Invariant Patterns for Program Reasoning

Andrew Ireland and Bill J. Ellis and Tommy Ingulfsen
School of Mathematical & Computer Sciences
Heriot-Watt University, Edinburgh, Scotland, UK
a.ireland@hw.ac.uk bill@macs.hw.ac.uk tommying@online.no

Abstract

We address the problem of integrating standard techniques for automatic invariant generation within the context of program reasoning. We propose the use of *invariant patterns* which enable us to associate common patterns of program code and specifications with invariant schemas. This allows crucial decisions relating to the development of invariants to be delayed until a proof is attempted. Moreover, it allows patterns within the program to be exploited in patching failed proof attempts.

1 Introduction

Within the context of program reasoning, we address the problem of automating loop invariant generation. There are two basic kinds of invariant generation techniques. Firstly, *bottom-up* analysis techniques generate inductive invariants by analysing program code. Secondly, *top-down* analysis techniques use specifications (assertions) as the basis for generating inductive invariants. In practice, a third kind of analysis, what we will call *proof-failure* analysis, also plays a crucial role within invariant generation.

We propose the use of *invariant patterns* as a means of achieving a effective integration of these three kinds of analyses. An invariant pattern represents an invariant schema together with a selection criteria. We build upon *proof planning* [3], a technique for automating theorem proving. In particular, we use *middle-out reasoning* [4] which supports an incremental style of invariant discovery and *proof critics* [9, 11] which supports proof-failure analysis. The context for our work is the application of proof planning to the verification of programs written in SPARK [1], a programming language designed for the development of critical software systems. SPARK is derived from Ada and includes an annotation language which supports flow analysis and formal proof. In §2, §3, and §4 we outline our general approach, while in §5, we present a detailed application.

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

Figure 1: Meta predicates

2 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 notion 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.

3 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 respectively. 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. Determining the identity of the replacement term is a key problem. 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:

$$(\forall q : int.\ ((l \leq q) \wedge (q \leq F_1(i))) \rightarrow P(q))$$

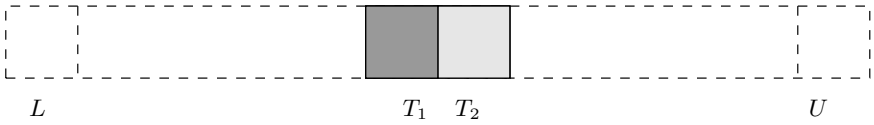
Note that F_1 denotes a second-order meta-variable. If the accessible range associated with i was i to u and i was monotonically decreasing, then the l would have been replaced by $F_1(i)$. Within the context of nested loops, invariant schemas are generated for an outer-loop before its inner-loop. An inner-loop will inherit the invariant schemas generated for its outer-loop.

4 Proof-Failure Analysis

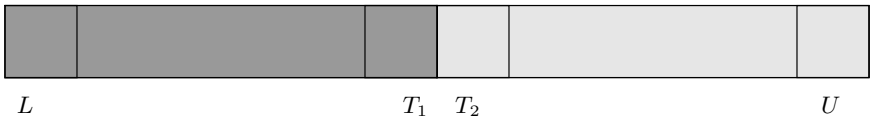
Given a failed proof attempt, a common theorem proving strategy is to conjoin the failed goal onto the original conjecture (invariant) and attempt the proof again. We extend this strategy by introducing two alternative generalization steps. We describe each generalization in terms of a proof critic.

4.1 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]:

[1]Note that $ele(X, [Y])$ denotes the value of element Y within array X.

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 meta predicates outlined in §2. The associated patch involves generalizing with respect to both T_1 and T_2, *i.e.*

$$(\forall X : int.((L \leq X) \wedge (X \leq T_1)) \rightarrow \\ (\forall Y : int.((T_1 < Y) \wedge (Y \leq U)) \rightarrow ele(A,[X])\ Rel\ ele(A,[Y])))$$

This generalized goal represents an auxiliary invariant which 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.

4.2 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 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, 5]. 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 involves annotating the differences between the post-fertilization goal and 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 ripple proof to proceed. The patch uses the available wave-rules (background theory) to guide the discovery of the missing hypothesis (invariant), *i.e.* we look for wave-rule matches that fail because of missing wave-front annotations within the goal. The preconditions to the critic are as follows:

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.*

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

3. Application of the wave-rule would progress the proof planning.

Note that shading is used to represent wave-front annotations. Here annotations are missing from the goal, preventing the application of the wave-rule. 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 is not essential, but further constrains the search for an auxiliary invariant by looking ahead into the proof planning. 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.

5 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, which is verified, is given in figure 2. Note that the code is annotated with preconditions and postconditions, but no loop invariants are specified. Moreover, given that the code involves nested loops then two loop invariants will be required in order to prove partial correctness.

5.1 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 [11]) where *rippling in reverse* is used to determine alternative induction schemas.

5.2 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:

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

Unfolding using (5), the `Ordered` predicate becomes:

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

This is a candidate for the range restriction invariant pattern. From our bottom-up analysis of the bubble sort code (see §5.1), 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 (3) and (4). 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:

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

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 $G_1(j)$ within (7) to give:

$$(\forall p : int.((G_1(j) \leq p) \wedge (p < F_1(i))) \rightarrow ele(table, [p]) \leq ele(table, [p + 1])) \quad (8)$$

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 the proof planning for (7) is not successful.

```
package BubbleSort is
subtype Index_Type is Integer range 0..9;
type Array_Type is array (Index_Type) of Integer;
...
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 2: A SPARK implementation of Bubble Sort

$$(\boxed{(X+1)}^{\uparrow} - Y \;\Rightarrow\; \boxed{(X-Y)+1}^{\uparrow} \tag{9}$$

$$(\forall X : int.((L \le X) \wedge (X < \boxed{U+1}^{\uparrow})) \rightarrow P(X)) \;\Rightarrow\; \boxed{(\forall X : int.((L \le X) \wedge (X < U)) \rightarrow P(X)) \wedge ((L \le U) \rightarrow P(U))}^{\uparrow} \tag{10}$$

$$(\forall X : int.((\boxed{L-1}^{\uparrow} < X) \wedge (X \le U)) \rightarrow P(X)) \;\Rightarrow\; \boxed{(\forall X : int.((L < X) \wedge (X \le U)) \rightarrow P(X)) \wedge ((L \le U) \rightarrow P(L))}^{\uparrow} \tag{11}$$

$$\begin{aligned}(\forall X : int.((L \le X) \wedge (X \le \boxed{M+1}^{\uparrow})) \rightarrow \\ (\forall Y : int.((\boxed{M+1}^{\uparrow} < Y) \wedge (Y \le U)) \rightarrow P(X,Y))) \;\Rightarrow\; \boxed{(\forall X : int.((L \le X) \wedge (X \le M)) \rightarrow} \\ \boxed{(\forall Y : int.((M < Y) \wedge (Y \le U)) \rightarrow P(X,Y)))} \wedge \\ \boxed{(\forall Z : int.((M+1 < Z) \wedge (Z \le U)) \rightarrow P(M+1, Z))}^{\uparrow}\end{aligned} \tag{12}$$

Figure 3: Wave-rules

5.3 Proof Planning and Proof-Failure Analysis

The proof planning requires a number of attempts, where each attempt refines the candidate invariants. Success corresponds to the generation of a concrete set of invariants (proof annotations) and proof tactics for the associated verification conditions (VCs).

5.3.1 First Proof Planning Attempt:

The analysis outlined above gives rise to a set of schematic VCs. We focus on the `Ordered` predicate. Following a rippling style of proof, we have schematic hypothesis (7) and an annotated goal of the form:

$$(\forall p : int.((0 \leq p) \wedge (p < F_1(\boxed{(\boxed{i}+1)}^{\uparrow}))) \rightarrow ele(table, [p]) \leq ele(table, [p+1])) \quad (13)$$

Using wave-rules (9) and (10) rippling rewrites (13) to give:

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

Note that as a side-effect, F_1 is partially instantiated, *i.e.* F_1 becomes $\lambda x.x - F_2(x)$. Fertilization with hypothesis (7) would leave a residue of the form:

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

Decomposing the implication gives rise to a new hypothesis

$$0 \leq (i - F_2(i+1)) \quad (14)$$

and a goal of the form:

$$\underbrace{ele(table, [i - F_2(i+1)]) \leq ele(table, [i - F_2(i+1)+1])}_{\textbf{blocked}} \quad (15)$$

Note that this goal is *blocked* as no proof methods are applicable, *i.e.* rippling, simplification or fertilization. Proof-failure analysis applies the range generalization critic. The associated proof patch generates the following auxiliary invariant schema:

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

The proof patching process is completed by conjoining (16) onto the refined outer-loop invariant schema, from which a revised set of VCs are generated.

5.3.2 Second Proof Planning Attempt:

With the refined outer-loop invariant, the proof context on the second proof attempt contains (16). Proof proceeds initially, as described for the first attempt. However, where the proof previously was blocked, hypothesis (16) can be specialized (using (14)) in order to prove (15). To complete the proof of goal (13) we need to complete the instantiation of the schema. To achieve this we have to exploit constraints imposed by other parts of the proof. Testing a candidate invariant on loop entry will typically detect over generalizations[3]. For instance, on entry to the outer-loop `I` has the value 1 and schema (16) becomes:

$$(\forall p : int.((0 \leq p) \wedge (p \leq 1 - F_2(2))) \rightarrow \\ (\forall q : int.((1 - F_2(2) < q) \wedge (q \leq last)) \rightarrow ele(table, [p]) \leq ele(table, [q])))$$

This schematic goal is trivial to prove if F_2 is instantiated to be $\lambda x.2$, We return to the mechanization of such a step in §7. Note that the instantiated invariant schema asserts that the array *table* is *partitioned* such that elements below $i-2$ (inclusive) are less than or equal to the elements above $i-2$.

5.3.3 Third Proof Planning Attempt:

We now consider the proof of the *partitioned* invariant discovered above. In particular, we focus on the VC corresponding to the path from the inner-loop invariant to the outer-loop invariant, *i.e.* where i is equal to j and we have a hypothesis of the form

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

and an annotated goal of the form:

$$(\forall p : int.((0 \leq p) \wedge (p \leq \boxed{(i+1)}^{\uparrow} - 2)) \rightarrow \\ (\forall q : int.((\boxed{(i+1)}^{\uparrow} - 2 < q) \wedge (q \leq last)) \rightarrow ele(table, [p]) \leq ele(table, [q]))) \quad (18)$$

Using wave-rules (9) and (12) rippling rewrites (18) to give:

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

Fertilization with hypothesis (17) leaves a residue which simplifies to give:

$$\underbrace{(\forall q' : int.((i - 1 < q') \wedge (q' \leq last)) \rightarrow ele(table, [i - 1]) \leq ele(table, [q']))}_{\textbf{blocked}} \quad (19)$$

[3]This is analogous to testing base cases within the context of proof by mathematical induction in order to guard against an over generalization.

No proof methods are applicable so the goal is *blocked*. Motivated by a partial match with wave-rule (11), proof-failure analysis applies the difference generalization critic. Given that i and j are equal, 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:

$$(\forall q' : int.((j < q') \wedge (q' \leq last)) \rightarrow ele(table, [j]) \leq ele(table, [q'])) \quad (20)$$

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, (20) is most closely aligned with the bubble sort algorithm. By this combination of proof-failure and bottom-up analysis, (20) is selected as a second auxiliary invariant. Note that (20) asserts that for the partition above j, the element indexed by j is the *minimum*. The proof patching process is completed by conjoining (20) onto the inner-loop invariant schema, from which a revised set of VCs are generated.

5.3.4 Fourth and Fifth Proof Planning Attempts:

With the refined inner-loop invariant, the proof context on the fourth proof attempt contains (20). Proof proceeds initially as described for the third attempt. However, where the proof previously was blocked (see (19)), hypothesis (20) provides the basis for a simple rippling proof. To complete the reasoning, (20) must be shown to be invariant with respect to the inner-loop, again a relatively simple application of rippling is required.

5.4 Summary of Invariant Discovery Results

Bottom-up analysis generated counter variable properties which contributed to both the outer-loop and inner-loop invariants. Top-down analysis, constrained by bottom-up analysis, generated an invariant schema. Through proof-failure analysis, this schema was refined to give the *partitioned* invariant, *i.e.* (17). Proof-failure analysis also generated the *minimum* invariant, *i.e.* (20).

6 Comparison with Related Work

Research into heuristic rules for both bottom-up and top-down analysis have a long history [15, 16, 18, 19]. The work of Wegbreit led to the development of a prototype system called VISTA [7]. The VISTA system, and the later RUNCHECK [6] 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 extend 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.

7 Current Implementation and Future Work

Our bottom-up analysis techniques have been implemented and tested within a prototype called AutoGap [8]. The application of rippling presented here is not new and we have a proof planner that supports the incremental instantiation of schematic conjectures and proof patching [10, 11, 12, 13, 17, 14]. The implementation of the proposed generalization proof critics is under-way. In terms of future work, the style of proof planning outlined above requires the ability to opportunistically switch between VCs. Moreover, the ability to exploit counter-examples in instantiating invariant schemas (see second proof planning attempt §5.3) is an area that requires further investigation.

8 Conclusion

We propose an integration of invariant discovery techniques. The approach relies upon the incremental instantiation of invariant schemas and the use of program patterns during the patching of failed proof 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.

Acknowledgements: Thanks to Peter Amey, Alan Bundy, Rod Chapman, Jonathan Hammond and Ian O'Neill for their feedback and support. The research is funded by EPSRC grant GR/R24081 and is a collaboration with Praxis Critical Systems Ltd.

References

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

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

[3] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[4] A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L.H. Clarke, editor, *Proceedings of UK IT 90*, pages 221–6. IEE, 1990. Also available from Edinburgh as DAI Research Paper 448.

[5] 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.

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

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

[8] 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.

[9] A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proofs. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag, 1992. Also available from Edinburgh as DAI Research Paper 592.

[10] A. Ireland and A. Bundy. Extensions to a Generalization Critic for Inductive Proof. In M. A. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction*, pages 47–61. Springer-Verlag, 1996. Springer Lecture Notes in Artificial Intelligence No. 1104. Also available from Edinburgh as DAI Research Paper 786.

[11] 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.

[12] A. Ireland and A. Bundy. Automatic Verification of Functions with Accumulating Parameters. *Journal of Functional Programming: Special Issue on Theorem Proving & Functional Programming*, 9(2):225–245, March 1999. A longer version is available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/11.

[13] 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.

[14] 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.

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

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

[17] J. Stark and A. Ireland. Invariant discovery via failed proof attempts. In P. Flener, editor, *Logic-Based Program Synthesis and Transformation*, LNCS 1559, 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.

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

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