Andrew Ireland
September 10, 2002

On Combining Proof and Algorithmic Patterns

1 Introduction

Proof plans enable us to represent common proof patterns. As well as communicating high-level proof explanations, the computational nature of the proof plan representation means that it provides a mechanism for guiding proof search. For each proof plan there exists a rigorous meta-logic that provides the basis for specifying the proof tactics that form the foundation of the proof plan. Meta-logical predicates are used to classify the role that formulae, sub-formulae, lemmas, definitions, *etc* will potentially play within the application of a proof plan to a particular conjecture.

Our previous work on imperative program verification has focused upon the notion of proof plans as described above [2, 5, 3]. However, in the area of program verification we have felt that the power of proof planning would be significantly increased if we combined our "proof patterns" with "algorithmic patterns". Put another way, proof planning potentially provides a common framework within which meta-level properties of families of conjectures and algorithms can be represented. It has been long argued[1] that a benefit of formal program construction is that our intuitions as theorem provers can inform our programming and vice versa. We believe that the more modest goal of formal program verification can also benefit from such synergy, and moreover that proof planning provides a framework for achieving this goal.

In this note we begin to explore our hypothesis that combining proof and algorithmic patterns will significantly increase our program verification capabilities. We focus on loop-based programs since reasoning about loops, and the generation of inductive assertions (loop invariants) in particular, remains a major hurdle to automatic program verification. Moreover, in order to focus our search for common algorithmic patterns that are useful in guiding verification proofs, we restrict ourselves to loop-based programs that involve array manipulation. We already have some success in generating invariants for loops involving arrays [3] through the analysis of failed proof. However, if our hypothesis is correct then we should be able to improve upon our previous work.

2 Polish Flag Revisited

We illustrate by considering again the *Polish Flag Problem* (see CS-Note 10), where the goal is to:

> *... partition a mixture of coloured objects into distinct colours. In the case of the Polish Flag Problem, there are two distinct colours,* i.e. *red and white, corresponding to the colours of the Polish National Flag.*

For completeness, a SPARK [1] implementation for the Polish Flag Problem is given in appendix A. Presented in figure 1 are the precondition, postcondition and loop invariant for our SPARK implementation. The loop invariant assertion is a eureka step. We are interested in automating the generation of such steps. As mentioned above, our search for loop invariants has focused on the analysis of failures during the planning of verification proofs. Here we seek to complement our existing methods by exploiting patterns within the actual code that is being verified. Methods that apply directly to code are typically called *bottom-up*. Until now we have focused upon so called *top-down* methods that work at the level of specifications (and proofs). Our idea is to use the conventional proof planning approach (top-down) to drive the verification process, but to call upon bottom-up methods when we encounter failure. This is based upon our belief that our proof patterns provide more constraints than the algorithmic patterns.

The SPARK Examiner generates 4 (partial correctness) verification conditions (VCs) for the Polish Flag program. Here we focus upon the VC-3 associated with the `else`-branch of the loop[2] :

*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] Argued by the likes of Dijkstra, Gries, Backhouse et al within the context of program construction as the preserve of the human programmer.

[2] Note that *ele* and *upd* are used as abbreviations for the array *element* access and *update* operations respectively.

Precondition: All elements within the initial `Flag` are either `Red` or `White`.

```
--# pre (for all I in IndexRange => (Flag(I)=Red or Flag(I)=White));
```

Loop invariant: Loop execution maintains a lower partition of all `Red` elements and an upper partition of `White` elements.

```
--# assert Flag'First<=I and
--#        J<=(Flag'Last+1) and
--#        I<=J and
--#        (for all Q in Integer range Flag'First..(I-1) => (Flag(Q)=Red)) and
--#        (for all R in Integer range J..Flag'Last => (Flag(R)=White));
```

Postcondition: On termination all the `Red` elements within `Flag` will proceed the `White` elements, where the existential variable `P` is used to indicate the lower bound of the `White` elements.

```
--# post for some P in Integer range (Flag'First) .. (Flag'Last+1) =>
--#              ((for all Q in Integer range Flag'First..(P-1) => (Flag(Q)=Red)) and
--#               (for all R in Integer range P..Flag'Last => (Flag(R)=White)));
```

Figure 1: SPARK annotations associated with the Polish Flag Implementation

VC-3

H1: $(\forall q : integer.((q \geq indexrange\_first) \wedge (q \leq P-1)) \rightarrow ele(flag, [q]) = red)$

H2: $(\forall r : integer.((r \geq P) \wedge (r \leq indexrange\_last)) \rightarrow ele(flag, [r]) = white)$

H3: $\neg(i = j)$

H4: $\neg(ele(flag, [i]) = red)$

$\rightarrow$

C1: $(\forall q : integer.((q \geq indexrange\_first) \wedge (q \leq P-1)) \rightarrow$

$ele(\boxed{upd(upd(\underline{flag}, [i], ele(flag, [j-1])), [j-1], ele(flag, [i]))}^{\uparrow}, [q]) = red)$

C2: $(\forall r : integer.((r \geq P) \wedge (r \leq indexrange\_last)) \rightarrow$

$ele(\boxed{upd(upd(\underline{flag}, [i], ele(flag, [j-1])), [j-1], ele(flag, [i]))}^{\uparrow}, [r]) = white)$

Note that VC-3 is generated using the given postcondition (see figure 1) as the basis of the loop invariant, where the existential variable p has been replaced by the meta-variable P. Note also that conclusions C1 and C2 are annotated in preparation for an application of the ripple plan. By inspection, we can see that a wave rule that manipulates an array update is required, *i.e.* a wave rule that matches the following pattern:

$$ele(\boxed{upd(\underline{\ldots}, \ldots, \ldots)}^{\uparrow})$$

Wave rule (4) (see figure 2) matches this pattern, however, its application is blocked by the failure to prove the

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

$$(\forall X : integer.((X \geq L) \wedge (X \leq \boxed{\underline{U}+1}^{\uparrow})) \rightarrow P(X)) \;\Rightarrow\; \boxed{\underline{(\forall X : integer.((X \geq L) \wedge (X \leq U)) \rightarrow P(X))} \wedge P(U+1)}^{\uparrow} \tag{2}$$

$$(\forall X : integer.((X \geq \boxed{\underline{L}-1}^{\uparrow}) \wedge (X \leq U)) \rightarrow P(X)) \;\Rightarrow\; \boxed{\underline{(\forall X : integer.((X \geq L) \wedge (X \leq U)) \rightarrow P(X))} \wedge P(L-1)}^{\uparrow} \tag{3}$$

$$\neg(X = Z) \rightarrow ele(\boxed{upd(\underline{W}, X, Y)}^{\uparrow}, Z) \;\Rightarrow\; ele(W, Z) \tag{4}$$

Figure 2: Wave rules

associated side condition. In the case of $C1$ the side conditions are $\neg(i = q)$ and $\neg((j-1) = q)$ while in the case of $C2$ the side conditions are $\neg(i = r)$ and $\neg((j-1) = r)$. The problem is that the ranges of quantified formulae C1 and C2 are only partially defined. That is, both the upper bound of q and the lower bound of r involve the existential variable P.

In order to guide the instantiation of P we turn to algorithmic patterns. The initial pattern we are exploring for loops can be expressed by the following slogan:

$$\text{overall result} = \text{result computed so far} + \text{result of the remaining computation.}$$

This is an informal way of expressing what is known as a tail-invariant. In essence, the overall computation is partitioned into the "work done" and the "work still to do". In the case of many array based programs this partitioning involves identifying sequences of elements where manipulations are complete and sequences where manipulations are still required.

In the Polish Flag Problem we need to partition the array `Flag` so that wave rule (4) is applicable. Based upon where the existential variable P appears within the range of the quantified formulae will determine the position of the partition and the direction in which it grows. For instance, in the case of C1 the range of the quantified formulae takes the form:

$$((q \geq indexrange\_first) \wedge (q \leq P-1))$$

Here the existential variable P occurs in the upper bound of the range of q. This suggests the need for a partition that is bounded below by $indexrange\_first$ and which increases monotonically. We use the structure of the program to constrain the search for P. For a given loop, we can determine its associated array partitions by identifying the program variables that occur as array subscripts on the left-hand side of assignment statements. In addition, for each partition we can determine additional information, such as the bounds of the partition, how the partition grows and under what conditions the growth of the partition will terminate. Applying these ideas to the Polish Flag Problem gives rise to 2 explicit partitions:

1. $indexrange\_first$ to i
2. j to $indexrange\_last$

These partitions represent the "work done", while implicitly the partition between i and j denotes the "work still to do". Note that the idea is that these partitions are generated independently of the program specification. Therefore they only provide a guide as to how the array is being manipulated during the execution of the program. The constraints of the specification and verification proof determine the exact definition of the variable bounds given by i and j. Note that from the program code we can determine that the program variable i increases monotonically while the program variable j decreases monotonically. Taken together with the proof failure analysis, this code level analysis suggests instantiating P to be i:

$$((q \geq indexrange\_first) \wedge (q \leq i-1))$$

In the case of C2 the range of the quantified formulae takes the form:

$$((r \geq P) \wedge (r \leq indexrange\_last))$$

Here the existential variable P occurs in the lower bound of the range of r. This suggests the need for a partition that is bounded above by $indexrange\_last$ and which deceases monotonically. Here the analysis of the code suggests instantiating P to be j:

$$((r \geq j) \wedge (r \leq indexrange\_last))$$

Note that these alternative instantiations for P are consistent with the required postcondition because on termination $i = j$.

Typically when partitions are introduced the relative position of the partition bounds play an important role in establishing the loop invariant. Here this means generating that $indexrange\_first \leq i$, $j \leq indexrange\_last + 1$ and $i \leq j$ are also invariant. We envisage generating such facts using bottom-up techniques as part of the patching process.

Using the meta-logical predicates defined in figure 3 we provide a first attempt at defining critics that exploit both proof and algorithmic patterns in the search for a patch. The effect on the proposed patch on the VC-3 is shown below:

$occurs\_in(X, E)$	means that X occurs in the expression E.
$existential\_occurs\_in(V, E)$	means that the existential variable V occurs in the expression E.
$partition(A, L, U)$	means that the array A is partitioned with lower and upper bounds defined in terms of L and U respectively.
$mono\_dec(V)$	means that the program variable V monotonically decreases during the execution of the program.
$mono\_inc(V)$	means that the program variable V monotonically increases during the execution of the program.

Figure 3: Meta-logical predicates

Name	$range\_split\_lower$
Input	$H \Longrightarrow (\forall X : integer.((X \geq L) \wedge (X \leq U)) \rightarrow P)$
Preconditions	$method\_applicability($ $ripple, [$ 1. $wave\_term(\_, \_, \_)$, 2. $wave\_rule(C \rightarrow \_ \Rightarrow \_, \_)$, 3. $false]) \wedge$ $occurs\_in(X, C) \wedge$ $existential\_occurs\_in(EV, L) \wedge$ $partition(A, PV, U) \wedge$ $mono\_dec(PV)$
Patch	$replace(EV, PV, L, NewL)$ $induce(PV, PVHs)$, $append(H, PVHs, NewH)$
Output	$NewH \Longrightarrow (\forall X : integer.((X \geq NewL) \wedge (X \leq U)) \rightarrow P)$

Name	$range\_split\_upper$
Input	$H \Longrightarrow (\forall X : integer.((X \geq L) \wedge (X \leq U)) \rightarrow P)$
Preconditions	$method\_applicability($ $ripple, [$ 1. $wave\_term(\_, \_, \_)$, 2. $wave\_rule(C \rightarrow \_ \Rightarrow \_, \_)$, 3. $false]) \wedge$ $occurs\_in(X, C) \wedge$ $existential\_occurs\_in(EV, U) \wedge$ $partition(A, L, PV) \wedge$ $mono\_inc(PV)$
Patch	$replace(EV, PV, U, NewU)$, $induce(PV, PVHs)$, $append(H, PVHs, NewH)$
Output	$NewH \Longrightarrow (\forall X : integer.((X \geq L) \wedge (X \leq NewU)) \rightarrow P)$

An additional precondition could be added to the range splitting critics to check whether or not the proposed patch results in the application of the blocked conditional wave rule. However, such a check corresponds to performing a 1-step look-ahead into the proof. In order to eliminate redundancy we opt for simply applying the patch and proceeding with the next proof planning iteration.

Figure 4: Range splitting critics

VC-3 [**patched**]

H1: $indexrange\_first \leq i$

H2: $j \leq indexrange\_last + 1$

H3: $i \leq j$

H4: $(\forall q : integer.((q \geq indexrange\_first) \land (q \leq i - 1)) \rightarrow ele(flag, [q]) = red)$

H5: $(\forall r : integer.((r \geq j) \land (r \leq indexrange\_last)) \rightarrow ele(flag, [r]) = white)$

H6: $\neg(i = j)$

H7: $\neg(ele(flag, [i]) = red)$

$\rightarrow$

C1: $indexrange\_first \leq i$

C2: $j - 1 \leq indexrange\_last + 1$

C3: $i \leq j - 1$

C4: $(\forall q : integer.((q \geq indexrange\_first) \land (q \leq i - 1)) \rightarrow$
$ele(\boxed{upd(upd(\underline{flag}, [i], ele(flag, [j-1])), [j-1], ele(flag, [i]))}^{\uparrow}, [q]) = red)$

C5: $(\forall r : integer.((r \geq \boxed{\underline{j} - 1}^{\uparrow}) \land (r \leq indexrange\_last)) \rightarrow$
$ele(\boxed{upd(upd(\underline{flag}, [i], ele(flag, [j-1])), [j-1], ele(flag, [i]))}^{\uparrow}, [r]) = white)$

Note that H1, H2 and H3 are introduced as part of the invariant via $induce(PV, PVHs)$. C1 through to C3 are trivial and are discharged by the SPADE Simplifier. Wave rule (4) is directly applicable to C4. In the case of C5 an application of wave rule (3) splits the range of r enabling the application of wave rule (4). For the full details of how the proof planning of the above VC proceeds see §3.3 of CS-Note 10.

3 Conclusion

We tentatively outline our first steps towards combining proof and algorithmic patterns. The range splitting critics outlined are still very embryonic. They are motivated at the proof level by the failure to apply a conditional rule within the context of a particular class of conjecture that involves bounded quantification. At the algorithm level, the form of the array partitions within the program are used to suggest how the conjecture may be modified in order to enable the application of the conditional rule.

In order to further explore our hypothesis we need to test our ideas more examples. Having sketched solutions, however, to other text book solutions, *e.g.* Dutch Flag Problem (3-colour partition), Swapping equal length segments and various sorting algorithms, we believe that our ideas are worth developing.

One aspect that requires strengthening is the mechanism for inducing the relationship between program variables. Such invariant relationships typically play a crucial role when developing partial correctness proofs. Moreover, combining such bottom-up methods with our top-down (proof driven) methods should provide value-added when dealing with buggy code as noted by Katz and Manna [4].

Finally, we have been focusing on programs where the underlying algorithm involves incrementally constructing partition(s). Another important class of algorithms involve deconstructing partitions, *e.g.* searching algorithms. We aim to include this class of algorithms within our investigation.

References

[1] J. Barnes. *High Integrity Ada: The Spark Approach.* Addison-Wesley, 1997.

[2] A. Ireland and J. Stark. On the Automatic Discovery of Loop Invariants. In *Proceedings of the Fourth NASA Langley Formal Methods Workshop – NASA Conference Publication 3356*, 1997. Also available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/1.

[3] 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.

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

[5] J. Stark and A. Ireland. Invariant discovery via failed proof attempts. In P. Flener, editor, *Logic-Based Program Synthesis and Transformation*, number 1559 in LNCS, pages 271–288. Springer-Verlag, 1998. An earlier version is available from the Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/98/2.

Appendix A: SPARK code for the Polish Flag Problem

```
-- Coded by Bill J. Ellis
--
IndexUpper: constant := 4;
IndexLower: constant := 1;
subtype IndexRange is Integer range IndexLower .. IndexUpper;
subtype PointerRange is Integer range IndexRange'First .. IndexRange'Last+1;
type Colour is (Red, White);
type ArrayOfColours is array (IndexRange) of Colour;

...

-- Partition the colours of flag.
procedure Partition_Section(Flag: in out ArrayOfColours)
is
    subtype JustBiggerRange is Integer range Flag'First .. Flag'Last+1;
    I: JustBiggerRange;
    J: JustBiggerRange;
    T: Colour;
begin
    I:=Flag'First;
    J:=Flag'Last+1;

    loop
        --# assert Flag'First<=I    and
        --#        J<=(Flag'Last+1) and
        --#        I<=J and
        --#        (for all Q in Integer range Flag'First..(I-1) => (Flag(Q)=Red)) and
        --#        (for all R in Integer range J..Flag'Last => (Flag(R)=White));
        exit when I=J;
        if Flag(I)=Red then
            I:=I+1;
        else
            J:=J-1;
            T:=Flag(I);
            Flag(I):=Flag(J);
            Flag(J):=T;
        end if;
    end loop;
end Partition_Section;
```