CS-Note* 41

Andrew Ireland
April 11, 2003

Automation for Exception Freedom Proofs: Take-2

1 Introduction

In CS-note 34 a strategy was proposed for automating exception freedom proofs within the context of loop based programs. In particular, verification of overflow exception freedom was considered, *i.e.* ensuring that program variables are not assigned values outwith their legal range. The strategy was based upon coloured rippling and a process of analyzing the differences between the actual upper (lower) bounds of program variables over a small number of iterations. Further investigation has shown that a coloured rippling approach is not actually applicable[1]. Within coloured rippling, all the target hypotheses (skeletons) must share the same structure, *i.e.* they differ only in terms variable names. However, within the context of exception freedom proofs the target hypotheses will typically not share the same structure. All is not lost, as basic monochromatic rippling still has a role to play as is outlined below. The aim of this note therefore is to highlight how the basic ideas have evolved since writing CS-note 34.

2 Phil's example revisited

Recall that the example used to illustrate the ideas took the following form:

```
subtype Index_Type is Integer range 0 .. 9;
type Data_Set_Type is array (Index_Type) of Integer;
...
Result:= 0;
for I in Index_Type loop
   --# assert true;
   if Data_Set(I) >= 0 and Data_Set(I) <= 100 then
      Result:= Result+Data_Set(I);
   end if;
end loop;
```

Our point of focus is the assignment:

```
Result:= Result+Data_Set(I);
```

That is, we need to guarantee that the assignment will never cause `Result` to exceed the integer bounds `integer__first` and `integer__last`. This proof obligation is expressed by the following VC[2]:

```
procedure_data_filter_6.
H1:    true .
H2:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
           i___1 <= index_type__last)) -> ((element(
           data_set, [i___1]) >= integer__first) and (element(
           data_set, [i___1]) <= integer__last))) .
H3:    loop__1__i >= index_type__first .
H4:    loop__1__i <= index_type__last .
H5:    loop__1__i >= index_type__first .
H6:    loop__1__i <= index_type__last .
H7:    loop__1__i >= index_type__first .
H8:    loop__1__i <= index_type__last .
H9:    element(data_set, [loop__1__i]) >= 0 .
H10:   element(data_set, [loop__1__i]) <= 100 .
        ->
C1:    result + element(data_set, [loop__1__i]) >= integer__first .
C2:    result + element(data_set, [loop__1__i]) <= integer__last .
C3:    loop__1__i >= index_type__first .
C4:    loop__1__i <= index_type__last .
```

*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]Thanks to Bill and Alan who hinted at this independently through their feedback to CS-note 34.

[2]For completeness all the VCs arising from this example are given in appendix A.

The crucial conclusions here are C1 and C2, since neither can be discharged by the SPADE Simplifier. Note that the above VC is concerned with the overflow check on `result`, *i.e.* it is not the loop invariant VC. The significance of this will be made clearer below. Focusing on C2, consider the following (pretty-printed) version of the VC[3]:

$$
\begin{array}{lrcl}
\text{Givens:} & \ldots & & \ldots \\
 & ele(data\_set, i) & \leq & 100 \\
 & \ldots & & \ldots \\
\text{Goal:} & r + ele(data\_set, i) & \leq & integer\_last
\end{array}
$$

The failure analysis of this VC presented in CS-note 34 was flawed. The problem is that the analysis was expressed in terms of a failed ripple proof, but we are not dealing here with a loop invariant proof so rippling is not directly applicable. Informally, our revised failure analysis is as follows:

Pattern: We are required to show that the upper bound on r + *ele(data_set, i)* is less-than-or-equal to the largest integer value that r can be assigned. While we know the upper bound on *ele(data_set, i)* we have no knowledge of the upper bound on r.

Patch: Search for an upper bound on r that will hold true for all iterations of the loop.

Note that the above analysis changes the focus of the proof patching activity. That is, in order to introduce an upper bound on r within the unproven overflow check VC we are required to speculate a loop invariant based upon the bounds of r. Put another way, by strengthening the loop invariant we will seek to provide, as a side effect, enough context in order to discharge the unproven overflow check VC. Note however that strengthening the loop invariant will give rise to an invariant VC that the SPADE Simplifier will now fail to discharge.

3 Loop Invariant Discovery

As mentioned in CS-note 34, adopting a middle-out style of reasoning would result in a schematic loop invariant VC of the form:

$$
\begin{array}{lrcl}
\text{Givens:} & \ldots & & \ldots \\
 & r & \leq & F(i, r) \\
 & ele(data\_set, i) & \leq & 100 \\
 & \ldots & & \ldots \\
\text{Goal:} & \boxed{\boxed{r} + ele(data\_set, i)}^{\uparrow} & \leq & F(\boxed{\boxed{i} + 1}^{\uparrow}, \boxed{\boxed{r} + ele(data\_set, i)}^{\uparrow})
\end{array}
$$

where F denotes a second-order meta-variable. This approach has proved to be very successful in the past within the context of lemma discovery and conjecture generalization [3]. However, within the current application we believe that the approach is not so useful. Our reasoning is as follows. Middle-out reasoning, within the context of rippling, relies upon there being sufficient concrete term structure within a goal in order to constrain the instantiation of higher-order meta-variables. Here we have minimal concrete term structure. As a consequence, we are pursuing the approach of deducing actual upper bounds for the first n loop iterations and then seeking to identify invariants by observing general patterns across the properties deduced. Below we present a slight variation on the basic idea presented in CS-note 34:

$$
\begin{array}{lcllcl}
i_0 & = & 0 & \wedge \;\; r_0 & \leq & 0 \\
\ldots & & \ldots & \;\;\;\;\; \ldots & & \\
i_1 & = & 0 \underbrace{+1}_{\alpha} & \wedge \;\; r_1 & \leq & 0 \underbrace{+100}_{\beta} \\
\ldots & & \ldots & \;\;\;\;\; \ldots & & \\
i_2 & = & (0 \underbrace{\underbrace{+1}_{\alpha}) + 1}_{2*\alpha} & \wedge \;\; r_2 & \leq & (0 \underbrace{\underbrace{+100}_{\beta}) + 100}_{2*\beta} \\
\ldots & & \ldots & \;\;\;\;\; \ldots & & \\
i_3 & = & ((0 \underbrace{\underbrace{\underbrace{+1}_{\alpha}) + 1}_{2*\alpha}) + 1}_{3*\alpha} & \wedge \;\; r_3 & \leq & ((0 \underbrace{\underbrace{\underbrace{+100}_{\beta}) + 100}_{2*\beta}) + 100}_{3*\beta}
\end{array}
$$

[3]Note that r is an abbreviation for `result` and *ele* is an abbreviation for `element`.

To analyze such structures we will introduce some new notation. To illustrate, consider the following sequence:

$$((a \oplus b_1) \oplus b_2) \ldots \oplus b_n$$

where $\oplus$ denotes an arbitrary binary operator. In order to express the repeated pattern of $\ldots \oplus b$ we could use the following notation:

$$a \underbrace{\oplus\, b}_{n}$$

where $\underbrace{\ldots\, b}_{n}$ is a meta-level construction with the following definition:

$$\begin{cases} X \underbrace{\oplus Y}_{0} & = & X \\ X \underbrace{\oplus Y}_{N+1} & = & (X \underbrace{\oplus Y}_{N}) \oplus Y \end{cases}$$

Returning to our example, this notation allows us to succinctly express the patterns observed with respect to the loop variable i and the upper bound on r:

$$i_n = 0 \underbrace{+1}_{n} \;\wedge\; r_n \leq 0 \underbrace{+100}_{n} \tag{1}$$

Note that n denotes the number of iterations. The final step in generating a candidate loop invariant involves rewriting (1) so as to eliminate all occurrences of n, *e.g.*

$$\begin{array}{rcl} i_n = 0 \underbrace{+1}_{n} & \wedge & r_n \leq 0 \underbrace{+100}_{n} \\ i_n = 0 + (n * 1) & \wedge & r_n \leq 0 \underbrace{+100}_{n} \\ i_n = 0 + (n * 1) & \wedge & r_n \leq 0 + (n * 100) \\ i_n = n * 1 & \wedge & r_n \leq 0 + (n * 100) \\ i_n = n & \wedge & r_n \leq 0 + (n * 100) \\ i_n = n & \wedge & r_n \leq n * 100 \\ i_n = n & \wedge & r_n \leq i * 100 \end{array}$$

Note that $r \leq i * 100$ provides half of the required loop invariant. While the majority of the above derivation corresponds to conventional equation solving, note that the first 2 steps make use of the following rewrite rule:

$$X \underbrace{+Y}_{N} \Rightarrow X + (N * Y) \tag{2}$$

This rule expresses the fact that a repeated sequence of summations can be rewritten as a product. Essentially, this rule provides the key to generating the invariant. For other proofs we will require additional rules, *e.g.* for a repeated sequence of products we might use:

$$X \underbrace{*Y}_{N} \Rightarrow X * Y^N$$

Obviously the value of this approach will depend upon the number of such rewrite rules that will be required in practice. Our belief is that a relatively small number of such rewrite rules will provide significant results on the experimental data – time will tell!

There appears to be a strong connection between the idea presented above and the notion of a *recurrence relation*[4]. Recurrence relations are typically presented in terms of complexity analysis, *i.e.* calculating upper bounds on the running times of programs. Here we are interested in the upper bounds on the values assigned to variables. A closer look at the work on recurrence relationships may well prove very useful.

As an aside, given that such rewrite rules appear to provide the key to discovering certain kinds of invariant, then could we automate the discovery of such rules for new operators? In general the answer if no. However, the

[4] See [1] for a good introduction to the concept.

interesting question is how far might a technique like middle-out synthesis take us? To be more specific, consider the verification and synthesis of (2). Firstly, in order to verify the correctness of (2) we would have to prove the following:

$$\forall N, X, Y.\ \ X \underbrace{+Y}_{N} = X + (N * Y)$$

Proof is by structural induction on N:

Base case

$$\begin{array}{lrcl}
\text{Goal:} & X \underbrace{+Y}_{0} & = & X + (0 * Y) \\
& X & = & X + 0 \\
& X & = & X \\
& & \mathit{true} &
\end{array}$$

Step case

$$\begin{array}{lrcl}
\text{Givens:} & X \underbrace{+Y}_{N} & = & X + (N * Y) \\
\text{Goal:} & X \underbrace{+Y}_{\boxed{N+1}^{\uparrow}} & = & X + (\boxed{N+1}^{\uparrow} * Y) \\
& \boxed{(X \underbrace{+Y}_{N}) + Y}^{\uparrow} & = & \boxed{(X + (N * Y)) + Y}^{\uparrow} \\
& & \boxed{X \underbrace{+Y}_{N} = X + (N * Y) \wedge Y = Y}^{\uparrow} & \\
& & \mathit{true} \wedge Y = Y & \\
& & \mathit{true} &
\end{array}$$

So verification corresponds to a simple application of the induction strategy.

Secondly, what is required to move from verification to synthesis, *e.g.* once a repeated pattern is identified we could attempt to plan a proof of the following synthesis conjecture:

$$\exists F.\ \ \forall N, X, Y.\ \ X \underbrace{+Y}_{N} = F(N, X, Y)$$

where F denotes a second-order meta-variable. Note that this task has similarities to the work by Jane Hesketh on synthesizing tail-recursive functions from naive recursive function definitions [2]. In practice, I suspect, there will be little demand for such synthesis, *i.e.* a relatively small fixed set of rewrite rules for a fixed set of operators will be sufficient for SPARK applications. However, I think the synthesis angle is an interesting line of research to pursue at some point.

4 Loop Invariant Verification

Once generated, the candidate loop invariant must be verified. This firstly involves introducing the candidate within the actual loop code. In our example this gives us:

```
subtype Index_Type is Integer range 0 .. 9;
type Data_Set_Type is array (Index_Type) of Integer;
...
Result:= 0;
for I in Index_Type loop
   --# assert Result >= 0 and Result <= I * 100;
   if Data_Set(I) >= 0 and Data_Set(I) <= 100 then
      Result:= Result+Data_Set(I);
   end if;
end loop;
```

Note that the loop body contains an if-then statement, consequently 2 loop invariant VCs are generated[5]. The first VC corresponds to the verification of the invariant when the then-branch is traversed. The second VC corresponds to the verification of the invariant when the then-branch is not traversed. Ripple based proofs for both VCs are given below. The required wave-rules and rewrite rules are as follows:

$$\boxed{(X + Y)}^{\uparrow} * Z \ \ \Rightarrow\ \ \boxed{(X * Z) + (Y * Z)}^{\uparrow} \tag{3}$$

[5]Note that appendix B gives all the VCs that arise from the revised code.

$$\boxed{(W + X)}^{\uparrow} \leq \boxed{(Y + Z)}^{\uparrow} \quad \Rightarrow \quad \boxed{W \leq Y \wedge X \leq Z}^{\uparrow} \tag{4}$$

$$X \leq \boxed{(Y + Z)}^{\uparrow} \quad \Rightarrow \quad \boxed{X \leq Y \wedge Z \geq 0}^{\uparrow} \tag{5}$$

$$1 * X \Rightarrow X \tag{6}$$

Invariant proof including then-branch

$$\begin{array}{lrcl} \text{Givens:} & r & \leq & i * 100 \\ & ele(data\_set, i) & \leq & 100 \end{array}$$

$$\text{Goal:} \quad \boxed{r + ele(data\_set, i)}^{\uparrow} \leq \boxed{(i + 1)}^{\uparrow} * 100$$

by (3)

$$\boxed{r + ele(data\_set, i)}^{\uparrow} \leq \boxed{(i * 100) + (1 * 100)}^{\uparrow}$$

by (4)

$$\boxed{r \leq i * 100 \wedge ele(data\_set, i) \leq 1 * 100}^{\uparrow}$$

by given: $r \leq i * 100$

$$ele(data\_set, i) \leq 1 * 100$$

by (6)

$$ele(data\_set, i) \leq 100$$

by given: $ele(data\_set, i) \leq 100$

$$true$$

Invariant proof excluding then-branch

$$\text{Givens:} \quad r \leq i * 100$$

$$\text{Goal:} \quad r \leq \boxed{(i + 1)}^{\uparrow} * 100$$

by (3)

$$r \leq \boxed{(i * 100) + (1 * 100)}^{\uparrow}$$

by (5)

$$\boxed{r \leq i * 100 \wedge 1 * 100 \geq 0}^{\uparrow}$$

by given: $r \leq i * 100$

$$1 * 100 \geq 0$$

by arith

$$true$$

5 Conclusion

A key observation to come from this re-evaluation is that there are a number of inter-dependent parts that need to be considered:

1. The automatic discovery of loop invariants that will, as a side-effect, support the process of proving freedom from overflow exceptions;
2. The automatic proof of such loop invariants;
3. The automatic proof of overflow exception freedom once the generated loop invariant has been introduced within the code.

Our ideas on how to achieve the first step are reasonably well developed on paper. The second step is a simple application of rippling (monochromatic), and is well understood – moreover, the beginnings of a working prototype in NuSPADE are now in place. The third step is yet to be addressed, and will involve inequality reasoning.

References

[1] A.V. Aho and J.D. Ullman. *Foundations of Computer Science*. Computer Science Press, 1995.

[2] J. Hesketh, Alan Bundy, and A. Smaill. Using middle-out reasoning to control the synthesis of tail-recursive programs. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, pages 310–324, Saratoga Springs, NY, USA, June 1992. Published as Springer Lecture Notes in Artificial Intelligence, No 607.

[3] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996. Also available as DAI Research Paper No 716, Dept. of Artificial Intelligence, Edinburgh.

Appendix A: data_filter VCs – without loop invariant

```
For path(s) from start to run-time check associated with statement of line 7:

procedure_data_filter_1.
H1:    true .
H2:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
           i___1 <= index_type__last)) -> ((element(
           data_set, [i___1]) >= integer__first) and (element(
           data_set, [i___1]) <= integer__last))) .
        ->
C1:    0 >= integer__first .
C2:    0 <= integer__last .

For path(s) from start to assertion of line 9:
procedure_data_filter_2.
H1:    true .
H2:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
           i___1 <= index_type__last)) -> ((element(
           data_set, [i___1]) >= integer__first) and (element(
           data_set, [i___1]) <= integer__last))) .
H3:    0 >= integer__first .
H4:    0 <= integer__last .
        ->
C1:    true .
C2:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
           i___1 <= index_type__last)) -> ((element(
           data_set, [i___1]) >= integer__first) and (element(
           data_set, [i___1]) <= integer__last))) .
C3:    index_type__first >= index_type__first .
C4:    index_type__first <= index_type__last .

For path(s) from assertion of line 9 to assertion of line 9:
procedure_data_filter_3.
H1:    true .
H2:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
           i___1 <= index_type__last)) -> ((element(
           data_set, [i___1]) >= integer__first) and (element(
           data_set, [i___1]) <= integer__last))) .
H3:    loop__1__i >= index_type__first .
H4:    loop__1__i <= index_type__last .
H5:    loop__1__i >= index_type__first .
H6:    loop__1__i <= index_type__last .
H7:    loop__1__i >= index_type__first .
H8:    loop__1__i <= index_type__last .
H9:    element(data_set, [loop__1__i]) >= 0 .
H10:   element(data_set, [loop__1__i]) <= 100 .
H11:   result + element(data_set, [loop__1__i]) >= integer__first .
H12:   result + element(data_set, [loop__1__i]) <= integer__last .
H13:   loop__1__i >= index_type__first .
H14:   loop__1__i <= index_type__last .
H15:   not (loop__1__i = index_type__last) .
        ->
C1:    true .
C2:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
           i___1 <= index_type__last)) -> ((element(
           data_set, [i___1]) >= integer__first) and (element(
           data_set, [i___1]) <= integer__last))) .
C3:    loop__1__i + 1 >= index_type__first .
C4:    loop__1__i + 1 <= index_type__last .

procedure_data_filter_4.
H1:    true .
H2:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
```

```
          i___1 <= index_type__last)) -> ((element(
          data_set, [i___1]) >= integer__first) and (element(
          data_set, [i___1]) <= integer__last))) .
H3:    loop__1__i >= index_type__first .
H4:    loop__1__i <= index_type__last .
H5:    loop__1__i >= index_type__first .
H6:    loop__1__i <= index_type__last .
H7:    loop__1__i >= index_type__first .
H8:    loop__1__i <= index_type__last .
H9:    not ((element(data_set, [loop__1__i]) >= 0) and (element(
          data_set, [loop__1__i]) <= 100)) .
H10:   not (loop__1__i = index_type__last) .
        ->
C1:    true .
C2:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
          i___1 <= index_type__last)) -> ((element(
          data_set, [i___1]) >= integer__first) and (element(
          data_set, [i___1]) <= integer__last))) .
C3:    loop__1__i + 1 >= index_type__first .
C4:    loop__1__i + 1 <= index_type__last .

For path(s) from assertion of line 9 to run-time check associated with statement of line 10:
procedure_data_filter_5.
H1:    true .
H2:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
          i___1 <= index_type__last)) -> ((element(
          data_set, [i___1]) >= integer__first) and (element(
          data_set, [i___1]) <= integer__last))) .
H3:    loop__1__i >= index_type__first .
H4:    loop__1__i <= index_type__last .
        ->
C1:    loop__1__i >= index_type__first .
C2:    loop__1__i <= index_type__last .
C3:    loop__1__i >= index_type__first .
C4:    loop__1__i <= index_type__last .

For path(s) from assertion of line 9 to run-time check associated with statement of line 11:
procedure_data_filter_6.
H1:    true .
H2:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
          i___1 <= index_type__last)) -> ((element(
          data_set, [i___1]) >= integer__first) and (element(
          data_set, [i___1]) <= integer__last))) .
H3:    loop__1__i >= index_type__first .
H4:    loop__1__i <= index_type__last .
H5:    loop__1__i >= index_type__first .
H6:    loop__1__i <= index_type__last .
H7:    loop__1__i >= index_type__first .
H8:    loop__1__i <= index_type__last .
H9:    element(data_set, [loop__1__i]) >= 0 .
H10:   element(data_set, [loop__1__i]) <= 100 .
        ->
C1:    result + element(data_set, [loop__1__i]) >= integer__first .
C2:    result + element(data_set, [loop__1__i]) <= integer__last .
C3:    loop__1__i >= index_type__first .
C4:    loop__1__i <= index_type__last .

For path(s) from assertion of line 9 to finish:
procedure_data_filter_7.
*** true .          /* trivially true VC removed by Examiner */

procedure_data_filter_8.
*** true .          /* trivially true VC removed by Examiner */
```

Appendix B: data_filter VCs – with loop invariant

```
For path(s) from start to run-time check associated with statement of line 7:
procedure_data_filter_1.
H1:    true .
H2:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
           i___1 <= index_type__last)) -> ((element(
           data_set, [i___1]) >= integer__first) and (element(
           data_set, [i___1]) <= integer__last))) .
        ->
C1:    0 >= integer__first .
C2:    0 <= integer__last .


For path(s) from start to assertion of line 9:
procedure_data_filter_2.
H1:    true .
H2:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
           i___1 <= index_type__last)) -> ((element(
           data_set, [i___1]) >= integer__first) and (element(
           data_set, [i___1]) <= integer__last))) .
H3:    0 >= integer__first .
H4:    0 <= integer__last .
        ->
C1:    0 >= 0 .
C2:    0 <= index_type__first * 100 .
C3:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
           i___1 <= index_type__last)) -> ((element(
           data_set, [i___1]) >= integer__first) and (element(
           data_set, [i___1]) <= integer__last))) .
C4:    index_type__first >= index_type__first .
C5:    index_type__first <= index_type__last .


For path(s) from assertion of line 9 to assertion of line 9:
procedure_data_filter_3.
H1:    result >= 0 .
H2:    result <= loop__1__i * 100 .
H3:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
           i___1 <= index_type__last)) -> ((element(
           data_set, [i___1]) >= integer__first) and (element(
           data_set, [i___1]) <= integer__last))) .
H4:    loop__1__i >= index_type__first .
H5:    loop__1__i <= index_type__last .
H6:    loop__1__i >= index_type__first .
H7:    loop__1__i <= index_type__last .
H8:    loop__1__i >= index_type__first .
H9:    loop__1__i <= index_type__last .
H10:   element(data_set, [loop__1__i]) >= 0 .
H11:   element(data_set, [loop__1__i]) <= 100 .
H12:   result + element(data_set, [loop__1__i]) >= integer__first .
H13:   result + element(data_set, [loop__1__i]) <= integer__last .
H14:   loop__1__i >= index_type__first .
H15:   loop__1__i <= index_type__last .
H16:   not (loop__1__i = index_type__last) .
        ->
C1:    result + element(data_set, [loop__1__i]) >= 0 .
C2:    result + element(data_set, [loop__1__i]) <= (
           loop__1__i + 1) * 100 .
C3:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
           i___1 <= index_type__last)) -> ((element(
           data_set, [i___1]) >= integer__first) and (element(
           data_set, [i___1]) <= integer__last))) .
C4:    loop__1__i + 1 >= index_type__first .
```

```
C5:    loop__1__i + 1 <= index_type__last .

procedure_data_filter_4.
H1:    result >= 0 .
H2:    result <= loop__1__i * 100 .
H3:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
           i___1 <= index_type__last)) -> ((element(
           data_set, [i___1]) >= integer__first) and (element(
           data_set, [i___1]) <= integer__last))) .
H4:    loop__1__i >= index_type__first .
H5:    loop__1__i <= index_type__last .
H6:    loop__1__i >= index_type__first .
H7:    loop__1__i <= index_type__last .
H8:    loop__1__i >= index_type__first .
H9:    loop__1__i <= index_type__last .
H10:   not ((element(data_set, [loop__1__i]) >= 0) and (element(
           data_set, [loop__1__i]) <= 100)) .
H11:   not (loop__1__i = index_type__last) .
        ->
C1:    result >= 0 .
C2:    result <= (loop__1__i + 1) * 100 .
C3:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
           i___1 <= index_type__last)) -> ((element(
           data_set, [i___1]) >= integer__first) and (element(
           data_set, [i___1]) <= integer__last))) .
C4:    loop__1__i + 1 >= index_type__first .
C5:    loop__1__i + 1 <= index_type__last .

For path(s) from assertion of line 9 to run-time check associated with statement of line 10:
procedure_data_filter_5.
H1:    result >= 0 .
H2:    result <= loop__1__i * 100 .
H3:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
           i___1 <= index_type__last)) -> ((element(
           data_set, [i___1]) >= integer__first) and (element(
           data_set, [i___1]) <= integer__last))) .
H4:    loop__1__i >= index_type__first .
H5:    loop__1__i <= index_type__last .
        ->
C1:    loop__1__i >= index_type__first .
C2:    loop__1__i <= index_type__last .
C3:    loop__1__i >= index_type__first .
C4:    loop__1__i <= index_type__last .

For path(s) from assertion of line 9 to run-time check associated with statement of line 11:
procedure_data_filter_6.
H1:    result >= 0 .
H2:    result <= loop__1__i * 100 .
H3:     for_all (i___1: integer, ((i___1 >= index_type__first) and (
           i___1 <= index_type__last)) -> ((element(
           data_set, [i___1]) >= integer__first) and (element(
           data_set, [i___1]) <= integer__last))) .
H4:    loop__1__i >= index_type__first .
H5:    loop__1__i <= index_type__last .
H6:    loop__1__i >= index_type__first .
H7:    loop__1__i <= index_type__last .
H8:    loop__1__i >= index_type__first .
H9:    loop__1__i <= index_type__last .
H10:   element(data_set, [loop__1__i]) >= 0 .
H11:   element(data_set, [loop__1__i]) <= 100 .
        ->
C1:    result + element(data_set, [loop__1__i]) >= integer__first .
C2:    result + element(data_set, [loop__1__i]) <= integer__last .
C3:    loop__1__i >= index_type__first .
```

```
C4:    loop__1__i <= index_type__last .

For path(s) from assertion of line 9 to finish:
procedure_data_filter_7.
*** true .          /* trivially true VC removed by Examiner */

procedure_data_filter_8.
*** true .          /* trivially true VC removed by Examiner */
```