SIZE = 0 ?

yes
```

The constraint solver has successfully found that a proof is not possible, and that the only sticking case is where SIZE is 0. This is a very valuable result, illustrating the power constraints solvers might present if appropriately integrated within NuSPADE.

Exhaustive proof planning - The proof patching approach can be used to provide a a very powerful constraint solver. A case split could be performed on every possible value for *fld_size(table)*. This would result in 101 subgoals for *0* through to *100*. In all cases but *fld_size(table)=0* the proof should be found using straight forward proof planning. The unproved case is the missing information. This crude, but viable, approach gives the general power of a constraint solver, without the fussy details often seen in such systems[1].

Note that the combitorial explosion, efficiently handled by constraint solvers, would prevent more complex case splits from being considered. If the conclusion contained two variables in the range 0 to 100, this would result in 101*101=10201 different subgoals. This is small to a constraint solver, but may be well outside the the scope of regular proof planning.

Decision procedures - Another option is to use a decision procedure for a subset of the logic. One such procedure is available in the MONA system [2]. This implements decision procedures for the weak second-order theory of one or two successors (WS1S and WS2S). Like for constraint solvers, it may be tricky to suitably encode the problem. For example, in MONA there can only be constants on the right hand side of addition.

A possible encoding of the problem in MONA was attempted. MONA supports first order variables and second order variables as finite sets. A set could hold all of the values dictated by the hypothesis which can then be tested against the conclusion. Note that this is probably far from the most appropriate or efficient MONA program. However, it should serve to convey the general idea:

```
#Declare variables.
var2 HYP;

#Describe hypothesis as set of values.
(all1 inhyp:
    (inhyp >= 0 &
     inhyp <= 100) =>

    inhyp in HYP);

#Test to see if all items in this set
#meets the conclusion.
(all1 inconc:
    inconc in HYP =>

    (inconc >= 1));
```

After being presented with this MONA returns[2]:

[1] In general, the constraints in NuSPADE problems will be first order formulae. This is not the target domain of constraint solvers.
[2] For the curious, an interactive demo of MONA is available on-line at: `http://www.brics.dk/mona/interactive/`.

```
MONA v1.3 for WS1S/WS2S
Copyright (C) 1997-1998 BRICS

PARSING
Time: 00:00:00.01

CODE GENERATION
DAG hits: 15, misses: 29, nodes: 29
Time: 00:00:00.03

AUTOMATON CONSTRUCTION

100% completed
Time: 00:00:02.07

Conjunctive automaton has 2 states,
1 BDD-node and 2 transitions

ANALYSIS
Formula is unsatisfiable

A counter-example (for assertion => main)
of least length (0) is:
HYP             X

HYP = {}

Total time: 00:00:02.12
```

The MONA system has solved that the problem is not possible. It presents the empty set as a counter example, rather than 0^3. Adding this missing information will show that this becomes satisfiable.

```
#Declare variables.
var2 HYP;

#Describe hypothesis as set of values.
(all1 inhyp:
    (inhyp >= 0 &
     inhyp <= 100 &
     inhyp ~= 0) =>

    inhyp in HYP);

#Test to see if all items in this set
#meets the conclusion.
(all1 inconc:
    inconc in HYP =>

    (inconc >= 1));
```

MONA now finds a satisfying example, returning the hypothesis set.

```
MONA v1.3 for WS1S/WS2S
Copyright (C) 1997-1998 BRICS

PARSING
Time: 00:00:00.02

CODE GENERATION
DAG hits: 18, misses: 35, nodes: 35
Time: 00:00:00.01

AUTOMATON CONSTRUCTION

100% completed
Time: 00:00:02.95

Conjunctive automaton has 104 states,
204 BDD-nodes and 205 transitions

ANALYSIS
A counter-example (for assertion => main)
of least length (0) is:
HYP             X

HYP = {}

A satisfying example (for assertion & main)
of least length (101) is:
HYP             X 01111111111111111111111111
11111111111111111111111111111111111111111111
1111111111111111111111111111111

HYP = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,
47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,
62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,
77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,
92,93,94,95,96,97,98,99,100}

Total time: 00:00:03.01
```

- **Patching find -** Performing fairly sophisticated bottom-up analysis, the following return statement might be generated to describe the postcondition of function Find:

```
--#return Result =>
--#    ((Result=0) ->
--#        (Result in Base_Index_Type))
--#    and
--#    ((Result/=0) ->
--#        (Result in 1 .. Table.Size))
--#    and
--#    ((Result=0) or (Result/=0));
```

If the condition in the for loop Table.Elem(I) = Target is true then I must be in the range of array Table.Elem, Index_Type. Stronger still, in the context of the for loop, I must be in the range 1 .. Table.Size. The value of I is then stored in Result, thus it must be the case that

[3] It is expected that with deeper understanding of MONA behaviour the unusual empty set case could be pruned, resulting in the the 0 counter example being presented instead.

Result is also in the range of I. The value of Result is initially 0 of type Base_Index_Type. If assigned a value, it is known that result is not 0. The final ((Result=0) or (Result/=0)) ensures that all possible situations are covered.

The principles here are sound, however the detailed knowledge on Result suggests very good, and perhaps too expensive, bottom-up analysis. The following might be more likely:

```
--#return Result =>
--#    (Table.Size in Base_Index_Type)
--#    and
--#    (for all I in Index_Type =>
--#        (Table.Elem(I) in Contents_Type))
--#    and
--#    Target in Contents_Type
--#    and
--#    Result in Base_Index_Type
--#    and
--#    ((Table.Elem(Result) = Target) ->
--#        (Result in 1 .. Table.Size));
```

This includes all of the type information, in a SPARK friendly form[4], plus the extra information known from performing the assignment within the for loop.

To aid analysis, the function Find is rewritten as a procedure, and the annotation changed accordingly. Using functions in SPARK requires the generation of SPADE rules to propagate the extra information. In using procedures this propagation is automatic. This aids analysis here.

See Appendix C for the procedure find version of the table utilities example. The postcondition corresponding to the return annotation introduced earlier is below:

```
--#post
--#    (Table.Size in Base_Index_Type)
--#    and
--#    (for all I in Index_Type =>
--#        (Table.Elem(I) in Contents_Type))
--#    and
--#    Target in Contents_Type
--#    and
--#    Result in Base_Index_Type
--#    and
--#    ((Table.Elem(Result) = Target) ->
--#        (Result in 1 .. Table.Size));
```

• **Powerful bottom-up -** An even stronger assertion can be made about Find by performing very advanced bottom-up analysis:

```
--#return ((Result=0) ->
--#           (Result in Base_Index_Type and
--#           (for all I in Index_Type
--#            range 1 .. Table.Size =>
--#            (Table.Elem(I) /= Target))))
--#       and
--#       ((Result/=0) ->
--#           ((Result in 1 .. Table.Size) and
--#           (Table.Elem(Result) = Target)))
--#       and
--#       Result in Base_Index_Type;
```

Where Result is 0, it is known that the condition in the for loop Table.Elem(I) = Target is never entered. Due to the placement of the exit statement in the condition it is known that if Result is not zero then Table.Elem(Result) = Target is true. This almost repeats the true Find annotation:

```
--  returns Key s.t.
--      Key/=0 ->
--          Table.Elem(Key)=Target
--      and
--      Key=0  ->
--          for all i in 1..Table.Size,
--          Table.Elem(i) /= Target;
```

This is rather surprising. The bottom-up analysis on Find describes the operation of an algorithm. This also matches very closely to the formal specification, illustrating the potential power of bottom-up techniques.

Note that implementing this level of bottom-up analysis would be very difficult in the general case. And even if possible, extracting a specification from the code is at odds with the process being conduced. Buggy code might be proved correct against a buggy specification.

• **Addressing functions -** The following presents an attempt at the appropriate extra rule required for specifying the operation of Find were it kept as a function.

```
rule_family find:
        X and Y        requires [X:any, Y:any] &
        Table          requires [Table:any]    &
        Target         requires [Target:any].

find(1):
find(Table, Target) may_be_replaced_by
fld_size(Table) >= base_index_type__first and
fld_size(Table) <= base_index_type__last  and
for_all(ii: base_index_type,
    ((ii >= index_type__first)            and
    (ii <= index_type__last)) ->
    ((element(fld_elem(Table), [ii]) >=
         contents_type__first) and
     (element(fld_elem(Table), [ii]) <=
         contents_type__last)))           and
Target >= contents_type__first            and
Target <= contents_type__last             and
find(Table, Target) >=
    base_index_type__first                and
find(Table, Target) <=
    base_index_type__last                 and
```

[4] If generated automatically it may be more straight forward to replace the X in Y notation with the more general X >= Y'First and X<=Y'Last.

```
(element(fld_elem(Table),
    [find(Table, Target)]) = Target) ->
    ((find(Table, Target) >= 1) and
     (find(Table, Target) <=
          fld_size(Table)))             .
```

Unfortunately, there are major SPADE obstacles here. Essentially, to SPADE, `find(Table, Target)` is a function that returns a value of type integer. It is defined in `delete.fdl` as:

```
function find(table_type, integer) : integer;
```

The type of the above replacement for this function is Boolean, preventing the substitution. If the function did return a Boolean type, then the extra information could be brought in this form. A similar strategy is used in [5] to add range constraints to an ordered function. Even if the type constraints are overcome the lack of a variable to describe the function result makes for a recursive definition. A new variable might be added through fdl and an equality with this variable established in a rule:

```
FDL:
var findresult : integer;

RUL:
find(1):
find(Table, Target) may_be_replaced_by
findresult.
```

Alternatively, the extra information may be brought in indirectly through a SPADE `may_be_deduced` rule. The first step of these proofs would be to infer the extra information. The function call can serve as the identifier for the function result. Care will need to be taken to ensure that these rules are used in the correct context. Further, there may still be issues in dealing with the SPADE type system[5].

As functions are a key feature of the SPARK language, a suitable supporting mechanism will be required. In a positive view, functions present access to the more flexible assertion language inside SPARK and are applicable to both functions and procedures[6]. It may be beneficial to treat procedures more like functions, rather than functions more like procedures.

2.1.5 Describing VC 4

Location information

```
For path(s) from start to run-time check associated with statement of line 113:

procedure_delete_4.
```

Code snippet

```
107     begin
108         I := Find(Table, Value);
109         if I = 0 then
110             Success := false;
111         else
112             Success := true;
113             for J in Index_Type range I .. Table.Size - 1 loop
114                 Table.Elem(J) := Table.Elem(J+1);
```

Before simplification

```
H1:    true .
H2:     for_all (i___1: base_index_type, ((i___1 >=
           index_type__first) and (i___1 <= index_type__last)) -> ((element(
           fld_elem(table), [i___1]) >= contents_type__first) and (element(
           fld_elem(table), [i___1]) <= contents_type__last))) .
H3:    fld_size(table) >= base_index_type__first .
H4:    fld_size(table) <= base_index_type__last .
H5:    value >= contents_type__first .
H6:    value <= contents_type__last .
H7:    value >= contents_type__first .
H8:    value <= contents_type__last .
H9:    find(table, value) >= base_index_type__first .
H10:   find(table, value) <= base_index_type__last .
H11:   find(table, value) >= base_index_type__first .
H12:   find(table, value) <= base_index_type__last .
H13:   not (find(table, value) = 0) .
H14:   (find(table, value) <= fld_size(table) - 1) -> ((fld_size(
           table) - 1 >= index_type__first) and (fld_size(
           table) - 1 <= index_type__last)) .
H15:   (find(table, value) <= fld_size(table) - 1) -> ((
           find(table, value) >= index_type__first) and (
           find(table, value) <= index_type__last)) .
H16:   fld_size(table) - 1 >= base_index_type__first .
H17:   fld_size(table) - 1 <= base_index_type__last .
H18:   find(table, value) >= base_index_type__first .
H19:   find(table, value) <= base_index_type__last .
        ->
C1:    fld_size(table) - 1 >= base_index_type__base__first .
C2:    fld_size(table) - 1 <= base_index_type__base__last .
```

After simplification

```
H1:    for_all(i___1 : base_index_type, 1 <= i___1 and i___1 <= 100 -> - 1000
          <= element(fld_elem(table), [i___1]) and element(fld_elem(table), [
          i___1]) <= 1000) .
H2:    fld_size(table) <= 100 .
H3:    value >= - 1000 .
H4:    value <= 1000 .
H5:    0 < find(table, value) .
H6:    find(table, value) <= 100 .
H7:    find(table, value) <= fld_size(table) - 1 -> fld_size(table) >= 2 and
          fld_size(table) <= 101 .
H8:    find(table, value) <= fld_size(table) - 1 -> find(table, value) >= 1 and
          find(table, value) <= 100 .
H9:    fld_size(table) >= 1 .
       ->
C1:    fld_size(table) - 1 >= base_index_type__base__first .
C2:    fld_size(table) - 1 <= base_index_type__base__last .
```

VC shows

This shows that the expression Table.Size - 1 returns a value that does not result in an overflow. There is no information about this type, as the expression is performed while determining the range, which does not have a named type in SPARK.

2.1.6 Proving `simp-C1` and `simp-C2`

This can not be proved, as there is no information available on *`base_index_type__base__last`*. Adding this information, assuming it is within the range 0 to 99, is sufficient to complete the proof.

2.1.7 Describing VC 7

Location information

```
For path(s) from default assertion of line 113 to run-time check associated
         with statement of line 114:

procedure_delete_7.
```

Code snippet

```
112             Success := true;
113             for J in Index_Type range I .. Table.Size - 1 loop
114                 Table.Elem(J) := Table.Elem(J+1);
115             end loop;
```

[5] Correctly finding the type headers that appear at the top of rule declarations is not likely to be straight forward.

[6] The precondition and postcondition of a procedure can be described using functions in SPARK annotations that refer to proof rules in SPADE. Using Boolean functions, these can serve as macros allowing the introduction of a complete specification in SPADE rule notation.

Before simplification

```
H1:     for_all (i___1: base_index_type, ((i___1 >=
           index_type__first) and (i___1 <= index_type__last)) -> ((element(
           fld_elem(table), [i___1]) >= contents_type__first) and (element(
           fld_elem(table), [i___1]) <= contents_type__last))) .
H2:    fld_size(table) >= base_index_type__first .
H3:    fld_size(table) <= base_index_type__last .
H4:    value >= contents_type__first .
H5:    value <= contents_type__last .
H6:    loop__1__j >= index_type__first .
H7:    loop__1__j <= index_type__last .
        ->
C1:    element(fld_elem(table), [loop__1__j + 1]) >=
           contents_type__first .
C2:    element(fld_elem(table), [loop__1__j + 1]) <=
           contents_type__last .
C3:    loop__1__j + 1 >= index_type__first .
C4:    loop__1__j + 1 <= index_type__last .
C5:    loop__1__j + 1 >= base_index_type__base__first .
C6:    loop__1__j + 1 <= base_index_type__base__last .
C7:    loop__1__j >= index_type__first .
C8:    loop__1__j <= index_type__last .
```

After simplification

```
H1:    for_all(i___1 : base_index_type, 1 <= i___1 and i___1 <= 100 -> - 1000
          <= element(fld_elem(table), [i___1]) and element(fld_elem(table), [
          i___1]) <= 1000) .
H2:    fld_size(table) >= 0 .
H3:    fld_size(table) <= 100 .
H4:    value >= - 1000 .
H5:    value <= 1000 .
H6:    loop__1__j >= 1 .
H7:    loop__1__j <= 100 .
       ->
C1:    element(fld_elem(table), [loop__1__j + 1]) >= - 1000 .
C2:    element(fld_elem(table), [loop__1__j + 1]) <= 1000 .
C3:    loop__1__j <= 99 .
C4:    loop__1__j + 1 >= base_index_type__base__first .
C5:    loop__1__j + 1 <= base_index_type__base__last .
```

VC shows

This shows that in the expression Table.Elem(J) := Table.Elem(J+1), both J and J+1 are within the valid range for array Table.Elem, of index type Index_Type. It also shows that the value assigned to *`Table.Elem(J)`* is within the valid range of the items in array Table.Elem, of element type Contents_Type. There are also overflow checks for the expression J+1.

2.1.8 Proving `simp-C1`

Information available about the array Table.Elem has been provided by the SPARK inserted invariant. However, this is not all of the type information available. Type information on the range of J should be added.

```
--#assert J in Index_Type and
--#       J in I .. Table.Size - 1;
```

2.2 Failure analysis 2

2.2.1 Situation

#	From	To	-----Proved In----- vcg	siv	plg	TO DO
1	start	rtc check @ 110		YES		
2	start	rtc check @ 110		YES		
3	start	rtc check @ 115				YES
4	start	rtc check @ 115				YES
5	start	assert @ 116		YES		
6	116	assert @ 116		YES		
7	116	rtc check @ 118				YES
8	start	rtc check @ 120		YES		
9	116	rtc check @ 120		YES		
10	start	assert @ finish	YES			
11	start	assert @ finish	YES			
12	116	assert @ finish	YES			

2.2.2 Describing VC 3

Location information

```
For path(s) from start to run-time check associated with
statement of line 115:

procedure_delete_3.
```

Code snippet

```
109     begin
110         Find(Table, Value, I);
111         if I = 0 then
112             Success := false;
113         else
114             Success := true;
115             for J in Index_Type range I .. Table.Size - 1 loop
116                 --#assert J in Index_Type and
117                 --#       J in I .. Table.Size - 1;
```

Before simplification

```
H1:    true .
H2:     for_all (i___1: base_index_type, ((i___1 >=
           index_type__first) and (i___1 <= index_type__last)) -> ((element(
           fld_elem(table), [i___1]) >= contents_type__first) and (element(
           fld_elem(table), [i___1]) <= contents_type__last))) .
H3:    fld_size(table) >= base_index_type__first .
H4:    fld_size(table) <= base_index_type__last .
H5:    value >= contents_type__first .
H6:    value <= contents_type__last .
H7:    value >= contents_type__first .
H8:    value <= contents_type__last .
H9:    fld_size(table) >= base_index_type__first .
H10:   fld_size(table) <= base_index_type__last .
H11:    for_all (i_: base_index_type, ((i_ >= index_type__first) and (
           i_ <= index_type__last)) -> ((element(fld_elem(
           table), [i_]) >= contents_type__first) and (element(fld_elem(
           table), [i_]) <= contents_type__last))) .
H12:   value >= contents_type__first .
H13:   value <= contents_type__last .
H14:   i__1 >= base_index_type__first .
H15:   i__1 <= base_index_type__last .
H16:   (element(fld_elem(table), [i__1]) = value) -> ((i__1 >= 1) and (
           i__1 <= fld_size(table))) .
H17:   i__1 >= base_index_type__first .
H18:   i__1 <= base_index_type__last .
H19:   i__1 >= base_index_type__first .
H20:   i__1 <= base_index_type__last .
H21:   not (i__1 = 0) .
        ->
C1:    (i__1 <= fld_size(table) - 1) -> ((fld_size(table) - 1 >=
           index_type__first) and (fld_size(table) - 1 <=
           index_type__last)) .
C2:    (i__1 <= fld_size(table) - 1) -> ((i__1 >=
           index_type__first) and (i__1 <= index_type__last)) .
C3:    fld_size(table) - 1 >= base_index_type__first .
C4:    fld_size(table) - 1 <= base_index_type__last .
C5:    i__1 >= base_index_type__first .
C6:    i__1 <= base_index_type__last .
```

After simplification

```
H1:    for_all(i___1 : base_index_type, 1 <= i___1 and i___1 <= 100 -> - 1000
          <= element(fld_elem(table), [i___1]) and element(fld_elem(table), [
          i___1]) <= 1000) .
H2:    fld_size(table) >= 0 .
H3:    fld_size(table) <= 100 .
H4:    value >= - 1000 .
H5:    value <= 1000 .
H6:    for_all(i_ : base_index_type, 1 <= i_ and i_ <= 100 -> - 1000 <= element(
          fld_elem(table), [i_]) and element(fld_elem(table), [i_]) <= 1000) .
H7:    0 < i__1 .
H8:    i__1 <= 100 .
H9:    element(fld_elem(table), [i__1]) = value -> i__1 >= 1 and i__1 <=
          fld_size(table) .
       ->
C1:    i__1 <= fld_size(table) - 1 -> fld_size(table) >= 2 and fld_size(table)
          <= 101 .
C2:    fld_size(table) >= 1 .
```

2.2.3 Proving `simp-C2`

A proof of conclusion `simp-C1` has already been presented. `simp-C1` is tackled again with the added information.

The key to achieving proof is to exploit the extra hypothesis `simp-H9`. Unfortunately, this requires demonstrating that *`element(fld_elem(table), [i__1]) = value`*. From `simp-H1` it can be found that *`element(fld_elem(table), [i__1])`* for any *`i__1`* is in the range of *`value`*, but no more. The added information is not sufficiently strong.

An extra observation can be made about the bottom-up analysis that led to the currently available `simp-H9` and others. If the procedure Find is free from exceptions, then each statement that might raise an exception must be within its legal bounds. In particular, in merely evaluating the expression Table.Elem(I) = Target it must be the case that I is in the valid index range of the array, and that Target is in the valid range of values for this array. This information is available, even if the expression evaluates to false. Looking at the code for Find:

```
procedure Find(Table: in Table_Type;
               Target: in Contents_Type;
               Result: out Base_Index_Type)
is
begin
   Result:= 0;
   for I in Index_Type
         range 1 .. Table.Size loop
      if Table.Elem(I) = Target then
         Result := I;
         exit;
      end if;
   end loop;
end Find;
```

The if statement is evaluated so long as the for loop is executed at least once, which requires that 1 <= Table.Size. Thus the following might be included in a postcondition:

```
--#post
--#    ...
--#    ((1 <= Table.Size) ->
--#          (1          in Table.Elem'Range and
--#           Table.Size in Table.Elem'Range and
--#           (for all J in 1 .. Table.Size =>
--#                (Target in Contents_Type))))
--#
--#    ...
```

There are some problems here. Firstly, this postcondition need not necessarily be true. Where the for loop contains an exit statement, it is possible that the whole range of the for is never performed. For example, the for loop in find could range over 1 .. 100000, and never raise an exception, so long as a value is found inside the legal bounds of the array. Either an heuristic decision is made that for loops always iterate over their full range, or some analysis of exit statements are performed.

Another problem is that scope may prevent the postcondition discussing internal details of the procedure. Here the for range 1 to Table.Size the array Table.Elem and the comparison Target are all visible outside the scope of the procedure. However, this need not be the case in general. Where scope restricts entries in the postcondition existential quantification may be used to introduce and describe such private information.

Yet another problem is that at the point of the postcondition many variables may have changed. It is necessary to describe the situation in respect to the values at that point. For example, if Find performed the statement Table.Size:=Table.Size + 1; after the for loop the postcondition might add:

```
--#post
--#    ...
--#    Table.Size=Table.Size~ + 1
--#    ...
```

It is now apparent that Table.Size in the previous postcondition was actually the initial value, Table.Size˜, subjected to null changes. Were the statement Table.Size:=Table.Size + 1; after before for loop the postcondition might again change to:

```
--#post
--#    ...
--#    Table.Size=Table.Size~ + 2
--#    ...
```

The value Table.Size encountered at the for is actually the initial value subject to changes, Table.Size + 1 making the postcondition:

```
--#post
--#    ...
--#    Table.Size:=Table.Size~ + 2
--#    and
--#    ((1 <= Table.Size~ + 1) ->
--#          (1                in Table.Elem'Range and
--#           Table.Size~ + 1 in Table.Elem'Range and
--#           (for all J in 1 .. Table.Size~ + 1 =>
--#                (Target in Contents_Type))))
--#    ...
```

Identifying these expressions is quite decidable, however will require the bottom-up analysis to be equipped with knowledge of Floyd-Hoare VC generation. Further, identifying the exception freedom information will require detailed knowledge of exceptions in SPARK. However, this technique in general seems promising for completing exception freedom proofs where additional information beyond type, but less than full partial correctness, is required.

Using this technique the following refined post condition for Find is proposed:

```
--#post
--#    (Table.Size in Base_Index_Type)
--#    and
--#    (for all I in Index_Type =>
--#         (Table.Elem(I) in Contents_Type))
--#    and
--#    Target in Contents_Type
--#    and
--#    Result in Base_Index_Type
--#    and
--#    ((Table.Elem(Result) = Target) ->
--#          (Result in 1 .. Table.Size))
--#    and
--#    ((1 <= Table.Size) ->
```

```
--#          (1           in Table.Elem'Range and
--#           Table.Size in Table.Elem'Range and
--#           (for all J in 1 .. Table.Size =>
--#                (Target in Contents_Type))));
```

This is not sufficient. The constraint that 1 <= Table.Size is too focused on particular variables. It is beneficial to start again, adopting a more principled quantification notation. The code for the for loop is:

```
for I in Index_Type range 1 .. Table.Size loop
    if Table.Elem(I) = Target then
        Result := I;
        exit;
    end if;
end loop;
```

Any information X true inside the for loop can be described as for all I in Index_Type range 1 .. Table.Size => (X). The exit command will be ignored for the moment. There are no assignments always executed in each loop iteration, a popular target for documented bottom-up techniques[7]. However, there is a constant expression being evaluated, Table.Elem(I) = Target. This leads to the exception freedom annotation:

```
(for all I in Index_Type
              range 1 .. Table.Size =>
     (I      in Table.Elem'Range and
      Target in Contents_Type))
```

Unfortunately, in this particular example, such information is not very valuable. As exit is currently being ignored, it is assumed that this is a regular for loop. It may be the case that the expression Table.Elem(I) = Target evaluates to true on one or more occasions, or on no occasions. Where this expression is true, type information can be added about its statements. In this case the statement Result := I allows for exception freedom information that I is in the valid type for Result. This gives:

```
(for all I in Index_Type
              range 1 .. Table.Size =>
     (Table.Elem(I)=Target ->
          (I in Base_Index_Type)))
```

Further information can be added about the operation of these statements. This is not bundled with the above to separate type from operational semantics. The exit is now taken into account. The first occurrence of the expression Table.Elem(I) = Target being true results in the loop exiting. All information extracted from operations after Table.Elem(I) = Target being true and the exit will hold when the loop exists.

```
(for all I in Index_Type
              range 1 .. Table.Size =>
     ((Table.Elem(I)=Target and
       (for all II in Index_Type
                    range 1 .. (I-1) =>
            (not (Table.Elem(II)=Target)))) ->
                 (Result=I)));
```

This describes the first occurrence of Table.Elem(I) = Target, as there being no occurrences previously. This is still not sufficient for proof. All of the information inside the for loop is unlocked with the knowledge that I lies in the range 1 .. Table.Size. Key to showing this in VC 3 is knowing *fld_size(table) >= 1*, the very conclusion being proved. All is not lost, however, as one assignment statement has not been considered. Prior to entering the for loop Result is initialised to 0. The only place Result can change is inside the condition inside the for, as described above. If result is 0 then the if statement is not entered, otherwise it was.

```
(Result=0 ->
     (for all I in Index_Type
                 range 1 .. Table.Size =>
          (Table.Elem(I) /= Target)))
((not (Result=0)) ->
      (for some II in Index_Type
                     range 1 .. Table.Size =>
           (Table.Elem(II) = Target and
            Result=II)));
```

This is now enough information to perform a proof. Note that at no stage was information inserted that required external program knowledge, and more importantly no special intelligent reasoning. Essentially the program has been analysed in three phases of increasing complexity. The first phase adds type information, the second phase adds code information falling with the flow of program execution, then the third phase adds more code information climbing up against the flow of program execution.

Observations

- **Type is constant -** Many of the problems seen in describing information extracted from the code exist as the information is based on modifiable variables. Type does not change during the execution of a program. Thus it is often much more straight forward to extract and present type information.

- **Description functions -** As common description patterns emerge it may be beneficial to use functions to encode these patterns. For example, a "first occurrence in array" function may have been used in this example, rather than the directly nested quantified expressions.

2.2.4 About VC 4 and VC 7

Both VC 4 and VC 7 have been proved, except for those conclusions that require information about *base_index_type__base_last*.

[7] A number of invariant discovery papers are being picked through. A survey of these, in the context of the considered SPARK examples, would make for an valuable note.

2.3 Failure analysis 3

2.3.1 Situation

#	From	To	Proved In vcg	Proved In siv	Proved In plg	TO DO
1	start	rtc check @ 110		YES		
2	start	rtc check @ 110		YES		
3	start	rtc check @ 115				YES
4	start	rtc check @ 115				YES
5	start	assert @ 116		YES		
6	116	assert @ 116		YES		
7	116	rtc check @ 118				YES
8	start	rtc check @ 120		YES		
9	116	rtc check @ 120		YES		
10	start	assert @ finish	YES			
11	start	assert @ finish	YES			
12	116	assert @ finish	YES			

2.3.2 Describing VC 3

Location information

```
For path(s) from start to run-time check associated with
statement of line 115:

procedure_delete_3.
```

Code snippet

```
109    begin
110       Find(Table, Value, I);
111       if I = 0 then
112          Success := false;
113       else
114          Success := true;
115          for J in Index_Type range I .. Table.Size - 1 loop
116             --#assert J in Index_Type and
117             --#      J in I .. Table.Size - 1;
```

Before simplification

```
H1:    true .
H2:     for_all (i___1: base_index_type, ((i___1 >=
          index_type__first) and (i___1 <= index_type__last)) -> ((element(
          fld_elem(table), [i___1]) >= contents_type__first) and (element(
          fld_elem(table), [i___1]) <= contents_type__last))) .
H3:    fld_size(table) >= base_index_type__first .
H4:    fld_size(table) <= base_index_type__last .
H5:    value >= contents_type__first .
H6:    value <= contents_type__last .
H7:    value >= contents_type__first .
H8:    value <= contents_type__last .
H9:    fld_size(table) >= base_index_type__first .
H10:   fld_size(table) <= base_index_type__last .
H11:    for_all (i_: base_index_type, ((i_ >= index_type__first) and (
          i_ <= index_type__last)) -> ((element(fld_elem(
          table), [i_]) >= contents_type__first) and (element(fld_elem(
          table), [i_]) <= contents_type__last))) .
H12:   value >= contents_type__first .
H13:   value <= contents_type__last .
H14:   i__1 >= base_index_type__first .
H15:   i__1 <= base_index_type__last .
H16:    for_all (i_: base_index_type, ((i_ >= 1) and (i_ <= fld_size(
          table))) -> (((i_ >= index_type__first) and (i_ <=
          index_type__last)) and ((value >= contents_type__first) and (
          value <= contents_type__last)))) .
H17:    for_all (i_: base_index_type, ((i_ >= 1) and (i_ <= fld_size(
          table))) -> ((element(fld_elem(table), [i_]) =
          value) -> ((i_ >= base_index_type__first) and (i_ <=
          base_index_type__last)))) .
H18:   (i__1 = 0) -> ( for_all (i_: base_index_type, ((i_ >= 1) and (
          i_ <= fld_size(table))) -> (element(fld_elem(
          table), [i_]) <> value))) .
H19:   (not (i__1 = 0)) -> ( for_some (ii_: base_index_type, ((
          ii_ >= 1) and (ii_ <= fld_size(table))) and ((element(
          fld_elem(table), [ii_]) = value) and (i__1 = ii_)))) .
H20:    for_all (i_: base_index_type, ((i_ >= 1) and (i_ <= fld_size(
          table))) -> (((element(fld_elem(table), [i_]) =
          value) and ( for_all (ii_: base_index_type, ((ii_ >= 1) and (
          ii_ <= i_ - 1)) -> (not (element(fld_elem(table), [
          ii_]) = value))))) -> (i__1 = i_))) .
H21:   i__1 >= base_index_type__first .
H22:   i__1 <= base_index_type__last .
H23:   i__1 >= base_index_type__first .
H24:   i__1 <= base_index_type__last .
H25:   not (i__1 = 0) .
        ->
C1:    (i__1 <= fld_size(table) - 1) -> ((fld_size(table) - 1 >=
          index_type__first) and (fld_size(table) - 1 <=
          index_type__last)) .
C2:    (i__1 <= fld_size(table) - 1) -> ((i__1 >=
          index_type__first) and (i__1 <= index_type__last)) .
C3:    fld_size(table) - 1 >= base_index_type__first .
C4:    fld_size(table) - 1 <= base_index_type__last .
C5:    i__1 >= base_index_type__first .
C6:    i__1 <= base_index_type__last .
```

After simplification

```
H1:    for_all(i___1 : base_index_type, 1 <= i___1 and i___1 <= 100 -> - 1000
         <= element(fld_elem(table), [i___1]) and element(fld_elem(table), [
         i___1]) <= 1000) .
H2:    fld_size(table) >= 0 .
H3:    fld_size(table) <= 100 .
H4:    value >= - 1000 .
H5:    value <= 1000 .
H6:    for_all(i_ : base_index_type, 1 <= i_ and i_ <= 100 -> - 1000 <= element(
         fld_elem(table), [i_]) and element(fld_elem(table), [i_]) <= 1000) .
H7:    0 < i__1 .
H8:    i__1 <= 100 .
H9:    for_all(i_ : base_index_type, 1 <= i_ and i_ <= fld_size(table) -> 1 <=
         i_ and i_ <= 100 and (- 1000 <= value and value <= 1000)) .
H10:   for_all(i_ : base_index_type, 1 <= i_ and i_ <= fld_size(table) and
         element(fld_elem(table), [i_]) = value -> 0 <= i_ and i_ <= 100) .
H11:   i__1 = 0 -> for_all(i_ : base_index_type, 1 <= i_ and i_ <= fld_size(
         table) -> element(fld_elem(table), [i_]) <> value) .
H12:   for_some(ii_ : base_index_type, ii_ >= 1 and ii_ <= fld_size(table) and (
         element(fld_elem(table), [ii_]) = value and i__1 = ii_)) .
H13:   for_all(i_ : base_index_type, 1 <= i_ and i_ <= fld_size(table) and (
         element(fld_elem(table), [i_]) = value and for_all(ii_ :
         base_index_type, 1 <= ii_ and ii_ <= i_ - 1 -> not element(fld_elem(
         table), [ii_]) = value)) -> i__1 = i_) .
       ->
C1:    i__1 <= fld_size(table) - 1 -> fld_size(table) >= 2 and fld_size(table)
         <= 101 .
C2:    fld_size(table) >= 1 .
```

2.3.3 Proving `simp-C2`

With the extra information a proof is now possible. The hypothesis `simp-H12` has been made available by the SIMPLIFIER using result of Find is not zero. Unwrapping this gives:

```
*** New H14:  bas_ii__1 >= 1 and bas_ii__1 <= fld_size(table)
*** New H15:  element(fld_elem(table), [bas_ii__1]) = value and i__1 =
          bas_ii__1
*** New H14:  bas_ii__1 >= 1
*** New H16:  bas_ii__1 <= fld_size(table)
*** New H15:  element(fld_elem(table), [bas_ii__1]) = value
*** New H17:  i__1 = bas_ii__1
```

Together the new hypothesis `simp-H14` and `simp-H16` allow it to be inferred that ***`fld_size(table) >= 1`***, proving the conclusion `simp-C2`. Note that the operation of inferring a new inequality from two inequalities sharing a common variable may be useful in general.

3 Conclusions

The following observations are made:

• **Notation-** In adding additional information to SPARK programs either SPARK annotations or SPADE formula can be used. As SPADE formula is more flexible and the only available annotations for describing called functions it seems the best candidate.

• **Bottom-up explosion -** There are a number of heuristics and guaranteed strategies that can be used to extract information using bottom-up techniques. Comprehensively adding all this information is likely to hinder proof, as the volume of information makes it increasingly difficult to focus on certain aspects and select appropriate actions. To avoid this information explosion some constraints can be applied during the generation of bottom-up information. Trying to patch proof failures may reveal these constraints.

• **Relevance of top-down -** By their very nature, top-down techniques rely on some existing specification before being applicable. Such specifications are not expected on code subjected to SPARK exception freedom proofs. However, top-down techniques may apply once some specifications has been generated using bottom-up techniques. This might lead to a general strategy of proof failure suggesting a constrained bottom-up search which adds specification information available for top-down analysis.

• **Falling down and climbing up -** Falling down heuristics describe what will happen if something is true now. Climbing up heuristics describe what must have happened if something is true now. This view was used to justify the discovery of what happens when not(Result=0). In general this approach seems viable only for shorter code.

• **Using exception freedom -** The problem being tackled is exception freedom. As there is a limited number of exceptions there is a limited number of kinds of problems. Bottom-up analysis could certainly identify what needs to be solved and where. This additional information might be helpful in constraining bottom-up analysis.

• **Exception freedom annotations -** Knowing where exception freedom might occur, allows additional information to be added in code that is know to be free from exceptions. This information may be fairly straightforward to collect. However, it may not be particularly useful.

• **Strengthening postconditions -** Where calling other procedures, a stronger postcondition in this procedure will bring extra information. It might be argued that the low level procedures should be tackled first. However, by tackling them from the perspective of their callers the information required might be known as additional constraints.

• **Is Find provable -** The proof of the Delete procedure now relies on proving the postcondition of Find. It seems likely that the Find should be provable. However, it may be difficult to prove automatically.

• **Practicalities -** Most of the documented effort in automating the generation of extra program annotations do not make any particular hints at implementation. The various heuristics and strategies are often significantly different in operation. In particular they may require combining top-down and bottom-up strategies. A very flexible architecture, surprisingly close to the blackboard model for proof planning, might be required.

It is considered unlikely that the EXAMINER could be naturally modified to provide this flexible architecture. However, the EXAMINER may be able to provide information that is directly applicable to bottom-up analysis. There is a requirement for a SPARK parser, and probably some Floyd-Hoare operations on SPARK code, both of which the EXAMINER is capable of. It is expected that in supporting full SPARK annotations, flow analysis, etc. it may be difficult to isolate these EXAMINER components.

It is noted that SPARK is very well defined and thus should not present a problem to parse. It might even be assumed that the EXAMINER has shown the code to be syntactically correct, reducing the need for any sophisticated error checking. This could be implemented using a suitable parsing tool[8]. Once parsed a suitable structure may be constructed[9] and bottom-up analysis performed on this.

[8]For example, in Prolog using Definite Causal Grammar.

[9]A flow-chart may be ideal.

A Table Utilities Example

A.1 Code

A.1.1 Specification

```
-- Table Utilities for Run-Time Errors Case Study
-- for vc generation with rtc and simplification
-- (Taken from:
-- Proof Of The Absence Of Run-Time Errors from
-- SPARK Programs -- A Case Study)
package Table_Utilities is
    Max_Table_Size: constant := 100;
    type Base_Index_Type is range 0 .. Max_Table_Size;
    subtype Index_Type is Base_Index_Type range 1 .. Max_Table_Size;
     type Contents_Type is range -1000 .. 1000;
     type Array_Type is array (Index_Type) of Contents_Type;
     type Table_Type is record
                         Size: Base_Index_Type;
                         Elem: Array_Type;
                       end record;
     function Find(Table: in Table_Type;
                   Target: in Contents_Type) return Base_Index_Type;
     --  returns Key s.t. Key/=0 -> Table.Elem(Key)=Target and
     --                   Key=0  -> for all i in 1..Table.Size,
     --                                Table.Elem(i) /= Target;

     procedure Sort(Table: in out Table_Type);
     --#derives Table from Table;
     --  sorts the Table.Elem array into ascending numerical order, preserving
     --  contents and size

     procedure Insert(Table: in out Table_Type;
                      Sorted, Unique: in Boolean;
                      Value: in Contents_Type;
                      Success: out Boolean);
     --#derives Table, Success from Table, Sorted, Unique, Value;
     --  inserts Value into the Table.Elem, such that:
     --    - if Sorted is true, inserts at a point in ascending numerical
     --      order, otherwise inserts at end of array;
     --    - if Unique is true, does not create multiple entries, otherwise
     --      will allow more than one entry with same value;
     --    - if there is no room for the insertion (Table.Size=100 and an
     --      insert is needed), does nothing to Table and returns a Success of
     --      False, otherwise returns a Success of True.

     procedure Delete(Table: in out Table_Type;
                      Value: in Contents_Type;
                      Success: out Boolean);
     --#derives Table, Success from Table, Value;
 end Table_Utilities;
```

A.1.2 Body

```
package body Table_Utilities is
    function Find(Table: in Table_Type;
                  Target: in Contents_Type) return Base_Index_Type
    is
       Result: Base_Index_Type := 0;
    begin
       for I in Index_Type range 1 .. Table.Size loop
          if Table.Elem(I) = Target then
             Result := I;
              exit;
          end if;
       end loop;
       return Result;
     end Find;
     -------------------------------------------------------------------------
     procedure Sort(Table: in out Table_Type)
     is
        Key: Index_Type;
        function Find_Smallest(Arr: Array_Type;
                               L, U: Index_Type) return Index_Type
        is
           K: Index_Type;
        begin
           K := L;
           for I in Index_Type range L+1 .. U loop
              if Arr(I) < Arr(K) then
                 K := I;
              end if;
           end loop;
           return K;
        end Find_Smallest;
        procedure Swap_Elements(T: in out Table_Type; I, J: in Index_Type)
        --#derives T from T, I, J;
        is
           Temp: Contents_Type;
        begin
           Temp := T.Elem(I);
           T.Elem(I) := T.Elem(J);
           T.Elem(J) := Temp;
        end Swap_Elements;

     begin  -- Sort
         for Low in Index_Type range 1 .. Table.Size-1 loop
             Key := Find_Smallest(Table.Elem, Low, Table.Size);
             if Key /= Low then
                 Swap_Elements(Table, Low, Key);
             end if;
         end loop;
     end Sort;
     -------------------------------------------------------------------------
     procedure Insert(Table: in out Table_Type;
                      Sorted, Unique: in Boolean;
                      Value: in Contents_Type;
                       Success: out Boolean)
     is
         Successful: Boolean;
         I: Base_Index_Type;
     begin
         if Sorted then
             if Unique and Find(Table, Value) /= 0 then
                 Successful := true;
             else
                 Successful := Table.Size < Max_Table_Size;
                 if Successful then
                     --! Sorted, there's space & insert, so find place and insert
                     I := 1;
                     loop
                         exit when I > Table.Size or else
                                   Table.Elem(I) >= Value;
                         I := I + 1;
                     end loop;
                     for X in reverse Index_Type range I .. Table.Size loop
                         Table.Elem(X+1) := Table.Elem(X);
                     end loop;
                     Table.Elem(I) := Value;
                     Table.Size := Table.Size + 1;
                 end if;
             end if;
         else
             if Unique then
                 --! not Sorted, Unique: insert at end if not already in Table
                 Successful := Find(Table, Value) /= 0;
                 if not Successful then
                     Successful := Table.Size < Max_Table_Size;
                     if Successful then
                         Table.Size := Table.Size + 1;
                         Table.Elem(Table.Size) := Value;
                     end if;
                 end if;
             else
                 --! not Sorted, not Unique: insert regardless, at end
                 Successful := Table.Size < Max_Table_Size;
                 if Successful then
                     Table.Size := Table.Size + 1;
                     Table.Elem(Table.Size) := Value;
                 end if;
             end if;
         end if;
         Success := Successful;
     end Insert;
     -------------------------------------------------------------------------
     procedure Delete(Table: in out Table_Type;
                      Value: in Contents_Type;
                      Success: out Boolean)
     is
         I: Base_Index_Type;
     begin
         I := Find(Table, Value);
         if I = 0 then
             Success := false;
         else
             Success := true;
             for J in Index_Type range I .. Table.Size - 1 loop
                 Table.Elem(J) := Table.Elem(J+1);
             end loop;
             Table.Size := Table.Size - 1;
         end if;
     end Delete;

 end Table_Utilities;
```

B Delete

B.1 Extra Information

B.1.1 RLS File

```
          /*********************************************************/
                          /*Proof Rule Declarations*/
       /*SPARK95 Examiner with VC and RTC Generator Release 6.0 / 11.01*/
                          /*Demonstration Version*/
          /*********************************************************/


                       /*DATE : 19-JUL-2002 15:56:55.49*/

                      /*procedure Table_Utilities.Delete*/


rule_family delete_rules:
     X      requires [X:any] &
     X <= Y requires [X:ire, Y:ire] &
     X >= Y requires [X:ire, Y:ire].

delete_rules(1): character__pos(X) may_be_replaced_by X.
delete_rules(2): character__val(X) may_be_replaced_by X.
delete_rules(3): base_index_type__first may_be_replaced_by 0.
delete_rules(4): base_index_type__last may_be_replaced_by 100.
delete_rules(5): base_index_type__base__first <= base_index_type__base__last may_be_deduced.
delete_rules(6): base_index_type__base__first <= base_index_type__first may_be_deduced.
delete_rules(7): base_index_type__base__last >= base_index_type__last may_be_deduced.
delete_rules(8): index_type__first may_be_replaced_by 1.
delete_rules(9): index_type__last may_be_replaced_by 100.
delete_rules(10): index_type__base__first <= index_type__base__last may_be_deduced.
delete_rules(11): index_type__base__first <= index_type__first may_be_deduced.
delete_rules(12): index_type__base__last >= index_type__last may_be_deduced.
delete_rules(13): contents_type__first may_be_replaced_by -1000.
delete_rules(14): contents_type__last may_be_replaced_by 1000.
delete_rules(15): contents_type__base__first <= contents_type__base__last may_be_deduced.
delete_rules(16): contents_type__base__first <= contents_type__first may_be_deduced.
delete_rules(17): contents_type__base__last >= contents_type__last may_be_deduced.
```

B.1.2 FDL File

```
          {*******************************************************}
                              {FDL Declarations}
```

```
        {SPARK95 Examiner with VC and RTC Generator Release 6.0 / 11.01}
                        {Demonstration Version}
           {*******************************************************}


                      {DATE : 19-JUL-2002 15:56:55.49}

                     {procedure Table_Utilities.Delete}


title procedure delete;

  function round__(real) : integer;
  type contents_type = integer;
  type array_type = array [integer] of integer;
  type table_type = record
        size : integer;
        elem : array_type
     end;
  type base_index_type = integer;
  const base_index_type__base__last : integer = pending;
  const base_index_type__base__first : integer = pending;
  const contents_type__last : integer = pending;
  const contents_type__first : integer = pending;
  const index_type__last : integer = pending;
  const index_type__first : integer = pending;
  const base_index_type__last : integer = pending;
  const base_index_type__first : integer = pending;
  var loop__1__j : integer;
  var value : integer;
  var table : table_type;
  function find(table_type, integer) : integer;

end;
```

B.2 Exception Freedom VCs (Delete)

B.2.1 Unsimplified

```
           *******************************************************
                       Semantic Analysis of SPARK Text
        SPARK95 Examiner with VC and RTC Generator Release 6.0 / 11.01
                            Demonstration Version
           *******************************************************


                       DATE : 19-JUL-2002 15:56:55.49

                      procedure Table_Utilities.Delete




For path(s) from start to run-time check associated with statement of line 108:

procedure_delete_1.
H1:    true .
H2:     for_all (i___1: base_index_type, ((i___1 >=
          index_type__first) and (i___1 <= index_type__last)) -> ((element(
          fld_elem(table), [i___1]) >= contents_type__first) and (element(
          fld_elem(table), [i___1]) <= contents_type__last))) .
H3:    fld_size(table) >= base_index_type__first .
H4:    fld_size(table) <= base_index_type__last .
H5:    value >= contents_type__first .
H6:    value <= contents_type__last .
        ->
C1:    value >= contents_type__first .
C2:    value <= contents_type__last .


For path(s) from start to run-time check associated with statement of line 108:

procedure_delete_2.
H1:    true .
H2:     for_all (i___1: base_index_type, ((i___1 >=
          index_type__first) and (i___1 <= index_type__last)) -> ((element(
          fld_elem(table), [i___1]) >= contents_type__first) and (element(
          fld_elem(table), [i___1]) <= contents_type__last))) .
H3:    fld_size(table) >= base_index_type__first .
H4:    fld_size(table) <= base_index_type__last .
H5:    value >= contents_type__first .
H6:    value <= contents_type__last .
H7:    value >= contents_type__first .
H8:    value <= contents_type__last .
H9:    find(table, value) >= base_index_type__first .
H10:   find(table, value) <= base_index_type__last .
        ->
C1:    find(table, value) >= base_index_type__first .
C2:    find(table, value) <= base_index_type__last .


For path(s) from start to run-time check associated with statement of line 113:

procedure_delete_3.
H1:    true .
H2:     for_all (i___1: base_index_type, ((i___1 >=
          index_type__first) and (i___1 <= index_type__last)) -> ((element(
          fld_elem(table), [i___1]) >= contents_type__first) and (element(
          fld_elem(table), [i___1]) <= contents_type__last))) .
H3:    fld_size(table) >= base_index_type__first .
H4:    fld_size(table) <= base_index_type__last .
H5:    value >= contents_type__first .
H6:    value <= contents_type__last .
H7:    value >= contents_type__first .
H8:    value <= contents_type__last .
H9:    find(table, value) >= base_index_type__first .
H10:   find(table, value) <= base_index_type__last .
H11:   find(table, value) >= base_index_type__first .
H12:   find(table, value) <= base_index_type__last .
H13:   not (find(table, value) = 0) .
        ->
C1:    (find(table, value) <= fld_size(table) - 1) -> ((fld_size(
          table) - 1 >= index_type__first) and (fld_size(
          table) - 1 <= index_type__last)) .
C2:    (find(table, value) <= fld_size(table) - 1) -> ((
          find(table, value) >= index_type__first) and (
          find(table, value) <= index_type__last)) .
C3:    fld_size(table) - 1 >= base_index_type__first .
C4:    fld_size(table) - 1 <= base_index_type__last .
C5:    find(table, value) >= base_index_type__first .
C6:    find(table, value) <= base_index_type__last .


For path(s) from start to run-time check associated with statement of line 113:

procedure_delete_4.
H1:    true .
H2:     for_all (i___1: base_index_type, ((i___1 >=
          index_type__first) and (i___1 <= index_type__last)) -> ((element(
          fld_elem(table), [i___1]) >= contents_type__first) and (element(
          fld_elem(table), [i___1]) <= contents_type__last))) .
H3:    fld_size(table) >= base_index_type__first .
H4:    fld_size(table) <= base_index_type__last .
H5:    value >= contents_type__first .
H6:    value <= contents_type__last .
H7:    value >= contents_type__first .
H8:    value <= contents_type__last .
H9:    find(table, value) >= base_index_type__first .
H10:   find(table, value) <= base_index_type__last .
H11:   find(table, value) >= base_index_type__first .
H12:   find(table, value) <= base_index_type__last .
H13:   not (find(table, value) = 0) .
H14:   (find(table, value) <= fld_size(table) - 1) -> ((fld_size(
          table) - 1 >= index_type__first) and (fld_size(
          table) - 1 <= index_type__last)) .
H15:   (find(table, value) <= fld_size(table) - 1) -> ((
          find(table, value) >= index_type__first) and (
          find(table, value) <= index_type__last)) .
H16:   fld_size(table) - 1 >= base_index_type__first .
H17:   fld_size(table) - 1 <= base_index_type__last .
H18:   find(table, value) >= base_index_type__first .
H19:   find(table, value) <= base_index_type__last .
        ->
C1:    fld_size(table) - 1 >= base_index_type__base__first .
C2:    fld_size(table) - 1 <= base_index_type__base__last .


For path(s) from start to default assertion of line 113:

procedure_delete_5.
H1:    true .
H2:     for_all (i___1: base_index_type, ((i___1 >=
          index_type__first) and (i___1 <= index_type__last)) -> ((element(
          fld_elem(table), [i___1]) >= contents_type__first) and (element(
          fld_elem(table), [i___1]) <= contents_type__last))) .
H3:    fld_size(table) >= base_index_type__first .
H4:    fld_size(table) <= base_index_type__last .
H5:    value >= contents_type__first .
H6:    value <= contents_type__last .
H7:    value >= contents_type__first .
H8:    value <= contents_type__last .
H9:    find(table, value) >= base_index_type__first .
H10:   find(table, value) <= base_index_type__last .
H11:   find(table, value) >= base_index_type__first .
H12:   find(table, value) <= base_index_type__last .
H13:   not (find(table, value) = 0) .
H14:   (find(table, value) <= fld_size(table) - 1) -> ((fld_size(
          table) - 1 >= index_type__first) and (fld_size(
          table) - 1 <= index_type__last)) .
H15:   (find(table, value) <= fld_size(table) - 1) -> ((
          find(table, value) >= index_type__first) and (
          find(table, value) <= index_type__last)) .
H16:   fld_size(table) - 1 >= base_index_type__first .
H17:   fld_size(table) - 1 <= base_index_type__last .
H18:   find(table, value) >= base_index_type__first .
H19:   find(table, value) <= base_index_type__last .
H20:   fld_size(table) - 1 >= base_index_type__base__first .
H21:   fld_size(table) - 1 <= base_index_type__base__last .
H22:   find(table, value) <= fld_size(table) - 1 .
        ->
C1:     for_all (i___1: base_index_type, ((i___1 >=
          index_type__first) and (i___1 <= index_type__last)) -> ((element(
          fld_elem(table), [i___1]) >= contents_type__first) and (element(
          fld_elem(table), [i___1]) <= contents_type__last))) .
C2:    fld_size(table) >= base_index_type__first .
C3:    fld_size(table) <= base_index_type__last .
C4:    value >= contents_type__first .
C5:    value <= contents_type__last .
C6:    find(table, value) >= index_type__first .
C7:    find(table, value) <= index_type__last .


For path(s) from default assertion of line 113 to default assertion of line 113:

procedure_delete_6.
H1:     for_all (i___1: base_index_type, ((i___1 >=
          index_type__first) and (i___1 <= index_type__last)) -> ((element(
          fld_elem(table), [i___1]) >= contents_type__first) and (element(
          fld_elem(table), [i___1]) <= contents_type__last))) .
H2:    fld_size(table) >= base_index_type__first .
H3:    fld_size(table) <= base_index_type__last .
H4:    value >= contents_type__first .
H5:    value <= contents_type__last .
H6:    loop__1__j >= index_type__first .
H7:    loop__1__j <= index_type__last .
H8:    element(fld_elem(table), [loop__1__j + 1]) >=
          contents_type__first .
H9:    element(fld_elem(table), [loop__1__j + 1]) <=
          contents_type__last .
H10:   loop__1__j + 1 >= index_type__first .
H11:   loop__1__j + 1 <= index_type__last .
H12:   loop__1__j + 1 >= base_index_type__base__first .
H13:   loop__1__j + 1 <= base_index_type__base__last .
H14:   loop__1__j >= index_type__first .
H15:   loop__1__j <= index_type__last .
H16:   not (loop__1__j = fld_size(upf_elem(table, update(fld_elem(
          table), [loop__1__j], element(fld_elem(table), [
          loop__1__j + 1])))) - 1) .
        ->
C1:     for_all (i___1: base_index_type, ((i___1 >=
          index_type__first) and (i___1 <= index_type__last)) -> ((element(
```

```
          fld_elem(upf_elem(table, update(fld_elem(table), [
          loop__1__j], element(fld_elem(table), [loop__1__j + 1])))), [
          i___1]) >= contents_type__first) and (element(fld_elem(
          upf_elem(table, update(fld_elem(table), [
          loop__1__j], element(fld_elem(table), [loop__1__j + 1])))), [
          i___1]) <= contents_type__last))) .
C2:    fld_size(upf_elem(table, update(fld_elem(table), [
          loop__1__j], element(fld_elem(table), [loop__1__j + 1])))) >=
          base_index_type__first .
C3:    fld_size(upf_elem(table, update(fld_elem(table), [
          loop__1__j], element(fld_elem(table), [loop__1__j + 1])))) <=
          base_index_type__last .
C4:    value >= contents_type__first .
C5:    value <= contents_type__last .
C6:    loop__1__j + 1 >= index_type__first .
C7:    loop__1__j + 1 <= index_type__last .


For path(s) from default assertion of line 113 to run-time check associated with
statement of line 114:

procedure_delete_7.
H1:     for_all (i___1: base_index_type, ((i___1 >=
          index_type__first) and (i___1 <= index_type__last)) -> ((element(
          fld_elem(table), [i___1]) >= contents_type__first) and (element(
          fld_elem(table), [i___1]) <= contents_type__last))) .
H2:    fld_size(table) >= base_index_type__first .
H3:    fld_size(table) <= base_index_type__last .
H4:    value >= contents_type__first .
H5:    value <= contents_type__last .
H6:    loop__1__j >= index_type__first .
H7:    loop__1__j <= index_type__last .
        ->
C1:    element(fld_elem(table), [loop__1__j + 1]) >=
          contents_type__first .
C2:    element(fld_elem(table), [loop__1__j + 1]) <=
          contents_type__last .
C3:    loop__1__j + 1 >= index_type__first .
C4:    loop__1__j + 1 <= index_type__last .
C5:    loop__1__j + 1 >= base_index_type__base__first .
C6:    loop__1__j + 1 <= base_index_type__base__last .
C7:    loop__1__j >= index_type__first .
C8:    loop__1__j <= index_type__last .


For path(s) from start to run-time check associated with statement of line 116:

procedure_delete_8.
H1:    true .
H2:     for_all (i___1: base_index_type, ((i___1 >=
          index_type__first) and (i___1 <= index_type__last)) -> ((element(
          fld_elem(table), [i___1]) >= contents_type__first) and (element(
          fld_elem(table), [i___1]) <= contents_type__last))) .
H3:    fld_size(table) >= base_index_type__first .
H4:    fld_size(table) <= base_index_type__last .
H5:    value >= contents_type__first .
H6:    value <= contents_type__last .
H7:    value >= contents_type__first .
H8:    value <= contents_type__last .
H9:    find(table, value) >= base_index_type__first .
H10:   find(table, value) <= base_index_type__last .
H11:   find(table, value) >= base_index_type__first .
H12:   find(table, value) <= base_index_type__last .
H13:   not (find(table, value) = 0) .
H14:   (find(table, value) <= fld_size(table) - 1) -> ((fld_size(
          table) - 1 >= index_type__first) and (fld_size(
          table) - 1 <= index_type__last)) .
H15:   (find(table, value) <= fld_size(table) - 1) -> ((
          find(table, value) >= index_type__first) and (
          find(table, value) <= index_type__last)) .
H16:   fld_size(table) - 1 >= base_index_type__first .
H17:   fld_size(table) - 1 <= base_index_type__last .
H18:   find(table, value) >= base_index_type__first .
H19:   find(table, value) <= base_index_type__last .
H20:   fld_size(table) - 1 >= base_index_type__base__first .
H21:   fld_size(table) - 1 <= base_index_type__base__last .
H22:   not (find(table, value) <= fld_size(table) - 1) .
        ->
C1:    fld_size(table) - 1 >= base_index_type__first .
C2:    fld_size(table) - 1 <= base_index_type__last .


For path(s) from default assertion of line 113 to run-time check associated with
statement of line 116:

procedure_delete_9.
H1:     for_all (i___1: base_index_type, ((i___1 >=
          index_type__first) and (i___1 <= index_type__last)) -> ((element(
          fld_elem(table), [i___1]) >= contents_type__first) and (element(
          fld_elem(table), [i___1]) <= contents_type__last))) .
H2:    fld_size(table) >= base_index_type__first .
H3:    fld_size(table) <= base_index_type__last .
H4:    value >= contents_type__first .
H5:    value <= contents_type__last .
H6:    loop__1__j >= index_type__first .
H7:    loop__1__j <= index_type__last .
H8:    element(fld_elem(table), [loop__1__j + 1]) >=
          contents_type__first .
H9:    element(fld_elem(table), [loop__1__j + 1]) <=
          contents_type__last .
H10:   loop__1__j + 1 >= index_type__first .
H11:   loop__1__j + 1 <= index_type__last .
H12:   loop__1__j + 1 >= base_index_type__base__first .
H13:   loop__1__j + 1 <= base_index_type__base__last .
H14:   loop__1__j >= index_type__first .
H15:   loop__1__j <= index_type__last .
H16:   loop__1__j = fld_size(upf_elem(table, update(fld_elem(
          table), [loop__1__j], element(fld_elem(table), [
          loop__1__j + 1])))) - 1 .
        ->
C1:    fld_size(upf_elem(table, update(fld_elem(table), [
          loop__1__j], element(fld_elem(table), [loop__1__j + 1])))) -
          1 >= base_index_type__first .
C2:    fld_size(upf_elem(table, update(fld_elem(table), [
          loop__1__j], element(fld_elem(table), [loop__1__j + 1])))) -
          1 <= base_index_type__last .
```

```
For path(s) from start to finish:

procedure_delete_10.
*** true .          /* trivially true VC removed by Examiner */


procedure_delete_11.
*** true .          /* trivially true VC removed by Examiner */


For path(s) from default assertion of line 113 to finish:

procedure_delete_12.
*** true .          /* trivially true VC removed by Examiner */
```

B.2.2 Simplified

```
        *******************************************************
                        Semantic Analysis of SPARK Text
     SPARK95 Examiner with VC and RTC Generator Release 6.0 / 11.01
                            Demonstration Version
        *******************************************************


     CREATED 19-JUL-2002, 15:56:55  SIMPLIFIED 19-JUL-2002, 15:57:19
             (Simplified by SPADE Simplifier, Demo Version)

                    procedure Table_Utilities.Delete




For path(s) from start to run-time check associated with statement of line 108:

procedure_delete_1.
*** true .          /* all conclusions proved */


For path(s) from start to run-time check associated with statement of line 108:

procedure_delete_2.
*** true .          /* all conclusions proved */


For path(s) from start to run-time check associated with statement of line 113:

procedure_delete_3.
H1:    for_all(i___1 : base_index_type, 1 <= i___1 and i___1 <= 100 -> - 1000
          <= element(fld_elem(table), [i___1]) and element(fld_elem(table), [
          i___1]) <= 1000) .
H2:    fld_size(table) >= 0 .
H3:    fld_size(table) <= 100 .
H4:    value >= - 1000 .
H5:    value <= 1000 .
H6:    0 < find(table, value) .
H7:    find(table, value) <= 100 .
       ->
C1:    find(table, value) <= fld_size(table) - 1 -> fld_size(table) >= 2 and
          fld_size(table) <= 101 .
C2:    fld_size(table) >= 1 .


For path(s) from start to run-time check associated with statement of line 113:

procedure_delete_4.
H1:    for_all(i___1 : base_index_type, 1 <= i___1 and i___1 <= 100 -> - 1000
          <= element(fld_elem(table), [i___1]) and element(fld_elem(table), [
          i___1]) <= 1000) .
H2:    fld_size(table) <= 100 .
H3:    value >= - 1000 .
H4:    value <= 1000 .
H5:    0 < find(table, value) .
H6:    find(table, value) <= 100 .
H7:    find(table, value) <= fld_size(table) - 1 -> fld_size(table) >= 2 and
          fld_size(table) <= 101 .
H8:    find(table, value) <= fld_size(table) - 1 -> find(table, value) >= 1 and
          find(table, value) <= 100 .
H9:    fld_size(table) >= 1 .
       ->
C1:    fld_size(table) - 1 >= base_index_type__base__first .
C2:    fld_size(table) - 1 <= base_index_type__base__last .


For path(s) from start to default assertion of line 113:

procedure_delete_5.
*** true .          /* all conclusions proved */


For path(s) from default assertion of line 113 to default assertion of line 113:

procedure_delete_6.
*** true .          /* all conclusions proved */


For path(s) from default assertion of line 113 to run-time check associated
          with statement of line 114:

procedure_delete_7.
H1:    for_all(i___1 : base_index_type, 1 <= i___1 and i___1 <= 100 -> - 1000
          <= element(fld_elem(table), [i___1]) and element(fld_elem(table), [
          i___1]) <= 1000) .
H2:    fld_size(table) >= 0 .
H3:    fld_size(table) <= 100 .
H4:    value >= - 1000 .
H5:    value <= 1000 .
H6:    loop__1__j >= 1 .
H7:    loop__1__j <= 100 .
       ->
C1:    element(fld_elem(table), [loop__1__j + 1]) >= - 1000 .
C2:    element(fld_elem(table), [loop__1__j + 1]) <= 1000 .
C3:    loop__1__j <= 99 .
```

```
C4:    loop__1__j + 1 >= base_index_type__base__first .
C5:    loop__1__j + 1 <= base_index_type__base__last .


For path(s) from start to run-time check associated with statement of line 116:

procedure_delete_8.
*** true .          /* all conclusions proved */


For path(s) from default assertion of line 113 to run-time check associated
          with statement of line 116:

procedure_delete_9.
*** true .          /* all conclusions proved */


For path(s) from start to finish:

procedure_delete_10.
*** true .          /* all conclusions proved */



procedure_delete_11.
*** true .          /* all conclusions proved */



For path(s) from default assertion of line 113 to finish:

procedure_delete_12.
*** true .          /* all conclusions proved */
```

C Table Utilities Procedure Find

C.1 Code

C.1.1 Specification

```
-- Table Utilities for Run-Time Errors Case Study
-- for vc generation with rtc and simplification
-- (Taken from:
-- Proof Of The Absence Od Run-Time Errors from
-- SPARK Programs -- A Case Study)
package Table_Utilities is
    Max_Table_Size: constant := 100;
    type Base_Index_Type is range 0 .. Max_Table_Size;
    subtype Index_Type is Base_Index_Type range 1 .. Max_Table_Size;
    type Contents_Type is range -1000 .. 1000;
    type Array_Type is array (Index_Type) of Contents_Type;
    type Table_Type is record
                        Size: Base_Index_Type;
                        Elem: Array_Type;
                      end record;
    procedure Find(Table: in Table_Type;
                   Target: in Contents_Type;
                   Result: out Base_Index_Type);
    --#derives Result from Table, Target;
    --  returns Key s.t. Key/=0 -> Table.Elem(Key)=Target and
    --                   Key=0  -> for all i in 1..Table.Size,
    --                                 Table.Elem(i) /= Target;

    procedure Sort(Table: in out Table_Type);
    --#derives Table from Table;
    --  sorts the Table.Elem array into ascending numerical order, preserving
    --  contents and size

    procedure Insert(Table: in out Table_Type;
                     Sorted, Unique: in Boolean;
                     Value: in Contents_Type;
                     Success: out Boolean);
    --#derives Table, Success from Table, Sorted, Unique, Value;
    --  inserts Value into the Table.Elem, such that:
    --    - if Sorted is true, inserts at a point in ascending numerical
    --      order, otherwise inserts at end of array;
    --    - if Unique is true, does not create multiple entries, otherwise
    --      will allow more than one entry with same value;
    --    - if there is no room for the insertion (Table.Size=100 and an
    --      insert is needed), does nothing to Table and returns a Success of
    --      False, otherwise returns a Success of True.

    procedure Delete(Table: in out Table_Type;
                     Value: in Contents_Type;
                     Success: out Boolean);
    --#derives Table, Success from Table, Value;
end Table_Utilities;
```

C.1.2 Body

```
package body Table_Utilities is
    procedure Find(Table: in Table_Type;
                   Target: in Contents_Type;
                   Result: out Base_Index_Type)
    is
    begin
       Result:= 0;
       for I in Index_Type range 1 .. Table.Size loop
          if Table.Elem(I) = Target then
             Result := I;
             exit;
          end if;
       end loop;
    end Find;
    ------------------------------------------------------------------------
    procedure Sort(Table: in out Table_Type)
    is
       Key: Index_Type;
       function Find_Smallest(Arr: Array_Type;
                              L, U: Index_Type) return Index_Type
       is
          K: Index_Type;
       begin
          K := L;
          for I in Index_Type range L+1 .. U loop
             if Arr(I) < Arr(K) then
                K := I;
             end if;
          end loop;
          return K;
       end Find_Smallest;
       procedure Swap_Elements(T: in out Table_Type; I, J: in Index_Type)
       --#derives T from T, I, J;
       is
          Temp: Contents_Type;
       begin
          Temp := T.Elem(I);
          T.Elem(I) := T.Elem(J);
          T.Elem(J) := Temp;
       end Swap_Elements;

    begin   -- Sort
        for Low in Index_Type range 1 .. Table.Size-1 loop
            Key := Find_Smallest(Table.Elem, Low, Table.Size);
            if Key /= Low then
                Swap_Elements(Table, Low, Key);
            end if;
        end loop;
    end Sort;
    ------------------------------------------------------------------------
    procedure Insert(Table: in out Table_Type;
                     Sorted, Unique: in Boolean;
                     Value: in Contents_Type;
                      Success: out Boolean)
    is
        Successful: Boolean;
        I: Base_Index_Type;
    begin
        if Sorted then
            Find(Table, Value, I);
            if Unique and I/= 0 then
                Successful := true;
            else
                Successful := Table.Size < Max_Table_Size;
                if Successful then
                    --! Sorted, there's space & insert, so find place and insert
                    I := 1;
                    loop
                        exit when I > Table.Size or else
                                  Table.Elem(I) >= Value;
                        I := I + 1;
                    end loop;
                    for X in reverse Index_Type range I .. Table.Size loop
                        Table.Elem(X+1) := Table.Elem(X);
                    end loop;
                    Table.Elem(I) := Value;
                    Table.Size := Table.Size + 1;
                end if;
            end if;
        else
            if Unique then
                --! not Sorted, Unique: insert at end if not already in Table
                Find(Table, Value, I);
                Successful := I /= 0;
                if not Successful then
                    Successful := Table.Size < Max_Table_Size;
                    if Successful then
                        Table.Size := Table.Size + 1;
                        Table.Elem(Table.Size) := Value;
                    end if;
                end if;
            else
                --! not Sorted, not Unique: insert regardless, at end
                Successful := Table.Size < Max_Table_Size;
                if Successful then
                    Table.Size := Table.Size + 1;
                    Table.Elem(Table.Size) := Value;
                end if;
            end if;
        end if;
        Success := Successful;
    end Insert;
    ------------------------------------------------------------------------
    procedure Delete(Table: in out Table_Type;
                     Value: in Contents_Type;
                     Success: out Boolean)
    is
        I: Base_Index_Type;
    begin
        Find(Table, Value, I);
        if I = 0 then
            Success := false;
        else
            Success := true;
            for J in Index_Type range I .. Table.Size - 1 loop
                --#assert J in Index_Type and
                --#       J in I .. Table.Size - 1 and
                --#       I in Base_Index_Type;
                Table.Elem(J) := Table.Elem(J+1);
            end loop;
            Table.Size := Table.Size - 1;
        end if;
    end Delete;

end Table_Utilities;
```

References

[1] J. G. P. Barnes. *High Integrity Ada The SPARK Approach.* Addison-Wesley, 1997.

[2] N. Klarlund M. Biehl and T. Rauhe.

[3] Praxis Critical Systems. Proof of the absence of run-time errors from spark programs a case study. 1996.

[4] Praxis Critical Systems. Proof of a sorting algorithma case study. 1997.

[5] Praxis Critical Systems. Spade proof checker version 1.3 rules manual. 1998.