Debugging Code Using Separation Logic

Pfalzgraf Martin,\(^1\)*

\(^1\)Department of Computer Science, Australia National University,

*To whom correspondence should be addressed; E-mail: pfalzgraf@drmartin.com

When working with complex code or with an unfamiliar code base programmers are often faced with making changes that may result in undesired changes to existing features. To foresee all of the effects of a change in a side-effecting environment is nearly impossible even for programmers who are familiar with the code base. Often to make sense of what a program is doing the programmer needs to use GDB, breakpoints, and other methods to step though their code at critical points to correct their program. Here I am investigating the use of Hoare Logic extended by Separation Logic to do symbolic execution of code as a means of debugging and analyzing code.

1 Introduction

The reason we can use logic to reason about code is due to the Curry-Howard isomorphism between code and proofs (\(I\)). This makes prefect sense since code is made up of simple instructions that carry the program from one state to the next and have some paper on the side to jot down notes as it goes. All logics have some rules we call axioms that are ”trivially” correct. That is to say each rule requires no further thought, so if you can put them together to show
something then the non-trivial conclusion must be correct. Section 2 is where I outline the 
Hoare Logic and Separation Logic rules I use.

Normally Hoare Logic and Separation Logic are used in formal methods and software veri-
fication. A common use case for using these logics is verifying that code meets some specifica-
tion. SmallFoot is as example of automated software verification by checking Separation Logic 
specifications (2). Automated verification involves symbolic execution of the program where 
each action the program takes is simulated using the logic rules. Essentially the verification 
software is running your code with variables replacing any user defined input. If your code has 
no loops this can be completely automatically (3). However, automatically generating proofs is 
a undecidable problem since there is no formula for determining loop invariants; this does not 
mean it cannot be done in many cases (4) (5). Separation Logic was developed to add the ability 
to reason about heaps (6) (7) (8) to Hoare Logic.

The combination of the Hoare Logic and Separation Logic provides a means to prove the 
outcome of a program given some context. Programs are modular by nature and by applying the 
rules of Hoare logic with separation logic we have granularity up to a single action the program 
takes with the corresponding heap constraints and context. As a result we can look at any part 
of a program and see how it fits into the context it is run. Knowing the context a statement is 
run is invaluable as a debugging tool.

By applying the logic rules it is possible to create a set of constraints that can be converted 
into first order logic formulae and sent off to a SMT solver (9). In the paper (9) their program is 
able to take a precondition and postcondition, which they call requires and ensures, and validate 
linked list structures. This has also been replicated by (10) with a subset of separation logic. 
However, first order logic is not decidable so the SMT solver may not return anything. When 
this happens it is unclear if the program is at fault or the constraints are incorrect. To deal 
with undecidability I have built an interactive solver to see if making the user solve the proof
themselves might give the user a better insight into what their code is doing. By requiring the user to verify problematic parts of their code with what they believe are the pre and post conditions should give the user a clear understanding of what their code is doing contrary to their intention.

2 Rules

The Hoare Logic Rules are as follows:

- \( P, Q \) and \( R \) are the precondition and postcondition
- \( w \) and \( s \) stand for weaker and stronger predicate
- \( S \) is the sequent which is made up of these rules
- \( \Rightarrow \) is just implication

Precondition Strengthening

This rule and its successor are important to connect the axioms from Separation Logic with the tree generated by Hoare Logic.

\[
P_s \Rightarrow P_w \quad \{P_w\} S \{Q\}
\]

\[
\{P_s\} S \{Q\}
\]

Postcondition Weakening

\[
\{P\} S \{Q_s\} \quad Q_s \Rightarrow Q_w
\]

\[
\{P\} S \{Q_w\}
\]

Sequence

To connect two sequents you must have the postcondition of the first and precondition of the second the same
\[
\begin{align*}
\{P\} & S_1 \{Q\} & \{Q\} & S_2 \{R\} \\
\{P\} & S_1; S_2 \{R\}
\end{align*}
\]

**Conditional**

\[
\begin{align*}
\{P \land b\} & S_1 \{Q\} & \{P \land \neg b\} & S_2 \{Q\} \\
\{P\} & \text{if } b \text{ then } S_1 \text{ else } S_2 \{Q\}
\end{align*}
\]

**While Loop**

\[
\begin{align*}
\{P \land b\} & S \{P\} \\
\{P\} & \text{while } b \text{ do } S \{P \land \neg b\}
\end{align*}
\]
The Separation Logic Rules are as follows:

- $e(v/x)$ means replace all occurrences of $x$ in the equation $e$ with $v$
- $e \mapsto x$ means $e$ maps to $x$ in the heap, where $e$ is some expression and $x$ is some value
- $emp$ stands for empty heap
- $x := e$ stands for the variable $x$ is assigned the expression $e$
- the name after the : is the abbreviation I use in my proof

**Floyd Store Axiom: FloStrAxm**

This replaces the Hoare Logic store Axiom that I did not list. $v$ here is an auxiliary variable which does not occur in $e$.

$$\{x = v \land emp\} \ x := e \{x = e(v/x) \land emp\}$$

**Derived Floyd Store Axiom: DerFloStrAxm**

where $x$ does not occur in $e$

$$\{emp\} \ x := e \{x = e \land \emp\}$$

**Fetch Assignment Axiom: FetAssAxm**

where $v_1$ and $v_2$ are auxiliary variables which do not occur in $e$

$$\{x = v_1 \land e \mapsto v_2\} \ x := [e] \{x = v_2 \land (e(v_1/x) \mapsto v_2)\}$$

**Derived Fetch Assignment Axiom: DerFetAssAxm**

where $v_2$ and $x$ do not occur in $e$

$$\{x = v_2\} \ x := [e] \{(x = v_2) \land (e \mapsto v_2)\}$$
Heap Assignment Axiom: HepAssAxm

where \((e \mapsto -)\) abbreviates \((\exists z. e \mapsto z)\) and \(z\) does not occur in \(e\)

\[\{e \mapsto -\}[e] := e_1 \{e \mapsto e_1\}\]

Allocation Assignment Axiom: AllAssAxm

where \(v\) is an auxiliary variable different from \(x\) and not appearing in \(e_1, e_2, ..., e_n\)

\[\{x = v \land emp\} x := cons(e_1, e_2, ..., e_n) \{x \mapsto e_1(v/x), e_2(v/x), ..., e_n(v/x)\}\]

Derived Allocation Assignment Axiom: DerAllAssAxm

where \(x\) does not appear in \(e_1, e_2, ..., e_n\)

\[\{emp\} x := cons(e_1, e_2, ..., e_n) \{x \mapsto e_1, e_2, ..., e_n\}\]

Dispose Axiom: DisAxm

where \((e \mapsto -)\) means \(e\) is a valid address

\[\{e \mapsto -\\} dispose(e) \{emp\}\]

The Frame Rule: Frame Rule

where no variable modified by \(S\) appears free in \(R\)

\[\frac{\{P\} S \{Q\} \quad \{P \ast R\} S \{Q \ast R\}}{\{P \ast R\} S \{Q \ast R\}}\]

The rules I use are from the Comp2600 course at ANU.
3 Results

The interactive solver was able to show how conditions flowed through the code and showed promise for being a simple proof assistant. The proof assistant can deal with large proofs and applies the rules smartly. So the user does not need to specify every subsequent line, rather they need only say what rule to use and the assistant will do the rest. Since I can zoom in on any part of a proof the user only has to deal with solving that section while the program takes care of how the changes affect the rest of the proof. Finally, I use GHCI as an interface, the interpreter for the Glasgow Haskell Compiler, so the user has the full power of the Haskell language and nice auto completion.

3.1 Example Proof

see linked: online example.

3.2 Speed of code

The code is optimized for modifying the sequents and predicates. However, as a result the slowest operation is printing to the screen. The reason for this is each sequent only contains its own predicate, operation and rule with tags to the next sequent. In this sense it is a lot like a linked list where the only way to view the proof is to traverse the list and look up what each tag corresponds to. I store the sequents in a map of key value pairs where the key is their tag and the value is the sequent. As a result I get $O(\log n)$, where $n$ is the number of pairs in the map, to modify any sequent. I have the same complexity for the predicates, written in my assistant as "Qn" where $n$ is a number. Printing, on the other hand, is $O(n \log n)$, where $n$ is the number of sequents being printed, to print the entire proof. The printing is actually closer to $O(n^2 \log n)$ since for any sequent I may also need to lookup the predicate and rules can contain up to two tags. Regardless this is of little concern since there isn’t any point in printing more than you can
read to the screen. If instead the entire proof had to be traversed to update a field, the program could not operate on large proofs.

4 Discussion

The assistant would benefit from integrating with an SMT solver. This was not the point of the project since I wanted to see what it would be like to work through the code with an aid that did the grunt work. By not automating the steps the user is forced to see exactly what their code is doing and deal with any contradictions they create and discover any misconceptions of what their code is doing. I’m not sure automating, or giving the option of automating this would benefit the user in this regard. Rather I’m thinking the parts of the code the user is not concerned about should be automated.

Its a delicate decision, since complex code could have any number of issues that plague it. Lets say the user writes the precondition and postcondition such that they do not prohibit the behavior that is creating a bug in their code. If the user gets back that their code matches their conditions, they would have completely overlooked the issue in their code. While if they have to solve it themselves I think users would most likely see what is happening. It is a sanity check that should be easy to use. There are already enough solvers that verify code based specifications like Key-C (11) and ESC/Java (12), whose purpose is not to help the user debug their code. There is the KeY project that integrates with Eclipse, updates proofs as code is changed (13) and gives an interactive experience with a Symbolic Execution Debugger (14). Which is on the right track of what I am interested in accomplishing. Ideally the system would require less user input.

To reduce user input I could be automate more of the assistant. This would require an in-depth investigation into what interface would be ideal since the interface would dictate what should be automated and how it should be automated. The proof of concept I built is enough to
show what is needed to have a real interactive proof assistant to help users work through their code. I would prefer to make a plug-in for emacs that allowed the user to click on parts of the proof they wanted to investigate as well as use some of emacs plug-ins like its undo tree. This would make tracking changes made to large proof much more manageable and would leverage the modularity of the proofs. Not to mention integrating this with emacs lets the program interface with other language specific plug-ins. I envision this working side by side with your code where you can open blocks of code in the proof assistant to work with them. Since I have written the proof assistant as a library it should be very simple to turn it into a plug-in for any text editor.

### 4.1 Verifying The Proofs

There is a certified verifier for a fragment of separation logic (15) that I could export my proof to and thus ensure correctness. It would be better to integrate it into my program, although meshing OCaml with Haskell is less than ideal. Regardless, I can send each part of the proof individually to the certified verifier to let the user know if their proof is correct.

### 4.2 Choice of Language

I used Haskell to build the tool largely due to my prior experience writing a compiler in Haskell. When I first thought about writing the proof assistant I realized that I was effectively writing a compiler from code to the Hoare Logic proof structure. This makes any functional language more attractive since it is quite a bit easier to write a compiler in a functional language. The obvious alternative to Haskell is OCaml, since a large number of verification software is written in OCaml. However, the purpose of this project is to build a proof of concept, meaning writing in a more familiar language makes more sense.
References and Notes


INDEPENDENT STUDY CONTRACT

Note: Enrolment is subject to approval by the projects co-ordinator

SECTION A (Students and Supervisors)

UniID: u5214945
SURNAME: Martin
FIRST NAMES: Pfalzgraf
PROJECT SUPERVISOR (may be external): Rajeev Gore, Zhe Hou
COURSE SUPERVISOR (a RSCS academic): Weifa Liang
COURSE CODE, TITLE AND UNIT: Comp3710 Topics in Computer Science Project

SEMESTER X S1 □ S2 YEAR: 2015

PROJECT TITLE: A Theorem Prover for the Specification Language of Separation Logic

LEARNING OBJECTIVES:
The student will have to master the following:
- separation logic
- ternary semantics of substructural logics
- functional programming
- formal verification
- labelled sequent calculi
- backward proof search using labelled sequent calculi
- lexical analysis and parsing
- compiler construction
PROJECT DESCRIPTION:

Separation Logic is an extension of Hoare Logic which allows us to reason about computer programs with mutable memory and pointers. It consists of an assertional substructural logic which extends classical first-order logic with three extra connectives, and a specification logic which consists of a finite collection of axioms and inference rules in a style similar to Hoare Logic. The validity problem for the assertion language is not recursively enumerable, hence a finite, sound and complete proof calculus for separation logic cannot exist.

Nevertheless, Separation Logic is already being used in industry, most notably in the London research and development laboratory of Facebook. Since the validity problem is undecidable, most tools concentrate on decidable fragments so these tools cannot deal with a wide range of cases that arise in the verification of real programs. Thus there is a need for a tool that handles the full complement of logical connectives from Separation Logic, ideally in a fully automatic way.

The project is to build a theorem prover that accepts a Separation Logic triple containing pre- and post-conditions built from the full set of separation logic connectives, and to return a proof of that Hoare triple, or return a counter-model which makes the triple false. There is no hope of obtaining a terminating and complete prover, so the aim is to design and build a prover that is practical in real-world examples.

The choice of programming language is up to the student: probably Haskell or Ocaml.

Our initial thoughts are that the prover would extend Zhe Hou’s theorem prover for the full assertion language of separation logic (without Hoare triples) to handle the specification logic of Separation Logic with Hoare triples. But the undecidability of the underlying validity problem also indicates that a more traditional interactive approach using LCF-style theorem provers such as HOL, Isabelle or Coq may be appropriate. These issues will be sorted out during the course of the project mainly because LCF-style theorem provers themselves often require 3 months of study to master them and this may not be appropriate for Pfalzgraf.

Zhe Hou is currently completing his PhD on this topic and Rajeev Gore has ample experience in substructural logics. Successful completion of this project will also require the student to build a tool that aims to automate the task for proving Hoare triples utilising separation logic.

Duration: currently this is set as a one-semester Topics in Computer Science project with expectations of 10-12 hours per week of student time. However, there is an opportunity to extend this project into second semester since the project topic is broad enough to allow this. Decision to be made jointly by Raj and Pfalzgraf.
ASSESSMENT (as per course’s project rules web page, with the differences noted below):

<table>
<thead>
<tr>
<th>Assessed project components:</th>
<th>% of mark</th>
<th>Due date</th>
<th>Evaluated by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report: name style: __________________________</td>
<td>50%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(e.g. research report, software description...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Artefact: name kind: __________________________</td>
<td>40%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(e.g. software, user interface, robot...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Presentation:</td>
<td>10%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MEETING DATES (IF KNOWN):

STUDENT DECLARATION: I agree to fulfil the above defined contract:

Signature .......................................................... 2/03/2015
Date

SECTION B (Supervisor):

I am willing to supervise and support this project. I have checked the student's academic record and believe this student can complete the project.

See Attached Email ........................................ 2/03/2015
Signature .......................................................... Date

REQUIRED DEPARTMENT RESOURCES:

SECTION C (Course coordinator approval)

Signature .......................................................... 3/3/15
Date

SECTION D (Projects coordinator approval)

Signature .......................................................... Date

Research School of Computer Science

Form updated Jun-12