Refinement and Term Synthesis in Loop Invariant Generation

Ewen Maclean[1], Andrew Ireland[1], Robert Atkey and Lucas Dixon

[1] Heriot-Watt University
{eahm2,air}@macs.hw.ac.uk
[2] University of Edinburgh
{bob.atkey,lucas.dixon}@ed.ac.uk

Abstract

We present a technique for refining incorrect or insufficiently strong loop invariants in correctness proofs for imperative programs. We rely on previous work [16] in combining program analysis and Proof Planning, and exploit IsaPlanner's use of meta-variables and goal-naming to generate correct loop invariants. We present a simple example in detail and discuss how this might scale to more complex problems.

1 Introduction

Our primary interest is the automatic generation of correct and sufficiently strong loop invariants to allow correctness proofs about imperative programs to succeed. Here we present a simple example for proving exception freedom from array bounds errors. When proving imperative programs correct, sufficiently strong loop invariants must be discovered. Existing tools and techniques for discovering loop invariants are successful for array bound exception freedom proofs – where the access to an array is shown not to fall outside its bounds – but we propose to extend these techniques by using proof failure analysis to refine candidate loop invariants.

This paper outlines a cooperative approach where program analysis systems can be combined with theorem proving in a Higher Order setting. Our hypothesis is that combining the two we can successfully infer correct loop invariants for difficult problems about functional correctness. This is an extension of an idea described for example in [16]. In the work described here we focus on Higher Order Logic but as is described in Section 6, it is being performed in parallel with work in automating proof in Separation Logic [19].

2 Existing Technologies

We describe first existing systems which find candidate loop invariants, and go on to a description of the important features of IsaPlanner – the theorem proving system we adopt.

2.1 Program Analysis Systems

In our work to date we have explored different systems in order to determine the likely loop invariant candidates which would be suggested. Existing techniques for discovering loop invariants roughly fall into four main categories:

Dynamic Approaches This technique runs input templates on the loops and uses techniques such as machine learning to try and work out invariants from the results after the application of the loop. Work of this kind is the Daikon system, which has been successfully integrated in ESC/Java2, as presented in [22].

Top-down Static Approaches [21] presents work in taking the postcondition of a loop, and propagating it through the loop backwards in order to generate candidates for the loop invariant.

Bottom-up Static Approaches The most research has been done on this technique, which works by propagating the precondition of a loop forwards through the loop in order to yield candidates for loop invariants. The various different techniques used are too numerous for this report, but some pertinent techniques are those used in [9] for calculating linear equalities in loop invariants, and in [11] for evaluating program states using difference equations.

Predicate Abstraction Techniques Predicate abstraction is used widely in conjunction with model checking for program verification. It places abstracted predicates at every choice point in the control flow of a program, and refines them as necessary. Predicate abstraction has also been used successfully in the Houdini system [12] to generate loop invariants for ESC/Java.

Constraint-based Methods Possibly the most sophisticated and modern techniques use complex mathematics such as Gröbner bases to calculate non-linear constraints on invariants and then extract actual program invariants. This is demonstrated in [17] for example.

Some of these systems are combined with provers and can verify the correctness of a loop invariant. We are interested in cases where the loop invariant is not strong enough and must be modified.

2.2 IsaPlanner

Proof assistants, such as Isabelle [23], Coq [27] and HOL [13], provide a framework for formalisation tasks such as software verification and mechanised mathematics. Typically, automation is developed by writing programs, called *tactics*, that combine operations from a small trusted kernel. Although many forms of proof automation are already available, developing new tactics and extending existing ones can be difficult. Higher-level concepts, such as search space and heuristic guidance, must be developed on top of the the logical kernel. *Proof Planning* is an approach to providing this kind of high-level machinery in order to encode and apply common patterns of reasoning [6]. Further to this machinery there is a mechanism for recognising common failure patterns and suggesting ways of "patching" the failed proof, which is encapsulated in *Proof Critics*. We refer to the set of critics built-up for a particularly theory as its library of critics.

IsaPlanner [10] is a Proof Planner for the Isabelle proof assistant. The features of the system which are of particular importance for the work in this paper are:

- it provides a programatic way to manage the names of assumptions and goals;
- it efficiently manages the meta-variables which can be shared across different conjectures, and which once instantiated result in several theorems;
- it allows a large collection of automated proof methods to be combined, including counter-example finding, inductive theorem proving with rippling [4], linear arithmetic decision procedures, simplification, and tableaux methods.

We now briefly outline the nature of these features and the arguments that justify finding a solution for them.

2.2.1 Naming Goals and Assumptions

Most tactic based theorem provers represent the assumptions to a theorem (and those to a goal) as a list of terms. A tactic that performs forward reasoning specifies the assumptions on which it works by indices in the list. Similarly, applying tactics to goals typically involves providing an index for the goal. The result is that the list has new elements inserted, and possibly old elements re-arranged. Such an approach has two problems:

- To perform further reasoning on another assumption its position must be re-determined, possibly involving a complete traversal of all assumptions, even if its old position were known. Positions, which are used as names, are not stable over a tactics application. The same problem occurs for subgoals held in a list. Index-based names for referring to goals are not stable over tactic application. Tactics can re-order goals and insert new ones, as well as accidentally solve the goal you planned to tackle with another procedure.
- Having too many assumptions is not a theoretical problem in logics with weakening, as they can be ignored. But from the perspective of automated theorem proving they are problematic. For instance, certain unfortunate assumptions can cause simplification tactics to loop forever, or simply perform the wrong simplification. Some mechanism to say which assumptions should be used is needed.

Structured proof languages, such as Isar and Mizar, provide the means for the user to name assumptions as they interactively write a proof script. However, most proof systems have no such mechanisms for managing the names of assumptions, conjectures, and goals. IsaPlanner is the first system we know of to explicitly give stable names to assumptions and goals, and provide a programatic means for the generation of names to the tactic language. This allows proof machinery to refer to names of assumptions and goals which helps make the Proof Planning techniques applied more robust and simpler to express.

2.2.2 Meta-variables

Most tactic based systems struggle with proof automation when goals contain meta-variables - such as those resulting from existentially quantified variables in the goal or universally quantified ones in assumptions. The inherent logical problem with such variables for proof automation is that the search space suffers an exponential explosion. Furthermore, from a pragmatic point of view, the work involved in managing instantiations is difficult to achieve in a verifiable and efficient manner.

When systems do provide a form of meta-variables, it can only be used within a single proof attempt. Typically, several partial proof attempts cannot be started if they share meta-variables. This is for two main reasons. Firstly, meta-variables are free-variables over a sequent; there is usually no global name space for free/meta-variables - having one would cause several other problems including the inability to perform proof checking or proof search in parallel. The second reason is that meta-variables are not stable over tactic application. Tactics often instantiate old ones, rename them, and introduce new ones. The pragmatic problem with meta-variables is that upon instantiation, every term in which the variable occurs must be updated. Without efficient means to track the dependencies, or lazily instantiate variables, this becomes a slow operation.

IsaPlanner deals with these issues by naming meta-variables in a manner that is stable over tactic application and shared over a collection of proof attempts. Managing dependencies then reduces to operations on a binary relation between two finite sets. The two sets being

the variable names and the set of names for assumptions, conjecture and goals. Allowing meta-variables in conjectures also allows synthesis to be treated in a first class manner where a conjecture in refined and modified so that the final result is not abstracted away to an existential variable. This allows the instantiation found to be used in other proof attempts. For instance, this has been used to synthesise a custom induction scheme as part of the inductive proof attempt for proving the correctness of quicksort [5].

2.2.3 Combining Proof Tools

The final feature of IsaPlanner used in this paper is automated proof tools. These include many powerful automatic tactics from Isabelle such as the simplifier, the classical reasoner, counter-example finding, and linear arithmetic [8] as well as the efficient inductive theorem proving tools in IsaPlanner [10].

2.3 Current Automation

At present we are able to automatically generate the verification conditions and generate correct loop invariants. We can prove array bounds exception freedom for some simple programs which manipulate arrays which have no user provided annotation. Section 4 shows a solution of an exception freedom verification problem which has been done completely automatically. Throughout this section when we write "we" we are referring to the reader's interpretation of the automatic proof as produced by IsaPlanner.

Although the problem shown in Section 4 is one which would be solved by most program analysis systems, we describe our methodology and extend it to the more difficult functional correctness problem in Section 5. In this Section the methodology is combined by hand, and we claim no more automation than the verification condition generation. We fully expect however that the proof itself and synthesis problem which is central to the invariant generation are automatable.

3 Verification Condition Generation

Our verification condition generation procedure follows standard Hoare style rules [14] to generate entailments in Higher Order Logic. Initially, we ignore the values of array elements, only concentrating on exception freedom. The Hoare Logic rules for arrays are initially ignored, and assertions are inserted into the code where array elements are accessed to ensure the index is within the requisite bounds. We introduce here the rules we use to generate the weakest-precondition entailment, and give an example program. Extensions to functional correctness are given in Section 5.

3.1 Imperative Language

The version of Java we consider is based on Middleweight Java [3], extended with loops and arrays. This is a cut-down version of Java that retains enough features to demonstrate the properties we require, but is small enough to allow quick experimentation. In the proofs that follow, all types of variables are implicitly integers since this is the type given in the programming languages. For this reason we write $i + 1$ instead of $s(i)$ since this is more pertinent to the inductive structure of the natural numbers.

3.2 Exception Freedom Rules

We use the standard notion of Hoare Triples to generate entailments for correctness proofs. In the case of verification of array bounds correctness – i.e. that no array index is out of the bounds of the array – we are not concerned with the value of array elements. Figure 1 shows the set of Hoare Logic rules that we use. Importantly, the only rule which requires creative input is the rule for the `while` construct in our imperative language since a loop invariant must be discovered.

$$\frac{}{\{P[E/x]\}x := E\{P\}} \qquad \frac{\{P\}C\{Q\} \quad \{Q\}D\{R\}}{\{P\}C; D\{R\}}$$

$$\frac{\{Cond \wedge P\}C\{Q\} \quad \{\neg Cond \wedge P\}D\{Q\}}{\{P\}\texttt{if } Cond \;\; C \;\; \texttt{else } D\{Q\}} \qquad \frac{\{Cond \wedge P\}C\{P\}}{\{P\}\texttt{while } \; Cond \;\; C\{\neg Cond \wedge Q\}}$$

Figure 1: The Hoare Rules used to Generate Verification Conditions

4 Example Problem

Figure 2 shows a program which makes a copy of an array with the elements reversed. We use a weakest precondition analysis to generate the set of hypotheses and conclusions shown in Figure 3. Here $\mathcal{X}$ stands for a meta-variable which depends on the free variables in the hypotheses and conclusions. This is equivalent to an existentially quantified higher-order variable standing for the function which describes the loop invariant. For example, a natural candidate for the loop invariant would be

$$\mathcal{X} \equiv \lambda a, i.\ i \geq 0 \wedge i < len(a)$$

since this describes the limits of the loop variable imposed in the code, noticing that the index is incremented each pass through the loop. At this point we are disregarding the pre and post conditions since we are interested purely in exception freedom.

```
 procedure revArray(int[ ] a) {
{α0 = elements(a, 0, len(a))}
    int[ ] b = new int[a.length];
    int i = 0;
    while (i! = a.length){
        b[i] = a[a.length − (i + 1)];
        i = i + 1;
    }
{elements(a, 0, len(a)) = α0 ∧ elements(b, 0, len(b)) = rev(α0)}          }
```

Figure 2: A procedure for copying an array with the elements reversed

We consider proving that there are no array bounds errors in the loop of the program shown in Figure 2. This verification has been performed in IsaPlanner and the proof is totally automatic thanks to the mechanisms described in Section 2.2, using Isabelle's built in `auto` tactic

and its counter-example checker. We choose this particular verification in order demonstrate the invariant generation techniques we develop. Existing systems such as ESC/Java2 [22] and Houdini [12] are already capable of proving this program free from array bounds exceptions. We discuss a more complicated example in Section 5.

We imagine first of all that no loop invariant has been given and that it is represented by a meta variable $\mathcal{X}$ as described in Section 4. After a weakest precondition analysis, we perform Isabelle's `safe` tactic which reduces the goal to a set of hypotheses and conclusions as shown in Figure 3.

We omit some details of the result of the verification generation process, and present slightly simplified versions for presentation clarity. While we are just proving the correctness of the loop, we can assume $len(a) = len(b)$ since this is set at the start of the code when the array b is allocated. At the assignment of an array cell, the variable is changed and an extra hypothesis is added ensuring that the length of the array is unchanged. This is represented by $len(b') = len(b'')$ in the code, which is a simplification of a full description of array update as applied in Section 5. In the presentation, b' is the state of array b at some point at the start of the loop, and b'' is the state of array b at the end of the loop code.

h1 :$\mathcal{X}(a, b', i)$	c1 :$i \geq 0$
h2 :$i \neq len(l)$	c2 :$i < len(b')$
h3 :$len(a) = len(b)$	c3 :$len(a) - (i + 1) \geq 0$
h4:$len(b') = len(b'')$	c4 :$len(a) - (i + 1) < len(a)$
	c5 :$\mathcal{X}(a, b'', i + 1)$

Figure 3: The Verification Conditions Generated with no Loop Invariant

4.1 Dependent Bi-Abduction

[7] presents a method for automatically annotating programs with pre and post conditions using a method known as *bi-abduction*. This is focussed on verification of programs which use pointers, and uses separation logic to verify properties about the shape of inductive structures on the heap (see Section 6).

Abduction is a proof-technique which determines what must be true in hypotheses in order to render a conclusion correct. For example, consider the following theorem:

$$i, j, n : \mathbb{N}.\ j \leq n \ \vdash\ n - (i + j) \geq 0 \wedge n - j \geq 0$$

The conclusion $n - (i + j) \geq 0$ cannot be proven from the hypotheses. In order for the proof to go ahead, we must add the hypothesis that $i \leq n - j$. Determining this extra hypothesis is a form of abduction.

The premise behind bi-abduction is that it should be possible, by program analysis, to determine not only hypotheses which allow verification proofs to succeed, but also to determine extra conclusions which can be proved from the hypotheses. In the case of the work presented in [7] for example, this happens in program verification where such extra hypotheses pertain to a part of memory which is not passed to a procedure. The extra conclusions which are added pertain to parts of memory which must exist in order for post-conditions of procedures to be satisfied.

In our work we see a parallel between this work, and the synthesis and refinement of loop invariants. When using a proof system to determine a correct loop invariant, we need to both

use it as part of the hypotheses and in the conclusion. In order to prove the conclusions correct we can use abduction to determine what must be true of the loop invariant. However in adding extra hypotheses through instantiation of a loop invariant we also add extra conclusions. As a result this is bi-abduction, since we are determining new hypotheses and conclusion, but we observe dependencies between them.

4.2 Refinement

As discussed in [15], Proof Planning techniques can be used to analyse failure of a verification in order to discover correct loop invariants. We consider here the verification conditions shown in Figure 2, imagining that we are trying to prove the program correct given an insufficiently strong suggestion for a loop invariant – i.e. one which contains some necessary conditions, but not all of them.

IsaPlanner has a critics mechanism built-in as described in Section 2.2. Although some of the refinements described here have been done interactively, the proposal is to extend the library of critics to analyse the failure patterns we discuss here. This is an extension of the work described for example in [16].

4.2.1 Invariant Candidate Refinement by Addition of Conjuncts

Imagine that we have been given the suggestion

$$\mathcal{X} \equiv \lambda a, i.\ i \geq 0 \wedge i \leq len(a)$$

which creates the set of hypotheses and conclusions shown in Figure 4. When IsaPlanner attempts to prove this set of verification conditions it is capable of annotating which of the goals it has not been possible to prove. This report is motivated by the use of a counter-example checker, as described in Section 2.2. Each conclusion in Figure 4 is marked with a tick or a cross denoting whether it is provable from the hypotheses.

h1 : $i \geq 0$	c1 : $i \geq 0$	✔
h2 : $i \leq len(a)$	c2 : $i < len(b')$	✘
h3 : $i \neq len(a)$	c3 : $len(a) - (i+1) \geq 0$	✔
h4 : $len(a) = len(b)$	c4 : $len(a) - (i+1) < len(a)$	✔
h5 : $len(b') = len(b'')$	c5 : $i + 1 \geq 0$	✔
	c6 : $i + 1 \leq len(a)$	✔

Figure 4: The Verification Conditions Generated with Loop Invariant Suggestion (1)

In this case we analyse the failure pattern of the conclusions and see that the failing goal is $i < len(b')$. A naive patch would be to add this to the hypotheses by means of extending the loop invariant. This would involve instantiating the loop invariant to

$$\mathcal{X} \equiv \lambda a, i.\ i \geq 0 \wedge i \leq len(a) \wedge i < len(b).$$

This results in divergence since the set of generated verification conditions include the conclusion $i + 1 < len(b)$ which is included by the array access assertion conditions. Analysing this divergence is similar to the fixed point analysis presented in [20], and the divergence critic developed and discussed in [28].

Our solution is to augment the existing loop invariant candidate with an extra uninstantiated conjunct. We are motivated to do this in this case noticing that the existing loop invariant does not contain any relation between the variables which exist within the failed conclusion. Accordingly we write

$$\mathcal{X} \equiv \lambda a, b, i.\ i \geq 0 \wedge i \leq len(a) \wedge \mathcal{Y}(a, b, i)$$

which renders the verification complete given IsaPlanner's term synthesis machinery as described in Section 4.3.

4.2.2 More Complicated Invariant Candidate Refinement

Imagine now that we have been given the flawed invariant candidate

$$\mathcal{X} \equiv \lambda a, i.\ i \geq 0 \wedge i < len(a)$$

based purely on the initial inspection that all accesses to array elements should be within the bounds of the arrays. Given this loop invariant IsaPlanner produces the set of verification conditions shown in Figure 5.

h1 : $i \geq 0$	c1 : $i \geq 0$	✔
h2 : $i < len(a)$	c2 : $i < len(b')$	✘
h3 : $i \neq len(a)$	c3 : $len(a) - (i + 1) \geq 0$	✔
h4 : $len(a) = len(b)$	c4 : $len(a) - (i + 1) < len(a)$	✔
h5 : $len(b') = len(b'')$	c5 : $i + 1 \geq 0$	✔
	c6 : $i + 1 < len(a)$	✘

Figure 5: The Verification Conditions Generated with Loop Invariant Suggestion (1)

Now inspecting the failed conclusions we see that $i + 1 < len(a)$ and $i < len(b')$ cannot be solved. This time we have a failing conclusion which is not independent of the existing conjuncts in the loop invariant. We therefore identify the failing conjunct within the loop invariant to be $i < len(a)$. We want to replace this with a more general term, so we insert a corresponding meta-variable with arguments a, i. We also want to add an extra conjunct as described above to include the extra variable b. We thus remove the conjunct $i < len(a)$ and add the two uninstantiated conjuncts

$$\mathcal{Y}(a, i) \qquad\qquad \mathcal{Z}(a, i, b)$$

Since $\mathcal{Z}$ subsumes $\mathcal{Y}$ we adjust the loop invariant to

$$\mathcal{X} \equiv \lambda a, b, i.\ i \geq 0 \wedge \mathcal{Z}(a, i, b).$$

which renders the verification complete given IsaPlanner's term synthesis machinery.

4.2.3 Counter-Example Driven Refinement

When a verification attempt fails, IsaPlanner uses the built-in Isabelle counter-example checker to give values of variables for which the verification fails. This helps in giving suggestions for how the invariants should be refined. IsaPlanner exploits its nominals to identify which goals have failed. We can analyse the counter-example given. For example, in some cases the counter-example found involves mismatching list lengths, so we can automatically add *len* to the term synthesis machinery.

4.3 Term Synthesis

Given a refined loop invariant which has meta-variables inserted as described above, we can exploit IsaPlanner's term synthesis mechanism to generate and test loop invariant candidates. The set of function and constant symbols that we add to the term synthesis machinery are

$$0 \quad 1 \quad \lambda x.len(x) \quad \lambda x, y.x \leq y \quad \lambda x, y.x = y \quad \lambda x, y.x < y \quad \lambda x, y.x \wedge y$$

The current machinery for term synthesis combines terms using meta-variables until a ground term is found, which is then checked using the counter-example checker. Once a term is constructed for which a counter-example is not found, the proof is attempted using IsaPlanner and Isabelle's `auto` tactic.

This yields a correct loop invariant automatically in IsaPlanner. The process of narrowing the possible function symbols down to those shown above is currently hard-coded. In general choosing a complete set of such functions automatically is very difficult, but we aim to address this problem. Since we use breadth-first search, this is guaranteed to find a correct candidate if one exists.

5 Functional correctness

We extend the techniques described up to this point to verify the functional correctness of the program shown in Figure 2. This requires an extension of the verification condition generation to array assignment, and a language for proving the resulting entailment from the weakest precondition analysis.

5.1 Additional Rules

The rule for array assignment is given as

$$\overline{\{P[upd(a, i, E)/a]\}a[i] := E\{P\}}$$

and all occurrences of array lookup are replaced by the function *ele*. The function *elements* allows inductive decomposition of an array so we can reason about the structure of the data. This mimics the linked list structures found when reasoning about pointer programs [29, 2]. Extending Separation Logic to account for arrays is briefly discussed further in Section 6. Rewrite rules augmented with rippling annotations, known as wave rules, are given for these in Figure 6.

5.2 Entailment With Unknown Loop Invariant

When proving the functional correctness of the program, there are three stages to consider, all of which rely on a correct loop invariant. We represent the unknown loop invariant as usual with the meta-variable $\mathcal{X}$. This makes the bi-abduction more complicated than the example shown in Section 4. We must determine that the loop invariant follows from the pre-condition and code before the loop; that the loop invariant is indeed invariant; finally that the post-condition follows from the loop invariant and code after the loop. These three entailments are combined into the set of hypotheses and conclusions shown in Figure 7.

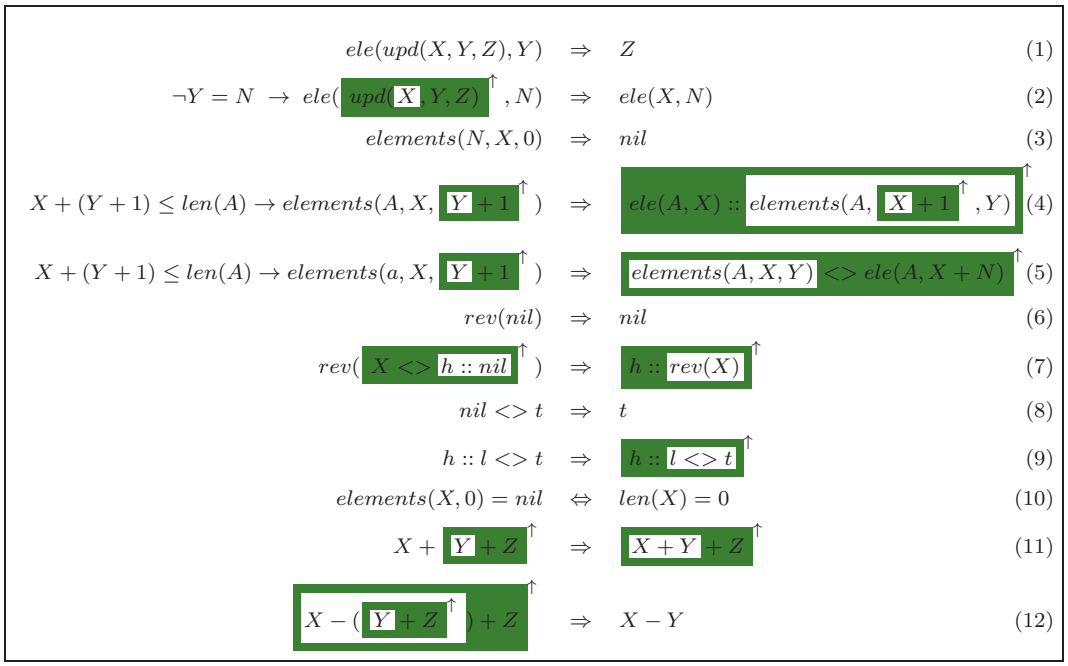

Figure 6: Wave Rules Used in the Functional Verification

h1 :$elements(a,0,len(a)) = \alpha_0$	c1 :$\mathcal{X}(0,a,b)$
h2 :$len(a) = len(b)$	c2 :$\mathcal{X}(i+1,a,b')$
h3 :$\mathcal{X}(i,a,b)$	c3 :$elements(a) = \alpha_o$
h4 :$b' = upd(b,i,ele(a,len(a)-(i+1)))$	c4 :$elements(b,0,len(b)) = rev(\alpha_0)$
h5 :$\neg i = len(a)$	
h6 :$\mathcal{X}(len(a),a,b)$	

Figure 7: Verification Conditions with an Uninstantiated Loop Invariant

5.3 Program Analysis Suggestion for Loop Invariant

A standard interpretation of how to generate a loop invariant for such a program is to create an invariant by analysing the "work done" and "work left to do" in a loop. In the case of the program we are analysing, we can suggest that the update to b has been done up to some point between $i = 0$ and $i < len(a)$, since i is incremented at each stage. A good initial candidate then assumes that the array b is updated up to, but not including, the value of i:

$$\mathcal{X} \equiv \lambda i, a, b.\ i \geq 0\ \wedge\ i \leq len(a) \wedge \forall j.0 \leq j < i\ \rightarrow\ ele(b,j) = \mathcal{Y}(a,b,j,i)$$

This yields a set of hypotheses and conclusions as shown in Figure 8.

5.4 Term Synthesis

We study the case where the set of verification conditions shown in Figure 8 is the initial state for the synthesis attempt. We use the post-condition, program analysis, and the set of function

h1 : $elements(a, 0, len(a)) = \alpha_0$	c1 : $\forall j.0 \leq j < 0 \ \rightarrow \ ele(b, j) = \mathcal{Y}(a, b, j, 0)$
h2 : $len(a) = len(b)$	c2 : $\forall j.0 \leq j < i + 1 \ \rightarrow \ ele(b', j) = \mathcal{Y}(a, b', j, i + 1)$
h3 : $\forall j.0 \leq j < i$ *to* $ele(b, j) = \mathcal{Y}(a, b, j, i)$	c3 : $elements(a, 0, len(a)) = \alpha_o$
h4 : $b' = upd(b, i, ele(a, len(a) - (i + 1)))$	c4 : $elements(b, 0, len(b)) = rev(\alpha_0)$
h5 : $\neg i = len(a)$	c5 : $0 \geq 0$
h6 : $\forall j.0 \leq j < len(a)$ *to* $ele(b, j) = \mathcal{Y}(a, b, j, len(a))$	c6 : $0 \leq len(a)$
h7 : $i \geq 0$	c7 : $i + 1 \geq 0$
h8 : $i \leq len(a)$	c8 : $i + 1 \leq len(a)$

Figure 8: Verification Conditions with Partial Instantiation

symbols available to create a large set of possible term symbols over which to instantiate the possible loop invariant. The set of possible term symbols is

$$0 \quad 1 \quad \lambda x, y.x + y \quad \lambda x, y.x - y \quad \lambda x, y.x = y \quad \lambda x, y.x \wedge y \quad \lambda x.len(x) \quad \lambda x, y.ele(x, y)$$

which will generate the correct loop invariant:

$$\mathcal{X} \equiv \lambda i, a, b.\ i \geq 0 \ \wedge \ i \leq len(a) \wedge \forall j.0 \leq j < i \ \rightarrow \ ele(b, j) = ele(a, len(a) - (j + 1)).$$

5.5 Proof

Given an instantiation of

$$\mathcal{Y} = \lambda a, b, i, j. \equiv ele(a, len(a) - (j + 1))$$

the interesting and non-trivial conjuncts to prove are c2 and c4 in Figure 8. We do not discuss here the actual proof in IsaPlanner, or the reasoning techniques which must be employed. Our interest is in the correct synthesis of a loop invariant. The proofs that follow are all automtable in IsaPlanner.

5.5.1 Goal c2

After some simplification for clarity of presentation, the entailment resulting from a weakest precondition analysis is

$$\begin{array}{r} elements(a, 0, len(a)) = \alpha_0 \ \ , i < len(a), i \geq 0 \\ \forall j.0 \leq j < i \ \rightarrow \ ele(b', j) = ele(a, len(a) - (j + 1)) \vdash \\ \forall j.0 \leq j < i + 1 \ \rightarrow \ ele(upd(b', i, ele(a, len(a) - (i + 1))), j) = ele(a, len(a) - (j + 1)). \end{array}$$

We now perform first a case split on i, proving a branch where $i = 0$ and a branch where $i > 0$. The first branch leads with $i = 0$ to

$$\begin{array}{r} elements(a, 0, len(a)) = \alpha_0 \ \ , i < len(a), i = 0 \\ \forall j.0 \leq j < 0 \ \rightarrow \ ele(b', j) = ele(a, len(a) - (j + 1)) \vdash \\ \forall j.0 \leq j < 1 \ \rightarrow \ ele(upd(b', 0, ele(a, len(a) - 1)), j) = ele(a, len(a) - (j + 1)) \end{array}$$

which reduces to

$$\begin{array}{r} elements(a, 0) = \alpha_0 \ \ , i < len(a), i = 0, j = 0 \\ \texttt{true} \vdash \\ ele(upd(b', 0, ele(a, len(a) - 1)), 0) = ele(a, len(a) - 1) \end{array}$$

which reduces to an identity with application of rule (1). In the second branch where $i > 0$ we obtain

$$\begin{array}{l} elements(a,0) = \alpha_0 \ \ , i < len(a), i > 0 \\ \quad \forall j.0 \leq j < i \ \rightarrow \ ele(b', j) = ele(a, len(a) - (j+1)) \vdash \\ \forall j.0 \leq j < i+1 \ \rightarrow \ ele(upd(b', i, ele(a, len(a) - (i+1))), j) = ele(a, len(a) - (j+1)) \end{array}$$

for which we now split up the conclusion into cases where $0 \leq j < i$ and $j = i$. This yields firstly the goal

$$\begin{array}{l} elements(a,0) = \alpha_0 \ \ , i < len(a), i > 0 \\ \quad \forall j.0 \leq j < i \ \rightarrow \ ele(b', j) = ele(a, len(a) - (j+1)) \vdash \\ \forall j.0 \leq j < i \ \rightarrow \ ele(upd(b', i, ele(a, len(a) - (i+1))), j) = ele(a, len(a) - (j+1)) \end{array}$$

for which rule (3) applies since we can prove that $0 \leq j < i \ \rightarrow \ j \neq i$. This reduces to

$$\begin{array}{l} elements(a,0) = \alpha_0 \ \ , i < len(a), i > 0 \\ \quad \forall j.0 \leq j < i \ \rightarrow \ ele(b', j) = ele(a, len(a) - (j+1)) \vdash \\ \qquad \forall j.0 \leq j < i \ \rightarrow \ ele(b', j) = ele(a, len(a) - (j+1)). \end{array}$$

For the case where $j = i$ we yield

$$\begin{array}{l} elements(a,0) = \alpha_0 \ \ , i < len(a), i > 0 \\ \quad \forall j.0 \leq j < i \ \rightarrow \ ele(b', j) = ele(a, len(a) - (j+1)) \vdash \\ \qquad ele(upd(b', i, ele(a, len(a) - (i+1))), i) = ele(a, len(a) - (i+1)) \end{array}$$

which using rule (1) reduces to an identity.

5.5.2 Goal `c4`

The goal we need to prove is

$$\begin{array}{l} elements(a,0,len(a)) = \alpha_0, \ \ len(a) = len(b) \\ \quad \forall j.0 \leq j < len(a) \ to \ ele(b, j) = ele(a, len(a) - (j+1)) \vdash \\ \qquad elements(b,0,len(b)) = rev(\alpha_0). \end{array}$$

We write this in a simpler form as

$$\begin{array}{l} \forall j.0 \leq j < len(a) \ \rightarrow \ ele(b, j) = ele(a, len(a) - (j+1)) \vdash \\ \qquad elements(b,0,len(a)) = rev(elements(a,0,len(a)) \end{array}$$

in order to prove this, we generalise the theorem. The Generalisation of the theorem to segments within the array is described by the diagram in Figure 5.5.2. This sort of generalisation is possible within IsaPlanner, and can be done automatically since the class of induction is bounded. This means that in the theorem the induction variable is constrained to a finite range. Once generalised we can set up an inductive proof on the length of the segment.

$$\begin{array}{l} \forall j.0 \leq j < len(a) \ \rightarrow \ ele(b, j) = ele(a, len(a) - (j+1)) \vdash \\ \qquad \forall m, n.len(a) < m \geq 0 \ \wedge \ len(a) \leq n \geq 0 \ \wedge \ len(a) \leq m + n \geq 0 \rightarrow \\ \qquad\qquad elements(b, len(a) - (m+n), n) = rev(elements(a, m, n)) \end{array}$$

This can be proved by IsaPlanner by induction on n and with subsequent case-splits on $s(n) > 0 \ \vee \ s(n) = 0$. In the case where $s(n) = 0$ the conclusion in trivial, so we concentrate here on the step case when we know that $s(n) > 0$. For ease of presentation we omit the precedents to the implications. This complicates the proof, but IsaPlanner is capable of dealing with this sort of problem in inductive proof. We now have the goal

$$\forall j.\ 0 \leq j < len(a) \ \rightarrow \ ele(b, j) = ele(a, len(a) - (j + 1)) \qquad \dagger$$
$$\forall m'.elements(b, len(a) - (' m + n), n) = rev(elements(a, m', n)) \ \vdash$$
$$elements(b, len(a) - (m + \boxed{n + 1}^{\uparrow}), \boxed{n + 1}^{\uparrow}) = rev(elements(a, m, \boxed{n + 1}^{\uparrow}))$$

We apply rules (4),(6) on the left and right of the equality to obtain the conclusion

$$\boxed{ele(b, len(a) - (m + n + 1)) :: elements(b, \boxed{len(a) - (m + \boxed{n + 1}^{\uparrow}) + 1}^{\uparrow}, n)}^{\uparrow} =$$
$$rev(\boxed{elements(a, m, n) <> ele(a, m + n)}^{\uparrow})$$

Now applying rule (7),(11) and (12) we obtain

$$\boxed{ele(b, len(a) - (m + n + 1)) :: elements(b, len(a) - (m + n), n)}^{\uparrow} =$$
$$\boxed{ele(a, m + n) :: rev(elements(a, m, n))}^{\uparrow}$$

at which point the induction hypothesis applies leaving us to prove the equality

$$ele(b, len(a) - (m + n + 1)) = ele(a, m + n)$$

which we can solve using hypotheses †. The hypothesis applies since we are in a branch where $n + 1 > 0$.

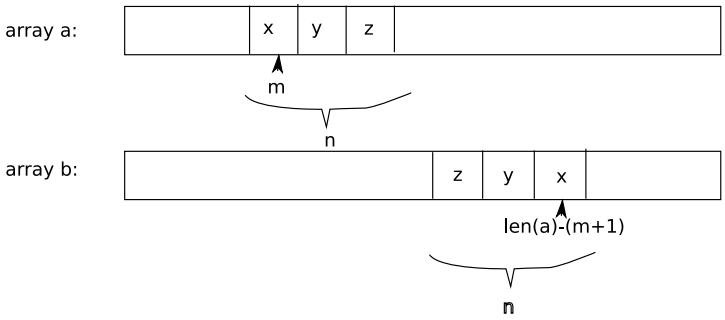

Figure 9: The generalised form of goal `c4` pertaining to array segments

5.6 Middle-Out Reasoning

As described in [18] and [26] for example, middle out reasoning can be applied in these situations to instantiate meta-variables such as $\mathcal{Y}$ during the progress of the proof. Our approach

here differs in that a term is constructed and tested in order to complete the proof. The key of the work is to make sure that the set of possible constant and function symbols is small, and that the eventual term which must be synthesise is minimal.

6 Separation Logic And Future Work

Separation Logic as described in [25] for example, is a substructural logic which is based on the Logic of Bunched Implications [24] and has been extended with Hoare Logic rules to describe manipulation of data on the heap. It is primarily designed for pointer programs where aliasing and resource management have traditionally been a difficult problem. We intend to extend our work on arrays by exploiting existing and ongoing work on functional correctness proofs in pointer programs, to treat arrays as indexed data structures on the heap.

The advantage of separation logic is that it exploits a semantics which introduces a separating conjunction which is associative-commutative and does not allow weakening or contraction. This means that one can ensure memory safety properties. Since we are interested in functional correctness we exploit the existing tools such as Smallfoot [1] to indicate how a proof proceeds when the values of data cells are not the focus of the proof. Arrays introduce a new problem, which is that the separating conjunction must be indexed to describe allocation and deallocation of array elements. Reynolds calls this the *Iterated Separating Conjunction*. This is denoted by an expression of the form $\odot_{exp=m}^{n} p(exp)$ where m and n are indices to the array. In this formalisation *emp* is given to mean the empty heap.

In the formalisation demonstrated here, since we are using IsaPlanner in conjunction with Isabelle/HOL, we have extended the logic with the function symbols *elements* and *ele* which describe the data within the array. The inductive definition of *elements*, given by equations (4) and (6), is comparable to the construction of the iterated separating conjunction whose definitions. The main difference is that instead of start and end markers for the array indices in the case of Separation Logic, we have a start marker and then a number of subsequent array cells. Figure 10 shows the equivalences between the definitions, and demonstrates that the Iterating Separating Conjunction approach is more expressive. The equivalent statement of the conclusion for example for goal `c4` then becomes

$$\odot_{i=len(a)-(m+n)}^{len(a)-(m+1)} b(i) = rev(\odot_{i=m}^{m+n-1} a(i)).$$

This definition is more elegant, and the underlying separating logic allows for a important safety aspects of arrays to be proved. In particular we can use the Iterating Separating Conjunction to reason about the size of arrays. For functional correctness we need to define new rules for the manipulation of the arrays. This means we will be adding terms to the *pure* part of the goal – i.e. that which does not contain any non-classical connectives. For example, a possible recursive definition of the *rev* function applied to the iterated separating conjunction would be:

$$rev(\odot_{i=m}^{m+s(n)} p(i)) \iff q(m) * \odot_{i=m}^{m+n} p(i) \ \wedge \ q(m) = p(m + s(n))$$

Here the *pure* part is what follows after the classical conjunction. After the heap assertions have been satisfied we will be left with a pure residue which is similar to the reasoning we have presented in Section 5. Other Proof Planning techniques such as rippling extend to Separation Logic formulae.

$m > n \to \odot_{i=m}^{n} p(i) \;\leftrightarrow\; emp$	$n \leq 0 \to elements(p, m, n) = nil$
$m = n \to \odot_{i=m}^{n} p(i) \;\leftrightarrow\; p(m)$	$elements(p, n, 1) = ele(p, n)$
$k \leq m \leq n + 1 \to \odot_{i=k}^{n} p(i) \iff$	$n + k + m \leq len(a) \;\to\; elements(a, m, n + k) =$
$(\odot_{i=k}^{m-1} p(i) * \odot_{i=m}^{n} p(i))$	$elements(a, m, n) <> elements(a, m + n, k)$
$\odot_{i=m}^{n} p(i) \iff \odot_{i=m-k}^{n-k} p(i + k)$	

Figure 10: A comparison between the Iterating Separation Conjunction and Our Approach

7 Conclusion

The approach presented here is the initial findings of our work on loop invariant generation techniques for functional properties of programs. We exploit existing work on Program Analysis to collect good candidates for loop invariants. These are often insufficient and need to be modified. In order to do this we use a refinement technique and make use of IsaPlanner's treatment of meta-variables and its ability to name hypotheses and goals.

Although the work is incipient we believe the success of the approach at verifying array bounds exception freedom will scale to functional correctness problems. In particular this work progresses in parallel with other work on verifying functional correctness of programs which manipulate pointers.

References

[1] J. Berdine, C. Calcagno, and P.W. OHearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137. Springer, 2006.

[2] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In Kwangkeun Yi, editor, *APLAS 2005*, volume 3780, pages 52–68. Springer-Verlag, 2005.

[3] G. M. Bierman, M. J. Parkinson, and A. M. Pitts. Mj: An imperative core calculus for Java and Java with effects. Technical Report UCAM-CL-TR-563, University of Cambridge, 2003.

[4] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, April 2005.

[5] A. Bundy, L. Dixon, J. Gow, and J. Fleuriot. Constructing induction rules for deductive synthesis proofs. In S. Allen, J. Crossley, K.K. Lau, and I. Poernomo, editors, *Proceedings of the ETAPS-05 Workshop on Constructive Logic for Automated Software Engineering (CLASE-05), Edinburgh*, pages 4–18. LFCS University of Edinburgh, 2006. Appears in ENTCS 2006.

[6] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *JAR*, 7(3):303–324, 1991.

[7] C. Calcagno, D. Distefano, P.W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.

[8] Amine Chaieb. Parametric linear arithmetic over ordered fields in isabelle/hol. In *Proceedings of the 9th AISC international conference, the 15th Calculemas symposium, and the 7th international MKM conference on Intelligent Computer Mathematics*, pages 246–260, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[10] L. Dixon and J. D. Fleuriot. Higher order rippling in IsaPlanner. In *Theorem Proving in Higher Order Logics*, volume 3223 of *LNCS*, pages 83–98, 2004.

[11] Bernard Elspas, Karl N. Levitt, Richard J. Waldinger, and Abraham Waksman. An assessment of techniques for proving program correctness. *ACM Computing Surveys*, 4(2):97–147, June 1972.

[12] C. Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proceedings of FME 2001*. LNCS 2021, Springer-Verlag, 2001.

[13] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 1969.

[15] A. Ireland. Towards automatic assertion refinement for separation logic. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering*. IEEE Computer Society, 2006.

[16] A. Ireland, B.J. Ellis, A. Cook, R. Chapman, and J. Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning: Special Issue on Empirically Successful Automated Reasoning*, 36(4):379–410, 2006.

[17] Neil D. Jones and Xavier Leroy, editors. *Non-linear loop invariant generation using Gröbner bases.* ACM, 2004.

[18] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for synthesis and induction. *Journal of Automated Reasoning*, 16(1–2):113–145, 1996. Also available from Edinburgh as DAI Research Paper 729.

[19] E. Maclean, A. Ireland, and I. Hind. Proving functional properties in separation logic. In *Automated Reasoning Workshop – Bridging the Gap Between Theory and Practice*, 2008.

[20] S. Magill, A. Nanevski, E. Clarke, and L. Lee. Inferring invariants in separation logic for imperative list-processing programs. In *Proceedings of the Third Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE '06)*, pages 47–60, 2006.

[21] Markus Müller-Olm and Helmut Seidl. Computing polynomial program invariants. *Inf. Process. Lett.*, 91(5):233–244, 2004.

[22] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001.

[23] L.C. Paulson. *Isabelle: A generic theorem prover*. Springer-Verlag, 1994.

[24] D. J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.

[25] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science*, 2002.

[26] 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.

[27] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, April 2004.

[28] T. Walsh. A divergence critic for inductive proof. *Journal of Artificial Intelligence Research*, 4:209–235, 1996.

[29] H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *Proceedings of FOSSACS*, 2002.