CS-Note* 42

Bill J. Ellis
March 6, 2003

Rippling and Proofs of Exception Freedom

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

This note explores heuristics intended to control the higher level proof planning processes within NuSPADE. Emphasis is placed on exception freedom proofs that occur within loops. These are typically the most difficult kind of exception freedom proofs as they usually require a good invariant to be discovered. Some ideas based on coloured rippling and proof patching have already been considered (CS-note 34) and are continued here.

1.1 A note on SPARK

The SPARK approach to program verification is based on the classic Floyd-Hoare inductive assertion method. However, unlike many traditional approaches, SPARK generates the absolute minimum number of Verification Conditions (VCs) by having a single VC for each path between cutpoints. This approach is logically valid and arguably more presentable[1].

A result of the SPARK style is that each VC may have a number of hypotheses and conclusions. For our analysis it is helpful to break down each VC into n VCs, $\mathrm{VC_n}$, one for each conclusion of the VC. Each $\mathrm{VC_n}$ will share the hypotheses of a VC and one conclusion of this VC. Note that exactly the same separation is implemented within NuSPADE (see CS-note 20).

2 Rippling

Rippling is a well defined heuristic that emerged from automated reasoning research. The many aspects of rippling are discussed in several other locations [2, 1, 4]. The short discussion here describes the fundamental principles of rippling in both its normal monochromatic form and more general coloured form.

2.1 Monochromatic rippling

Traditional monochromatic rippling[2] emerged from the analysis of the step case of induction proofs. It is a heuristic that exploits the common pattern of the induction conclusion containing a slightly mutated copy of the induction hypothesis. For example:

$$\begin{array}{ll} \text{Induction hypothesis:} & result \leq i * 100 \\ \text{Induction conclusion:} & \boxed{\underline{\boldsymbol{result}} + element(data\_set, [i])}^{\uparrow} \leq \boxed{(\underline{\boldsymbol{i}} + 1)}^{\uparrow} * 100 \end{array}$$

The underlined syntax is called the wave hole. The syntax inside a box and not underlined is called the wave front. The wave front and wave hole annotation are added to the conclusion such that all of the mismatching syntax is inside a wave front. Rippling focuses on eliminating the mismatching syntax in the wave front. If this is achieved then the conclusion will match (fertilise) with the hypothesis completing the proof.

The transformations available to rippling are specially annotated rewrite rules (wave rules) which can only be applied where they are seen to move the mismatching syntax in a direction that will lead to proof. These restricted transformations are much more constrained than regular rewriting. In particular, rippling is always guaranteed to terminate and usually does so inside a small search space and hence reasonable time.

2.1.1 Applying monochromatic rippling

Fundamentally monochromatic rippling can be applied wherever a conclusion contains a copy of a hypothesis:

$$\textbf{Hypothesis} \rightarrow \boxed{\begin{array}{l}\textbf{Conclusion}\\ \underline{\textbf{Hypothesis}}\end{array}}$$

However, to be applicable in practise it is necessary to deal with examples where this property naturally emerges. This situation occurs in the step case of *every* proof by induction:

$$\textbf{Induction hypothesis} \rightarrow \boxed{\begin{array}{l}\textbf{Induction conclusion}\\ \underline{\textbf{Induction hypothesis}}\end{array}}$$

[1] If given the option, I would prefer many smaller traditional VCs than a few large SPARK VCs.

[2] Rippling and monochromatic rippling are synonyms. The term monochromatic rippling was invented to distinguish rippling from coloured rippling.

Further, there are very strong similarities between proofs by induction and proving the invariant to invariant case in program verification:

$$\textbf{Invariant before} \rightarrow \boxed{\begin{array}{l}\textbf{Invariant after}\\ \underline{\textbf{Invariant before}}\end{array}}$$

2.2 Coloured rippling

Coloured rippling extends monochromatic rippling by allowing the simultaneous progress of multiple hypotheses matching with the conclusion. Colour is used to distinguish between the different structures of different hypothesis matching inside the conclusion[3]. Where more than one hypothesis matches with the same syntax in the conclusion their colours are mixed. Thus any syntax marked ***Purple*** indicates a match with both a ***Red*** and ***Blue*** hypothesis. Note that only wave holes have colour associated with them. This restriction means that all syntax outside a wave front must match with every hypothesis. For example[4]:

Hypothesis 1: $\underline{maxht(l) \geq minht(l)}$

Hypothesis 2: $\underline{\underline{maxht(r) \geq minht(r)}}$

Conclusion: $maxht(\boxed{node(\underline{l},\underline{\underline{r}})}^{\uparrow}) \geq minht(\boxed{node(\underline{l},\underline{\underline{r}})}^{\uparrow})$

Just like monochromatic rippling, coloured rippling focuses on eliminating the mismatching syntax using specially configured rewrite rules. In particular, coloured rippling continues to support the favourable termination and small search space properties of monochromatic rippling. In fact, by imposing additional colour restrictions on matching, coloured rippling can be more behaved that monochromatic rippling.

2.2.1 Applying colour rippling

Coloured rippling is a super-set of monochromatic rippling. Thus it is only interesting to consider the cases where monochromatic rippling is not applicable and coloured rippling is. This occurs where multiple hypothesis are required to prove a goal and these have syntax embedded within the conclusion:

$$\begin{array}{c}\underline{\textbf{Hypothesis 1}}\\ \cdots\\ \underline{\underline{\textbf{Hypothesis N}}}\end{array} \rightarrow \boxed{\begin{array}{l}\textbf{Conclusion}\\ \quad\underline{\textbf{Hypothesis 1}}\\ \quad\quad\cdots\\ \quad\underline{\underline{\textbf{Hypothesis N}}}\end{array}}$$

This situation occurs in the step case of *every* proof by induction where reasoning with structures that are recursive in more than one argument. An example of such a data structure is a binary tree, where at a given node there is recursion down both the left and right branches:

$$\begin{array}{c}\underline{\textbf{Induction Hypothesis 1}}\\ \cdots\\ \underline{\underline{\textbf{Induction Hypothesis N}}}\end{array} \rightarrow \boxed{\begin{array}{l}\textbf{Conclusion}\\ \quad\underline{\textbf{Induction Hypothesis 1}}\\ \quad\quad\cdots\\ \quad\underline{\underline{\textbf{Induction Hypothesis N}}}\end{array}}$$

Unlike monochromatic rippling their is no universal relationship between coloured rippling and proving the invariant to invariant case in program verification. However, it is possible for coloured rippling to be applicable where reasoning about data structures that are recursive in more than one argument.

2.3 Accidental rippling

Both kinds of rippling are based on fundamental properties of induction that produce induction conclusions which include mutated copies of induction hypotheses.

However, it is still possible and fairly common for conclusions to "accidentally" contain syntax of other hypotheses that are key to proof. The reason might be that hypothesis which contain syntax seen in the conclusion are more likely to be describing similar concepts and thus may have a role to play in the conclusions proof.

[3] As colour printing is at a premium (It involves struggling to get LaTeX through a Windows network, waiting a few hours for the job to be processed and wandering outdoors to the computer centre to collect the result) an N underline notation is also used: ***Red***, ***Blue***, ***Purple***.

[4] This example is taken from [4].

Unfortunately, this accidental shared syntax is unlikely to include the deeper structural similarity relied upon by the rippling process. It may be possible squeeze ripple proofs out of this accidental shared syntax. However such proofs are not suitable for automation in general.

3 Rippling and program proof in SPARK

Depending on the VC_n being proved, different heuristics about the relationship between hypothesis and the conclusion are applicable.

3.1 From invariant to invariant

The two different kinds of rippling are considered:

- **Monochromatic rippling -**
  - **Cover nearly all loops -** Monochromatic rippling is applicable in all loops with invariants describing properties recursive on a single argument. Given that SPARK excludes pointers, recursion and dynamic memory allocation properties recursive in more than more argument are expected to be rare in partial correctness. In exception freedom proofs, where the full range of checks are known, this is never expected to be seen.
  - **Refinement for extra hypothesis -** Unlike coloured rippling it is not possible to collect additional hypotheses within the rippling process. A proof of an invariant to invariant VC_n may rely on an extra hypothesis beyond the induction hypothesis. In such cases some additional technique will be required to reference and use this extra hypothesis.

    Whatever technique adopted, the additional reasoning will take place outside the context of rippling, losing the positive terminating and constrained search of rippling. However, this additional proof work will be performed on a subcomponent of the main proof. This smaller goal may be tackled successfully using less constrained reasoning techniques.
- **Coloured rippling -**
  - **Hardly ever applicable -** Coloured rippling will only be applicable over monochromatic rippling where invariants describe properties recursive in more than one argument. This is expected to be very rare in SPARK code.
  - **Beware accidental rippling -** It is thought likely that additional hypotheses key to proof will accidentally match with the conclusion. However, these additional hypotheses do not necessarily observe the deeper relationships expected by rippling, making ripple proofs unnatural.

    Another problem is that, even in the promising example of CS-note 34, some weakenings to the matching of coloured rippling were made. This is a direct result of the accidental hypothesis not containing the deeper relationship expected by rippling. It may be that the weakenings applied are small and well behaved. However, it is difficult to envisage the fundamental behaviour of rippling remaining intact after even a few adjustments to the matching process.

    If weakenings to rippling matching are rejected then their is a consequential problem of missing hypotheses that are key to proof. A method might be found to include these hypothesis later in the proof. However, this would essentially repeat the same refinements suggested to monochromatic rippling to include extra hypothesis. Such a hybrid approach of coloured rippling refined to include extra hypothesis not found by matching may successfully cover additional problems. However, it would seem more appropriate to separately implement monochromatic rippling with a refinement for extra hypothesis and regular coloured rippling[5].

3.2 From invariant to exception freedom check

Exception freedom VC_ns ($EFVC_n$s) are from cutpoints to checks inserted by the EXAMINER. The general pattern is shown in Figure 1. Note how the checks added to generate $EFVC_n$s are all final nodes. This means that the cutpoint to cutpoint property seen in the invariant to invariant case is never seen for $EFVC_n$s, irrespective as to

[5]This is unfortunate. Coloured ripple proofs would be more straight forward as they do not require the final proof stages that seem necessary for monochromatic rippling. However, incorporating accidentally matching hypothesis within coloured rippling does not work.

whether or not they occur inside loops. The ellipsis indicate that their can be any number of check statements and their can be any number of paths between cutpoints and from cutpoints to check statements.

Of most interest to this note are EFVC$_n$s describing the traversal from the invariant to an exception freedom check. The proof of these will require invariant discovery, which is the major hurdle to automated program proof.

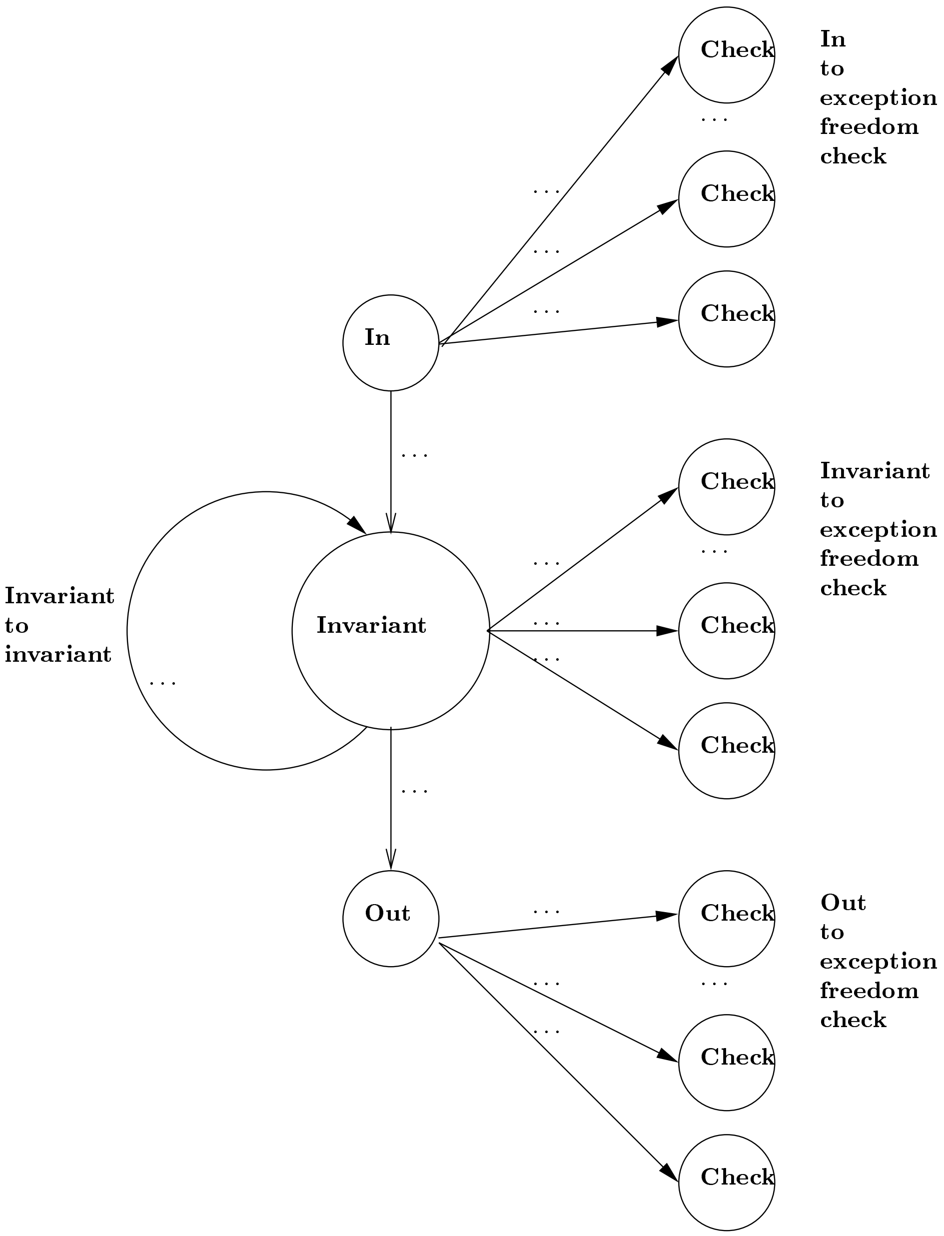

Figure 1: Generation of exception freedom VCs

3.2.1 Automating exception freedom proofs

The structure of the conclusion of EFVC$_n$s depends on the structure of the exception freedom checks. The structure of the hypotheses of EFVC$_n$s arises from the structure of the starting cutpoint, in these cases the invariant. Their is no guaranteed relationship between the structure of the conclusion and the structure of the hypotheses.

However, to prove the conclusion of a EFVC$_n$ it is necessary to select an invariant. It is likely that a good invariant, introducing information allowing the discharge of EFVC$_n$s, will have a strong similarity to the structure of the conclusion.

Proving EFVC$_n$s that arise from paths starting from an invariant will probably require invariant discovery. The invariant discovered must also be proved to meet its invariant property. Thus there is a balancing act between adding properties which led to straight forward exception freedom proofs while also continuing to support the legality and proof of the invariant.

3.2.2 Rippling does not work here

The strong similarity seen between good invariants and $\mathrm{EFVC_n}$s suggests that rippling may be applicable. However, this strong similarity lacks the deeper relationship seen in invariant to invariant $\mathrm{VC_n}$s. Rippling is not applicable here.

3.2.3 Invariants as conjunction of bounds

Most $\mathrm{EFVC_n}$s will require proving that a variable lies within some bound. For every variable that is referenced it is thus appropriate to add two conjuncts to the invariant, describing the variables upper and lower bound. The result is that invariants added to aid exception freedom proofs tend to include a conjunction of bounds rather than a single property, as might be seen in partial correctness.

Note that a variable may be referenced more than once in the code, generating more than one $\mathrm{EFVC_n}$. In such cases it might be thought appropriate to add multiple bounds on the variable in the invariant. However, this is only correct if the bounds are equal, otherwise the invariant will be contradictory.

Note that the situation where their are multiple equal bounds on a variable is expected to occur occasionally in practise. Consider finding the smallest element in an array indexed by type `T` between a supplied lower `L` and upper `U` bound also of type `T`. The array will be indexed by a variable `I` with upper bound `T'Last`. Taking into account the behaviour of the algorithm, the upper bound on `I` is actually `U`. However, the the upper bound of `U` is also `Type'Last`. Thus the upper bound on `I` can be both `Type'Last` and `U` as they reduce to the same value. Adding both upper bounds is likely to aid exception freedom proofs as their is more information available.

It might be expected that the bounds in the invariant should describe the tightest fit. The lower bound on a variable should be the highest possible lower bound and the upper bound should be the lowest possible upper bound. In many cases proofs may be possible where the bounds specified in the invariant are not the tightest possible. It may even be the case that these proofs are more straight forward. It would not be contradictory to include the weaker bounds on the invariant alongside the tightest possible bound. Including such weaker bounds may be a possible refinement made to the invariant as a result of encountering difficult proofs with the tightest bounds.

4 Automating SPARK exception freedom proofs

There are three different problems that must be successfully solved to fully automate the proof of $\mathrm{EFVC_n}$s that occur within a loop:

- **Invariant discovery -** An invariant must to be generated that is sufficiently descriptive to allow for the proof of every $\mathrm{EFVC_n}$.
- **Proving invariant to invariant -** The invariant to invariant $\mathrm{VC_n}$ must be provable by the proof planner.
- **Proving invariant to exception freedom check -** The invariant to exception freedom check $\mathrm{EFVC_n}$ must be provable by the proof planner.

These problems do not occur in isolation. The trick is balancing the strength of the invariant and complexity of proofs to find a combination where the whole process can be performed automatically.

4.1 Invariant discovery

Plans for invariant discovery have so far included the following flawed approaches:

- **Guidance from coloured rippling -** Coloured rippling is not applicable for the proofs and thus does not aid in guiding invariant discovery.
- **Code level analysis**[6] **-** The reasoning required ultimately involves heuristics and theorem proving that can not be naturally incorporated into this stage.

[6] This was not pursued for long enough to spawn a note. The approach involved an algorithmic analysis of the output from CPA. This might be reasonable for extracting type information or weak properties. However, any such algorithmic approach would produce a subset of the results that an automated reasoning approach would reveal.

- **Backwards application of rippling with some middle out reasoning -** Rippling needs a syntactically deep structure for this approach to be successful. EFVC_ns do not yield such structures. Further, problems were encountered bridging the gap between single proofs and the general form required for an invariant.

Some additional information on these flawed approaches is given in §6. Fortunately, it does appear that the strategy taken in CS-note 34 can be applied to more general formula using various common generalisations. The details are currently sketchy and will be explored in a later note.

4.2 Proving invariant to invariant

Three different approaches have been considered:

- **Conditional wave rules -** Monochromatic rippling is not sufficient on its own as their is often a need to use extra hypotheses. This might be achieved through conditional wave rules. Where a conditional wave rule is applied its conditions are tackled as separate subgoals. Each of these subgoals can then be proved with access to all of the hypotheses.

  This approach requires the careful formulation of suitably general conditional wave rules. Conditional wave rules have a tendency to represent larger jumps in the proof. This can be problematic as these larger jumps may not be fitting with the incremental process of rippling.

  Note that this does not rule out conditional wave rules. Many conditional rules can be naturally incorporated within rippling. However, it does exclude some richer wave rules that would be required to support the extra hypotheses reasoning in these proofs.

- **Rippling to a residue -** Rippling tries to eliminate the syntactic changes made to variables between iterations. The general approach is to ripple mismatching syntax outwards until a rule apples allowing the mismatching syntax to be eliminated, for example subtracting an equal amount from both sides of an equation.

  Invariants aimed at exception freedom proofs typically state that a variable lies within a bound. This bound many be expressed in terms of other variables. Where dependent on other variables the ripple proof will result in two different structures emerging:

  - **Induction hypothesis -** The induction hypothesis will emerge as the syntactic differences are eliminated.
  - **Residue -** The changes made to the induction hypothesis can not be rippled away. Instead they gather together as a residue, describing constraints on the other variables.

  The induction hypothesis in the conclusion can be eliminated by matching with the real induction hypothesis. This leaves the residue to be proved. This smaller subgoal can be tackled using regular[7] theorem proving, outside the rippling approach.

- **Integrated simplification -** It is possible to augment rippling with a terminating simplification procedure. This procedure has the ability to eliminate syntax inside the wave front, external to the ripple strategy. This approach could be used to perform additional reasoning whenever the ripple proof became stuck.

  In practise it is expected that this approach would be essentially the same as rippling to a residue. The point at which a ripple proof gets stuck is the point where the residue is reached. The difference is that this integrated simplification must be terminating. This property will be difficult to maintain if requiring fairly sophisticated reasoning.

  Note that the normal implementation of integrated simplification is regarded as an unfortunate, but useful, extension to rippling. It is preferable to avoid this if possible.

The technique of rippling to a residue and then tackling this as a separate proof is the most natural way to exploit normal rippling while still catering for these proofs.

4.3 Proving invariant to exception freedom check

Once a good invariant has been found these proofs can be quite straight forward. It is conjectured that the technique constructed to discharge the residue from rippling will also be applicable to these proofs[8].

[7] Probably proof planning with various methods tuned for reasoning with inequalities.

[8] This is a little surprising. The invariant has been selected to allow for exception freedom proofs. To remain invariant the invariant must take a general form. Rippling successfully removes the general part of the invariant, leaving the residue as a single case proof. Like the EFVC_ns these single case proofs do not have any recursive aspects. Further, their relationship to the invariant means they have light structural similarity to the EFVC_ns.

5 Example using monochromatic rippling and residue proof

5.1 Phil Thornley example

```
procedure SumArray(Data_Set: in DataArray ; Result: out Integer)
is
begin
    Result:=0;
    For I in Index loop
        --# assert Result <= I*100;
        if Data_Set(I) >= 0 and Data_Set(I) <= 100 then
            Result := Result + Data_Set(I);
        end if;
    end loop;
end SumArray;
```

5.2 procedure_sumarray_3

5.2.1 The full VC

```
For path(s) from assertion of line 8 to assertion of line 8:

procedure_sumarray_3.
H1:    result <= loop__1__i * 100 .
H2:     for_all (i___1: integer, ((i___1 >= index__first) and (
           i___1 <= index__last)) -> ((element(data_set, [
           i___1]) >= integer__first) and (element(data_set, [
           i___1]) <= integer__last))) .
H3:    loop__1__i >= index__first .
H4:    loop__1__i <= index__last .
H5:    loop__1__i >= index__first .
H6:    loop__1__i <= index__last .
H7:    loop__1__i >= index__first .
H8:    loop__1__i <= index__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__first .
H14:   loop__1__i <= index__last .
H15:   not (loop__1__i = index__last) .
        ->
C1:    result + element(data_set, [loop__1__i]) <= (
           loop__1__i + 1) * 100 .
C2:     for_all (i___1: integer, ((i___1 >= index__first) and (
           i___1 <= index__last)) -> ((element(data_set, [
           i___1]) >= integer__first) and (element(data_set, [
           i___1]) <= integer__last))) .
C3:    loop__1__i + 1 >= index__first .
C4:    loop__1__i + 1 <= index__last .
```

5.2.2 Establish the goal for monochromatic rippling

Hyp $result \leq i * 100$

Conc $\boxed{\underline{\boldsymbol{result}} + element(data\_set, [i])}^{\uparrow} \leq \boxed{(\underline{\boldsymbol{i}} + 1)}^{\uparrow} * 100$

5.2.3 Wave rules

Number	(1)
Theroem	$(X + Y) * Z = (X * Z) + (Y * Z)$
Wave rule	$\boxed{(\underline{\boldsymbol{X}} + Y)}^{\uparrow} * Z \Rightarrow \boxed{(\underline{\boldsymbol{X * Z}}) + (Y * Z)}^{\uparrow}$

Number	(2)
Theroem	$(W \leq Y) \wedge (X \leq Z) \rightarrow (W + X) \leq (Y + Z)$
Wave rule	$(\boxed{\underline{\boldsymbol{W}} + X}^{\uparrow}) \leq (\boxed{\underline{\boldsymbol{Y}} + Z}^{\uparrow}) \Rightarrow \boxed{(\underline{\boldsymbol{W \leq Y}}) \wedge (X \leq Z)}^{\uparrow}$

5.2.4 Start rippling

$$\boxed{\underline{\boldsymbol{result}} + element(data\_set, [i])}^{\uparrow} \leq \boxed{(\underline{\boldsymbol{i}} + 1)}^{\uparrow} * 100 \qquad \textbf{(1)}$$

$$\boxed{\underline{\boldsymbol{result}} + element(data\_set, [i])}^{\uparrow} \leq \boxed{(\underline{\boldsymbol{i * 100}}) + (1 * 100)}^{\uparrow} \qquad \textbf{(2)}$$

$$\boxed{\underline{\boldsymbol{result \leq i * 100}} \wedge (element(data\_set, [i]) \leq (1 * 100)}^{\uparrow} \qquad \textbf{Fertilise}$$

$$element(data\_set, [i]) \leq (1 * 100) \qquad \textbf{Residue}$$

5.2.5 Rewrite rules

Number	(3)
Theroem	$1 * X = X$
Rewrite rule	$1 * X \Rightarrow X$

5.2.6 Prove residue

$$[\ldots element(data\_set, [i]) \leq 100 \ldots] \rightarrow element(data\_set, [i]) \leq (1 * 100) \qquad \textbf{(3)}$$

$$[\ldots element(data\_set, [i]) \leq 100 \ldots] \rightarrow element(data\_set, [i]) \leq 100 \qquad \textbf{Match with hypothesis}$$

5.3 procedure_sumarray_4

5.3.1 The full VC

```
procedure_sumarray_4.
H1:    result <= loop__1__i * 100 .
H2:     for_all (i___1: integer, ((i___1 >= index__first) and (
           i___1 <= index__last)) -> ((element(data_set, [
           i___1]) >= integer__first) and (element(data_set, [
           i___1]) <= integer__last))) .
H3:    loop__1__i >= index__first .
H4:    loop__1__i <= index__last .
H5:    loop__1__i >= index__first .
H6:    loop__1__i <= index__last .
H7:    loop__1__i >= index__first .
H8:    loop__1__i <= index__last .
H9:    not ((element(data_set, [loop__1__i]) >= 0) and (element(
           data_set, [loop__1__i]) <= 100)) .
H10:   not (loop__1__i = index__last) .
        ->
C1:    result <= (loop__1__i + 1) * 100 .
C2:     for_all (i___1: integer, ((i___1 >= index__first) and (
           i___1 <= index__last)) -> ((element(data_set, [
           i___1]) >= integer__first) and (element(data_set, [
           i___1]) <= integer__last))) .
C3:    loop__1__i + 1 >= index__first .
C4:    loop__1__i + 1 <= index__last .
```

5.3.2 Establish the goal for monochromatic rippling

Hyp $result \leq i * 100$

Conc $result \leq \boxed{(\underline{\boldsymbol{i}} + 1)}^{\uparrow} * 100$

5.3.3 Wave rules

Number	(4)
Theroem	$(X + Y) * Z = (X * Z) + (Y * Z)$
Wave rule	$\boxed{(\underline{\boldsymbol{X}} + Y)}^{\uparrow} * Z \Rightarrow \boxed{(\underline{\boldsymbol{X * Z}}) + (Y * Z)}^{\uparrow}$

Number	(5)
Theroem	$((W \leq X) \wedge (0 \leq Y) \rightarrow W \leq (X + Y))$
Wave rule	$(W \leq \boxed{(\underline{\boldsymbol{X}} + Y)}^{\uparrow} \Rightarrow \boxed{\underline{(\boldsymbol{W} \leq \boldsymbol{X})} \wedge (0 \leq Y)}^{\uparrow})$[9]

5.3.4 Start rippling

$$result \leq \boxed{(\underline{\boldsymbol{i}} + 1)}^{\uparrow} * 100 \qquad \textbf{(4)}$$

$$result \leq \boxed{(\underline{\boldsymbol{i * 100}}) + (1 * 100)}^{\uparrow} \qquad \textbf{(5)}$$

$$\boxed{\underline{(\boldsymbol{result} \leq (\boldsymbol{i} * \mathbf{100}))} \wedge (0 \leq 1 * 100)}^{\uparrow} \qquad \textbf{Fertilise}$$

$$0 \leq (1 * 100) \qquad \textbf{Residue}$$

5.3.5 Rewrite rules

Number	(6)
Theroem	$1 * X = X$
Rewrite rule	$1 * X \Rightarrow X$

5.3.6 Prove residue

$$[\ldots] \rightarrow 0 \leq (1 * 100) \qquad \textbf{(6)}$$

$$[\ldots] \rightarrow 0 \leq 100 \qquad \textbf{Trivially true}$$

[9] This rule was suggested by Andrew. I previously attempted this using the correct but cumbersome:
$[(X \geq 0) \wedge (Y \geq 0)] \rightarrow (W \leq \boxed{(\underline{\boldsymbol{X}} + Y)}^{\uparrow} \Rightarrow \boxed{\underline{(\boldsymbol{W} \leq \boldsymbol{X})} \wedge (W \leq Y)}^{\uparrow})$
This illustrates that a key factor in the success of rippling is the quality of the rules that are available. Finding a good number of quality rules will be an important task in implementing this approach in practise.

6 Some notes on invariant discovery

The following describes some of the problems encountered in invariant discovery.

6.1 Proof patching by learning from failure

In theorem proving, the failure to prove a conjecture can lead to insights that later allow for the same conjecture to be proved. This approach has been successfully employed in assisting in the proof of VC_ns [3]. Where failing to prove a VC_n during proof planning the points of failure can provide strong hints towards patches allowing the proof to succeed. Such failures might indicate one of the following proof patches:

- **Add missing lemma -** A lemma provides additional information about some domain allowing greater transformations to be performed during a proof. Through carefully selecting lemmas the extra transformations introduced are suitable to complete the proof.
- **Add missing property -** The VC_ns are generated based on properties in the code specified as preconditions, postconditions and invariants. By appropriately modifying these properties each VC_n can be tackled in a context where their is sufficient information for proof. Note that a consequence of modifying properties is that the VC_ns will also change.

6.2 Learning from failure of exception freedom proofs

Learning from failure is appropriate where, at the point of failure, their is a structure hinting at the nature of the patch. Through experience it has been found that the deeper the proof progresses before failing, the more likely a good patch will be found.

Unfortunately, where exception freedom proofs fail, they tend to do so very quickly. Their is a fundamental gap between the failure of an exception freedom proof and the discovery of the generalised invariant that would allow for proof. It remains conceivable that learning from failure can be used to refine a reasonable existing invariant. However, it is difficult to envisage how this might be used to initially be discover the invariant.

6.3 Learning from failure of best invariant

The ideal hypothesis for each $EFVC_n$ is an exact copy of the conclusion. It is then trivial to prove the $EFVC_n$ by matching the conclusion with this hypothesis. Where the exception freedom checks are from the invariant this ideal hypothesis can be provided by annotating the invariant accordingly. Of course, this will almost certainly result in an invariant which is not invariant.

This does, however, provide a starting point from which analysis can progress. The failure of the invariant to invariant VC_n may provide hints as to what a more suitable invariant would look like. A similar technique is used to discover invariants for partial correctness proofs, where the postcondition is directly copied into the invariant and the failure to prove the invariant to invariant VC_n case leads to helpful insights (see CS-note 10 for an example of this).

For example, the following invariant to invariant VC_n is reached where applying this technique to the Phil Thornley example:

```
For I in Index loop
    --# assert Result+Data_Set(I)<=Integer'Last;
    if Data_Set(I) >= 0 and Data_Set(I) <= 100 then
        Result := Result + Data_Set(I);
    end if;
end loop;

For path(s) from assertion of line 8 to assertion of line 8:

procedure_sumarray_3.
H1:    result + element(data_set, [loop__1__i]) <= 32767 .
H2:    for_all(i___1 : integer, 0 <= i___1 and i___1 <= 9 -> - 32768 <= element(
          data_set, [i___1]) and element(data_set, [i___1]) <= 32767) .
H3:    loop__1__i >= 0 .
H4:    loop__1__i < 9 .
H5:    element(data_set, [loop__1__i]) >= 0 .
H6:    element(data_set, [loop__1__i]) <= 100 .
H7:    result + element(data_set, [loop__1__i]) >= - 32768 .
       ->
C1:    result + element(data_set, [loop__1__i]) + element(data_set, [loop__1__i
          + 1]) <= 32767 .
```

```
procedure_sumarray_4.
H1:    result + element(data_set, [loop__1__i]) <= 32767 .
H2:    for_all(i___1 : integer, 0 <= i___1 and i___1 <= 9 -> - 32768 <= element(
          data_set, [i___1]) and element(data_set, [i___1]) <= 32767) .
H3:    loop__1__i >= 0 .
H4:    loop__1__i < 9 .
H5:    element(data_set, [loop__1__i]) < 0 or 100 < element(data_set, [
          loop__1__i]) .
       ->
C1:    result + element(data_set, [loop__1__i + 1]) <= 32767 .
```

Unfortunately the fundamental gap between the single proof and the general invariant prevents any straight forward failure patch:

```
result <= i*100
```

Is very different from:

```
result + element(data_set, [i + 1]) <= 32767
result + element(data_set, [i]) + element(data_set, [i + 1]) <= 32767
```

It is possible to find `result <= i*100` from looking at the surrounding information, however the failure of the best invariant gives no useful guidance.

7 Conclusions

Despite some promising initial results, coloured rippling is rarely applicable in program proof. However, regular monochromatic rippling plus additional reasoning to tackle any residue is applicable.

Attempts at invariant discovery for exception freedom proofs by examining the failed VC_ns have not been successful. Their is not sufficient information in the initial failing proofs to naturally suggest the structure of the invariant. However, from the behaviour of exception freedom proofs, it is known that the invariant will be a conjunction of bounds on variables. This weak constraint on the invariant is sufficient to make progress using additional pattern based reasoning of the sort seen in CS-note 34.

References

[1] Alan Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. Research Paper 567, 1991. In the Journal of Artificial Intelligence.

[2] Alan Bundy, F. van Harmelen, A. Smaill, and A. Ireland. Extensions to the rippling-out tactic for guiding inductive proofs. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 132–146. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh a s DAI Research Paper 459.

[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.

[4] Tetsuya Yoshida, Alan Bundy, Ian Green, Toby Walsh, and David Basin. Coloured rippling: An extension of a theorem proving heuristic. In A. G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence*, pages 85–89, Chichester, 8–12 1994. John Wiley and Sons.