Program Verification with Separation Logic and Rely Guarantee

Victoria University of Wellington

Allan Martinez Tabilog

Thesis submitted in fulfillment of the requirements for the degree of Master of Science in Computer Science
2017

Abstract

This thesis explores two kinds of program logics that have become important for modern program verification - separation logic, for reasoning about programs that use pointers to build mutable data structures, and rely guarantee reasoning, for reasoning about shared variable concurrent programs. We look more closely into the motivations for merging these two kinds of logics into a single formalism that exploits the benefits of both approaches - local, modular, and explicit reasoning about interference between threads in a shared memory concurrent program. We discuss in detail two such formalisms - RGSep and Local Rely Guarantee (LRG), in particular we analyse how each formalism models program state and treats the distinction between global state (shared by all threads) and local state (private to a given thread) and how each logic models actions performed by threads on shared state, and look into the proof rules specifically for reasoning about atomic blocks of code. We present full examples of proofs in each logic and discuss their differences. This thesis also illustrates how a weakest precondition semantics for separation logic can be used to carry out calculational proofs. We also note how in essence these proofs are data abstraction proofs showing that a data structure implements some abstract data type, and relate this idea to a classic data abstraction technique by Hoare. Finally, as part of the thesis we also present a survey of tools that are currently available for doing manual or semi-automated proofs as well as program analyses with separation logic and rely guarantee.

Contents

Chapter 1

Introduction

Deductive program verification is a branch of formal methods that aims to demonstrate the correctness of computer programs mathematically, by axiomatising reasoning about programs within a formal logic and then constructing formal proofs. One of the first such axiomatisations, known as Hoare logic, was invented by C.A.R Hoare in 1969 [15]. Hoare logic is a logic for proving that an imperative program satisfies its specification. Over the past decades there has been great progress in this area, including various extensions of Hoare logic for different kinds of programming languages, as well as tool support for these formalisms. Two major developments involve a formalism known as separation logic and a formalism called rely-guarantee reasoning, which are the main focus of this thesis.

Separation logic is an extension of Hoare logic for reasoning about programs that manipulate mutable data structures. It extends Hoare logic with assertions that can talk about disjoint resources and supports "local reasoning", i.e. where proofs only mention those resources that are touched by a program (its memory footprint), instead of having to specify the global state of the system. The logic provides connectives such as the "separating conjunction" to precisely express the disjointness of program states and hence facilitate reasoning about e.g. the absence of pointer aliasing. There have been many developments with this logic in the recent years, including extensions to concurrent programs and object-oriented programs, and various techniques and tools for automatic program analyses.

One weakness of separation logic, specifically when dealing with concurrent programs, is that it relies on the use of invariants to describe constraints on shared state. This leads to the heavy use of auxiliary variables, as shown for example, in a published proof of a non-blocking stack[1] using concurrent separation logic

[1]The term "non-blocking" encompasses a range of guarantees that can be made regarding the progress of concurrent programs: *lock-free* ensures that the delay of a thread does not

[32]. There is another well-known formalism in the program verification literature that can be used to naturally describe interference on shared state, i.e. "rely guarantee reasoning". Created by Cliff Jones in 1983 [21, 22], rely guarantee is a method that explicitly takes into account the interference between a program and its environment (other programs) that arise especially in the context of shared-variable concurrent programming. In a rely-guarantee specification of a concurrent program, the "rely" condition is a relation that summarises the interference that a program can tolerate from its environment and the"guarantee" condition is a relation that summarises the interference that a program can produce on its environment.

Given these formalisms it was naturally considered by researchers that a formalism merging the strengths of separation logic and rely guarantee would be of great benefit to the verification of concurrent programs. A seminal work on this topic is the formalism called RGSep which was constructed by Viktor Vafeiadis and Matthew Parkinson in the paper "A Marriage of Rely Guarantee and Separation Logic" [43]. In the RGSep formalism, program state is split into two parts – shared state and local state – which are disjoint, although it is possible to turn one into the other. RGSep uses "actions" to describe atomic updates to the shared state and encodes rely and guarantee conditions as sets of actions. In his PhD thesis, Vafeiadis demonstrates the use of RGSep in the verification of various concurrent data structures, including some which use non-blocking synchronisation.

Another formalism which merges separation logic and rely guarantee is Feng's local rely guarantee [13]. Local rely guarantee lifts the separating conjunction operator of sequential separation logic from being an operator on states to become an operator on state transitions (i.e. actions), and lifts this further to rely and guarantee conditions, which are just sets of actions. Feng's formalism also makes it possible to reason formally about "sub-transitions", which are actions operating over disjoint resources; it also weakens the requirement (from the original rely guarantee formalisms) that all resources should be known globally and allows hiding of resources from the environment.

Given all this context, in this research project we aim to compare RGSep and LRG, understand their motivations for merging separation logic and rely guarantee, and compare their similarities and differences. We also aim to construct

cause delays in other threads, *wait-free* means that every thread completes execution in a finite number of steps, and *obstruction-free* means that when executing in isolation, a thread eventually finishes in a finite number of steps example proofs within each formalism and also find a way to carry out proofs in a more calculational way using weakest precondition semantics. As part of this project we also want to do a survey of tools that are available for doing proofs or program analyses with separation logic and rely guarantee.

The rest of the thesis is structured as follows: in chapter 2 we discuss the basics of separation logic and rely guarantee, as well as weakest precondition and strongest postcondition semantics and techniques for generating verification conditions; in chapter 3 we discuss logics that combine rely guarantee and separation logic; in chapter 4 we illustrate the use of weakest precondition semantics in separation logic proofs and also propose a way of enriching specifications by extending the assertion language with elements from Hoare's data abstraction technique. We conclude in chapter 5 and finally we include an appendix in which we survey some useful tools for doing proofs or program analyses in separation logic or rely guarantee.

Chapter 2

Background

In this chapter we discuss the key ideas and formalisms of separation logic, concurrent separation logic, and rely guarantee. We also discuss a weakest precondition semantics for separation logic. The aim of this chapter is to provide enough detail on these formalisms that we can use and build on in subsequent chapters.

2.1 Separation Logic

Separation logic is an extension of Hoare logic for reasoning about pointer programs, i.e., programs that access and mutate data structures via pointers. In this section we present the basics of separation logic. We consider a simple imperative programming language with commands that support pointers, whose syntax is given in figure 2.1.

This language is built up of the skip command, sequential composition, assignment, conditional, iteration, record creation, field lookup, field update, and record deletion. A record ρ just a set of fields and their corresponding values; formally, it has the syntax $[f_1 : E_1, \dots f_n : E_n]$ where the f_k are from a collection of field names and the E_k are expressions denotaing values, formally specified below. Execution of the statement `r:= new(`ρ`)` creates a new instance of the record ρ on the heap, pointed to by the variable r, and executing `delete(`r`)` deletes the record pointed to by r. Lookup of the value of a field f_k of r is done by executing the command $x := f_k(r)$, which retrieves that value and stores it in the variable x. Updating the value of a field f_k of r is done by executing the statement $f_k(r) := E$ where E is some expression.

Let us define a formal semantics for this language. First we need to define

$$\begin{array}{l} \mathrm{C} ::= \texttt{skip} \mid \texttt{x := E} \mid \mathrm{C};\mathrm{C} \mid \texttt{if}\ \mathrm{b}\ \texttt{then}\ \mathrm{C}\ \texttt{else}\ \ \mathrm{C} \mid \texttt{while}\ \mathrm{b}\ \texttt{do}\ \mathrm{c} \mid \\ \qquad \texttt{r := new (}f_1 : E_1, \ldots f_n : E_n\texttt{)} \mid \texttt{x := f}_{\texttt{k}}\texttt{(r)} \mid \texttt{f}_{\texttt{k}}\texttt{(r)} := \texttt{E} \mid \texttt{delete (r)} \end{array}$$

Figure 2.1: Syntax of a simple imperative language

our semantic domains: values, stores, and heaps:

$$Val = Int \cup Loc \cup Atom \tag{2.1.1}$$

$$Store = Var \rightharpoonup_{fin} Val \tag{2.1.2}$$

$$Heap = Loc \rightharpoonup_{fin} Val \tag{2.1.3}$$

The set Val of values is the union of three other sets: Int for integers, Loc for locations, and $Atom$ for atoms. The last set contains constant symbols and here the only atom of interest is nil, which represents the null pointer.

Stores and heaps are used to model program state. A store represents a program stack and is modelled as a finite partial function from the set of variables (identifiers) to values. A heap represents a program heap, i.e., the storage for dynamically allocated objects, and is modelled as a finite partial function from locations to values. In our programming model, the objects that reside in the heap are the record structures that we have defined above.

The semantics discussed in section 2.1.1 is a small-step operational semantics. This captures the notion of a program as the execution of a sequence of atomic steps. We choose this semantic style because later we shall be interested in reasoning about concurrency, in which programs are modelled as interleaving of atomic steps of multiple processes. The meaning of `skip`, sequential composition, conditional, and iteration are standard. The pointer-accessing commands require more explanation.

2.1.1 Small-step operational semantics of the programming language

The semantics is defined as a relation $\longrightarrow$ between configurations, of which there are two kinds: non-terminal configurations $C, s, h \in Command \times Store \times Heap$ and terminal configurations $s, h \in Store \times Heap$. This is defined in figure 2.2. The rules for the imperative commands are standard, see e.g. [45], and the rules for the heap-accessing commands are adapted from [18]. Explanation follows:

- the `skip` command does nothing, thereby leaving the program state unchanged
- the assignment statement `x:=E` updates the store such that the variable x has the value $[\![E]\!]s$ of the expression E in state s. (Note: we haven't

$$\texttt{skip}, s, h \longrightarrow s, h$$

$$x := E, s, h \longrightarrow s[x \mapsto [\![E]\!]s], h$$

$$\frac{c_1, s, h \longrightarrow c_1', s', h'}{c_1; c_2, s, h \longrightarrow c_1'; c_2, s', h'}$$

$$\frac{[\![b]\!]s}{\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2, s, h \longrightarrow c_1, s, h}$$

$$\frac{\neg[\![b]\!]s}{\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2, s, h \longrightarrow c_2, s, h}$$

$$\texttt{while } b \texttt{ do } c, s, h \longrightarrow \texttt{if } b \texttt{ then } c\texttt{; while } b \texttt{ do } c \texttt{ else skip}, s, h$$

$$\frac{\begin{array}{c} l \in \mathit{Loc} \quad l \notin \mathit{dom}(h) \\ [\![E_1]\!]s = v_1 \quad [\![E_2]\!]s = v_2 \quad \ldots \quad [\![E_n]\!]s = v_n \end{array}}{r \texttt{ := new (}f_1 : E_1, f_2 : E_2, \ldots, f_n : E_n\texttt{)}, s, h \longrightarrow s[r \mapsto l], h[l \mapsto v_1, v_2, \ldots v_n]}$$

$$\frac{[\![E]\!]s = l \in \mathit{Loc} \qquad l \in \mathit{dom}(h) \qquad h(l) = r}{x := f_i(E), s, h \longrightarrow s[x \mapsto \pi_i(r)], h}$$

$$\frac{[\![E]\!]s = l \in \mathit{Loc} \qquad l \in \mathit{dom}(h) \qquad h(l) = r \qquad [\![E']\!]s = v'}{f_i(E) := E', s, h \longrightarrow s, h[l \mapsto r[i \mapsto v']]}$$

$$\frac{l \in \mathit{Loc} \qquad l \in \mathit{dom}(h) \qquad [\![E]\!]s = l}{\texttt{delete(}E\texttt{)}, s, h \longrightarrow s, (h - l)}$$

Figure 2.2: Small-step semantics of commands

specified the syntax and semantics for arithmetic and boolean expressions yet, but this should be standard. For now it suffices to think of such expressions as functions from stores to values, hence the expression $[\![E]\!]s$ denotes a value).

- the sequential composition $c_1; c_2$ of two commands c_1 and c_2 executes by first executing c_1, thereby updating the state and transforming c_1 into c_1', and then executing $c_1'; c_2$ in the updated state

- there are two rules for the conditional command `if b then` c_1 `else` c_2. If the guard b holds in the state s then the command c_1 executes from the original state; otherwise c_2 executes. (Note: we haven't defined boolean expressions and their semantics yet, but that should be standard. Also, we assume that the expression of the guard does not mutate the initial state of the conditional command).

- the iteration command "unwinds" into the equivalent conditional command; it executes until the guard b fails to hold.

The next definitions describe how heap-accessing commands modify the program state:

- Record creation: the command `new` $(f_1 : E_1, f_2 : E_2, \ldots, f_n : E_n)$ creates a new record object in the heap with its fields $f_1, \ldots, f_n$ intialised to the values of the expressions $E_1, \ldots, E_n$. Here the location l of the new record must not already be an allocated address (i.e. it should not already be in the domain of the current heap). For now we assume that execution of *new* will always succeed, i.e. it will always return a previously unallocated address.

- Field lookup: we define the projection operator π_i for $i = 1, 2, \ldots, n$ such that $\pi_i r$ evaluates to the the value of the ith field of the record r. Then the command `x :=` $f_i(E)$ first evalutes the expression E; if that evaluates to the address l of a record r in the current heap, it then stores the value of the ith field of r in the variable x.

- Field update: provided that the expression E evaluates to the address l of a record r in the current heap, updating the ith field of E to the value of E' results in a heap where l maps to the updated record r with field i having new value v'.

- Record deletion: provided that E evaluates to an address l in the domain of the current heap, executing `delete(E)` results in an updated heap where

$$\boxed{\begin{array}{lll} \mathrm{P, Q, R} \quad ::= & \mathrm{B} \mid \mathrm{false} \mid \mathrm{P} \Rightarrow \mathrm{Q} \mid \forall\, \mathrm{x} \cdot \mathrm{P} & \text{Classical logic} \\ & \mid \mathrm{emp} \mid \mathrm{E} \mapsto \mathrm{F} \mid \mathrm{P} * \mathrm{Q} \mid \mathrm{P} \mathrel{-\!\!*} \mathrm{Q} & \text{Spatial assertions} \end{array}}$$

Figure 2.3: Assertion language of separation logic

the location l is removed. The updated heap is denoted by the expression $h - l$, which is just like the heap h except that $dom(h - l) = dom(h) \setminus \{l\}$

For the last three commands above, the program goes into an error state whenever the expression E does not evaluate to an address of a record in the current heap. In this case we also say that the configuration C, s, h is "stuck" when C causes a memory fault, i.e. references an expression that denotes a memory location that is not in the domain of h. Formally, the configuration C, s, h is said to be stuck when there is no configuration K such that $C, s, h \longrightarrow K$. On the other hand we say that C, s, h is *safe* when there is a configuration K such that $C, s, h \overset{*}{\longrightarrow} K$ where K is a terminal configuration, or when C, s, h is not stuck.

2.1.2 Assertion language of separation logic

Next we shall consider the assertion language of separation logic. This is summarised in figure 2.3. There are two kinds of assertions: classical predicate logic , which are already used in Hoare logic, including standard boolean expressions B (variables $P, Q, \ldots$, as well as $\neg P$, $P \vee Q$, $P \wedge Q$, $P \longrightarrow Q$); and "spatial assertions" for talking about the program heap:

- *emp* – an assertion that is true of an empty heap
- $E \mapsto F$ – the "points-to" assertion, denoting a singleton heap, where the expression E, evaluating to a location, points to the value denoted by expression F
- $P * Q$ – "separating conjunction", asserting that the current heap can be split into two disjoint parts which separately satisfy the assertions P and Q
- $P \mathrel{-\!\!*} Q$ – "separating implication", asserting that if the current heap is extended with a heap that satisfies P then the resulting extension satisfies Q.

We give a formal semantics for this assertion language in terms of a satisfaction relation $s, h \models P$, which says that the assertion P is true (or holds) in a given store s and heap h. We are assuming here that the domain of the store s contains all the free variables of the assertion P. Also, given two heaps h and h', $h \# h'$

denotes that h and h' are domain-disjoint, and $h * h'$ denotes the union of the two domain-disjoint heaps.

$$s, h \models B \quad \text{iff } [\![B]\!]s = true \tag{2.1.4}$$

$$s, h \models E \mapsto F \quad \text{iff } dom(h) = \{[\![E]\!]s\} \text{ and } h([\![E]\!]s) = [\![F]\!]s \tag{2.1.5}$$

$$s, h \models false \quad \text{iff never} \tag{2.1.6}$$

$$s, h \models P \Rightarrow Q \quad \text{iff if } s, h \models P \text{ then } s, h \models Q \tag{2.1.7}$$

$$s, h \models \forall x \cdot P \quad \text{iff } \forall v \in Val \cdot s[x \mapsto v], h \models P \tag{2.1.8}$$

$$s, h \models emp \quad \text{iff } h = [] \text{ i.e. the empty heap} \tag{2.1.9}$$

$$s, h \models P * Q \quad \text{iff } \exists h_0, h_1 \cdot h_0 \# h_1 \text{ and } h_0 * h_1 = h \text{ and } s, h_0 \models P \text{ and } s, h_1 \models Q \tag{2.1.10}$$

$$s, h \models P -\!\!* Q \quad \text{iff } \forall h' \cdot \text{ if } h' \# h \text{ and } s, h' \models P \text{ then } s, h * h' \models Q \tag{2.1.11}$$

2.1.3 Some properties of separation logic connectives

We present some properties of the connectives of separation logic that will be useful in proofs. First we define a semantic consequence relation between formulae of separation logic. The relation $P \models Q$ holds between formulae P and Q iff for all s, h, if $s, h \models P$ then $s, h \models Q$. We are assuming here that $dom(s) \supset free(P) \cup free(Q)$. The following are some useful properties:

- $P * emp = emp * P = P$ (emp as the unit of $*$)
- $P * (Q * R) = (P * Q) * R$ (associativity of ast)
- $P * Q = Q * P$ (commutativity of $*$)
- If $P' \models P$ and $Q' \models Q$ then $P' * Q' \models P * Q$
- If $R * P \models Q$ then $R \models P -\!\!* Q$
- If $R \models P -\!\!* Q$ and $R' \models P$ then $R * R' \models Q$

2.1.4 Separation logic specifications and inference rules

The notion of a program specification in separation logic is similar to the Hoare triple. If C is a program that is constructed as per the syntax defined above, and P and Q are separation logic assertions, then a separation logic triple has the form $\{P\}\ C\ \{Q\}$. We formally define a semantics for triples as follows: we say that $\{P\}\ C\ \{Q\}$ holds when, for all states s, h, if $s, h \models P$ then the configuration C, s, h does not get stuck, i.e. abort or cause memory faults (as given in section 2.1.1), and if $C, s, h \xrightarrow{*} (s', h')$ for some terminal configuration s', h', it follows

that $s', h' \models Q$.
The inference rules are classified into "small axioms" and "structural rules". The small axioms are so-called because each axiom mentions only the area of heap that is accessed by a command. One axiom is given for each command. Structural rules are inference rules that do not involve program constructs, but deal with triples as whole. The small axioms are:

$$\{emp\}\ \texttt{r:=new(}f_1 : E_1, \ldots, f_n : E_n\texttt{)}\ \{r \mapsto [f_1 : E_1, \ldots, f_1 : E_n]\} \tag{2.1.12}$$

$$\begin{aligned}&\{E \mapsto [f_1 : E_1, \ldots, f_i : E_i, \ldots f_n : E_n] \wedge [\![E_i]\!]s = v\}\\ &\texttt{x := } f_i(E)\\ &\{x = v \wedge E \mapsto [f_1 : E_1, \ldots, f_i : E_i, \ldots f_n : E_n]\}\end{aligned} \tag{2.1.13}$$

$$\begin{aligned}&\{E \mapsto [f_1 : E_1, \ldots, f_i : E_i, \ldots f_n : E_n]\}\\ &f_i(E) := E'\\ &\{E \mapsto [f_1 : E_1, \ldots, f_i : E', \ldots f_n : E_n]\}\end{aligned} \tag{2.1.14}$$

$$\{E \mapsto -\}\ \texttt{delete(E)}\ \{emp\} \tag{2.1.15}$$

The structural rules are:

- Frame Rule

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}} \qquad \text{where } fv(R) \cap mod(C) = \varnothing$$

- Rule of Consequence

$$\frac{P' \Rightarrow P \qquad \{P\}\ C\ \{Q\} \qquad Q \Rightarrow Q'}{\{P'\}\ C\ \{Q'\}}$$

The rule of consequence is taken from Hoare logic. The frame rule is unique to separation logic and it formalises the idea of local reasoning. Intuitively the rule says that if a command modifies a portion of state, it leaves any other disjoint portion unmodified. The side condition of the rule requires that the command does not modify any variable that the other disjoint portion talks about.

2.2 Concurrent separation logic

Separation logic has been extended to concurrent separation logic (CSL) by Peter O'Hearn and Stephen Brookes [8, 30] to support reasoning about programs that access shared resources. For this the simple imperative language given in figure 2.1 is extended with constructs for concurrent programming – see figure 2.5.The new constructs are:

- Resource declaration: The basic computational model used by CSL consists of multiple processes sharing a set of resources (shared data structures). To support this the programming language allows the declaration of resources which have a name r and an associated set of protected variables. There are a few well-formedness constraints for resource declarations (carried over from Owicki-Gries [31]): a variable is said to *belong* to a resource r if it is in the associated variable list in a resource declaration. The following requirements must hold:

  1. A variable belongs to at most one resource
  2. If a variable x belongs to a resource r, it cannot appear in a parallel process except when inside a critical region for r
  3. If a variable x is modified in one process, it cannot appear in another unless it belongs to a resource.

- Conditional critical region (CCR): Access by processes to shared resources is controlled via the CCR, which has the form

  ```
  with r when b do C endwith
  ```

  Here, r is the name of a resource, b is a heap-independent boolean expression and C is a command. A process that executes a CCR tries to acquire the resource r and then evaluates the boolean guard b. If b holds, it proceeds to execute the command C. Otherwise, it releases the resource and tries again. As noted in [30], a CCR is a unit of mutual exclusion; two `with` commands for the same resource cannot be executed simultaneously. The execution of a CCR for a resource r can only proceed if no other region for r is currently executing and the boolean guard holds; othewise a process must wait until the conditions for it to proceed are fulfilled.

- Parallel composition: The construct

  $$\mathtt{C_1} \parallel \ldots \parallel \mathtt{C_n}$$

  denotes interleaved execution of the commands C_1 through C_n.

$init$;
`resource` $\texttt{r}_1(variable\ \ list), \ldots, \texttt{r}_\texttt{m}(variable\ \ list)$;
$C_1 \parallel \ldots \parallel C_n$

Figure 2.4: Format of a CSL program

C ::= `skip` | `x := E` | C;C | `if` b `then` C `else` C | `while` b `do` c |
    `r := new` $(f_1 : E_1, \ldots f_n : E_n)$ | `x :=` $\texttt{f}_\texttt{k}(\texttt{r})$ | $\texttt{f}_\texttt{k}(\texttt{r})$:= `E` | `delete (r)`
    `resource` $\texttt{r}_1(variable\ \ list), \ldots, \texttt{r}_\texttt{m}(variable\ \ list)$ |
    `with r when b do C` |
    $\texttt{C}_1 \parallel \ldots \parallel \texttt{C}_\texttt{n}$

Figure 2.5: Syntax of a simple imperative language with shared variable concurrency

In CSL as presented in [30], every program must be written in a certain restricted form shown in figure 2.4. Here, *init* stands for an intialisation sequence, whose effect is to establish the invariants maintained by the shared resources. The `resource` statement declares a set of shared resources with names r_k and associated lists of protected variables. Finally, the main body of a CSL program is a parallel composition $C_1 \parallel \ldots \parallel C_n$ of n commands.

Next we introduce the proof rules for these programming constructs. To reason about a program of the form given in figure 2.4, we must first specify a formula called the *resource invariant* RI_{r_i} for each resource r_i. These formula must satisfy the following constraints:

1. Any command `x :=` ... that updates a variable x which is free in RI_{r_i} must occur within a critical region for r_i.

2. Every resource invariant must be a *precise* assertion. This is a technical term in separation logic: an assertion P is said to be precise if for all stores s and heaps h, there is at most one heap h' that is a subheap of h such that $s, h' \models P$.

- The proof rule in CSL for a concurrent program is as follows:

$$\frac{\{P\}\ init\ \{RI_{r_1} * \ldots * RI_{r_m} * P'\}\quad \{P'\}\ C_1 \parallel \ldots \parallel C_n\ \{Q\}}{\begin{array}{l}\{P\}\\ init;\\ \texttt{resource}\ \texttt{r}_1(variable\ list), \ldots, \texttt{r}_\texttt{m}(variable\ list)\ ;\\ C_1 \parallel \ldots \parallel C_n\\ \{RI_{r_1} * \ldots * RI_{r_n} * Q\}\end{array}}$$

  The entire program has precondition P and the postcondition has two parts: $RI_{r_1} * \ldots * RI_{r_n}$, which is the conjunction of all the resource invariants, re-established on termination, and Q, which is the overall postcondition established by the program on termination. In order to prove that the CSL program satisfies its specification, it suffices to prove two things:

  1. The initialisation sequence separately establishes the conjunction of the resource invariants of all resources as well as some condition P', which can be thought of as an additional portion of state that is given to the parallel processes for access outside of critical regions; and
  2. The parallel composition satisfies its specification $\{P'\}\ C_1 \parallel \ldots \parallel C_n\ \{Q\}$. This is shown by applying the following rule for parallel composition:

     $$\frac{\{P_1\}\ C_1\ \{Q_1\} \ldots \{P_n\}\ C_n\ \{Q_n\}}{\{P_1 * \ldots * P_n\}\ C_1 \parallel \ldots \parallel C_n\ \{Q_1 * \ldots * Q_n\}}$$

     together with the rule of consequence, generating two proof obligations: $P' \Rightarrow P_1 * \ldots * P_n$ and $Q_1 * \ldots * Q_n \Rightarrow Q$. The rule for parallel composition has the side condition that no variable that is free in P_i or Q_i is modified in C_j when $j \neq i$.

- The proof rule for the conditional critical region is as follows:

$$\frac{\{(P * RI_r) \wedge b\}\ C\ \{Q * RI_r\}}{\{P\}\ \texttt{with r when b do C endwith}\ \{Q\}}$$

  subject to the side condition that no other process modifies variables that are free in P or Q.
  The idea with this rule is that when inside a critical region, the process executing the code C can access the state associated with the resource (RI_r) as well as the state that is local to the process, while when outside a critical region, command reasoning proceeds without knowledge of the resource's state.

Ownership transfer in CSL. Another key feature of CSL is the support for reasoning about transfer of ownership of portions of program state between processes. This is illustrated by the code in figure 2.6. Suppose that we have a one-place buffer that is used by processes to store a pointer to an object. Using CSL we can reason about the following code, which involves two concurrent processes, one storing a pointer and the other retrieving a pointer from the buffer. This code is effectively sequential because the getter blocks until the putter finishes storing the pointer into the buffer, but this is a simple illustration of ownership transfer. The left process executes $put(buff, c)$ to store the pointer c into the

```
put(buff, c):                        get(buff, y):
with buff when not full do      ||   with buff when full do
    buff := c; full := true              y:= buff; full := false
end with                             end with
```

Figure 2.6: Code for a putter and getter process in parallel execution

buffer $buff$. The buffer has the following resource invariant:

$$RI_{buff} \triangleq (\neg full \wedge emp) \vee (full \wedge buff \mapsto \_)$$

The boolean variable $full$ is true when the buffer owns the pointer $buff$ and false when it is empty. In CSL it has been suggested to gove the points-to assertion "$E \mapsto F$" an "ownership reading": "I own the location E" (and hence have the ability to access it). So here we can think of the putter code as transferring ownership of the pointer c from the left process to the buffer. The right process, on the other hand, acquires ownership of the pointer c from the buffer by executing $get(buff, y)$. A proof outline for each of the parallel branches of figure 2.6 are given in figures 2.7 and 2.8. In figure 2.7, the process starts execution with ownership of the pointer c. When it acquires the resource $buff$ in the conditional critical region, it assumes that the resource invaiant RI_{buff} holds. The proof steps from (a) to (b) are justified by the following steps:

$$\begin{aligned}
& RI_{buff} \wedge \neg full \\
& \Longrightarrow ((\neg full \wedge emp) \vee (full \wedge c \mapsto \_)) \wedge \neg full \\
& \Longrightarrow ((\neg full \wedge \neg full \wedge emp) \vee (\neg full \wedge full \wedge c \mapsto \_) \\
& \Longrightarrow ((\neg full \wedge emp) \vee (False \wedge c \mapsto \_) \\
& \Longrightarrow ((\neg full \wedge emp) \vee False \\
& \Longrightarrow \neg full \wedge emp
\end{aligned}$$

The justification for proof steps (c) to (d) is similar, and also uses the following lemma:

$$(P \wedge emp) * Q = (P \wedge Q) * emp$$

```
{$c \mapsto \_$}
with buff when not full do
   {$(RI_{buff} \wedge \neg full) * c \mapsto \_$}   $(a)$
   {$(\neg full \wedge emp) * c \mapsto \_$}   $(b)$
   buff := c;
   {$(\neg full \wedge emp) * buff \mapsto \_$}
   full := true;
   {$(full \wedge emp) * buff \mapsto \_$}   $(c)$
   {$(full \wedge buff \mapsto \_) * emp$}   $(d)$
   {$RI_{buff} * emp$}   $(e)$
end with
{$emp$}
```

Figure 2.7: Proof outline for $put(buff, c)$

which holds as long as P is a heap-free assertion and Q is any assertion. The resource invariant is re-established from steps (d) to (e) because $full \wedge buff \mapsto \_ \implies (full \wedge buff \mapsto \_) \vee (\neg full \wedge emp) \equiv RI_{buff}$. When the process exits the CCR, the assertion that holds of the final state is emp, i.e. the process does not own any heap; thus it has relinquished ownership of the pointer c to the buffer. The proof outline for $get(buff, y)$ is shown in figure 2.8. This proves that the process executing $get()$ gains ownership of the pointer from the buffer.

Comparison of the parallel composition rule with Owicki-Gries. The presentation of this version of CSL is based on that of Owicki-Gries [31], specifically the use of resources and resource invariants and the use of conditional critical regions. In Owicki-Gries the rule for parallel composition is given as follows (with the notation slightly adjusted to be consistent with the one used here):

If: $\{P_1\}\ C_1\ \{Q_1\} \ldots \{P_n\}\ C_n\ \{Q_n\}$ and no variable that is free in P_i or Q_i is modified in C_j when $j \neq i$, and all variables in RI_r belong to the resource r, then:

$\{P_1 \wedge P_2 \wedge \ldots \wedge P_n \wedge RI_r\}$
`resource` r : `cobegin` $C_1 \parallel C_2 \parallel \ldots \parallel C_n$ `coend`
$\{Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n \wedge RI_r\}$

The differences are:

1. the Owicki-Gries rule uses ordinary logical conjunction whereas CSL uses separating conjunction, therefore the former is unsound in the presence of pointers and aliasing;

```
{$emp$}
with buff when full do
  {$(RI_{buff} \wedge full) * emp$}
  {$(full \wedge buff \mapsto \_) * emp$}
  y := buff;
  {$(full \wedge y \mapsto \_) * emp$}
  full := false;
  {$(\neg full \wedge y \mapsto \_) * emp$}
  {$(\neg full \wedge emp) * y \mapsto \_$}
  {$RI_{buff} * y \mapsto \_$}
end with
{$y \mapsto \_$}
```

Figure 2.8: Proof outline for $get(buff, y)$

2. the former requires all variables of RI_r to belong to r, while in CSL the requirement is that whenever a variable x is modified by a command and x is free in some RI_r then the command must occur only within a critical region for r; and

3. the former explicitly includes the resource invariant in the pre- and post-condition; in the CSL rule this is implicit because parallel composition is always used in the context of a bigger program, which has an intialisation sequence that ensures that the conjunction of all the resource invariants holds outside critical sections. But the idea is the same – the resource invariants are expected to hold at all times when outside a critical section. In CSL this is ensured by the intialisation sequence and the rule for CCR, which requires any process that accesses a resource to re-establish the resource invariant after using the resource.

2.3 Rely guarantee reasoning

Rely guarantee is a method invented by Jones [21, 22] for reasoning about interfering programs. In Hoare logic, the pre-condition P of a program restricts the initial states that the program can execute in; it can be considered as an assumption that a programmer can make about the environment of the program so that it will run correctly. On the other hand the post-condition Q describes the effect of the program on the state. It can be modelled either as a predicate on state or a relation between the initial and final state, as done by Jones in his VDM formalism. Hoare triples $\{P\}\ C\ \{Q\}$ are sufficient to reason about sequential programs, which can be thought of as transforming some initial state to some final state; however for concurrent programs one must also consider the interactions between a program and other running processes (i.e. the environment) and the possible interference on the shared state. For this, rely guarantee extends pre- and post-conditions to include rely and guaratee conditions, which are relations between the intial and final states of a step of an execution by the environment or the program under consideration. A rely condition encodes the interference that the program can tolerate from the environment, and a guarantee condition encodes the interference that the program can produce on the shared state. To elaborate on this, it would be useful to present the computational model in terms of a modified small-step semantics, where the environment steps and the steps taken by the program or one of its components is made explicit. Such a semantics is given, for example in [29] and [Moreira et al.]. We can define a small-step relation where the transitions between configurations are labelled as follows: $C, \sigma \xrightarrow{\delta} C', \sigma'$, where C denotes a command, σ denotes a state, and $\delta = \{e, c\}$, with $\xrightarrow{e}$ denoting a step by the environment, and $\xrightarrow{c}$ denotes a step by a program component. We then allow environment steps to modify the state component of a configuration but not the command component: $C, \sigma \xrightarrow{e} C, \sigma'$. Component steps are allowed to modify both the command and state components of configurations. The definition of component steps will be the same as in figure 2.2, except that all transitions are labelled ($\xrightarrow{c}$). With this formal semantics, we can define a computation as a sequence of interleaved transitions taken by both the environment and the program components: $C_1, \sigma_1 \xrightarrow{\delta} C_2, \sigma_2 \xrightarrow{\delta} \ldots \xrightarrow{\delta} C_k, \sigma_k \ldots$ In this way we can formalise reasoning about interference and rely and guarantee conditions (this would be particularly useful in proofs of soundness of the inference rules presented below).

Rely guarantee specifications have the form $\{R, P\}\ C\ \{Q, G\}$ and the meaning of this is that: if the program starts execution in a state satisfying pre-condition P, every execution step of the environment affects the shared state according to the rely condition R, and every execution step of the program affects the shared state according to the guarantee condition G, then if the program terminates the final step satisfies the post-condition Q. The inference rules for rely guarantee are presented in figure 2.11, which is adapted from [Moreira et al.][1]. The inference rules are explained below. Firstly note that some of these rules require showing the stability of pre- and post-conditions with respect to the rely condition. This is because in the computation model as explained above, a program step can be preceeded by zero or more environment steps and followed by zero or more

[1]There are various presentations of RG inference rules but we chose to use this one because the rules are formulated in a goal-oriented way, which makes it amenable to mechanisation, as will be explained later.

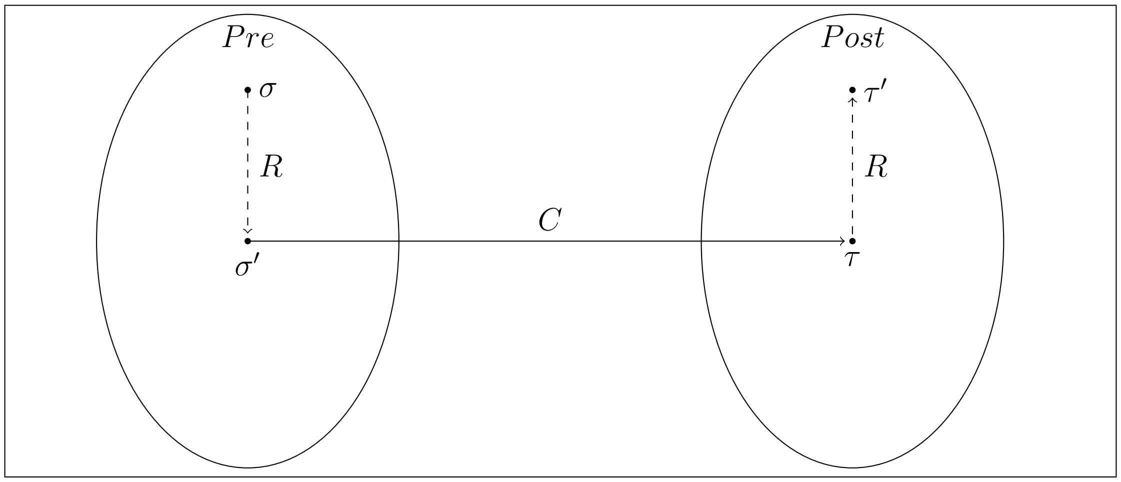


Figure 2.9: Stability of Pre and $Post$ with respect to a rely condition R

environment steps. Formally stability of an assertion P with respect to a relation R is defined as follows [Moreira et al.]:

$$\textit{stable}\ R\ P \triangleq \forall \sigma, \sigma' \in \Sigma \cdot P(\sigma) \Rightarrow (\sigma, \sigma') \in R \Rightarrow P(\sigma')$$

where Σ denotes the set of all states. If σ is a state that satisfies a precondition P and if the environment takes a step that updates that state to the state σ', then the updated state still satisfies the precondition P, thereby allowing the environment to "move" the satisfiability of the precondition to the state where the actual execution of C starts; and equally for a postcondition. Figure 2.9 illustrates this idea: here, $\sigma \in Pre$ is updated by the environment interference R to a state σ' that is also in Pre, where the execution of C actually starts. The execution terminates in state $\tau \in Post$, which the environment transitions to τ', which is still in $Post$.
Explanation of the inference rules presented in figure 2.11: for the standard imperative commands, the basic idea is to assume that the inference rules from Hoare logic are still sound, and then add any necessary premisses in the rule to ensure that the rules remain sound in the presence of interference. This means checking that the pre- and post-conditions as well as any boolean guards are stable with respect to the rely condition.

1. Rule for skip
   The skip command does not involve any program steps hence no state transitions. The proof obligations are to show the stability of P and Q with respect to the rely condition and also to show that P implies Q.

2. Rule for assignment
   For this we need to show the stability of P and Q, that P implies $Q[e/x]$

as with standard Hoare logic, and also that the change in state due to the assignment is included in the guarantee condition.

3. Rule for sequential composition
We need to show that the commands C_1 and C_2 satisfy their own specifications. Each specification uses the same rely and guarantee condition, and the postcondition of C_1 becomes the precondition of C_2, as per standard Hoare logic.

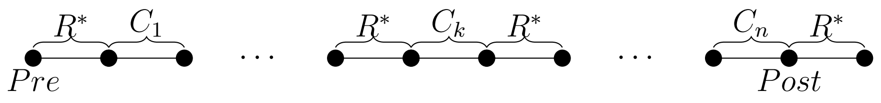
4. Rule for conditional
We need to show that each branch of the conditional satisfies its own specification, and that the boolean guard and the precondition of the whole conditional are stable with respect to the rely condition.

5. Rule for while
We need to show that the body of the loop satisfies its own specification and the boolean guard is stable with respect to the rely condition.

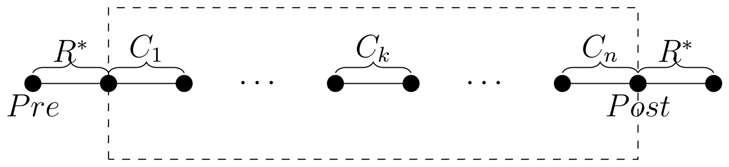
6. Rule for atomic
An atomic block ensures that the enclosed command C executes without interference from the environment. In this case we must then show that the command is a valid sequential command in the absence of interference, i.e., when the rely condition is just the identity relation ID on states. Moreover, the atomic block can suffer environmental interference *before* or *after* the execution of the block, hence we must show that the pre- and post-condition are stable with respect to a rely condition R that is not necessarily the identity relation. Figure 3.7 illustrates this idea.

7. Rule of consequence
This is just an extension of the rule of consequence of Hoare logic. In addition to strengthening the precondition and weakening the postcondition, this rule also allows for strengthening the rely condition by replacing it with $R \subseteq R'$ and weakening the guarantee condition by replacing it with $G \supseteq G'$.

8. Rule for parallel composition
Firstly we need to show that each component program satsifies its own specification. Then we show that the rely and guarantee conditions of the component programs satisfy certain relationships: given a component C_i, it must be able to tolerate interference from the environment as well as from the actions of the other component C_j (for $i \neq j$); hence we must show that $R \cup G_i \subseteq R_j$ for $i \neq j$. In addition we must show that the guarantee condition of the parallel composition is at least equal to the union of the guarantee condition of each component (hence $G_1 \cup G_2 \subseteq G$.



(a) Non-atomic execution of the command sequence $C_1; \ldots ; C_n$ interleaved with environmental interference



(b) Atomic execution of $C_1; \ldots ; C_n$ – interference only happens before or after the execution

Figure 2.10: Non-atomic vs. atomic execution

Finally, since the parallel composition starts execution in a common state satisfying P we must show that this state is at least the intersection of the preconditions of each component $P_1 \cap P_2$ and equally, the terminal state of the composition at least contains the intersection of the component postconditions i.e. $Q_1 \cap Q_2 \subseteq Q$.

Note re. scope of rely and guarantee conditions. In the papers by Jones and susbequent studies of rely guarantee, the formalism assumes a programming model in which there is one shared, global state. There is the question of whether or not the specification of rely and guarantee conditions must cover the entire global state. Most examples (of small to medium-sized programs) do specify the entire state. More recent developments such as RGSep and LRG (discussed in the next chapter) do claim that traditional rely guarantee conditions must specify the entire state, and improve upon these by using separation logic concepts to prove the ability to specify local states or parts of the shared state.
There is also the general, related question of what can be done to program variables that are not mentioned in assertions about program state. There are at least two different approaches:

- Using explicit frames e.g. in refinement calculus, the specification construct $\overline{w} : P$ – "modify any of the variables in the list $\overline{w}$ such that P holds. This sets out explicitly what can be modified hence, x is modifiable iff $x \in \overline{w}$. Separation logic takes the same approach, where assertions specify the memory footprints of commands.

- In Hoare logic, if a variable is not mentioned it may be modified. For example, the triple $\{x = 0\}\ y := 1\ \{x = 0\}$ is provable. Here y does

$$\frac{\mathit{stable}\ R\ P \qquad \mathit{stable}\ R\ Q \qquad P \Rightarrow Q}{\{R, P\}\ \mathit{skip}\ \{Q, G\}}\ \text{(SKIP)}$$

$$\frac{\begin{array}{l}\mathit{stable}\ R\ P\\ \mathit{stable}\ R\ Q\end{array} \qquad \forall s \in \Sigma \cdot (s, s[E/x]) \in G \qquad P \Rightarrow Q[E/x]}{\{R, P\}\ x := E\ \{Q, G\}}\ \text{(ASSIGN)}$$

$$\frac{\{R, P\}\ C_1\ \{Q', G\} \qquad \{R, Q'\}\ C_2\ \{Q, G\}}{\{R, P\}\ C_1; C_2\ \{Q, G\}}\ \text{(SEQ)}$$

$$\frac{\begin{array}{l}\mathit{stable}\ R\ [\![B]\!]\\ \mathit{stable}\ R\ [\![\neg B]\!]\end{array} \qquad \mathit{stable}\ R\ \ P \qquad \begin{array}{l}\{R, P \wedge B\}\ C_1\ \{Q, G\}\\ \{R, P \wedge \neg B\}\ C_2\ \{Q, G\}\end{array}}{\{R, P\}\ \mathit{if}\ B\ \mathit{then}\ C_1\ \mathit{else}\ C_2\ \mathit{fi}\ \{Q, G\}}\ \text{(COND)}$$

$$\frac{\mathit{stable}\ R\ [\![B]\!] \qquad \mathit{stable}\ R\ [\![\neg B]\!] \qquad \{R, B \wedge P\}\ C\ \{P, G\}}{\{R, P\}\ \mathit{while}\ B\ \mathit{do}\ C\ \mathit{done}\ \{\neg B \wedge P, G\}}\ \text{(WHILE)}$$

$$\frac{\begin{array}{l}\mathit{stable}\ R\ P\\ \mathit{stable}\ R\ Q\end{array} \qquad \{ID, P\}\ C\ \{Q, G\}}{\{R, P\}\ \mathit{atomic}(C)\ \{Q, G\}}\ \text{(ATOMIC)}$$

$$\frac{\begin{array}{l}P \Rightarrow P'\\ Q' \Rightarrow Q\end{array} \qquad \begin{array}{l}R \subseteq R'\\ G' \subseteq G\end{array} \qquad \{R', P'\}\ C\ \{Q', G'\}}{\{R, P\}\ C\ \{Q, G\}}\ \text{(CONSEQ)}$$

$$\frac{\begin{array}{l}\{R_1, P_1\}\ C_1\ \{Q_1, G_1\}\\ \{R_2, P_2\}\ C_2\ \{Q_2, G_2\}\end{array} \qquad \begin{array}{l}R \cup G_1 \subseteq R_2\\ R \cup G_2 \subseteq R_1\\ G_1 \cup G_2 \subseteq G\end{array} \qquad \begin{array}{l}P \subseteq P_1 \cap P_2\\ Q_1 \cap Q_2 \subseteq Q\end{array}}{\{R, P\}\ C_1 \parallel C_2\ \{Q, G\}}\ \text{(PAR)}$$

Figure 2.11: Inference rules for rely guarantee

not occur freely in the pre- or post-condition but can be modified by a command. To get around this, either mention y explicitly, or use "ownership assertions" like in some versions of separation logic (e.g see [7]).

End of Note.

2.4 Weakest precondition calculus for separation logic

In his paper [11], Dijkstra presented the notion of predicate transformer semantics of programs which views programs as functions that map state predicates to state predicates. This is an equivalent re-formulation of Hoare logic: while Hoare logic is a deductive system, predicate transformer semantics are complete strategies for building valid deductions in Hoare logic and reduce the problem of proving a Hoare triple to that of finding a first-order formula whose validity implies the provability of the Hoare triple. Dijkstra first introduced the weakest precondition (wp) as an example of a predicate transformer. Formally the relationship between Hoare triples and wp is given by $\vdash \{P\}\ C\ \{Q\} \iff (P \Rightarrow wp(C, Q))$. The wp equations for the standard imperative programming commands are given by Dijkstra in [11].

The weakest precondition calculus for (sequential) separation logic is given in some of the early papers on separation logic. In [18] the authors present the logic of bunched implications (BI) as an assertion language for mutable data structures. In their discussion of the completeness of the logic, the authors show that the assertion language allows the weakest precondition to be expressed. They show how the wp for each atomic statement can be expressed in the logic, together with a proof that these assertions are indeed the weakest preconditions. The wp equations for SL are also presented by John Reynolds in [34] where he includes "backwards reasoning" formulations of the small axioms of separation logic. Here we present a version of the weakest precondition calculus for the language we are working with, which supports records in the heap.

As discussed in section 2.1.1 we are working with a version of separation logic in which the heap stores records of the form $(f_1 : E_1, \dots, f_n : E_n)$ where the f_k are field names and E_k expressions denoting values (which can be pointers to records, hence allowing for the construction of complex linked data structures). Here we present the weakest preconditions for heap-altering commands, where P is an arbitrary postcondition. Note that these equations assume that records have two fields (as that will suffice for most of the examples we will be treating) but the extension to $n > 2$ fields should be straightforward.

- Record creation

$$wp(\mathrm{r} := \mathbf{new}[f_1 : e_1, f_2 : e_2],\ P) = \forall r' \cdot r' \mapsto [f_1 : e_1, f_2 : e_2] -\!\!* P[r'/r] \tag{2.4.1}$$

- Field lookup

$$wp(\mathrm{x} := f_1(r),\ P) = \exists a \cdot P[a/x] \wedge \exists b \cdot r \mapsto [f_1 : a, f_2 : b] \tag{2.4.2a}$$
$$wp(\mathrm{x} := f_2(r),\ P) = \exists b \cdot P[b/x] \wedge \exists a \cdot r \mapsto [f_1 : a, f_2 : b] \tag{2.4.2b}$$

- Field update

$$wp(f_1(r) := a',\ P) = \exists ab \cdot r \mapsto [f_1 : a, f_2 : b] * ((r \mapsto a', b) -\!\!* P) \tag{2.4.3a}$$
$$wp(f_2(r) := b',\ P) = \exists ab \cdot r \mapsto [f_1 : a, f_2 : b] * ((r \mapsto a, b') -\!\!* P) \tag{2.4.3b}$$

- Record deallocation

$$wp(\mathbf{delete}(r),\ P) = P * \exists ab \cdot r \mapsto [f_1 : a, f_2 : b] \tag{2.4.4}$$

The benefit of wp calculus is that it can be used to do proofs in a "calculational style" by working backwards and deriving verification conditions from the separation logic specifications. We shall demonstrate the use of this in chapter 4 where we present some fully worked-out proofs.

Chapter 3

Combining separation logic with rely guarantee

In this chapter we consider various formalisms which combine separation logic with rely guarantee. RGSep, developed by Viktor Vafeiadis and Matthew Parkinson ([39, 43]), and Local Rely Guarantee (LRG) by Xinyu Feng ([13]) both combine the strengths of separation logic (local reasoning) and rely guarantee (reasoning about interference). This combination has shown to be beneficial to correctness proofs of fine-grained concurrent algorithms (e.g. see Vafeiadis PhD thesis [39]). RGSep is interesting because it is the first logic that combines ideas from SL and RG, and Vafeiadis has done substantial proofs of algorithms with it, and LRG is interesting because it explores the SL+RG combination further, as discussed in this chapter, although there has not been much work in terms of practical proofs with LRG as compared to RGSep.

This chapter is organised as follows: first we discuss the motivations for combining SL and RG in a single formalism; we then discuss the key ideas and features of RGSep, and then we discuss LRG; we then present proofs of a simple counter and a linked list stack in both formalisms, and then compare these proofs.

Note on notation In our introduction to rely guarantee in section 2.3 we use the notation $\{R, P\}\ C\ \{Q, G\}$ to denote a rely-guarantee specification. In this chapter we shall switch to a different notation used by Vafeiadis, et. al. in their papers: $C\ \underline{sat}\ (P, R, G, Q)$. This denotes that the program C satisifies the rely-guarantee specification expressed as a 4-tuple (precondition P, rely condition R, guarantee condition G, postcondition Q). The semantics is exactly the same.

3.1 Motivations for merging separation logic and rely guarantee

The basic problem with reasoning about shared variable concurrency is reasoning about inter-thread interference – i.e. showing that updates by different processes to the shared state are totally disjoint, or can be controlled via some synchronisation mechanism such that the resulting updates are not in conflict or leave the shared data structure in some invalid state. Rely guarantee reasoning provides a nice way to reason about interference via rely and guarantee conditions, which are relations that precisely describe state updates performed either by the components of a program or by the environment. However, the main weakness of rely guarantee is that the checking for interference is done globally – it must be checked against every state update even when it is obvious that an update cannot interfere with everything else (as noted by Vafeiadis in [43]). On the other hand, separation logic enables modular reasoning – the separating conjunction ($*$) operator plus the frame rule can be used to remove irrelevant state out of a specification and focus only on the state that matters for the execution of a component or thread. While CSL was invented as an extension of SL for concurrency and is good for reasoning about "ownership transfer" of state between process, its main weakness is that it uses invariants to specify thread interaction – this makes the expression of the relational nature of interference difficult and often requires auxiliary variables. So what is needed is a single formalism that will exploit the benefits of both approaches – local, modular, explicit reasoning about interference between threads in a shared variable concurrent program. This is the main motivation for the formalisms that we will discuss in detail in this chapter.

3.2 Key ideas and features of RGSep

RGSep combines ideas from separation logic and rely guarantee in the following ways:

- Enabling local reasoning by splitting the model of program state into local and shared parts
- Use of separating conjunction ($*$) to allow splitting of local state into disjoint parts
- Use of local reasoning when describing thread interference on shared state (this includes the application of the "ownership transfer" concept to describe updates to shared state (explained in section 2.2))

- Use of the SL "magic wand" operator to reason about the stability of assertions with respect to thread interference

We discuss each of these in turn.

3.2.1 RGSep model of program state

In RGSep, program state is split into two components, local and shared, where each component can be thought of as a finite partial function from locations to values. The domains of the two components are required to be disjoint, so that the total state is simply the disjoint union of the two components. In RGSep the assertion language makes a syntactic distinction between assertions on local state and assertions on shared state: local state is described by "unboxed assertions" P and shared state by "boxed assertions" $\boxed{P}$. When the separating conjunction $(*)$ is used to join two local assertions $P * Q$ it denotes the splitting of the local state into disjoint parts; however when it is used to conjoin to boxed assertions as in $\boxed{P} * \boxed{Q}$ this reduces to the assertion $\boxed{P \wedge Q}$, i.e., both P and Q are true of the shared state. This does not mean though that the separating conjunction cannot be used *inside* a boxed assertion i.e. $\boxed{P * Q}$ – the shared state can also be split into disjoint parts just like local state. For example, if a variable L points to a linked list modelling a sequence α on the heap that is globally shared, the assertion describing it is $\boxed{\exists t \cdot L \mapsto (hd\ \alpha), t\ *\ list(t,\ tl\ \alpha)}$. (Note: in his first paper on RGSep ([43]) Vafeiadis constructs a model of state where there can be multiple local states (owned by different threads) but only one region of shared memory that is shared by all threads and the paper says, somewhat misleadingly, that one can split local state but not shared state. But what he is saying is that one cannot use $*$ to conjoin boxed assertions because there is only one region of shared state; in a later version of RGSep in his PhD thesis Vafeiadis presents a model where there can be multiple regions of shared states and it is possible to conjoin boxed assertions e.g. $\boxed{P}_r * \boxed{Q}_s$ where r and s are distinct regions of shared state).

3.2.2 Parallel composition

A key insight of Vafeiadis in RGSep is to use the separating conjunction connective $(*)$ from separation logic in rely guarantee specifications. Recall that in separation logic, if $P*Q$ is true of some resource (e.g. a heap), then that resource can be split into two disjoint parts, where P holds of one part and Q holds of the other. In the rely guarantee inference rule for disjoint parallel composition, where C_1 and C_2 are programs executed by separate threads and accessing disjoint parts of program state, Vafeiadis introduced the separating conjunction in the pre- and post-conditions for the parallel composition:

$$\frac{C_1 \ \underline{sat}\ (P_1, R \cup G_2, G_1, Q_1) \qquad C_2 \ \underline{sat}\ (P_2, R \cup G_1, G_2, Q_2)}{C_1 \parallel C_2 \ \underline{sat}\ (P_1 * P_2, R, G_1 \cup G_2, Q_1 * Q_2)}$$

This expresses the idea that the two parallel components operate on disjoint parts of program state. Note that the assertions P_i, Q_i in the pre- and post-conditions can talk about both local and shared state, and a syntactic distinction is made between the two (discussed above).

3.2.3 Local reasoning about shared state

RGSep uses "actions" to describe changes performed to the shared state. An action is of the form $P \rightsquigarrow Q$ where P and Q are (separation logic) assertions. Intuitively an action replaces part of the state that satisfies P with a part that satisfies Q, leaving the rest unchanged, in the spirit of local reasoning of separation logic. Formally, an action is defined as follows: given program states h_1, h_2, and h_0, an interpretation i that maps logical variables to values, and separation logic assertions P and Q, an action is given by the following relation:

$$[\![P \rightsquigarrow Q]\!] = \{(h_1 \uplus h_0, h_2 \uplus h_0) | h_1, i \models_{SL} P \text{ and } h_2, i \models_{SL} Q\}$$

where h_1 is the initial state that is transformed into the final state h_2 and h_0 is the frame, i.e. the part of the state that is unchanged.

RGSep also extends the frame rule from separation logic. Recall the frame rule from section 2.1.4:

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}} \qquad \text{where } fv(R) \cap mod(C) = \varnothing$$

This says that if C satisfies the specification which turns P into Q then it also satisfies the bigger specification which turns $P * R$ into $Q * R$, where R (the "frame") is a piece of state that is disjoint from both P and Q, and where the side condition ensures that the code of C does not modify any variable mentioned by the frame. The extension to RGSep is

$$\frac{\vdash C \ \underline{sat}\ (P, R, G, Q)}{\vdash C \ \underline{sat}\ (P * R', R, G, Q * R')} \quad [stable\ R'\ (R \cup G)]$$

This says that if C can run safely without the extra state R', then it can run safely with R'. Since in general the frame R' can mention the shared state, we require it to be stable under the union of the rely and guaratee conditions.[1]

Examples.

1. Action incrementing the value of a counter x, where M, N are logical variables, which are existentially bound with scope ranging over the pre- and post-conditions:
$$x \mapsto M \leadsto x \mapsto N \wedge N > M$$

2. For some suitable definition of the predicate $\mathit{Stack}(x)$, this is an action that describes pushing a node M onto a stack whose top is currently pointing to N:
$$\mathit{Top} \mapsto N * \mathit{Stack}(N) \leadsto \mathit{Top} \mapsto M * M \mapsto N * \mathit{Stack}(N)$$

Having modelled state updates as actions, rely and guarantee specifications are then modelled as sets of actions in RGSep

Example. Given a simple counter that stores a monotonically increasing value (this is treated in detail in section 3.4.2) we define the following actions to model (a) incrementing the value of the counter by exactly one and (b) increasing the value of the counter by an arbitrary amount:

$$\begin{aligned} \mathit{Inc}_1 &\triangleq \exists AB \cdot \mathit{ctr} \mapsto \mathit{val} : A \leadsto \mathit{ctr} \mapsto \mathit{val} : B \wedge B = 1 + A \\ \mathit{Inc}_N &\triangleq \exists AB \cdot \mathit{ctr} \mapsto \mathit{val} : A \leadsto \mathit{ctr} \mapsto \mathit{val} : B \wedge A \leq B \end{aligned}$$

then we can define the rely and guarantee conditions for a system of concurrent threads incrementing the counter, where the environment is allowed to increase the value of the counter by an arbitrary amount as follows:

$$\begin{aligned} R &\triangleq \{\mathit{Inc}_N\} \\ G &\triangleq \{\mathit{Inc}_1\} \end{aligned}$$

Stability of assertions with respect to an (RGSep) action. Rely guarantee proofs require showing that every pre- and post-condition in a proof is stable under interference from the environment. Recall that in section 2.3 we defined

[1]Note: in the previous version there was a requirement that C "contains no atomics" but Vafieadis removed that in the (later) version of RGSep, presented in his thesis, as it is too limiting.

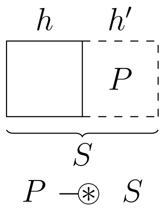


Figure 3.1: Septraction: removing P from S

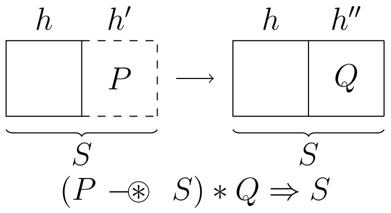


Figure 3.2: State update modelled by septraction

what it means for an assertion to be stable with respect to a relation (e.g. a rely condition). RGSep extends this concept to define the stability of an assertion with respect to an RGSep action: given an assertion S and an action $a \triangleq P \rightsquigarrow Q$, S is said to be stable with respect to a if and only if $\models_{SL} (P \mathrel{-\!\!\circledast} S) * Q \Rightarrow S$. Here $(\mathrel{-\!\!\circledast})$ is the "septraction" operator, which is the existential dual of the "magic wand" $(-\!\!*)$ operator from separation logic [2]. It is formally defined as:

$$s, h \models_{SL} P \mathrel{-\!\!\circledast} S \text{ iff } \exists h' \cdot h \# h' \text{ and } s, h' \models_{SL} P \text{ and } s, h \uplus h' \models_{SL} S$$

where s is a store, h a heap, and h' a heap that is disjoint from h. The definition says that for a given heap h there is a heap h' that satisfies the assertion P and the heap that results from extending h with h' satisfies S. Thinking backwards one can think of this as "subtracting P from S" (see figure 3.1). Given this, we can model a state update using septraction: from a state S, replace the part that satisfies P with one satisfying Q, i.e. $(P \mathrel{-\!\!\circledast} S) * Q$ (see figure 3.2).

3.3 Key ideas and features of Local Rely Guarantee

Like RGSep, LRG also splits program state into thread-private and shared parts, with RG reasoning applied only to the shared part. However it borrows even more from separation logic to support the following features:

[2]Formally the relationship between the two is $P \mathrel{-\!\!\circledast} Q \iff \neg(P -\!\!* \neg Q)$

- More fine-grained model of program state
- Separating conjunction lifted to actions
- Extensions of the frame rule to support local rely and guarantee conditions
- Extensions to allow hiding of some resources by a subset of threads

The key result of LRG is that it relaxes the original restriction from rely guarantee reasoning that R and G specify all of shared state (see note at the end of section 2.3). Now it is possible to split shared state, talk about "local" rely and guarantee conditions, and introduce scope (as well as hide some shared resources) to allow for proofs of more fine-grained algorithms.

3.3.1 Model of program state

LRG allows for a more fine-grained model by allowing not just a splitting of state into local and shared but also a further splitting of shared state into parts that can be separately owned by different sets of threads. It supports the following sharing patterns:

- State owned locally by a thread
- State shared by a group of threads
- State shared globally by all threads

These patterns of sharing are enabled by a "hiding" rule (described below). It allows, for example, for a parent thread to distribute its own local resources to the threads it spawns.

3.3.2 Actions and the separating conjunction

In LRG, actions are used to specify state transitions over both local and shared state. They are of the form $P \ltimes Q$, which means that the initial state of the transition satisfies P and the final state satisfies Q. An identity transition that satisfies P is denoted by $[P]$. It is possible to compose two actions using the $*$ operator. Intuitively, $a * a'$ is an action over a state that can be split into two disjoint parts, where each component action operates over a disjoint part, and the resulting final states are also disjoint (see figure 3.3). The semantics of an action is given over pairs of triples $\sigma = (s, i, h)$ where s is a store, i an interpretation for logical variables, and h a heap.

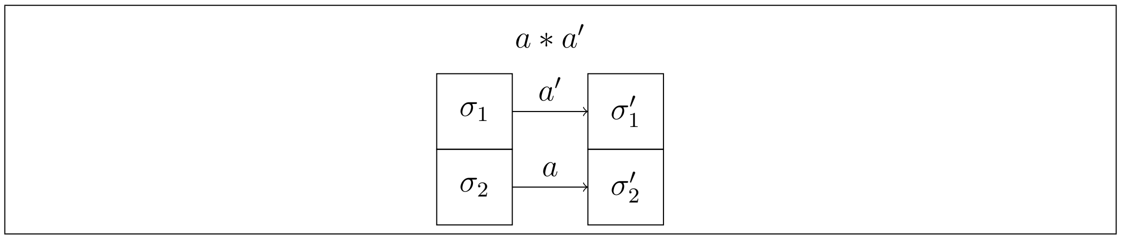


Figure 3.3: The action $a * a'$ operating over disjoint pieces of state

$$
\begin{array}{ll}
(\sigma, \sigma') \models P \ltimes Q & \text{iff } \sigma.i = \sigma'.i \text{ and } \sigma \models_{SL} P \text{ and } \sigma' \models_{SL} Q \\
(\sigma, \sigma') \models [P] & \text{iff } \sigma = \sigma' \text{ and } \sigma \models_{SL} P \\
(\sigma, \sigma') \models a * a' & \text{iff for some } \sigma_1, \sigma_2, \sigma'_1, \sigma'_2 \text{ have } \sigma = \sigma_1 \uplus \sigma_2 \text{ and } \sigma' = \sigma'_1 \uplus \sigma'_2 \\
& \text{and } (\sigma_1, \sigma'_1) \models a \text{ and } (\sigma_2, \sigma'_2) \models a'
\end{array}
$$

Besides $*$ the other logical operators are also lifted to actions. The following are special actions in LRG:

- The empty action: $\mathit{Emp} \triangleq \mathit{emp} \ltimes \mathit{emp}$ where emp denotes empty store and empty heap
- The arbitrary action: $\mathit{True} \triangleq \mathit{true} \ltimes \mathit{true}$
- The identity transition $\mathit{Id} \triangleq [\mathit{true}]$

3.3.3 Local rely and guarantee conditions and the frame rule

Since rely and guarantee conditions are just actions, and LRG lifts the separating conjunction to actions, it is now possible to split rely and guarantee conditions as well. Hence $R * R'$ describes two sub-actions R, R' that occur over disjoint parts of shared state (equivalently for guarantees, $G * G'$). The frame rule can now be extended to:

$$\frac{\vdash C\ \underline{sat}\ ((p, r), R, G, (q, r'))}{\vdash C\ \underline{sat}\ ((p, r * m), R * R', G * G', (q, r' * m))} \quad m \text{ stable wrt to } R'$$

Here we have written the pre- and post-conditions as pairs (l, s) where l, s are local and shared state, respectively. (Also note that we have written an RG specification as $\vdash C\ \underline{sat}\ (p, R, G, q)$ where Feng et. al. write $R; G \vdash \{p\}\{C\}\{q\}$, and I'm thinking of switching back to that later).

$$\dfrac{\begin{array}{l}\vdash C\ \underline{sat}\ (P,\emptyset,\emptyset,Q)\\ (P \rightsquigarrow Q)\subseteq G\\ \boxed{\mathrm{Q}}\ \textit{stable under } R\end{array}}{\vdash \mathbf{atomic}(B)\ \{C\}\ \underline{sat}\ (\boxed{\mathrm{P}},R,G,\boxed{\mathrm{Q}})}\quad [\text{ATOMIC}]$$

Figure 3.4: Proof rule for atomic blocks in RGSep

3.3.4 Hiding of shared resources by a subset of threads

LRG allows hiding of the local sharing of resources by a subset of threads via the hiding rule:

$$\frac{\vdash C\ \underline{sat}\ ((p, r * m), R * R', G * G', (q, r' * m'))}{\vdash C\ \underline{sat}\ (p * m, r, R, G, q * m', r')}$$

This rule says that if the resource specficed by m and m' is shared locally inside C, and transitions over the resource are specified by R', G', then we can treat it as private and hide $R'G'$ as well so that it is invisible from the outside world.

3.4 Example proofs and comparison

3.4.1 Example proof 1: A simple atomic counter

In this and the subsequent section, we present some examples to illustrate proofs within RGSep and LRG. Refer to figure 3.4 for the RGSep inference rules and 3.6 for the LRG inference rules.

In this section we prove the correctness of concurrent code that uses a counter whose value can be atomically incremented. Listing 3.1 shows the initialisation of the counter and listing 3.2 shows the code for incrementing the value of the counter. Figure 3.7(a) shows the counter data structure, a record with a single field named *val*, and figure 3.7(b) shows the pre- and post-states of the counter incrementation.

Listing 3.1: Initialisation of a simple counter

```
init(){ ctr := new (val: 0) }
```

Listing 3.2: Incrementing the value of the simple counter

```
inc() {
  atomic(true) {
```

$$\frac{}{x \mapsto y \rightsquigarrow x \mapsto y \subseteq G} \quad \text{[G-EXACT]}$$

$$\frac{P \rightsquigarrow Q \in G}{P \rightsquigarrow Q \subseteq G} \quad \text{[G-AX]}$$

$$\frac{P_1 \rightsquigarrow S * Q_1 \subseteq G \qquad P_2 * S \rightsquigarrow Q_2 \subseteq G}{P_1 * P_2 \rightsquigarrow Q_1 * Q_2 \subseteq G} \quad \text{[G-SEQ]}$$

$$\frac{P \rightsquigarrow Q \subseteq G}{P[e/x] \rightsquigarrow Q[e/x] \subseteq G} \quad \text{[G-SUB]}$$

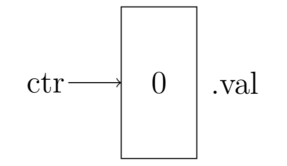
$$\frac{\models_{SL} P' \Rightarrow P \qquad P \rightsquigarrow Q \subseteq G \qquad \models_{SL} Q' \Rightarrow Q}{P' \rightsquigarrow Q' \subseteq G} \quad \text{[G-CONS]}$$

$$\frac{P * F \rightsquigarrow Q * F \subseteq G}{P \rightsquigarrow Q \subseteq G} \quad \text{[G-COFRM]}$$

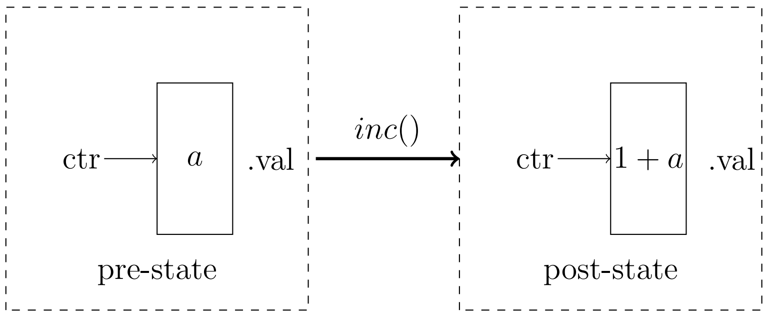
Figure 3.5: RGSep rules for actions allowed by a guarantee condition

$$\frac{\begin{array}{l} P \Rightarrow B = B \\ \{P \wedge B\}\ C\ \{Q\} \\ Sta(P, Q, R * Id) \\ P \ltimes Q \Rightarrow G * true \\ P \vee Q \Rightarrow I * true \\ I \rhd R, G \end{array}}{\textbf{atomic}(B)\ \{C\}\ \underline{sat}\ (R, G, I, P, Q)} \quad \text{[ATOMIC]}$$

Figure 3.6: Proof rule for atomic blocks in LRG



(a) Counter data structure



(b) Incrementing a counter

Figure 3.7: A simple counter

```
    t := val(ctr);
    val(ctr) := t + 1;
    return t
  }
}
```

In the next subsections we treat the correctness of concurrent programs that use this counter. We shall do the proofs in RGSep and LRG in order to compare these two formalisms. Note (as defined in previous sections) that RGSep specifications are 5-tuples $\vdash C\ \underline{sat}\ (P, R, G, Q)$, with the pre- and post-conditions P, Q being pairs (l, s) specifying local and shared state, while LRG specifications are 6-tuples $C\ \underline{sat}\ (P, R, G, I, Q)$. In the latter, the "fencing invariant" I specifies the boundaries between the shared and local parts of state (or equivalently, the boundaries of the rely and guarantee actions), while in the former this is not required because it explicitly splits state into shared and local parts.

```
$\{\boxed{\exists A \cdot ctr \mapsto val : A}\}$
inc() {
  atomic(true) {
    $\{\boxed{\exists A \cdot ctr \mapsto val : A}\}$
    $\{\boxed{ctr \mapsto val : A}\}$
    local t := val(ctr);
    $\{\boxed{ctr \mapsto val : A} * t \mapsto A\}$
    val(ctr) := t + 1;
    $\{\boxed{ctr \mapsto val : t + 1} * t \mapsto A\}$
    $\{\boxed{ctr \mapsto val : A + 1} * t \mapsto A\}$
    $\{\boxed{ctr \mapsto val : A + 1 \wedge A \leq A + 1} * t \mapsto A\}$
    $\{\boxed{\exists A, B \cdot ctr \mapsto val : B \wedge A \leq B} * t \mapsto A\}$
    return t
  }
}
$\{\boxed{\exists A, B \cdot ctr \mapsto val : B \wedge A \leq B}\}$
```

Figure 3.8: Proof outline for *inc*()

3.4.2 Simple counter: RGSep proof

First we define the rely and guarantee conditions. We assume that the environment can increment the counter and a thread can do the same. Hence:

$$R \triangleq \exists A \cdot ctr \mapsto val : A \leadsto \exists A, B \cdot ctr \mapsto val : B \wedge A \leq B$$
$$G \triangleq \exists A \cdot ctr \mapsto val : A \leadsto \exists A, B \cdot ctr \mapsto val : B \wedge B = 1 + A$$

Also, the pre- and post-conditions are:

$$P \triangleq \exists A \cdot ctr \mapsto val : A$$
$$Q \triangleq \exists A, B \cdot ctr \mapsto val : B \wedge A \leq B$$

We annotate the code with assertions as shown in figure 3.8. The atomic rule (see figure 3.4) applies, and the proof obligations are:

$$\vdash inc\ \underline{sat}\ (\exists A \cdot ctr \mapsto val : A, \emptyset, \emptyset, \exists A, B \cdot ctr \mapsto val : B \wedge A \leq B) \tag{3.4.1a}$$

$$(\exists A \cdot ctr \mapsto val : A) \tag{3.4.1b}$$
$$\leadsto (\exists A, B \cdot ctr \mapsto val : B \wedge A \leq B) \subseteq \exists A \cdot ctr \mapsto val : A \leadsto \exists A, B \cdot ctr \mapsto val : B \wedge B = 1 + A$$

$$(\exists A, B \cdot ctr \mapsto val : B \wedge A \leq B) \tag{3.4.1c}$$
$$\texttt{stable under}\ (\exists A \cdot ctr \mapsto val : A \leadsto \exists A, B \cdot ctr \mapsto val : B \wedge A \leq B)$$

Intuitively, these proof obligations require that:

1. the code for inc is sequentially correct, i.e., it is correct in the absence of interference (formally expressed by setting the rely and guarantee conditions to be just empty sets)

2. the action that models the overall effect of inc is allowed by the guarantee condition

3. the post-condition is stable under interference (i.e. wrt to the rely condition)

3.4.3 Simple counter: LRG proof

Suppose that a thread calls *ctr.inc*() in an environment where other threads are allowed to do the same. We want to show that the following specification holds:

$$inc\ \underline{sat}\ (P, R, G, I, Q)$$

where

$$\begin{aligned} P &\triangleq \exists a \cdot ctr \mapsto val : a \\ Q &\triangleq \exists a, b \cdot ctr \mapsto val : b \wedge (a + 1 = b) \\ R &\triangleq \exists a \cdot ctr \mapsto val : a \ltimes ctr \mapsto val : a \\ G &\triangleq \exists a \cdot ctr \mapsto val : a \ltimes ctr \mapsto val : (a + 1) \\ I &\triangleq \exists v \cdot ctr \mapsto v \end{aligned}$$

where a, b are logical variables.
For this proof, the ATOMIC rule (figure 3.6) is applicable. The following are the proof obligations:

$$P \Rightarrow (B = B) \tag{3.4.2a}$$
$$\{P \wedge B\}\ inc\ \{Q\} \tag{3.4.2b}$$
$$Sta(\{P, Q\}, R * Id) \tag{3.4.2c}$$
$$P \ltimes Q \Rightarrow G * true \tag{3.4.2d}$$
$$P \vee Q \Rightarrow I * true \tag{3.4.2e}$$
$$I \rhd R, G \tag{3.4.2f}$$

$\boxed{\textbf{Proof of 3.4.2a}}$ LRG uses the "variables as a resource" formulation of separation logic. In this formulation, an expression of the form $E = E$ is not necessarily a tautology; it does not suffice for E to be true at a given store – the store must also contain the variables needed to evaluate the expression E. This proof obligation is a consequence of this. However in this case the proof is trivial – $P \Rightarrow B = B$ with $B = true$ means no variables are required to evaluate the expression B; this $P \Rightarrow true$ which reduces to $true$. $\Box$

$\boxed{\textbf{Proof of 3.4.2b}}$ This amounts to a proof of the sequential specification for *inc*. We want to show:

$$\{\exists a \cdot ctr \mapsto val : a \wedge true\}$$

```
t:=val(ctr); val(ctr) := t + 1; return t
```

$$\{\exists a, b \cdot ctr \mapsto val : b \wedge (a < b)\}$$

Proof: A proof outline is shown in 3.8. $\Box$

$\boxed{\textbf{Proof of 3.4.2c}}$ We want to show the stability of the pre-condition and post-condition under rely condition, which is an action that preserves the value of the counter. The proof for the pre-condition is straightforward. Fix $\sigma, \sigma' \in \Sigma$ and let $\sigma \models \exists v \cdot ctr \mapsto val : v$. Instantiating v to v_0 we obtain $\sigma \models ctr \mapsto val : v_0$. Now suppose that $\sigma, \sigma' \models \exists v \cdot ctr \mapsto val : v \ltimes ctr \mapsto val : v$ and by instantiating, $\sigma, \sigma' \models ctr \mapsto val : v_0 \ltimes ctr \mapsto val : v_0$. This implies that $(\sigma \models ctr \mapsto val : v_0) \implies (\sigma' \models ctr \mapsto val : v_0)$ and the result follows by modus ponens.
The stability of the post-condition holds intuitively because the rely condition does not modify the value of the counter. Hence if in the pre-state of an environment transition, $ctr \mapsto val : b_0$ for some value b_0 such that $b_0 = a_0 + 1$ then in the post-state the counter still has the same value and the relation $b_0 = a_0 + 1$ still holds. $\Box$

$\boxed{\textbf{Proof of 3.4.2d}}$ We want to show

$$\begin{aligned}&(\exists a \cdot ctr \mapsto val : a) \ltimes (\exists a, b \cdot ctr \mapsto val : b \wedge (a + 1 = b))\\ &\Rightarrow (\exists a \cdot ctr \mapsto val : a) \ltimes (ctr \mapsto val : (1 + a) * True)\end{aligned}$$

Proof: To prove this we will need a rule or lemma for expanding the scope of a quantifier over an action, i.e.

$$\exists x \cdot Px \ltimes \exists x \cdot Qx \Rightarrow \exists x \cdot (Px \ltimes Qx)$$

or even a stronger lemma

$$\exists x \cdot Px \ltimes \exists x \cdot Qx \equiv \exists x \cdot (Px \ltimes Qx)$$

With this we can rewrite the proof obligation to

$$\begin{aligned}&(\exists a, b \cdot ctr \mapsto val : a \ltimes ctr \mapsto val : b \wedge (a + 1 = b))\\ &\Rightarrow (\exists a \cdot ctr \mapsto val : a) \ltimes (ctr \mapsto val : (1 + a) * True)\end{aligned}$$

Instantiating all the quantified variables gives

$$\begin{aligned}&(ctr \mapsto val : a_0 \ltimes ctr \mapsto val : b_0 \wedge (a_0 + 1 = b_0))\\ &\Rightarrow (ctr \mapsto val : a_0) \ltimes (ctr \mapsto val : (1 + a_0) * True)\end{aligned}$$

and we can rewrite the left hand side of the implication to

$$\begin{aligned}&(ctr \mapsto val : a_0 \ltimes ctr \mapsto val : 1 + a_0)\\ &\Rightarrow (ctr \mapsto val : a_0) \ltimes (ctr \mapsto val : (1 + a_0) * True)\end{aligned}$$

This implication holds because of the following rule for actions:

$$\frac{p \Rightarrow p' \qquad q \Rightarrow q'}{p \ltimes q \Rightarrow p' \ltimes q'}$$

and also because

$$\begin{aligned}&ctr \mapsto val : (1 + a_0)\\ &\quad\Rightarrow ctr \mapsto val : (1 + a_0) * Emp\\ &\quad\Rightarrow ctr \mapsto val : (1 + a_0) * True\end{aligned}$$

$\square$

$\boxed{\textbf{Proof of 3.4.2e}}$ We want to show

$$(\exists a \cdot ctr \mapsto val : a) \vee (\exists a, b \cdot ctr \mapsto val : b \wedge (a + 1 = b)) \Rightarrow \exists v \cdot ctr \mapsto v * True$$

Proof: We show that the two disjuncts individually imply the right hand side. First:

$$\begin{aligned}&\exists a \cdot ctr \mapsto val : a\\ &\Rightarrow \exists v \cdot ctr \mapsto val : v\\ &\Rightarrow \exists v \cdot ctr \mapsto val : v * Emp\\ &\Rightarrow \exists v \cdot ctr \mapsto val : v * True\end{aligned}$$

Also:

$$\begin{aligned}
&\exists a, b \cdot ctr \mapsto val : b \wedge (a + 1 = b) \\
&\Rightarrow \exists b \cdot ctr \mapsto val : b \Rightarrow \exists v \cdot ctr \mapsto val : v \\
&\Rightarrow \exists v \cdot ctr \mapsto val : v * Emp \\
&\Rightarrow \exists v \cdot ctr \mapsto val : v * True
\end{aligned}$$

□

$\boxed{\textbf{Proof of 3.4.2f}}$ We want to show

$$\exists v \cdot ctr \mapsto v \rhd \exists a \cdot ctr \mapsto val : a \ltimes ctr \mapsto val : a$$

and

$$\exists v \cdot ctr \mapsto v \rhd \exists a \cdot ctr \mapsto val : a \ltimes ctr \mapsto val : (1 + a)$$

Proof: Recall that $I \rhd a$ holds iff the following conditions hold:

(i) $[I] \Rightarrow a$

(ii) $a \Rightarrow I \ltimes I$

(iii) $precise(I)$

Hence we show the following:

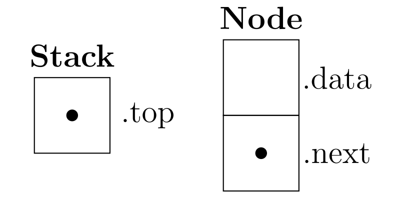
(i) $[\exists v \cdot ctr \mapsto val : v] \Rightarrow \exists v \cdot ctr \mapsto val : v \ltimes ctr \mapsto val : v$
Proof: this is trivial because $[\exists v \cdot ctr \mapsto val : v] \equiv \exists v \cdot ctr \mapsto val : v \ltimes ctr \mapsto val : v$

(ii)

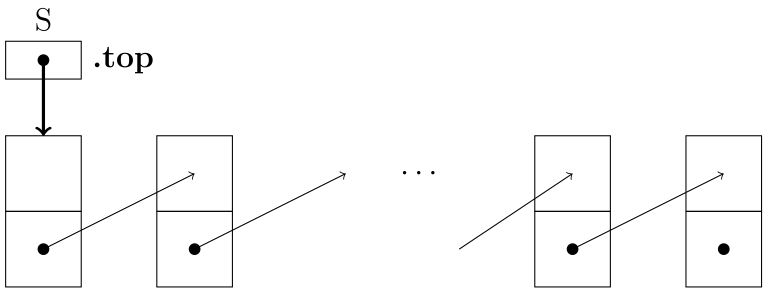
$$\begin{aligned}
&\exists v \cdot ctr \mapsto val : v \ltimes ctr \mapsto val : v \\
&\Rightarrow \exists v \cdot ctr \mapsto val : v \ltimes ctr \mapsto val : v \ltimes \exists v \cdot ctr \mapsto val : v \ltimes ctr \mapsto val : v
\end{aligned}$$

Proof : this follows immediately from the lemma $\exists x \cdot Px \ltimes Qx \equiv \exists x \cdot Px \ltimes \exists x \cdot Qx$.

(iii) $precise(\exists v \cdot ctr \mapsto val : v)$
Proof: Follows from a standard result in separation logic showing that $E \mapsto F$ is a precise assertion, and so is $\exists F \cdot E \mapsto F$. □



(a) A stack record and a node record



(b) Top of stack pointer

Figure 3.9: Stack data structure

3.4.4 Example proof 2: a linked-list stack

In this section we treat a simple linked-list stack data structure. Firstly we define two kinds of records: a stack record and a node record. A stack record has only one field, called "top", which is a pointer to a node record. The node record has two fields: *val*, which is a value of some type T, and *next*, which is a pointer to a node (figure 3.9(a)). The *top* field of a stack points to the "top of stack" and is the only way to access the data structure (figure 3.9(b)). We show the listings for the initialisation and the push and the pop operations. To support concurrent access, the operations enclose all updates to the *top* pointer within an atomic block.

Listing 3.3: Initialising the stack

```
init(S:Stack) { top(S) := nil }
```

Listing 3.4: Push operation for the stack

```
push (S:Stack, v: T) {
  local n := new [data: v, next: nil];
  atomic(true) {
    next(n) := top(S);
    top(S) := n
  }
  return true;
}
```

Listing 3.5: Pop operation for the stack

```
pop(S): T {
  local l, n;
  atomic(true) {
    if(top(S) = nil) {
      l := nil;
    } else {
    l := val(top(S));
    n := next(top(S));
    top(S):= n;
  }
  return l;
}
```

3.4.5 Linked list stack: RGSep proof

For our RGSep proofs we firstly want to define a predicate that asserts that a variable points to a stack data structure. This predicate will also define a correspondence between an abstract sequence of values and the values stored on the stack. For this proof it is convenient to define a mathematical sequence that provides a prepend operation. We let $\langle\rangle$ denote an empty sequence and $a^\frown s$ denote prepending the value a to the sequence s such that the head of the sequence is $hd(a^\frown s) = a$ and the tail is $tl(a^\frown s) = s$. Given this we can define a predicate $Stack(S, \sigma)$ that asserts that S is a stack with contents modelled by the sequence σ. The predicate is a disjunction of two assertions: the first disjunct holds of an empty stack, and the second disjunct holds of a non-empty stack:

$$\begin{aligned} stack(S,\sigma) \triangleq & (\sigma = \langle\rangle) \wedge top(S) = nil \\ & \vee \\ & (\exists a, as \cdot \sigma = a^\frown as) \wedge \exists t \cdot top(S) \mapsto t * \exists u \cdot lseg(a^\frown as)(t,u) \end{aligned} \tag{3.4.3}$$

where $lseg\ \alpha\ (i,j)$ is the list segment assertion from separation logic (see e.g., chapter 4 of the lecture notes from [33]): it asserts that there is a list segment between locations i and j that contains a list segment α and is defined inductively as follows:

$$\begin{aligned} lseg\ \langle\rangle\ (i,j) &\triangleq emp \wedge i = j \\ lseg\ (a^\frown as)\ (i,k) &\triangleq \exists j \cdot i \mapsto a, j * lseg\ as\ (i,k) \end{aligned} \tag{3.4.4}$$

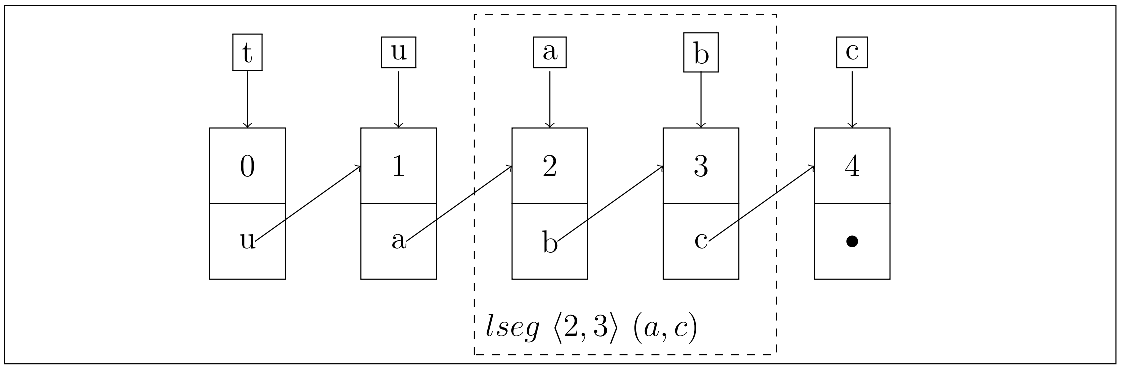


Figure 3.10: A list segment (bounded by the dashed area) satisfying the assertion shown

Note that the right endpoint j in $lseg\ \alpha\ (i, j)$ is the value of the next pointer of the last node in the list segment, and is not actually contained in the segment. This is illustrated in figure 3.10. Having defined the stack invariant, we now proceed with the RGSep proof. First we define the actions that correspond to the updates to the shared stack that are performed by the $push(S, v)$ and $pop(S)$ actions. Note that we have defined two *pop* actions – one corresponding to popping an empty stack and another to a non-empty stack.

$$push(S, v) \triangleq stack(S, \sigma) \leadsto stack(S,\ v^\frown\sigma) \tag{3.4.5}$$

$$pop_E(S) \triangleq stack(S, \langle\rangle) \leadsto stack(S,\ \langle\rangle) \tag{3.4.6}$$

$$pop_{NE}(S) \triangleq stack(S,\ a^\frown\sigma) \leadsto stack(S,\ \sigma)) \tag{3.4.7}$$

The interpretations of these actions are straightforward: $push(S, v)$ updates the shared stack such that if in the pre-state the stack contains sequence σ, then in the post-state the value v is prepended onto it by the push action. Popping an empty stack leaves the stack unchanged, while popping a non-empty stack updates the stack such that the post-state contains the tail of the sequence from the pre-state.

Proof of $push(S, v)$ **in RGSep.** We want to prove that $push(S, v)$ satisfies its specification:

$$\vdash push(S,\ v)\ \underline{sat}\ (P, R, G, Q)$$

where

$$P \triangleq emp * \boxed{stack(S,\ \sigma)} \tag{3.4.8}$$

$$R \triangleq \exists \sigma \cdot stack(S,\ \sigma) \rightsquigarrow stack(S,\ \sigma) \tag{3.4.9}$$

$$G \triangleq \exists \sigma, v \cdot stack(S,\ \sigma) \rightsquigarrow stack(S,\ v^\frown\sigma) \tag{3.4.10}$$

$$Q \triangleq emp * \boxed{stack(S,\ v^\frown\sigma)} \tag{3.4.11}$$

In the precondition P and postcondition Q, emp is the local state and the boxed assertion is the shared state. The rely condition is an action that preserves the stack invariant, and the guarantee condition is an action that allows for values to be prepended to the sequence of values contained in the stack, while preserving the stack invariant. We annotate the push code as shown below:

```
{$\boxed{stack(S,\ \sigma)}$}
push (S:Stack, v: T) {
  local n := **new** [data: v, next: nil];
  {$n \mapsto v,\ nil * \boxed{stack(S,\ \sigma)}$}
  **atomic**(true) {
  {$n \mapsto v,\ nil * \boxed{stack(S,\ \sigma)}$}

  $\left\{ n \mapsto v,\ nil \;*\; \boxed{\begin{array}{c} \sigma = \langle\rangle \wedge top(S) = nil \\ \vee \\ \exists a, as \cdot \sigma = a^\frown as \wedge \exists t \cdot top(S) \mapsto t * \exists u \cdot lseg\ a^\frown as\ (t,\ u) \end{array}} \right\}$

  $\left\{ \begin{array}{l} n \mapsto v,\ nil \;*\; \boxed{\sigma = \langle\rangle \wedge top(S) = nil} \\ \vee \\ n \mapsto v,\ nil \;*\; \boxed{\exists a, as \cdot \sigma = a^\frown as \wedge \exists t \cdot top(S) \mapsto t * \exists u \cdot lseg\ a^\frown as\ (t,\ u)} \end{array} \right\}$

  $\left\{ \begin{array}{l} n \mapsto v,\ nil \;*\; \boxed{\sigma = \langle\rangle \wedge top(S) = nil} \\ \vee \\ n \mapsto v,\ nil \;*\; \boxed{\exists a, as \cdot \sigma = a^\frown as \wedge \exists t \cdot top(S) \mapsto t * \exists u \cdot \exists u' \cdot t \mapsto a, u' * lseg\ as\ (u',\ u)} \end{array} \right\}$

  $\left\{ \begin{array}{l} n \mapsto v,\ nil \;*\; \boxed{\sigma = \langle\rangle \wedge top(S) = nil} \\ \vee \\ n \mapsto v,\ nil \;*\; \boxed{\exists a, as \cdot \sigma = a^\frown as \wedge top(S) \mapsto t * t \mapsto a, u' * lseg\ as\ (u',\ u)} \end{array} \right\}$
    next(n) := top(S);
```

$$\left\{\begin{array}{l} n \mapsto v,\ nil \ \ * \ \boxed{\sigma = \langle\rangle \wedge top(S) = nil} \\ \qquad\qquad\quad \vee \\ n \mapsto v,\ t \ \ * \ \boxed{\exists a, as \cdot \sigma = a \frown as \wedge top(S) \mapsto t * t \mapsto a, u' * lseg\ as\ (u',\ u)} \end{array}\right\}$$

```
top(S) := n
```

$$\left\{\begin{array}{c} \boxed{\sigma = \langle\rangle \wedge top(S) \mapsto n * n \mapsto v,\ nil} \\ \vee \\ \boxed{\exists a, as \cdot \sigma = a \frown as \wedge top(S) \mapsto n * n \mapsto v,\ t * t \mapsto a,\ u' * lseg\ as\ (u',\ u)} \end{array}\right\}$$

$$\left\{\begin{array}{c} \boxed{top(S) \mapsto n * lseg\ v \frown \langle\rangle\ (n,\ nil)} \\ \vee \\ \boxed{\exists a, as \cdot \sigma = a \frown as \wedge top(S) \mapsto n * lseg\ v \frown \sigma\ (n,\ u)} \end{array}\right\}$$

$\{\boxed{stack(S, v \frown \sigma)}\}$

```
    }
  return true;
}
```

$\{\boxed{stack(S,\ v \frown \sigma)}\}$

Proof details:
For the $push(S, v)$ operation, the main part of the proof involves the ATOMIC rule (figure 3.4), which gives the following proof obligations:

$$\vdash \texttt{next(n) := top(S); top(S) := n}\ \underline{sat}\ (P, \emptyset, \emptyset, Q) \tag{3.4.12a}$$

$$(P \rightsquigarrow Q) \subseteq G \tag{3.4.12b}$$

$$\boxed{Q}\ \texttt{stable under}\ R \tag{3.4.12c}$$

where

$$P \triangleq n \mapsto v,\ nil * \boxed{\exists a, as \cdot \sigma = a \frown as \wedge top(S) \mapsto t * t \mapsto a, u' * lseg\ as\ (u', u)}$$

$$Q \triangleq \boxed{stack(s, v \frown \sigma)}$$

$$R = \exists \sigma \cdot stack(s,\ \sigma) \rightsquigarrow stack(S,\ \sigma)$$

$$G = R$$

Proof of 3.4.12a. In the absence of rely and guarantee conditions, this is just a (sequential) proof of the body of the atomic block, which is shown in the proof outline. {*End of proof.*}

Proof of 3.4.12b. We want to show that

$$stack(S,\ \sigma) \rightsquigarrow stack(S, v^\frown\ \sigma) \subseteq stack(S,\ \sigma) \rightsquigarrow stack(S, v^\frown\ \sigma)$$

This follows immediately from the rule G-AX in figure 3.5. {*End of proof.*}

Proof of 3.4.12c. We want to show that

$$\boxed{stack(S, v^\frown \sigma)}\ \texttt{stable under}\ stack(S,\sigma) \rightsquigarrow stack(S,\sigma)$$

Recall from section 3.2.3 that for an assertion S and a rely condition $R \triangleq P \rightsquigarrow Q$, stability of S with respect to R is defined as

$$S; [\![P \rightsquigarrow Q]\!] \Rightarrow S \iff (P -\!\circledast\ S) * Q \Rightarrow S$$

Hence we must show that

$$(stack(S,\sigma) -\!\circledast\ stack(S, v^\frown \sigma)) * stack(S,\sigma) \Rightarrow stack(S, v^\frown \sigma)$$

The intuition behind this is that since the environment preserves the stack invariant $\exists\sigma.stack(S,\ \sigma)$ it necessarily preserves the postcondition, which is just another form of the stack invariant. To show this in detail, one must unwind the definition of the stack invariant and use properties of septraction ($-\!\circledast$) operator.

Proof of *pop*(S) in RGSep. We want to prove that *pop*(S) satisfies its specification:

$$\vdash pop(S)\ \underline{sat}\ (P, R, G, Q) \tag{3.4.13}$$

where

$$P \triangleq emp * \boxed{stack(S,\ \sigma)} \tag{3.4.14}$$

$$Q \triangleq emp * \boxed{stack(S,\ \sigma)} \tag{3.4.15}$$

$$R \triangleq \exists\sigma \cdot stack(S,\ \sigma) \rightsquigarrow stack(S,\ \sigma) \tag{3.4.16}$$

$$R \triangleq \exists\sigma \cdot stack(S,\ \sigma) \rightsquigarrow stack(S,\ \sigma) \tag{3.4.17}$$

The proof outline for *pop*(S) is shown below:

$\{\boxed{stack(S,\ \sigma)}\}$
```
pop(S): T {
  local l, n;
```

```
atomic(true) {
  $\{\boxed{stack(S,\ \sigma)}\}$

  $\left\{\begin{array}{c}\boxed{\sigma = \langle\rangle \wedge top(S) = nil} \\ \vee \\ \boxed{\exists a, as \cdot \sigma = a\frown as \wedge \exists t \cdot top(S) \mapsto t * \exists u \cdot lseg\ a\frown as\ (t,\ u)}\end{array}\right\}$

  if(top(S) = nil) {
  $\{\boxed{\sigma = \langle\rangle \wedge top(S) = nil}\}$
    l := nil;
  $\{\boxed{\sigma = \langle\rangle \wedge top(S) = nil}\}$

  $\left\{\begin{array}{c}\boxed{\sigma = \langle\rangle \wedge top(S) = nil} \\ \vee \\ \boxed{\exists a, as \cdot \sigma = a\frown as \wedge \exists t \cdot top(S) \mapsto t * \exists u \cdot lseg\ a\frown as\ (t,\ u)}\end{array}\right\}$

  $\{\boxed{stack(S,\ \sigma)}\}$

  } else {
  $\{\boxed{\exists a, as \cdot \sigma = a\frown as \wedge \exists t \cdot top(S) \mapsto t * \exists u \cdot lseg\ a\frown as\ (t,\ u)}\}$

  $\{\boxed{\exists a, as \cdot \sigma = a\frown as \wedge \exists t \cdot top(S) \mapsto t * \exists u, u' \cdot t \mapsto a,\ u' * lseg\ as\ (u',\ u)}\}$
  l := val(top(S));

  $\{\boxed{\exists a, as \cdot \sigma = a\frown as \wedge \exists t \cdot top(S) \mapsto t * \exists u, u' \cdot t \mapsto a,\ u' * lseg\ as\ (u',\ u)} * l = a\}$

  $\{\boxed{\exists a, as \cdot \sigma = a\frown as \wedge top(S) \mapsto t * t \mapsto a,\ u' * lseg\ as\ (u',\ u)} * l = a\}$

  n := next(top(S));
  $\{\boxed{\exists a, as \cdot \sigma = a\frown as \wedge top(S) \mapsto t * t \mapsto a,\ u' * lseg\ as\ (u',\ u)} * l = a * n \mapsto u'\}$

  top(S):= n;

  $\{\boxed{\exists a, as \cdot \sigma = a\frown as \wedge top(S) \mapsto u' * lseg\ as\ (u',\ u)} \wedge l = a * n \mapsto u' * t \mapsto a,\ u'\}$
```

```
      {$\boxed{stack(S,\ tl(\sigma))} * l = a * n \mapsto u' * t \mapsto a,\ u'$}
    }
    return l;
}
{$\boxed{stack(S,\ tl(\sigma))}$}
```

Proof details:
The form of the $pop(S)$ operation is just an atomic block, similar to the $push(S,\ v)$ operations, hence the proof obligations are similar:

$$\vdash C\ \underline{sat}\ (P, \emptyset, \emptyset, Q) \tag{3.4.18a}$$
$$(P \rightsquigarrow Q) \subseteq G \tag{3.4.18b}$$
$$\boxed{Q}\ \texttt{stable under}\ R \tag{3.4.18c}$$

where

$$\begin{aligned}
C &\triangleq (the\ body\ of\ the\ atomic\ block)\\
P &\triangleq \boxed{stack(S, \sigma)}\\
Q &\triangleq \boxed{stack(S, tl(\sigma))}\\
R &= \exists \sigma \cdot stack(S,\ \sigma) \rightsquigarrow stack(S,\ \sigma)\\
G &= \exists \sigma \cdot stack(S,\ \sigma) \rightsquigarrow stack(S,\ tl(\sigma))
\end{aligned}$$

3.4.6 Linked list stack: LRG proof

For the proofs of $push(S,\ v)$ and $pop(S)$ in LRG, we can make use of the same stack invariant that we defined for the RGSep proofs. However for LRG, in defining the rely and guarantee conditions R, G we also need to find a "fencing invariant" I such that $I \rhd R$ and $I \rhd G$. In this case we choose $R \triangleq stack(S,\ \sigma) \ltimes stack(S,\ \sigma)$, i.e we rely on the environment to preserve the stack invariant, and the guarantee conditions are the same as for the RGSep proofs, i.e. we have

$$G_{push(S,v)} \triangleq a_{push} \triangleq stack(S,\ \sigma) \ltimes stack(S, v^\frown\ \sigma) \tag{3.4.19}$$
$$G_{pop(S)} \triangleq a_{pop} \triangleq stack(S,\ \sigma) \ltimes stack(S,\ tl(\sigma)) \tag{3.4.20}$$

We choose the fencing invariant $I \triangleq \mathit{stack}(S,\ \sigma)$ and for $p \in \{R, G_{push}, G_{pop}\}$ we can show formally that $I \rhd p$ by showing that the following conditions hold:

$$[\mathit{stack}(S,\ \sigma)] \Rightarrow a_{(push,pop)} \tag{3.4.21}$$
$$a_{(push,pop)} \Rightarrow \mathit{stack}(S,\ \sigma) \ltimes \mathit{stack}(S,\ \sigma) \tag{3.4.22}$$
$$\mathit{precise}(\mathit{stack}(S,\ \sigma)) \tag{3.4.23}$$

The proof outlines will be the same as with RGSep, and the applicable inference rule for both operations is the ATOMIC rule. The proof obligations that are unique to LRG are as follows:
For the proof of $\mathit{push}(S, v)$:

$$\mathit{Sta}(\{\mathit{stack}(S,\ \sigma), \mathit{stack}(S,\ v^\frown\sigma)\}, \mathit{stack}(S,\ \sigma) \ltimes \mathit{stack}(S,\ \sigma) * \mathit{Id}) \tag{3.4.24}$$
$$\mathit{stack}(S,\ \sigma \ltimes \mathit{stack}(S,\ v^\frown\sigma) \Rightarrow \mathit{stack}(S,\ \sigma \ltimes \mathit{stack}(S,\ v^\frown\sigma) * \mathit{True} \tag{3.4.25}$$
$$\mathit{stack}(S,\ \sigma) \vee \mathit{stack}(S,\ v^\frown\sigma) \Rightarrow \mathit{stack}(S,\ \sigma) * \mathit{True} \tag{3.4.26}$$
$$\mathit{stack}(S,\ \sigma) \rhd \mathit{stack}(S,\ \sigma) \ltimes \mathit{stack}(S,\ \sigma) \tag{3.4.27}$$
$$\mathit{stack}(S,\ \sigma) \rhd \mathit{stack}(S,\ \sigma) \ltimes \mathit{stack}(S,\ v^\frown\sigma) \tag{3.4.28}$$
$$\tag{3.4.29}$$

For the proof of $\mathit{pop}(S)$:

$$\mathit{Sta}(\{\mathit{stack}(S,\ \sigma), \mathit{stack}(S,\ \mathit{tl}(\sigma))\}, \mathit{stack}(S,\ \sigma) \ltimes \mathit{stack}(S,\ \sigma) * \mathit{Id}) \tag{3.4.30}$$
$$\mathit{stack}(S,\ \sigma \ltimes \mathit{stack}(S,\ \mathit{tl}(\sigma)) \Rightarrow \mathit{stack}(S,\ \sigma \ltimes \mathit{stack}(S,\ \mathit{tl}(\sigma)) * \mathit{True} \tag{3.4.31}$$
$$\mathit{stack}(S,\ \sigma) \vee \mathit{stack}(S,\ \mathit{tl}(\sigma)) \Rightarrow \mathit{stack}(S,\ \sigma) * \mathit{True} \tag{3.4.32}$$
$$\mathit{stack}(S,\ \sigma) \rhd \mathit{stack}(S,\ \sigma) \ltimes \mathit{stack}(S,\ \sigma) \tag{3.4.33}$$
$$\mathit{stack}(S,\ \sigma) \rhd \mathit{stack}(S,\ \sigma) \ltimes \mathit{stack}(S,\ \mathit{tl}(\sigma)) \tag{3.4.34}$$
$$\tag{3.4.35}$$

3.4.7 Discussion of proofs

The main difference between the LRG and RGSep proofs so far have been in the treatment of the shared state: RGSep divides the shared and local states explicitly, but LRG does not, and requires a "fencing invariant" to specify the boundaries of an action on the shared state. Otherwise we can see that the idea of the RGSep proof and the LRG proof are the essentially the same. However, LRG has more proof obligations for the atomic block than RGSep, and the latter are all covered by the former. The proof obligations that are required specifically by LRG stem from 1) LRG's use of "variables as resource" logic (the first proof obligation) and 2) the fact in LRG there is no syntactic distinction between shared and local state assertions, and the fencing invariant I specifies the boundary between shared and local. The fifth proof obligation for LRG, which involves I is also a requirement for showing that the invariant of the counter holds before and after program execution (i.e. the invariant is implied by the pre- and post-conditions). The sixth proof obligation is to show that the rely and guarantee conditions are fenced by I.

Chapter 4

Data abstraction and weakest preconditions in separation logic

The main aim of this chapter is to illustrate how to do separation logic proofs in a calculational way using verification conditions derived from the weakest precondition semantics defined in section 2.4. For this we present fully worked-out examples of proofs of correctness of an array-based and linked-list sequential stack data structures. In addition, since the proofs are in essence data abstraction proofs showing a data structure implements an abstract data type, we relate it to a classic proof technique developed by Hoare. We first discuss that technique, and then note how it relates to sequential separation logic proofs, and then present the proofs.

4.1 Data abstraction technique due to Hoare

In a paper [16] written in 1972, C.A.R. Hoare presents a proof technique for showing that a data structure implementation correctly represents an abstract data type. In this section we present a slight adaptation of the ideas from that paper. Let T be some abstract data type. Suppose that T supports n operations $op_1, \ldots op_n$, through which the state of an instance of T may be queried or updated. We define an "abstract program" to be a program that operates on an instance t of T by calling one or more of its allowed operations in order to transform it. We say that the abstract program operates in abstract state space. Also suppose that the abstract type T has one one or more concrete implementations using concrete data structures; we call such implementations "concrete programs" operating on concrete state space. The essence of Hoare's proof method is to show some kind of simulation between a concrete program and the related abstract program. For example, let T be a mathematical set, and

$$
\begin{aligned}
t \cdot \mathit{init}() &\equiv t := \emptyset \\
t \cdot \mathit{insert}(i) &\equiv t := t \cup \{i\} \\
t \cdot \mathit{remove}(i) &\equiv t := t \cap \neg\{i\} \\
t \cdot \mathit{has}(i) &\equiv i \in t
\end{aligned}
$$

Figure 4.1: Set operations and their intended effects

let s be an instance of T. Then the set operations define some of the allowed operations on the abstract set:

- $s \cdot \mathit{insert}(i)$ – insert a value i into the set s
- $s \cdot \mathit{remove}(i)$ – remove the value i from the set s
- $s \cdot \mathit{has}(i)$ – test to see if the value i is contained in the set s

Let t be an instance of some abstract type T and let *op* be some operation defined on T. Assume that an invocation of an abstract operation of the form $t \cdot op(a_1, \ldots, a_n)$, where the a_k is a set of arguments, will transform t such that its resulting state is given by $\phi(t, a_1, \ldots a_n)$ for some function ϕ. Intuitively ϕ is a function that models the "intended effect" of *op*. Then the invocation $t \cdot op(a_1, \ldots, a_n)$ is equivalent to the assignment statement $t := \phi(t, a_1 \ldots, a_n)$. When this holds we say that *op* models ϕ. See figure **??** for examples. The invocation of *op* satisfies the following Hoare logic specification, where S is some state predicate:

$$\{S^t_{\phi(t,a_1,\ldots,a_n)}\}\ t \cdot op(a_1, \ldots, a_n)\ \{S\}$$

With these ideas, we may now define correctness.

Definition of correctness of data representations. A concrete representation of an abstract data type T is correct if every operation *op* of T models the intended function ϕ for that operation, and if the initialisation sequence for the concrete representation yields a value that models the corresponding abstract initial value. Formally this entails showing the validity of Hoare triples as above for each operation *op*. Consequently, a program operating on abstract variables may be validly replaced by one carrying out equivalent operations on the concrete representation.

Abstraction functions and representation invariants. A requirement for the proof technique is a formal definition of the relationship between the abstract state and the concrete state. This can be done by providing an *abstraction function* $\mathcal{A}(c_1, \ldots, c_n)$ which maps concrete variables c_k to the abstract variables which they represent. For example, suppose that an array `A` with 1-based indexing is used to represent a set of integers; then the function

$$\mathcal{A}(m, \mathtt{A}) = \{i : \mathtt{integer} | \exists k \cdot 1 \leq k \leq m \wedge A[k] = i\}$$

maps the first m elements of the array to the abstract values in the set which the array represents. In general $\mathcal{A}$ will be a many-to-one function as there are many possible concrete representations of abstract values. If we then let t denote the abstract value $\mathcal{A}(c_1, \ldots, c_n)$ before the execution of *op* then we must prove that after execution, the following holds:

$$\mathcal{A}(c_1, \ldots, c_n) = \phi(t, v_1, \ldots, v_n)$$

where the v_k are the formal parameters of *op*. In terms of Hoare logic, we need to prove the following triple:

$$\{t = \mathcal{A}(c_1, \ldots, c_n)\}\ op\ \{\mathcal{A}(c_1, \ldots, c_n) = \phi(t, v_1, \ldots, v_n)\}$$

In addition to abstraction functions, another requirement for the proof method is the *representation invariant*. This is a condition $I(c_1, \ldots, c_n)$ that defines some relationship between the concrete variables, thus constraining the possible combinations of values which they may take. Each operation (except initialisation) may assume that I holds when the operation is entered, and must then ensure that I holds on completion. In the running example of the set data type, the correctness of the `remove()` operation depends on the array elements $\mathtt{A}[k]$ being all distinct. This can be expressed by the following representation invariant:

$$I \equiv size(\mathcal{A}(m, \mathtt{A}) = m$$

4.2 Relation to separation logic

Here we discuss how the data abstraction technique can be incorporated with sequential separation logic.

We note that we can generalise Hoare's abstraction functions to abstraction relations, and then use the assertion language of separation logic to specify these relations. This is based on the observation that a separation logic assertion can be read as a kind of specification of a relationship between concrete states (program variables) and abstract (mathematically-specified) states. Indeed in his 2002 paper on separation logic [34], Reynolds remarks that in order to specify a program adequately, it is usually necessary to describe more than the form of its structures or the sharing patterns between them; one must relate the states of the program to the abstract values that they denote. He gives the example of a list-reversal program – "to specify such a program, it would hardly be enough to say that if the heap location denoted by i is a list before execution, then the heap location j will be a list afterwards. One needs to say that if i is a list representing the sequence α before execution, then afterwards j will be a list representing the sequence that is the reflection of α. To do this, it is necessary to define the set of abstract values (sequences, in this case), along with their primitive operations, and then to define predicates on the abstract values by structural induction."

Given this we can view the following inductively defined predicate expressing that some heap location points to a structure that models an abstract sequence as a kind of abstraction relation:

$$\mathit{list}\ \epsilon\ i \triangleq \mathit{emp} \wedge i = \ \mathit{nil} \tag{4.2.1}$$

$$\mathit{list}\ (a \cdot \alpha)\ i \triangleq \exists j \cdot i \mapsto a, j *\ \mathit{list}\ \alpha\ J \tag{4.2.2}$$

We then redefine the proof obligations for each operation op of an abstract data type instance t to be the following set of Hoare triples:

$$\{\mathcal{A}(t, \overline{c})\}\ t \cdot op(a_1, \ldots, a_n)\ \{\mathcal{A}(\phi(t), \overline{c})\}$$

where $\mathcal{A}$ is an abstraction relation and $\overline{c}$ denotes a list of concrete variables.

In the next sections we illustrate these ideas, as well as the use of the weakest precondition calculus, in a proof of correctness of stack data structure.

4.3 Proof of correctness of an array-based sequential stack

First we define the algorithm and its data structures. Here is a simple sequential last-in first-out (LIFO) stack that uses an array to store its values. In order to ensure the LIFO discipline, a value can only be inserted to the designated top of the stack and likewise, a value can only be removed from the top of stack. Let us just assume for now that the values are integers. The data structure consists of an integer array S, indexed from 0, and an integer top which stores the index of the top of the stack. There are three operations: *init* for initializing the structure, *push* for inserting a value on to the top of the stack, and *pop* for removing a value from the top of the stack.

The operation *init* firstly allocates space for a new array and sets the top of stack pointer *top* to -1, which indicates that the stack is initially empty.

Listing 4.1: Initialisation code for the stack

```
def init ==
  S = new int[N]; top := -1
end
```

We define the return values of the *push*() and *pop*() operations to be members of the type $\mathbb{Z} \cup \{FULL, EMPTY\}$ where $FULL$ and $EMPTY$ are special values denoting a full stack and an empty stack respectively. We also use a special variable *ret* to indicate the return value. The operation *push*(v) inserts a value v of type integer to the stack. If the stack is full, it sets *ret* to $FULL$. Otherwise, it increments *top* to point to the next available slot, writes the value to that slot, and sets *ret* to v.

Listing 4.2: Pushing a value onto the stack

```
def push(v) ==
  if top + 1 == N then
  ret := FULL
  else
    top := top + 1;
    S[top] := v;
    ret := v;
  endif
end
```

The *pop* operation sets *ret* to $EMPTY$ when the stack is empty; otherwise it saves the value of $S[top]$ to a temporary variable, decrements *top* and returns the temporary variable.

Listing 4.3: Popping the stack

```
def pop() ==
  if top == -1
  then ret := EMPTY
  else
    val := S[top];
    top := top - 1;
    ret := val;
    endif
end
```

We verify the algorithm using a combination of separation logic and Hoare's data refinement technique. Following the latter technique, we show that the concrete stack implementation given above is a correct data representation of a stack, i.e., the procedures defined above model the intended abstract stack operations. Abstractly we model a stack as a finite sequence σ of values. We then show that the procedures of the data structure we have defined above correspond to the following operations on the sequence σ:

$$S.init() \equiv \sigma := \epsilon \tag{4.3.1a}$$
$$S.push(v) \equiv \sigma := append(\sigma, v) \tag{4.3.1b}$$
$$S.pop() \equiv \sigma := butlast(\sigma) \tag{4.3.1c}$$

i.e. $init()$ results in a state that corresponds to the empty sequence ϵ, $push(v)$ models appending a value v to σ, and $pop()$ models removing the last element from σ. To show this we first define an abstraction function, i.e. a function that maps the set of concrete variables to the abstract state that it represents. For this data structure we know intuitively that the abstraction function maps the contents of the section of the array S from index 0 through to index top to the value of the sequence σ. Hence we want to define a function that turns an array section into a sequence. Following a standard technique in the literature (see e.g. [14]) we can view an array as a function from indexes to values (and consequently, array subscripting is just function application of an array to an index to obtain a value). In this instance the set of indexes is the set $I_{top} = \{i | 0 \leq i \leq top\}$. We can then define the required abstraction function as

$$\mathcal{A}(S, top) = map\ S\ I_{top} \tag{4.3.2}$$

where map is a higher-order function that takes a function and a sequence returns a new sequence which is the result of applying the function to every member of the input sequence:

$$map = \lambda f \cdot \lambda L \cdot if\ L = \epsilon\ then\ \epsilon\ else\ (f\ (head\ L))\#(map\ f\ (tail\ L)) \tag{4.3.3}$$

In order to show (4.3.1a) it suffices to show that the following Hoare triple is true:

$$\{True\}\ \text{S.init()}\ \{\mathcal{A}(S, top) = \epsilon \wedge top < N\} \tag{4.3.4}$$

This means that the initialisation procedure establishes a concrete state that maps to the desired abstract state (the empty sequence ϵ). The additional conjunct that must be shown, i.e., $top < N$ is a structural invariant which expresses

the constraint that the top pointer never goes beyond the bounds of the array.
To show (4.3.1b), we must establish the triple

$$\{\mathcal{A}(S, top) = \sigma \wedge top < N\}$$
S.push(v)
$$\{((ret = FULL \wedge \mathcal{A}(S, top) = \sigma) \vee (ret = v \wedge \mathcal{A}(S, top) = append(\sigma, v)) \wedge (top < N)\} \tag{4.3.5}$$

To show (4.3.1c), we must establish the triple

$$\{\mathcal{A}(S, top) = \sigma \wedge top < N\}$$
S.pop()
$$\{((ret = EMPTY \wedge \mathcal{A}(S, top) = \sigma) \vee (ret = v \wedge \mathcal{A}(S, top) = butlast(\sigma)) \wedge (top < N)\} \tag{4.3.6}$$

Proof of 4.3.4

To prove (4.3.4) we must show that $True \implies wp(init(), \mathcal{A}(S, top) = \epsilon \wedge top < N)$.
The wp calculation is as follows:

$$\begin{aligned}
&wp(\text{S.init()}, \mathcal{A}(S, top) = \epsilon \wedge top < N)\\
&= wp(\text{A := new int[N]; top := -1}, \mathcal{A}(S, top) = \epsilon \wedge top < N)\\
&= wp(\text{A := new int[N]}, \mathcal{A}(S, -1) = \epsilon \wedge -1 < N)\\
&= wp(\text{A := new int[N]}, \mathcal{A}(S, -1) = \epsilon \wedge True)\\
&= wp(\text{A := new int[N]}, map\ S\ \emptyset = \epsilon)\\
&= wp(\text{A := new int[N]}, \epsilon = \epsilon)\\
&= wp(\text{A := new int[N]}, True)\\
&= True
\end{aligned}$$

{*End of proof.*}

Proof of 4.3.5

Let $POST \triangleq ((ret = FULL \wedge \mathcal{A}(S, top) = \sigma) \vee (ret == v \wedge \mathcal{A}(S, top) = append(\sigma, v)) \wedge top < N$. Since the body of the push() procedure is a two-armed conditional, the weakest precondition has the form

$$\begin{aligned}
&wp(\text{push(v)},\ POST) =\\
&top == N - 1 \implies wp(\text{ret := FULL},\ POST)\\
&\wedge\\
&\neg(top == N - 1) \implies wp(\text{top := top + 1; S[top] := v; ret := True },\ POST)
\end{aligned}$$

Calculating the wp in the true branch:

$wp(\text{ret := FULL, } POST)$
$= wp(\text{ret := FULL}, ((ret = FULL \wedge \mathcal{A}(S, top) = \sigma) \vee (ret == v \wedge \mathcal{A}(S, top) = append(\sigma, v)) \wedge top <$
$= ((FULL = FULL \wedge \mathcal{A}(S, top) = \sigma) \vee (FULL = v \wedge \mathcal{A}(S, top) = append(\sigma, v)) \wedge top < N))$
$= ((True \wedge \mathcal{A}(S, top) = \sigma) \vee (False \wedge \mathcal{A}(S, top) = append(\sigma, v)) \wedge top < N))$
$= (\mathcal{A}(S, top) = \sigma \vee False) \wedge top < N$
$= \mathcal{A}(S, top) = \sigma \wedge top < N$

Calculating the wp in the false branch:

$wp(\text{top := top + 1; S[top] := v; ret := v},$
$((ret = FULL \wedge \mathcal{A}(S, top) = \sigma) \vee (ret == v \wedge \mathcal{A}(S, top) = append(\sigma, v)) \wedge top < N))$
$= wp(\text{top := top + 1; S[top] := v},$
$((v = FULL \wedge \mathcal{A}(S, top) = \sigma) \vee (v = v \wedge \mathcal{A}(S, top) = append(\sigma, v)) \wedge top < N))$
$\equiv wp(\text{top := top + 1; S[top] := v},$
$((False \wedge \mathcal{A}(S, top) = \sigma) \vee (True \wedge \mathcal{A}(S, top) = append(\sigma, v)) \wedge top < N))$
$\equiv wp(\text{top := top + 1; S[top] := v}, (False \vee \mathcal{A}(S, top) = append(\sigma, v) \wedge top < N)$
$\equiv wp(\text{top := top + 1; S[top] := v}, \mathcal{A}(S, top) = append(\sigma, v) \wedge top < N$
$\equiv wp(\text{top := top + 1}, \mathcal{A}(S \oplus \{top \mapsto v\}, top) = append(\sigma, v) \wedge top < N$
$\equiv \mathcal{A}(S \oplus \{top + 1 \mapsto v\}, top + 1) = append(\sigma, v) \wedge top + 1 < N$
$\equiv map\ (S \oplus \{top + 1 \mapsto v\})\ I_{top+1} = append(\sigma, v) \wedge top + 1 < N$

Hence the verification condition is

$\mathcal{A}(S, top) = \sigma \wedge top < N \Longrightarrow$
$(top = N - 1 \Longrightarrow \mathcal{A}(S, top) = \sigma \wedge top < N)\ \wedge$
$(\neg(top = N - 1) \Longrightarrow map\ (S \oplus \{top + 1 \mapsto v\})\ I_{top+1} = append(\sigma, v) \wedge top + 1 < N)$

The first conjunct on the RHS easily follows from the LHS. To show the second conjunct:

$\mathcal{A}(S, top) = \sigma \wedge top < N \Longrightarrow$
$(\neg(top = N - 1) \Longrightarrow map\ (S \oplus \{top + 1 \mapsto v\})\ I_{top+1} = append(\sigma, v) \wedge top + 1 < N)$

i.e.

$$\mathcal{A}(S, top) = \sigma \wedge top < N \wedge \neg(top = N - 1) \Longrightarrow$$
$$map\ (S \oplus \{top + 1 \mapsto v\})\ I_{top+1} = append(\sigma, v) \wedge top + 1 < N)$$

From the LHS it follows that $top < N - 1$ from which follows that $top + 1 < N$, establishing the second conjunct on the RHS. Also from the LHS it follows that $\mathcal{A}(S, top) = \sigma$ which rewrites to $map\ S\ I_{top} = \sigma$ hence we must show that

$$map\ S\ I_{top} = \sigma \Longrightarrow map\ (S \oplus \{top + 1 \mapsto v\})\ I_{top+1} = append(\sigma, v)$$

To show this note that the map expression of the consequent can be rewritten to

$$\begin{aligned} & map\ (S \oplus \{top + 1 \mapsto v\})\ I_{top+1} \\ & \quad \equiv (map\ S\ I_{top})\#(S \oplus \{top + 1 \mapsto v\})\ (top + 1)) \\ & \quad \equiv \sigma\#v \\ & \quad \equiv append(\sigma, v) \end{aligned}$$

{*End of proof.*}

Proof of 4.3.6

We shall omit the details of the proof of 4.3.6. The more interesting part involves showing the identity $map\ S\ I_{top-1} = butlast(\sigma)$ which holds assuming $map\ S\ I_{top} = \sigma$.

4.4 Linked data structures

In this section we shall treat linked-list data structures, beginning with stacks. We are mainly interested in finding out how to express assertions and invariants in separation logic and how to mechanise some parts of the proof using weakest preconditions.

4.4.1 Traversing a linked list

As a simple exercise we construct a proof of the following simple algorithm for traversing a linked list. Let L be a pointer to a linked list. Every node of a list is a record type that has a field *next* which points to the next node on the list. We write $next(t)$ to mean accessing the *next* field of node t:

Listing 4.4: Traversing a linked list

```
t := L;
while t != nil do
  t := next(t)
end
```

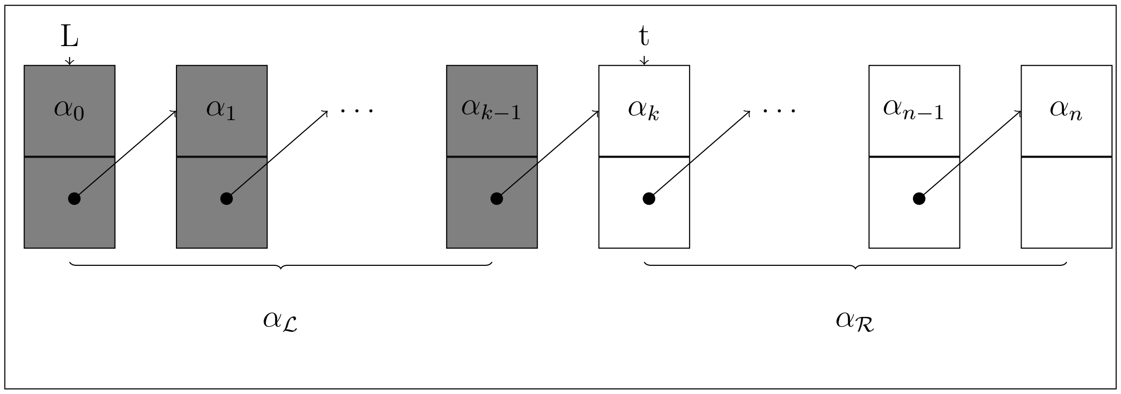


Figure 4.2: Idea for loop invariant: the pointer t conceptually separates the list into two list segments: the shaded region representing $\alpha_{\mathcal{L}}$ and unshaded region region $\alpha_{\mathcal{R}}$, with $\alpha = \alpha_{\mathcal{L}} \frown \alpha_{\mathcal{R}}$.

What are the pre- and post-conditions and loop invariant for this piece of code? The pre-condition should say that L points to a linked list and likewise for the postcondition, since the loop does not modify the list at all. We shall express this using an assertion in separation logic that describes singly-linked lists on the heap. We let α be a mathematical sequence and let ϵ be the empty sequence. We write $\textbf{list}\ \alpha\ (i,\ j)$ when there is a list segment from i to j that represents the sequence α. This is inductively defined on the structure of sequences:

$$\textbf{list}\ \epsilon\ (i,\ j) \triangleq \textbf{emp} \wedge i = j \tag{4.4.1a}$$

$$\textbf{list}\ a \cdot \alpha\ (i,\ j) \triangleq \exists k \cdot i \mapsto a, k * \textbf{list}\ \alpha\ (k,\ j) \tag{4.4.1b}$$

With these definitions we can define the pre- and post-conditions as

$$pre \triangleq post \triangleq \textbf{list}\ \alpha\ (L,\ nil) \tag{4.4.2}$$

Now we must define the loop invariant. Looking at the code, the list traversal is done by using a node pointer t which initially points to the same node that L points to, which is the head of the list $head(L)$. Then t is made to follow the *next* pointer of every node until it points to *nil*, at which point the loop terminates. Every time that the loop body executes, t points to a node on the list and one can think of it as partitioning the list into two list segments: the first segment is from the head to the node just before the one pointed by t, and the second segment from t to the end of the list. One can think of the first segment as representing a sequence $\alpha_{\mathcal{L}}$ and the second segment as representing a sequence $\alpha_{\mathcal{R}}$, where $\alpha_{\mathcal{L}} \frown \alpha_{\mathcal{R}} = \alpha$. This is illustrated in figure 4.4.1. This idea suggests the following loop invariant:

$$\exists \alpha_{\mathcal{L}} \cdot \exists \alpha_{\mathcal{R}} \cdot (\alpha = \alpha_{\mathcal{L}} \frown \alpha_{\mathcal{R}}) \wedge \textbf{list}\ \alpha_{\mathcal{L}}\ (L,\ t) * \textbf{list}\ \alpha_{\mathcal{R}}\ (t,\ nil) \tag{4.4.3}$$

The proof of the list traversal algorithm amounts to proving that 4.4.3 is indeed a loop invariant:

1. Show that the initialisation establishes (4.4.3)

$$\begin{array}{l}\{\ \textbf{list}\ \alpha\ (L,\ nil)\ \}\\ \text{t := L}\\ \{\exists\alpha_{\mathcal{L}}\cdot\exists\alpha_{\mathcal{R}}\cdot(\alpha=\alpha_{\mathcal{L}}\frown\alpha_{\mathcal{R}})\wedge\ \textbf{list}\ \alpha_{\mathcal{L}}\ (L,\ t)\ *\ \textbf{list}\ \alpha_{\mathcal{R}}\ (t,\ nil)\ \}\end{array} \tag{4.4.4}$$

2. Show that the loop body preserves (4.4.3)

$$\begin{array}{l}\{(\exists\alpha_{\mathcal{L}}\cdot\exists\alpha_{\mathcal{R}}\cdot(\alpha=\alpha_{\mathcal{L}}\frown\alpha_{\mathcal{R}})\wedge\ \textbf{list}\ \alpha_{\mathcal{L}}\ (L,\ t)\ *\ \textbf{list}\ \alpha_{\mathcal{R}}\ (t,\ nil)\ )\wedge t\neq nil\}\\ \text{t := next(t)}\\ \{\exists\alpha_{\mathcal{L}}\cdot\exists\alpha_{\mathcal{R}}\cdot(\alpha=\alpha_{\mathcal{L}}\frown\alpha_{\mathcal{R}})\wedge\ \textbf{list}\ \alpha_{\mathcal{L}}\ (L,\ t)\ *\ \textbf{list}\ \alpha_{\mathcal{R}}\ (t,\ nil)\ \}\end{array} \tag{4.4.5}$$

3. Show that the conjunction of (4.4.3) and the negation of the guard implies the postcondition.

$$\begin{array}{l}(\exists\alpha_{\mathcal{L}}\cdot\exists\alpha_{\mathcal{R}}\cdot(\alpha=\alpha_{\mathcal{L}}\frown\alpha_{\mathcal{R}})\wedge\ \textbf{list}\ \alpha_{\mathcal{L}}\ (L,\ t)\ *\ \textbf{list}\ \alpha_{\mathcal{R}}\ (t,\ nil)\ )\wedge t=nil\Longrightarrow\\ \textbf{list}\ \alpha\ (L,\ nil)\end{array} \tag{4.4.6}$$

Proof of 4.4.4

$$\begin{array}{l}wp(\text{t := L},\ \exists\alpha_{\mathcal{L}}\cdot\exists\alpha_{\mathcal{R}}\cdot(\alpha=\alpha_{\mathcal{L}}\frown\alpha_{\mathcal{R}})\wedge\ \textbf{list}\ \alpha_{\mathcal{L}}\ (L,\ t)\ *\ \textbf{list}\ \alpha_{\mathcal{R}}\ (t,\ nil)\ )\\ =\exists\alpha_{\mathcal{L}}\cdot\exists\alpha_{\mathcal{R}}\cdot(\alpha=\alpha_{\mathcal{L}}\frown\alpha_{\mathcal{R}})\wedge\ \textbf{list}\ \alpha_{\mathcal{L}}\ (L,\ L)\ *\ \textbf{list}\ \alpha_{\mathcal{R}}\ (L,\ nil)\\ =(\alpha=\epsilon\frown\alpha)\wedge\ \textbf{list}\ \epsilon\ (L,\ L)\ *\ \textbf{list}\ \alpha\ (L,\ nil)\\ =True\wedge\textbf{emp}*\ \textbf{list}\ \alpha\ (L,\ nil)\\ =\ \textbf{list}\ \alpha\ (L,\ nil)\end{array}$$

Hence the V.C. is just $\textbf{list}\ \alpha\ (L,\ nil)\ \Longrightarrow\ \textbf{list}\ \alpha\ (L,\ nil)$.
{*End of proof.*}

Proof of 4.4.5

{*Proof synopsis*}
First calculate the weakest precondition:

$$wp(\mathrm{t} := \mathrm{next(t)},\ \exists\alpha_{\mathcal{L}}\cdot\exists\alpha_{\mathcal{R}}\cdot(\alpha = \alpha_{\mathcal{L}} \frown \alpha_{\mathcal{R}}) \wedge\ \textbf{list}\ \alpha_{\mathcal{L}}\ (L,\ t)\ *\ \textbf{list}\ \alpha_{\mathcal{R}}\ (t,\ nil)\ )$$
$$= \exists\alpha_{\mathcal{L}}\cdot\exists\alpha_{\mathcal{R}}\cdot(\alpha = \alpha_{\mathcal{L}} \frown \alpha_{\mathcal{R}}) \wedge\ \textbf{list}\ \alpha_{\mathcal{L}}\ (L,\ next(t))\ *\ \textbf{list}\ \alpha_{\mathcal{R}}\ (next(t),\ nil)$$

and then prove the verification condition:

$$\exists\alpha_{\mathcal{L}}\cdot\exists\alpha_{\mathcal{R}}\cdot(\alpha = \alpha_{\mathcal{L}} \frown \alpha_{\mathcal{R}}) \wedge\ \textbf{list}\ \alpha_{\mathcal{L}}\ (L,\ t)\ *\ \textbf{list}\ \alpha_{\mathcal{R}}\ (t,\ nil)\ ) \wedge t \neq nil$$
$$\Longrightarrow$$
$$\exists\alpha_{\mathcal{L}}\cdot\exists\alpha_{\mathcal{R}}\cdot(\alpha = \alpha_{\mathcal{L}} \frown \alpha_{\mathcal{R}}) \wedge\ \textbf{list}\ \alpha_{\mathcal{L}}\ (L,\ next(t))\ *\ \textbf{list}\ \alpha_{\mathcal{R}}\ (next(t),\ nil)$$

To see why this is true, we instantiate the subsequences on the left side of the implication to $\alpha_{\mathcal{L}} := \lambda$ and $\alpha_{\mathcal{R}} := \rho$ and those on the right hand side to $\lambda \frown head(\rho)$ and $tail(\rho)$ respectively, and argue that the following property of sequences hold:

$$\alpha = \lambda \frown \rho \Longleftrightarrow \alpha = (\lambda \frown head(\rho)) \frown tail(\rho) \tag{4.4.7}$$

We also show that the following holds, assuming that $t \neq nil$:

$$\textbf{list}\ \lambda\ (L,\ t) * \textbf{list}\ \rho\ (t,\ nil)\ \Longrightarrow\ \textbf{list}\ (\lambda \frown head(\rho))\ (L,\ next(t)) * \textbf{list}\ (tail(\rho))\ (next(t),\ nil) \tag{4.4.8}$$

{*End of proof.*}

Proof of 4.4.6

$$(\exists\alpha_{\mathcal{L}}\cdot\exists\alpha_{\mathcal{R}}\cdot(\alpha = \alpha_{\mathcal{L}} \frown \alpha_{\mathcal{R}}) \wedge\ \textbf{list}\ \alpha_{\mathcal{L}}\ (L,\ t)\ *\ \textbf{list}\ \alpha_{\mathcal{R}}\ (t,\ nil)\ ) \wedge t = nil$$
$$\Longrightarrow \exists\alpha_{\mathcal{L}}\cdot\exists\alpha_{\mathcal{R}}\cdot(\alpha = \alpha_{\mathcal{L}} \frown \alpha_{\mathcal{R}}) \wedge\ \textbf{list}\ \alpha_{\mathcal{L}}\ (L,\ nil)\ *\ \textbf{list}\ \alpha_{\mathcal{R}}\ (nil,\ nil)$$
$$\Longrightarrow (\alpha = \alpha \frown \epsilon) \wedge\ \textbf{list}\ \alpha\ (L,\ nil)\ *\ \textbf{list}\ \epsilon\ (nil,\ nil)$$
$$\Longrightarrow True \wedge\ \textbf{list}\ \alpha\ (L,\ nil)\ * \textbf{emp}$$
$$\Longrightarrow\ \textbf{list}\ \alpha\ (L,\ nil)$$

{*End of proof.*}

4.4.2 Linked-list implementation of a stack

We present a stack as a singly-linked list of records called nodes where every node has two fields named *data* and *next*, and a pointer *top* to the head of the linked list. Initially *top* is set to *nil*. The *push*() and *pop*() operations are shown in the listings.

Listing 4.5: Linked list stack: push

```
def push(val)
  n := new node();
  data(n) := val;
  next(n) := top;
  top := n;
end
```

Listing 4.6: Linked list stack: pop

```
def pop()
  if (top == nil) then
     ret := STACK_EMPTY;
  else
     t := top;
     ret := data(t)
     top := next(t);
     free(t);
  end if
end
```

The verification of these procedures can be done within separation logic, by proving that the following triples hold:

$$\{\ \textbf{list}\ \alpha\ (top,\ nil)\ \}\ \text{S.push(v)}\ \{\ \textbf{list}\ (v\cdot\alpha)\ (top,\ nil)\ \} \tag{4.4.9a}$$

$$\{\ \textbf{list}\ \alpha\ (top,\ nil)\ \}\ \text{S.pop()}\ \left\{\begin{array}{c}(ret = STACK\_EMPTY \land (\alpha=\epsilon))\\ \lor\\ (ret = head(\alpha) \land\ \textbf{list}\ tail(\alpha)\ (top,\ nil)\ )\end{array}\right\} \tag{4.4.9b}$$

In this section we shall not give wp proofs in detail. For now we shall present the proofs of 4.4.9a and 4.4.9b using “proof outlines”. A proof outline is a form of “condensed proof” that uses forward reasoning and shows the intermediate assertions that hold at points between the initial and final states of a program. In principle it is possible to work out the full details of a proof from the proof outline.
In the following proofs we shall use the weakest precondition calculus defined in the background section (see section 2.4).

Proof outline for $push()$

```
def push(val)
{ $\textbf{list}\ \alpha\ (top,\ nil)$ }
   n := new node();
   { $\textbf{list}\ \alpha\ (top,\ nil)\ * n \mapsto [data : 0, next : nil]$}
   data(n) := v;
   next(n) := top;
   { $\textbf{list}\ \alpha\ (top,\ nil)\ * n \mapsto [data : v, next : top]$}
   $\Longrightarrow$ {$n \mapsto [data : v, next : top] *\ \textbf{list}\ \alpha\ (top,\ nil)$ }
     $\Longrightarrow$ { $\textbf{list}\ v\ (n,\ top)\ *\ \textbf{list}\ \alpha\ (top,\ nil)$ }
     $\Longrightarrow$ { $\textbf{list}\ v \cdot \alpha\ (n,\ nil)$ }
   top := n;
   { $\textbf{list}\ v \cdot \alpha\ (top,\ nil)$ }
end
```

Proof outline for *pop*()

```
def pop()
{ $\textbf{list}\ \alpha\ (top,\ nil)$ }
   if (top == nil) then
   { $\textbf{list}\ \alpha\ (top,\ nil)\ \wedge top = nil$}
   $\Longrightarrow$ { $\textbf{list}\ \alpha\ (nil,\ nil)$ }
   $\Longrightarrow$ {$\alpha = \epsilon$}
        ret := STACK_EMPTY
        {$ret = STACK\_EMPTY \wedge \alpha = \epsilon$}
   else
   { $\textbf{list}\ \alpha\ (top,\ nil)\ \wedge top \neq nil$}
        t := top;
        { $\textbf{list}\ \alpha\ (t,\ nil)\ \wedge t \neq nil$}
        $\Longrightarrow$ {$\exists n \cdot t \mapsto [data : head(\alpha), next : n] *\ \textbf{list}\ tail(\alpha)\ (n,\ nil)$ }
        ret := data(t)
        {$\exists n \cdot t \mapsto [data : head(\alpha), next : n] *\ \textbf{list}\ tail(\alpha)\ (n,\ nil)\ \wedge ret = head(\alpha)$}
        top := next(t);
        {$t \mapsto [data : head(\alpha), next : top] *\ \textbf{list}\ tail(\alpha)\ (top,\ nil)\ \wedge ret = head(\alpha)$}
        free(t);
        {$emp *\ \textbf{list}\ tail(\alpha)\ (top,\ nil)\ \wedge ret = head(\alpha)$}
        $\Longrightarrow$ {$ret = head(\alpha) \wedge\ \textbf{list}\ tail(\alpha)\ (top,\ nil)$ }
   end if
end
```

Chapter 5

Conclusions and Future Work

In this thesis we studied in detail two formalisms that merge separation logic and rely guarantee reasoning: RGSep and Local Rely Guarantee (LRG). We saw that both formalisms effectively combine the strengths of SL and RG to provide a single formalism that supports local, modular, and explicit reasoning about interference between threads in a shared variable concurrent program. We have studied in detail their features and noted similarities and differences:

- Both apply the concept of local reasoning to the rely and guarantee conditions, which were previously only treated globally in the original formulations of rely guarantee logic.
- RGSep has syntax for assertions on shared vs. private state while LRG has none; the latter uses the concept of a ”fencing invariant” to determine the boundary between shared and private state
- Both logics model updates against shared state in essentially the same way as actions, but LRG offers more ways to combine actions together
- LRG offers more ways to do local reasoning about rely and guarantee reasoning through its extended frame rule and hiding rule

We also constructed full proofs of correctness of simple but interesting algorithms in both formalisms and noted the differences in the proofs, in particular the differences in the proof obligations, and how these stem from the way in which each logic models program state and state updates.

In this thesis we also explored the use of a weakest precondition (wp) calculus in separation logic proofs through fully worked-out examples of proofs of correctness of array-based and linked-list stack algorithms. We illustrated how wp calculus can help do proofs in a more calculational way . We also noted how these proofs are essentially data abstraction proofs that show how each stack data structure implements a stack abstract data type, and related this to a classic data abstraction technique by Hoare.

A possible direction for future work is to explore the more advanced features of LRG, specifically its extension of the frame rule and its hiding rule in proofs of algorithms that involve more fine-grained concurrency. Furthermore it would be interesting to see how the wp calculus for separation logic can applied to either RGSep or LRG and how this can be encoded within a proof assistant to achieve some degree of proof automation.

Bibliography

[1] Appel, A. W. (2011). Verismall: Verified smallfoot shape analysis. In Jouannaud, J. and Shao, Z., editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 231–246. Springer.

[2] Appel, A. W. and Blazy, S. (2007). Separation logic for small-step cminor. *CoRR*, abs/0707.4389.

[3] Berdine, J., Calcagno, C., and O'Hearn, P. W. (2005a). Smallfoot: Modular automatic assertion checking with separation logic. In de Boer, F. S., Bonsangue, M. M., Graf, S., and de Roever, W. P., editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer.

[4] Berdine, J., Calcagno, C., and O'Hearn, P. W. (2005b). Symbolic execution with separation logic. In Yi, K., editor, *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer.

[5] Berdine, J., Calcagno, C., and O'Hearn, P. W. (2012). Verification condition generation and variable conditions in smallfoot. *CoRR*, abs/1204.4804.

[6] Berdine, J., Cook, B., and Ishtiaq, S. (2011). Slayer: Memory safety for systems-level code. In *CAV*.

[7] Bornat, R., Calcagno, C., and Yang, H. (2006). Variables as resource in separation logic. *Electronic Notes in Theoretical Computer Science*, 155:247–276.

[8] Brookes, S. (2007). A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1):227–270.

[9] Calcagno, C., Parkinson, M., and Vafeiadis, V. (2007). Modular safety checking for fine-grained concurrency.

[10] Chang, B. E., Rival, X., and Necula, G. C. (2007). Shape analysis with structural invariant checkers. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, pages 384–401.

[11] Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457.

[12] Distefano, D. and Parkinson, M. J. (2008). jstar: towards practical verification for java. In Harris, G. E., editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 213–226. ACM.

[13] Feng, X. (2009). Local rely-guarantee reasoning. In *ACM SIGPLAN Notices*, volume 44, pages 315–327. ACM.

[14] Gries, D. (2012). *The science of programming.* Springer Science & Business Media.

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

[16] Hoare, C. A. R. (1972). *Proof of correctness of data representations.* Springer.

[17] Hobor, A., Appel, A. W., and Nardelli, F. Z. (2008). Oracle semantics for concurrent separation logic. In Drossopoulou, S., editor, *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4960 of *Lecture Notes in Computer Science*, pages 353–367. Springer.

[18] Ishtiaq, S. S. and O'Hearn, P. W. (2001). Bi as an assertion language for mutable data structures. In *ACM SIGPLAN Notices*, volume 36, pages 14–26. ACM.

[19] Jacobs, B. and Piessens, F. (2008). The verifast program verifier. Technical report, Department of Computer Science, Katholieke Universiteit Leuven, Belgium.

[20] Jacobs, B., Smans, J., and Piessens, F. (2010). A quick tour of the verifast program verifier. In Ueda, K., editor, *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, volume 6461 of *Lecture Notes in Computer Science*, pages 304–311. Springer.

[21] Jones, C. B. (1983a). Specification and design of (parallel) programs.

[22] Jones, C. B. (1983b). Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619.

[23] Magill, S., Tsai, M., Lee, P., and Tsay, Y. (2008). THOR: A tool for reasoning about shape and arithmetic. In Gupta, A. and Malik, S., editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 428–432. Springer.

[24] Mansky, W. (2008). Automating separation logic for concurrent c minor. Undergraduate Thesis.

[Moreira et al.] Moreira, N., Pereira, D., and de Sousa, S. M. On the mechanisation of rely-guarantee in coq.

[26] Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., and Birkedal, L. (2008a). Ynot: dependent types for imperative programs. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 229–240.

[27] Nanevski, A., Morrisett, J. G., and Birkedal, L. (2008b). Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911.

[28] Nguyen, H. H., David, C., Qin, S., and Chin, W. (2007). Automated verification of shape and size properties via separation logic. In Cook, B. and Podelski, A., editors, *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, volume 4349 of *Lecture Notes in Computer Science*, pages 251–266. Springer.

[29] Nieto, L. P. (2003). The rely-guarantee method in isabelle/hol. In *Programming Languages and Systems*, pages 348–362. Springer.

[30] O'hearn, P. W. (2007). Resources, concurrency, and local reasoning. *Theoretical computer science*, 375(1):271–307.

[31] Owicki, S. and Gries, D. (1976). Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285.

[32] Parkinson, M., Bornat, R., and O'Hearn, P. (2007). Modular verification of a non-blocking stack. In *ACM SIGPLAN Notices*, volume 42, pages 297–302. ACM.

[33] Reynolds, J. (2011). Introduction to separation logic (course).

[34] Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE.

[35] Stewart, G., Beringer, L., and Appel, A. W. (2012). Verified heap theorem prover by paramodulation. In Thiemann, P. and Findler, R. B., editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 3–14. ACM.

[36] Tuch, H., Klein, G., and Norrish, M. (2007). Types, bytes, and separation logic. In Hofmann, M. and Felleisen, M., editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 97–108. ACM.

[37] Tuerk, T. (2008). A separation logic framework in hol. In Otmane Ait Mohamed, C. M. and Tahar, S., editors, *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*, pages 116–122.

[38] Tuerk, T. (2009). A formalisation of smallfoot in HOL. In Berghofer, S., Nipkow, T., Urban, C., and Wenzel, M., editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 469–484. Springer.

[39] Vafeiadis, V. (2008). *Modular fine-grained concurrency verification.* PhD thesis, University of Cambridge.

[40] Vafeiadis, V. (2009). Shape-value abstraction for verifying linearizability. In Jones, N. D. and Müller-Olm, M., editors, *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, volume 5403 of *Lecture Notes in Computer Science*, pages 335–348. Springer.

[41] Vafeiadis, V. (2010a). Automatically proving linearizability. In Touili, T., Cook, B., and Jackson, P., editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 450–464. Springer.

[42] Vafeiadis, V. (2010b). Rgsep action inference. In Barthe, G. and Hermenegildo, M. V., editors, *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, volume 5944 of *Lecture Notes in Computer Science*, pages 345–361. Springer.

[43] Vafeiadis, V. and Parkinson, M. (2007). A marriage of rely/guarantee and separation logic. In *CONCUR 2007–Concurrency Theory*, pages 256–271. Springer.

[44] Villard, J., Lozes, É., and Calcagno, C. (2010). Tracking heaps that hop with heap-hop. In Esparza, J. and Majumdar, R., editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 275–279. Springer.

[45] Winskel, G. (1993). *The formal semantics of programming languages: an introduction.* MIT press.

Appendix A

Survey of Tools for Separation Logic and Rely Guarantee

A.1 Introduction

This is a survey of the literature on formalisations of separation logic (and related logics) within interactive proof assistants as well as other related tools. We also include an annotated bibliography. Separation logic tools can be classified under two general categories: a) those which support full functional verification within an interactive proof assistant utilising an embedding of separation logic and a verification condition generator (vgc); b) automated tools which aim to verify more limited program properties - this category includes shape analysis, model checking, and separation logic decision procedures.

A.2 Verification within proof assistants

A.2.1 Holfoot

Logic	Abstract Separation Logic
Technique	Theorem Proving (with some degree of automation)
Kinds of proof	Full functional correctness
Proof Assistant	HOL4

Tuerk, T. (2008). A separation logic framework in hol. In Otmane Ait Mohamed, C. M. and Tahar, S., editors, *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*, pages 116–122

The author proposes building a separation logic framework that is general

enough to express different flavours of SL and that can be easily instantiated into different programming languages. A formalisation of Abstract Separation Logic in HOL is presented as a first step towards the proposed framework.
Contributions:

- A formalisation of Abstract Separation Logic in HOL, extended with procedure calls.
- Formalisation of a big part of the "variables as a resource" formulationof SL
- Implementation of a fully automated tool for a language that is very similar to smallfoot

Tuerk, T. (2009). A formalisation of smallfoot in HOL. In Berghofer, S., Nipkow, T., Urban, C., and Wenzel, M., editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 469–484. Springer

An HOL encoding of abstract separation logic with procedures (as described in the previous citation) is used as the basis for a tool that can parse Smallfoot specifications and verify most of these fully automatically. In addition the tool can reason about the content of data structures and not just their shapes, thereby enabling the verification of fully functional specifications. Nontrivial examples that have been verified with the tool include parallel mergesort and an interacive filter function for singly-linked lists. The paper is also useful for its concise presentation of abstract separation logic.

Note: Holfoot supports verification of concurrent programs using conditional critical regions and parallel composition. Tuerk has demonstrated this with at least two example verifications of parallel mergesort and parallel tree disposal.

A.2.2 Ynot

Logic	Hoare and separation logic
Technique	Theorem Proving
Kinds of proof	Functional correctness
Proof Assistant	Coq

Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., and Birkedal, L. (2008a). Ynot: dependent types for imperative programs. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 229–240

Ynot is an axiomatic extension of Coq that supports writing, reasoning about, and extracting higher-order dependently-typed programs with side effects. Ynot adds to Coq support for effectful computations such as non-termination, accessing a mutable store, and throwing/catching exceptions. Ynot is based on Hoare Type Theory (see related paper). Ynot has been used to build modules that implement imperative maps with support for effectul iterators.

Nanevski, A., Morrisett, J. G., and Birkedal, L. (2008b). Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911

Proposes a novel approach to specification and verification that smoothly combines dependent types and a Hoare-style logic for a language with higher-order functions and imperative commands. The key mechanism is a type constructor for Hoare (partial) triples $\{P\}\ x : A\ \{Q\}$ which describes the effect of imperative commands. Intuitively, this type can be ascribed to a stateful computation if, when executed in a heap satisfying the precondition P, the computation either diverges or results in a heap satisfying the postcondition Q and returns a result of type A.

Hoare types can be viewed as a kind of monad, which in functional programming is a datatype that represents effectful computations. In Hoare type theory, the supported operations on state involve allocation, deallocation, lookup, and strong update, such that a location may be updated with values of varying types. Hoare types can be nested, combined with other types, and abstracted within terms, types, and predicates alike, thus improving upon the data abstraction and information hiding mechanisms of the original Hoare logic, and leading to a unified system for specifying, programming, and reasoning about programs.

A.2.3 Concurrent C minor

Logic	Concurrent Separation Logic
Technique	Theorem Proving
Kinds of proof	Functional correctness
Proof Assistant	Coq

The research work around C Minor and Concurrent C Minor is part of the project of building a “verified software toolchain” at Princeton University (`http://vst.cs.princeton.edu`). In this context, the C Minor / Concurrent C Minor languages are typically used as intermediate languages between C and some low-level language and the semantics for these have been mechanised and proved-sound separation logics have been built as well.

Appel, A. W. and Blazy, S. (2007). Separation logic for small-step cminor. *CoRR*, abs/0707.4389

C minor is a mid-level imperative programming language ; there are proved-correct optimizing compilers from C to Cminor and from Cminor to assemly langueg. In this paper the authors redesign the language to make it suitable for Hoare logic reasoning, and then construct a separation logic for it. A small-step semantics is constructed, and a machine-checked proof of soundness of the separation logic w.r.t. to the semantics is constructed in the Coq proof assistant. The small-step semantics is based on continuations mainly to allow a uniform representation of statement execution. The small-step semantics deals with nonlocal control constructs (return, exit) and is designed to extend to a concurrent setting. In their construction of SL, the authors extended classical Hoare logic to sextuples in order to take account of the nonlocal constructs. The authors have also proved semantic equivalence between their small-step semantics and the big-step semantics of Leroy's CompCert certified compiler; hence the programs that they prove in the separation logic can also be compiled by CompCert.

Hobor, A., Appel, A. W., and Nardelli, F. Z. (2008). Oracle semantics for concurrent separation logic. In Drossopoulou, S., editor, *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4960 of *Lecture Notes in Computer Science*, pages 353–367. Springer

Concurrent Cminor is an extension of Cminor with support for shared memory, spawnable threads, and first-class locks. In this paper the authors present a modular operational semantics for Concurrent Cminor, as well as a concurrent separation logic with first class locks and threads, and prove its soundness w.r.t. the semantics.

Mansky, W. (2008). Automating separation logic for concurrent c minor. Undergraduate Thesis

In this undergraduate thesis the author demonstrates the implementation of separation logic for Concurrent C Minor within Coq. The syntax and semantics of Concurrent C Minor is formalised and semi-automated tactics for C Minor are extended to build a framework for proofs of safety of concurrent programs. In addition the author describes a thread-modular shape analysis algorithm for inferring lock invariants (due to Gotsman, et.al.) and uses it to transform a simple C-like language into verified Concurrent C Minor.

A.2.4 L4.verified

Logic	Separation Logic
Technique	Theorem Proving
Kinds of proof	Safety and functional correctness
Proof Assistant	Isabelle

The L4.verified project is a project within the National ICT Australia (`http://ssrg.nicta.com.au/projects/TS/l4.verified/`) with the aim of formally verifying the correctness of the L4 operating system microkernel.

Tuch, H., Klein, G., and Norrish, M. (2007). Types, bytes, and separation logic. In Hofmann, M. and Felleisen, M., editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 97–108. ACM

Extends previous work (described in the paper ”Unified Memory Model for Pointers”) in which the authors present a formalisation of the low-level, untyped machine state in a way that is suitable for proof abstractions and its application to the verification of the virtual memory subsystem of the L4 OS microkernel. In this paper the implementation is extended to reflect the real semantics of a significant, strict subset of C; also the memory model is extended to support separation logic constructs and a shallow embedding is done in Isabelle; case studies are presented of a simple list reversal and a full formal verification of the L4 kernel memory allocator.

A.3 Automated tools

A.3.1 Smallfoot

Logic	Concurrent separation Logic
Technique	Shape Analysis
Kinds of proof	Memory safety, data structure invariants
Proof Assistant	n/a

Smallfoot is an automatic verification tool that checks separation logic specifications of sequential and concurrent programs that manipulate recursive dynamically-allocated (linked) data structures.

Berdine, J., Calcagno, C., and O'Hearn, P. W. (2005a). Smallfoot: Modular automatic assertion checking with separation logic. In de Boer, F. S., Bonsangue, M. M., Graf, S., and de Roever, W. P., editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer

This paper is a tutorial-style introduction to Smallfoot, which is a tool for checking certain lightweight specifications in separation logic. The assertions describe the shapes of data structures rather than their detailed contents, and this allows reasoning to be fully automatic. The input language allows first-order procedure with reference and value parameters, and operations for allocating, deallocating, reading, and mutating heap cells. Smallfoot requires annotations for pre- and post-conditions and loop invariants; in addition it supports annotations for concurrency, based on a concurrent extension of separation logic.

On a high level the way Smallfoot works is that it chops an annotated program into Hoare triples for certain symbolic instructions. The validity of these triples is then decided using a symbolic execution mechanism (discussed in a related paper). The symbolic execution reduces these triples into entailments $P \vdash Q$, which are the verification conditions. The vcg algorithm is discussed in another related paper.

The paper presents examples of the use of Smallfoot to verify shape properties of linked lists, trees, and double-ended queues.

Berdine, J., Calcagno, C., and O'Hearn, P. W. (2005b). Symbolic execution with separation logic. In Yi, K., editor, *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer

This paper describes the symbolic execution mechanism used by Smallfoot. It is a method for automatically proving Hoare triples for loop-free code within Separation Logic, for certain pre- and post-conditions which model "symbolic heaps". Intuitively a symbolic heap is a notion of program state that separates state into two parts: a pure (heap-independent) part and spatial (heap-dependent) part. In this heap model a location maps to a record of values. Formally a symbolic heap has the form $\Pi|\Sigma$ where Π is the pure part and Σ the spatial part. The symbolic execution mechanism applies a set of operational and rearrangement rules for turning a Hoare triple of the form $\{\Pi|\Sigma\}\ C\ \{\Pi'|\Sigma'\}$, where C is an (inductively-defined) loop-free program, into a set of entailments of the form $\Pi|\Sigma \vdash \Pi'|\Sigma'$.

The paper also discusses a proof theory to for entailments $\Pi|\Sigma \vdash \Pi'|\Sigma'$ and a method for inferring frame axioms.

Berdine, J., Calcagno, C., and O'Hearn, P. W. (2012). Verification condition generation and variable conditions in smallfoot. *CoRR*, abs/1204.4804

This paper is a technical note to accompany the two papers above and describes:

- the variable conditions that Smallfoot checks
- the analysis used to check them
- the algorithm used to compute a set of verification conditions corresponding to an annotated program, and
- the treatment of concurrent resource initialisation code

A.3.2 Space Invader

Space Invader is a prototype analyser for C programs based on separation logic. It was developed between 2005-2008 and is no longer currently being maintained. The website refers instead to the tools SLAyer and Predator.

A.3.3 SLAyer

Logic	Separation Logic
Technique	Program analysis
Kinds of proof	Memory safety
Proof Assistant	n/a

SLAyer is an automatic formal verification tool in development within Microsoft Research which is aimed at proving properties of programs that may involve reasoning inductively about data-structures. This is in response to the shortfalls of first-generation software model checking tools, like SLAM or BLAST, which are able only to build a shallow finite approximation of the data-structures created during a program's execution.

The initial goal of the SLAyer project is to develop an automatic tool that will allow us to prove non-trivial properties of data-structures constructed during the execution of industrial software components with 100,000 lines of code or less.

Berdine, J., Cook, B., and Ishtiaq, S. (2011). Slayer: Memory safety for systems-level code. In *CAV*

This paper describes SLAyer, a program analysis tool designed to automatically prove memory safety of industrial systems code. The authors discuss its implementation as well as their experience in applying the tool to Windows device drivers. This paper accompanies the first release of SLAyer.

SLAyer can prove the absence of memory safety errors such as dangling pointer dereferences, double frees and memory leaks. Towards this goal, SLAyer searches for invariants that form proofs in separation logic. The algorithms implemented in the tool are aimed at verifying moderately sized (10K - 30K LOC) systems code written in C. It is fully automatic and does not require any annotations or hints from the user.

The assertion language used by SLAyer is an extension of that used by Smallfoot. The pure, heap-independent part of the logic, which includes formulas for linear arithmetic and equality over addresses, is passed through to the Z3 SMT solver. Proof search is performed using a sequent calculus that includes deduction rules specific to the fragment's atomic formulas. A particular collection of axioms involving $\mapsto$ and ls are built into the calculus. SLAyer uses a version of the Reps-Horowitz-Sagiv algorithm to perform a whole-program interprocedural analysis. Individual instructions, including specification statements, are symbolically executed following Smallfoot.

A.3.4 SmallfootRG

Logic	RGSep
Technique	Shape analysis
Kinds of proof	Memory safety
Proof Assistant	n/a

SmallfootRG is an extension of Smallfoot with Vafeiadis and Parkinson's combination of rely/guarantee and separation logic. It is now obsolete and superseded by a tool called Cave (see next section).

Calcagno, C., Parkinson, M., and Vafeiadis, V. (2007). Modular safety checking for fine-grained concurrency

Main contribution: the authors automate a suitable subset of RGSep and implement a modular tool that automatically verifies safety properties of class of intricate concurrent algorithms.

The tool symbolically executes the code and produces verification conditions that, if proved valid, imply that the program is correct wrt the user-supplied pre/post-condition pairs. The tool splits the state (i.e. the program heap) into local and shared state and maintains this partition throughout the symbolic execution. The assertions are restricted to a fragment of separation logic that is suitable for symbolic execution as used by the smallfoot tool. The tool starts with the precondition and then symbolically executes the code to derive a postcondition, and then checks that this derived postcondition implies the user-supplied postcondition.

In order to handle fine-grained concurrency, the tool requires the user to describe therad interference in terms of action annotations. The tool is limited to checking the following safety properties: data integrity, memory leaks, and race conditions.

Other technical contributions include:

- Enriching the set of SL operators that are handled automatically
- A procedure for calculating the interference imposed by the environment
- A symbolic execution of RGSep assertions
- An automatic safety checker for list-manipulating programs
- Verification of a series of fine-grained concurrent algorithms

A.3.5 Cave: Concurrent Algorithm VErifier

Logic	RGSep
Technique	Program analysis
Kinds of proof	Memory safety and linearizability
Proof Assistant	n/a

Cave is an automated verification tool for proving memory safety and linearizability (that is, atomicity and functional correctness) of concurrent data structures. The tool consists of a program analyser implementing the RGSep action inference algorithm using the shape-value abstract domain, as well as a procedure for searching for linearisation point assignments. There is also a prototype extension of Cave for verifying lock-freedom of concurrent algorithms.

Vafeiadis, V. (2010a). Automatically proving linearizability. In Touili, T., Cook, B., and Jackson, P., editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 450–464. Springer

The main contribution of this paper is an algorithm for verifying linearizability of a concurrent shared data structure (which we shall also call a "library"). The algorithm is implemented in a tool called CAVE which is then used to verify a number of practical concurrent stack, queue, and set algorithms in the literature. A key insight used in the algorithm is that linearisation points of an operation can occur both in "pure" executions, i.e., those which complete without modifying the shared state, and in "effectful" operations, i.e. those which modify the shared state. A procedure is given for constructing a "pure linearizability checker" which identifies potential linearisation points for any pure executions of an operation. A procedure is then given for identifying all candidate linearisation points and then all the operations are then instrumented with these linearisation points. These instrumented operations are then validated by a verification procedure. This verification procedure constructs the most general client of the library, which is a client consisting of a constructor for the library followed by an unbounded parallel composition of threads, each of which non-determinstically executes one of the library's operations in a loop. The procdure then uses an automatic static analysis technique to prove that the library is memory safe and that the assertions in any assert statement in the library are always satisfied. The static analysis procedure is the RGSep action inference technique, described in a separate paper.

Vafeiadis, V. (2010b). Rgsep action inference. In Barthe, G. and Hermenegildo, M. V., editors, *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, volume 5944 of *Lecture Notes in Computer Science*, pages 345–361. Springer

This paper presents an algorithm for calculating (i.e. automatically inferring) a set of RGSep actions that overapproximate the interference that a thread causes to its concurrent environment.

Vafeiadis, V. (2009). Shape-value abstraction for verifying linearizability. In Jones, N. D. and Müller-Olm, M., editors, *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, volume 5403 of *Lecture Notes in Computer Science*, pages 335–348. Springer

Contributions: Presents a simple proof method for verifying linearizability given a set of linearisation points (the user must annotate the locations of the l.p's). The method can handle lp's occuring in a different thread other than the one being verified. The technique employs a shape analysis technique that can remember an adjustable amount of information about the values being stored by the data structure. It also employs a version of RGSep with simplified proof rule for atomic sections, thereby simplifying action specifications for operations such as CAS. The tool compares favourably with other known tools, and succeeded in proving several concurrent algorithms that are known to be linearizable.

The technique makes the following assumptions:

- Memory model is sequentially consistent (what is the implication of this assumption)? This means that parallel composition can be viewed as trace interleaving
- the program must be accurately analysable by shape analysis - this limits

the analysis to programs operating on linked lists

- The user must annotate the locations of the linearisation points.
- The user must describe the interference imposed by the module.

More details on the technique: Given a linked list data structure with and algorithms for its operations including an init() operation for setting up the initial state of the data structure, the tool proves linearizability as follows: First the user provides the following additional inputs:

1. The user annotates the locations of the linearisation points in the code
2. The user specifies a set of "actions" i.e. precondition-postcondition pairs that summarise the possible atomic effects of the algorithms. These actions are specified in separation logic. If the user does not supply these, the tool can extract the specification by symbolically executing the code in a sequential environment

The technique then does the following:

1. Infers an abstraction map i.e. a relation between the concrete states and the abstract states of the data structure
2. Inlines the specifications in the parts of the code annotated with lp's
3. Checks automatically that the following properties hold:
    (a) the abstraction map is an invariant of the system: it does this by first inferring the postcondition of the init() method and then applies a "stabilization" algorithm which is a fixed point computation that approximates an assertion that is invariant wrt to the actions specified by the user.
    (b) Every method execution trace, whether terminating or not, has at most one lp
    (c) Every terminating execution trace of a method has at least one lp
    (d) Whenever a method terminates, it returns the same value as the specification embedded at the lp (b,c, and d are done by symbolic execution)

A.3.6 Hip

Logic	Separation Logic
Technique	Shape analysis and decision procedures
Kinds of proof	Shape and size properties
Proof Assistant	n/a

Nguyen, H. H., David, C., Qin, S., and Chin, W. (2007). Automated verification of shape and size properties via separation logic. In Cook, B. and Podelski, A., editors, *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, volume 4349 of *Lecture Notes in Computer Science*, pages 251–266. Springer

Contributions:

- Provides a shape predicate specification mechanism that can capture a wide range of data structures together with size properties, such as various height-balanced trees, priority heaps, and sorted lists.
- Provides a mechanism to soundly approximate each shape predicate by a heap-independent invariant
- Designs a new procedure to check entailment of separation heap constraints; the procedure uses unfold/fold reasoning to deal with shape definitions
- Developed a prototype verification system with the above features.

A.3.7 JStar

Logic	Separation logic
Technique	Theorem proving, symbolic execution, and abstract interpretation
Kinds of proof	Functional correctness of object-oriented programs
Proof Assistant	n/a

Distefano, D. and Parkinson, M. J. (2008). jstar: towards practical verification for java. In Harris, G. E., editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 213–226. ACM

Presents an automatic verification tool based on separation logic aiming at object-oriented programs written in Java. It integrates a theorem prover and a symbolic execution and abstraction techique for SL that are tailored to OO verification. The abstract interpretation technique is able to perform fixed point computations on properties resulting in the combination of heap information as well as data contents. It also automatically infers loop invariants.

Demonstrates the use of the tool in verifying some OO design patterns: visitor, subject/observer, factory, and pooling.

Note that there is also a jStar eclipse plugin for better usability. See (`http://research.microsoft.com/apps/pubs/default.aspx?id=180042`)

A.3.8 Verifast

Logic	Separation logic
Technique	Symbolic execution / SMT
Kinds of proof	Full functional correctness of lock-free data structures
Proof Assistant	n/a

VeriFast is a verifier for single-threaded and multithreaded C and Java programs annotated with preconditions and postconditions written in separation logic. To enable rich specifications, the programmer may define inductive datatypes, primitive recursive pure functions over these datatypes, and abstract separation logic predicates. To enable verification of these rich specifications, the programmer may write lemma functions, i.e., functions that serve only as proofs that their precondition implies their postcondition. The verifier checks that lemma functions terminate and do not have side-effects. Since neither VeriFast itself nor the underlying SMT solver need to do any significant search, verification time is predictable and low.

Jacobs, B. and Piessens, F. (2008). The verifast program verifier. Technical report, Department of Computer Science, Katholieke Universiteit Leuven, Belgium

Jacobs, B., Smans, J., and Piessens, F. (2010). A quick tour of the verifast program verifier. In Ueda, K., editor, *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, volume 6461 of *Lecture Notes in Computer Science*, pages 304–311. Springer

A.3.9 VeriSmall

Logic	Separation logic
Technique	Shape analysis
Kinds of proof	Shape propertes
Proof Assistant	Coq

Appel, A. W. (2011). Verismall: Verified smallfoot shape analysis. In Jouannaud, J. and Shao, Z., editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 231–246. Springer

Implements a version of Smallfoot in Coq that uses a "paramodulation"-based heap theorem prover (see notes below). The implementation is extractable to an effiicient ML program. The program is verified correct wrt the Separation Logic for CMinor, which in turn is verified correct with respect to Leroy's operational semantics for CMinor.

How the tool relates to Smallfoot:
This is a re-implementation of Smallfoot in Coq. Specifically the author re-implements the following in Gallina (which is Coq's specification language):

- Smallfoot's decision procedures for entailment, and its soundness proof
- The algorithms for rearrangement of preconditions to isolate conjuncts of the form $e \mapsto e'$
- The symbolic execution mechanism, and its soundness proof

The author does not implement a frame inference mechanism (which Smallfoot does).

How the tool relates to Holfoot:
As the author notes: Holfoot is "proof-generating" rather than "verified". Holfoot moves from fully automatic shape proofs to semiautomatic functional correctness proofs, generating lemmas that must be manually proven (or proven with an SMT solver). Holfoot does not connect to a semantics of a real programming language but to an abstract local-action semantics. In contrast, the author focuses on an efficient and verifiable static analysis algorithm for a real programming language connected to a real compiler, but unlike Holfoot do not progress from shape analysis to functional correctness proofs.

What paramodulation is and what can it do:

Paramodulation is a resolution theorem-proving algorithm. In a paper by Navarro and Rybalchencko ("Separation Logic + Superposition Calculus = Heap Theorem Prover") the authors apply paramodulation to the heap entailment decision problem and obtain a program that is significantly faster than the original Smallfoot implementation. Paramodulation permits modular introduction of theories, including "superposition calculus", which is a theory of equalities and inequalities. Navarro and Rybalchenko extend paramodulation with superposition and with Smallfoot's spatial terms to yield a "heap theorem prover". In his work, Appel implements this heap theorem prover within Coq (more details on this in a related paper – see the reference for VeriStar in the next section).

A.3.10 VeriStar

Logic	Separation Logic
Technique	Theorem proving
Kinds of proof	Heap entailments
Proof Assistant	Coq

Stewart, G., Beringer, L., and Appel, A. W. (2012). Verified heap theorem prover by paramodulation. In Thiemann, P. and Findler, R. B., editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 3–14. ACM

VeriStar is a verified theorem prover for a decidable subset of separation logic. It is

- purely functional – implemented in Gallina (the pure functional language embedded in Coq)
- machine-checked – there is a proof in Coq that whenever the tools says that an entailment is valid, the entailment holds in a proved-sound separation logic for C minor
- end-to-end – when the analysis + prover says that a C minor program is safe, the program will be compiled to a semantically equivalent assembly language program that runs on real hardware
- efficient – the prover implements a state-of-the-art algorithm based on paramodulation for deciding heap entailments and uses highly tuned verified functional data structures
- modular – Veristar can be retrofitted to other static analyses as a compatible entailment checker and its soundness proof can be easily ported to other separation logics

A.3.11 Thor

Logic	Separation logic
Technique	Shape analysis and decision procedures
Kinds of proof	Memory safety and arithemtic properties
Proof Assistant	n/a

Magill, S., Tsai, M., Lee, P., and Tsay, Y. (2008). THOR: A tool for reasoning about shape and arithmetic. In Gupta, A. and Malik, S., editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 428–432. Springer

THOR stands for Tool for Heap-Oriented Reasoning. The tool's goal is to prove properties of programs that manipulate heap-based data structures. The tool currently supports proving memory safety of programs that manipulate doubly-linked lists and also supports arithmetic reasoning via the approach described another paper. In this context "arithmetic reasoning" refers to the ability to prove properties involving arithmetic inequalities over integer program variables, integers stored in the heap, and list lengths. The tool outputs "arithmetic abstractions", which are purely stack-based programs with the property that unreachability of the error state in the arithmetic program implies memory safety of the original program. Such programs provide an interesting source of examples for tools targeting arithmetic reasoning.

A.3.12 HeapHop

Logic	Separation logic
Technique	Verification condition generation
Kinds of proof	Memory safety, race freedom, functional correctness
Proof Assistant	n/a

Villard, J., Lozes, É., and Calcagno, C. (2010). Tracking heaps that hop with heap-hop. In Esparza, J. and Majumdar, R., editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 275–279. Springer

Heap-Hop can check concurrent programs that manipulate the heap, particularly list and tree structures, and that synchronise using Hoare monitors and copyless message passing (i.e. passing pointers instead of data structures). Heap-hop also supports checking programs that communicate asynchronously via channels. It is based on vcgen and checking and requires the user to provide pre/post-conditions and loop invariants. In order to establish global properties such as absence of memory leaks and progress properties, Heap-Hop uses "contracts", which is a form or session type or communicating finite state machine that dictates which messages are admissible on a channel. Heap-hop can prove the following properties:

- memory safety – i.e. a program does not fault on memory accesses
- race freedom
- contract obedience
- compliance with user specifications (pre/post-conditions)
- deadlock freedom
- absence of memory leaks

The tool has been used in several case studies including concurrent programs for copyless list transfer, communication protocols, and parallel tree disposal.

A.3.13 Xisa (Extensible Inductive Shape Analysis

Logic	Separation logic
Technique	Shape analysis
Kinds of proof	Shape properties
Proof Assistant	n/a

Xisa is an automatic program analysis and verification tool for reasoning about recursive data structures, such as pointer-based lists and trees. The Xisa approach is unique in that it utilizes high-level, program developer-oriented specifications to focus the analysis to properties of interest to the developer.

Chang, B. E., Rival, X., and Necula, G. C. (2007). Shape analysis with structural invariant checkers. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, pages 384–401

A.4 Tabular summary of tools

Tool	Logic	Concurrency	Technique				Properties		
			theorem proving	static analysis	symbolic execution	decision procedures	full correctness	memory safety	shape invariants
Holfoot	Abstract SL	✓	✓	x	x	x	✓	x	✓
Ynot	Hoare Logic / SL	x	✓	x	x	x	✓	x	x
Concurrent C Minor	Concurrent SL	✓	✓	x	x	x	✓	x	x
L4.verified	SL	x	✓	x	x	x	✓	x	x
Smallfoot	Concurrent SL	✓	x	✓	✓	✓	x	✓	✓
Space Invader	SL	x	x	✓	x	x	x	✓	x
SmallfootRG	RGSep	✓	x	✓	✓	✓	x	✓	✓
CAVE	RGSep	✓	x	✓	✓	✓	x	✓	✓
Hip	SL	x	x	✓	✓	✓	x	x	✓
JStar	SL	x	✓	✓	✓	x	✓	x	x
Verifast	SL	✓	x	x	✓	✓(SMT)	✓	x	x
Verismall	Concurrent SL	✓	x	✓	✓	✓	x	✓	✓
Veristar	SL	x	✓	x	x	✓	x	x	✓
Thor	SL	x	x	✓	x	✓	x	✓	x
HeapHop	SL	✓	✓	x	x	x	✓	✓	x
Xisa	SL	x	x	✓	x	x	x	✓	✓

Table A.1: Summary of separation logic tools