A Cooperative Approach to Loop Invariant Discovery for Pointer Programs

Andrew Ireland
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, Scotland, UK

Abstract. We propose a cooperative approach to reasoning about the correctness of pointer programs within the context of separation logic. A key hurdle to scalable program proof is the burden imposed by the need for loop invariants. Our proposed approach involves an integration of two complementary techniques. Firstly, a technique developed at CMU for discovering invariants based upon symbolic evaluation. Secondly, middle-out proof planning, a technique which has had significant success in automating the discovery of inductive lemmas and invariants. We believe that the complementary nature of these techniques will deliver significant advantages in terms of scalable pointer program proof.

1 Introduction

Pointers are a powerful and widely used programming mechanism. Developing and maintaining correct pointer programs, however, is notoriously hard. It is therefore highly desirable to be able to routinely reason about the correctness of pointer programs. The stumbling block to achieving such a goal has been the lack of scalable methods of reasoning. At the level of proof search, the need for invariants, and in particular loop invariants, remains a major obstacle to proof automation, and thus scalability. Here we propose a cooperative approach to reasoning. Using an invariant discovery technique developed at CMU [19], we describe how symbolic evaluation and middle-out proof planning [8] could be combined in order to support the automation of partial correctness proofs. In terms of logics, our proposal is grounded in separation logic [22,23], which addresses aspects of the scalability problem through its support for local reasoning.

The paper is structured as follows. Background on separation logic and a motivating example are introduced in §2. In §3, an overview of the CMU invariant discovery technique is provided, together with background material of middle-out proof planning. A high-level description of our proposed cooperative approach is given in §4, while the details are presented §4.1 and §4.2. The potential benefits of the approach are discussed in §5, while related and future work are described in §6 and §7 respectively.
2 Separation logic and the burden of loop invariants

Separation logic was developed as an extension to Hoare logic with the aim of supporting pointer program proofs. The key underlying idea is that the logic focusing the reasoning effort on only those parts of the heap that are relevant to a program. Here we give a brief introduction to separation logic, for a full account see [23]. Separation logic extends predicate calculus with new forms of assertions for describing the heap:

- empty heap: the assertion $\textit{emp}$ holds for a heap that contains no cells.
- singleton heap: the assertion $X \mapsto E$ holds for a heap that contains a single cell.
- separating conjunction: the assertion $P \land Q$ holds for a heap if the heap can be divided into two disjoint heaps $H_1$ and $H_2$, where $P$ holds in $H_1$ and $Q$ holds in $H_2$ simultaneously.
- separating implication: the assertion $P \rightarrow Q$ holds if the current heap $H_1$ is extended with a disjoint heap $H_2$ in which $P$ holds, then $Q$ will hold in the extended heap.

In order to assert that $X$ points to $E$ within an arbitrary heap we use $(X \mapsto E)$, which is defined to be $(\textit{true} \land (X \mapsto E))$. In what follows we will focus on pointers that reference a pair of adjacent heap cells. So we will use $(X \mapsto E_1, E_2)$ to denote $(X \mapsto E_1) \land (X + 1 \mapsto E_2)$. Moreover, we will use the dot operator to dereference components, e.g. $X:1$ evaluates to $E_1$.

To motivate our proposal we outline the application of our approach to verifying the partial correctness of in-place list reversal. This example is essentially the version of list reversal given by Reynolds in [23]. Following Reynolds, we introduce a predicate called $\textit{list}$ which relates the notion of a singly-linked list to the abstract notion of sequences. An inductive definition for $\textit{list}$ is given below:

$$
\text{list}([], Y, Z) \leftarrow \textit{emp} \land Y = Z
$$
$$
\text{list}([W|X], Y, Z) \leftarrow (\exists p. (Y \mapsto W, p) \land \text{list}(X, p, Z))
$$

Note that the first argument denotes a sequence, where sequences are represented using the Prolog list notation. The second and third arguments delimit the corresponding linked-list structure. Note also that the definition excludes cycles. The specification of list reversal requires additional constraints on sequences. These constraints can be achieved via the following definitions for sequence concatenation and reversal:

$$
\text{rev}([]) = []
$$
$$
\text{rev}([X|Y]) = \text{app}(\text{rev}(Y), [X])
$$
$$
\text{app}([], Z) = Z
$$
$$
\text{app}([X|Y], Z) = [X|\text{app}(Y, Z)]
$$

Using these definitions, we can now specify the list reversal program as shown in Fig 1. A key burden in automating the verification of such a program is the discovery of $R$, the loop invariant. In our proposed approach, we consider the
\{P\}  
j := \text{nil};  
\{R\}  
\textbf{while not}(i = \text{nil}) \textbf{do}  
k := i.2;  
i.2 := j;  
j := i;  
i := k  
\textbf{od}  
\{Q\}  

Note that $\alpha_{\text{init}}$ denotes the sequence of data values corresponding to the initial linked-list.

\begin{figure}[ht]
\centering
\begin{tabular}{ll}
\textbf{P} & $P : (\exists \alpha. \text{list}(\alpha, i, \text{nil}) \land \alpha_{\text{init}} = \alpha)$ \\
\textbf{Q} & $Q : (\exists \beta. \text{list}(\beta, j, \text{nil}) \land \alpha_{\text{init}} = \text{rev}(\beta))$
\end{tabular}
\caption{Specification and code for in-place list reversal}
\end{figure}

\begin{tabular}{ll}
\textbf{R} & $R : (\exists \alpha, \beta. \text{list}(\alpha, i, \text{nil}) \land \alpha_{\text{init}} = \text{rev}(\beta))$
\end{tabular}

We propose the use different techniques to support the discovery of each part. However, we will argue that the techniques are complementary.

3 Symbolic evaluation and middle-out proof planning

As mentioned in §1, our proposal builds upon a technique described in [19] for automatically inferring loop invariants in separation logic for imperative list-processing programs. Starting with a programmer supplied precondition, symbolic evaluation is used to calculate the effect of the code on the heap. In terms of loops, the repeated evaluation of the loop body is undertaken in order to identify a fixed point. The search for a fixed point is not guaranteed to converge, as a consequence the technique is not complete.

Proof planning [5] is a technique for automating the search for proofs through the use of high-level proof outlines, known as proof plans. A distinctive feature of proof planning is middle-out reasoning [8], where meta-variables are used to delay choice during the search for a proof. Middle-out reasoning has been used to greatest effect within the context of proof critics [11], a technique that supports the automatic analysis and patching of failed proof attempts. Proof critics have been applied successfully to the problems of inductive conjecture generalization [13, 14] and loop invariant discovery [16]. Most recently, the loop invariant discovery techniques have been integrated with light-weight program analysis and applied to industrial strength problems [15].
4 Proposed cooperation

Given a program, annotated with pre- and postconditions, our proposed approach aims to:

1. Discover a shape invariant via the CMU symbolic evaluation technique [19].
2. Generate a speculate data invariant, based upon shape invariant and the given postcondition.
3. Verify the shape invariant via proof planning.
4. Discover an instantiation for the speculative data invariant via middle-out reasoning.

A key feature of this proposal is that we treat the process by which the shape invariant is discovery as an oracle. That is, we are not concerned about soundness of teh discovery process since the shape invariant is verified via proof planning.

4.1 Symbolic evaluation

We focus first on the algorithm presented in [19] for automatically inferring shape invariants. A simplified list predicate, called $ls$ is used for encoding memory descriptions:

$$ls(p_1, p_2) \equiv (\exists x, k.\ p_1 \mapsto (x, k) \ast ls(k, p_2)) \land (p_1 = p_2 \land \text{emp})$$

Where a list is known to be non-empty, then $ls^+$, a special case of the $ls$ predicate can be used:

$$ls^+(p_1, p_2) \equiv (\exists x, k.\ p_1 \mapsto (x, k) \ast ls(k, p_2))$$

The symbolic evaluation algorithm operates on symbolic description of a memory, i.e. given a memory descriptor and a command the evaluator returns a new symbolic description corresponding to the execution of the command. Programs are annotated with a precondition of the form $H \land P$, where $H$ denotes the shape of the heap while $P$ imposes constraints on the store. Memory descriptors take the form of a triple, and are represented as follows:

$$(\exists v. (H'; S; P'))$$

Here $S$ denotes a list of equalities of the form $x = v$, where $x$ is a program variable and $v$ is a new symbolic variable. $H'$ and $P'$ are generated from $H$ and $P$ by replacing all occurrences of $x$ by the corresponding $v$ respectively. Where the algorithm gets interesting is when it encounters loops. The algorithm simply iterates the approach used for straight-line code. That is, the loop body is symbolically evaluated for a small number of iterations. At the end of each iteration a weakening, i.e. folding, step is performed on the resulting memory description. This iterative process terminations if the description of the memory converges to a fixed point. However, termination is not guaranteed. Theorem proving is used to check that a fixed point has been reached. For the list reversal
program we derived the following loop invariant using the CMU technique by-hand:

\[(\text{ls}(i, \text{nil}) \land j = \text{nil}) \lor (\text{ls}(i, \text{nil}) \ast (j \mapsto (\_ \cdot \text{nil}))) \lor (\text{ls}(i, \text{nil}) \ast \text{ls}^+(j, \text{nil}))\]  

While this does not correspond exactly to \(R\), our hand-crafted invariant, intuitively one can see that it is equivalent. We will return to this point in §7.

### 4.2 Middle-out proof planning

We now consider the proof planning perspective. Recall that the postcondition associated with the list reversal program takes the form:

\[(\exists \beta. \text{list}(\beta, j, \text{nil}) \land \alpha_{\text{init}} = \text{rev}(\beta))\]  

This asserts that if the program terminates then \(j\) will point to a list containing the sequence \(\beta\). The symbolic evaluation described above, tells us that within the loop an auxiliary list, pointed to by \(i\), is introduced. If we let \(\alpha\) denote the contents of the auxiliary list, then what remains to be discovered is the relationship between \(\alpha\) and \(\beta\). This is where middle-out reasoning is required.

We use meta-variables to speculate the existence of the auxiliary data sequence, as shown below:

\[(\exists \alpha, \beta. \text{list}(\alpha, i, \text{nil}) \ast \text{list}(\beta, j, \text{nil}) \land \alpha_{\text{init}} = F_1(\text{rev}(\beta), \alpha))\]  

Note that \(F_1\) denotes a second-order meta-variable. Verifying the loop given this schematic invariant gives rise to the following verification condition:

**Given:**

\[(\exists \alpha', \beta'. \text{list}(\alpha', i, \text{nil}) \ast \text{list}(\beta', j, \text{nil}) \land \alpha_{\text{init}} = F_1(\text{rev}(\beta'), \alpha')) \land \neg(i = \text{nil})\]

**Goal:**

\[(\exists x_2. (\exists x, y. (i \mapsto x, y) \ast ((i \mapsto x, j) \mapsto

\(\exists \alpha, \beta. \text{list}(\alpha, x_2, \text{nil}) \ast \text{list}(\beta, i, \text{nil}) \land \alpha_{\text{init}} = F_1(\text{rev}(\beta), \alpha))) \land (\exists x_1. (i \mapsto x_1, x_2)))\]

The proof planning of this conjecture can be viewed in terms of two parts. Firstly, the verification of the shape invariant. Secondly the discovery and verification of the data invariant. The proof plan, known as *rippling*, is applicable to both parts. Rippling is a rewrite strategy that uses a difference reduction criterion to select applicable rewrite rules. This difference reduction criterion is made explicit via meta-level annotations. Rippling is typically used within a functional context, however, a relational version also exists. A complete account of rippling can be found in [1, 7]. Here we require a hybrid form of rippling, i.e. functional rippling extended to deal differences arising from pointer references. To achieve this, the meta-level annotations need to be extended. We summarise the key features of functional rippling together with our proposed hybrid extension in Fig 2.
The shading above denotes meta-level annotations which highlight the difference between the goal and the given hypothesis. These annotations are known as wave-fronts. The uparrow and downward arrows are used to guarantee termination of rippling. In the case of purely functional case, a wave-front contains at least one unannotated subterm corresponding to a subterm within the given hypothesis. This is known as a wave-hole. Note that in the case of pointer rippling wave-holes do not occur. Instead we record the pointer within the given hypothesis which causes the difference. Note that existential variables denotes a potential wave-front and is represented here as a dotted box, i.e. additional wave-fronts can be generated via existential witness terms. Wave-fronts are manipulated by annotated rewrite rules are known as wave-rules. Example wave-rules are shown below:\(^1\):

\[
\text{rev}(\ldots X \ldots Y \ldots) \Rightarrow \text{app}(\text{rev}(Y), [X])
\]

(7)

\[
\text{app}(\text{app}(X, Y), Z) \Rightarrow \text{app}(X, \text{app}(Y, Z))
\]

(8)

\[
(\exists V. (\exists W. P(\text{list}(\ldots X \ldots Y)) \Rightarrow \exists W. P(\text{list}(\ldots Y V \ldots X))
\]

(9)

Note that the process of annotating goals and generating wave-rules is fully automatic. Note that the [...] around the existential \(\alpha\). This meta-level annotation, known as a sink, is used as part of the rippling-sideways strategy, in which wave-fronts are directed towards universally quantified variables within the given hypothesis. Here the notion of a sink is generalized to include existentially quantified variables.

Fig. 2. Functional and pointer rippling
Given:

\[(\exists \alpha', \beta' \ldots \alpha_{\text{init}} = F_1(\text{rev}(\beta'), \alpha')) \land \neg(i = \text{nil})\] (10)

Goal and ripple:

\[(\exists \alpha', \beta' \ldots \alpha_{\text{init}} = F_1(\text{rev}(\beta'), \alpha')) \land \neg(i = \text{nil})\]

ripple using (7)

\[(\exists \alpha', \beta' \ldots \alpha_{\text{init}} = F_1(\text{rev}(\beta'), \alpha')) \land \neg(i = \text{nil})\]

ripple using (8)

unblock using (3)

unblock using (2)

sink inward wave-front

Note that a ripple proof may require non-rippling steps, known as unblocking steps, as illustrated above. An unblocking step will involve the manipulation of a wave-front so as to enable further rippling. The class of unblocking steps must obviously be defined so as to ensure that termination of rippling is not lost. Note that as a side effect of the above ripple proof, \(F_1\) is instantiated to be \(\lambda x.\lambda y.\text{app}(x, y)\). In order to achieve a match between the rippled goal and (10), the existential variable \(\alpha'\) must be unpacked. Currently, sinks correspond to universally quantified variables. Existential sinks while represent a generalization of the current mechanism.

Fig. 3. Middle-out discovery of the data invariant via rippling
For reasons of space we will focus here on the discovery of the data invariant. In Fig 3 we show how rippling, via middle-out reasoning, can discover an appropriate instantiation for (6), i.e.

\[(\exists \alpha, \beta. \text{list}(\alpha, i, \text{nil}) \ast \text{list}(\beta, j, \text{nil}) \land \alpha_{\text{init}} = \text{app}(\text{rev}(\beta), \alpha))\]

5 Benefits of a cooperative approach

A cooperative approach to reasoning requires complementary processes which work together to achieve a common goal. As noted in [6], to be cooperative, implies that the achievements of the all over approach is greater than if the individual processes worked in isolation. With regards to our proposed approach, we have two distinct processes, i) symbolic evaluation, and ii) middle-out proof planning. The achievements of each process can be summarised as follows:

- Shape invariant is achieved via symbolic evaluation.
- Verification of the shape invariant is achieved via proof planning.
- Data invariant is achieved via middle-out reasoning.

In theory, rippling via middle-out reasoning could discover the shape invariant. However, the syntactic nature of rippling makes it better suited to discovering functional properties where the term structure is more deeply nested. So the symbolic evaluation processes mitigates for a weakness within middle-out reasoning. Likewise the correctness of the shape invariant could be dealt with as an add-on to the symbolic evaluation process, as is the case in [19]. However, proof planning provides a uniform framework in which to verify the invariant as a whole. In terms of discovering the data invariant, the evidence of our previous work suggests that middle-out reasoning and rippling are well suited to this task. It should be noted that in [19] predicate abstraction is proposed as a means of obtaining data invariants. However, we would argue that our middle-out reasoning approach is ultimately more general.

6 Related work

Closely related to the work presented in [19], is the SMALLFOOT tool [2]. SMALLFOOT is an experimental tool that supports the automatic verification of shape properties specified in separation logic. Based upon a symbolic evaluation [3], SMALLFOOT verifies a program against user supplied pre- and postconditions. Unlike [19], SMALLFOOT does not attempt to discovery loop invariants.

Out with separation logic, the Pointer Assertion Logic Engine (PALE) [20] allows relatively complex specifications to be expressed in monadic second-order logic. Verification is achieved via the MONA tool [18]. PALE requires, however, a user to supply loop invariants. In contrast, the TVLA tool [24] provides significant verification automation, while not requiring loop invariants, TVLA
may require a user to supply *instrumentation predicates*, i.e., predicates that encode local properties of datatypes. Instrumentation predicates, however, may be reused between applications.

With regards to proof planning, and in particular middle-out reasoning, the technique known as lazy thinking [4] is closely related to our proposal. Lazy thinking, like middle-out reasoning, allows the discovery process to proceed hand-in-hand with the process of verification. Lazy thinking, as applied to program synthesis, employs algorithm schemas in order to constrain search while middle-out reasoning is typically applied within the context of rippling, where the meta-level annotations provide the constraints. The work reported in [17] also makes use of second-order meta-variables to support of inductive generalization within the context of proving properties of tail-recursive functions. This is similar to our earlier work on accumulator generalization [13, 12, 14] and tail-invariant discovery [16].

7 Future work

The proposal presented in this paper represents work in progress. Achieving the proposal will involve more than just the combination of symbolic evaluation and proof planning described above. We require a theorem proving environment upon which to develop our proof plans. A strong candidate is Schirmer’s verification environment for sequential imperative programs [25]. Integrated within ISABELLE/HOL [21], Schirmer provides a generic framework for modelling and verifying imperative programs. Moreover, the framework has already been extended with separation logic, for the purposes of reasoning about pointer programs written in C [10]. Building upon ISABELLE/HOL also makes sense from the proof planning perspective given that ISAPlanner [9], the current state-of-the-art proof planner, is Isabelle based. In terms of the symbolic evaluation component, if possible, we would like to build upon either SMALLFOOT or the CMU tool. Finally, as described above, rippling as currently implemented within ISAPlanner will need to be generalized in order to deal with pointer references and existential sinks.

8 Conclusion

We propose an integration of symbolic evaluation with middle-out proof planning. We have emphasized the cooperative nature of the integration, i.e., we believe that the CMU invariant discovery technique complements the our middle-out proof planning work. The next step will involve system building which will enable us to empirically test this hypothesis.

References


version is available as Research Memo RM/00/3, Dept. of Computing and Electrical Engineering, Heriot-Watt University.


