Invariant Discovery via Failed Proof Attempts

Jamie Stark* and Andrew Ireland**

Dependable Systems Group,
Department of Computing & Electrical Engineering,
Heriot-Watt University, Riccarton,
Edinburgh EH14 4AS Scotland, UK
{jamie,air}@cee.hw.ac.uk
http://www.cee.ac.uk/~dsg

Abstract. We present a framework for automating the discovery of loop invariants based upon failed proof attempts. The discovery of suitable loop invariants is a bottle neck for automatic verification of imperative programs. Using the proof planning framework we reconstruct standard heuristics for developing invariants. We relate these heuristics to the analysis of failed proof attempts allowing us to discover an invariant through a process of refinement.

1 Introduction

Loop invariants are a well understood technique for specifying the behaviour of imperative loop programs. The discovery of suitable invariants, however, is a major bottle neck for automatic verification of imperative programs. Early research into heuristic based methods [18, 23] focused on the heuristic and the theorem proving components separately. The proof planning framework, in which we reconstruct the standard heuristics, couples the heuristic and theorem proving components tightly, allowing us to focus on both of these components together. This enables us to exploit the relationship between failed proof attempts and invariant discovery. The framework described here builds upon previous work [16] and shows how successive approximations are used to find a suitable invariant by a system automatically. This work was motivated by previous work on discovering generalisations for inductive proofs. We exploit the relationship between while loops and tail recursive functions, but rather than transform our loop programs into tail recursive functions and discover a generalisation we choose instead to transfer heuristics across domains. This enables us to communicate proof attempts, and specifications in the same language. Ultimately this will have significant benefits when integrating our approach within an interactive environment.

In the rest of the paper we introduce some background material, focuses on proof planning and the rippling heuristic. Next we give a proof plan for the

* This work is supported by an EPSRC student ship a ward 96307451
** This work is supported by EPSRC grant G L/L/11724

verification of loops. We then discuss how we can analysis the partial success of this proof plan with respect to two standard heuristics in the literature, namely replacing constants by terms and strengthening invariants. We show how this is done for an example and report on the implementation and results. We discuss related and future work before ending with a brief discussion.

2 Background

Hoare [12] first introduced his axiomatic proof rules for imperative programs involving loops in 1969. The aim was to assist the understanding of programs using rigourous logical techniques. The proof rules allow the transformation of a program with suitable assertions, into a set of purely logical implications called verification conditions. These assertions are relationships between the program variables and program constants[1]. Program constants are important since they are used to capture initial inputs to a program and are used in the post condition to relate a desired property in terms of these initial values.

The verification of simple imperative programs involving loops falls short of full automation since Hoare's rule for while loops involves synthesising an assertion known as the invariant. The while rule takes the form:

$$\frac{P \to I, \quad \{I \wedge B\}S\{I\}, \quad (I \wedge \neg B) \to Q}{\{P\}\textbf{while} \ \ B \ \textbf{begin} \ S \ \textbf{end} \ \{Q\}}$$

The second premise is used to generate a verification condition using Hoare's other proof rules. It is this verification condition which establishes I to be an invariant of the loop body S and is the focus of our work.

The literature [1, 8, 10, 17] contains many heuristics for constructing invariants in a systematic way, which involves making successive approximations at the invariant. These heuristics involve replacing a constant by a variable or term, strengthening an invariant by adding a conjunct and tail invariants. Previously [16] we showed how proof planing was able to successfully discover tail invariants. Here this work is extended by adding heuristics to replace program constants and strengthen invariants. A general strategy is presented that enables the successive approximation of invariants using these heuristics.

2.1 Proof Planning

Proof planning [3] consists of tactics, methods and critics. Tactics are rules of inference, either primitive or derived. Methods partial specify a tactic and allow the use of heuristics. A method has preconditions and effects. Proof planning involves the search for a plan at the meta-level by using the methods and critics. A proof at the object level is extracted from a plan and run to ensure soundness. Critics [13] are used to handle the failure of a proof attempt. They patch the failed proof attempt and provide a powerful mechanism for dealing with failure.

[1] Program constants are denoted as calligraphic letters throughout (*e.g.* $\mathcal{X}$).

The CLAM proof planner [6] has been used, amongst other applications, for inductive proof [5]. The success of the proof plan for induction is due to the ripple heuristic which is explained more fully below.

Middle-out reasoning is a proof planning strategy where meta terms are used to delay choice within the proof. The motivation is that the middle of a proof is typically more constrained than the start. The aim of middle-out reasoning is to use the proof to add structure where we do not know what it is. The implementation makes use if the higher-order features of λ-prolog [21]. Rippling is used to constrain the matching of wave-rules, by taking the annotations into account. Middle-out techniques have been used for generalisation [11, 14], logic program synthesis [19], lemma discovery and induction revision [15].

2.2 The Ripple Heuristic

Rewriting often involves the manipulation of a goal formula so that a target formula can be applied. We are interested in proving theorems of the form:

$$Hypothesis \rightarrow Conclusion$$

Where *Hypothesis* is the target formula and *Conclusion* is the goal formula we want to manipulate. Rippling is a difference reduction technique which exploits syntactic similarities between the goal and target in guiding the selective application of rewrite rules. In particular it identifies term structure within the goal which prevents a match with the target. For example consider the following implication:

$$(x + (y + z) = (x + y) + z) \rightarrow (s(x) + (y + z) = (s(x) + y) + z) \qquad (1)$$

The conclusion differs syntactically from the hypothesis in two places *i.e.* in the application of the successor function s around the two occurrences of x. We identify this difference with shading. Such term structures are called *wave-fronts*. Conversely any term structure within the goal which corresponds to the target is called *skeleton*. The parts of the skeleton inside a wave-front are called *wave-holes*. Wave-fronts, wave-holes and skeletons are meta-level notions. For our example above the conclusion is given the following meta-level annotation:

$$(\ldots) \rightarrow \left(\boxed{s(\,x\,)} + (y + z) = (\boxed{s(\,x\,)} + y) + z \right)$$

The wave-fronts are shaded. The skeleton of the conclusion is all the unshaded terms *i.e.* this corresponds to the hypothesis. An arrow is often used to indicate the direction in which a wave-front can be moved with respect to the skeleton term structure. Directed wave-fronts enable the termination of rippling to be guaranteed [2]. The direction of wave-fronts tells us how the difference is moving with respect to the skeleton term. For ease of presentation we have dropped the directions attached to the wave fronts since in this paper we will only be concerned with outward ($\uparrow$) directed wave-fronts. The movement of wave-fronts is performed by *wave-rules*, a syntactic class of rewrite rule which preserves skeleton while making progress to eliminating wave-fronts. An algorithm exists which automatically generates wave-rules from all definitions and properties loaded into the system. For any equation there are usually a number of corresponding wave-rules.

Consider the following recursive definition of plus and a commutative property[2] :

$$A + 0 = 0 \tag{2}$$

$$s(A) + B = s(A + B) \tag{3}$$

$$A + B = B + A \tag{4}$$

Example wave-rules derived from (3) and (4) respectively are[3]:

$$s(A) + B \Rightarrow s(A + B) \tag{5}$$

$$A + B \Rightarrow B + A \tag{6}$$

Notice that the skeleton on both sides of the wave-rules are the same. For (5) the skeleton is $A + B$ and for (6) it is A. To use a wave-rule all the object level and meta-level structure on the left hand side must match with a sub-term of the term we wish to rewrite. This sub-term is then replaced with right hand side of the wave-rule, taking into account any unification. We are able to reduce the differences between the hypothesis and conclusion in (1), by successive applications of wave-rule (5):

$$(x + (y + z) = (x + y) + z) \rightarrow \Big(s(x) + (y + z) = (s(x) + y) + z \Big)$$

$$x + (y + z) = (x + y) + z \rightarrow \Big(s(x + (y + z)) = (s(x) + y) + z \Big)$$

$$x + (y + z) = (x + y) + z \rightarrow \Big(s(x + (y + z)) = (s(x + y)) + z \Big)$$

$$x + (y + z) = (x + y) + z \rightarrow \Big(s(x + (y + z)) = s((x + y) + z) \Big)$$

Notice how the wave-fronts move to the top of the term structure. We were not able to use wave-rule (6); while the underlying object level structure matched the meta-level structure did not. This meta-level structure helps to guide the search for a proof. Now we have a copy of part of our hypothesis in the wave-hole. Since our hypothesis is an equality we can use it as rewrite rule to rewrite one of the wave-holes, and then we may remove our meta-level annotation. This we call (weak) fertilization and gives us the trivial goal:

$$(x + (y + z) = (x + y) + z) \rightarrow (s((x + y) + z) = s((x + y) + z))$$

A stronger versions of fertilization is applicable when a complete copy of the hypothesis appears within a wave-hole.

[2] We use the Prolog notation of denoting first order variables with capital letters throughout.

[3] Where $\Rightarrow$ denotes rewriting.

2.3 Summary

Rippling is a heuristic that enables us to apply rewriting in a directed way. This is achieved by identifying differences between two formulas and using rewrite rules which are guaranteed to keep those parts of the formula we want to stay the same, the same whilst moving the parts that are different in a particular direction. Proofs guided by rippling can be classified in terms of the direction in which wave-fronts are moved with respect to the skeleton term structure. For a full account of rippling see [2, 5]. The annotation described here can also been seen as embedding the target formula within the goal formula. Rippling is easily formulated in this framework. See [22] for more details.

Identifying differences and constructing wave-rules is automatic. The advantage of using the ripple heuristic is that, by adding this extra meta-level structure we can guide the search of the rewriting process. This is especially the case in the presence of meta-variables where we use our expectation of how the proof will be constructed to severely reduce the space searched.

3 A Proof Plan for Verification

```
exp1:   {x = 𝒳 ∧ y = 𝒴}
        r := 1;
        while (y > 0) do
        begin
                r := r * x;
                y := y - 1
        end
        {r = exp(𝒳, 𝒴)}
```

Fig. 1. Algorithm for computing exponentiation

Consider the program in figure 1. To verify this program we need a loop invariant. A suitable invariant is not immediately obvious and finding one is often referred to as a *eureka* step. One such invariant for this loop is:

$$r = exp(\mathcal{X}, \mathcal{Y} - y) \wedge x = \mathcal{X}$$

In order to verify that this property is indeed invariant involves proving the following verification condition generated by the loop body:

$$\begin{aligned}(r = exp(\mathcal{X}, \mathcal{Y} - y) \wedge x = \mathcal{X} \wedge y \neq 0)&\\ \rightarrow (r * x = exp(\mathcal{X}, \mathcal{Y} - (y-1)) \wedge x = \mathcal{X})&\qquad (7)\end{aligned}$$

The differences between the hypothesis and conclusion arise from the assignment statements in the loop body. To show that this is invariant we must rewrite the conclusion and move these differences up through the term structure until we are able to appeal to our hypothesis. Rippling provides us with the power to do this if we are able to annotate the conclusion with respect to the hypothesis. This suggests the following proof plan:

Either *a goal can be simplified* **or** *we annotate the conclusion with respect to the hypothesis* **then** *apply the wave method* **until** *we can fertilize with the hypothesis.*

The power of the proof plan is not that it can do the verification, but that it can be used to encode the heuristics we need to use to discover a suitable invariant from an initial guess. Figure 2 shows `Inv_Strat`, the proof plan we shall use.

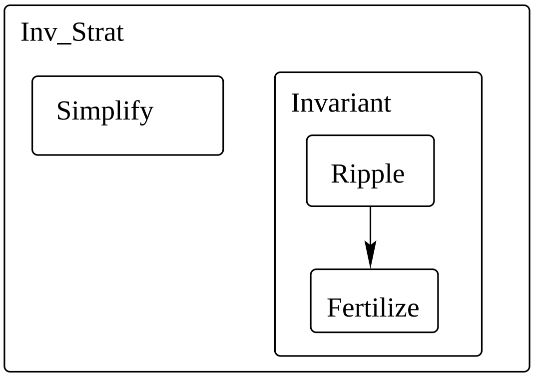

Fig. 2. Proof Plan for Verification

The methods are shown in more detail in figures 3, 4, 5 and 6. Rippling consists of repeated application of the the wave method.

A Verification Proof

Consider the following wave-rules:

$$A - \boxed{(B - C)} \Rightarrow \boxed{(A - B) + C} \tag{8}$$

$$exp(A, \boxed{B + 1}) \Rightarrow \boxed{exp(A, B) * A} \tag{9}$$

$$\boxed{A * C} = \boxed{B * D} \Rightarrow \boxed{A = B \wedge C = D} \tag{10}$$

and the verification condition (7) above: In the proof of (7) we are given

$$r = exp(\mathcal{X}, \mathcal{Y} - y) \wedge x = \mathcal{X} \wedge y \neq 0$$

from which we must show the conclusion

$$r * x = exp(\mathcal{X}, \mathcal{Y} - (y - 1)) \wedge x = \mathcal{X}$$

Input sequent:

$$H \vdash G_1 \wedge \ldots \wedge G_i \wedge \ldots \wedge G_n$$

Method preconditions:

1. There exists a G_i such that G_i is unannotated.
2. And G_i follows from H trivially *i.e.* it is a tautology *e.g.* $r = r$ or it follows from some simply data type property *e.g.* $0 \neq s(0)$.

Output sequent:

$$H \vdash G_1 \wedge \ldots \wedge G_n$$

Fig. 3. The Simplify Method

Input sequent:

$$H \vdash G_1[f_1(\ldots)] \wedge \ldots \wedge G_n[f_n(\ldots)]$$

Method preconditions:

1. $G_1[f_1(\ldots)] \wedge \ldots \wedge G_n[f_n(\ldots)]$ is unannotated.
2. for all $G_i[f_i(\ldots)]$ there exists a member H_j of H such that $G_i[f_i(\ldots)]$ can be annotated with respect to H_j, *e.g.*

$$G_1[\ \boxed{f_1(\ldots)}\ ] \wedge \ldots \wedge G_n[\ \boxed{f_n(\ldots)}\ ]$$

Output sequent:

$$H \vdash G_1[\ \boxed{f_1(\ldots)}\ ] \wedge \ldots \wedge G_n[\ \boxed{f_n(\ldots)}\ ]$$

Sub Methods:
Repeat wave then fertilize.

Fig. 4. The Invariant Method

Input sequent:

$$H \vdash G[f_1(\ \boxed{c_1(\ldots)}\ )]$$

Method preconditions:

1. there exists a sub-term T of G which contains wave-front(s), *e.g.*

$$f_1(\ \boxed{c_1(\ldots)}\ )$$

2. there exists a wave-rule which matches T, *e.g.*

$$C \rightarrow f_1(\ \boxed{c_1(X)}\ ) \Rightarrow \boxed{c_2(f_1(X))}$$

3. any condition C follows from the context, *e.g.*

$$H \vdash C$$

4. resulting inward directed wave-fronts are potentially removable.

Output sequent:

$$H \vdash G[\ \boxed{c_2(f_1(\ldots))}\ ]$$

Fig. 5. The Wave Method

Input sequent:

$$H[exp_1] \vdash \boxed{G_1 \wedge \ldots \wedge \boxed{exp_1} \wedge \ldots \wedge G_n}$$

Method preconditions:

1. There exists a hypothesis that matches with a conjunct in a wave hole in the conclusion, *e.g.*

$$exp_1$$

Output sequent:

$$H[exp_1] \vdash G_1 \wedge \ldots \wedge G_n$$

Sub Methods:
Use `Inv_Strat` on the remaining subgoals.

Fig. 6. The Fertilize Method

by the application of the `Inv_Strat` proof plan. The proof found by the proof planner is:

$$r * x = exp(\mathcal{X}, \mathcal{Y} - (y-1)) \wedge x = \mathcal{X} \qquad \text{Conclusion}$$

$$r * x = exp(\mathcal{X}, \mathcal{Y} - (y-1)) \qquad \text{By Simplify}$$

$$\boxed{\boxed{r} * x} = exp(\mathcal{X}, \mathcal{Y} - \boxed{(\boxed{y} - 1)}) \qquad \text{By Invariant}$$

$$\boxed{\boxed{r} * x} = exp(\mathcal{X}, \boxed{(\boxed{\mathcal{Y} - y}) + 1}) \qquad \text{By Wave with (8)}$$

$$\boxed{\boxed{r} * x} = \boxed{\boxed{exp(\mathcal{X}, \mathcal{Y} - y)} * \mathcal{X}} \qquad \text{By Wave with (9)}$$

$$\boxed{\boxed{r = exp(\mathcal{X}, \mathcal{Y} - y)} \wedge x = \mathcal{X}} \qquad \text{By Wave with (10)}$$

$$x = \mathcal{X} \qquad \text{By Fertilize}$$

$$true \qquad \text{By Simplify}$$

4 Analysis of Failure

In this section first we review two standard heuristics for discovering loop invariants from the literature [1, 8, 10, 17], namely replacing constants by terms and strengthening invariants. We formalise these heuristics within the proof planning framework as critics. Finally we discuss how these critics relate to earlier critics for inductive proof [13].

The full benefits of the rippling heuristic become apparent when the proof attempt fails. By relating this failure to a standard heuristic for developing loop invariants, we are able to suggest a patch which will give a new candidate invariant.

4.1 Replacement of Constants

A standard heuristic for developing loop invariants is replacing constants by terms containing program variables. Constants are replaced with term structure containing a program variable in a relation to increase the state space that the relation describes. In this state space this relation may be more easily established.

Rippling can fail because it fails to match the left hand side of a wave-rule with part of the goal. Wave-rule matching is defined as:

Wave-rule Match: Given a goal T and the left hand side of a wave-rule L, then T and L match iff:
- The object level term structures of L and T match.
- For all wave-fronts in L there exists a matching wave-front in T.

Constants can block a ripple because they can prevent a match between the left hand side of a wave-rule and a sub-term of the goal. This happens when at least one wave-front within a wave-rule is aligned with a term containing a constant in the goal. However we can define a potential match which suggests we have the potential to match if we could introduce some difference by replacing constants:

Potential Match: Given a goal T and the left hand side of a wave-rule L, then T and L potentially match iff:
- For all wave-fronts in L there exists either a matching wave-front in T or a term containing a constant.
- The object level term structures of L and T match everywhere expect at the positions where a wave-front in L matches a term containing a constant.

For example, attempting to match $\boxed{r * x} = exp(\mathcal{X}, \mathcal{Y})$ and the left hand side of the wave-rule (10):

$$\boxed{A * C} = \boxed{B * D}$$

would be a potential match because there is a term containing a constant on the right hand side of the equality in the sub-term, *e.g.* $exp(\mathcal{X}, \mathcal{Y})$, where the wave-rule has a wave, *e.g.* $\boxed{B * D}$ and all other structure matches.

Having a potential match tells us that we need a wave-front where we have a term containing a constant. The replacement of constants heuristic tells us to replace a constant with a term containing a variable which has been assigned within the loop body. To coerce this wave-rule match we need to do two things. Firstly we need to decide which constant to replace, if there is more than one.

Secondly we need to discover the term. The first step is achieved by taking the mismatched wave in the wave-rule and pushing the differences back towards the constants, by applying the wave-rules backwards, with a schematic wave. For the potential match above we want to coerce $exp(\mathcal{X}, \mathcal{Y})$ to be the wave $B * D$. Since we are only replacing constants we know the skeleton must be $exp(T_1, T_2)$, where T_1 and T_2 are either new terms or the constants in those positions. Starting then with the wave $exp(T_1, T_2) * D$ we use wave-rule (9) backwards to get the term:

$$exp(D, B + 1)$$

We now have identified that the wave needed can be generated by replacing the second argument of exp with a term containing a wave. We could continue backwards to find the term we are looking for. However the search space is less controlled in this direction. We now use middle out reasoning by replacing the constant identified with a higher order meta-term. The constant replaced and all the variables assigned to in the body are used as arguments for the meta-term. This then gives us a new candidate invariant, which we use with the expectation that rippling will instantiate the meta-term. The replacing constants by terms patch is given in more detail in figure 7.

Blockage:

$$H \vdash G[g(f_1(\ldots), f_2(\mathcal{X}, \ldots))]$$

Critic preconditions:

- precondition 1 of the wave method succeeds.
- precondition 2 of wave has the potential to succeed, *i.e.*
  1. There exists a sub-term T of G that contains wave-front(s).

$$g(f_1(\ldots), f_2(\mathcal{X}, \ldots))$$

  2. There exists a wave-rule which has the potential to match with T

$$g(f_1(\ldots), f_3(\ldots)) \Rightarrow \ldots$$

Patch specification:
Coerce precondition 2 by replacing the blocking constant with a meta-variable *e.g.*

$$f_2(\mathcal{X}, \ldots) \text{ becomes } f_2(M(\ldots), \ldots)$$

Where M is a meta-variable.

Fig. 7. Patch: Replacing Constants by Terms

4.2 Strengthening invariants

Gries [9] explains a simple, but powerful, heuristic for dealing with blocked proof obligations. The idea is to assume the very thing that we are trying to prove. This has the effect of instantly making the proof obligation trivial. However now we have to show that this proof obligation is itself invariant.

At the end of a ripple proof we are sometimes left with some proof residue. This is a sub-term that requires some further proof. This residue can sometimes be proved using the simplify method, but when it cannot, we have to start the invariant proof process again. The invariant method tries to annotate a subgoal. If we are unable to annotate it then we are blocked since we will not be able to ripple.

Using this failure to find any annotatable differences between a subgoal and the hypothesis we can refine our candidate invariant using the heuristic above. *i.e.* the conclusion we are unable to annotate is added to our candidate invariant and we start the proof again. For example suppose a proof attempt fails on the goal:

$$(r = exp(\mathcal{X}, \mathcal{Y} - y) \wedge y \neq 0) \rightarrow x = \mathcal{X}$$

The failure is due to the proof residue $x = \mathcal{X}$ which we can neither derive from the hypothesis nor annotate with respect to any of them. The patch is to add $x = \mathcal{X}$ to the invariant. Thus the new candidate invariant becomes:

$$r = exp(\mathcal{X}, \mathcal{Y} - y) \wedge x = \mathcal{X}$$

The strengthening invariant critic is given in more detail in figure 8.

Blockage:

$$H \vdash G$$

Critic preconditions:

- precondition 2 of invariant method fails, *i.e.*
  1. The conclusion is unannotated.
  2. We cannot annotate G with respect to a member of H.

Patch specification:
Coerce precondition 2 by introducing those parts of G that we are unable to annotate in our candidate invariant.

Fig. 8. Patch: Strengthening invariants

4.3 Relation to Inductive Proof

Previously critics have been used in the context of inductive proof [13–15]. The critics described patch various failed inductive proof attempts by either doing a generalisation, revising the type of induction chosen, doing a case split or discovering a missing lemma. Previously [16] we showed how we were able to move a generalisation critic for inductive proof [14] over to the domain of imperative program verification to discover tail invariants. The critics report here, however, have been developed straight from standard heuristics. So how do these critics relate to any thing in the inductive proof?

In fact the replace constants by terms critic is very similar to the induction revision critic. This is not unsurprising as often we are replacing a constant by a local loop variable used to ensure termination *i.e.* an induction variable. This is reflected in the pattern of failure we observe in the ripple method. Both the replace constants by variables critic and the induction revision critic, achieve partial success with the second precondition of rippling, *i.e.* there is a wave-rule match.

The failure pattern for the strengthening invariants critic corresponds to a complete failure to annotate the conclusion with respect to a hypothesis. It has been observed that a similar failure pattern occurs when dealing with inductions involving mutually recursive functions.

5 Refining Invariants

In this section we briefly explain the overall strategy used to find invariants and thus construct proofs of loop programs. We then show this in action using an example.

The above methods and critics have been encoded within the CLAM proof planning system. CLAM is allowed to exhaustively search before attempting to fire any critics.

The replacement of constants and strengthening invariants critics together allow us to successively refine approximation to our invariant. The idea is to use the post condition as our initial guess and successively refine our invariant. If we have proved that our new property is invariant we then have to show that it actually implies the post condition *i.e.* it is not too weak. For example suppose we have shown I to be invariant over successive refinements of our post condition Q we have to show that $(I \wedge \neg B) \rightarrow Q$ where B is the loop guard.

Example

For example consider the program in figure (1), and the wave-rules (8), (9) and (10) introduced above.

First Approximation (Post-condition): We take the post-condition:

$$r = exp(\mathcal{X}, \mathcal{Y})$$

as our first approximation to the invariant. This gives the verification condition:

$$(r = exp(\mathcal{X}, \mathcal{Y}) \wedge y \neq 0) \rightarrow r * x = exp(\mathcal{X}, \mathcal{Y})$$

This goal cannot be simplified so we apply the invariant method giving:

$$(r = exp(\mathcal{X}, \mathcal{Y}) \wedge y \neq 0) \rightarrow r * x = exp(\mathcal{X}, \mathcal{Y})$$

Now we are stuck. We are unable to apply any of the wave rules. We are left resorting to using the critics to try and patch our goal. Firstly we notice that we have a potential match between the goal:

$$r * x = exp(\mathcal{X}, \mathcal{Y})$$

and the left hand side of the wave-rule (10):

$$A * C = B * D$$

We apply a look back to decide which constant to replace. The right hand side of the wave-rule (9):

$$exp(A, B) * A \quad \text{matches with} \quad exp(\mathcal{X}, \mathcal{Y}) * D$$

the result of our potential match. Looking at the left hand side of this wave-rule we are able to determine that the difference came from the second argument of exp, so we replace the $\mathcal{Y}$ with the meta term $M_1(r, y, \mathcal{Y})$ giving the new candidate invariant:

$$r = exp(\mathcal{X}, M_1(r, y, \mathcal{Y}))$$

Second Approximation (Result of replacement of constants critic): This new candidate invariant is used to set up a verification condition:

$$(r = exp(\mathcal{X}, M_1(r, y, \mathcal{Y})) \wedge y \neq 0) \rightarrow r * x = exp(\mathcal{X}, M_1(r * x, y - 1, \mathcal{Y}))$$

The proof attempt continues:

$$r * x = exp(\mathcal{X}, M_1(r, y, \mathcal{Y})) \qquad \text{Conclusion}$$

$$r * x = exp(\mathcal{X}, M_1(r * x, y - 1, \mathcal{Y})) \qquad \text{By Invariant}$$

$$r * x = exp(\mathcal{X}, (M_2(r * x, y - 1, \mathcal{Y}) - y) + 1) \qquad \text{By Wave with (8)}$$

$$r * x = exp(\mathcal{X}, M_2(r * x, y - 1, \mathcal{Y}) - y) * \mathcal{X} \qquad \text{By Wave with (9)}$$

$$r = exp(\mathcal{X}, M_2(r * x, y - 1, \mathcal{Y}) - y) \wedge x = \mathcal{X} \qquad \text{By Wave with (10)}$$

$$x = \mathcal{X} \qquad \text{By Fertilize}$$

The first application of wave has the effect of finding an instantiation for M_1 to be $\lambda X.\lambda Y.\lambda Z.\ M_2(X,Y,Z) - Y$. The fertilization step has the the effect of projecting the meta-variable onto it's third argument *i.e.* M_2 is instantiated to $\lambda X.\lambda Y.\lambda Z.\ Z$, *i.e.* $M_1(r,y,\mathcal{Y}) = \mathcal{Y} - y$.

Now we are stuck again, this proof obligation is not trivial and we are unable to annotate it. The strengthening critic now fires. Our critics tell us to try showing that it is invariant, by adding it to our previous invariant giving the next approximation:

$$r = exp(\mathcal{X}, \mathcal{Y} - y) \wedge x = \mathcal{X}$$

Third Approximation (Result of strengthening critic): Again we set up a verification condition:

$$(r = exp(\mathcal{X}, \mathcal{Y} - y) \wedge x = \mathcal{X} \wedge y \neq 0) \rightarrow (r * x = exp(\mathcal{X}, \mathcal{Y} - (y-1)) \wedge x = \mathcal{X})$$

The proof now follows:

$$\begin{array}{cr}
r * x = exp(\mathcal{X}, \mathcal{Y} - (y-1)) \wedge x = \mathcal{X} & \text{Conclusion} \\[1em]
r * x = exp(\mathcal{X}, \mathcal{Y} - (y-1)) & \text{By Simplify} \\[1em]
r * x = exp(\mathcal{X}, \mathcal{Y} - (y-1)) & \text{By Invariant} \\[1em]
r * x = exp(\mathcal{X}, (\mathcal{Y} - y) + 1) & \text{By Wave with (8)} \\[1em]
r * x = exp(\mathcal{X}, \mathcal{Y} - y) * \mathcal{X} & \text{By Wave with (9)} \\[1em]
r = exp(\mathcal{X}, \mathcal{Y} - y) \wedge x = \mathcal{X} & \text{By Wave with (10)} \\[1em]
x = \mathcal{X} & \text{By Fertilize} \\[1em]
true & \text{By Simplify}
\end{array}$$

So we have shown that $r = exp(\mathcal{X}, \mathcal{Y} - y) \wedge x = \mathcal{X}$ is an invariant to the loop body. To finish the proof we check that our invariant satisfies the first and third premises of the while rule.

6 Results and Implementation

This work on invariant discovery has been implemented within the CLAM proof planner. It describes a systematic framework for refining initial approximations to a loop invariant automatically in a principled way. Initial results using examples from the literature are promising, see table (1). The heuristics described

Program	Post Condition	Invariant Discovered
exp	$r = exp(\mathcal{X}, \mathcal{Y})$	$r = exp(\mathcal{X}, \mathcal{Y} - y) \wedge x = \mathcal{X}$
sum	$r = sum(\mathcal{X})$	$r = sum(x)$
times	$r = \mathcal{X} * \mathcal{Y}$	$r = (\mathcal{X} - x) * \mathcal{Y} \wedge y = \mathcal{Y}$
factorial	$r = fac(\mathcal{X})$	$r = fac(x)$
min-array	$m = min(i, 0 \leq i < \mathcal{N} : a[i])$	$m = min(i, 0 \leq i < n : a[i])$
sum-array	$s = \sum_{i=0}^{i<\mathcal{N}} a[i]$	$s = \sum_{i=0}^{i<n} a[i]$
min-sum-seg	$s = min\,(i, j, 0 \leq i < j \leq \mathcal{N}$ $\sum_i^j a[i])$	$s = min(i, j, 0 \leq i < j \leq n : \sum_i^j a[i])$ $\wedge c = min(i, 0 \leq i < n : \sum_i^n a[i])$

Program `exp` is given in figure (1). Programs `sum` and `factorial` computes the sum and factorial of the first $\mathcal{X}$ natural numbers respectively. Program `times` multiplies two natural numbers by successive addition. Programs `min-array`, `sum-array` and `min-sum-seg` find the least element of an array, the sum of an array and the minimum sum segment of an array respectively.

Table 1. Invariant Discovery Results

Program	Critic used
exp	RC and S
sum	RC
times	RC and S
factorial	RC
min-array	RC
sum-array	RC
min-sum-seg	RC and S

RC indicates the application of the replace constant critic. S indicates the application of the strengthening critic

Table 2. Critic Usage

have been used for more complex problems involving arrays, notably Gries' minimum sum segment problem [9]. Our approach is syntactically driven and requires defining equations and properties needed to complete the proof. There is no restriction that these defining equations are recursive or the specification is immediately executable. In the example above for the program in figure 1 we use a constructive definition of *exp* to prove a destructive version of the program.

7 Related Work

Early research into Automatic Programming investigated heuristic based methods for discovering loop invariants. Wegbreit [23] developed an approach in which a candidate invariant was incrementally refined using both domain-independent and domain-specific heuristics. The choice of refinement was guided by the satisfiability and validity of the current candidate invariant. The theorem proving and heuristic components were only loosely coupled. This was reflected in other heuristic approaches at the time [18]. Wegbreit hinted, however, that a closer relationship between the heuristic guidance and the theorem prover would be desirable. The proof planning framework in which our heuristics are expressed enables this close relationship to be achieved.

Our approach, like all heuristic based techniques, is not complete. A complete approach has been designed [7] based upon a novel unskolemization technique for deriving logical consequences of first-order formulae. Completeness, however, comes with a price. In practice this means that any inductive lemmas required for a particular verification task must be provided by hand.

Our technique is very tightly coupled to the syntactic structure of the initial invariant. A more semantic approach to verification is described by Mili [20], where program semantics are found independently of their specifications and heuristics are given for discovering invariant properties. Here programs are described by relations of input sets to output sets. Bundy and Lombart [4] have formalised the notion of relational rippling to reason with relations describing logic programs.

8 Future Work

Extensions to the above framework build upon the premise that the initial invariant may be wrong so a proof will not be found. The critic mechanism will be used to refine this initial invariant based upon the partial success of our methods. The nature of the planner's methods allows a systematic analysis of method failure. Another way to find critics is to look at the standard heuristics and find out how these can be encoded by analysing how the proof attempt fails.

Other future work will involve extending the system to deal with examples to those with nested loops, branching constructs within loop bodies and array assignment.

We are currently investigating whether Mili's relations can be used in conjunction with relation rippling and whether a combined approach has merit.

9 Conclusion

The discovery of loop invariants represents a bottle neck within the automatic verification of imperative programs. Building upon the proof planning framework, we have shown that by the productive use of failed proof attempts we are able to automate the synthesis of suitable invariants.

This work exploits previous work on identifying critics for inductive proof. The proof plans for both families of proofs are very similar, which is unsurprising given the relationship between iteration and recursion. We move the heuristics across domains to facilitate an integration with an interactive environment.

Acknowledgments

The authors wish to thank Greg Michaelson for his help by misunderstanding rippling for us. We would also like to thank the LOPSTR'98 anonymous reviewers for their helpful comments of the extended abstract.

References

1. R.C. Backhouse. *Program Construction and Verification.* Prentice Hall, 1986.
2. David Basin and Toby Walsh. Annotated rewriting in inductive theorem proving. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.
3. A. Bundy. The use of explicit plans to guide inductive proofs. Research Paper 349, Dept. of Artificial Intelligence, University of Edinburgh, 1988. Short version published in the proceedings of CADE-9.
4. A. Bundy and V. Lombart. Relational rippling: a general approach. In C. Mellish, editor, *Proceedings of IJCAI-95*, pages 175–181. IJCAI, 1995. Longer version to appear as a DAI research paper.
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. A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. Research Paper 507, Dept. of Artificial Intelligence, University of Edinburgh, 1990. Appeared in the proceedings of CADE-10.
7. Chadha and Plaisted. On the mechanical derivation of loop invariants. *JSL*, 15:705–744, 1993.
8. E. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
9. D. Gries. A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming*, 2:207–214, 1982.
10. David Gries. *The Science of Programming.* Springer-Verlag, New York, 1981.
11. J.T. Hesketh. *Using Middle-Out Reasoning to Guide Inductive Theorem Proving.* PhD thesis, University of Edinburgh, 1991.
12. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.
13. 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.
14. A. Ireland and A. Bundy. Extensions to a Generalization Critic for Inductive Proof. In M.A. McRobbie and J.K. Slaney, editors, *13th 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.
15. 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.
16. A. Ireland and J. Stark. On the Automatic Discovery of Loop Invariants. In *Fourth NASA Langley Formal Methods Workshop*, number 3356 in NASA Conference Publications, 1997. Also available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/1.

17. A. Kaldewaij. *Programming: The Derivation of Algorithms.* Prentice Hall, 1990.
18. S.M. Katz and Z. Manna. A heuristic approach to program verification. In *Proceedings of IJCAI-73.* International Joint Conference on Artificial Intelligence, 1973.
19. I. Kraan, D. Basin, and A. Bundy. Logic program synthesis via proof planning. In K.K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, pages 1–14. Springer-Verlag, 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-92-244 and Edinburgh DAI Research Report 603.
20. A. Mili, J. Desharhais, and F. Mili. *Computer Program Construction.* Oxford University Press, 1994.
21. D. Miller and G. Nadathur. An overview of λProlog. In R. Bowen, K. & Kowalski, editor, *Proceedings of the Fifth International Logic Programming Conference/ Fifth Symposium on Logic Programming.* MIT Press, 1988.
22. A. Smaill and I. Green. Higher-order annotated terms for proof search. Technical report, Dept. of Artificial Intelligence, University of Edinburgh, 1996. To appear in proceedings of TPHOLs'96.
23. Wegbreit. Heuristic methods for mechanically deriving inductive assertions. In *Proceedings of IJCAI-73.* International Joint Conference on Artificial Intelligence, 1973.