CS-Note* 63

Andrew Ireland
October 17, 2003

Invariant Patterns for Program Reasoning

1 Introduction

This note continues the theme of proof planning for partial correctness. As such, it represents the evolution of the ideas described in notes 10, 30 and 62, as well as the work reported in [8]. This line of investigation is motivated by a desire to combine and extend current approaches to inductive invariant discovery. The two basic approaches are known as *bottom-up* and *top-down*. Within a bottom-up approach inductive invariants are generated by analysing the code directly with no reference to proof annotations (specifications). In contrast, within a top-down approach proof annotations provide the basis for generating inductive assertions. In practise, a third kind of analysis, what we will call *proof-failure* analysis, also plays a crucial role within invariant generation.

In note 10, the lack of bottom-up analysis was identified as a weakness of our proof critics approach to invariant discovery. The proof critics approach can be viewed as a form of top-down analysis integrated with proof-failure analysis, *i.e.* inductive invariants are generated from the given proof annotations (postcondition) via the analysis of proof planning failures.

This need for both bottom-up and top-down analysis is well understood [10, 13]. In terms of experimental systems, VISTA [5] is the most sophisticated and implements many of the ideas discussed in [13]. However, Wegbreit[1] hinted that a closer relationship between the heuristic guidance and the theorem proving components would lead to a more effective approach to the problem of invariant generation.

In [8], we believe that progress was made in this direction, *i.e.* high-level theorem proving guidance (proof plans) were integrated with invariant discovery heuristics via proof critics. In note 30 the ideas were taken further with a proposal to use high-level theorem proving guidance (proof plans) and high-level program knowledge (algorithmic patterns) in combination with invariant discovery heuristics, again using proof critics. The algorithmic patterns aspect of this proposal has now been implemented by Tommy in his AutoGap system [6].

Here we consider an extension to the ideas summarized above. We propose the use of *invariant patterns* as a means of achieving an effective integration of bottom-up, top-down and proof-failure analysis. An invariant pattern represents an invariant schema together with a selection criteria. It is envisage that the selection criteria may give rise to a fully instantiated schema in some contexts. The role of invariant patterns within the automation of partial correctness proofs is as follows:

1. For a given program and specification, a set of applicable invariant patterns is used to generate candidate invariants.
2. The code is annotated with the candidate invariants and verification conditions (VCs) are generated.
3. Proof planning of the VCs is attempted with the expectation, that in the case of schematic VCs, the schemas will be instantiated as a side-effect of the planning.
4. Proof failure analysis is attempted when the proof methods break-down. It is envisaged that failure analysis may result in the generation of additional schematic invariants.

Note that invariant patterns relate to both bottom-up and top-down analysis. Note also that where an invariant pattern gives rise to an invariant schema, the corresponding VCs will also be schematic. In terms of the SPARK approach [1], this means that verification will be an iterative process involving both the NuSPADE and the Examiner. That is, refinements to invariant schemas during NuSPADE's proof planning will be fed back into the Examiner in order to generate a refined set of VCs.

2 Analysis for program reasoning

Below we explore the potential for invariant patterns. We begin by considering each of the three kinds of analysis highlighted above before exploring a non-trivial verification example in detail.

*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] Wegbreit was one of the developers of VISTA.

$mono\_dec(V, W)$:	Means that V is a loop counter that monotonically decreases during the execution of loop W.
$mono\_inc(V, W)$:	Means that V is a loop counter that monotonically increases during the execution of loop W.
$constant(V, W)$:	Means that V is a loop counter that is constant during the execution of loop W.

Figure 1: Meta predicates

2.1 Bottom-up analysis

Traditional bottom-up analysis techniques generate light-weight invariant properties, such as relationships between loop counter variables. Such invariants are typically required in order to complete a proof. In addition to generating invariants, our extended bottom-up analysis technique generates information that supports both top-down and proof-failure analyses. This involves identifying common patterns in terms of how program variables and data structures are used within an algorithm. Such patterns can be used in guiding the discovery of invariants. By way of illustration, invariant discovery involves identifying the relationship between "work done" and "work still to do" during a computation. Within the context of array based programs, this relationship typically corresponds to partitioning an array, where partition boundaries are defined in terms of loop counter variables. Knowing how counter variables change during a computation provides guidance in determining the structure of invariants. We explicitly represent these changes by means of predicates as defined in figure 1. Making these notions explicit means that the information can be exploited by our top-down and failure-analysis techniques, as will be illustrated later. Note that where loops are nested an outer-loop counter will typically remain unchanged during the execution of an inner-loop. This notation is expressed by the predicate *constant*. It is envisaged that this set of predicates will evolve as new patterns between algorithms and invariants are identified.

2.2 Top-down analysis

Our top-down analysis technique is novel in that it generates schematic invariants. To illustrate the general mechanism, consider the following pattern of postcondition for an array based program:

$$(\forall q : int.\ ((l \leq q) \wedge (q \leq u)) \rightarrow P(q)) \tag{1}$$

Note that l and u denote the lower and upper bounds on q. Typically these bounds will correspond to the array bounds and the predicate $P(q)$ will define a property of the array. Weakening a postcondition corresponding to (1) can be achieved by restricting the range of q. We call this pattern of invariant *range restriction*. In order to tailor this pattern to a particular algorithm we use the information generated via bottom-up analysis. Let us assume that the predicate P specifies a property of an array t, moreover that t is partitioned with respect to a loop counter i with accessible range l to i. If i is monotonically increasing then this suggests that (1) should be weakened by replacing u, the upper bound on q. The question is in determining the identity of the replacement term. The conventional strategy involves generate and test, where test involves a theorem prover. Here we propose the use of meta-variables, this allows us to delay the choice until we plan the proof. In terms of (1), this gives an invariant schema of the form:

$$(\forall q : int.\ ((l \leq q) \wedge (q \leq F(i))) \rightarrow P(q))$$

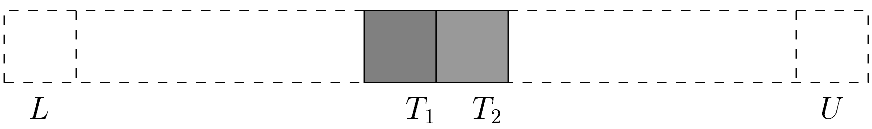
Note that F denotes a second-order meta-variable. If the accessible range associated with i was i to u and i was monotonically decreasing, then the l would have been replaced by $F(i)$. Within the context of nested loops, invariant schemas are generated first for the outer loop then for inner loops. An inner loop will obviously inherit the invariant schemas generated for the outer loops.

2.3 Proof-failure analysis

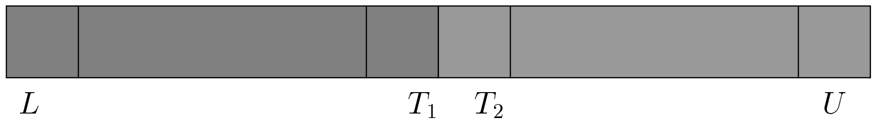
Given a failed proof attempt, a common theorem proving strategy is to conjoin the failed goal onto the original conjecture (invariant) and attempt the proof again. We extend this strategy by introducing two alternative generalization steps. We describe each generalization in terms of a proof critic.

2.3.1 Range generalization critic

Using a "picture" notation, consider the following array:



Note that the elements indexed by T_1 and T_2 are adjacent while L and U denote the lower and upper bounds on the array respectively. When proving a relationship between adjacent elements it is often the case that one needs to consider a range of elements rather than just the individuals, *i.e.*



Considering a range of elements provides a stronger invariant (inductive) hypothesis. Here we represent this observation as a proof critic. The preconditions for what we call the *range generalization critic* are as follows:

1. A goal is unprovable within the current proof context and matches the following pattern:

$$\underbrace{ele(A, [T_1])\ Rel\ ele(A, [T_2])}_{\textbf{blocked}}$$

   where A denotes an array, terms T_1 and T_2 index adjacent elements within the array A and Rel denotes a transitive relation.

2. Terms T_1 and T_2 contain a counter variable in common.

Note that precondition 2 exploits the notion of a partition as outlined in §2.1. The associated patch involves generalizing with respect to both T_1 and T_2, *i.e.*

$$\begin{aligned}&(\forall X : int.((L \leq X) \wedge (X \leq T_1)) \rightarrow \\ &\quad (\forall Y : int.((T_1 < Y) \wedge (Y \leq U)) \rightarrow ele(A, [X])\ Rel\ ele(A, [Y])))\end{aligned}$$

This generalized goal represents an auxiliary invariant that is then conjoined onto the original invariant. We envisage situations where a weaker generalization may be appropriate. For instance, if T_1 denotes a constant then only T_2 would be generalized, and vice versa (see figure 2).

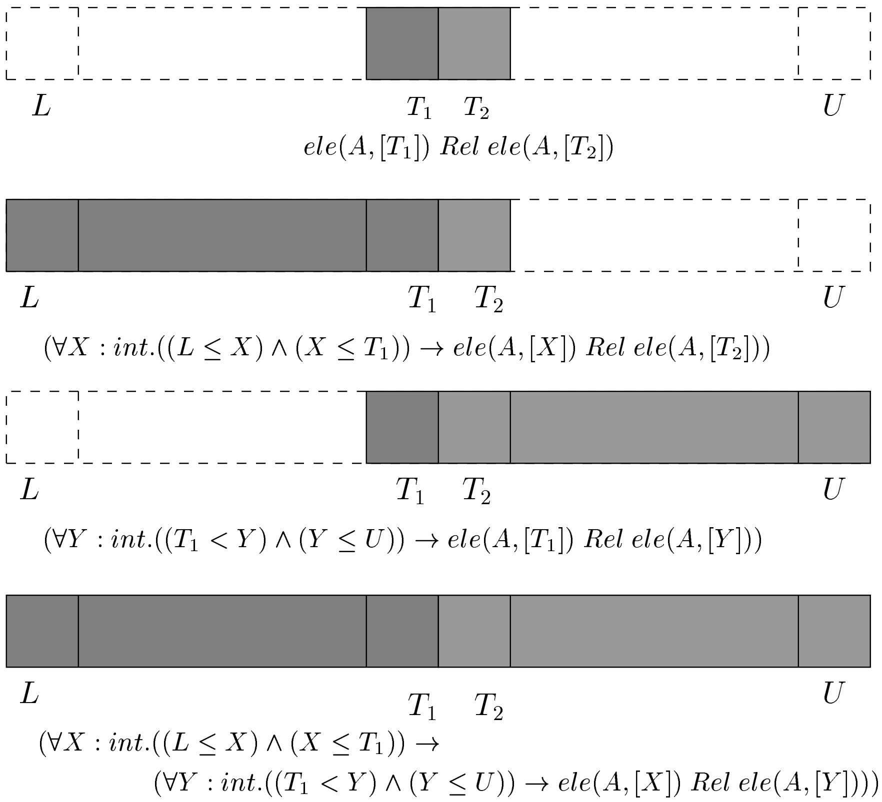


Figure 2: Range Generalization Choices

2.3.2 Difference generalization critic

Our second generalization critic builds upon the *rippling* proof plan. Rippling is a rewriting technique in which annotations are used to guide the selection of rewrite rules. Selection is based upon a difference reduction heuristic. The difference between a goal and a hypothesis are annotated, where the annotations are called *wave-fronts*. Annotated rewrite rules, known as wave-rules, are then used to reduce the differences between goal and hypothesis. Rippling is successful if a match between the goal and the hypothesis is made possible. This matching is known as *fertilization*. A completely formal account of the ripple method can be found in [2, 3]. Our second generalization critic is motivated by the observation that an unproven goal resulting from a successful fertilization often requires a subsequent ripple proof. This in turn requires difference matching the post-fertilization goal against another hypothesis (invariant) within the proof context. This change of the "rippling focus" breaks down if the proof context is missing the hypothesis (invariant) that is necessary for the proof to proceed. The patch uses the available wave-rules (background theory) to guide the discovery of the missing hypothesis (invariant). That is, we look for wave-rule matches that fail because of missing wave-front annotations within the goal. The preconditions to the critics are enumerated below:

1. A post-fertilization goal is unprovable within the current proof context, *i.e.*

$$\underbrace{f(g(c(a,b)))}_{\textbf{blocked}}$$

2. There exists a wave-rule that matches modulo missing wave-front annotations, *i.e.*

$$g(\ \boxed{c(\underline{a},b)}^{\uparrow}\ ) \Rightarrow \boxed{h(\underline{g(a)},\delta)}^{\uparrow}$$

3. If the wave-rule is applied then the instantiation of δ will be provable in the current proof context.

Note that shading is used to represent wave-front annotations. Here annotations are missing from the goal so that the wave-rule is not applicable. With regards to precondition 2, a criteria for evaluating the closeness of a near-miss would be necessary in order to rank candidate wave-rules. Note that precondition 3 further constrains the search for an auxiliary invariant by looking ahead into the proof planning. Note that precondition 3 is not essential. The associated patch involves eliminating the terms within the unproven goal that correspond to the missing wave-front annotations. In the general case this gives $f(g(a))$. This modified formula represents an auxiliary invariant that is then conjoined to the original invariant. Where multiple sources for the missing wave-front annotations exist then alternative schemas need to be considered. Using *rippling in reverse* [2] alternative sources for the missing annotations can be identified. Each alternative gives rise to an unique candidate invariant schema. Again information gathered during bottom-up analysis can be used to impose an ordering on the schemas, as will be illustrated later.

3 Verification of a bubble sort program

We now apply the ideas described above to the verification of bubble sort. The SPARK version of bubble sort that is verified is given in figure 3. Note that the code is annotated with pre- and postconditions, but no loop invariants are specified. Moreover, given that the code involves 2 nested loops then 2 loop invariants will be required in order to prove partial correctness. Using the bubble sort example, we illustrate below the 3 kinds of analysis that were described in §2.

3.1 Bottom-up analysis

In terms of proof construction, our bottom-up analysis of the bubble sort code generates a couple of invariant properties. Firstly, the analysis identifies the bounds on loop counter I: $1 \leq i \ \wedge \ i \leq last$. Secondly, bounds on loop counter J are also identified: $i \leq j \ \wedge \ j \leq last$. Note that the second invariant is with respect to the inner loop only. These are generated by analysing the initial and final values of both loop counters. In terms of proof search, the following properties are established:

$$mono\_inc(i, for\_loop\_i) \tag{2}$$
$$mono\_dec(j, for\_loop\_j) \tag{3}$$
$$constant(i, for\_loop\_j) \tag{4}$$

Note that these meta predicates are defined in figure 1.

[2]This is analogous to the *induction revision critic* (see [7]) where *rippling in reverse* is used to determine alternative induction schemas.

```
package BubbleSort is

   subtype Index_Type is Integer range 0 .. 9;
   type Array_Type is array (Index_Type) of Integer;

   --# function Ordered(A: Array_Type; L, U: Index_Type) return Boolean;
   --# function Perm(A, B: Array_Type) return Boolean;

   procedure Bubble_Sort(Table: in out Array_Type);
   --# derives Table from Table;
   --# pre  true;
   --# post Ordered(Table, 0, Index_Type'Last) and
   --#      Perm(Table, Table~);

end BubbleSort;
package body BubbleSort is

   procedure Bubble_Sort(Table: in out Array_Type)
   is
     T: Integer;
   begin
     for I in Index_Type range 1 .. Index_Type'Last loop
        for J in reverse Index_Type range I .. Index_Type'Last loop
           if Table(J-1) > Table(J) then
              T:= Table(J-1);
              Table(J-1):= Table(J);
              Table(J):= T;
           end if;
        end loop;
     end loop;
   end Bubble_Sort;

end BubbleSort;
```

Figure 3: A SPARK implementation of Bubble Sort

3.2 Top-down analysis

We now turn to the specification of bubble sort. The predicate `Ordered`, that forms part of the postcondition, is defined as follows:

$$ordered(A, L, U) \leftrightarrow (\forall P : int.(L \leq P \wedge P < U) \rightarrow ele(A, [P]) \leq ele(A, [P + 1])) \tag{5}$$

Unfolding using (5), the postcondition becomes:

$$(\forall p : int.(0 \leq p \wedge p < last) \rightarrow ele(table, [p]) \leq ele(table, [p + 1])) \tag{6}$$

This is a candidate for the range restriction invariant pattern. From our bottom-up analysis of the bubble sort code (see §3.1), we identify nested loops. The outer-loop is associated with a single partition defined by (2), while the inner-loop is associated with partitions defined by (2) and (3). As mentioned above, we consider the outer most loop first, then the second outer most loop and so on. So in weakening (6) we consider the partition defined by I. By (2), we know that I monotonically increases during the execution of the outer loop, which suggests replacing $last$ by $F_1(i)$ to give an outer-loop invariant schema of the form:

$$(\forall p : int.(0 \leq p \wedge p < F_1(i)) \rightarrow ele(table, [p]) \leq ele(table, [p + 1])) \tag{7}$$

This invariant schema is inherited by the inner-loop. By (4) we known that I remains constant within the inner-loop. As a consequence we only consider a partition defined by (3) as the basis for a further weakening of (7). By (3), we know that J monotonically decreases during the execution of the inner loop, which suggests replacing 0 by $F_2(j)$ within (7) to give:

$$(\forall p : int.(F_2(j) \leq p \wedge p < F_1(i)) \rightarrow ele(table, [p]) \leq ele(table, [p + 1])) \tag{8}$$

Clearly the more meta-variables that appear within a schema the greater the search control problems. To minimize these problems we organize the search by ordering schemas according to the number of meta-variables they contain. For instance, schema (7) has less meta-variables than schema (8), so proof planning with respect to (8) will only be undertaken if a proof of (7) fails completely.

3.3 Proof planning and proof-failure analysis

The proof planning requires a number of iterations. Each iteration gives rise to refinement of the loop invariants. The refinements are achieved through the proof planning and via proof-failure analysis. Success corresponds to the generation of a concrete set of invariants (proof annotations) and proof tactics for the associated VCs.

3.3.1 First proof planning attempt

In the case of bubble sort, the starting point is a VC relating to property (5), where we have a schematic hypothesis of the form

$$ordered(table, 0, F_1(i)) \tag{9}$$

and a schematic goal of the form:

$$ordered(table, 0, F_1(i+1)) \tag{10}$$

Annotating (10) with respect to (9) gives:

$$ordered(table, 0, F_1(\boxed{(i+1)}^{\uparrow}))$$

Unfolding using definition (5) gives:

$$(\forall p : int.((0 \leq p) \wedge (p < F_1(\boxed{(i+1)}^{\uparrow}))) \rightarrow ele(table, [p]) \leq ele(table, [p+1])) \tag{11}$$

We can now ripple (11), using (26) to give:

$$(\forall p : int.((0 \leq p) \wedge (p < \boxed{(i - F_2(i)) + 1}^{\uparrow})) \rightarrow ele(table, [p]) \leq ele(table, [p+1])) \tag{12}$$

Using wave-rule (27), (12) ripples to give:

$$\boxed{\begin{array}{l}(\forall p : int.((0 \leq p) \wedge (p < i - F_2(i))) \rightarrow ele(table, [p]) \leq ele(table, [p+1])) \wedge \\ \quad (0 \leq (i - F_2(i))) \rightarrow ele(table, [i - F_2(i)]) \leq ele(table, [i - F_2(i) + 1])\end{array}}^{\uparrow}$$

Fertilization (modulo folding) with (9) leaves a proof residue of the form:

$$(0 \leq (i - F_2(i))) \rightarrow ele(table, [i - F_2(i)]) \leq ele(table, [i - F_2(i) + 1])$$

Decomposing the implication gives rise to a new hypothesis of the form:

$$0 \leq (i - F_2(i)) \tag{13}$$

This leaves the following residue to prove:

$$\underbrace{ele(table, [i - F_2(i)]) \leq ele(table, [i - F_2(i) + 1])}_{\textbf{blocked}} \tag{14}$$

The range generalization critic triggers at this point, and suggests that the outer-loop invariant should be strengthened to include the following schematic property:

$$(\forall p : int.((0 \leq p) \wedge (p \leq i - F_2(i))) \rightarrow (\forall q : int.((i - F_2(i) < q) \wedge (q \leq last)) \rightarrow ele(table, [p]) \leq ele(table, [q]))) \tag{15}$$

3.3.2 Second proof planning attempt

The second proof attempt requires us to conjoin (15) to original outer-loop invariant and generate a fresh set of VCs. The proof proceeds as before, however, where the proof previously became blocked (see (14)), we can now use (15) and (13) to infer:

$$(\forall q : int.((i - F_2(i) < q) \wedge (q \leq last)) \rightarrow ele(table, [i - F_2(i)]) \leq ele(table, [q]))$$

To complete the proof we must complete the instantiation of the invariant schema. F_1 is $\lambda x.x - F_2(x)$, so we require an instantiation for F_2. To achieve this we have to exploit constraints imposed by other parts of the proof. In particular we are looking for constraints that will prevent us from over generalizing the invariant. Within the context of inductive theorem proving base cases typically provide guidance in ensuring against over generalization. By analogy, checking the validity of an invariant on entry to a loop for the first time provides similar guidance. For example, based upon (15) we must show that the following holds on entry to the inner-loop where I has an initial value 1:

$$(\forall p : int.((0 \leq p) \wedge (p \leq 1 - F_2(1))) \rightarrow (\forall q : int.((1 - F_2(1) < q) \wedge (q \leq last)) \rightarrow ele(table, [p]) \leq ele(table, [q]))) \quad (16)$$

With no prior knowledge of the values assigned to the array *table*, the only way to make this schematic goal provable is to instantiate F_2 such that the condition associated with the implication is false. Instantiating F_2 to be 2 achieves the required effect and (16) becomes:

$$(\forall p : int.((0 \leq p) \wedge (p \leq 1 - 2)) \rightarrow (\forall q : int.((1 - 2) < q) \wedge (q \leq last)) \rightarrow ele(table, [p]) \leq ele(table, [q]))) \quad (17)$$

Note that the left-hand side of the outer implication evaluates to $(0 \leq p) \wedge (p \leq -1)$ which is false, consequently (17) is trivially true. While 2 is not the only value that works, it is the greatest value that makes the left-hand side of the outer implication false. More generally, the ability to exploit counter-examples in instantiating invariant schemas is an area that requires further investigation. The complete instantiation for F_1 is $\lambda x.(x - 2)$. Applying the instantiation for F_1 to (7), original schematic invariant, gives:

$$(\forall p : int.(0 \leq p \wedge p < i - 2) \rightarrow ele(table, [p]) \leq ele(table, [p + 1]))$$

While applying the instantiation for F_1 to (15) gives:

$$(\forall p : int.((0 \leq p) \wedge (p \leq i - 2)) \rightarrow (\forall q : int.((i - 2 < q) \wedge (q \leq last)) \rightarrow ele(table, [p]) \leq ele(table, [q]))) \quad (18)$$

Note that (18) asserts that the array *table* is *partitioned* such that the elements between 0 and $i - 2$ (inclusive) are less than the elements between $i - 1$ and *last* (inclusive). This auxiliary invariant requires proof.

3.3.3 Third proof planning attempt

The auxiliary invariant generated by the second proof attempt gives rise to a hypothesis of the form

$$(\forall p : int.((0 \leq p) \wedge (p \leq i - 2)) \rightarrow (\forall q : int.((i - 2 < q) \wedge (q \leq last)) \rightarrow ele(table, [p]) \leq ele(table, [q]))) \quad (19)$$

and a goal of the form:

$$(\forall p : int.((0 \leq p) \wedge (p \leq (i + 1) - 2)) \rightarrow \quad (20)$$
$$(\forall q : int.(((i + 1) - 2 < q) \wedge (q \leq last)) \rightarrow ele(table, [p]) \leq ele(table, [q]))) \quad (21)$$

Annotating (21) with respect to (19) gives rise to the following rippling problem:

$$(\forall p : int.((0 \leq p) \wedge (p \leq \boxed{(\underline{i} + 1)}^{\uparrow} - 2)) \rightarrow$$
$$(\forall q : int.((\boxed{(\underline{i} + 1)}^{\uparrow} - 2 < q) \wedge (q \leq last)) \rightarrow ele(table, [p]) \leq ele(table, [q])))$$

We can now ripple using (26) to give:

$$(\forall p : int.((0 \leq p) \wedge (p \leq \boxed{(\underline{i - 2}) + 1}^{\uparrow})) \rightarrow$$
$$(\forall q : int.((\boxed{(\underline{i - 2}) + 1}^{\uparrow} < q) \wedge (q \leq last)) \rightarrow ele(table, [p]) \leq ele(table, [q])))$$

Using (29) we ripple to give:

$$\boxed{(\forall p : int.((0 \leq p) \wedge (p \leq i-2)) \rightarrow (\forall q : int.((i-2 < q) \wedge (q \leq last)) \rightarrow ele(table,[p]) \leq ele(table,[q])))} \wedge$$
$$\boxed{(\forall q' : int.(((i-2)+1 < q') \wedge (q' \leq last)) \rightarrow ele(table,[i-2+1]) \leq ele(table,[q'])))}^{\uparrow}$$

Strong fertilization is now applicable (hypothesis (19)), leaving a proof residue of the form:

$$(\forall q' : int.(((i-2)+1 < q') \wedge (q' \leq last)) \rightarrow ele(table,[i-2+1]) \leq ele(table,[q']))$$

The proof residue simplifies to give:

$$\underbrace{(\forall q' : int.((i-1 < q') \wedge (q' \leq last)) \rightarrow ele(table,[i-1]) \leq ele(table,[q']))}_{\textbf{blocked}} \tag{22}$$

Within the given proof context i is equal to j. As a consequence, two alternative invariant schemas can be generated. The first asserts a notion of *minimum* element based upon i:

$$(\forall q' : int.((i < q') \wedge (q' \leq last)) \rightarrow ele(table,[i]) \leq ele(table,[q']))$$

The second makes a similar assertion for j:

$$(\forall q' : int.((j < q') \wedge (q' \leq last)) \rightarrow ele(table,[j]) \leq ele(table,[q'])) \tag{23}$$

Note that from our bottom-up analysis we know that i denotes the upper boundary of a partition (see (2)) while j denotes the lower boundary of a partition (see (3)). Consequently (23) is most closely aligned with the bubble sort algorithm. By this combination of proof-failure and bottom-up analysis, (23) is selected as a second auxiliary invariant. Note that (23) asserts that for the partition above j, the element indexed by j is the *minimum*. The proof patching process is completed by conjoining (23) onto the inner-loop invariant schema, from which a revised set of VCs are generated.

3.3.4 Fourth proof planning attempt

The fourth proof attempt requires us to conjoin (23) to the outer-loop invariant to give the inner-loop invariant. Proof of the associated VCs proceeds as before, however, where the proof previously became blocked (see (22)), we are now able to difference match the goal against (23). This give rise to a rippling problem of the form:

$$(\forall q' : int.((\boxed{(j-1)}^{\uparrow} < q') \wedge (q' \leq last)) \rightarrow ele(table,[\boxed{(j-1)}^{\uparrow}]) \leq ele(table,[q'])) \tag{24}$$

Using wave-rule (28), (24) reduces to give:

$$\boxed{(\forall q' : int.((j < q') \wedge (q' \leq last)) \rightarrow ele(table,[\boxed{(j-1)}^{\uparrow}]) \leq ele(table,[q'])) \wedge}$$
$$\boxed{(j \leq last) \rightarrow ele(table,[j-1]) \leq ele(table,[j]))}^{\uparrow}$$

Wave-rule (30) completes the rippling:

$$\boxed{(\forall q' : int.((j < q') \wedge (q' \leq last)) \rightarrow ele(table,[j]) \leq ele(table,[q'])) \wedge}$$
$$\boxed{(j \leq last) \rightarrow ele(table,[j-1]) \leq ele(table,[j]) \wedge}$$
$$\boxed{(j \leq last) \rightarrow ele(table,[j-1]) \leq ele(table,[j])}^{\uparrow}$$

Strong fertilization with (23) is now applicable. Fertilization leaves a residue that simplifies to give:

$$ele(table,[j-1]) \leq ele(table,[j]) \tag{25}$$

Note that this simplified residue follows from the proof context. To complete the reasoning (23) must also be shown to be invariant with respect to the inner-loop. We skip this proof here because it is a relatively vanilla rippling proof (see CS-note 62 for the details).

$$\boxed{(X+1)}^{\uparrow} - Y \;\Rightarrow\; \boxed{(X-Y)+1}^{\uparrow} \tag{26}$$

$$(\forall X : int.((L \leq X) \land (X < \boxed{U+1}^{\uparrow})) \rightarrow P(X)) \;\Rightarrow\; \boxed{(\forall X : int.((L \leq X) \land (X < U)) \rightarrow P(X)) \land ((L \leq U) \rightarrow P(U))}^{\uparrow} \tag{27}$$

$$(\forall X : int.((\boxed{L-1}^{\uparrow} < X) \land (X \leq U)) \rightarrow P(X)) \;\Rightarrow\; \boxed{(\forall X : int.((L < X) \land (X \leq U)) \rightarrow P(X)) \land ((L \leq U) \rightarrow P(L))}^{\uparrow} \tag{28}$$

$$\begin{aligned}
&(\forall X : int.((L \leq X) \land (X \leq \boxed{M+1}^{\uparrow})) \rightarrow \\
&\quad (\forall Y : int.((\boxed{M+1}^{\uparrow} < Y) \land (Y \leq U)) \rightarrow P(X,Y))) \;\Rightarrow\; \boxed{\begin{array}{l}(\forall X : int.((L \leq X) \land (X \leq M)) \rightarrow \\ \quad (\forall Y : int.((M < Y) \land (Y \leq U)) \rightarrow P(X,Y))) \land \\ \quad\quad (\forall Z : int.((M+1 < Z) \land (Z \leq U)) \rightarrow P(M+1,Z)\end{array}}^{\uparrow}
\end{aligned} \tag{29}$$

$$(\forall X : int.((L < X) \land (X \leq U)) \rightarrow P(\boxed{L-1}^{\uparrow}, X) \;\Rightarrow\; \boxed{\begin{array}{l}(\forall X : int.((L < X) \land (X \leq U)) \rightarrow P(L,X)) \land \\ ((L \leq U) \rightarrow P(L-1,L))\end{array}}^{\uparrow} \tag{30}$$

Figure 4: Wave-rules

3.3.5 Summary of proof planning

The style of proof planning presented above is highly iterative. The first proof planning iteration partially instantiates the initial invariant schema (ordered). In addition, failure analysis generates an auxiliary invariant schema (partitioned). During the second iteration the proof planning of initial invariant schema (ordered) is completed together with the instantiation of the initial (ordered) and auxiliary schemas (partitioned). The proof planning for instantiated auxiliary schema (partitioned) takes place in the third iteration. Failure analysis during the third iteration also gives rise to the generation of a second auxiliary invariant (minimum). A fifth and final iteration, not presented here, is then required to complete the planning of this auxiliary invariant (minimum).

3.4 Summary of invariant discovery results

In terms of the outer-loop, our bottom-up analysis guided the weakening of the given postcondition and constrained the proof-failure analysis that generated the *partitioned* invariant. In terms of the inner-loop, bottom-up analysis generated counter variable properties while proof-failure analysis generated the *minimum* invariant. We now introduce definitions for these auxiliary invariants:

$$\begin{aligned}
partitioned(A,L,M,U) \;&\leftrightarrow\; (\forall P : int.((L \leq P) \land (P \leq M)) \rightarrow \\
&\qquad (\forall Q : int.((M < Q) \land (Q \leq U)) \rightarrow ele(A,[P]) \leq ele(A,[Q]))) \\
minimum(A,L,U) \;&\leftrightarrow\; (\forall P : int.((L < P) \land (P \leq U)) \rightarrow ele(A,[L]) \leq ele(A,[P]))
\end{aligned}$$

Using these definitions the invariants generated for the bubble sort loops are presented below:

Outer-loop invariant: `Table` is ordered from `0` to `I-2`. Elements from `0` to `I-2` are less than or equal to all the elements from `I-1` upwards. All elements are preserved and `I` remains within its type.

```
--# assert Ordered(Table, 0, I-2) and
--#        Partitioned(Table, 0, I-2, Index_Type'Last) and
--#        Perm(Table, Table~) and
--#        1 <= I and I <= Index_Type'Last;
```

Inner-loop invariant: Extends the outer invariant to include that element `J` has the minimum value of the elements from `J` upwards. `I` remains within its type. `I` is less than or equal to `J`. The upper bound on `J` is `Index_Type'Last`.

```
--# assert Ordered(Table, 0, I-2) and
--#        Partitioned(Table, 0, I-2, Index_Type'Last) and
--#        Perm(Table, Table~) and
```

```
--#       Minimum(Table, J, Index_Type'Last) and
--#       1 <= I and I <= Index_Type'Last and
--#       I <= J and J <= Index_Type'Last;
```

It is worth emphasizing the interplay between the three analysis techniques that gave rise to these invariants. The predicate `ordered` was weakened by a combination of all there analysis techniques. While `partitioned` and `minimum` were a product of proof failure analysis. The properties relating to the loop counter variables were all generated via bottom-up analysis. In [10], six heuristic rules are presented in order to generate properties such as `minimum` in a purely bottom-up manner. The rules are quite complicated to understand, involving a generalization that is a "relatively difficult step, not yet completely investigated". Our approach that delays the need for generalization appears to be more straight forward and intuitive.

4 Comparison with related work

Research into heuristic rules for both bottom-up and top-down analysis have a long history [9, 10, 12, 13]. The work of Wegbreit led to the development of a prototype system called VISTA [5]. The VISTA system, and the later RUNCHECK [4] system used the strategy of conjoining failed goals onto a conjecture. The VISTA system was also able to extract information from failed proofs. Our proof critics extends theses ideas. In particular there are two key differences between our approach and previous approaches. Firstly, the use of schematic invariants which allows an incremental style of generation (cf generate-and-test). Secondly, the use of program knowledge in constraining proof patches during proof planning.

5 Current implementation and future requirements

In terms of bottom-up analysis, a prototype exists in the form of Tommy's AutoGap system [6]. Given AutoGap, the development of the top-down analysis component will be relatively straight forward. The proof critics introduced are new and have not been implemented. However, the application of rippling is not new and we have a proof planner that supports the incremental instantiation of schematic conjectures. The style of proof planning outlined above, however, requires the ability to alter focus between VCs during a proof attempt. For instance, within the second iteration of the proof planning (see §3.3.2), the focus was switched from the general loop-iteration to the first time through the loop. This was motivated in order to constrain the instantiation of the auxiliary invariant schema. Our planner will need to be strengthened in order to support such opportunistic planning. Moreover, this aspect of the proof planning revealed the potential for a new kind of proof critic. Consider again the way in which (16) was instantiated. An instantiation was selected that would make the goal trivially true, by making the condition associated with the goal false. This approach has strong similarities with the Lakatos notion of "monstor-barring" [11]. One could image a critic that constrained the instantiation of meta-variables using monstor-barring. Note that such a critic would not be triggered by the failure of a method. It would be triggered by the presence of meta-variables, possibly within critical VCs such as those associated with the first and last iterations of a loop.

6 Conclusion

We propose an approach to integrating invariant discovery techniques. The approach relies upon the incremental instantiation of invariant schemas and the use of program patterns during the patching of failed attempts. Our implementation work is ongoing, but we believe that this work will demonstrate the synergies that can be achieved through the integration of static analysis techniques.

References

[1] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, 2003.

[2] David Basin and Toby Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.

[3] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[4] S.M. German. Automating proof of the absence of common runtime errors. In *Proceedings of 5th ACM Conference on Principles of Programming Languages*. 1978.

[5] S.M. German and B. Wegbreit. A synthesizer of inductive assertions. *IEEE Trans. on Software Engineering*, SE-1(1):68–75, 1975.

[6] T. Ingulfsen. *Automatic Generation of Algorithmic Properties (AutoGAP)*. Undergraduate bsc computer science project dissertation, School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, 2003.

[7] 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.

[8] A. Ireland and J. Stark. Proof planning for strategy development. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):65–97, February 2001. An earlier version is available as Research Memo RM/00/3, Dept. of Computing and Electrical Engineering, Heriot-Watt University.

[9] S.M. Katz and Z. Manna. A heuristic approach to program verification. In *Proceedings of IJCAI-73*. International Joint Conference on Artificial Intelligence, 1973.

[10] S.M. Katz and Z. Manna. Logical analysis of programs. *Communications of the ACM*, 19(4):188–206, 1976.

[11] I. Lakatos. *Proofs and refutations: The logic of Mathematical discovery*. Cambridge University Press, 1976.

[12] B. Wegbreit. Heuristic methods for mechanically deriving inductive assertions. In *Proceedings of IJCAI-73*. International Joint Conference on Artificial Intelligence, 1973.

[13] B. Wegbreit. The synthesis of loop predicates. *Comm. ACM*, 17(2):102–122, 1974.